ББК 32.973.26-018.2.75
С50
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Р.Г. Имамутдиновой*, канд. физ.-мат. наук *А.А. Минько*

Под редакцией канд. физ.-мат. наук *А.А. Минько*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Смит, Билл.

С50 Методы и алгоритмы вычислений на строках. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2006. — 496 с. : ил. — Парал. тит. англ.

ISBN 5–8459–1081–1 (рус.)

Книга представляет собой фундаментальное введение в алгоритмы и методы, эффективно вычисляющие паттерны в строковых последовательностях. Речь идет об общих алгоритмах и методах, которые находят применение во многих областях науки и информационных технологий: сжатие данных, криптография, распознавание речи и компьютерное зрение, вычислительная геометрия и молекулярная биология. Рассмотренные в книге алгоритмы предназначены для нахождения в строковых последовательностях определенных типов паттернов — частных, характеристических и внутренних. Каждому типу паттернов посвящена соответствующая часть книги. Книга отличается последовательным изложением материала, большим количеством иллюстративных примеров, свободным обсуждением текущих исследований в этой области, содержит более 500 упражнений, поясняющих и расширяющих материал, изложенный в тексте книги.

Книга предназначена для тех, кто имеет достаточную подготовку в математике и информатике и хочет познакомиться с этой интересной и важной областью. Материал книги доступный для студентов старших курсов и аспирантов соответствующих специальностей.

ББК 32.973.26–018.2.75

ISBN 5–8459–1081–1 (рус.)
ISBN 0–201–39839–7 (англ.)

ОГЛАВЛЕНИЕ

СОДЕРЖАНИЕ

Предисловие

В начале было Слово,
и Слово было у Бога,
и Слово было Бог.

— Иоанн 1.1

Вычисление паттернов в строковых последовательностях — это фундаментальная проблема, которая возникает во многих областях науки и информационных технологий. Манипулирование текстом в текстовых редакторах, лексический анализ компьютерных программ, работа конечных автоматов, извлечение информации из баз данных — это малая часть тех процессов, которые требуют нахождения или вычисления паттернов. Алгоритмы вычисления паттернов находят применение в таких областях, как сжатие данных, криптография, распознавание речи и компьютерное зрение, вычислительная геометрия и молекулярная биология. Тема вычисления паттернов в строковых последовательностях важна не только из-за своего практического применения. Она является частью комбинаторики, где, как известно, существует много просто формулируемых задач, для которых, однако, очень сложно найти решение, и интерпретация таких задач, как вычисление паттернов, часто позволяет найти элегантное и точное их решение.

В этой связи вызывает большое удивление, что академические факультеты математики или компьютерных наук в своем большинстве не включают в магистерские курсы или в курсы для аспирантов эту интересную, важную и сложную для исследований тему. Еще более удивительно, что существует всего несколько книг и обзоров, где собраны вместе и основополагающие теоретические результаты, и практические алгоритмы, которые появились в последнюю четверть прошедше-

го столетия. Я знаю всего пять книг [214, 67, 108, 206, 61] и три объемных обзора научных статей [21, 2, 191], содержимое которых значительно перекрывает материал данной книги. Обзорные статьи и книги [214, 67] написаны в большей мере как обобщение итогов исследования авторов, нежели как книги для студентов. Книги [108, 206] касаются, в основном, применения строковых алгоритмов в молекулярной биологии. Последняя монография [61], написанная легко и элегантно, сочетает в себе как монографию, так и руководство по строковым алгоритмам. К сожалению, в настоящее время она доступна только на французском языке.

Данная книга ставит целью заполнить этот пробел: дать общее введение в алгоритмы вычисления паттернов, которое было бы полезно в качестве отправного пункта для научных исследований и давало бы серьезный фактический материал, доступный для изучения студентам старших курсов и аспирантам. Задержимся на некоторое время на трех словах из предыдущего предложения: "доступный", "алгоритм" и "паттерн".

Основное "свойство" этой книги — сделать материал *доступным* для студентов старших курсов и аспирантов, имеющих специализацию по математике или компьютерным наукам и которые знакомы с дискретными структурами и алгоритмами, оперирующими этими структурами.

Первым условием доступности материала книги является достаточная математическая подготовка читателя, а не его знакомство со строковыми последовательностями. Предполагается, что студент прослушал стандартный курс дискретной математики, курсы по структурам данных и анализу алгоритмов. В этом случае он знает, что такое стеки, очереди, связанные списки и массивы, имеет понятие об анализе алгоритмов и о том, как записать "асимптотическую сложность" алгоритма, имеет некоторый опыт работы с математическими утверждениями и методами, используемыми для доказательства корректности алгоритмов, также знаком с алгоритмами, выполняемыми на графах и деревьях. В дополнение к перечисленному, предполагается, что читатель знаком с какими-либо языками программирования и способен читать и понимать алгоритмы, записанные на этих языках.

Второе условие не связано с компетентностью и подготовленностью читателя: моя цель — "соблазнить" студента и читателя интереснейшей и увлекательной областью новых знаний. Я не собирался писать энциклопедию алгоритмов, вычисляющих паттерны в строковых последовательностях. Вместе с тем я хотел представить результаты, которые (я надеюсь) важны и которые можно обобщить и расширить. Но это неизбежно ведет к тому, что некоторые интересные результаты были опущены (нельзя охватить необъятное!). И все-таки я надеюсь, что выбранный подход к изложению материала будет стимулировать читателя и далее к самостоятельному изучению им научной литературы.

Особым "субъектом" материала этой книги является математический объект, который в компьютерных науках называется "строка" (string) или "строковая последовательность" (в Европе, в среде математиков, более распространен термин

"слово"). Но основное внимание в книге уделяется *алгоритмам*, т.е. точным методам и процедурам, предназначенным для выполнения "чего-то". Исходя из этого данную книгу скорее можно отнести к книгам по компьютерным наукам, чем к математическим книгам. Поэтому она значительно отличается от классической монографии [164], посвященной этой теме, и ее "потомков" [162, 163], ставящих во главу угла математические аспекты данной темы. Нас в первую очередь будут интересовать алгоритмы, находящие в строковых последовательностях различного рода паттерны, и только во вторую очередь — математические свойства самих строк. Это, конечно, не означает, что математические результаты не будут представлены строго и последовательно. Это означает, что будут представлены только те математические результаты, которые необходимы для пояснения построения и поведения алгоритмов. И последнее замечание: я сознательно ограничился изложением последовательных алгоритмов для обработки одномерных строк, не делая ссылок на обширную литературу по алгоритмам с распараллеливанием процесса вычисления или на быстро растущую литературу по многомерным (особенно, двухмерным) строкам.

Другой основной "герой" нашей книги — это *паттерн*. Все рассмотренные в книге алгоритмы предназначены для нахождения в строковых последовательностях определенных типов паттернов. Я говорю "определенных типов", поскольку будем различать три основных типа паттернов — частные, характеристические и внутренние. Каждому типу паттернов посвящена соответствующая часть книги.

Частный паттерн (specific pattern) — это единственный вид паттернов, который можно задать в виде списка символов в нужном порядке. Например, в строке $\boldsymbol{x} = abaababaabaab$ мы можем найти (трижды) паттерн $\boldsymbol{u} = abaab$, но не найдем паттерн $\boldsymbol{u} = ababab$. (Иногда паттерн может содержать специальные "символы замещения", и в этом случае возможно только "приближенное" (в некотором точно определенном смысле) сравнение паттерна и строки.)

Характеристические паттерны (generic patterns) основаны на специальных представлениях структурной информации о строковых последовательностях. Например, мы можем говорить о "повторениях" в строке $\boldsymbol{x}$ — в этом случае в строке $\boldsymbol{x}$ есть несколько смежных одинаковых подстрок. (Например, в приведенной выше строке $\boldsymbol{x}$ присутствуют повторяющиеся подстроки $(aba)(aba)$, $(abaab)(abaab)$, aa (три отдельные серии) и несколько других, если вы сможете их найти.)

Последний тип паттернов, которые будут рассмотрены в книге, я назвал *внутренними* (intrinsic). Эти паттерны отображают внутреннюю структуру строковых последовательностей. Мы рассмотрим различные паттерны, которые показывают наличие периодических структур в строках, например нормальную форму, дерево суффиксов, линдонскую декомпозицию, s-факторизацию. Эти паттерны вездесущие: они используются почти во всех алгоритмах вычисления частных и характеристических паттернов. Другими словами, они формируют основу для эффективных процедур обработки строковых последовательностей. Раз-

нообразие внутренних паттернов поразительно: для строки x нашего примера нормальная форма имеет вид $(abaababa)(abaab)$, тогда как линдонскую декомпозицию можно записать как $(ab)(aabab)(aab)(aab)$, а s-факторизацию — как $(a)(b)(a)(aba)(baaba)(ab)$, и все эти паттерны полезны и эффективны с вычислительной точки зрения.

Эта книга имеет следующую организацию. В части I приведены основные сведения о строковых последовательностях и алгоритмах обработки строк. Здесь даны терминология, формы записи и основные свойства строковых последовательностей. Глава 2 является ключом к остальной части книги: здесь четко поставлены задачи, алгоритмы для решения которых описаны в последующих частях книги. На основе материала этой главы читатель может выбрать для себя направление дальнейшего чтения в виде тех глав, которые представляют для него наибольший интерес. В этой части также обсуждаются качества "хороших" алгоритмов и вопросы их реализации на практике. В частях II–IV описаны алгоритмы для вычисления внутренних, частных и характеристических паттернов соответственно.

Всего в книге 13 глав, разбитых на четыре части. Каждая глава, как видно из оглавления, разделена на несколько разделов, каждый из которых заканчивается набором упражнений. Там, где это необходимо, главы заканчиваются разделами, обобщающими изложенный материал и рассматривающими смежные вопросы, дополнительные темы и нерешенные проблемы.

Отметим, что в книге приведено примерно 500 упражнений, которые составляют неотъемлемую часть книги и могут использоваться для следующих целей.

- Для проверки степени усвоения читателем прочитанного материала.
- Сделать более ясными или показать в другом контексте понятия или результаты, приведенные в тексте.
- Показать расширения или модификации алгоритмов и математических результатов, которые приведены в тексте без подробного обсуждения.
- Показать детали алгоритмов и доказательств, которые не включены в основной текст из-за их громоздкости или их "ухода" от основой темы.

С помощью упражнений я также пытался вовлечь читателя в процесс разработки и анализа представленных алгоритмов. Этим я хотел показать, что в основе большинства алгоритмов и их модификаций лежат простые идеи, проникнуть в суть которых не составляет труда, но которые, может быть, "затемнены" или отброшены предыдущими исследователями. Если идея понята и принята, остается только ее техническая реализация. Это общее замечание относится и к строковым алгоритмам.

Признаюсь, что непосредственно в процессе написания книги я мог допускать ошибки. Поэтому при ее вычитке я старался исправить все замеченные ошибки и оплошности и сгладить шероховатости изложения материала. Но, конечно, я не

могу гарантировать, что внимательный читатель не найдет "дефектов" в моей книге. Я поддерживаю Web-узел

`http://www.cs.curtin.edu.au/~smyth/patterns.shtml`,

где вы можете оставить свои сообщения о замеченных ошибках и предложения по улучшению книги. Я также буду благодарен, если читатели по этой же причине свяжутся со мной по электронной почте

`smyth@computing.edu.au` или `smyth@mcmaster.ca`

Материал книги можно использовать, как минимум, для чтения двухсеместрового курса (каждый семестр по 12–14 недель) для студентов старших курсов и аспирантов. Материал первых глав в разное время уже читался аспирантам факультета компьютерных наук и систем и факультета вычислительной техники и программного обеспечения университета Мак-Мастера, г. Гамильтон, Онтарио, Канада, и аспирантам факультета компьютерных наук университета Дебрецена, Венгрия. Аспиранты, прослушавшие эти курсы, внесли свой вклад в создание этой книги.

Я хотел бы выразить глубочайшую благодарность школе вычислительной техники университета Кортина, Перт, Западная Австралия, и ее бывшим и настоящему руководителям Деннису Муру, Терри Силли, Свете Венкатеш и Джеффу Уесту (Dennis Moore, Terry Cealli, Svetha Venkatesh, Geoff West) за щедрую поддержку и за содействие, как интеллектуальное, так и практическое, на протяжении нескольких лет. Бóльшая часть книги написана во время моих продолжительных визитов в Кортин. Я также благодарен профессору Петью Аттиле (Pethő Attila), декану факультета компьютерных наук университета Дебрецена, за его интерес к моей работе и поддержку, особенно на последнем этапе создания электронного варианта книги. Хотел бы также выразить свою глубокую благодарность моим друзьям и коллегам Лейле Багдади, Джерри Чепплу, Франью Франеку, Костасу Илиопулосу, Терри Лекроку, Ян Ли, Деннису Муру, Пату Риану, Джеми Симпсону и Ксиангдонгу Ксиао (Leila Baghdadi, Jerry Chapple, Franya Franěk, Costas Iliopoulos, Thierry Lecroq, Yin Li, Dennis Moore, Pat Ryan, Jaime Simpson, Xiangdong Xiao) за их дружеское и полезное содействие. Большая благодарность двум анонимным рецензентам, которые сделали конструктивные замечания и предложения по книге. Наконец, возношу хвалу своей дочери Жаклин за ее восхитительный выбор и попытки попробовать на вкус слова "строка" и "слово".

W.F.S.

Моим родителям

ЧАСТЬ I

Строковые последовательности и алгоритмы

Слово спасет мир (если не будет поздно)

— Аноним

ГЛАВА 1

Свойства строковых последовательностей

Истинные слова не могут быть красивыми;
Красивые слова не могут быть истинными.

— Лао-цзы (604–531 гг. до н.э.). Путь Лао-цзы

1.1 Последовательность жемчужин

Предположим, что у нас есть кучка жемчужин. Для наглядности расположим жемчужины на столе слева направо в одну линию рядом друг с другом. Пусть всего есть n жемчужин, на каждую жемчужину приклеим маленькую бирку с меткой. Предположим, что метками являются целые числа от 1 до n и метки присваиваются жемчужинам по следующим правилам.

- Метка самой левой жемчужины равна 1.
- Для жемчужины с меткой i ($i = 1, 2, \ldots, n-1$) соседняя справа жемчужина имеет метку $i + 1$.

Эти правила удовлетворяют нашему интуитивному представлению о том, как сделать из кучки жемчужин "строковую последовательность": жемчужины расположены в одномерную цепочку (ряд), по которой можно перемещаться из одного конца в другой через соседние жемчужины. Но опираясь на приведенные выше правила, можно сформулировать более общее понятие последовательности. Во-первых, можно отказаться от "жемчужин", если ввести понятие ***элемент*** последовательности. Во-вторых, метки не обязаны принимать значения только целых

положительных чисел. Метки могут быть, например, цветными или буквами какого-нибудь алфавита. Что действительно важно, так это следующие правила, определяющие строковую последовательность.

0. Каждый элемент последовательности имеет уникальную метку.

1. Каждый элемент с некоторой меткой x (за исключением не более одного элемента, который называется ***самый левый элемент***) имеет единственный ***предшествующий элемент*** с меткой $p(x)$.

2. Каждый элемент с некоторой меткой x (за исключением не более одного элемента, который называется ***самый правый элемент***) имеет единственный ***последующий элемент*** с меткой $s(x)$.

3. Для любого элемента с меткой x, который не является самым левым, выполняется равенство $x = s(p(x))$.

4. Для любого элемента с меткой x, который не является самым правым, выполняется равенство $x = p(s(x))$.

5. Для двух различных элементов с метками x и y существует такое целое положительное число k, что $x = s^k(y)$ или $x = p^k(y)$.

Эти правила охватывают сущность понятия ***сочленения*** и операции ***конкатенации***, выполняющей это сочленение: каждый элемент последовательности, за исключением самого левого и самого крайнего элементов, имеет единственный предшествующий и единственный последующий элементы. Самый левый и самый крайний элементы имеют или единственного последующего, или единственного предшествующего элемента. Более того, начиная с любого элемента с меткой x мы, перебирая конечную последовательность предшествующих и последующих элементов, можем достичь любого другого элемента с меткой y. Отсюда вытекает следующее утверждение.

Определение 1.1.1. Строковая последовательность — это набор элементов, которые удовлетворяют правилам 0–5. ■

Критическим (но не очевидным) фактором этого определения являются условия в правилах 1 и 2, что может быть *не более одного* самого левого и *не более одного* самого правого элементов. Для примера рассмотрим, что случится, если из цепочки жемчужин сформировать петлю, т.е. расположить их по кругу. Теперь в этой последовательности жемчужин нет "самого левого" и "самого правого" элементов. Однако отметим, что это не порождает проблем, поскольку все правила 0–5 выполняются.

Теперь предположим, что количество жемчужин бесконечно — есть левый конец цепочки жемчужин, а правый конец "скрыт за горизонтом". Для этой бесконечной последовательности также выполняется определение 1.1.1, при этом существует самый левый элемент, но не существует самого правого. Нетрудно

проверить, что правила 0–5 будут выполняться, если цепочку жемчужин продолжить до бесконечности и в левом направлении; в этом случае нет ни самого левого, ни самого правого элементов.

В этой книге в свое время будут рассмотрены все возможные виды строковых последовательностей. Примем следующее соглашение по терминологии. Строковую последовательность (string) для краткости будем называть ***строкой***.[1] Строку с конечным числом элементов, содержащую как самый левый, так и самый правый элементы, назовем ***линейной строковой последовательностью*** или ***линейной строкой***. Строковую последовательность с конечным (не нулевым) числом элементов, не имеющую ни самого левого, ни самого правого элементов, назовем ***строковой петлей*** (в литературе такой вид последовательности также имеет название ***циклическая*** (или ***круговая***) ***строковая последовательность***). Строковую последовательность с бесконечным количеством элементов, содержащую самый левый элемент, назовем ***бесконечной строковой последовательностью*** или ***бесконечной строкой***; бесконечную строковую последовательность, не имеющую ни самого левого, ни самого правого элементов, назовем ***бесконечной петлей***. Далее термин "строковая последовательность" (или просто "строка") будет означать любой объект, который удовлетворяет определению 1.1.1. Тип строки обычно легко определить из контекста. Кроме того, на практике нетрудно отличить линейную строку от петли, поскольку, как правило, петля определяется через соответствующую линейную строку $\boldsymbol{x}$ и записывается как $C(\boldsymbol{x})$, т.е. петля $C(\boldsymbol{x})$ формируется из линейной строки $\boldsymbol{x}$ путем назначения самого левого элемента этой последовательности последующим элементом для самого правого элемента.

Упражнения 1.1

1. Объясните, почему для сочленения строковой последовательности необходимо правило 3? Приведите пример математического объекта, для которого выполняются правила 0–2, но не выполняется правило 3.
2. Можно ли правило 4 вывести из правил 0–3? Объясните свой ответ.
3. Объясните, почему для сочленения строковой последовательности необходимо правило 5? Приведите пример математического объекта, для которого выполняются правила 0–4, но не выполняется правило 5.
4. Является ли бесконечное множество $\{a,\ a^2,\ \ldots\}$ бесконечной строковой последовательностью?
5. Может ли в соответствии с определением 1.1.1 строковая последовательность состоять из одного элемента a? Может ли такая последовательность быть петлей?

[1] В русской математической литературе строковые последовательности часто называют "словами". — *Примеч. ред.*

6. Может ли в соответствии с определением 1.1.1 строковая последовательность не иметь элементов (такая последовательность или строка называется ***пустой***)? Может ли пустая строка быть линейной строковой последовательностью? А петлей?

7. С учетом предыдущих упражнений определите, сколько различных типов строк нужно дополнительно включить в классификацию строковых последовательностей, данную в этом разделе?

8. Классификация строковых последовательностей, приведенная в этом разделе, не включает следующие типы строк.

а) строковая последовательность с бесконечным числом элементов, имеющая самый правый элемент, но не имеющая самого левого;

б) строковая последовательность с конечным числом элементов, имеющая или самый правый элемент, или самый левый, но не оба сразу.

Объясните эти исключения.

9. Существует ли какой-нибудь способ *доказать*, что определение 1.1.1 определяет строковую последовательность?

1.2 Линейные строковые последовательности

Определение строковых последовательностей, данное в предыдущем разделе, очень общее и охватывает многие типы последовательностей, встречающиеся на практике. Так, строками будут

- слова в английском языке, здесь элементами последовательностей будут прописные и строчные буквы английского алфавита вместе с апострофом и дефисом;
- текстовые файлы, здесь элементами будут символы в кодировке ASCII;
- книга, написанная на китайском языке, где элементами последовательности являются китайские иероглифы;
- компьютерная программа, где элементами будут определенные (пробел, двоеточие, точка с запятой и т.п.) вместе со "словами", со "словами", заключенными между этими разделителями;
- последовательность молекул ДНК (иногда длиной до трех миллиардов элементов-молекул), состоящая только из букв C, G, A и T, которые соответствуют нуклеотидам цитозину, гуанину, аденину и тимину;
- поток частиц, бомбардирующих космический аппарат;
- список длин сторон выпуклого многоугольника.

Это примеры того, что в предыдущем разделе мы назвали "линейной строковой последовательностью". В книге, в основном, рассматриваются именно такие строки, поэтому в данном разделе даны некоторые определения и соответствующая терминология, необходимые для работы с линейными строковыми последовательностями. Большинство (но не все) из представленного ниже применимо также к петлям, бесконечным и пустым строкам.

В приведенных выше примерах видно, что важным свойством любой строки является природа ее элементов, будь-то действительные числа, слова какого-либо языка, иероглифы и т.п. На практике удобно представлять элементы строк как члены некоторого множества. Такое множество называется ***алфавитом***, поэтому члены этого множества естественно назвать ***буквами*** (далее мы увидим, что термин "буква" можно интерпретировать значительно шире, чем просто буква какого-нибудь естественного языка).[2] Будем говорить, что строковая последовательность ***определена на этом алфавите***. Конечно, если строка $\boldsymbol{x}$ определена на алфавите A, то она определена и на любом множестве, подмножеством которого является алфавит A. Другими словами, алфавит для данной строки $\boldsymbol{x}$ определяется неоднозначно. Можно определить минимальный алфавит для строки $\boldsymbol{x}$ как множество различных элементов, каждый из которых обязательно входит в $\boldsymbol{x}$. Иногда принимается соглашение, что алфавитом может быть только минимальный алфавит (например, "двоичный" алфавит), иногда это условие снимается (пример — алфавит "действительных чисел").

В книге алфавит будет обозначаться как A, а $\alpha = |A|$ будет обозначать его мощность. В случае α, равного 2, 3 и 4, алфавит будет называться ***бинарным***, ***тернарным*** и ***кватернарным*** соответственно. Далее мы увидим, сколько интересных строковых последовательностей можно построить на бинарном алфавите. Последовательности на основе кватернарного алфавита находят применение при анализе последовательностей ДНК.

В общем случае, кроме требования "различности" элементов множества A, на само это множество особых условий не накладывается. Алфавит может быть конечным (даже нулевым!) множеством, счетным множеством (например, множество целых чисел) и даже несчетным (как континуальное множество вещественных чисел). Кроме того, элементы алфавита могут быть упорядоченными (тогда для любой пары различных элементов определен результат их сравнения "больше" или "меньше"), неупорядоченными и частично упорядоченными. Для многих алгоритмов, описанных в книге, используется неупорядоченный алфавит. В главе 4 показано, что для многих алгоритмов свойство упорядоченности алфавита значительно влияет на их вычислительную эффективность и на выбор тестовых данных для проверки алгоритмов.

[2]Из этих определений вытекает обоснованность терминов "строковая последовательность" и "строка". — *Примеч. ред.*

Для данного алфавита A обозначим через A^+ множество всех возможных непустых конечных сочетаний букв алфавита A. Например, если $A = \{a\}$, тогда $A^+ = \{a,\ a^2,\ a^3,\ \ldots\}$, где запись a^2 обозначает строку aa, запись a^3 — строку aaa и т.д. Если $A = \{0,\ 1\}$, тогда A^+ состоит из всех различных непустых конечных двоичных (битовых) последовательностей или, что то же самое, A^+ представляет множество всех неотрицательных целых чисел. Как предлагалось в упражнении 1.1.6, имеет смысл ввести ***пустую строковую последовательность*** (или ***пустую строку***), которая обычно обозначается символом ε. Если определена пустая строка ε, тогда можно определить еще два множества: $A' = A \cup \{\varepsilon\}$ и $A^* = A^+ \cup \{\varepsilon\}$. В этих обозначениях можно дать другое определение линейной строковой последовательности, эквивалентное данному в предыдущем разделе.

Определение 1.2.1. Линейными строковыми последовательностями (линейными строками) на алфавите A называются элементы множества A^+. Элементы множества A^* называются **конечными линейными строковыми последовательностями (конечными линейными строками)** на алфавите A. ■

Таким образом, множество A^+ является множеством всех линейных строк на алфавите A. Отметим, что пустая строка *не является* линейной, поскольку не имеет ни самого левого, ни самого правого элементов.

На протяжении всей книги строковые последовательности будут обозначаться строчными жирными буквами: $\boldsymbol{p}$, $\boldsymbol{t}$, $\boldsymbol{x}$ и т.п. Мы будем трактовать строковую последовательность, удовлетворяющую правилам 0–5 из раздела 1.1, как одномерный массив. Конечно, можно использовать и другие представления строковых последовательностей, например в виде связанных списков, но одномерный массив является наиболее естественной моделью для таких строк. Исходя из этой модели произвольную строку $\boldsymbol{x}$ на алфавите A в виде массива можно задать следующим образом:[3]

$$\boldsymbol{x} : \textbf{array}[1..n]\ \textbf{of}\ A.$$

В этом случае будем говорить, что строка $\boldsymbol{x}$ имеет ***длину*** $n = |\boldsymbol{x}|$ и ***позиции*** $1, 2, \ldots, n$. Для любого целого $i \in 1..n$ буква в позиции i обозначается как $\boldsymbol{x}[i]$, поэтому можно записать

$$\boldsymbol{x} = \boldsymbol{x}[1]\boldsymbol{x}[2]\cdots\boldsymbol{x}[n],$$

что, в соответствии с определением 1.1.1, является сочленением n строк длины 1. Фактически, здесь позиция i играет роль метки из правил 0–5.

Отметим, что модель массива применима также к пустым последовательностям $\boldsymbol{x} = \varepsilon$, которым соответствует пустой массив и которые имеют нулевую длину.

[3] Автор по всей книге, не оговаривая это специально, широко использует нотацию, применяемую для записи алгоритмов. Считаем, что у читателя такая нотация не вызовет недоразумений или трудностей. — *Примеч. ред.*

Обсуждение 1.2.1. Выше было сказано, что массивы являются "простыми и естественными" представлениями строковых последовательностей. Это утверждение в значительной степени отражает "вычислительную" точку зрения на данную проблему. Действительно, массив — это простая структура данных, но будет ли массив "естественен", зависит от наших представлений о механизмах получения доступа к элементам последовательностей. Поскольку строковые последовательности определяются через понятие сочленения и операцию конкатенации, то, очевидно, "естественно" осуществить поиск нужного элемента последовательности (получить доступ к нему) через предшествующие или последующие элементы. Такой механизм получения доступа лучше реализуется на представлении строк с помощью связанных списков,[4] где для получения элемента в позиции i из "текущей" позиции j требуется время, пропорциональное $|j - i|$. С другой стороны, для получения доступа к элементу массива путем простого указания его позиции требуется фиксированное (постоянное) время, не зависящее от положения искомого элемента и "текущего" элемента. С такой точки зрения представление строковых последовательностей в виде массивов кажется более предпочтительным, чем представление в виде связанных списков, поскольку получаем оценки времени выполнения алгоритмов, построенных на использовании массивов, меньшие, чем аналогичные оценки для алгоритмов, использующих списки.

На практике алгоритмы, выполняемые над строками, почти всегда начинаются с самого левого элемента (с позиции 1) или самого правого элемента (с позиции n) и далее проверяются соседние (предшествующие или последующие) элементы (или позиции) один за другим. С другой стороны, выходом таких алгоритмов часто являются позиции определенных элементов в строке, и здесь неявно предполагается, что получить доступ к этим позициям можно за фиксированное время. Один из способов "примерить" эти противоречивые представления о доступе к элементам строки заключается в том, что первоначально строка предоставляет доступ к своим элементам как связанный список — элементы доступны только один за другим либо слева направо или справа налево — и в таком порядке элементы копируются в массив. Для такого копирования требуется времени порядка $O(n)$ и дополнительной памяти такого же порядка $O(n)$. Такое преобразование не повлияет на асимптотические показатели любых алгоритмов, рассмотренных в книге. Поэтому примем в некоторой степени "лишнее" соглашение, что на входе алгоритмов строковые последовательности обрабатываются как связанные списки, а на выхо-

[4] Связанным списком называется структура для хранения данных, например элементов a_1, a_2, $\ldots$, a_n, где каждая "ячейка", в которой хранится элемент a_i, имеет также указатель на "ячейку", в которой хранится элемент a_{i+1}. Как правило, связанный список имеет заголовок, где хранится указатель на "ячейку" с элементом a_1. Таким образом, чтобы получить доступ к некоторому элементу a_i, надо последовательно пройти по указателям от "ячейки" с элементом a_1 до "ячейки", содержащей элемент a_i. — *Примеч. ред.*

де алгоритмов, если необходимо, они представимы в виде массивов. Мы обещаем предупреждать читателя, когда будем отступать от этого соглашения. ■

Равенство строковых последовательностей определяется обычным способом. Строка $\boldsymbol{x}$ длины n и строка $\boldsymbol{y}$ длины m ***равны*** (записывается как $\boldsymbol{x} = \boldsymbol{y}$) тогда и только тогда, когда $n = m$ и выполняется равенство $\boldsymbol{x}[i] = \boldsymbol{y}[i]$ для всех $i = 1, \ldots, n$. Из этого определения следует, что пустая строка ε равна только самой себе. Отметим на будущее, что присоединение пустой строки слева или справа к любой строке $\boldsymbol{x}$ не изменяет эту строку, т.е. $\boldsymbol{x} = \varepsilon\boldsymbol{x}\varepsilon$.

Для любой пары целых чисел i и j, таких, что $1 \leqslant i \leqslant j \leqslant n$, определим ***подстроку*** $\boldsymbol{x}[i..j]$ строки $\boldsymbol{x}$ следующим образом:

$$\boldsymbol{x}[i..j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1]\ldots\boldsymbol{x}[j].$$

Будем говорить, что подстрока $\boldsymbol{x}[i..j]$ ***начинается*** с позиции i (или, что тоже самое, что ее ***начальная позиция*** i) и что ее ***длина*** равна $j - i + 1$. Если $j - i + 1 < n$, то подстрока $\boldsymbol{x}[i..j]$ называется ***собственной подстрокой*** строки $\boldsymbol{x}$. Отметим два крайних случая подстрок: подстрока $\boldsymbol{x} = \boldsymbol{x}[1..n]$ длины n и подстрока $\boldsymbol{x}[i] = \boldsymbol{x}[i..i]$ длины 1. Для пары целых чисел i и j, таких, что $i > j$, примем соглашение, что $\boldsymbol{x}[i..j] = \varepsilon$, т.е. является пустой подстрокой нулевой длины. Поскольку $\boldsymbol{x} = \varepsilon\boldsymbol{x}\varepsilon$, то можно утверждать, что ε является подстрокой любой строки из множества A^*.

Пусть k — целое число, $k \in 1..n$, и рассмотрим позиции $i_k = 1, 2, \ldots, n-k+1$ в строке $\boldsymbol{x}$. Каждая из этих $n - k + 1$ позиций является начальной позицией подстроки $\boldsymbol{x}[i_k..i_k + k - 1]$ длины k. Таким образом, любая строка длины n имеет $n - k + 1$ подстрок (не обязательно различных) длины k.

Если $\boldsymbol{u}$ является подстрокой (соответственно, собственной подстрокой) строки $\boldsymbol{x}$, тогда $\boldsymbol{x}$ называется ***надстрокой*** (соответственно, ***собственной надстрокой***) строки $\boldsymbol{u}$. Конечно, подстроки и надстроки являются строковыми последовательностями.

Если множество A упорядочено, то с помощью подстрок можно индуцировать упорядоченность элементов множества A^*. Такой порядок называется ***лексикографическим***. Дадим более точное определение: пусть имеется две строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ и $\boldsymbol{y} = \boldsymbol{y}[1..m]$, где $n \geqslant 0$ и $m \geqslant 0$. Будем говорить, что $\boldsymbol{x}$ лексикографически меньше $\boldsymbol{y}$ (записывать это будем как $\boldsymbol{x} < \boldsymbol{y}$), если выполняется одно из следующих (взаимно исключающих) условий:

- $n < m$ и $\boldsymbol{x}[1..n] = \boldsymbol{y}[1..n]$ (говоря кратко, здесь строка $\boldsymbol{x}$ является "собственным префиксом" строки $\boldsymbol{y}$);
- существует целое число i ($i \in 1..\min\{n, m\}$), такое, что $\boldsymbol{x}[1..i-1] = \boldsymbol{y}[1..i-1]$ и $\boldsymbol{x}[i] < \boldsymbol{y}[i]$ (здесь i — первая позиция, в которой элементы строк $\boldsymbol{x}$ и $\boldsymbol{y}$ различны).

Например, на основе порядка английского алфавита можно сделать такие сравнения строк.

- $ab < abc$ (поскольку $i = 2 = n < 3 = m$).
- $\varepsilon < a$ (поскольку $i = 0 = n < 1 = m$).
- $ab < aca$ (поскольку $i = 2$ и $b < c$).

Отметим, что данное выше определение работает и в случае, когда одна или обе сравниваемые строки бесконечны. Основываясь на этом определении, нетрудно ввести другие отношения порядка: $\boldsymbol{x} \leqslant \boldsymbol{y}$ тогда и только тогда, когда $\boldsymbol{x} = \boldsymbol{y}$ или $\boldsymbol{x} < \boldsymbol{y}$; неравенство $\boldsymbol{x} > \boldsymbol{y}$ выполняется только тогда, когда справедливо $\boldsymbol{y} < \boldsymbol{x}$; наконец, $\boldsymbol{x} \geqslant \boldsymbol{y}$, если $\boldsymbol{y} \leqslant \boldsymbol{x}$.

Запись $\boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2 \cdots \boldsymbol{u}_k$, где $\boldsymbol{u}_i$ — непустые подстроки, называется ***декомпозицией*** или ***факторизацией*** строки $\boldsymbol{x}$ по ***факторам*** $\boldsymbol{u}_i$ (применение этого определения см. в разделе 1.4 и главе 6).

Есть два типа подстрок $\boldsymbol{x}[i..j]$, которые очень важны и поэтому имеют специальные названия. Для произвольного целого $j \in 0..n$ подстрока $\boldsymbol{x}[1..j]$ называется ***префиксом*** строки $\boldsymbol{x}$, иногда такая подстрока обозначается как $\mathrm{pref}(\boldsymbol{x})$; если $j < n$, то подстрока $\boldsymbol{x}[1..j]$ называется ***собственным префиксом*** строки $\boldsymbol{x}$. Аналогично для произвольного целого $i \in 1..n+1$ подстрока $\boldsymbol{x}[i..n]$ называется ***суффиксом*** строки $\boldsymbol{x}$ (записывается как $\mathrm{suff}(\boldsymbol{x})$) и собственным суффиксом, если $i > 1$. Отметим, что из данного определения и равенства $\boldsymbol{x} = \varepsilon\boldsymbol{x}\varepsilon$ следует, что пустая строка ε является собственным префиксом и собственным суффиксом любой строковой последовательности $\boldsymbol{x}$. Для примера, строка

$$\boldsymbol{f} = abaababaabaab$$

имеет префиксы ε, a, ab, aba, ..., $\boldsymbol{f} = abaababaabaab$ и суффиксы ε, b, ab, aab, ..., $\boldsymbol{f}$. Собственные префиксы и собственные суффиксы получаем из приведенных путем исключения из них самой строки $\boldsymbol{f}$.

Важным понятием, которое будет полезным во многих задачах, является понятие *грань* (border).

Определение 1.2.2. Гранью $\boldsymbol{b}$ строки $\boldsymbol{x}$ называется любой собственный префикс этой строки, равный суффиксу $\boldsymbol{x}$. ■

Из этого определения видно, что любая строка $\boldsymbol{x}$ всегда имеет пустую грань $\boldsymbol{b} = \varepsilon$ длины $\beta = 0$, но сама строка $\boldsymbol{x}$ не является собственной гранью. Будем использовать символ β для обозначения длины $|\boldsymbol{b}|$ грани $\boldsymbol{b}$. Часто представляет интерес ***наибольшая грань*** $\boldsymbol{b}^*$ длины $\beta^* = |\boldsymbol{b}^*|$, где $0 \leqslant \beta^* \leqslant n-1$. Строковая последовательность $\boldsymbol{f}$, приведенная выше, имеет две непустые грани: ab и $abaab$. Строка $\boldsymbol{g} = abaabaab$ имеет в точности такие же две грани, но в этом случае наибольшая грань $abaab$ частично перекрывает сама себя (точнее, перекрываются

грань и суффикс, с которым совпадает эта грань). Аналогично строка a^n имеет грани a, a^2, ..., a^{n-1}, среди которых грани a^i, a^{i+1}, ..., a^{n-1}, где $i = \lceil 9(n + 1)/2 \rceil$, перекрываются.[5] Далее будет показано, что перекрывающиеся грани являются характеристикой строковых последовательностей, содержащих повторяющиеся подстроки. Например, строку $\boldsymbol{g}$ можно записать в следующем виде $(aba)^2ab = (aba)(aba)ab$, где видны повторяющиеся подстроки.

Применим понятие грани для получения так называемой "нормальной формы" для данной непустой строки $\boldsymbol{x}$ длины n. Предположим, что для этой строки $\boldsymbol{x}$ найдена грань $\boldsymbol{b}$ и вычислена ее длина β. (В разделе 1.3 показано, как найти любую грань строки $\boldsymbol{x}$ за время порядка $\Theta(n)$.[6]) Из определения 1.2.2 имеем

$$\boldsymbol{x}[1..\beta] = \boldsymbol{x}[n - \beta + 1..n]. \tag{1.1}$$

Отсюда видно, что величина $p = n - \beta \geqslant 1$ определяет сдвиг позиций равных элементов этих подстрок (отметим, чем больше значение β, тем меньше значение p), т.е. для любых целых $i \in 1..\beta$ должно выполняться равенство

$$\boldsymbol{x}[i] = \boldsymbol{x}[i + p]. \tag{1.2}$$

Если $\beta = 0$ ($p = n$), то равенства (1.1) и (1.2) тривиальны. Если же $2\beta \geqslant n$ ($2p \leqslant n$), тогда строка $\boldsymbol{x}$ содержит, по крайней мере, две одинаковые смежные подстроки $\boldsymbol{x}[1..p]$ и $\boldsymbol{x}[p+1..2p]$. Более точно, строка $\boldsymbol{x}$ содержит $\lfloor n/p \rfloor$ одинаковых подстрок, каждая из которых имеет длину p, за которыми следует (возможно, пустой) суффикс длины $n \bmod p$.[7] Положив $r = n/p$ и введя обозначение $\boldsymbol{u} = \boldsymbol{x}[1..p]$, получаем следующее выражение для любой строки $\boldsymbol{x}$:

$$\boldsymbol{x} = \boldsymbol{u}^{\lfloor r \rfloor}\boldsymbol{u}', \tag{1.3}$$

где $\boldsymbol{u}' = \boldsymbol{x}[1..n - \lfloor r \rfloor p]$ — собственный префикс (возможно, пустой) подстроки $\boldsymbol{u}$. Если представить число r как сумму целой и дробной частей $r = \lfloor r \rfloor + k/p$ ($k \in 0..p - 1$) и интерпретировать запись $\boldsymbol{u}^{k/p} = \boldsymbol{u}[1..p]^{k/p}$ как подстроку $\boldsymbol{u}[1..k]$, тогда равенство (1.3) можно переписать в более компактной форме

$$\boldsymbol{x} = \boldsymbol{u}^r. \tag{1.4}$$

Число p называется ***периодом***, а число r — ***порядком*** (***показателем степени***) строковой последовательности $\boldsymbol{x}$. Префикс $\boldsymbol{u} = \boldsymbol{x}[1..p]$ назовем ***образующей*** (***образующей подстрокой***) строки $\boldsymbol{x}$. Отметим, что поскольку любая строка $\boldsymbol{x}$

[5] Здесь и далее запись $\lceil a \rceil$ обозначает наименьшее целое число, не меньшее числа a. — *Примеч. ред.*

[6] Объяснение обозначению $\Theta(n)$ дано в разделе 1.3. — *Примеч. ред.*

[7] Здесь и далее запись $\lfloor a \rfloor$ обозначает наибольшее целое число, не превышающее числа a. Для положительных чисел $\lfloor a \rfloor$ совпадает с целой частью этих чисел. — *Примеч. ред.*

имеет пустую грань $\boldsymbol{b} = \varepsilon$, то она всегда имеет ***тривиальный период*** $p = n$, ***тривиальный порядок*** $r = 1$ и ***тривиальную образующую*** $\boldsymbol{x}$.

Представляет интерес вычисление периода $p = p(\beta)$ и порядка $r = r(\beta)$ как функций длины β грани $\boldsymbol{b}$. Очевидно, что p является монотонно убывающей, а r монотонно возрастающей функциями от β. Поэтому если $\beta = \beta^*$ (β^* — длина наибольшей грани), то p принимает минимальное значение p^*, а r — максимальное значение r^*, которые называются ***минимальным периодом*** и ***максимальным порядком*** соответственно. В общем случае именно значения p^* и r^* представляют наибольший интерес, поэтому, если это не приводит к неопределенности, значение p^* будем называть просто ***периодом***, а значение r^* — ***порядком***. Аналогично в таком случае ***образующей*** будем называть подстроку $\boldsymbol{u} = \boldsymbol{x}[1..p^*]$.

Определение 1.2.3. Пусть строковая последовательность $\boldsymbol{x} = \boldsymbol{x}[1..n]$ имеет минимальный период p^* и, соответственно, $r^* = n/p^*$ и $\boldsymbol{u} = \boldsymbol{x}[1..p^*]$. Тогда декомпозиция

$$\boldsymbol{x} = \boldsymbol{u}^{r^*} \tag{1.5}$$

называется **нормальной формой** строковой последовательности $\boldsymbol{x}$. ■

На основе нормальной формы (1.5) можно ввести следующую полезную классификацию строковых последовательностей.

- Строковая последовательность $\boldsymbol{x}$ называется ***примитивной***, если $r^* = 1$. Если $r^* \neq 1$, то строковая последовательность называется ***периодической***.
- В случае $r^* \geqslant 2$ строковая последовательность $\boldsymbol{x}$ называется ***сильно периодической***, а при $1 < r^* < 2$ — ***слабо периодической***.
- Если r^* целое число и $r^* \geqslant 2$, то строковая последовательность $\boldsymbol{x}$ называется ***кратной*** (repetition).[8] В частных случаях $r^* = 2$ и $r^* = 3$ строку $\boldsymbol{x}$ будем называть ***квадратом*** и ***кубом*** соответственно.

Итак, строковая последовательность $\boldsymbol{x}$ может быть только или примитивной или периодической. Если строка $\boldsymbol{x}$ периодическая, то она может быть сильно или слабо периодической. Если $\boldsymbol{x}$ кратная, то она обязательно сильно периодическая. Отметим, что неравенство $r^* \geqslant 2$ возможно только тогда, когда $\boldsymbol{x}$ имеет грань длиной $\beta \geqslant n/2$.

Приведем несколько примеров применения введенных определений.

- Строка $\boldsymbol{x} = aaabaabab$ примитивна ($p^* = n$).
- Строка $\boldsymbol{f} = abaababaabaab = (abaababa)(abaab)$ слабо периодическая с периодом $p^* = 8$, порядком $r^* = 13/8$ и образующей $abaababa$.

[8] В русской литературе такие строки называются ***строго периодическими*** или даже просто ***периодическими***. — *Примеч. ред.*

- Строка $\boldsymbol{g} = abaabaab = (aba)^2ab$ строго периодическая с периодом $p^* = 3$, порядком $r^* = 8/3$ и образующей aba.
- Строка $\boldsymbol{x} = (ab)^4$ кратная с периодом $p^* = 2$, порядком $r^* = 4$ и образующей ab.
- Строка $\boldsymbol{x} = (abcabd)^2$ является квадратом с периодом $p^* = 6$ и образующей $abcabd$.

Отметим, что нормальная форма (1.5) фактически является "внутренним" паттерном в том смысле, каком это понятие применяется в части II этой книги: каждая строка $\boldsymbol{x}$ имеет паттерн, называемый "нормальной формой", с помощью которого можно охарактеризовать периодические свойства этой строки. В будущем те простые понятия и определения (грань, период, порядок и т.д.), которые введены в данном разделе, будут постоянно возникать при исследовании различных алгоритмов, рассмотренных в книге.

Упражнения 1.2

1. Попытайтесь "примирить" определения линейных строк, данные в разделах 1.1 и 1.2. Другими словами, докажите, что строка, являющаяся линейной в соответствии с одним определением, будет удовлетворять условиям линейности строк другого определения.

2. Предположим, что алфавит A состоит из α букв. Для произвольного неотрицательного целого n определите, сколько элементов множества A^* имеют длину n'. Сколько элементов этого множества имеют длину, не превышающую n?

3. Пусть в условиях предыдущего упражнения $A = \{0, 1\}$. Каждому элементу множества A^+ можно поставить в соответствие неотрицательное целое число. Определите, какому количеству элементов этого множества соответствует каждое целое число?

4. Основываясь на определении равенства строк, докажите, что $\varepsilon = \varepsilon^2$.

5. На основе определений равенства строк и лексикографического упорядочения докажите, что для произвольных строк $\boldsymbol{x}$ и $\boldsymbol{y}$ на упорядоченном алфавите равенство $\boldsymbol{x} = \boldsymbol{y}$ выполняется тогда и только тогда, когда $\boldsymbol{x} \not< \boldsymbol{y}$ и $\boldsymbol{y} \not< \boldsymbol{x}$. В частности, покажите, что этот результат справедлив в случае $\boldsymbol{x} = \boldsymbol{y} = \varepsilon$. Также покажите, что ε является лексикографически наименьшим элементом множества A^*.

6. Основываясь на обычном порядке строчных букв английского алфавита, расположите следующие строки в возрастающем лексикографическом порядке.

$$abbac, abbba, abb, abc, a, \varepsilon^2, ab, aba, \varepsilon b.$$

7. Докажите, что оператор $<$ сравнения строк, определенный в этом разделе, обладает свойством транзитивности, т.е. докажите, что

$$\boldsymbol{x} < \boldsymbol{y} \text{ и } \boldsymbol{y} < \boldsymbol{z} \Rightarrow \boldsymbol{x} < \boldsymbol{z}.$$

8. Дайте независимое определение отношения порядка $>$, затем на основе этого определения совместно с определением отношения $<$, данного в этом разделе, покажите, что отношение $\boldsymbol{x} > \boldsymbol{y}$ имеет место только тогда, когда выполняется отношение $\boldsymbol{y} < \boldsymbol{x}$.

9. Пусть дана некоторая строка $\boldsymbol{x}$ длины n. Определите длины следующих подстрок строки $\boldsymbol{x}$ и наложите условия на i и k, при которых будут справедливы ваши ответы.

а) $\boldsymbol{x}[i..i+k-1]$;

б) $\boldsymbol{x}[i-k+1..i]$;

в) $\boldsymbol{x}[i+1..k-1]$;

г) $\varepsilon\boldsymbol{x}[1..k]$.

10. Какое максимальное число различных подстрок можно образовать из строки длины n? Приведите пример строки, на которой достигается этот максимум. Как можно охарактеризовать множество таких строк?

11. В предыдущем упражнении не накладывается неявного ограничения на размер алфавита. Эта задача становится более интересной (и значительно более трудной), если наложить условие, что размер алфавита конечен и фиксирован. Такая задача на сегодняшний день пока не решена, что вы можете предложить для ее решения?

Совет. По-видимому, решение этой задачи следует начать с изучения следующего вопроса: для данных положительных чисел $\alpha = |A|$ и k надо найти наибольшую строку на алфавите A, которая содержит не более одной заданной подстроки длины k.

12. Опишите алгоритм, который находил бы все различные подстроки строки $\boldsymbol{x}$. Докажите корректность вашего алгоритма и оцените его асимптотическую сложность (постарайтесь добиться порядка $O(n^2)$).

13. Пусть $\boldsymbol{y}$ — непустая строка длины m, а k — положительное целое число. Определите $|\boldsymbol{x}|$ как функцию от m и k в каждом из следующих случаев.

а) $\boldsymbol{x} = \boldsymbol{y}^k$;

б) $\boldsymbol{x} = \boldsymbol{y}^{|y|}$

в) $x = y^{|y|^k}$.

14. Какова длина строки

$$\boldsymbol{x} = (ab)^n a(ab)^{n-1} a \cdots (ab)^2 a(ab)a?$$

15. Как определить префикс, суффикс и грань для пустой строки ε?

16. Определите наибольшую грань, период и порядок для каждой из следующих строк и затем классифицируйте их как примитивную, сильно периодическую, слабо периодическую или кратную.

а) $abaababaabaababaababa$;

б) $abcabacabcbacbcacb$;

в) $abcabdabcabdabcabd$;

г) $a(ab)^3a(ab)^3aa$.

17. Строка $\boldsymbol{x}$ называется ***палиндромной последовательностью*** (или просто ***палиндромом***), если она читается справа налево так же, как слева направо. Если говорить более точно, то строка $\boldsymbol{x}$ называется ***палиндромом***, если для каждого $i = 1, 2, \ldots, \lfloor n/2 \rfloor$ выполняется равенство $\boldsymbol{x}[i] = \boldsymbol{x}[n - i + 1]$. Докажите или опровергните утверждение, что каждая грань палиндрома сама является палиндромом.

Примечание. Некоторые нетривиальные примеры палиндромов можно найти по электронным адресам

`www.growndodo.com/wordplay/palindromes/dogseesada.html`

`complex.gmu.edu/neural/personnel/ernie/witty/palindromes.html`

18. Покажите, что период и порядок "хорошо определены"; другими словами, если имеет место равенство $\boldsymbol{x} = \boldsymbol{u}^k\boldsymbol{u}'$ для некоторой строки $\boldsymbol{u}$ и целого числа k, где $\boldsymbol{u}'$ является собственным префиксом строки $\boldsymbol{u}$, тогда существует единственная соответствующая грань строки $\boldsymbol{x}$.

19. Докажите, что если $\boldsymbol{u}^{r^*}$ является нормальной формой строки $\boldsymbol{x}$, тогда подстрока $\boldsymbol{u}$ не может быть кратной. (Важность и полезность этого утверждения будут показаны в разделе 2.3.)

20. Пусть $(ab)^{3/2} = aba$. Какое из следующих равенств верно: $((ab)^{3/2})^2 = (aba)^2$ или $((ab)^{3/2})^2 = (ab)^{(3/2)2} = (ab)^3$?

21. Докажите, что не существует строк с двумя различными примитивными образующими.

22. Предложите способ вычисления количества строк на алфавите $\{a, b\}$, которые имеют заданную длину n и являются примитивными.

Совет. Заметьте, что при любом нечетном n произвольную строку $\boldsymbol{x}[1..n]$ можно сформировать из строк $\boldsymbol{x}[1..(n-1)/2]$ и $\boldsymbol{x}[(n+1)/2..n]$, а при четном n — из строк $\boldsymbol{x}[1..n/2]$ и $\boldsymbol{x}[n/2+1..n]$. Если вы встречаете затруднения при решении этого упражнения, обратитесь к работам [105, 106].

23. Приведенная в этом разделе классификация строковых последовательностей основана на значениях порядка (показателя степени) r^*. Повторите эту же классификацию на основе значений длины β^* наибольшей грани.

24. Нормальную форму строки $\boldsymbol{x}$, введенную в этом разделе, более правильно назвать *левой* нормальной формой, поскольку вместо формулы $\boldsymbol{x} = \boldsymbol{u}^r\boldsymbol{u}'$, где $\boldsymbol{u}'$ — префикс $\boldsymbol{u}$, можно также ввести запись $\boldsymbol{x} = \boldsymbol{v}'\boldsymbol{v}^s$, где $\boldsymbol{v}'$ — суффикс $\boldsymbol{v}$.

а) получите формулу для *правой* нормальной формы;

б) покажите, что классификация строковых последовательностей, построенная на основе левых нормальных форм, совпадает с такой же классификацией, построенной на основе правых нормальных форм.

25. Может ли петля быть подстрокой линейной строки? Может ли линейная строка быть подстрокой петли?

26. Нарисуйте дерево классификации строковых последовательностей, введенной в этом разделе.

1.3 Периодичность

Как будет показано в последующих главах, наибольшая грань строковой последовательности $\boldsymbol{x}$ порождает ряд числовых характеристик, которые полезны не только для классификации строк. Поэтому в данном разделе мы подробно рассмотрим вопрос о нахождении наибольших граней, в частности, покажем алгоритм нахождения наибольших граней, выполняющий вычисления за время порядка $\Theta(n)$.

Существует очевидный способ вычисления длины β^* наибольшей грани $\boldsymbol{b}^*$ строки $\boldsymbol{x}$. Сначала положим $\beta^* \leftarrow 0$.[9] Затем сравниваем подстроки $\boldsymbol{x}[1]$ и $\boldsymbol{x}[n]$; если они равны, то полагаем $\beta^* \leftarrow 1$. Далее сравниваем $\boldsymbol{x}[1..2]$ и $\boldsymbol{x}[n-1..n]$; если они равны, полагаем $\beta^* \leftarrow 2$. Продолжаем этот процесс, каждый раз сравнивая префиксы и суффиксы, длины которых на 1 больше, чем на предыдущем шаге, и увеличивая на 1 значение β^*, если они равны. На последнем шаге сравниваются подстроки $\boldsymbol{x}[1..n-1]$ и $\boldsymbol{x}[2..n]$, и если они равны, то β^* получает значение $n-1$. Конечное значение β^* равно длине наибольшей грани $\boldsymbol{b}^*$ строки $\boldsymbol{x}$. Мы не даем этот алгоритм в формальной записи, поскольку он не эффективен — в упражнении 1.3.1 определяется его асимптотическая сложность.

Предположим, что в описанном алгоритме последовательные значения β^* сохраняются в массиве (или в строке!) $\boldsymbol{\beta}[1..n]$. Тогда для любого $i = 1, 2, \ldots, n$ $\boldsymbol{\beta}[i]$ дает длину наибольшей грани подстроки $\boldsymbol{x}[1..i]$. Назовем $\boldsymbol{\beta}[1..n]$ ***массивом граней*** строки $\boldsymbol{x}[1..n]$. Здесь $\boldsymbol{\beta}[n]$ представляет длину наибольшей грани строки $\boldsymbol{x}$, т.е. это то значение, которое необходимо для построения нормальной формы строки $\boldsymbol{x}$.

Сделаем следующие заключения относительно массива граней и самих граней.

[9] Здесь и далее автор использует одну из форм нотаций для записи вычислительных алгоритмов. В этой нотации символ "$\leftarrow$" означает оператор присваивания, который в других нотациях обозначается как "$=$" или "$:=$". — *Примеч. ред.*

- $\beta[1] = 0$ (поскольку ε является наибольшей гранью строки $\boldsymbol{x}[1..1]$).
- Если $\boldsymbol{x}[1..i]$ имеет грань длины $k > 0$ (здесь $2 \leqslant i \leqslant n$), тогда подстрока $\boldsymbol{x}[1..i-1]$ имеет грань длины $k-1$. Отсюда следует, что для всех i, $1 \leqslant i \leqslant$ $\leqslant n-1$, справедливо неравенство $\boldsymbol{\beta}[i+1] \leqslant \boldsymbol{\beta}[i]+1$.
- Для всех i, $1 \leqslant i \leqslant n-1$, равенство $\boldsymbol{\beta}[i+1] = \boldsymbol{\beta}[i]+1$ выполняется только тогда, когда $\boldsymbol{x}[i+1] = \boldsymbol{x}[\boldsymbol{\beta}[i]+1]$ (поскольку $\boldsymbol{\beta}[i]+1$ — позиция в строке $\boldsymbol{x}$ элемента, расположенного справа от префикса $\boldsymbol{x}[1..\boldsymbol{\beta}[i]]$, который является наибольшей гранью подстроки $\boldsymbol{x}[1..i]$).
- Если $\boldsymbol{b}$ — грань строки $\boldsymbol{x}$, а $\boldsymbol{b}'$ — грань подстроки $\boldsymbol{b}$, тогда $\boldsymbol{b}'$ является гранью строки $\boldsymbol{x}$.

Эти заключения, в частности, третье и четвертое, показывают, что имеется возможность вычислить $\boldsymbol{\beta}[i+1]$ на основании значений $\boldsymbol{\beta}[1]$, $\boldsymbol{\beta}[2]$, ..., $\boldsymbol{\beta}[i]$. Здесь основная трудность возникает тогда, когда $\boldsymbol{\beta}[i] > 0$ и одновременно $\boldsymbol{x}[i+1] \neq$ $\neq \boldsymbol{x}[\boldsymbol{\beta}[i]+1]$. В этом случае надо обратиться ко второй по величине грани подстроки $\boldsymbol{x}[1..i]$. Если эта грань пуста, тогда $\boldsymbol{\beta}[i+1] = 1$, если $\boldsymbol{x}[1] = \boldsymbol{x}[i+1]$, или $\boldsymbol{\beta}[i+1] = 0$, если $\boldsymbol{x}[1] \neq \boldsymbol{x}[i+1]$. Если же эта грань не пуста (обозначим ее длину как $\boldsymbol{\beta}^2[i] > 0$), тогда необходимо сравнить элементы $\boldsymbol{x}[i+1]$ и $\boldsymbol{x}[\boldsymbol{\beta}^2[i]+1]$. В случае их равенства полагаем $\boldsymbol{\beta}[i+1] \leftarrow \boldsymbol{\beta}^2[i]+1$. Если же эти элементы не равны, то в этом случае надо рассмотреть третью по величине грань подстроки $\boldsymbol{x}[1..i]$ и повторить описанные выше действия. При необходимости рассматриваются следующие по величине грани подстроки $\boldsymbol{x}[1..i]$.

Рассмотрим следующий пример. Пусть $\boldsymbol{x} = abaababa*$, где символ $*$ показывает, что 9-я буква в строке $\boldsymbol{x}$ пока не определена. Массив граней строки $\boldsymbol{x}$ имеет вид 00112323?, где символ ? указывает на то, что остался не вычисленным только последний элемент массива. Отметим, что для $i = 8 \boldsymbol{\beta}[i] = 3$. Если выполняются равенства

$$\boldsymbol{x}[9] = \boldsymbol{x}[i+1] = \boldsymbol{x}[\boldsymbol{\beta}[i]+1] = \boldsymbol{x}[4] = a,$$

тогда сразу получаем $\boldsymbol{\beta}[9] = 4$. Если эти равенства не выполняются, тогда надо рассмотреть вторую по величине грань подстроки $\boldsymbol{x}[1..8]$. Имеем

$$\boldsymbol{\beta}^2[8] = \boldsymbol{\beta}[\boldsymbol{\beta}[8]] = \boldsymbol{\beta}[3] = 1.$$

Если выполняются равенства

$$\boldsymbol{x}[8+1] = \boldsymbol{x}[\boldsymbol{\beta}^2[8]+1] = \boldsymbol{x}[2] = b,$$

тогда получим $\boldsymbol{\beta}[9] = 2$. Если же элемент $\boldsymbol{x}[9]$ не совпадает ни с a ни с b, а принимает значение новой буквы (например, c), то в этом случае $\boldsymbol{\beta}[9] = 0$.

Рассмотрим величины $\boldsymbol{\beta}^j[i]$, $j = 1, 2, \ldots, k$, представляющие значения длин j-х по величине граней подстроки $\boldsymbol{x}[1..i]$. (Здесь мы полагаем, что $\boldsymbol{\beta}^1[i] = \boldsymbol{\beta}[i]$

и $\boldsymbol{\beta}^k[i] = 0$.) Поскольку для каждого j ($j = 2, 3, \ldots, k$) $\boldsymbol{\beta}^j[i] < \boldsymbol{\beta}^{j-1}[i]$, то отсюда следует, что $\boldsymbol{x}[1..\boldsymbol{\beta}^j[i]]$ является гранью подстроки $\boldsymbol{x}[1..\boldsymbol{\beta}^{j-1}[i]]$. Таким образом, j-я по величине грань подстроки $\boldsymbol{x}[1..i]$ будет гранью $(j-1)$-й по величине грани подстроки $\boldsymbol{x}[1..i]$. Символически это запишется как

$$\boldsymbol{\beta}^j[i] = \boldsymbol{\beta}[\boldsymbol{\beta}^{j-1}[i]]. \tag{1.6}$$

Если $\boldsymbol{x}[i+1] \neq \boldsymbol{x}[\boldsymbol{\beta}^{j-1}[i]+1]$, тогда с помощью формулы (1.6) можно определить следующую позицию элемента, с которым будет сравниваться элемент $\boldsymbol{x}[i+1]$, — это будет элемент $\boldsymbol{x}[\boldsymbol{\beta}^j[i]+1]$.

Приведенные рассуждения можно обобщить в виде следующей леммы.

Лемма 1.3.1. Пусть для любого целого $n \geqslant 1$ $\boldsymbol{x} = \boldsymbol{x}[1..n]$ — строковая последовательность с массивом граней $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$. Пусть k — наименьшее целое, такое, что $\boldsymbol{\beta}^k[n] = 0$. Тогда

а) для каждого целого $j \in 1..k$ строка $\boldsymbol{x}[1..n]$ имеет грань $\boldsymbol{x}[1..\boldsymbol{\beta}^j[i]]$;

б) для произвольной буквы λ возможны только такие грани строки $\boldsymbol{x}[1..n+1] = \boldsymbol{x}[1..n]\lambda$, длины которых являются членами убывающей последовательности

$$\langle \boldsymbol{\beta}[n]+1, \boldsymbol{\beta}^2[n]+1, \ldots, \boldsymbol{\beta}^k[n]+1, 0 \rangle. \tag{1.7}$$

■

На основе равенства (1.6) можно построить эффективный алгоритм вычисления массива $\boldsymbol{\beta}[1..n]$, который содержал бы значения длин наибольших граней всех префиксов $\boldsymbol{x}[1..i]$ данной строки $\boldsymbol{x}$. Этот алгоритм представлен ниже как алгоритм 1.3.1. Как будет показано в главе 7, все эти значения длин (не только $\boldsymbol{\beta}[n]$) находят применение при сравнении различных строковых последовательностей, когда $\boldsymbol{x}$ является паттерном.

Алгоритм 1.3.1. Вычисление массива граней

```
▷ Вычисление массива граней β строки x[1..n]
β[1] ← 0
for i ← 1 to n − 1 do
    b ← β[i]
    while b > 0 and x[i + 1] ≠ x[b + 1] do
        b ← β[b]
    if x[i + 1] = x[b + 1] then
        β[i + 1] ← b + 1
    else
        β[i + 1] ← 0
```

Теорема 1.3.2. Алгоритм 1.3.1 корректно вычисляет массив $\boldsymbol{\beta}[1..n]$.

Доказательство. Сначала рассмотрим цикл **while**: в этом цикле на основе равенства (1.6) $b = \boldsymbol{\beta}^{j-1}[i]$ заменяется на $b = \boldsymbol{\beta}^{j}[i]$. Поскольку b монотонно убывает, этот цикл должен завершиться. Цикл **for** завершится, когда будет достигнуто равенство $i = n$. Следовательно, алгоритм 1.3.1 завершится за $n - 1$ шагов.

Теперь рассмотрим, что происходит после завершения цикла **while**. Выход из этого цикла осуществится тогда, когда выполнится хотя бы одно из равенств $b = 0$ или $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$. Если выполняется равенство $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$, то должно произойти присваивание $\boldsymbol{\beta}[i+1] \leftarrow b+1$ независимо от того, будет ли b равно 0 или нет. С другой стороны, если $b = 0$, тогда должно произойти присваивание $\boldsymbol{\beta}[i+1] \leftarrow 0$. Эти варианты корректно обрабатываются условным оператором **if**, поскольку после выполнения цикла **while** имеем наибольшую возможную длину из последовательности (1.7). Таким образом, алгоритм 1.3.1 правильно вычисляет $\boldsymbol{\beta}[i+1]$ для любого $i \in 0..n-1$. ■

Теперь вычислим время, необходимое для выполнения алгоритма. Все шаги цикла **for**, за исключением, возможно, цикла **while**, выполняются за константное время. Поэтому алгоритм 1.3.1 выполняется за время $\Theta(n)$ плюс общее время выполнения циклов **while**. Для получения оценки этого общего времени рассмотрим те значения, которые может принимать b во время выполнения алгоритма: b первоначально равно нулю и возрастает на 1 при выполнении оператора $b \leftarrow \boldsymbol{\beta}[b]$ в начале $(i+1)$-й итерации цикла **for** только в том случае, если выполнялось присвоение $\boldsymbol{\beta}[i+1] \leftarrow b+1$ в конце i-й итерации. Таким образом, b возрастает на единицу не более $n-2$ раза, и каждое такое возрастание приходится на одну итерацию цикла **for**. В то же время уменьшение значения b может происходить только внутри цикла **while**. Поскольку количество уменьшений значения b (b всегда уменьшается не менее чем на 1) не может превосходить количества увеличений этого значения (увеличение всегда происходит в точности на 1), поэтому оператор присвоения $b \leftarrow \boldsymbol{\beta}[b]$ внутри цикла **while** выполняется не более $n-2$ раза. Таким образом, мы доказали следующую теорему.

Теорема 1.3.3. Алгоритм 1.3.1 для выполнения требует $\Theta(n)$ и фиксированного дополнительного и фиксированного дополнительного пространства памяти. ■

Обсуждение 1.3.1. В этом разделе мы впервые рассмотрели вопрос об асимптотической сложности алгоритма и об объеме памяти, необходимой для его выполнения. Эти вопросы будут многократно исследоваться на протяжении всей книги. Поэтому сейчас уделим достаточно внимания тем предположениям и рассуждениям, которые лежат в основе утверждений, подобных теореме 1.3.3.

- В алгоритме 1.3.1 необходимо хранить целые числа i и b и элементы массива $\boldsymbol{\beta}$, которые также являются неотрицательными целыми числами. Каждое из

этих чисел не превышает $n-1$, и поэтому для хранения одного числа надо зарезервировать $\lceil \log_2 n \rceil$ битов. Более точно, для каждого элемента массива $\boldsymbol{\beta}$ необходима память объемом $\log_2 n/w = O(\log n)$, где w — длина машинного слова компьютера.

Отсюда следует, что для хранения значений i и b также необходима память объемом $O(\log n)$, а для хранения всего массива $\boldsymbol{\beta}$ — память объема $O(n \log n)$. Таким образом, общее время выполнения алгоритма становится равным $O(n \log n)$, а не $\Theta(n)$, как указано в теореме 1.3.3. Мы разрешим это противоречие здесь и далее по всей книге путем дополнительного предположения, что размер задачи не может быть каким угодно большим, а всегда имеет некоторую, вполне реалистическую, границу. Так, если n — размер задачи, предположим существование такой константы k (по возможности, небольшой по величине), что выполняется неравенство $\log_2 n/w \leqslant k$. С практической точки зрения это не слишком ограничительное предположение, даже если мы примем, что $k = 1$. Действительно, поскольку современные компьютеры 32- или 64-разрядные, т.е. $w \geqslant 32$, то из неравенства $\log_2 n/w \leqslant 1$ получаем границу для числа n: $n \leqslant 2^{32} \approx 4{,}3 \times 10^9$. Если эта граница кажется слишком ограничительной, то можно принять $k = 2$ — тогда для n получаем границу $n \leqslant 2^{64} \approx 1{,}8 \times 10^{19}$.

Большое или маленькое последнее число? Предположим, что надо прочитать $1{,}8 \times 10^{19}$ машинных слов (строковых элементов) на компьютере со скоростью один миллиард слов в секунду. Тогда на выполнение всей операции чтения потребуется примерно 570 лет. Таким образом, только наши далекие потомки смогут увидеть завершение этой операции!

Я не являюсь оппонентом "блюстителей чистоты математических нравов", которые требуют при вычислении асимптотических оценок времени выполнения или сложности алгоритмов обязательно умножать размер задачи n на "вычислительный фактор" $\log n$. В определенном смысле такой фактор действительно существует и его следует учитывать "в теории", но в обычной вычислительной практике он излишен.

■ Остановимся также на обозначениях O, Ω и Θ, которые используются на протяжении всей книги для характеризации асимптотических оценок времени и объема памяти, необходимых для выполнения алгоритмов. Мы предполагаем, что читатель знаком с этими обозначениями; если это не так, советуем обратиться к работам [134, 198].[10] Здесь поясним различия между ними.

[10] Хотя автор отсылает читателя к соответствующим источникам, приведем краткое определение этих понятий. Говорят, что величина $T(n)$ имеет порядок $O(f(n))$, если существуют положительные константы c и n_0, такие, что для всех n, больших или равных n_0, выполняется неравенство $|T(n)| \leqslant cf(n)$. Величина $T(n)$ имеет порядок $\Omega(f(n))$, если существуют положительные константы c и n_0, такие, что для бесконечного числа значений n (но не обязательно для всех значений n),

Требует некоторых пояснений применение обозначения $O(n)$. Если мы говорим "время выполнения (алгоритма) имеет порядок $O(n)$", это означает, что мы рассматриваем время выполнения как функцию от размера n некоторой конкретной задачи, являющейся представителем *класса задач*. Другими словами, эта функция представляет характеристику целого класса задач, но "вычисленная" на некоторой конкретной (экстремальной) задаче. Как правило, утверждение "время выполнения (алгоритма) имеет порядок $O(n)$" выполняется только для очень больших значений n (фактически, для бесконечных значений n) и для любой задачи размера n из данного класса задач.

В книге мы трактуем время выполнения и требуемый объем памяти как функцию размера для всего данного класса задач. Тогда утверждение

"время выполнения имеет порядок $\Theta(n)$" (1.8)

мы будем понимать так, что для всех задач данного класса, размер которых превышает некоторое фиксированное значение n_0, алгоритм требует времени выполнения не меньше $k_1 n$ и не больше $k_2 n$, где k_1 и k_2 также фиксированные (не зависящие от n) числа. Утверждение (1.8) более слабое, чем аналогичное утверждение с использованием $O(n)$. Фактически, здесь утверждается, что для всех задач данного класса, размер которых превышает значение n_0, алгоритм требует времени выполнения не больше $k_2 n$. В этом случае возможны ситуации, когда в данном классе задач есть такие задачи, на которых для выполнения алгоритма требуется объем памяти, пропорциональный, например $\log n$, $n^{1/2}$, $n/\log n$ или даже 1, т.е. асимптотически требуемый объем памяти может быть меньше n. Причем таким свойством могут обладать *все* задачи данного класса.

В книге будем использовать обозначение Θ для описания свойств алгоритмов только тогда, когда будут строго выполняться все условия утверждения (1.8). При использовании обозначения $O(f(n))$ (соответственно, обозначения $\Omega(f(n))$) будем понимать, что в данном классе задач существует ряд задач, на которых данное свойство имеет в точности порядок $f(n)$; но в этом классе задач также существует другой набор задач, на которых данное свойство имеет меньший (соответственно, больший) порядок, чем $f(n)$. ■

Итак, алгоритм 1.3.1 требует времени выполнения $\Theta(n)$ на всех задачах размера n. Поскольку любой алгоритм, вычисляющий массив граней, обязательно хотя бы один раз обратится к каждой из n позиций в исходной строке, то отсюда следует вывод, что алгоритм 1.3.1 ***асимптотически оптимален***: не возможен

больших или равных n_0, выполняется неравенство $|T(n)| \geqslant cf(n)$. Определение обозначения $\Theta(f(n))$ приведено в последующих абзацах. — *Примеч. ред.*

алгоритм, вычисляющий массив $\boldsymbol{\beta}[1..n]$, с временем выполнения, меньшим $\Theta(n)$, при этом требующий фиксированного объема памяти для любой задачи размера n.

Как показано в разделе 4.1, алгоритм 1.3.1 также можно назвать, в определенном смысле, ***онлайновым***[11], поскольку на каждом шаге алгоритма результат вычислений для текущей позиции i получается путем просмотра исходной строки слева направо. Поэтому без повторных пересчетов алгоритм позволяет получить результат не только для исходной строки, но и для любого ее префикса. Однако если говорить более строго, алгоритм 1.3.1 нельзя считать в полном смысле онлайновым, поскольку на любом шаге алгоритма может потребоваться вся исходная строка.

В качестве примера для вычисления граней, еще раз рассмотрим строковую последовательность из раздела 1.2:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f} =$	a	b	a	a	b	a	b	a	a	b	a	a	b

Для этой строки массив $\boldsymbol{\beta}[1..n]$ содержит значения 0011232345645. Отметим, что если в строке есть кратные подстроки (как, например, в строке $\boldsymbol{f}$), то значения в массиве граней могут не составлять монотонно возрастающую последовательность, т.е. последовательные значения в массиве могут как убывать, так и возрастать. Например, в данном случае $\boldsymbol{\beta}[6..8] = 323$ и $\boldsymbol{\beta}[11..13] = 645$.

Способ вычисления массива $\boldsymbol{\beta}$ показывает, что *все* грани строки $\boldsymbol{x}$ (в действительности, грани всех префиксов строки $\boldsymbol{x}$) можно вычислить на основе значений массива $\boldsymbol{\beta}$. Это вытекает из равенства (1.6), которое показывает, что вторая по величине грань подстроки $\boldsymbol{x}[1..i]$ является наибольшей гранью подстроки $\boldsymbol{x}[1..\boldsymbol{\beta}[i]]$. Сформулируем этот важный результат в виде теоремы.

Теорема 1.3.4. Массив граней $\boldsymbol{\beta}$ любой строковой последовательности $\boldsymbol{x}$ содержит всю информацию, для вычисления всех граней, периодов и порядков любого непустого префикса и порядков любого непустого префикса строки $\boldsymbol{x}$. ■

Обсуждение 1.3.2. Массив граней — это одна из возможных структур строковых данных, представляющих информацию в линейной информацию в линейной форме. Чтобы показать это, вернемся к примеру со строкой $\boldsymbol{f}$. В данном случае с помощью значений массива $\boldsymbol{\beta}$ можно получить следующую информацию о гранях строки $\boldsymbol{f}$.

[11] Онлайновым считается алгоритм, который может работать в режиме реального времени или в режиме оперативной обработки данных (т.е. по мере поступления информации). — *Примеч. ред.*

i	*длина граней*
13	$5, 2, 0$
12	$4, 1, 0$
11	$6, 3, 1, 0$
10	$5, 2, 0$
$\vdots$	$\vdots$

Таким образом, информацию, содержащуюся в массиве граней, можно представить в явном виде в форме более удобной таблицы, подобной приведенной выше. Однако отметим, что для вычисления такой таблицы требуется уже не линейное (по n) время, а время порядка $\Theta(n \log n)$. ■

В заключение данного раздела приведем важный результат, который устанавливает числовую зависимость между различными периодами одной строки.

Теорема 1.3.5 (Лемма периодичности [87, 167]). Пусть p и q — два периода строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ и пусть $d = \gcd(p, q)$.[12] Если выполняется неравенство $p+q \leqslant n+d$, тогда d также является периодом строки $\boldsymbol{x}$.

Доказательство этой теоремы интересно. Оно построено на сведении условий теоремы, базирующихся на параметрах (d, p, q, n) к эквивалентному множеству условий на основе параметров $(d, p, q-n, n-p)$. Эта замена условий позволяет использовать алгоритм Евклида [135] для вычисления наибольшего общего делителя $d = \gcd(p, q)$, и, фактически, корректность доказательства теоремы зависит от корректности алгоритма Евклида. Напомним, что при $q > p$ этот алгоритм вычисляет значение d, основываясь на равенстве

$$d = \gcd(p, q) = \gcd(p, g - p), \tag{1.9}$$

и останавливается после конечного числа шагов при достижении $d = \gcd(d, d)$.

Обозначим через $\mathrm{H}(d, p, q, n)$ условия теоремы так, как они сформулированы: p и q являются периодами строки $\boldsymbol{x}[1..n]$, $d = \gcd(p, q)$ и выполняется неравенство $p + q \leqslant n + d$. Без потери общности можем положить, что $q > p$. (Если $q = p$, то теорема тривиальна.) В этих обозначениях запись $\mathrm{H}(d, p, q-p, n-p)$ определяет условия, что p и $q - p$ являются периодами строки $\boldsymbol{x}[1..n-p]$, $d = \gcd(p, q-p)$ и выполняется неравенство $q \leqslant n - p + d$. Покажем, что из условий $\mathrm{H}(d, p, q, n)$ следуют условия $\mathrm{H}(d, p, q-p, n-p)$. Это позволит вместо строки $\boldsymbol{x}[1..n]$ рассматривать строку $\boldsymbol{x}[1..n-p]$. Попытаемся сводить одни условия к другим последовательно и за конечное число шагов, пока в результате не будет достигнуто условие $\mathrm{H}(d, d, d, n')$, которое выполняется тривиально, где d — период

[12] Здесь и далее $\gcd(p, q)$ обозначает наибольший общий делитель чисел p и q. — *Примеч. ред.*

строки $\boldsymbol{x}[1..n']$. Далее покажем, что d является периодом $\boldsymbol{x}[1..n-p]$ только тогда, когда d будет периодом $\boldsymbol{x}$; следовательно, если d будет периодом строки $\boldsymbol{x}[1..n']$, то d будет и периодом строки $\boldsymbol{x}[1..n]$.

Итак, пусть выполняются условия $\mathrm{H}(d,p,q,n)$. Обозначим через $\beta_1 = n-p$ и $\beta_2 = n-q$ длины граней $\boldsymbol{b}_1$ и $\boldsymbol{b}_2$, соответствующих периодам p и q. Тогда, как следует из леммы 1.3.1, $\boldsymbol{b}_2$ является гранью $\boldsymbol{b}_1$. Другими словами, подстрока $\boldsymbol{x}[1..\beta_1] = \boldsymbol{x}[1..n-p]$ имеет грань $\boldsymbol{x}[1..\beta_2]$ с периодом

$$\beta_1 - \beta_2 = n - p - n + q = q - p.$$

Поскольку $d = \gcd(p,q)$, то должно выполняться неравенство $d \leqslant q-p$. Из последнего неравенства и неравенства $p+q \leqslant n+d$ получаем

$$p \leqslant q - d < n - p. \tag{1.10}$$

Следовательно, поскольку строка $\boldsymbol{x}$ имеет период p, то и подстрока $\boldsymbol{x}[1..n-p]$ должна иметь такой же период p. Далее надо показать, что $\boldsymbol{x}[1..n-p]$ имеет как период p, так и период q. Из (1.9) имеем $d = \gcd(p, q-p)$, а неравенство $p+q \leqslant n+d$ можно переписать как

$$p + (q-p) \leqslant (n-p) + d.$$

Это доказывает, что из условий $\mathrm{H}(d,p,q,n)$ вытекают условия $\mathrm{H}(d,p,q-p,n-p)$.

Осталось показать, что d является периодом строки $\boldsymbol{x}$ тогда и только тогда, когда он является периодом подстроки $\boldsymbol{x}[1..n-p]$. Очевидно, что если d является периодом $\boldsymbol{x}$, то он также является периодом $\boldsymbol{x}[1..n-p]$. Чтобы доказать утверждение в другую сторону, заметим, что если d является периодом $\boldsymbol{x}[1..n-p]$, то, согласно (1.10), он также является периодом строки $\boldsymbol{x}[1..p]$. Поскольку d должен быть делителем p, то справедливо равенство

$$\boldsymbol{x}[1..p] = \boldsymbol{x}[1..d]^{p/d}.$$

Так как подстрока $\boldsymbol{x}[1..n-p]$ является гранью строки $\boldsymbol{x}$, тогда d должен быть периодом строки

$$\boldsymbol{x}[1..d]^{p/d}\boldsymbol{x}[1..n-p] = \boldsymbol{x}[1..p]\boldsymbol{x}[p+1..n] = \boldsymbol{x}[1..n],$$

что и требовалось. Таким образом, мы доказали, что подстрока $\boldsymbol{x}[1..n-p]$ имеет период d только в том случае, если d является периодом исходной строки $\boldsymbol{x}$.

Итак, после конечного числа шагов применения алгоритма Евклида для некоторого $n' \in d..n-p$ мы пришли к условиям $\mathrm{H}(d,d,d,n')$, где d является периодом строки $\boldsymbol{x}[1..n']$ и, как доказано, периодом исходной строки $\boldsymbol{x}$. ■

В разделе 1.4 покажем одно из многих возможных применений этой теоремы. Здесь рассмотрим применение леммы периодичности к строке

$$\boldsymbol{x} = abaaabaaabaaabaaab,$$

которая имеет длину $n = 18$ и периоды $q = 12$, $p = 8$. Поскольку $d = \gcd(p, q) = 4$ и $p + q = 20 \leqslant n + d = 22$, то отсюда следует, что $d = 4$ также является периодом строки $\boldsymbol{x}$.

Иногда лемма периодичности выполняется даже тогда, когда не все ее условия реализованы. Например, строка

$$\boldsymbol{y} = abaaabaaabaaab$$

имеет длину $n = 18$ и периоды $q = 12$, $p = 8$. Но в этом случае $p + q = 20 > n + d = 18$. Однако все равно $d = 4$ является периодом строки $\boldsymbol{x}$.

Необходимость условия $p + q \leqslant n + d$ в теореме 1.3.5 показывает пример строки $\boldsymbol{z} = abaaba$ длиной $n = 6$ и с периодами $q = 5$, $p = 3$. В этом примере $p + q = 8 = n + d + 1$ и $d = \gcd(3, 5) = 1$ *не является* периодом строки $\boldsymbol{z}$.

Недавно лемма периодичности была обобщена на случай строк с тремя периодами [43].

Упражнения 1.3

1. Определите асимптотическую сложность "очевидного" алгоритма нахождения наибольших граней данной строки $\boldsymbol{x}$. Приведите пример строки произвольной длины n, на которой этот алгоритм будет выполняться наиболее долго. Подсчитайте, сколько необходимо произвести сравнений между буквами строки в самом худшем случае.
2. Поскольку необходимо найти наибольшие грани строки $\boldsymbol{x}$, то более эффективным кажется алгоритм, в котором возможные грани проверяются в *нисходящем* порядке относительно длин этих граней. Разработайте такой алгоритм и определите его асимптотическую сложность. В каком случае время выполнения такого алгоритма будет одинаковым с временем выполнения "обычного" алгоритма? В каком случае время его выполнения будет линейным по длине исходных строк?
3. Если массив $\boldsymbol{\beta}$ рассматривать как строку, то на каком алфавите определена эта строка?
4. Докажите утверждение, сделанное в этом разделе, что если $\boldsymbol{b}$ является гранью строки $\boldsymbol{x}$, а $\boldsymbol{b}'$ — гранью подстроки $\boldsymbol{b}$, то $\boldsymbol{b}'$ является гранью строки $\boldsymbol{x}$.
5. Докажите следующую лемму (доказанную Яндонгом Янгом (Jiandong Jiang)).

   Если $\boldsymbol{x} = \boldsymbol{b}\boldsymbol{x}'\boldsymbol{b}$, где $\boldsymbol{b}$ — примитивная, а $\boldsymbol{x}'$ — пустая или примитивная строка и $\boldsymbol{x}' \neq \boldsymbol{b}$, тогда $\boldsymbol{b}$ может быть только непустой гранью $\boldsymbol{x}$.

6. При доказательстве леммы 1.3.1 мы не показали, почему не существует грани строки $\boldsymbol{x}[1..n]$ длины, отличной от $\boldsymbol{\beta}^j[n]$, $j = 1, 2, \ldots, k$. Завершите доказательство этой леммы.

7. Разработчики программного обеспечения располагают богатым арсеналом формальных методов доказательства корректности программ. В частности, при исследовании циклических структур различают такие их составляющие.

- ***Инвариант цикла*** — условное выражение, принимающее значение "истина" в начале или в конце выполнения цикла.
- ***Вариант цикла*** — выражение, которое принимает целые положительные значения и уменьшается при каждом выполнении цикла.

Найдите в двух циклах алгоритма 1.3.1 инвариант и вариант цикла.

8. Примените алгоритм 1.3.1 к каждой из следующих строк и вычислите для них массивы граней $\boldsymbol{\beta}[1..n]$.

а) $abcdefg$;

б) a^n;

в) $(ab)^n$;

г) $abaababaabaab$;

д) $(aba)^3a(aba)^2a(aba)^3$;

е) a^4ba^5b.

9. Оператор **if** в алгоритме 1.3.1 немного раздражает, поскольку в нем дублируется сравнение букв, которое уже выполнялось в операторе **while**. Путем введения дополнительного элемента $\boldsymbol{\beta}[0]$ в массиве граней найдите способ исключить этот оператор.

10. В упражнении 1.2.24 утверждалось, что более правильно "нормальную форму" называть "левой нормальной формой", там же введено понятие "правой нормальной формы". Аналогично, поскольку массив граней содержит значения длин наибольших граней всех *префиксов* строки $\boldsymbol{x}$, его также имеет смысл назвать "левый массив граней". Тогда ***правый массив граней*** будет содержать значения длин наибольших граней всех *суффиксов* строки $\boldsymbol{x}$. Напишите алгоритм вычисления правого массива граней с линейным временем выполнения.

Замечание. Это упражнение представляет не только академический интерес. Правый массив граней находит широкое применение, например, в известном строковом алгоритме Бойера–Мура, описанном в разделе 7.2.

11. Обозначим через $\boldsymbol{\rho}$ правый массив граней строки $\boldsymbol{x}$ длины n и через $\boldsymbol{\beta}_T$ — массив граней транспонированной строки

$$\boldsymbol{x}_T = \boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1].$$

Докажите, что для каждого $i \in 1..n$ выполняется $\boldsymbol{\rho}[i] = \boldsymbol{\beta}_T[n - i + 1]$.

12. На основе леммы 1.3.1 разработайте алгоритм, который определяет, является ли данная строка $\boldsymbol{y} = \boldsymbol{y}[1..n]$ целых неотрицательных чисел массивом граней какой-нибудь строки на некотором алфавите. Ваш алгоритм должен выполняться за время порядка $\Theta(n)$, докажите, что это так и есть. И, конечно, докажите корректность алгоритма.

Предупреждение. Это трудное упражнение, которое лучше рассматривать как исследовательскую задачу. Она тесно связана с материалом разделов 4.2 и 4.3 и ведет к двум опубликованным работам [90, 79]. Несмотря на то что уже сделано по этой теме, здесь есть еще обширное поле для исследований.

13. Обозначим через $\boldsymbol{x}'$ строку, полученную из заданной строки $\boldsymbol{x}$ путем замены букв, входящих в эту строку, другими буквами, также входящими в эту строку. (Например, если $\boldsymbol{x} = abcab$, то после замены $a \to c$, $b \to a$, $c \to b$ получим строку $\boldsymbol{x}' = cabca$.) Покажите, что массив граней строки $\boldsymbol{x}'$ совпадает с массивом граней строки $\boldsymbol{x}$. Этот результат можно интерпретировать следующим образом: в любом алфавите, содержащем более одной буквы, не существует строк, которые однозначно определялись бы своим массивом граней. (Более подробно этот вопрос рассмотрим в разделе 4.3.)

14. В этом разделе было высказано утверждение, что алгоритм 1.3.1 асимптотически оптимален. Сохранится ли справедливость этого утверждения, если предположить, что алгоритм 1.3.1 будет вычислять только $\boldsymbol{\beta}[n]$?

15. В обсуждении, ведущем к теореме 1.3.3, было сделано несколько туманное утверждение, что "количество уменьшений значения $b\ldots$ не может превосходить количества увеличений этого значения". Чтобы прояснить этот вопрос, рассмотрим функции $x(v) = v + 1$ и $y(v) = v - 1$, определенные на всем множестве действительных чисел. Обозначим через $\boldsymbol{s}$ строку, состоящую из j и k ($j, k \geqslant 0$) последовательных результатов вычисления функций x и y соответственно. Тогда $|\boldsymbol{s}| = j + k$ и $\boldsymbol{s}(v)$ равно результирующему значению этих вычислений. Например, для $\boldsymbol{s} = xyx$ $|\boldsymbol{s}| = 3$ и $\boldsymbol{s}(0) = 1$.

Докажите, что $\boldsymbol{s}(v) - v = j - k$, и покажите, как этот результат связан с теоремой 1.3.3.

16. Докажите более общий результат, подобный предыдущему упражнению, для случая функций $x(v) = v + r$ и $y(v) = v - s$, где r и s — положительные действительные числа.

17. Покажите, что если выполняются условия леммы периодичности и неравенство $p < q$, тогда $p \leqslant n/2$.

1.4 Строковые петли

Строковые петли часто преподносят неожиданности строковым алгоритмам, даже если они открыто не связаны с решаемой задачей. В этом разделе рассмотрены некоторые свойства строковых петель.

Как упоминалось в разделе 1.1, строковые петли обычно определяются через линейные строки. Будем использовать обозначение $C(\boldsymbol{x})$ для строковой петли, сформированной из линейной строки $\boldsymbol{x}$ путем сочленения ее самого левого элемента $\boldsymbol{x}[1]$ с самым правым элементом $\boldsymbol{x}[n]$. Таким образом строка $\boldsymbol{x}$ преобразуется в строковую последовательность, не имеющую ни самого левого, ни самого правого элементов. Например, если такое преобразование применить к линейной строке $\boldsymbol{f} = abaababa$, тогда получим петлю $C(\boldsymbol{f})$ вида

$$\begin{array}{ccccc} & & a & & \\ & a & & & b \\ b & & & & a \\ & a & & & a \\ & & b & & \end{array}$$

В алгоритмах, формирующих петли, часто $C(\boldsymbol{x})$ выражают через циклические сдвиги строки $\boldsymbol{x}$. Для данной строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ и произвольного целого $j \in \in 0..n-1$ j-й ***циклический сдвиг*** (rotation или cyclic shift) строки $\boldsymbol{x}$ определяет строку[13]

$$R_j(\boldsymbol{x}) = \boldsymbol{x}[j+1..n]\boldsymbol{x}[1..j].$$

Из этого определения следует, что $R_0(\boldsymbol{x}) = \boldsymbol{x}$.

Приведем восемь сдвигов строки $\boldsymbol{f} = abaababa$:

$$\begin{aligned} R_0(\boldsymbol{f}) &= abaababa, \\ R_1(\boldsymbol{f}) &= baababaa, \\ R_2(\boldsymbol{f}) &= aababaab, \\ &\vdots \\ R_7(\boldsymbol{f}) &= aabaabab. \end{aligned}$$

Отметим, что сдвиги имеют подстроки, которые не являются подстроками исходной строки. Например, здесь строка $aabaa$ не является подстрокой строки $\boldsymbol{f}$, однако является подстрокой сдвига $R_7(\boldsymbol{f})$ и, следовательно, петли $C(\boldsymbol{f})$. Поэтому в процессе формирования петли важно сохранить все возможные циклические

[13] Здесь и далее термином "циклический сдвиг" обозначается как операция циклического сдвига, так и строка, полученная в результате применения этой операции. Как правило, это не приводит к недоразумениям, так как из контекста всегда ясно, о чем идет речь. — *Примеч. ред.*

сдвиги исходной строки. На практике для этого достаточно сохранить строку $\boldsymbol{x}^2$, которая содержит все сдвиги $R_j(\boldsymbol{x})$ в виде своих подстрок.

Теперь можно более четко определить, что мы будем понимать под термином ***подстрока петли*** $C(\boldsymbol{x})$ — это подстрока любого сдвига строки $\boldsymbol{x}$ или, что равнозначно, подстрока (длины не более n) строки $\boldsymbol{x}^2$. Но поскольку петля не имеет ни самого левого, ни самого правого элементов, особые подстроки, введенные в разделе 1.2, — префикс, суффикс и грань — можно определить только по отношению к конкретным сдвигам, но не к петле в целом. Такое понятие, как период, которое определялось через грани, также нельзя применить к петле. Вместо этого следующие теоремы показывают, как посредством циклических сдвигов можно определить различные свойства петель $C(\boldsymbol{x})$, если порождающая линейная строка $\boldsymbol{x}$ не является кратной. Сначала докажем такую теорему.

Теорема 1.4.1. Пусть $\boldsymbol{x}$ — строковая последовательность длиной $n \geqslant 3$ на двоичном алфавите ($\alpha a = 2$).

а) если строка $\boldsymbol{x}$ является кратной, то таким же будет каждый циклический сдвиг этой строки;

б) если строка $\boldsymbol{x}$ не является кратной, тогда существует по крайней мере один циклический сдвиг этой строки, который будет периодическим.

Доказательство. Сначала докажем утверждение a). Из условия кратности строки $\boldsymbol{x}$ следует, что $\boldsymbol{x} = \boldsymbol{u}^r$, где $r > 1$ и существует такое целое $p < n$, что $n = pr$ и $\boldsymbol{u} = \boldsymbol{x}[1..p]$. Тогда

$$R_1(\boldsymbol{x}) = \boldsymbol{x}[2..n]\boldsymbol{x}[1] = (\boldsymbol{u}[2..p]\boldsymbol{u}[1])^r.$$

Отсюда по индукции получаем требуемое утверждение.

Для доказательства утверждения *б*) попробуем построить строку $\boldsymbol{x}$, такую, чтобы любой ее циклический сдвиг был примитивной строкой. В такой строке $\boldsymbol{x}[1] \neq \boldsymbol{x}[n]$, иначе строка $\boldsymbol{x}$ будет периодической. Без потери общности положим, что $\boldsymbol{x}[1] = a$ и $\boldsymbol{x}[n] = b$. Тогда должно выполняться равенство $\boldsymbol{x}[2] = b$, в противном случае $R_1(\boldsymbol{x})[1] = R_1(\boldsymbol{x})[n] = a$, и поэтому $R_1(\boldsymbol{x})$ будет периодической строкой. Подобные рассуждения ведут к заключению, что $\boldsymbol{x}[n-1] = a$. Но тогда уже при $n = 3$ получаем противоречие: $x[2] \neq \boldsymbol{x}[3-1]$. Отсюда следует, что при $n > 3$ строка $\boldsymbol{x}$ имеет грань ab, которая является периодической. ■

Как указано в разделе 3.3, можно построить некратную строку произвольной длины на алфавите из трех букв. Если мы возьмем одну из таких строк длиной $n-1$ и затем присоединим к ней одну букву, которой пока нет в этой строке, тогда, как показано в упражнении 1.4.3, будем иметь строку на алфавите из четырех букв, каждый сдвиг которой является примитивной строкой. Поэтому утверждение *б*) теоремы 1.4.1 нельзя обобщить на алфавиты размером 4 и выше. Но утверждение *а*) теоремы 1.4.1 можно обобщить, что сделано в упражнении 1.4.2.

Другой полезный результат связан с циклическими сдвигами, кратностью и периодичностью строк. Согласно этому результату, если строка $\boldsymbol{x}$ некратная, то все ее циклические сдвиги различны. Доказательство этого результата основано на лемме периодичности.

Теорема 1.4.2. Пусть $\boldsymbol{x}$ — произвольная строковая последовательность длины n с минимальным периодом p^*. Тогда для любого целого числа $j \in 1..n-1$ равенство $R_j(\boldsymbol{x}) = \boldsymbol{x}$ будет выполняться только в том случае, когда $\boldsymbol{x}$ является кратной строкой и p^* является делителем j.

Доказательство. Для доказательства достаточности утверждения теоремы нетрудно показать, что если выполняется равенство $\boldsymbol{x} = \boldsymbol{x}[1..p^*]^{r^*}$, где $r^* = n/p^*$ — целое число, тогда для $j = p^*, 2p^*, \ldots, (r^*-1)p^*$ справедливо $R_j(\boldsymbol{x}) = \boldsymbol{x}$.

Для доказательства необходимости предположим, что $R_j(\boldsymbol{x}) = \boldsymbol{x}$. Тогда $\boldsymbol{x}[i+j] = \boldsymbol{x}[i]$ для всех целых $i \in 1..n-j$, поэтому $\boldsymbol{x}$ имеет грань $\boldsymbol{x}[1..n-j]$ и период j. Теперь применим теорему 1.3.5 с параметризованными условиями $H(d, j, n-j, n)$, где $d = \gcd(j, n-j)$. Поскольку выполняется $j + (n-j) < n + d$, то на основании теоремы делаем заключение, что d является периодом строки $\boldsymbol{x}$, т.е.

$$\boldsymbol{x} = \boldsymbol{x}[1..n-j]\boldsymbol{x}[1..j] = \boldsymbol{x}[1..d]^{(n-j)/d}\boldsymbol{x}[1..d]^{j/d} = \boldsymbol{x}[1..d]^{n/d}.$$

Таким образом, строка $\boldsymbol{x}$ является кратной.

Чтобы показать, что p^* делит d и, следовательно, делит j, снова применим теорему 1.3.5 с условиями $H(d', p^*, d, n)$, где $d' = \gcd(p^*, d)$. Поскольку $p^* \leqslant d \leqslant n/2$, из теоремы следует, что $p^* + d < n + d'$, тогда d' является периодом строки $\boldsymbol{x}$. Это означает, что $d' = p^*$ и, следовательно, p^* делит d, что и требовалось доказать. ■

Теорема 1.4.2 удручает, поскольку из нее следует, что, за исключением случая кратных строк, для проверки равенства $C(\boldsymbol{x}) = C(\boldsymbol{y})$ необходимо проверить все циклические сдвиги исходных строк $\boldsymbol{x}$ и $\boldsymbol{y}$. Такая проблема возникает в компьютерной графике и вычислительной геометрии, где выпуклый многоугольник можно представить посредством двух различных строк: одна содержит последовательность длин сторон многоугольника при его обходе по часовой стрелке, другая — последовательность углов между смежными сторонами, также полученную при обходе многоугольника по часовой стрелке. Для решения задачи, будут ли два представленных многоугольника конгруэнтными, надо проверить, будут ли равны строки, представляющие эти многоугольники (будь то строки длин сторон или строки углов). Пусть строка $\boldsymbol{x}$ представляет один многоугольник, а строка $\boldsymbol{y}$ — другой. Поскольку многоугольники могут иметь различную ориентацию, первое значение $\boldsymbol{x}[1]$ строки $\boldsymbol{x}$ может не совпадать с первым значением $\boldsymbol{y}[1]$ строки $\boldsymbol{y}$.

Другими словами, даже если в действительности $C(\boldsymbol{x}) = C(\boldsymbol{y})$, то совсем не обязательно, что $\boldsymbol{x} = \boldsymbol{y}$. Пример такой ситуации показан на рис. 1.1.

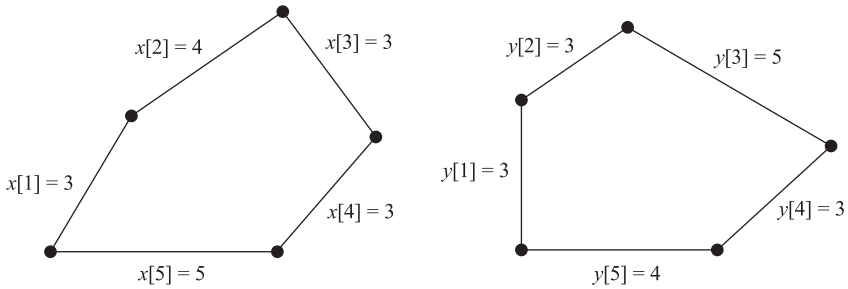


Рис. 1.1. Два конгруэнтных многоугольника

Естественным подходом к решению этой проблемы было бы введение "канонической формы" для строковых петель такой, чтобы она позволяла "напрямую" сравнивать различные петли. На практике такую роль могут выполнять циклические сдвиги исходной строки, которые можно вычислить достаточно эффективно. Если строки определены на упорядоченном алфавите, то решением проблемы может быть следующее определение.

Определение 1.4.3. Канонической формой строковой петли $C(\boldsymbol{x})$ на упорядоченном алфавите называется лексикографически наименьший циклический сдвиг $R_j(\boldsymbol{x})$ строки $\boldsymbol{x}$, т.е. $R_j(\boldsymbol{x}) \leqslant R_i(\boldsymbol{x})$ для всех $i \in 0..n-1$. ■

Каноническая форма строковых петель очень тесно связано еще с одним понятием.

Определение 1.4.4. Непустая строка $\boldsymbol{w}$ на упорядоченном алфавите называется **линдонским словом**, если $\boldsymbol{w} < R_j(\boldsymbol{w})$ для всех целых $j \in 1..n-1$. ■

Строка, состоящая из одной буквы, будет линдонским словом[14], поскольку, согласно теореме 1.4.2, такая строка не может быть кратной. Далее мы увидим, что строка ab является линдонским словом, строка ba — нет. Аналогично строка aab является линдонским словом, а строки aba и baa — нет. В общем случае, основываясь на следствии теоремы 1.4.2, которое заключается в том, что если строка $\boldsymbol{x}$ *некратная*, то все ее циклические сдвиги различны, можно сделать вывод: у некратной строки существует только один сдвиг, который будет линдонским словом. Таким образом, если строка $\boldsymbol{x}$ некратная, тогда существует линдонское

[14]Такое название строкам специального вида дано в честь Роджера Линдона (Roger C. Lyndon), который предложил особую факторизацию строк, которая позже получила название линдонской декомпозиции. Эта факторизация описана дальше в этом разделе. — *Примеч. ред.*

слово $\boldsymbol{w} = R_j(\boldsymbol{x})$, которое в точности совпадает с канонической формой петли $C(\boldsymbol{x})$. Это означает, что множество всех линдонских слов на данном упорядоченном алфавите A совпадает с множеством всех канонических форм, которые не являются кратными.

Далее докажем три леммы (взяты из работы [80]), которые описывают основные свойства линдонских слов. Эти леммы необходимы для теоремы о линдонской декомпозиции.

Лемма 1.4.5. Каждое линдонское слово примитивно.

Доказательство. Необходимо показать, что не существует линдонских слов, которые имели бы непустые грани. Предположим противное — существует линдонское слово $\boldsymbol{w}$, такое, что $\boldsymbol{w} = \boldsymbol{uv}' = \boldsymbol{v}''\boldsymbol{u}$, где $\boldsymbol{u}$, $\boldsymbol{v}'$ и $\boldsymbol{v}''$ — непустые строки. Очевидно, что $|\boldsymbol{v}'| = |\boldsymbol{v}''|$. Отсюда и из определения линдонского слова следует, что

$$\boldsymbol{uv}' = \boldsymbol{w} < \boldsymbol{uv}''\text{и } \boldsymbol{v}''\boldsymbol{u} = \boldsymbol{w} < \boldsymbol{v}'\boldsymbol{u}.$$

Итак, получаем два неравенства: $\boldsymbol{v}' < \boldsymbol{v}''$ и $\boldsymbol{v}'' < \boldsymbol{v}'$ — т.е. пришли к противоречию. ■

Лемма 1.4.6. Строковая последовательность $\boldsymbol{w}$ будет линдонским словом только тогда, когда для любого непустого префикса $\boldsymbol{v}$ строки $\boldsymbol{w}$ выполняется неравенство $\boldsymbol{w} < \boldsymbol{v}$.

Доказательство. Для доказательства достаточности условий леммы заметим, что если $\boldsymbol{w} = \boldsymbol{uv} < \boldsymbol{v}$ для непустых строк $\boldsymbol{u}$ и $\boldsymbol{v}$, тогда $\boldsymbol{uv} < \boldsymbol{vu}$.

Для доказательства необходимости предположим, что $\boldsymbol{w}$ — линдонское слово и $\boldsymbol{w} = \boldsymbol{uv}$, где $\boldsymbol{u}$ и $\boldsymbol{v}$ — непустые строки. Тогда $\boldsymbol{uv} < \boldsymbol{vu}$. Очевидно, что $\boldsymbol{uv} \neq \boldsymbol{v}$, поскольку $|\boldsymbol{uv}| > |\boldsymbol{v}|$. Предположим, что $\boldsymbol{uv} > \boldsymbol{v}$. Возможны два взаимоисключающих случая: $\boldsymbol{v}$ является префиксом строки $\boldsymbol{uv}$ и $\boldsymbol{v}$ не является префиксом этой же строки. Но первый случай невозможен, поскольку тогда $\boldsymbol{v}$ является гранью строки $\boldsymbol{w}$, что противоречит лемме 1.4.5. Второй случай тоже невозможен, так как тогда $\boldsymbol{w}[1..|\boldsymbol{v}|] > \boldsymbol{v}$, откуда следует неравенство $\boldsymbol{w} > \boldsymbol{uv}$, противоречащее свойствам линдонских слов. Таким образом, мы пришли к противоречию с предположением $\boldsymbol{uv} > \boldsymbol{v}$. Следовательно, $\boldsymbol{uv} < \boldsymbol{v}$, что и требовалось доказать. ■

Лемма 1.4.7. Пусть $\boldsymbol{w}_1$ и $\boldsymbol{w}_2$ являются линдонскими словами. Тогда строка $\boldsymbol{w}_1\boldsymbol{w}_2$ будет линдовским словом только в том случае, когда $\boldsymbol{w}_1 < \boldsymbol{w}_2$.

Доказательство. Необходимость условий леммы следует непосредственно из леммы 1.4.6: если $\boldsymbol{w}_1\boldsymbol{w}_2$ является линдонским словом, тогда $\boldsymbol{w}_1 < \boldsymbol{w}_1\boldsymbol{w}_2 < \boldsymbol{w}_2$.

Для доказательства достаточности обозначим через $\boldsymbol{v}$ непустой собственный префикс строки $\boldsymbol{w}_1\boldsymbol{w}_2$ и предположим, что $\boldsymbol{w}_1 < \boldsymbol{w}_2$. Возможна одна из следующих ситуаций.

- $v = w_1' w_2$, где w_1' — непустой собственный префикс строки $\boldsymbol{w}_1$. Поскольку $\boldsymbol{w}_1$ линдонским словом, то на основании леммы 1.4.6 имеем $w_1 < w_1'$. Так как $|w_1| > |w_1'|$, то отсюда получаем $w_1 w_2 < w_1' w_2 = v$.
- $\boldsymbol{v} = \boldsymbol{w}_2$. Если $\boldsymbol{w}_1$ не является префиксом $\boldsymbol{w}_2$, тогда $\boldsymbol{w}_1 \boldsymbol{w}_2 < \boldsymbol{w}_2 = \boldsymbol{v}$, поскольку $\boldsymbol{w}_1 < \boldsymbol{w}_2$. Если же $\boldsymbol{w}_1$ является префиксом $\boldsymbol{w}_2$, тогда можно записать , где w_2'' — собственный суффикс строки $\boldsymbol{w}_2$. Поскольку $\boldsymbol{w}_2$ является линдонским словом, можно применить лемму 1.4.6, вследствие которой имеем $w_2 < w_2''$. Следовательно, $w_1 w_2 < w_1 w_2'' = w_2 = v$.
- $\boldsymbol{v} = \boldsymbol{w}_2'$, где w_2' — собственный суффикс строки $\boldsymbol{w}_2$. Поскольку $\boldsymbol{w}_2$ является линдонским словом, из леммы 1.4.6 следует, что $w_1 < w_2'$. Так как $\boldsymbol{w}_2$ не является префиксом w_2' и имеется условие $\boldsymbol{w}_1 < \boldsymbol{w}_2$, отсюда следует, что $\boldsymbol{w}_1$ не может быть префиксом w_2'. Поэтому $w_1 w_2 < w_2 < w_2' = v$, что и требовалось.

В каждом из рассмотренных случаев получаем, что $\boldsymbol{w}_1 \boldsymbol{w}_2 < \boldsymbol{v}$, где $\boldsymbol{v}$ — суффикс строки $\boldsymbol{w}_1 \boldsymbol{w}_2$. На основании леммы 1.4.6 заключаем, что $\boldsymbol{w}_1 \boldsymbol{w}_2$ является линдонским словом. На этом завершается доказательство леммы. ■

Чтобы использовать полученные результаты, нам необходимо еще одно определение.

Определение 1.4.8. Пусть $\boldsymbol{x}$ — непустая строковая последовательность на упорядоченном алфавите A. Декомпозиция $\boldsymbol{x} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_k$ называется **линдонской**, если все $\boldsymbol{w}_i$ $(i = 1, 2, \ldots, k)$ являются линдонскими словами, которые удовлетворяют неравенствам $\boldsymbol{w}_1 \geqslant \boldsymbol{w}_2 \geqslant \cdots \geqslant \boldsymbol{w}_k$. ■

Теперь можно сформулировать и доказать одну замечательную теорему [48].

Теорема 1.4.9. Для любой непустой строковой последовательности $\boldsymbol{x}$ на упорядоченном алфавите существует линдонская декомпозиция, причем единственная.

Доказательство. Сначала покажем, что существует по крайней мере одна линдонская декомпозиция строки $\boldsymbol{x}$. Начнем с первоначальной декомпозиции

$$\boldsymbol{x} = \boldsymbol{w}_1^{(0)} \boldsymbol{w}_2^{(0)} \ldots \boldsymbol{w}_n^{(0)},$$

где факторы $w_i^{(0)} = x\,[i]$, $i \in 1..n$ представляют отдельные буквы. Здесь все факторы $w_i^{(0)}$ являются линдонскими словами, но сама декомпозиция не обязательно будет линдонской, поскольку возможны неравенства $w_j^{(0)} < w_{j+1}^{(0)}$ для некоторых номеров j. Для таких номеров сочленение строк $w_j^{(0)} w_{j+1}^{(0)}$ будет линдонским словом, что следует из леммы 1.4.7. Объединяя такие факторы, получим новую декомпозицию

$$\boldsymbol{x} = \boldsymbol{w}_1^{(1)} \boldsymbol{w}_2^{(1)} \cdots \boldsymbol{w}_{n_1}^{(1)},$$

где каждый фактор $\boldsymbol{w}_i^{(1)}$ является линдонским словом и $n_1 \leqslant n$. Если $n_1 < n$ и есть последовательные факторы, для которых выполняется неравенство $\boldsymbol{w}_j^{(1)} < \boldsymbol{w}_{j+1}^{(1)}$, продолжаем объединение таких факторов и получаем новую декомпозицию. Через конечное число таких шагов получим линдонскую декомпозицию

$$\boldsymbol{x} = \boldsymbol{w}_1^{(s)} \boldsymbol{w}_2^{(s)} \cdots \boldsymbol{w}_{n_s}^{(s)},$$

где будут выполнены условия $\boldsymbol{w}_1^{(s)} \geqslant \boldsymbol{w}_2^{(s)} \geqslant \cdots \geqslant \boldsymbol{w}_{n_s}^{(s)}$.

Теперь предположим, что линдонская декомпозиция не единственная, т.е. существуют две различные декомпозиции

$$\boldsymbol{x} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_k \text{ и } \boldsymbol{x} = \boldsymbol{w}'_1 \boldsymbol{w}'_2 \cdots \boldsymbol{w}'_k.$$

Без потери общности можно предположить, что существует минимальное положительное целое число $i < \min\{k, k'\}$, такое, что для некоторой непустой подстроки $\boldsymbol{v}$ и для всех $j \in 1..i-1$ выполняется равенство $w'_j = w_j$. Тогда существует единственное целое число $r \geqslant 1$ и непустая строка $\boldsymbol{v}^*$, такая, что

$$\boldsymbol{w}_i \boldsymbol{w}_{i+1} \cdots \boldsymbol{w}_{i+r-1} \boldsymbol{v}^* = \boldsymbol{w}_i \boldsymbol{v} \text{ и } |\boldsymbol{w}_i \boldsymbol{w}_{i+1} \cdots \boldsymbol{w}_{i+r}| \geqslant |\boldsymbol{w}_i \boldsymbol{v}|.$$

Обозначим $\boldsymbol{w}^* = \boldsymbol{w}_i \boldsymbol{w}_{i+1} \cdots \boldsymbol{w}_{i+r-1}$, так что $\boldsymbol{w}'_i = \boldsymbol{w}^* \boldsymbol{v}^*$. Поскольку w'_i является линдонским словом, из леммы 1.4.6 следует, что $w'_i < v^*$. Строка $\boldsymbol{w}^*$ — собственный префикс w'_i, поэтому $\boldsymbol{w}^* < \boldsymbol{v}^*$.

Далее, поскольку $\boldsymbol{v}^*$ является префиксом $\boldsymbol{w}_{i+r}$, можно записать $\boldsymbol{v}^* \leqslant \boldsymbol{w}_{i+r}$. Так как $\boldsymbol{w}_i$ — префикс $\boldsymbol{w}^*$, то $\boldsymbol{w}_i \leqslant \boldsymbol{w}^*$. Поэтому

$$\boldsymbol{w}_i \leqslant \boldsymbol{w}^* < \boldsymbol{v}^* \leqslant \boldsymbol{w}_{i+r}.$$

Последние неравенства противоречат предположению, что $\boldsymbol{x} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_k$ является линдонской декомпозицией. ■

В конце этого раздела вернемся к сделанному выше замечанию о том, что строка $\boldsymbol{x}$ не должна быть кратной и что каноническая форма петли $C(\boldsymbol{x})$ совпадает с циклическим сдвигом строки $\boldsymbol{x}$ и этот сдвиг является линдонским словом. Предположим, что имеется алгоритм (например, один из описанных в разделе 6.1), вычисляющий линдонскую декомпозицию для произвольной строки $\boldsymbol{x}$. Запишем линдонскую декомпозицию в более сжатой форме

$$\boldsymbol{x} = \boldsymbol{w}_1^{q_1} \boldsymbol{w}_2^{q_2} \cdots \boldsymbol{w}_k^{q_k}, \tag{1.11}$$

где $\boldsymbol{w}_1 > \boldsymbol{w}_2 > \cdots > \boldsymbol{w}_k$ и q_i — целые положительные числа, $i = 1, 2, \ldots, k$. Как показано в упражнении 1.4.14, с помощью нашего алгоритма можно сразу получить все значения q_i.

Кратко рассмотрим задачу определения в строке $\boldsymbol{x}$ позиции $\mathrm{MSP}(\boldsymbol{x}) = i^*$ (эта позиция называется ***минимальной начальной точкой***[15]), такой, что $R_{i^*-1}(\boldsymbol{x})$

[15] Отсюда название функции MSP — это сокращение от англ. Minimum Starting Point, что и переводится как *минимальная начальная точка. — Примеч. ред.*

будет минимумом по всем циклическим сдвигам строки $\boldsymbol{x}$. Как мы увидим дальше, позиция i^* определяется однозначно и $R_{i^*-1}(\boldsymbol{x})$ является линдонским словом только в том случае, когда строка $\boldsymbol{x}$ не будет кратной. Сначала рассмотрим несколько специальных случаев.

- Если $k = 1$, тогда $\mathrm{MSP}(\boldsymbol{x}) = 1$.
- Если $k = 2$, тогда $\mathrm{MSP}(\boldsymbol{x}) = q_1|\boldsymbol{w}_1| + 1$.
- Если $\boldsymbol{w}_k^{q_k}$ не является префиксом $\boldsymbol{w}_{k-1}^{q_{k-1}}$, тогда $\mathrm{MSP}(\boldsymbol{x}) = n - q_k|\boldsymbol{w}_k| + 1$.
- Если строка $\boldsymbol{x}$ кратная, т.е. $\boldsymbol{x} = \boldsymbol{u}^m$, $m > 1$, тогда $\mathrm{MSP}(\boldsymbol{x}) = \mathrm{MSP}(\boldsymbol{u})$.

Теперь рассмотрим применение алгоритма к строке $\boldsymbol{x}^2$. Если строка $\boldsymbol{x}$ не кратная, алгоритм создаст декомпозицию $\boldsymbol{v}_1 R_{i^*-1}(\boldsymbol{x})\boldsymbol{v}_2$, где $\boldsymbol{v}_1$ и $\boldsymbol{v}_2$ — декомпозиции строк $\boldsymbol{x}[1..i^*-1]$ и $\boldsymbol{x}[i^*..n]$ соответственно и $R_{i^*-1}(\boldsymbol{x})$ является линдонским словом. В этом случае для того, чтобы удостовериться в том, что $\mathrm{MSP}(\boldsymbol{x}) = |\boldsymbol{v}_1|+1$, необходимо просмотреть только линдонское слово длины n. В случае, когда $\boldsymbol{x} = \boldsymbol{u}^m$ (m целое и больше 1), необходимо просмотреть блок одинаковых линдонских слов $w_i^{q_i}$, таких, что $q_i|\boldsymbol{w}_i| = n$, и снова получим результат $\mathrm{MSP}(\boldsymbol{x}) = |\boldsymbol{v}_1| + + 1$. Таким образом, любой алгоритм для вычисления линдонской декомпозиции можно (с минимальными изменениями) использовать для вычисления канонических форм. Однако, как показано в упражнении 1.4.16, основываясь только на линдонской декомпозиции, нельзя непосредственно определить, какой фактор в декомпозиции определяет $\mathrm{MSP}(\boldsymbol{x})$. Мы вернемся к этому вопросу в разделе 6.2.

В разделе 6.2 также показано, что линдонская декомпозиция является основой для решения многих задач, в частности, эффективного нахождения лексикографически наименьшего и лексикографически наибольшего суффиксов любого префикса строки.

Упражнения 1.4

1. Покажите, что для сохранения всей информации о петле $C(\boldsymbol{x})$ достаточно сохранить строку $\boldsymbol{x}^{(2n-1)/n}$.

2. По индукции докажите, что для произвольной непустой строки $\boldsymbol{x} = \boldsymbol{u}^k$ и любого целого $j \in 0..|\boldsymbol{u}| - 1$ выполняется равенство $R_j(\boldsymbol{x}) = (R_j(\boldsymbol{x}))^k$. (Запишите в явном виде "индукцию", упоминавшуюся при доказательстве утверждения a) теоремы 1.4.1. Этим вы докажите, что утверждение a) этой теоремы справедливо для любого алфавита.)

3. Пусть $\boldsymbol{x}$ — некратная строка длиной $n-1$ на алфавите $A = \{a, b, c\}$. Докажите утверждение, сделанное в этом разделе, что любой циклический сдвиг строки $\boldsymbol{x}$ будет примитивной строкой.

4. Внимательный читатель, конечно, обратил внимание на то, что при рассмотрении вопроса о существовании примитивных циклических сдвигов

у кратных и некратных строк мы не исследовали случай, когда исходная строка примитивная на алфавите, состоящем из трех букв. Причина такого упущения станет ясной, если обратиться к следующему результату, полученному путем машинного моделирования. Оказывается, не существует примитивных строк длиной $n = 5$, 7, 9, 10, 14 и 17, у которых все циклические сдвиги примитивны. Для всех других значений $n \in 1..19$ такие строки существуют. Таким образом, имеем интересную исследовательскую задачу, поставленную Джейми Симпсоном (Jamie Simpson): определить значения n, для которых существуют примитивные строки на алфавите из трех букв, у которых все циклические сдвиги примитивны.

Дополнительная информация. После длительных исследований этой задачи, ее автор пришел к выводу, что для всех n, больших 17, такие примитивные строки существуют, но доказать это очень трудно. Однако недавно Джеймс Карри (James Currie) [74] нашел другой подход к решению этой задачи.

5. Без привлечения леммы периодичности покажите, что если для двух непустых строк $\boldsymbol{x}$ и $\boldsymbol{y}$ выполняется равенство $\boldsymbol{xy} = \boldsymbol{yx}$, тогда существуют положительные целые числа k_1 и k_2 и строка $\boldsymbol{u}$, такие, что и $y = u^{k_2}$. На этом строится альтернативное доказательство утверждения (см. теорему 1.4.2), что если $\boldsymbol{x} = R_j(\boldsymbol{x})$ для некоторого $j \in 1..n-1$, тогда строка $\boldsymbol{x}$ кратная.

6. Вдумчивый читатель заметит, что в доказательстве леммы 1.4.6 присутствует "потерянная лемма", т.е. доказательство этой леммы можно сделать более простым и понятным, если ранее доказать следующее утверждение.

> Пусть $\boldsymbol{x}$ и $\boldsymbol{y}$ — строки на упорядоченном алфавите A. Если $\boldsymbol{x} > \boldsymbol{y}$ и $\boldsymbol{y}$ не является префиксом строки $\boldsymbol{x}$, тогда для любых строк $\boldsymbol{z}$, $\boldsymbol{z}' \in A^*$ справедливо неравенство $\boldsymbol{xz} < \boldsymbol{yz}'$.

Докажите эту "потерянную лемму".

7. Подобно предыдущему упражнению, в доказательстве леммы 1.4.7 также можно выделить два утверждения, использование которых упростит доказательство леммы. Вот эти утверждения.

- Если $\boldsymbol{x} < \boldsymbol{y}$ и $\boldsymbol{x}$ не является префиксом строки $\boldsymbol{y}$, тогда для любых строк $\boldsymbol{z}, \boldsymbol{z}' \in A^*$ справедливо неравенство $\boldsymbol{xz} < \boldsymbol{yz}'$.
- Если $\boldsymbol{x} < \boldsymbol{y}$ и $|\boldsymbol{x}| \geqslant |\boldsymbol{y}|$, тогда для любых строк $\boldsymbol{z}, \boldsymbol{z}' \in A^*$ справедливо неравенство $\boldsymbol{xz} < \boldsymbol{yz}'$.

Докажите сформулированные утверждения.

8. В продолжение предыдущего упражнения докажите, что если строки $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x}'$, $\boldsymbol{y}' \in A^*$ такие, что $\boldsymbol{xy} < \boldsymbol{x}'\boldsymbol{y}'$ и строки $\boldsymbol{x}$ и $\boldsymbol{x}'$ не являются префиксами друг друга, тогда выполняется неравенство $\boldsymbol{x} < \boldsymbol{x}'$.

9. Первая часть доказательства теоремы 1.4.9 предлагает схему алгоритма для определения линдонской декомпозиции. Разработайте такой алгоритм и определите верхнюю границу его асимптотической сложности.

10. Найдите линдонскую декомпозицию для следующих строк.

а) $xyzabc$;

б) $abcxyz$;

в) $abaababa$;

г) $(baa)^n$;

д) $abacabadabacabac$;

е) $abacabadabacabad$;

ж) $abacabadabacabab$;

з) $abacabadabacabaa$.

11. Для каждой строки, перечисленной в предыдущем упражнении, найдите каноническую форму для соответствующих петель.

12. Приведите примеры

а) примитивной строки, которая содержит кратную составляющую;

б) некратной строки, которая не является примитивной.

13. Назовем строковую петлю $C(\boldsymbol{x})$ ***некратной***, если любой циклический сдвиг строки $\boldsymbol{x}$ является некратной строкой. Докажите следующие два утверждения.

- Каноническая форма петли $C(\boldsymbol{x})$ будет примитивной строкой только тогда, когда строка $\boldsymbol{x}$ не является кратной.
- Петля $C(\boldsymbol{x})$ будет некратной только тогда, когда любой циклический сдвиг строки $\boldsymbol{x}$ будет примитивной строкой.

14. Предположим, что алгоритм построения линдонской декомпозиции выполняется над строкой $\boldsymbol{x}$ слева направо, и пусть на некотором этапе алгоритма получено следующее представление строки $\boldsymbol{x}$:

$$x = w_1^{q_1} w_2^{q_2} \ldots w_j^{q_j} u^m \lambda \ldots$$

для некоторых целых $j \geqslant 0$, $m \geqslant 2$ и некоторой буквы λ. Опишите дальнейший процесс построения линдонской декомпозиции для следующих случаев.

а) $\lambda < \boldsymbol{u}[1]$.

б) $\lambda = \boldsymbol{u}[1]$.

в) $\lambda > \boldsymbol{u}[1]$.

15. Докажите правильность выводов для $\mathrm{MSP}(\boldsymbol{x})$ в четырех "специальных случаях", приведенных в конце этого раздела.

16. "Соблазнительно" предположить (автор однажды "купился" на это), что минимальную начальную точку можно вычислить как $\mathrm{MSP}(\boldsymbol{x}) = i_k$, где i_j — начальная точка строки $w_j^{q_j}$ в линдонской декомпозиции (1.11), $j = 1, 2, \ldots, k$. Приведите пример строки, где $\mathrm{MSP}(\boldsymbol{x}) = i_{k-1}$ или даже $\mathrm{MSP}(\boldsymbol{x}) = i_{k-2}$.

ГЛАВА 2

Паттерны? Что такое паттерны?

> Я еще не настолько растерял свои познания в лексикографии, чтобы забыть, что слова — это дочери земли, а вещи — сыновья Небес.
>
> — Сэмюэль Джонсон (1709–1784).
> Предисловие к словарю

В главе 1, посвященной свойствам строковых последовательностей, только раз упоминалось о паттернах. Пришло время разобраться, что же это такое. В этой главе сделан обзор паттернов, которые будут вычисляться алгоритмами, описанными в частях II–IV книги, и представлено компактное определение основных задач, решаемых этими алгоритмами. Соответственно, паттерны раздела 2.1 рассмотрены в части II, паттерны раздела 2.2 — в части III и паттерны раздела 2.3 — в части IV.

В этой главе, как и во всей книге, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ обозначает строковую последовательность длиной n.

2.1 Внутренние паттерны

Как упоминалось в предисловии, внутренние паттерны (intrinsic patterns) в определенном смысле являются "собственным" или "природным" свойством строковых последовательностей — не каждая строка содержит частный паттерн (specific pattern), например подстроку bb, или характеристический паттерн (generic pattern), такой как кратные строки. С другой стороны, внутренние паттерны можно найти

во всех строках, и они также в определенной степени характеризуют строковые последовательности. Так, "нормальная форма" из раздела 1.2 — это паттерн, который можно рассматривать как стандартное описание любых строк, которое не зависит от других возможных описаний строк. Интересно отметить, что существует много полезных и полностью различных таких "стандартных описаний".

Нормальная форма определена в разделе 1.2, она вычисляется путем двойной обработки массива граней, как показано в разделе 1.3. Сформулируем задачу вычисления нормальной формы как задачу нахождения периодических структур в строках.

Задача 2.1 (Вычисление нормальной формы). Выразить строковую последовательность $\boldsymbol{x}$ в форме $\boldsymbol{u}^r$, где $\boldsymbol{u}$ — префикс строки $\boldsymbol{x}$, выбранный так, чтобы максимизировать значение r (раздел 1.3). ■

Хотя массивы граней не рассматриваются как паттерны, по сути они являются своеобразными внутренними паттернами соответствующих строковых последовательностей. Основой для такой интерпретации массивов граней служит теорема 1.3.4, которая показывает роль этих массивов в вычислении граней, периодов, порядков и нормальных форм.

Задача 2.2 (Вычисление массива граней). Вычислить наибольшие грани каждого префикса строки $\boldsymbol{x}$ и представить их в виде массива (строки) $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ (раздел 1.3). ■

В главе 5 мы изучим два вида деревьев: дерево граней и дерево суффиксов.[1] Дерево граней — это структура, которая уже содержится в скрытом виде в массиве граней, как показано ниже. Смысл сделать эту структуру явной заключается в том, чтобы помочь решить проблему "оболочек", рассмотренную в разделе 2.3. В главе 13 показано, что каждая строка имеет внутренний шаблон, так называемый *массив оболочек*, который является структурным "образом" массива граней. Дерево граней имеет в точности $n+1$ узлов, помеченных различными целыми числами от 0 до n. Отношения наследования (родители-сыновья) на таком дереве определяются просто: узел i является сыном узла-родителя $\boldsymbol{\beta}[i]$. В упражнении 1.4.3 предложено доказать, что структура, получаемая на основе такого отношения наследования, действительно является деревом. В разделе 5.1 дан пример построения дерева граней.

Задача 2.3 (Вычисление дерева граней). Представить массив граней строковой последовательности $\boldsymbol{x}$ в виде корневого дерева (раздел 5.1). ■

[1] Деревом называется связный неориентированный граф без циклов и без петель, имеющий один узел (называется корнем дерева), из которого выходят ребра к другим узлам. Дерево хотя и является неориентированным графом, но его узлы подчиняются отношениям наследования, и поэтому дерево имеет упорядоченную структуру. Конечные узлы дерева называются листьями. — *Примеч. ред.*

Три сформулированные выше задачи рассматриваются уже в этом разделе, поскольку они тесно связаны между собой. Но следующая задача значительно отличается от них и дает новый взгляд на внутреннюю структуру строковых последовательностей. Чтобы подойти к понятию дерева суффиксов, рассмотрим сначала общую структуру данных, которая называется *синтаксическое дерево* [136].[2]

Пусть $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ — множество попарно различных строк. Тогда ***синтаксическое дерево*** на множестве X — это дерево поиска, имеющее в точности $m+1$ конечных узлов: по одному для каждой строки $\boldsymbol{x}_i$ $(i = 1, 2, \ldots, m)$ плюс один для пустой строки ε . Ребра синтаксического дерева помечены буквами, которые содержатся в строках из множества X, и специальным "сигнальным" символом \$, обозначающим конец строки. Строка $\boldsymbol{x}_i$ раскладывается на отдельные буквы — от корня дерева до конечного узла, которым оканчивается ребро, помеченное символом \$. В общем случае с *каждым* нисходящим путем от узла N_1 до узла N_2, например, ассоциируется некая строка $\boldsymbol{u}$. Чтобы избежать излишней сложности выражений и вместе с тем возможных терминологических осложнений, будем говорить, что узел N_2 — это строка $\boldsymbol{u}$, и если N_1 является корнем дерева, тогда строка $\boldsymbol{u}$ состоит из букв, которыми помечен путь от узла N_1 до узла N_2. Заметим, что в этом случае строка $\boldsymbol{u}$ является префиксом хотя бы одной какой-нибудь строки $\boldsymbol{x}_i\$$. Если же $N_1 = N_2$, тогда $\boldsymbol{u} = \varepsilon$. В частности, корень дерева является пустой строкой.

Использование символа \$ обусловлено тем, что $m+1$ конечных узлов являются листьями дерева и, как показано на рис. 2.1, *а* для множества $X = \{ab, abc\}$, без символа \$ невозможно было бы выделить узел 1, отображающий тот факт, что строка ab является префиксом строки abc. На этом рисунке можно заметить еще одно характерное свойство такой структуры данных — общие префиксы (такие, как ab или ε) на синтаксическом дереве отображаются только один раз. Еще раз подчеркнем, что каждый конечный узел соответствует строке, отличной от других.

На рис. 2.1, *б* показано ***компактное*** синтаксическое дерево (или синтаксическое дерево $\boldsymbol{Patricia}$[3]) [185], которое получено из синтаксического дерева путем исключения узлов степени 2, которые имеют родителя и только одного сына, т.е. корень дерева в число исключаемых узлов не входит.[4] Таким образом,

[2] В оригинале такая структура называется $trie$, это слово получено из букв, стоящих в середине слова re$trie$val (поиск, выборка, возврат). Устоявшегося термина для этой структуры в русской литературе пока нет. (О терминологическом разбросе можно судить по двум русским переводам книги [136], на которую ссылается автор.) На наш взгляд термин *синтаксическое дерево* наилучшим образом определяет это понятие. — *Примеч. ред.*

[3] Термин *Patricia trie* происходит от названия алгоритма Patricia — Practical Algorithm To Retrieve Information Coded In Alphanumeric, применяемый для построения специальных деревьев поиска и извлечения с его помощью информации на основе ключей [185]. — *Примеч. ред.*

[4] Степенью узла называется количество ребер, инцидентных с данным узлом, т.е. количество ребер, входящих и исходящих из данного узла. — *Примеч. ред.*

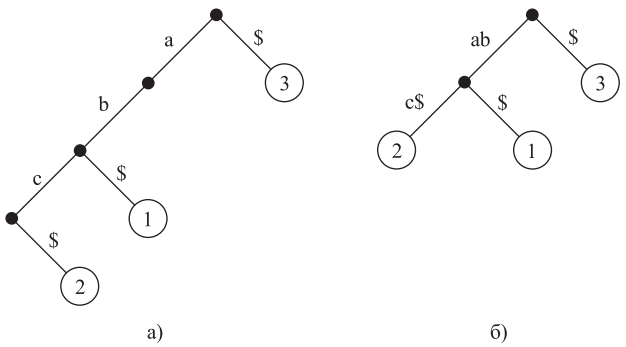


Рис. 2.1. Синтаксическое дерево для множества $X = \{ab, abc\}$

в компактном синтаксическом дереве будем иметь ребра, помеченные не отдельными буквами, а подстроками, как показано на рис. 2.1, *б*.

На каждом рис. 2.1 целыми числами помечены конечные узлы, соответствующие строкам $\boldsymbol{x}_i\$$, $i \leqslant n$. "Обычное" дерево, соответствующее данной строке $\boldsymbol{u}$, определяется, начиная от корня синтаксического дерева и далее через ребра, метки которых начинаются с очередной буквы строки $\boldsymbol{u}$. Если возникнет одна из следующих ситуаций: из данного узла не выходят ребра, метки которых начинаются с очередной буквы строки $\boldsymbol{u}$, либо достигнут неконечный узел, который не является префиксом $\boldsymbol{u}$, или конечный узел, который не является строкой $\boldsymbol{u}\$$, то во всех этих случаях $\boldsymbol{u} \notin X$. Во всех других возможных случаях $\boldsymbol{u} \in X$. В упражнении 2.1.3 предлагается доказать, что для любого узла $T_{\boldsymbol{x}}$ у всех ребер, выходящих из него, метки начинаются с различных букв.

Отметим, что синтаксическое дерево можно использовать для проверки не только того, что данная строка принадлежит множеству X, но и того, будет ли какой-нибудь ее префикс элементом этого множества. Однако следует помнить, что проверка префикса по компактному синтаксическому дереву несколько сложнее, чем по "обычному" синтаксическому дереву, где префикс $\boldsymbol{u}$ обязательно будет узлом дерева, тогда как в компактном дереве это не обязательно (например, префиксы a и abc на рис. 2.1, *б*). На компактном синтаксическом дереве конец префикса может быть *на ребре*, а не только в узле. Поэтому для решения подобных задач на компактном синтаксическом дереве (а также, как увидим в разделе 5.2.1, на дереве суффиксов) требуются дополнительные возможности для просмотра меток на ребрах, а не только для просмотра узлов.

Дерево суффиксов (suffix tree) строки $\boldsymbol{x}$ длиной n (иногда называемое ***деревом подслов*** (subword tree) [12]), если говорить кратко, — это компактное синтаксическое дерево для множества X всех суффиксов строки $\boldsymbol{x}$ (включая пустой суф-

фикс ε). Будем обозначать дерево суффиксов строки $\boldsymbol{x}$ как $T_{\boldsymbol{x}}$. Отметим, что $T_{\boldsymbol{x}}$ имеет в точности $n+1$ конечных узлов и, как показано в упражнении 2.1.4, не более n внутренних узлов, которые называются ***узлами ветвления*** (branch nodes). Таким образом, дерево суффиксов $T_{\boldsymbol{x}}$ имеет не более $2n+1$ узлов и не более $2n$ ребер. Память, необходимую для хранения метки каждого узла, можно уменьшить, если вместо самой метки хранить два целых числа, указывающих позиции (начало и конец) этой метки-подстроки в строке $\boldsymbol{x}$. (Но как указывалось в обсуждении 1.2, такая замена (подстроки на ее позиции в строке) предполагает, что получить доступ к подстроке на основе значений ее позиций можно за фиксированное время. Это, в свою очередь, предполагает, что исходная строка хранится в виде массива.) Таким образом, дерево суффиксов является "подходящей" структурой данных в том смысле, что для хранения требует память объемом порядка $\Theta(n)$ (см. также далее обсуждение 2.1). На рис. 2.2 показано дерево суффиксов $T_{\boldsymbol{g}}$ для строки $\boldsymbol{g}$

$$\begin{array}{c} 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\ \boldsymbol{g} = a\ b\ a\ a\ b\ a\ a\ b \end{array}$$

Отметим, что номера конечных узлов определяют позиции в исходной строке, где начинается данный суффикс. Узел с номером $n+1$ соответствует пустому суффиксу.

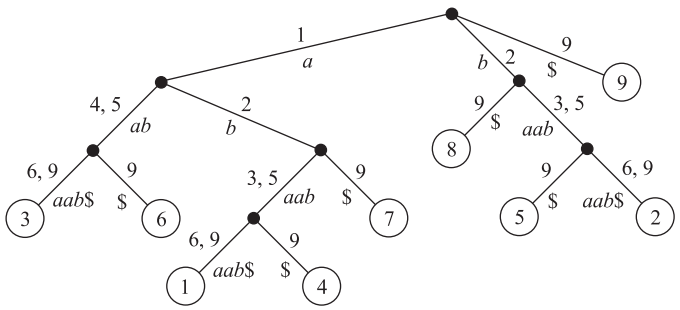


Рис. 2.2. Дерево суффиксов

Дерево суффиксов, введенное Винером [230], является очень важным внутренним паттерном. Для данной строки $\boldsymbol{x}$ дерево суффиксов $T_{\boldsymbol{x}}$ можно использовать для непосредственного определения того, является ли данная строка $\boldsymbol{u}$ суффиксом строки $\boldsymbol{x}$, но, конечно, это можно сделать более эффективно напрямую в строке $\boldsymbol{x}$. Поэтому более существенно то, что поскольку каждая подстрока строки $\boldsymbol{x}$ является префиксом некоторого суффикса этой же строки, дерево суффиксов можно использовать для определения того, является ли данная строка $\boldsymbol{u}$ подстрокой строки $\boldsymbol{x}$. Более того, на основе дерева суффиксов можно определить позицию первого вхождения подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$, а также позиции *всех* возможных

вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$. Можно также применить дерево суффиксов для нахождения кратных строк в данной строке [18, 107, 216]. Поскольку дерево суффиксов можно сформировать для набора строк путем их сочленения, все вычислительные возможности дерева суффиксов распространяются на *множества* строковых последовательностей. Эффективные алгоритмы построения дерева суффиксов описаны в разделе 5.2.

В главе 11 мы покажем, что дерево суффиксов можно естественным образом интерпретировать как конечные автоматы. В этой интерпретации поиск по дереву суффиксов строки $\boldsymbol{p}$ эквивалентен доказательству выводимости выходного слова $\boldsymbol{p}$ конечным автоматом.

Задача 2.4 (Вычисление дерева суффиксов). Построить дерево суффиксов $T_{\boldsymbol{x}}$ для данной строки $\boldsymbol{x}$ (раздел 5.2).

Обсуждение 2.1.1. Сделаем замечание об эффективности построения дерева суффиксов и выполнения поиска по нему.

В разделе 5.2 мы изучим три алгоритма построения дерева суффиксов, два из которых требуют упорядоченности алфавита (раздел 4.1). Напомним, что в упорядоченном алфавите для каждой пары различных букв λ и μ за фиксированное время можно определить, выполняется ли неравенство $\lambda > \mu$. Эти два алгоритма выполняются за время порядка $O(n \log \alpha)$, где α — размер алфавита. Поскольку возможно, что $\alpha \in \Theta(n)$, то в самом худшем случае время выполнения алгоритма имеет порядок $\Omega(n \log n)$. Это нижняя граница временно́й сложности алгоритмов построения деревьев суффиксов для упорядоченных алфавитов.

Третий алгоритм из раздела 5.2 — это недавнее "открытие"; время выполнения этого алгоритма имеет порядок $\Theta(n)$ для индексированных алфавитов (раздел 4.1). Индексированный алфавит можно трактовать как множество целых чисел $\{1, 2, \ldots, \alpha\}$ для некоторого $\alpha \in O(n)$. Индексированные алфавиты очень часто встречаются на практике, например двоичный алфавит можно рассматривать как индексированный, точно так же как любое подмножество символов ASCII (в том числе английский алфавит), алфавит ДНК и РНК и многие другие.

Фактически, если алфавит упорядочен, но не индексирован, в большинстве случае построение дерева суффиксов выполняется эффективно. В разделе 5.2 представлен ***онлайновый*** алгоритм построения дерева суффиксов, когда дерево $T_{\boldsymbol{x}\lambda}$ (для некоторой буквы λ) строится на основании дерева $T_{\boldsymbol{x}}$ за время порядка $O(\log n)$. Это означает, что для новых строк, которые получены путем добавления ***справа*** новых букв (т.е. путем добавления новых суффиксов), построение нового дерева суффиксов (на основе уже существующего дерева) требует только незначительных временны́х ресурсов. Подобным образом этот алгоритм можно эффективно использовать для построения дерева суффиксов при расширении набора строк.

Дополнительной к задаче построения дерева суффиксов является задача поиска по этому дереву. В этой задаче в каждом неконечном узле дерева $T_{\boldsymbol{x}}$ необходимо определить (основываясь на текущем значении буквы $\boldsymbol{u}[i]$ строки $\boldsymbol{u}$) нижележащие узлы, которые соответствуют этой строке. Это требует сравнения буквы $\boldsymbol{u}[i]$ с буквой по крайней мере одного из нижележащих узлов, а возможно и всех.

Если алфавит упорядочен, то такое сравнение для каждой буквы $\boldsymbol{u}[i]$ можно выполнить за время порядка $O(\log \alpha)$ путем использования подходящих структур данных (например, структуры дерева поиска или упорядоченного массива) для каждого узла. Для алфавитов малого размера эффективна реализация дерева $T_{\boldsymbol{x}}$ в виде бинарного дерева, как показано в упражнении 2.1.7. В общем случае для поиска строки $\boldsymbol{u}$ в строке $\boldsymbol{x}$ может потребоваться время порядка $\Omega(|\boldsymbol{u}| \log \alpha)$. Если $|\boldsymbol{u}|$ мало по сравнению с $n = |\boldsymbol{x}|$ или размер алфавита также относительно небольшой, то для поиска всей строки $\boldsymbol{x}$ обычно требуется время порядка $O(n)$.

Поиск по дереву суффиксов остается наиболее эффективным в случае индексированных алфавитов, поскольку здесь оценочный фактор $\log \alpha$ можно проигнорировать на основании того, что при использовании процедур просмотра таблиц определить необходимый узел можно за конечное время. С другой стороны, если α велико, эффективный по времени алгоритм может потребовать большого объема памяти (порядка $\Theta(\alpha)$) для каждого узла дерева $T_{\boldsymbol{x}}$. Когда алфавит упорядочен, иногда дает хороший результат применение техники хеш-таблиц, однако здесь снова для получения "почти постоянного" времени выполнения поиска приносится в жертву необходимый объем памяти.

В целом, дерево суффиксов — это чрезвычайно эффективное и широко используемое средство для поиска и сравнения строк. Его ахиллесовой пятой является необходимость использования больших объемов памяти, особенно в случае больших упорядоченных алфавитов (но и для малых алфавитов это может стать проблемой — см. обсуждение в подразделе 5.2.5). В общем случае применение дерева суффиксов для поиска и сравнения строк наиболее привлекательно, когда строка $\boldsymbol{x}$ неизменна, алфавит небольшой по размеру и фиксированный, а строка $\boldsymbol{u}$ относительно короткая. Этим условиям удовлетворяют, например, последовательности ДНК: алфавит фиксирован (C, G, A, T), как и $\boldsymbol{x}$, при этом, как правило, $|\boldsymbol{u}|$ значительно меньше n. ■

Кроме деревьев суффиксов, предложено большое количество разнообразных структур суффиксов. В разделе 5.3 мы рассмотрим две из них: ориентированные ациклические графы слов и массивы суффиксов.

Задача 2.5 (Вычисление структур суффиксов). Построить различные структуры суффиксов (раздел 5.3). ■

Внутренние паттерны представляют большой интерес для декомпозиции (факторизации) строковых последовательностей. В главе 6 рассмотрим линдонскую

декомпозицию (кратко описанную в разделе 1.4), где покажем ее связь с каноническими формами для петель. Решение задач декомпозиции основано на вычислении лексикографически наименьшего и наибольшего суффиксов для каждого префикса строки $\boldsymbol{x}$. Эффективные алгоритмы линдонской декомпозиции описаны в разделах 6.1 и 6.2.

Задача 2.6 (Вычисление линдонской декомпозиции). Вычислить линдонскую декомпозицию строковой последовательности $\boldsymbol{x}$ (раздел 6.1). ■

Еще одной важной формой декомпозиции, весьма отличной от линдонской декомпозиции, является так называемая s-факторизация [156], которая впервые была применена как средство сжатия строковых последовательностей [235] и затем послужила основой для эффективных алгоритмов вычисления периодических составляющих строковых последовательностей. Если говорить кратко, то ***s-факторизацией*** строки $\boldsymbol{x}$ называется декомпозиция $\boldsymbol{x} = \boldsymbol{w}_1\boldsymbol{w}_2\cdots\boldsymbol{w}_k$, где каждая подстрока $\boldsymbol{w}_j$ $(j = 1, 2, \ldots, k)$ является или одиночной буквой, которая не встречается в подстроке $\boldsymbol{w}_1\boldsymbol{w}_2\cdots\boldsymbol{w}_{j-1}$, или наибольшей подстрокой, которая одновременно будет суффиксом и собственной подстрокой строки $\boldsymbol{w}_1\boldsymbol{w}_2\cdots\boldsymbol{w}_j$. Мы определим s-факторизацию более формально и приведем алгоритмы для ее вычисления в разделе 6.3, а ее применение покажем в разделе 12.2.

Задача 2.7 (Вычисление s-факторизации). Вычислить s-факторизацию данной строковой последовательности $\boldsymbol{x}$ (раздел 6.3). ■

Упражнения 2.1

1. Покажите, что "дерево граней" является ациклическим и связным деревом с корнем в узле 0.[5]
2. Приведите примеры компактного синтаксического дерева и дерева суффиксов, у которых из корневого узла исходит только одно ребро.
3. Покажите, что метки ребер, исходящих из какого-нибудь узла компактного синтаксического дерева, не имеют общего непустого префикса.
4. Пусть в данном дереве с k конечными узлами каждый внутренний узел имеет в точности два исходящих ребра. Покажите, что это дерево содержит ровно $k - 1$ внутренних узлов. На основании этого утверждения докажите, что соответствующее дерево суффиксов содержит не более n внутренних узлов.

[5] В принципе, если структура является деревом, то дерево ациклично (т.е. не имеет циклов) по определению. Здесь предлагается доказать, что структура "дерево граней" не имеет циклов. — *Примеч. ред.*

5. Обобщите предыдущий результат: докажите, что для компактного синтаксического дерева, построенного для множества из k строк, требуется хранить информацию не более чем о $O(k)$ узлах и ребрах. Приведите пример обычного синтаксического дерева, для которого это утверждение не выполняется.

6. Нарисуйте дерево суффиксов для строки $\boldsymbol{x} = ababbabbaba$.

7. Предположим, что построено дерево суффиксов $T_{\boldsymbol{x}}$ для некоторой строки $\boldsymbol{x}$ на упорядоченном алфавите A, где $\alpha = |A|$ — известное фиксированное целое число. Для каждого из приведенных ниже алфавитов предложите структуру данных для узлов, которая позволяла бы эффективно находить правильное исходящее ребро при поиске по дереву. (Таких структур данных можно предложить несколько, в зависимости от того, считать ли α "маленьким" или "большим".)

а) $A = \{1, 2, \ldots, \alpha\}$.

б) Алфавит A содержит упорядоченные, но не обязательно последовательные буквы: $\lambda_1 < \lambda_2 < \cdots < \lambda_\alpha$.

Разработайте собственный алгоритм построения дерева суффиксов для строк на фиксированном алфавите. Попытайтесь разработать такой алгоритм, который был бы максимально эффективным, — время его выполнения не должно быть больше $O(n^2)$!

8. В упражнении 1.2.12 предлагалось разработать алгоритм для вычисления всех различных подстрок данной строки $\boldsymbol{x}$ за время порядка $O(n^2)$. Покажите, что если уже построено дерево суффиксов для этой строки $\boldsymbol{x}$, тогда найти все различные подстроки этой строки можно за время порядка $\Theta(n)$.

Совет. Заметьте, что различные подстроки, которые начинаются в позиции i (т.е. строки вида $\boldsymbol{x}[i..j]$, $i \leqslant j$), являются префиксами различных подстрок минимальной длины, которые также начинаются в позиции i. Таким образом, для нахождения искомых подстрок надо просто выписать все различные префиксы минимальной длины.

2.2 Частные паттерны

Паттерны, обсуждаемые в этом разделе, относятся к тому типу, которые вычисляются "классическими" алгоритмами обработки паттернов: имеется строка (иногда называемая ***текстом***) и требуется найти одно или все вхождения в нее другой заданной строки (которая называется ***паттерном***). Такую операцию, например, многократно (за один сеанс работы) выполняют текстовые редакторы. Другой пример этой задачи не такой очевидный: молекулярные биологи получают из международной базы данных генетической информации сегменты ДНК длиной

5 млн комплиментарных пар оснований нуклеиновых кислот (в нашей терминологии — это строка из 5 млн букв) для определения местоположения в этом большом сегменте отдельных коротких сегментов ДНК, состоящих из 300 комплиментарных пар оснований нуклеиновых кислот. Здесь опять применяются алгоритмы обработки паттернов.

Алгоритм 2.2.1

```
▷ Поиск всех вхождений строки p в строку x
for i ← 1 to n − m + 1 do
    j ← сравнить(i, m)
    if j = m + 1 then
        output i
```

Удивительно простой алгоритм! Но с другой стороны, ни для кого не составит труда написать подобный алгоритм, поскольку он очевиден: для каждой позиции i в строке $\boldsymbol{x}$ выполняется тривиальная процедура *сравнить*(i, m), которая сравнивает по одной букве слева направо строку $\boldsymbol{p}[1..m]$ с подстрокой $\boldsymbol{x}[i..i+m-1]$. Эта процедура возвращает значение $m+1$, если $\boldsymbol{p} = \boldsymbol{x}[i..i+m-1]$, либо наименьшее целое $j \in 1..m$, такое, что $\boldsymbol{p}[1..j] \neq \boldsymbol{x}[i..i+j-1]$. Таким образом, если $j = m+1$, то это означает, что строка $\boldsymbol{p}$ входит в строку $\boldsymbol{x}$ начиная с позиции i. Все очень просто! Алгоритм 2.2.1 выполняется корректно, что доказывается в упражнении 2.2.2.

Недостатком такого алгоритма является его неэффективность. Например, в случае $\boldsymbol{x} = a^n$, $\boldsymbol{p} = a^{m-1}b$ процедура *сравнить* должна выполнить m побуквенных сравнений для *каждого* $i \in 1..n-m+1$, т.е. всего $m\,(n-m+1)$ побуквенных сравнений. Таким образом, алгоритм 2.2.1 требует $O(mn)$ времени выполнения, что чрезвычайно много, когда, например, $n = 5\,000\,000$ и $m = 300$ (как в исследованиях ДНК). Хотя алгоритм 2.2.1 в среднем не требует столь большого времени выполнения, все же стараются избегать его применения из-за временно́й затратности и иногда непредсказуемого поведения. Как мы уже видели, использование дерева суффиксов позволяет решить ту же задачу за время порядка $O(m \log \alpha)$, по крайней мере тогда, когда уже построено дерево суффиксов. Как будет показано в главах 7 и 8, существует много алгоритмов, которые гарантируют, что все вхождения строки $\boldsymbol{p}$ в строку $\boldsymbol{x}$ можно найти за время порядка $O(n+m)$. В главе 7 рассмотрены четыре "базовых" алгоритма обработки паттернов, а в главе 8 — еще несколько дюжин (не сотен) более сложных алгоритмов, которые являются оптимальными или в теории или на практике (но редко будут оптимальными одновременно и на практике и в теории).

Задача 2.8 (Вычисление всех паттернов). С гарантированной эффективностью вычислить все вхождения непустой строки (паттерна) $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$ (главы 7 и 8). ■

Во многих практических ситуациях сформулированные выше задачи поиска и сравнения строковых последовательностей не всегда адекватно отображают действительность. Например, при исследовании ДНК частичные паттерны часто повторяются в различных позициях, но эти повторяющиеся подстроки не совпадают абсолютно точно. Другими словами, в практических ситуациях представленный паттерн $\boldsymbol{p}$ может распознаваться на каких-либо подстроках исследуемой строки $\boldsymbol{x}$, даже если между ними (между паттерном и сравниваемой подстрокой) нет полного совпадения. Например, если $\boldsymbol{p} = CGAT$ и допустима перестановка двух первых или двух последних букв, тогда $\boldsymbol{p} \approx GCAT$ и $\boldsymbol{p} \approx CGTA$. Поэтому в строке

$$\boldsymbol{x} = TCTAGG\underline{CGAT}TCG\underline{GCAT}ATTCG\underline{CGTA}GCTCTA \qquad (2.1)$$

подчеркнутые подстроки будут считаться совпадающими с паттерном $\boldsymbol{p}$.

Основная идея в использовании аппроксимирующих (приближенных) паттернов состоит в определении "расстояния" между двумя строками. В общем случае ***расстояние*** между строками $\boldsymbol{p}_1$ и $\boldsymbol{p}_2$ рассчитывается как взвешенное количество стандартных операций редактирования, необходимых для выполнения преобразования $\boldsymbol{p}_1 \rightarrow \boldsymbol{p}_2$. Обычно рассматривают следующие операции редактирования.

- ***Вставка*** — например, вставка символа G в строку $GCAT$ сформирует строку $GCGAT$.
- ***Удаление*** — например, удаление символа G из строки $GCGAT$ сформирует строку $CGAT$.
- ***Подстановка*** — например, подстановка символа G вместо символа C из строки $GCAT$ сформирует строку $GGAT$.

Конечно, перечисленные операции определены только для непустых символов. На основе этих операций можно определить несколько типов расстояния между строками, некоторые из которых мы сейчас рассмотрим.

1. Если две строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ имеют одинаковую длину n, ***расстояние Хемминга*** $d_H(\boldsymbol{x}_1, \boldsymbol{x}_2)$ [109] определяется как минимальное количество *подстановок*, необходимых для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$. Так, $d_H(GCAT, CGAT) = 2$. Отметим, что для замены некоего символа $\boldsymbol{x}_1[i]$ на какой-нибудь другой может потребоваться одна операция удаления этого символа и одна операция вставки нового символа после символа $\boldsymbol{x}_1[i-1]$.
2. Для произвольных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ ***расстояние Левенштейна*** $d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)$ [158] определяется как минимальное количество операций *удаления* и *вставки*, необходимых для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$. Так, $d_L(GCAT, CGAT) = 2$, но

$$d_L(GCGAT, CGAT) = d_L(CAT, CGAT) = 1.$$

Если строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ имеют одинаковую длину, то совсем не обязательно, что $d_H(\boldsymbol{x}_1, \boldsymbol{x}_2) = d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Например, если $\boldsymbol{x}_1 = CGA$ и $\boldsymbol{x}_2 = AGT$, то $d_H(CGA, AGT) = 2$, тогда как $d_L(CGA, AGT) = 4$. Отметим, что последовательность операций удаления и вставки, преобразующих строку $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$, может определяться неоднозначно, даже если общее количество таких операций одинаково. Например, строку $CGATA$ можно преобразовать в строку $ATACG$ путем применения четырех операций удаления символов CG и AT и последующих четырех вставок символов AT и CG либо путем применения двух операций удаления префикса CG и последующих двух вставок суффикса CG.

3. Для произвольных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ ***расстояние преобразования*** (edit distance) $d_E(\boldsymbol{x}_1, \boldsymbol{x}_2)$ [158] определяется как минимальное количество операций *удаления*, *вставки* и *подстановки*, необходимых для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$. Здесь предполагается, что подстановка одной буквы вместо другой выполняется за одну операцию, тогда как при вычислении расстояния d_L подстановка выполняется за две операции: удаление и последующая вставка. Чтобы лучше понять, как d_E соотносится с d_H и d_L, рассмотрим строки $\boldsymbol{x}_1 = CGACG$ и $\boldsymbol{x}_2 = GTCGA$. Для этих строк $d_H(\boldsymbol{x}_1, \boldsymbol{x}_2) = 5$, поскольку ни на одном месте в этих строках нет попарно совпадающих букв, и поэтому для получения из одной строки другой требуется ровно пять подстановок. Но $d_L(\boldsymbol{x}_1, \boldsymbol{x}_2) = 4$, так как необходимо удалить префикс C, вставить суффикс A и заменить букву A на букву T, для чего потребуется одна операция удаления и одна операция вставки. Однако $d_E(\boldsymbol{x}_1, \boldsymbol{x}_2) = 3$, поскольку здесь для замены буквы A на букву T требуется только одна операция.

   Читатель может заметить, что в литературе термины "расстояние Левенштейна" и "расстояние преобразования" используются не последовательно: иногда расстояние преобразования называется расстоянием Левенштейна и *наоборот*. Я обещаю, что в книге эти термины будут применяться именно так, как они определены выше.

4. Предыдущие определения расстояния основаны на том, что все операции редактирования подсчитываются с одинаковыми значениями. Однако по крайней мере для операции подстановки одинаковые значения не всегда верно отражают условия решаемой задачи. Например, если некоторые подстановки (скажем, $C \to G$) выполняются чаще или с большей вероятностью, чем другие подстановки (например, $C \to T$). Чтобы отобразить такие различия, разным подстановкам назначаются разные веса, которые представляются в виде ***матрицы весов*** W. (Так, например, делается в молекулярной биологии.) Матрица весов имеет размерность $\alpha \times \alpha$ и состоит из неотрицательных действительных чисел, где число $W[i, j]$ является весом замены

(подстановки) j-й буквы i-й буквой алфавита A. Для произвольных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ ***взвешенное расстояние*** $d_W(\boldsymbol{x}_1, \boldsymbol{x}_2)$ определяется как минимальная сумма количества операций удаления и вставки и весов операций подстановок, необходимых для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$, при этом веса задаются матрицей весов W. Например, предположим, что $A = \{C, G, A, T\}$ и матрица W имеет вид

	C	G	A	T
C	0,0	0,4	0,9	1,0
G	0,4	0,0	1,0	0,9
A	0,9	1,0	0,0	0,5
T	1,0	0,9	0,5	0,0

Тогда, используя пример из определения 3, получим

$$d_W(CGACG, GTCGA) = 2{,}5,$$

поскольку теперь подстановка $A \to T$ "стоит" только 0,5. Более сложные варианты матрицы весов рассматриваются в разделе 13.3.

Обычно требуется, чтобы функция расстояния d, определенная на какой-либо области D, удовлетворяла условиям ***метрики***: для любых u, v, $w \in D$

$$d(u, v) \geqslant 0, \qquad \text{(условие неотрицательности)} \tag{2.2}$$

$$d(u, v) = 0 \Leftrightarrow +u = v, \qquad \text{(условие единственности)} \tag{2.3}$$

$$d(u, v) = d(v, u), \qquad \text{(условие симметричности)} \tag{2.4}$$

$$d(u, w) \leqslant d(u, v) + d(v, w), \qquad \text{(неравенство треугольника)} \tag{2.5}$$

В упражнении 2.2.12 предложено доказать, что расстояния d_H, d_L и d_E в действительности являются метриками, а в упражнении 2.2.13 — что расстояние d_W будет метрикой в том случае, если матрица W симметричная. Однако во многих практических приложениях матрица W не симметричная или же необходимо, чтобы функция расстояния учитывала не только веса операции подстановки, но и возможные веса операций вставки и удаления. Кроме того, пример строки (2.1) показывает, что все приведенные функции расстояния не рассматривают возможность операции *взаимообмена* соседних (или даже не соседних) букв в строке вместо или в дополнение операций подстановки, удаления и вставки. Учет такой операции в функции расстояния — это не просто "корректировка" определения расстояния между строками, от этого очень сильно зависит эффективность алгоритмов сравнения и поиска строк.

Первой и основной задачей, связанной с расстоянием между строками, конечно, является следующая задача.

Задача 2.9 (Вычисление расстояния). Для произвольных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ на основании определения данной функции расстояния d вычислить $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ (глава 9).■

В этом контексте представляет интерес еще одна проблема. Для строки $\boldsymbol{x}$ построим строку

$$\boldsymbol{x}' = \boldsymbol{x}[i_1]\boldsymbol{x}[i_2]\cdots\boldsymbol{x}[i_k],$$

где $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$ и $1 \leqslant k \leqslant n$. Строка $\boldsymbol{x}'$ называется ***подпоследовательностью*** строки $\boldsymbol{x}$. Для полноты положим, что пустая строка ε является подпоследовательностью любой строки. Очевидно, что любая подстрока строки $\boldsymbol{x}$ является также ее подпоследовательностью. С другой стороны, строка $\boldsymbol{x}' = cold$ будет подпоследовательностью, но не подстрокой строки $\boldsymbol{x} = scrolled$. Имея две строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$, мы можем поставить задачу нахождения ***наибольшей общей подпоследовательности*** $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$[6]. Тогда, например,

$$cold = LCS(scrolled, could).$$

Эта задача возникает в молекулярной биологии и тесно связана с задачей 2.9 и особенно с расстоянием Левенштейна. Фактически, как показано в упражнении 2.2.9, любой алгоритм, определяющий минимальную последовательность операций удаления и вставки для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$, также эффективно находит $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Для этого сначала надо выполнить в обеих строках необходимые операции удаления, и результирующая строка будет LCS. Таким образом, алгоритм нахождения наибольшей общей подпоследовательности можно построить на основе алгоритма вычисления расстояния Левенштейна. Однако, как показано в упражнении 2.2.15, $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ не обязательно определяется однозначно.

Задача 2.10 (Вычисление LCS). Вычислить наибольшую строку $\boldsymbol{x}'$, которая будет подпоследовательностью двух заданных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ (глава 9). ■

Хотя задача вычисления $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k)$, $k > 2$, является NP-полной[7] [130], для случая $k = 2$ и строк длиной n показано, что эту задачу можно решить за время порядка $\Omega(n^2)$, если сравнения выполняются на буквах некоего алфавита [4], и за время порядка $\Omega(n \log n)$, если алфавит упорядочен [116]. Различные типы алфавитов рассматриваются в разделе 4.1.

При решении задачи вычисления аппроксимирующих паттернов используется решение задачи 2.9 для нахождения максимального расстояния между паттернами.

[6] Здесь LCS (от англ. *L*ongest *C*ommon *S*ubsequence) переводится как "наибольшая общая подпоследовательность". — *Примеч. ред.*

[7] NP-полной, если говорить кратко, называется такая задача, которая эквивалентна задаче полного перебора. — *Примеч. ред.*

Задача 2.11 (Вычисление всех аппроксимирующих паттернов). Для данных строки $\boldsymbol{x}$, непустого паттерна $\boldsymbol{p}$, функции расстояния d и числа $k \geqslant 0$ вычислить все подстроки $\boldsymbol{x}'$ строки $\boldsymbol{x}$, такие, что $d(\boldsymbol{p}, \boldsymbol{x}') \leqslant k$ (глава 10). ■

Еще один важный способ реализовать идею приближенного равенства строк заключается в том, чтобы сгруппировать строки, соответствующие некоторому "паттерну", основанному на определенных регулярных выражениях. Прежде чем описывать такие паттерны в полном объеме, рассмотрим применение в паттернах ***символов замещения***, которые замещают в строке одну или несколько букв данного алфавита. Такие символы используются для поиска определенных слов в тексте на каком-нибудь "стандартном" языке (например, английском), находят применение в вычислительной технике (например, шаблоны (маски) имен файлов в системе DOS или функция grep в системе Unix). Символы замещения являются "метасимволами", которые могут присутствовать в обычных выражениях. Существуют два общепринятых символа замещения:

символ $\bullet$ замещает одну любую букву из алфавита A;
символ $*$ замещает любую строку из множества A^*.

Например, в строке $\boldsymbol{x} = aaabaabbaabbba$ паттерну $\boldsymbol{p}_1 = a \bullet a$ соответствуют подстроки aaa и aba, тогда как паттерну $\boldsymbol{p}_2 = a * a$ — подстроки aa, aaa, aba, $abba$, $abbba$ и многие другие, включая саму строку $\boldsymbol{x}$. Данный пример показывает основную проблему, которая возникают при использовании паттернов с символами замещения, — это потеря свойства транзитивности при сравнении строк, т.е. строки, совпадающие с паттерном (например, с паттерном $a \bullet a$), могут не совпадать друг с другом. Это значительно затрудняет использование "обычных" алгоритмов сравнения строк с паттернами.

Задача 2.12 (Вычисление совпадений с паттернами, содержащими символы замещения). Вычислить все подстроки данной строки $\boldsymbol{x}$, совпадающие с непустым паттерном $\boldsymbol{p}$, который, возможно, содержит символы замещения $\bullet$ и $*$ (раздел 10.4).

Вернемся к паттернам, являющимся регулярными выражениями. Такие паттерны сочетают в себе свойства аппроксимирующих паттернов и паттернов, по которым можно сделать множественный выбор. Как увидим далее, многие паттерны можно записать с помощью регулярных выражений. Построение регулярных выражений начнем с введения ***метасимволов*** — специальных символов, присутствующих в паттерне, которые *не являются* буквами алфавита A и имеют определенное значение. Неявно мы уже использовали метасимволы, например, когда строку $abababaabaabacccc$ записывали в более краткой форме, как $(ab)^2(aba)^3c^4$. И, конечно, метасимволами являются символы замещения, рассмотренные выше.

С помощью метасимволов можно представить целые классы строковых последовательностей, которые невозможно указать путем задания "простых" отдельных паттернов. Дадим определение нескольких метасимволов, которые значительно расширяют "представительские возможности" регулярных выражений.

M1. Для любой пары паттернов $\boldsymbol{p}_1$ и $\boldsymbol{p}_2$ запись $\boldsymbol{p}_1 \mid \boldsymbol{p}_2$ означает паттерн $\{\boldsymbol{p}_1, \boldsymbol{p}_2\}$, т.е. в сравнении участвуют и паттерн $\boldsymbol{p}_1$ и паттерн $\boldsymbol{p}_2$. Например,

- паттерну $a^3 \mid b^3$ соответствуют и строка a^3 и строка b^3,
- паттерну $C \mid G \mid A$ соответствует любой элемент множества $\{C, G, A\}$,
- паттерну $\varepsilon \mid a \mid b \mid ab$ соответствует любой элемент множества $\{\varepsilon, a, b, ab\}$.

На конечном алфавите A метасимвол $\mid$ не только обобщает символ замещения $\bullet$, но и расширяет его возможности.

В операционной системе Unix и во многих языках формирования запросов к базам данных выражение $a \mid b$ обозначает запись $[ab]$. Если алфавит A упорядочен, то в дальнейшем мы, возможно, будем использовать выражение типа $[a-zA-Z]$, которое будет обозначать $a \mid b \mid \cdots \mid z \mid A \mid B \mid \cdots \mid Z$. Это только синтаксическое сокращение записи и никак не влияет на создание паттернов.

M2. Для любого паттерна $\boldsymbol{p}$ запись $\boldsymbol{p}^*$ означает нулевую или любую (конечную) конкатенацию (сочленение) строк $\boldsymbol{p}$. Например,

- паттерну a^* соответствует любой элемент множества $\{a^i\}$, где i — произвольное неотрицательное целое число и считается, что $a^0 \equiv \varepsilon$;
- паттерну $(a \mid b)^*$ соответствует любой элемент множества A^*, где $A = \{a, b\}$;
- паттерну $(a \mid b)^*(a \mid b)(a \mid b)^*$ соответствует любой элемент множества A^+, где $A = \{a, b\}$;
- паттерну $a^* \mid b^*$ соответствует любой элемент множества $\{a^i, b^i\}$, где i — произвольное неотрицательное целое число;
- паттерну a^*b^* соответствует любая строка вида a^ib^j, где i и j — неотрицательные целые числа;
- паттерну $(ab)^*$ соответствует любая строка вида $(ab)^i$, где i — неотрицательное целое число.

Из этих примеров видно, что комбинация метасимволов $\mid$ и * может соответствовать символу замещения $*$. Но с помощью метасимвола * можно представить больше паттернов, чем с помощью любой комбинации символов замещения, например паттерны a^* и $(ab)^*$ нельзя записать с помощью символов замещения.

M3. Для любого паттерна $\boldsymbol{p}$ запись $\sim\boldsymbol{p}$ означает любую строку, не соответствующую паттерну $\boldsymbol{p}$. Например, на алфавите $A = \{a, b\}$

- паттерну $\sim(a^*)$ соответствует паттерн $(a \mid b)^* b (a \mid b)^*$, т.е. любая строка, содержащая букву b;
- паттерну $\sim(a \mid b)^*$ не соответствует ни одна строка, даже пустая;
- паттерну $\sim((a \mid b)^*(a \mid b)(a \mid b)^*)$ соответствует только пустая строка ε;
- паттерну $\sim(a^* b^*)$ соответствует любая строка, в которой буквы b предшествуют буквам a, т.е. этот паттерн эквивалентен паттерну $a^* b b^* a (a \mid b)^*$;
- паттерну $\sim(a^* \mid b^*)$ соответствует любая строка, содержащая как букву a, так и букву b, т.е. этот паттерн эквивалентен паттерну

$$((a \mid b)^* a (a \mid b)^* b (a \mid b)^*) \mid ((a \mid b)^* b (a \mid b)^* a (a \mid b)^*) \qquad (2.6)$$

- паттерну $\sim(ab)^*$ соответствует любая непустая строка, в которой нет степеней строки ab, т.е. этот паттерн эквивалентен паттерну $(ab)^*(a \mid (aa \mid b)(a \mid b)^*)$.

Приведенные примеры показывают, что любые паттерны, которые включают метасимвол $\sim$, можно переписать без его использования. Такая ситуация имеет место и в общем случае — не существует классов строковых последовательностей, которые нельзя было бы описать с помощью лишь метасимволов $\mid$ и *. Таким образом, метасимвол $\sim$ является только "метасимволом для удобства", поскольку он позволяет сократить запись многих регулярных выражений.

Сделаем замечание: в приведенных выше выражениях с метасимволами была допущена небольшая вольность — символ $\boldsymbol{p}$ использовался как для обозначения отдельных строк, так и для паттерна, который представляет *множество* строк. Я надеюсь, что такая вольность не вызовет проблем у читателя. В оправдание могу сказать, что это сделано для сокращения текста и во избежание излишнего формализма.

Теперь можно сформулировать следующее определение.

Определение 2.2.1. Регулярным выражением называется паттерн, который может содержать метасимволы $\mid$ и *. **Регулярным множеством** называется множество строковых последовательностей, представленных регулярным выражением. ■

В связи с этим определением возникает следующая задача.

Задача 2.13 (Вычисление всех совпадений с регулярным выражением). Для данной строки $\boldsymbol{x}$ вычислить все ее подстроки, которые соответствуют данному регулярному выражению $\boldsymbol{p}$ (раздел 11.1). ■

Использование метасимволов позволяет реализовать идею сравнения строк не с одним паттерном, а с множеством паттернов. В разделе 11.2 соответствующая

задача будет представлена в двух формах: точной и приближенной. Задача нахождения точного совпадения множества паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ с подстроками строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ фактически является частным случаем задачи сравнения с регулярным выражением $\boldsymbol{p} = \boldsymbol{p}_1 \mid \boldsymbol{p}_2 \mid \cdots \mid \boldsymbol{p}_r$. Такое сравнение можно выполнить за время порядка $\Theta(m+n)$, где $m = |\boldsymbol{p}_1| + |\boldsymbol{p}_2| + \cdots + |\boldsymbol{p}_r|$. В разделе 11.2 также описаны два алгоритма для решения более сложной задачи нахождения приближенного совпадения подстрок строки $\boldsymbol{x}$ с множеством паттернов.

Задача 2.14 (Вычисление множественных паттернов). Вычислить все (точные или приближенные) совпадения всех паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ с подстроками заданной строки $\boldsymbol{x}$. ■

Упражнения 2.2

1. Напишите алгоритм "тривиальной" процедуры *сравнить* и докажите ее корректность.
2. Докажите, что алгоритм 2.2.1 корректно выполняется для любого паттерна $\boldsymbol{p}$. Удостоверьтесь, что вы рассмотрели все возможные случаи, включая случаи, когда $n = 0$, $m = 0$, $m = n$, $m > n$. Имеет ли смысл рассматривать случаи пустой строки и пустого паттерна?
3. Предположим, что алгоритм 2.2.1 применяется к строке $\boldsymbol{x}$ длиной n и паттернам $\boldsymbol{p}$ фиксированной длины m, которые генерируются "случайным образом" на алфавите размером $\alpha \geqslant 2$. Покажите, что ожидаемое количество сравнений можно вычислить по приближенной формуле
$$(n - m + 1)\left(\frac{\alpha}{\alpha - 1}\right)\left(1 - \left(\frac{1}{\alpha}\right)^m\right).$$
4. Для строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ длиной n покажите, что
   а) $d_L(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant 2d_H(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant 2n$;
   б) $d_H(\boldsymbol{x}_1, \boldsymbol{x}_2) - d_L(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant n - 2$, $n \geqslant 2$;
   в) $d_E(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant \min\{d_H(\boldsymbol{x}_1, \boldsymbol{x}_2), d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)\}$.

   Приведите примеры строк, для которых приведенные неравенства будут выполняться в виде равенств.
5. Вычислите следующие расстояния для функций расстояния d_H, d_L и d_E соответственно.
   а) $d(\boldsymbol{x}, \varepsilon)$;
   б) $d((ab)^3, (ba)^3)$;
   в) $d(ababbba, aabbaab)$.

6. При обсуждении расстояния d_L указывалось, что $d_L(CGACG, GTCGA) = 4$, поскольку для преобразования $CGACG \to GTCGA$ необходимы две операции удаления и две операции вставки. Предложите другие четыре операции удаления и вставки для выполнения этого преобразования.

7. Для произвольной строки $\boldsymbol{x}[1..n]$ покажите, что для любого $j \in 0..n-1$ $d_H(\boldsymbol{x}, R_j(\boldsymbol{x})) \neq 1$, где $R_j(\boldsymbol{x})$ — j-й циклический сдвиг строки $\boldsymbol{x}$ (см. раздел 1.4).

8. Для расстояний преобразования и Левенштейна найдите такие непустые строки $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ и $\boldsymbol{x}_3$, что $|\boldsymbol{x}_1| = |\boldsymbol{x}_2|$ и $d(\boldsymbol{x}_1, \boldsymbol{x}_2) > d(\boldsymbol{x}_1, \boldsymbol{x}_2\boldsymbol{x}_3)$.

9. Пусть имеется три строки $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ и $\boldsymbol{x}_3$, такие, что $\boldsymbol{x}_1$ является подпоследовательностью строки $\boldsymbol{x}_2\boldsymbol{x}_3$, и выполняется неравенство

$$|x_2x_3| - |\boldsymbol{x}_1| < d(\boldsymbol{x}_1, \boldsymbol{x}_2),$$

где d — расстояние преобразования или Левенштейна. Вейлин Лу (Weilin Lu) предположил, что в этом случае должно выполняться неравенство $d(\boldsymbol{x}_1, \boldsymbol{x}_2) > d(\boldsymbol{x}_1, \boldsymbol{x}_2\boldsymbol{x}_3)$. Прав ли Вейлин Лу?

10. Определим ***расстояние удаления*** (deletion distance) $d_D(\boldsymbol{x}_1, \boldsymbol{x}_2)$ между двумя строками $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ как минимальное количество операций удаления, необходимых для преобразования строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ в одинаковые строки. (Например, $d_D(ab, bac) = 3$, поскольку необходимо три операции удаления для преобразования строк ab и bac в строку a либо столько же операций удаления для преобразования в строку b.) Докажите, что

$$d_D(\boldsymbol{x}_1, \boldsymbol{x}_2) = |\boldsymbol{x}_1| + |\boldsymbol{x}_2| - 2|\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)| = d_L(\boldsymbol{x}_1, \boldsymbol{x}_2).$$

11. Определим ***асимметричное расстояние преобразования*** $d_A(\boldsymbol{x}_1, \boldsymbol{x}_2)$ между двумя строками $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ как минимальное количество операций удаления в строке $\boldsymbol{x}_2$ плюс минимальное количество операций подстановки/удаления в строке $\boldsymbol{x}_1$, необходимых для преобразования строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ в одинаковые строки. Покажите, что $d_A(\boldsymbol{x}_1, \boldsymbol{x}_2) = d_E(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

12. Покажите, что функции расстояния d_H, d_L и d_E являются метриками в произвольной области A^*, т.е. для них выполняются условия (2.2)–(2.5).

13. Покажите, что функция расстояния d_W будет метрикой тогда и только тогда, когда матрица W будет симметричной с нулевой диагональю и положительными внедиагональными элементами.

14. Покажите, что расстояние Левенштейна не зависит от порядка, в котором выполняются операции удаления и вставки. Интерпретируя операцию подстановки как последовательность операций удаления и вставки, докажите аналогичное утверждение для расстояния преобразования и взвешенного

расстояния. Какие метрические условия используются при доказательстве этих утверждений?

15. Пусть строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ такие, что ни одна из них не является подпоследовательностью другой. Покажите, что в этом случае $\boldsymbol{x} = \mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ только тогда, когда $d_L(\boldsymbol{x}_1, \boldsymbol{x}) + d_L(\boldsymbol{x}_2, \boldsymbol{x}) = d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

16. Покажите на примерах, что результат операции $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ не единственный. Далее покажите, что для произвольного положительного целого числа k можно найти две строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ длиной $2k$, такие, что для них различные наибольшие общие подпоследовательности будут различаться в k позициях.

17. Для произвольных двух строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ подобно $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ можно также вычислить ***наибольший общий фактор*** (подстроку), который обозначается как $\mathrm{LCF}(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Разработайте алгоритм вычисления $\mathrm{LCF}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ на основе деревьев суффиксов и найдите его алгоритмическую сложность.

18. На алфавите $A = \{a, b\}$ охарактеризуйте строки, которые соответствуют паттерну $\boldsymbol{p} = a \bullet b * a$.

19. Упростите следующие выражения для паттернов.

а) $(a \mid b)^*(a \mid b)(a \mid b)^*$;

б) $(a \mid b)^* b (a \mid b)^*$;

в) $(a^* \mid b^*)^*$.

20. Покажите, что выражение (2.6) эквивалентно выражению $((a^*ab) \mid (b^*ba))$ $(a \mid b)^*$. Можно ли еще упростить это выражение?

2.3 Характеристические паттерны

В этом разделе мы рассмотрим задачи, которые требуют вычисления в строках характеристических паттернов. Эти паттерны отображают внутреннюю структуру строк и не выражаются в виде набора определенных подстрок или классов подстрок, которые также со своей стороны характеризуют строковые последовательности. В большинстве случаев структура строк основана на идее периодичности, рассмотренной в разделе 1.2. Грубо говоря, характеристические паттерны в некотором смысле отображают "приближенную периодичность".

Начнем с паттернов, которые действительно отображают *точную* периодичность строк — наличие в строке кратных подстрок, определенных в разделе 1.2. Кратные строки привлекли внимание математиков еще в начале 20 столетия, когда Аксель Туе (Axel Thue) [220] рассмотрел конструкцию бесконечных некратных строковых последовательностей на алфавите из трех букв. В разделах 3.2 и 3.3 мы рассмотрим некоторые "строки Туе". Проблема возродилась в компьютерных

науках как задача вычисления всех кратных подстрок данной строки, которую поставили в 70-х годах (по-видимому, первыми) Мейн и Лоренц (Main and Lorentz) [169]. Эта задача и ее расширения нашли применение в молекулярной биологии и в алгоритмах сжатия данных.

В алгоритмах решения подобных задач критическим вопросом является определение максимального количества возможных кратных подстрок, находящихся в данной строке. Это количество определяет объем выходных результатов выполнения алгоритмов и устанавливает нижнюю границу алгоритмической сложности алгоритмов. Например, рассмотрим строку $\boldsymbol{x} = aaaaaa$. Эта строка содержит пять вхождений квадрата a^2, три вхождения квадрата $(aa)^2$ и одно вхождение квадрата $(aaa)^2$. Таким образом, данная строка имеет девять различных вхождений квадратов. В общем случае нетрудно показать, что строка a^n содержит $\lfloor n^2/4 \rfloor$ вхождений различных квадратов. Отсюда следует естественное заключение, что для вычисления всех квадратов строки длиной n необходимо время порядка $\Omega(n^2)$.

Но это "естественное" заключение ошибочно! Например, если мы действительно захотим "сделать перепись" всех квадратов строки a^6, то без труда заметим, что их легко получить на основании того факта, что подстрока $\boldsymbol{x}[1] = a$ повторяется шесть раз. Эту информацию можно записать с помощью "тройки Крочемора" [69]: $(i, p^*, r^*) = (1, 1, 6)$. Эта запись обозначает, что подстрока длиной $p^* = 1$, которая начинается в позиции $i = 1$, повторяется $r^* = 6$ раз. Таким образом, здесь p^* является периодом, а r^* — степенью строки $\boldsymbol{x}$, как они определены в разделе 1.2. Другими словами, тройка (i, p^*, r^*) предлагает разложение в виде нормальной формы любой максимальной кратной подстроки, присутствующей в данной строке $\boldsymbol{x}$. Например, строка $\boldsymbol{x} = a^2b^3a^2b^3a$ имеет кратные подстроки

$$(1, 1, 2), (3, 1, 3), (6, 1, 2), (8, 1, 3), (1, 5, 2), (2, 5, 2).$$

Поскольку нормальная форма полностью описывает любую строку, для решения задачи нахождения всех кратных подстрок необходимо найти нормальные формы всех максимальных кратных подстрок, содержащихся в данной строке $\boldsymbol{x}$. Однако напомним (см. упражнение 1.2.19), что в нормальной форме u^{r^*} любой строки образующая строка $\boldsymbol{u}$ не может быть кратной строкой.

Можно сказать, что тройка (i, p^*, r^*) — это способ ***кодирования*** кратных подстрок. В терминах такого кодирования в разделе 3.4 мы покажем, что строки Фибоначчи $\boldsymbol{f}_n$ содержат $\Theta(f_n \log f_n)$ максимальных кратных подстрок (здесь $f_n \equiv \equiv |\boldsymbol{f}_n|$), каждая из которых кодируется одной тройкой (i, p^*, r^*). Отсюда следует, что $\Omega(n \log n)$ является нижней границей сложности любого алгоритма поиска всех кратных подстрок данной строки $\boldsymbol{x}$ длиной n. В главе 12 мы покажем, что эта нижняя граница достигается на алгоритмах, решающих следующую задачу.

Задача 2.15 (Вычисление кратных подстрок). Вычислить все кратные подстроки данной строки $\boldsymbol{x}$ длиной n за время $O(n \log n)$ (раздел 12.1).

Обсуждение 2.3.1. Кодирование кратных подстрок, описанное выше, неявно предполагает, что имеется массив граней (см. обсуждение 1.3.2). Это позволяет нам (по соглашению, сделанному в обсуждении 1.3.2) хранить больше информации в памяти меньшего объема — с помощью массива граней в памяти объемом $\Theta(n)$ можно хранить информацию объемом $\Theta(n \log n)$. Поэтому информацию о кратных подстроках, общее количество которых имеет порядок $\Theta(n^2)$, можно с использованием нормальной формы представить (другими словами, сжать) в виде описанных выше троек, которых всего $\Theta(n \log n)$.

В случае массива граней мы не можем надеяться, что удастся уменьшить требуемый объем памяти до величины, меньшей $\Theta(n)$. Однако в случае кратных подстрок встает законный вопрос: можно ли найти такой способ кодирования всех кратных подстрок во "что-то такое", что потребует объема памяти меньшего $\Theta(n \log n)$, например объема $\Theta(n \log \log n)$ или даже $\Theta(n)$? Выше было сказано, что нижняя граница времени вычисления всех кратных подстрок составляет $\Omega(n \log n)$, но эта величина не связана с требуемым объемом памяти. С другой стороны, очевидна тривиальная оценка $\Omega(n)$ необходимого объема памяти для записи всех кратных подстрок. В разделе 3.4 мы покажем, что в случае строк Фибоначчи $\boldsymbol{f}_n$ для описания всех кратных подстрок достаточно памяти порядка $\Theta(f_n)$. Отсюда вытекают следующие вопросы: можно ли добиться линейного порядка необходимого объема памяти для других типов строк, и если это возможно, то почему невозможен линейный порядок времени вычислений?

Мы рассмотрим эти интересные вопросы в главе 12, а данное обсуждение поможет найти правильные ответы. ■

Задача 2.15 — это "классическая" задача, связанная с повторяемостью подстрок; она рассматривалась во многих работах вплоть до 90-х годов прошлого столетия. И по мере исследования этой задачи идея кратных подстрок изменялась и получала развитие в самых разных направлениях. Например, большой интерес представляют несмежные повторяемые подстроки. Они нашли применение в анализе последовательностей ДНК. Другой пример: хранение, извлечение и анализ музыкальных текстов [58], где необходимо распознавать повторяемые мотивы или мелодии в отдельных или последовательных музыкальных фрагментах.

Обсуждение "расширения" понятия кратных подстрок начнем со следующего определения.

Определение 2.3.1. Раппортом (repeat)[8] строки $\boldsymbol{x}$ называется кортеж

$$M_{\boldsymbol{x},\boldsymbol{u}} = (p; i_1, i_2, \ldots, i_r), r \geqslant 2,$$

[8]Слово *repeat* имеет множество значений, и более близким в данном случае был бы его перевод как "повторение". Однако слово "повторение" в разных вариациях будет многократно использоваться в различных ситуациях. Поэтому считаем рациональным использовать термин *раппорт*, который также является переводом слова *repeat*. — *Примеч. ред.*

где $i_1 < i_2 < \cdots < i_r$ и

$$\boldsymbol{u} = \boldsymbol{x}[i_1..i_1 + p - 1] = \boldsymbol{x}[i_2..i_2 + p - 1] = \cdots = \boldsymbol{x}[i_r..i_r + p - 1].$$

Подстрока $\boldsymbol{u}$ называется **повторяемой подстрокой** строки $\boldsymbol{x}$ и **производящей строкой** для $M_{\boldsymbol{x},\boldsymbol{u}}$. Как и в случае кратных подстрок, назовем $p = |\boldsymbol{u}|$ **периодом** повторения, а r — **показателем**. Если кортеж охватывает все вхождения $\boldsymbol{u}$ в строку $\boldsymbol{x}$, тогда раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ называется **полным** и обозначается как $M^*_{\boldsymbol{x},\boldsymbol{u}}$.[9] ■

С учетом этого определения нашу задачу можно сформулировать как вычисление раппортов $M^*_{\boldsymbol{x},\boldsymbol{u}}$ для каждой повторяемой подстроки $\boldsymbol{u}$. Как мы увидим далее, задачу можно упростить путем введения понятия "продолжаемости". Но сначала рассмотрим связь нового понятия раппорта с уже знакомым понятием кратных строк.

Очевидно, что раппорт является обобщением кратных строк с заимствованием соответствующих сопровождающих понятий производящей строки, периода и показателя. Чтобы показать связь между ними более четко, введем понятие ***лакуны*** как разности

$$g_j = i_{j+1} - i_j, \quad 1 \leqslant j \leqslant r - 1,$$

между последовательными элементами раппорта $M_{\boldsymbol{x},\boldsymbol{u}}$. Теперь можно классифицировать раппорты в соответствии со значениями лакун.

- Раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ назовем кратной строкой (или ***последовательным раппортом*** (tandem repeart)), если для всех лакун $g_j = p$.[10] В этом случае $M_{\boldsymbol{x},\boldsymbol{u}}$ можно записать в сокращенной форме

$$(i, p, r) \equiv (p; i, i + p, \ldots, i + (r - 1)p).$$

- Раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ назовем ***расщеплением*** (split), если для всех лакун $g_j > p$.[11]
- Раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ назовем ***покрытием*** (overlap), если для всех лакун $g_j < p$.
- Раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ назовем ***оболочкой*** (cover), если для всех лакун $g_j \leqslant p$. В этом случае будем говорить, что $M_{\boldsymbol{x},\boldsymbol{u}}$ является оболочкой строки $\boldsymbol{x}[i_1..i_r + p - 1]$, а эта строка ***имеет оболочку*** или $\boldsymbol{u}$-оболочку.
- Если раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ не попадает ни в одну из перечисленных категорий, назовем его ***смешанным***.

[9] Если отвлечься от технических деталей и формы записи, то раппорт можно определить как кратную строку $\boldsymbol{u}^r$, где $\boldsymbol{u}$ — *подстрока* строки $\boldsymbol{x}$. Сама строка-раппорт может быть подпоследовательностью или подстрокой строки $\boldsymbol{x}$ либо не быть ни тем, ни другим (см. классификацию раппортов ниже). — *Примеч. ред.*

[10] В этом случае раппорт является подстрокой строки $\boldsymbol{x}$. — *Примеч. ред.*

[11] В этом случае раппорт является подпоследовательностью строки $\boldsymbol{x}$. — *Примеч. ред.*

Чтобы пояснить приведенную классификацию раппортов, рассмотрим пример строки

$$\begin{array}{rccccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a & a & b & a & a & b & a & b & a & a & b & a & b & a \end{array}$$

Пусть $\boldsymbol{u} = aba$. Можно определить следующие раппорты.

- Раппорты (3; 1, 4) и (3; 6, 9, 12) являются кратными строками.
- Раппорт (3; 1, 6, 12, 17) является расщеплением.
- Раппорт (3; 4, 6) является покрытием.
- Раппорт (3; 1, 4, 6, 12) является смешанным.
- Полный раппорт $M^*_{\boldsymbol{x},\boldsymbol{u}} = (3;\ 1,\ 4,\ 6,\ 9,\ 12,\ 14,\ 17,\ 19)$ является оболочкой строки $\boldsymbol{x}$.

Заметим, что из полного раппорта $M^*_{\boldsymbol{x},\boldsymbol{u}}$ можно эффективно вычислить все повторения подстроки $\boldsymbol{u}$, все покрытия и все подстроки максимальной длины, имеющие $\boldsymbol{u}$-оболочки, — вся эта информация содержится в полном раппорте. Так в нашем примере просматривая полный раппорт, определяем, что

- раппорты (3; 1, 4), (3; 6, 9, 12) и (3; 14, 17) являются повторениями подстроки $\boldsymbol{u} = aba$ в строке $\boldsymbol{x}$;
- раппорты (3; 4, 6), (3; 12, 14) и (3; 17, 19) являются покрытиями;
- существует только одна подстрока максимальной длины, имеющая оболочку, и эта подстрока — сама строка $\boldsymbol{x}$.

В нашем примере рассмотрим также полный раппорт для $\boldsymbol{u} = b$:

$$M^*_{\boldsymbol{x},b} = (1; 2, 5, 7, 10, 13, 15, 18, 20).$$

Просмотр этого раппорта позволяет обнаружить интересный факт: за каждой буквой b следует буква a.[12] Из этого наблюдения сразу получаем раппорт

$$M^*_{\boldsymbol{x},ba} = (2; 2, 5, 7, 10, 13, 15, 18, 20),$$

который в точности совпадает с раппортом $M^*_{\boldsymbol{x},b}$! Из последнего раппорта получаем

$$\boldsymbol{x}[2..3] = \boldsymbol{x}[5..6] = \cdots = \boldsymbol{x}[20..21] \quad \text{и} \quad \boldsymbol{x}[2] = \boldsymbol{x}[5] = \cdots = \boldsymbol{x}[20].$$

Аналогично просмотр раппорта $M^*_{\boldsymbol{x},ba}$ позволяет сделать заключение, что каждой подстроке ba предшествует буква a. Отсюда получаем раппорт

$$M^*_{\boldsymbol{x},aba} = (3; 1, 4, 6, 9, 12, 14, 17, 19).$$

[12] Поскольку все лакуны имеют значения, не меньшие 2. — *Примеч. ред.*

Описанный способ последовательного вычисления раппортов подводит нас к способу кодирования раппортов (подобно кодированию граней и кратных подстрок, см. обсуждения 1.3.2 и 2.3.1), который может уменьшить необходимый объем памяти и время вычислений. Чтобы точно описать этот способ кодирования, введем еще несколько определений.

Определение 2.3.2. Раппорт $M_{\boldsymbol{x},\boldsymbol{u}} = (p; i_1, i_2, \ldots, i_r)$ называется **продолжаемым влево** (lefr-extendible, LE) (соответственно, **продолжаемым вправо** (right-extendible, RE)), если

$$(p; i_1 - 1, i_2 - 1, \ldots, i_r - 1) \quad \text{(соответственно, } (p; i_1 + 1, i_2 + 1, \ldots, i_r + 1))$$

также является раппортом. Если раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ не продолжаем ни влево, ни вправо, то он называется **непродолжаемым** (NE). Для краткости такие раппорты будем обозначать **LE**, **RE** и **NE** соответственно. ■

Интересно рассмотреть особый случай определения 2.3.2, когда раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ является кратной строкой $(\boldsymbol{u}[1..p])^r$, $r \geqslant 2$. Например, пусть $\boldsymbol{x} = \cdots a(aba)^r a \cdots$ и порождающая строка $\boldsymbol{u} = aba$ с периодом $p = 3$. Очевидно, что кратная строка $(aba)^r$ является раппортом типа LE (продолжаемым влево), поскольку $(aab)^r$ является раппортом с порождающей строкой aab с периодом $p = 3$. Из подобных соображений следует, что строка $(aba)^r$ является также раппортом типа RE (продолжаемым вправо). Этот пример показывает, что в случае кратной строки тот факт, что раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ продолжаемый влево, говорит о том, что подстроке $\boldsymbol{u}^r$ в строке $\boldsymbol{x}$ предшествует буква $\lambda = \boldsymbol{u}[p]$ и поэтому раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ можно преобразовать в раппорт $M_{\boldsymbol{x},\boldsymbol{u}'}$ с тем же периодом p, где $\boldsymbol{u}' = \lambda\boldsymbol{u}[1..p-1]$. Аналогично, если раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ продолжаемый вправо, тогда он преобразуется в раппорт $M_{\boldsymbol{x},\boldsymbol{u}''}$ с тем же периодом p, где $\boldsymbol{u}'' = \boldsymbol{u}[2..p]\boldsymbol{u}[1]$. В разделе 1.4 подстроки $\boldsymbol{u}'$ и $\boldsymbol{u}''$ определялись нами как *циклический сдвиг* подстроки $\boldsymbol{u}$. В терминах троек Крочемора сказанное можно записать так: если кратная строка (i, p^*, r^*) является раппортом типа LE, тогда строка $(i-1, p^*, r^*)$ также будет кратной; если же кратная строка (i, p^*, r^*) является раппортом типа RE, тогда кратной будет строка $(i+1, p^*, r^*)$. Эти рассуждения подводят нас еще к одному важному понятию.

Определение 2.3.3. Для данной строковой последовательности $\boldsymbol{x}$ 4-местный кортеж (i, p^*, r^*, t) назовем **серией** (run) строки $\boldsymbol{x}$, если выполняются следующие условия:

а) $t \in 0..p^* - 1$;

б) раппорт (i, p^*, r^*) является кратной строкой, не продолжаемой вправо;

в) раппорт $(i + t, p^*, r^*)$ является кратной строкой, не продолжаемой вправо.

Константа t называется **хвостом** (tail) серии. ■

Понятие серии впервые, по-видимому, было введено Мейном (Main) в работе [168], где для этого понятия использовался менее краткий, но более выразительный термин ***максимальная периодичность*** (maximal periodicity). Для этого же понятия в работе [139] используется термин ***максимальная кратная строка*** (maximal repetition).

В обсуждении 2.3.1 упоминалось, что кратные подстроки в строках Фибоначчи можно вычислить за линейное время; в разделе 3.4 мы также увидим, что за такое же время для этих строк можно вычислить серии. Случай произвольных строк исследован в разделе 12.2: там показано, что количество серий в произвольной строке фактически является линейной функцией от длины строки. Этот результат открывает возможность вычисления всех серий по крайней мере для некоторых строк за время, не превышающее $O(n \log n)$. Таким образом, имеем еще одну задачу.

Задача 2.16 (Вычисление серий). Эффективно вычислить все серии для данной строки $\boldsymbol{x}$ (раздел 12.2). ■

Введенное выше понятие серии подводит к формулировкам задач, связанных с вычислением оболочек и раппортов. Обобщение кратных строк путем введения раппортов позволяет рассмотреть не только расщепленные кратные строки, но и подстроки максимальной длины, имеющие оболочки. Таким способом можно обобщить задачу 2.15 путем замены кратных строк (когда лакуны равны p) на подстроки, имеющие оболочки (когда лакуны не превышают p).

Задача 2.17 (Вычисление подстрок, имеющих оболочки). Вычислить все подстроки данной строки $\boldsymbol{x}$, которые имеют непродолжаемые оболочки. ■

Известны три алгоритма [17, 42, 124], которые решают задачу, очень похожую на задачу 2.17, — задачу вычисления всех подстрок заданной строки, имеющих непродолжаемые вправо оболочки. К сожалению, эти алгоритмы очень сложные как в теоретическом плане, так и в практическом. Поэтому мы не будем исследовать их во всех деталях, но в разделе 13.2, где показано решение близкой задачи вычисления раппортов (задача 2.19), кратко их рассмотрим и обсудим возможное их усовершенствование.

Идея "оболочки" порождает интересную задачу, которая находит применение в сжатии данных: вычислить все оболочки данной строки. Оболочку строки можно рассматривать как обобщение производящей подстроки для кратных строк. С этой точки зрения оболочку строки иногда называют ***квазипериодом***, а строку, имеющую оболочку, — ***квазипериодической***.

Несколько алгоритмов для вычисления квазипериодов предложено в 90-х годах. В главе 13 мы рассмотрим один из них [160], который вычисляет внутренний паттерн, называемый "массивом оболочек", — аналог массива граней, описанно-

го в разделе 1.3. Подобно массиву граней, массив оболочек можно вычислить за линейное время. И так же, как массив граней определяет все периоды всех префиксов строки $\boldsymbol{x}$, так и массив оболочек определяет все квазипериоды всех префиксов строки $\boldsymbol{x}$.

Задача 2.18 (Вычисление оболочек). Вычислить все оболочки заданной строки $\boldsymbol{x}$ (раздел 13.1). ■

В связи с этой задачей очевидна необходимость вычисления всех раппортов типа NE (т.е. непродолжаемых). Это приводит к следующему обобщению задачи 2.15.

Задача 2.19 (Вычисление раппортов). Вычислить все непродолжаемые раппорты в заданной строке $\boldsymbol{x}$ (раздел 13.2). ■

Как будет показано в разделе 13.2, эту общую задачу можно решить за время порядка $\Theta(n \log n)$, и ее решение в свою очередь может послужить основой для решения задач 2.15 и 2.17. Алгоритмы, решающие данную задачу, находят применение в вычислительной биологии.

На практике также находят применение алгоритмы, вычисляющие аппроксимирующие раппорты подстрок, отличающиеся не более, чем на расстояние k, где "расстояние" может принимать одну из форм, описанных в разделе 2.2.

Задача 2.20 (Вычисление аппроксимирующих раппортов). Для данной функции расстояния d и целого числа $k \geqslant 0$ в заданной строке $\boldsymbol{x}$ вычислить все аппроксимирующие раппорты, отличающиеся не более, чем на расстояние k (раздел 13.3). ■

Несмотря на свою значимость, задача вычисления аппроксимирующих раппортов изучена относительно слабо, особенно в сравнении с задачей вычисления аппроксимирующих паттернов (раздел 2.2). Причиной является исключительная сложность данной задачи — даже лучшие алгоритмы имеют время выполнения порядка $O(n^2)$. Трудность задачи, как показано ниже, проявляется уже на этапе точной ее формулировки.

Ранее мы упоминали о нарушении транзитивности при сравнении строк, содержащих символы замещения. Такая же проблема возникает при попытке определить "приближенное равенство" двух подстрок $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ одной строки $\boldsymbol{x}$. Используя, например, расстояние Хемминга, имеем

$$d_H(ab, aa) = d_H(ab, bb) = 1.$$

Отсюда следует, что строки aa и bb можно считать "аппроксимирующими раппортами" для строки ab при условии, что "приближенное равенство" определяется как расстояние между строками, не превышающее $k = 1$. Но поскольку

$d_H(aa, bb) = 2$, при таком условии нельзя считать "равными" любые множества строк, содержащие подстроки или aa или bb. Таким образом, перед нами стоит нелегкий выбор — использовать аппроксимирующие раппорты только в паре (что увеличивает количество необходимых вычислений и все равно имеет на выходе неполный результат) либо использовать вместе только те аппроксимирующие раппорты, расстояние между которыми "мало" (что опять ведет к усложнению вычислений и порождает только частичный ответ).

Свойство продолжаемости, которое оказалось полезным для уменьшения объема вычислений при работе с точными раппортами, также находит применение при работе с аппроксимирующими раппортами. Как и выше, рассмотрим подстроки ab и aa некой строки $\boldsymbol{x}$, расстояние между которыми равно 1, т.е. они "приближенно равны". Предположим, что подстрока aa в строке $\boldsymbol{x}$ может появляться только как суффикс подстроки baa, а подстрока ab может появиться в двух вариантах: как подстрока строки bab или как подстрока строки aab (например, как в строках Фибоначчи). Поскольку $d_H(bab, baa) = 1$, то для подстрок bab аппроксимирующий раппорт $\{ab, aa\}$ в некотором смысле продолжаемый влево. Но для подстрок aab, где a предшествует подстроке ab, нельзя говорить о продолжаемости влево, так как $d_H(aab, baa) = 2 > k$. Таким образом, мы находимся в положении, когда с помощью аппроксимирующего раппорта $\{ab, aa\}$ можно определить только некоторые (но не все) вхождения подстроки ab в строку $\boldsymbol{x}$.

Возможно, описанные трудности уменьшатся (или даже исчезнут), если вместо расстояния Хемминга использовать другую функцию расстояния. Например, $d_L(ab, abc) = 1$, и поэтому при использовании расстояния Левенштейна строки ab и abc могут быть аппроксимирующими раппортами для любого $k \geqslant 1$. Если аппроксимирующие раппорты определять исходя из функции расстояния, отличной от функции Хемминга, то исчезает (или по крайней мере уменьшается) проблема нахождения точных периодов. В общем случае использование различных функций расстояния порождает различные последствия для алгоритмов вычисления аппроксимирующих раппортов, поскольку эти алгоритмы учитывают специфику используемых функций расстояния. Например, если "приближенное равенство" двух строк $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ устанавливается при выполнении неравенства $d(\boldsymbol{u}_1, \boldsymbol{u}_2) \leqslant k$, то при использовании расстояния Хемминга все строки, длина которых не превышает k, будут приближенно равны между собой. Если же использовать расстояние Левенштейна, то строки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ будут приближенно равны, если для них выполняется неравенство $|\boldsymbol{u}_1| + |\boldsymbol{u}_2| \leqslant k$.

Мы вернемся к обсуждению этих проблем в разделе 13.4. Сейчас сформулируем последнюю задачу, в сжатой форме подытоживающую это обсуждение.

Задача 2.21 (Вычисление аппроксимирующих кратных строк). Для данной функции расстояния d и целого числа $k \geqslant 0$ в заданной строке $\boldsymbol{x}$ вычислить

все аппроксимирующие кратные строки, отличающиеся не более, чем на расстояние k (раздел 13.4). ■

Упражнения 2.3

1. Докажите, что для любого целого $n \geqslant 0$ строка a^n содержит $\lfloor n^2/4 \rfloor$ вхождений различных квадратов.

2. Найдите максимальные кратные подстроки в строке Фибоначчи

$$\boldsymbol{f}_6 = abaababaabaab$$

и запишите их в виде троек (i, p^*, r^*). Подсчитайте количество таких троек. Аппроксимируйте этот результат на количество максимальных кратных подстрок в строке $\boldsymbol{f}_7 = \boldsymbol{f}_6(abaababa)$.

3. Строка $\boldsymbol{x}$ называется ***слабо кратной***, если она представима в виде $\boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2 \cdots \boldsymbol{u}_k$ для некоторого $k \geqslant 2$, где для любого $i \in 2..k$ подстрока $\boldsymbol{u}_i$ является перестановкой букв подстроки $\boldsymbol{u}_1$.

а) Покажите, что любая кратная строка также является слабо кратной.

б) Найдите все слабо кратные подстроки в строке $\boldsymbol{f}_6$.

4. Найдите все оболочки, если они существуют, для строк $\boldsymbol{f}_6$ и $\boldsymbol{f}_7$.

5. Предложите определение оболочек $\boldsymbol{u}$ для петель $C(\boldsymbol{x})$. Совпадает ли множество оболочек петли $C(\boldsymbol{x})$ с множеством оболочек всех циклических сдвигов строки $\boldsymbol{x}$?

6. Охарактеризуйте множество оболочек петли $C((abc)^n)$.

7. На основе определения 2.3.2 докажите, что раппорт

$$M_{\boldsymbol{x},\boldsymbol{u}} = (p; i_1, i_2, \ldots, i_r)$$

будет продолжаемым влево последовательным раппортом только тогда, когда раппорт

$$M_{-1\boldsymbol{x},\boldsymbol{u}'} = (p; i_1 - 1, i_2 - 1, \ldots, i_r - 1)$$

будет продолжаемым вправо последовательным раппортом.

8. На основе определения 2.3.3 покажите, что для любого целого $t' \in 1..t-1$ раппорт $(i + t', p^*, r^*)$ будет кратной строкой, продолжаемой как влево, так и вправо.

ГЛАВА 3

Такие разные строки

Там, где идеи терпят крах, становятся чрезвычайно необходимыми умелые слова.

— Иоганн Вольфганг Гете (1749–1837). Фауст

3.1 Проблема исключений и морфизмы

В этой главе мы рассмотрим три специальных типа строковых последовательностей, которые обладают интересными свойствами. Этим типам строк присущи следующие характеристические признаки.

- Строки можно классифицировать по свойствам их кратных подстрок.
- Строки можно сгенерировать путем применения специального вида подстановок, которые называются "морфизмами".

Такие строки представляют интерес как экстремальные входные данные для некоторых строковых алгоритмов, особенно для тех, которые вычисляют характеристические паттерны. Но прежде чем вплотную заняться изучением таких строк, сначала рассмотрим более общую задачу.

Строковая последовательность $\boldsymbol{x}$ называется ***исключаемой*** паттерном $\boldsymbol{p}$ (частным или характеристическим), если отсутствуют вхождения паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$. В этом случае также будем говорить, что строка $\boldsymbol{x}$ ***p-свободна***. Например, строка $\boldsymbol{f}_5 = abaababa$ исключаема паттерном $\boldsymbol{p} = bb$, а строка $\boldsymbol{x} = abacabca$ свободна от квадратов (т.е. в данной строке $\boldsymbol{x}$ нет квадратов — подстрок вида

$\boldsymbol{u}^2$). Классическая проблема исключений в обобщенном виде формулируется как задача построения бесконечных строк, свободных от кратных подстрок.

Задача 3.1 (Построение (α, r)-исключений). Построить бесконечную строковую последовательность (либо бесконечную петлю) на алфавите размером α, свободную от кратных подстрок порядка r, но содержащую кратные подстроки порядков $2, 3, \ldots, r-1$. ■

Отметим, что в некоторых случаях эта задача тривиальна. Например, для $\alpha = 1$ и для любого $r > 1$ не существует строк (состоящих из повторений одной буквы) длины r или большей длины, свободных от кратных подстрок порядка r. Таким образом, невозможно построить $(1, r)$-исключения. Упражнение 3.1.1 показывает, что также невозможно построить $(2, 2)$-исключения, т.е. не существует бесконечных строковых последовательностей на бинарном алфавите, свободных от квадратов. Однако, как покажем далее, можно построить на бинарном алфавите бесконечные строки, свободные от кубов и кратных подстрок порядка 4; другими словами, можно построить $(2, 3)$- и $(2, 4)$-исключения.

Аксель Туе (Axel Thue) [220] первым исследовал проблему исключений, в частности, задачу построения $(2, 3)$- и $(3, 2)$-исключений. Мы рассмотрим эти построения: в разделе 3.2 строки на бинарном алфавите, свободные от кубов, и в разделе 3.3 строки на тернарном алфавите, свободные от квадратов. Наконец, в разделе 3.4 исследуем строки Фибоначчи (порождаемые числами Фибоначчи) и примеры построения $(2, 4)$-исключений. Все эти примеры строковых последовательностей обладают замечательными свойствами, но они могут быть и представителями "плохих", в определенном смысле, строк. Так, $(2, 3)$-исключения Туе и строки Фибоначчи содержат "слишком много" кратных подстрок — фактически строки Фибоначчи содержат максимально (асимптотически) возможное количество кратных подстрок на множестве всех строковых последовательностей. Поэтому такие строки представляют наихудший случай входных данных для алгоритмов вычисления кратных строк. С другой стороны, $(3, 2)$-исключения Туе вообще не содержат кратных подстрок, и, следовательно, такие алгоритмы будут выполняться на них впустую без получения конечного результата (в виде найденных кратных подстрок). Мы также покажем, что строки Фибоначчи представляют наихудший случай для алгоритмов, вычисляющих характеристические паттерны.

Интересное обобщение проблемы исключений (назовем это обобщение ***задачей построения слабых (α, r)-исключений***) предложил Эрдёс (Erdës) [83] — построить, если возможно, бесконечные строковые последовательности на алфавите размера α, не содержащие слабо кратные подстроки порядка r (определение слабо кратных строк дано в упражнении 2.3.3). Поскольку любая кратная строка является слабо кратной, поэтому если задача построения "обычных" исключений

не имеет решения, то в этом случае не будет иметь решения и задача построения слабых исключений. Обратное утверждение, конечно, не верно.

В частности, Эрдёс поставил вопрос о минимальном значении α, при котором существуют бесконечные строки, не содержащие слабые (Абелевые) квадраты. В 1970 году Плесентс (Pleasants) [194] опубликовал решение задачи построения слабых $(5, 2)$-исключений, а недавно Керёнен (Kerënen) [132] нашел (путем машинного моделирования) решение задачи построения слабых $(4, 2)$-исключений. В упражнении 3.1.3 предложено доказать, что результат Керёнена наилучший из возможных.

Вернемся ко второму свойству рассматриваемых в этой главе строк, о котором упоминалось в начале раздела. Морфизмом называется отображение h, которое каждой букве λ из алфавита A ставит в соответствие строку $h(\lambda)$ из множества A^+. Отображение h распространяется на любую строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$ из множества A^+ путем использования следующего тождества

$$h(\boldsymbol{x}) = h(\boldsymbol{x}[1])h(\boldsymbol{x}[2]) \dots h(\boldsymbol{x}[n]). \tag{3.1}$$

Для полноты распространим отображение на множество A^*, положив, что для любого морфизма $h(\varepsilon) = \varepsilon$.

На основании равенства (3.1) любой морфизм h можно применять к исходной строке $\boldsymbol{x}_0$ любое число раз, тем самым генерируя ***последовательность итераций*** $h^*(\boldsymbol{x}_0)$ по следующему правилу

$$h^*(\boldsymbol{x}_0) = \{h^0(\boldsymbol{x}_0), h^1(\boldsymbol{x}_0), h^2(\boldsymbol{x}_0), \dots\},$$

где $h^0(\boldsymbol{x}_0) \equiv \boldsymbol{x}_0$ и для любого целого $k \geqslant 1$ $h^k(\boldsymbol{x}_0) = h(h^{k-1}(\boldsymbol{x}_0))$. Например, для ***тождественного морфизма***, который задается как

$$h(\lambda) = \lambda, \forall \lambda \in A,$$

для любой строки $\boldsymbol{x} \in A^*$ получим тожественную последовательность $h^*(\boldsymbol{x}) = \{\boldsymbol{x}, \boldsymbol{x}, \dots\}$. Для алфавита $A = \{a, b\}$ другой простой морфизм $h(a) = a$, $h(b) = ab$ порождает следующие последовательности для различных начальных строк:

$$\begin{aligned} h^*(a) &= \{a, a, \dots\}, \\ h^*(b) &= \{b, ab, a^2b, \dots, a^kb, \dots\}, \\ h^*(ab) &= \{ab, a^2b, \dots, a^kb, \dots\}, \\ h^*(ba) &= \{ba, aba, a^2ba, \dots, a^kba, \dots\} \end{aligned}$$

и т.д.

В следующих трех разделах данной главы изучаются бесконечные последовательности, порожденные следующими морфизмами.

Строки Туе, являющиеся $(2,3)$-исключениями: $h(a) = ab, h(b) = ba.$ (3.2)

Строки Туе, являющиеся $(3,2)$-исключениями:
$$h(a) = abcab,$$
$$h(b) = acabcb,$$
$$h(c) = acbcacb. \quad (3.3)$$

Строки Фибоначчи: $h(a) = ab, h(b) = a.$ (3.4)

Каждый из этих морфизмов обладает дополнительным важным свойством, которое мы сейчас рассмотрим. Обозначим через $h^*(A)$ множество строк, содержащее все последовательности $h^*(\lambda)$, $\lambda \in A$, т.е.

$$h^*(A) = \bigcup_{\lambda \in A} h^*(\lambda) \quad (3.5)$$

Будем говорить, что морфизм h является ***кодом***, если для любой строки $\boldsymbol{x} \in h^*(A)$ существует не более одной строки $\boldsymbol{y} \in A^*$, такой, что $h(\boldsymbol{y}) = \boldsymbol{x}$. Очевидно, что для любой строки $\boldsymbol{x} \in h^*(A) - A$ существует *не менее одной* такой строки $\boldsymbol{y}$ (что следует из равенства (3.5)), но если $\boldsymbol{x} \in A$, то такой строки $\boldsymbol{y}$ может не существовать. Когда h является кодом и $h(\boldsymbol{y}) = \boldsymbol{x}$, назовем строку $\boldsymbol{y}$ ***предшествующей*** строке $\boldsymbol{x}$ и будем писать $\boldsymbol{y} = h^{-1}(\boldsymbol{x})$, где h^{-1} — обратное отображение; если читатель позволит мне некоторую вольность языка, то в этом случае будем говорить об ***обратном морфизме***.

Таким образом, код является морфизмом, имеющим обратное отображение. Однако следует помнить, что обратное отображение ограничено только множеством $h^*(A) - A$, для строк из множества $A^* - h^*(A)$ может не существовать предшествующих строк либо их может быть несколько. Например, морфизм

$$h : a \to a, \quad b \to ab, \quad c \to abc$$

на алфавите $\{a, b, c\}$ является кодом, хотя такие простые строки, как ba и c^2, не имеют предшествующих строк. Аналогично морфизм

$$h : a \to a, \quad b \to ab, \quad c \to bab$$

также является кодом, но строка $abab$, которая не принадлежит множеству $h^*(A)$, имеет две различные предшествующие строки ac и b^2.

Тот факт, что морфизмы являются кодами, играет ключевую роль при исследовании основных свойств строк Туе, являющихся $(3,2)$-исключениями, и строк Фибоначчи.

Упражнения 3.1

1. Докажите, что не существует решения задачи построения $(2,2)$-исключений.

2. Для любого неотрицательного целого k на алфавите из $k+1$ букв постройте свободные от квадратов строки длиной $n = 2^k$.

3. Докажите, что не существует решения задачи построения слабых $(3,2)$-исключений.

а) Пусть задан морфизм на алфавите $A = \{a,b,c,d\} : h(a) = b, h(b) = c, h(c) = d, h(d) = \varepsilon$. Для каждой строки $\boldsymbol{x} \in A$ постройте $h^4(\boldsymbol{x})$ и $h^5(\boldsymbol{x})$.

б) Измените морфизм так, чтобы $h(d) = a$. Что в этом случае будет представлять собой строка $h^4(\boldsymbol{x})$?

4. Используя определение (3.1), покажите, что если $\boldsymbol{x} = \boldsymbol{uv}$, тогда выполняется равенство $h(\boldsymbol{x}) = h(\boldsymbol{u})h(\boldsymbol{v})$.

5. Пусть заданы морфизм h и начальная строка $\boldsymbol{x}_0$. Будем говорить, что $\boldsymbol{x} \equiv \boldsymbol{y}$ только в том случае, когда и $\boldsymbol{x}$ и $\boldsymbol{y}$ являются элементами последовательности $h^*(\boldsymbol{x}_0)$. Является ли это отношение "$\equiv$" отношением эквивалентности? Поясните свой ответ.

6. Пусть заданы алфавит A и морфизм h на этом алфавите. Возможно ли, чтобы последовательность $h^*(\boldsymbol{x}_0)$ имела конечное число членов?

7. Является ли обратный морфизм морфизмом?

8. Определите, являются ли следующие морфизмы кодами. Поясните свой ответ.

а) $h : a \to aba, b \to ab, c \to aab$;

б) $h : a \to a, b \to ab, c \to ab$;

в) $h : a \to aba, b \to ab$.

3.2 Строки Туе, являющиеся $(2,3)$-исключениями

В этом разделе мы изучим строки Туе $\boldsymbol{t}_n$, определенные с помощью морфизма (3.2): $h(a) = ab$, $h(b) = ba$. Положим

$$h^*(\boldsymbol{t}_0) = \{\boldsymbol{t}_0, \boldsymbol{t}_1, \boldsymbol{t}_2, \dots\},$$

где $\boldsymbol{t}_n = h(\boldsymbol{t}_{n-1})$ для любого $n \geqslant 1$. Пусть $\boldsymbol{t}_0 = a$, тогда

$$\begin{aligned}
\boldsymbol{t}_1 &= ab,\\
\boldsymbol{t}_2 &= abba,\\
\boldsymbol{t}_3 &= abbabaab,\\
\boldsymbol{t}_4 &= abbabaabbaababba,\\
&\vdots
\end{aligned}$$

Очевидно, что $|\boldsymbol{t}_n| = 2^n$. Заметим, что, по крайней мере, в первых нескольких строках $\boldsymbol{t}_n$ первые 2^{n-1} букв "сопряжены" со следующими 2^{n-1} буквами, т.е.

$$\boldsymbol{t}_n[i + 2^{n-1}] = a \Leftrightarrow \boldsymbol{t}_n[i] = b, \quad \forall i \in 1..2^{n-1}. \tag{3.6}$$

Если ввести запись $\overline{a} = b$ и $\overline{b} = a$ для обозначения сопряженности, тогда морфизм (3.2) для любой буквы $\lambda \in \{a, b\}$ можно переписать в виде

$$h : \lambda \to \lambda\overline{\lambda}. \tag{3.7}$$

Эту нотацию можно распространить и на произвольные строки из множества $\{a, b\}^*$: для данной строки $\boldsymbol{x}$ длиной n будем говорить, что строка $\boldsymbol{y}$ ***сопряжена*** со строкой $\boldsymbol{x}$ (запишем это как $\boldsymbol{y} = \overline{\boldsymbol{x}}$), если $|\boldsymbol{y}| = n$ и $\boldsymbol{y}[i] = \overline{\boldsymbol{x}[i]}$ для всех $i \in 1..n$. Отметим, что согласно этому определению любая строка $\boldsymbol{x}$ имеет единственную сопряженную строку. Исключение составляет пустая строка ε, которая сопряжена сама с собой.

Пусть $h(\boldsymbol{x})$ обозначает строку, сформированную путем применения к строке $\boldsymbol{x}$ правила подстановки (3.7). Тогда

$$h(\boldsymbol{x}) = \boldsymbol{x}[1]\overline{\boldsymbol{x}[1]}\boldsymbol{x}[2]\overline{\boldsymbol{x}[2]} \cdots \boldsymbol{x}[n]\overline{\boldsymbol{x}[n]},$$

и используя тот факт, что $\lambda = \overline{\overline{\lambda}}$, можем легко вычислить строку, сопряженную со строкой $h(\boldsymbol{x})$:

$$h(\boldsymbol{x}) = \boldsymbol{x}[1]\overline{\boldsymbol{x}[1]}\boldsymbol{x}[2]\overline{\boldsymbol{x}[2]} \cdots \boldsymbol{x}[n]\overline{\boldsymbol{x}[n]} = h(\overline{\boldsymbol{x}[1]\boldsymbol{x}[2]} \cdots \overline{\boldsymbol{x}[n]}) = h(\overline{\boldsymbol{x}}). \tag{3.8}$$

Теперь можно формально доказать наличие в строках $\boldsymbol{t}_n$ отношения сопряженности (3.6).

Лемма 3.2.1. Для любого целого $n \geqslant 1$ $\boldsymbol{t}_n = \boldsymbol{t}_{n-1}\overline{\boldsymbol{t}_{n-1}}$.

Доказательство. Мы знаем, что при $n = 1$ выполняется $\boldsymbol{t}_1 = ab = a\overline{a} = \boldsymbol{t}_0\overline{\boldsymbol{t}_0}$. Предположим, что утверждение леммы выполняется для некоторого целого $n' = n - 1 \geqslant 1$. Далее методом математической индукции с использованием соотношений (3.1) и (3.8) получаем

$$\boldsymbol{t}_n = h(\boldsymbol{t}_{n-1}) = h(\boldsymbol{t}_{n-2}\overline{\boldsymbol{t}_{n-2}}) = h(\boldsymbol{t}_{n-2})h(\overline{\boldsymbol{t}_{n-2}}) = \boldsymbol{t}_{n-1}\overline{\boldsymbol{t}_{n-1}}.$$

Лемма доказана. ■

Из этой леммы вытекает, что правило подстановки (т.е. морфизм) для строк $\boldsymbol{t}_n$ ***сохраняет префикс*** — строка $\boldsymbol{t}_n$ имеет $\boldsymbol{t}_{n-1}$ своим префиксом. Как мы увидим в разделе 3.4, этим свойством обладают как строки Туе $\boldsymbol{t}_n$, так и строки Фибоначчи $\boldsymbol{f}_n$. В случае строк Туе сохранение префикса означает, что строка $\boldsymbol{t}_n$ имеет префиксы $\boldsymbol{t}_i\overline{\boldsymbol{t}_i}$, $i = 0, 1, \ldots, n-1$, тогда как $\overline{\boldsymbol{t}_n}$ имеет префиксы $\overline{\boldsymbol{t}_i}\boldsymbol{t}_i$. Отсюда сразу следуют два утверждения.

Лемма 3.2.2.

а) Строки $\boldsymbol{t}_n$ и $\overline{\boldsymbol{t}_n}$ не имеют общего непустого префикса;

б) ни строка $\boldsymbol{t}_n$, ни строка $\overline{\boldsymbol{t}_n}$ не имеют префиксом квадрат. ■

Мы оставляем полное доказательство этой леммы в качестве упражнения, а также доказательство аналогичной леммы, в утверждении которой "префикс" заменен на "суффикс". Эти результаты будут использованы далее в доказательстве других лемм и теорем.

В следующей важной лемме встретим наших "старых друзей" — грани. Оказывается, что в строках Туе все грани являются строками Туе. Аналогичный результат также справедлив для строк Фибоначчи (лемма 3.4.4).

Лемма 3.2.3. Для любого целого $n \geqslant 2$ гранями строки $\boldsymbol{t}_n$ будут только строки $\boldsymbol{t}_i$, где $i = n-2, n-4, \ldots, n \bmod 2$.

Доказательство. Для доказательства леммы достаточно показать, что наибольшей гранью $\boldsymbol{t}_n$ является строка $\boldsymbol{t}_{n-2}$. Поскольку

$$\boldsymbol{t}_n = \boldsymbol{t}_{n-1}\overline{\boldsymbol{t}_{n-1}} = \boldsymbol{t}_{n-2}\overline{\boldsymbol{t}_{n-2}\boldsymbol{t}_{n-2}}\boldsymbol{t}_{n-2}, \tag{3.9}$$

то очевидно, что строка $\boldsymbol{t}_{n-2}$ является гранью строки $\boldsymbol{t}_n$. Теперь надо показать, что не существует большей грани.

Предположим, что в строке $\boldsymbol{t}_n$ есть грань $\boldsymbol{b}$, такая, что $|\boldsymbol{b}| \geqslant 2^{n-1}$. В этом случае, как следует из равенства (1.1), строка $\boldsymbol{t}_n$ должна быть сильно периодической и поэтому должна иметь префиксом квадрат, что противоречит утверждению б) леммы 3.2.2. Поэтому грань $\boldsymbol{b}$ может быть только такой, что $|\boldsymbol{b}| < 2^{n-1}$.

Поскольку обе строки $\boldsymbol{b}$ и $\boldsymbol{t}_{n-2}$ являются гранями строки $\boldsymbol{t}_n$ и $\boldsymbol{b}$ — наибольшая грань, то отсюда следует, что $\boldsymbol{t}_{n-2}$ должна быть гранью строки $\boldsymbol{b}$. Из этого замечания и условия $|\boldsymbol{b}| < 2^{n-1}$ вытекает, что строка $\boldsymbol{b}$ должна быть сильно периодической и поэтому должна иметь префиксом квадрат. Но $\boldsymbol{b}$ является префиксом строки $\boldsymbol{t}_n$, которая не может иметь префиксом квадрат. Это противоречие доказывает, что грани, большей $\boldsymbol{t}_{n-2}$, не существует. ■

Из последней леммы вытекает также "сопряженный" результат, что для любого $n \geqslant 2$ строка $\boldsymbol{t}_n$ может иметь гранями только строки $\overline{\boldsymbol{t}_i}$, где $i = n-2, n-4, \ldots,$

$n \bmod 2$. Отметим на будущее, что поскольку $\boldsymbol{t}_{n+1} = \boldsymbol{t}_n\overline{\boldsymbol{t}_n}$ (соответственно, $\overline{\boldsymbol{t}_{n+1}} = \overline{\boldsymbol{t}_n}\boldsymbol{t}_n$), лемма 3.2.3 эквивалентна утверждению о том, что префиксами строки $\boldsymbol{t}_n$ (соответственно, $\overline{\boldsymbol{t}_n}$) могут быть только суффиксы строки $\overline{\boldsymbol{t}_n}$ (соответственно, $\boldsymbol{t}_n$), т.е. строки $\boldsymbol{t}_i$ (соответственно, $\overline{\boldsymbol{t}_i}$), $i = n-2, n-4, \ldots, n \bmod 2$.

Теперь продемонстрируем, как строки $\boldsymbol{t}_n$ можно использовать для решения задачи построения $(2,3)$-исключений. Мы покажем, что хотя при любом $n \geqslant 2$ строка $\boldsymbol{t}_n$ имеет не менее одного квадрата, она не содержит кубов. Фактически, мы докажем следующий результат: любой квадрат в строке $\boldsymbol{t}_n$ является ***непродолжаемым*** в смысле определения, данного в разделе 2.3, т.е. докажем, что если для позиции i и некоторого целого $p \geqslant 1$ выполняется равенство

$$\boldsymbol{t}_n[i..i+2p-1] = (\boldsymbol{t}_n[i..i+p-1])^2,$$

то ни подстрока $\boldsymbol{t}_n[i-1..i+2p-2]$, ни подстрока $\boldsymbol{t}_n[i+1..i+2p]$ не будут квадратом.

Лемма 3.2.4. Пусть $\boldsymbol{u}^2$ является квадратом в строке $\boldsymbol{t}_n$ для некоторого $n \geqslant 2$. Если $|\boldsymbol{u}|$ — нечетное число, тогда строка $\boldsymbol{u}$ будет одним из элементов множества $\{a, b, aba, bab\}$.

Доказательство. Пусть $k = |\boldsymbol{u}| \geqslant 1$. Вследствие морфизма (3.7) строка $\boldsymbol{u}^2$ может иметь только одну из двух следующих форм:

а) $\boldsymbol{u}^2 = \lambda_1\overline{\lambda_1}\lambda_2\overline{\lambda_2}\cdots\lambda_k\overline{\lambda_k}$, где $\boldsymbol{v} = \lambda_1\lambda_2\cdots\lambda_k$ — подстрока строки $\boldsymbol{t}_{n-1}$;

б) $\boldsymbol{u}^2 = \overline{\lambda_0}\lambda_1\overline{\lambda_1}\cdots\lambda_{k-1}\overline{\lambda_{k-1}}\lambda_k$, где $\boldsymbol{v} = \lambda_0\lambda_1\cdots\lambda_k$ — подстрока строки $\boldsymbol{t}_{n-1}$.

В случае а), поскольку k нечетно, имеем

$$\boldsymbol{u} = \lambda_1\overline{\lambda_1}\cdots\overline{\lambda_{\lfloor k/2\rfloor}}\lambda_{\lfloor k/2\rfloor+1} = \overline{\lambda_{\lfloor k/2\rfloor+1}}\lambda_{\lfloor k/2\rfloor+2}\cdots\lambda_k\overline{\lambda_k}.$$

Без потери общности можем положить, что $\lambda_1 = a$, тогда $\overline{\lambda_1} = b$. В этом случае

$$\overline{\lambda_{\lfloor k/2\rfloor+1}} = \lambda_1 = a, \quad \lambda_{\lfloor k/2\rfloor+2} = \overline{\lambda_1} = b,$$
$$\lambda_2 = \overline{\lambda_{\lfloor k/2\rfloor+2}} = b, \quad \lambda_{\lfloor k/2\rfloor+3} = \overline{\lambda_2} = a.$$

Поскольку k нечетно, строка $\boldsymbol{u}$ должна начинаться и заканчиваться буквой a. С другой стороны, так как $\overline{\lambda_{\lfloor k/2\rfloor+1}} = a$, то, должно быть, $\boldsymbol{u}[k] = \lambda_{\lfloor k/2+1\rfloor} = b$. Получили противоречие с тем фактом, что строка $\boldsymbol{u}$ должна заканчиваться буквой a. Приходим к выводу, что форма а) представления строки $\boldsymbol{u}^2$ невозможна.

Теперь предположим, что строка $\boldsymbol{u}^2$ представима в форме б). Тогда

$$\boldsymbol{u} = \overline{\lambda_0}\lambda_1\overline{\lambda_1}\cdots\lambda_{\lfloor k/2\rfloor}\overline{\lambda_{\lfloor k/2\rfloor}} = \lambda_{\lfloor k/2\rfloor+1}\overline{\lambda_{\lfloor k/2\rfloor+1}}\cdots\lambda_{k-1}\overline{\lambda_{k-1}}\lambda_k.$$

Опять без потери общности положим $\lambda_k = a$. Тогда

$$\overline{\lambda_{\lfloor k/2\rfloor}} = \lambda_k = a, \quad \overline{\lambda_{k-1}} = \lambda_{\lfloor k/2\rfloor} = b,$$
$$\overline{\lambda_{\lfloor k/2\rfloor-1}} = \lambda_{k-1} = a, \quad \overline{\lambda_{k-2}} = \lambda_{\lfloor k/2\rfloor-1} = b.$$

Для $k = 1$ строка $\boldsymbol{u}$ равна a или b, а для $k = 3$ — aba или bab. Однако для $k \geqslant 5$ строка $\boldsymbol{u}$ имеет префикс $ababa$ или $babab$. Более того, поскольку строка $\boldsymbol{u}$ начинается с буквы $\overline{\lambda_0}$, в строке $\boldsymbol{t}_n$ ей должна предшествовать буква λ_0. Таким образом, в строке $\boldsymbol{t}_n$ должна присутствовать одна из подстрок $ababab$ или $bababa$. Но тогда в строке $\boldsymbol{t}_{n-1}$ обязательно будет присутствовать одна из "запретных" подстрок $\lambda_0\lambda_1\lambda_2$, равная aaa или bbb. Получили противоречие. Следовательно, $k < 5$. Лемма доказана. ■

Очевидно, что квадраты aa и bb действительно входят в строки $\boldsymbol{t}_n$ и $\overline{\boldsymbol{t}_n}$ при любом $n \geqslant 2$. Чтобы найти первые вхождения квадратов $(aba)^2$ и $(bab)^2$, рассмотрим строку

$$\boldsymbol{t}_5 = abbabaabbaa\underline{babbab}aababbaabbabaab.$$

Мы найдем квадрат $(bab)^2$ в строке $\boldsymbol{t}_5$ стоящим симметрично относительно позиции $2^4 - 1 = 15$ (вследствие симметрии сопряженный квадрат $(aba)^2$ центрирован относительно позиции 19). Тогда, вследствие леммы 3.2.1, можно утверждать, что все четыре различных квадрата, указанные в лемме 3.2.4, будут входить во все строки $\boldsymbol{t}_n$ при $n \geqslant 5$.

Также ясно, что если в строку $\boldsymbol{t}_n$ входит "нечетный" квадрат $\boldsymbol{u}^2$, тогда "четный" квадрат $(h^r(\boldsymbol{u}))^2$ будет входить в строку $\boldsymbol{t}_{n+r}$ при любом положительном целом r. Теперь покажем, что в строку $\boldsymbol{t}_\infty$ входят *только* четные квадраты, являющиеся r-ми итерациями нечетных квадратов.

Рассмотрим квадрат $\boldsymbol{u}^2$, присутствующий в строке $\boldsymbol{t}_n$, для которого $k = |\boldsymbol{u}| \geqslant 2$ четно. Строка $\boldsymbol{u}^2$ (как и в случае нечетных квадратов) может иметь только одну из двух следующих форм:

а) $\boldsymbol{u}^2 = \lambda_1\overline{\lambda_1}\lambda_2\overline{\lambda_2}\cdots\lambda_k\overline{\lambda_k}$, где $\boldsymbol{v} = \lambda_1\lambda_2\cdots\lambda_k$ — подстрока строки $\boldsymbol{t}_{n-1}$;

б) $\boldsymbol{u}^2 = \overline{\lambda_0}\lambda_1\overline{\lambda_1}\cdots\lambda_{k-1}\overline{\lambda_{k-1}}\lambda_k$, где $\boldsymbol{v} = \lambda_0\lambda_1\cdots\lambda_k$ — подстрока строки $\boldsymbol{t}_{n-1}$.

В случае а) имеем

$$\boldsymbol{u} = \lambda_1\overline{\lambda_1}\cdots\lambda_{k/2}\overline{\lambda_{k/2}} = \lambda_{k/2+1}\overline{\lambda_{k/2+1}}\cdots\lambda_k\overline{\lambda_k}.$$

Поэтому строка $(\lambda_1\lambda_2\cdots\lambda_{k/2})(\lambda_{k/2+1}\lambda_{k/2+2}\cdots\lambda_k)$ является квадратом, также принадлежащим строке $\boldsymbol{t}_{n-1}$. Аналогично исследуется случай б):

$$\boldsymbol{u} = \overline{\lambda_0}\lambda_1\overline{\lambda_1}\cdots\lambda_{k/2-1}\overline{\lambda_{k/2-1}}\lambda_{k/2} = \overline{\lambda_{k/2}}\lambda_{k/2+1}\overline{\lambda_{k/2+1}}\cdots\lambda_{k-1}\overline{\lambda_{k-1}}\lambda_k,$$

тогда строка

$$(\lambda_1\lambda_2\cdots\lambda_{k/2})(\lambda_{k/2+1}\lambda_{k/2+2}\cdots\lambda_k) \tag{3.10}$$

является квадратом, также принадлежащим строке $\boldsymbol{t}_{n-1}$.

Отсюда следует, что любой четный квадрат длиной $2k$, присутствующий в строке $\boldsymbol{t}_n$, можно получить из квадрата длиной k, принадлежащего строке $\boldsymbol{t}_{n-1}$. В свою очередь квадрат длиной k, принадлежащий строке $\boldsymbol{t}_{n-1}$, если он четный,

можно получить из соответствующего квадрата строки $\boldsymbol{t}_{n-2}$ либо, если он нечетный, он принимает одну из четырех возможных форм, определенных леммой 3.2.4. Таким образом, мы доказали следующий результат.

Лемма 3.2.5. Каждый квадрат, присутствующий в строке $\boldsymbol{t}_\infty$, представим в форме $(h^r(\boldsymbol{u}))^2$, где r — неотрицательное целое число и $\boldsymbol{u} \in \{a, b, aba, bab\}$. ■

Теперь рассмотрим нечетный квадрат $\boldsymbol{u}^2$ из строки $\boldsymbol{t}_n$ длиной $2k = 2$. В соответствии с утверждением б) леммы 3.2.2 квадрат $\boldsymbol{u}^2$ не может быть ни префиксом ни суффиксом строки $\boldsymbol{t}_n$. Поэтому в $\boldsymbol{t}_n$ должна быть подстрока $\boldsymbol{v} = b\boldsymbol{u}\boldsymbol{u}b$ длиной $2k + 2$ со свойствами

$$\boldsymbol{v}[1] \neq \boldsymbol{u}[k], \quad \boldsymbol{v}[2k + 2] \neq \boldsymbol{u}[1]. \tag{3.11}$$

Эти свойства показывают, что квадрат $\boldsymbol{u}^2$ непродолжаемый. Фактически, нет необходимости доказывать, что такими свойствами обладают все квадраты длиной $2k$, поскольку они получены путем применения преобразования h^r к строке $baab$, а все квадраты $h^r(aa)$ непродолжаемы. Этот факт предложено доказать в упражнении 3.2.7, а в упражнении 3.2.8 показано, что все квадраты $h^r(abaaba)$ также непродолжаемы. Все это доказывает основной результат данного раздела.

Теорема 3.2.6. Все квадраты в строках Туе $\boldsymbol{t}_n$ непродолжаемые. ■

Бесконечное множество строк $h^*(\boldsymbol{t}_0)$, введенное в начале этого раздела, содержит все конечные строки $\boldsymbol{t}_n$, но не содержит бесконечной строки Туе $\boldsymbol{t}_\infty$. Однако заметим, что если строка $\boldsymbol{t}_\infty$ содержит куб, тогда этот куб должен быть конечным и должен быть подстрокой некоторой конечной строки Туе, но последнее запрещено теоремой 3.2.6. Таким образом, получаем следующий результат.

Теорема 3.2.7. Бесконечная строка Туе $\boldsymbol{t}_\infty$ является решением задачи построения $(2, 3)$-исключений. ■

Краткое и, возможно, более элегантное доказательство этого результата приведено в работе [39]. Однако наше более длинное доказательство позволило получить много дополнительной информации о структуре строки $\boldsymbol{t}_\infty$.

В заключение этого раздела рассмотрим один важный вычислительный вопрос: сколько всего квадратов присутствует в строке $\boldsymbol{t}_n$ или, по крайней мере, сколько квадратов имеют форму $h^r(a)$ или $h^r(b)$. Условимся называть квадраты, которые можно записать в такой форме, ***регулярными***, поскольку любой такой квадрат строится путем сочленения суффикса строки $\boldsymbol{t}_n$ (соответственно, $\overline{\boldsymbol{t}_n}$) с префиксом строки $\overline{t_n}$ (соответственно, $\boldsymbol{t}_n$). Как следует из леммы 3.2.3, регулярные квадраты $\overline{\boldsymbol{t}_i}$ $(i = n - 2, n - 4, \ldots, n \bmod 2)$ формируются в центре строки $\boldsymbol{t}_n = \boldsymbol{t}_{n-1}\overline{\boldsymbol{t}_{n-1}}$. Квадратов в строке $\boldsymbol{t}_n$, которые сформированы из квадратов строк

$\boldsymbol{t}_{n-1}$ и $\overline{\boldsymbol{t}_{n-1}}$, всего $\lfloor n/2 \rfloor$. Отсюда получаем рекуррентное соотношение

$$q(n) - 2q(n-1) = \lfloor n/2 \rfloor, \quad n \geqslant 2, \tag{3.12}$$

где $q(n)$ — общее количество регулярных квадратов в строке $\boldsymbol{t}_n$. Предполагая, что n нечетное число, решаем уравнение (3.12) стандартным способом.

$$\begin{aligned} q(n) - 2q(n-1) &= (n-1)/2, \\ 2q(n-1) - 4q(n-2) &= 2(n-1)/2, \\ &\vdots \\ 2^{n-3}q(3) - 2^{n-2}q(2) &= 2^{n-3}(2/2), \\ 2^{n-2}q(2) - 2^{n-1}q(1) &= 2^{n-2}(2/2). \end{aligned}$$

С учетом того, что $q(1) = 0$, суммируем эти уравнения, в результате получаем

$$\begin{aligned} q(n) &= \frac{3}{2}\left[(n-1) + (n-3)2^3 + (n-5)2^4 + \cdots + (2)2^{n-3}\right] = \\ &= 3\sum_{i=0}^{(n-3)/2} (n-2i-1)2^{2i-1}. \end{aligned}$$

Отсюда нетрудно получить границы для $q(n)$:

$$3 \cdot 2^{n-3} \leqslant q(n) \leqslant 3 \cdot 2^{n-2}.$$

Следовательно, $q(n)$ имеет порядок $\Theta(2^n)$. Такой же порядок получим в случае четного n. Это доказывает следующую теорему.

Теорема 3.2.8. Количество $q(n)$ регулярных квадратов в строке $\boldsymbol{t}_n$ имеет порядок $\Theta(|\boldsymbol{t}_n|)$. ■

Более сложным способом Джеми Симпсон (Jamie Simpson) нашел точное значение величины $q(n)$, его формулу предложено доказать в упражнении 3.2.10.

В разделе 3.4 мы увидим, что строки Фибоначчи $\boldsymbol{f}_n$ длиной f_n содержат кратные подстроки, количество которых имеет порядок $\Theta(f_n \log f_n)$, т.е. строки Фибоначчи асимптотически содержат "больше" кратных подстрок, чем строки Туе $\boldsymbol{t}_n$.

Упражнения 3.2

1. Пусть морфизм (3.7) применяется к начальной строке $\boldsymbol{t}'_0 = b$, порождая ***обратные строки Туе*** $\boldsymbol{t}'_n$, $n = 0, 1, \ldots$. Покажите, что $\boldsymbol{t}'_n = \overline{\boldsymbol{t}_n}$.

2. Напомним, что строка называется ***палиндромом***, если она одинаково читается слева направо и справа налево. Покажите, что для любого целого n строка $\boldsymbol{t}_n$ является палиндромом.

3. Формула морфизма (3.7) и лемма 3.2.1 дают основу для ряда итерационных алгоритмов вычисления строк $\boldsymbol{t}_n$. Разработайте несколько таких алгоритмов и сравните их эффективность.

4. Докажите лемму, "сопряженную" с леммой 3.2.2.

5. Почему при доказательстве леммы 3.2.3 мы рассматривали только грани длиной больше $|\boldsymbol{t}_{n-2}|$? Завершите доказательство этой леммы.

6. В доказательстве леммы 3.2.5 есть один "неясный" момент: в строке $\boldsymbol{t}_{n-1}$, кроме квадрата (3.10), присутствует также квадрат

$$(\lambda_0\lambda_1\cdots\lambda_{k/2-1})(\lambda_{k/2}\lambda_{k/2+1}\cdots\lambda_{k-1}).$$

Поэтому квадрат (3.10) будет продолжаемым влево, что противоречит теореме 3.2.6. Как вы проясните этот "неясный" момент?

7. Покажите, что свойства (3.11) выполняются также для всех строк вида

$$\boldsymbol{v} = h^r(baab) = h^r(b)h^r(a^2)h^r(b),$$

где $\boldsymbol{u} = h^r(a)$, $|\boldsymbol{u}| = k$, $r \geqslant 0$. Отсюда будет следовать, что все квадраты $\boldsymbol{u}^2$ непродолжаемые.

8. Покажите, что в строке $\boldsymbol{t}_n$ все квадраты вида $h^r((aba)^2)$ непродолжаемые.

9. Докажите, что если строка $\boldsymbol{u}^2$ является квадратом, содержащимся в строке $\boldsymbol{t}_n$, тогда $|\boldsymbol{u}| = 2^r$ или $3 \cdot 2^r$ для некоторого целого $r \geqslant 0$.

10. Методом математической индукции докажите формулу Джеми Симпсона для $q(n)$, $n \geqslant 0$,

$$q(n) = 2^{n+1}/3 - n/2 - \varepsilon_n,$$

где $\varepsilon_n = 2/3$, если n четно, и $\varepsilon_n = 5/6$, если n нечетно.

11. Можно ли вычислить количество *всех* квадратов, включая регулярные, находящиеся в строке $\boldsymbol{t}_n$?

12. Покажите, что морфизм $(3, 2)$ является кодом.

13. В работе [39] предложено генерировать строку $\boldsymbol{t}_\infty$ следующим образом: сначала записываются все неотрицательные целые числа в двоичной форме

$$0, 1, 10, 11, 100, 101, 110, \ldots,$$

затем в этих числах цифры складываются по модулю 2:

$$0, 1, 1, 0, 1, 0, 0, \ldots,$$

после этого удаляются запятые и 0 заменяется на букву a, а 1 — на букву b. Все, строка $\boldsymbol{t}_\infty$ готова! Докажите или опровергните, что таким способом действительно можно построить строку $\boldsymbol{t}_\infty$.

3.3 Строки Туе, являющиеся (3, 2)-исключениями

В этом разделе мы изучим строки Туе $\tau\boldsymbol{\tau}_n$, которые в определенном смысле противоположны строкам Туе $\boldsymbol{t}_n$, изученным в предыдущем разделе: строки $\boldsymbol{\tau}_n$ вообще не содержат кратных подстрок. Эти строки формируются с помощью морфизма (3.3):

$$h(a) = abcab, \quad h(b) = acabcb, \quad h(c) = acbcacb.$$

Положим $h^*(\boldsymbol{\tau}_0) = \{\boldsymbol{\tau}_0, \boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \dots\}$, где $\boldsymbol{\tau}_n = h(\boldsymbol{\tau}_{n-1})$ для любого целого $n \geqslant 1$. Если $\boldsymbol{\tau}_0 = a$, тогда получим

$$\begin{aligned}
&\boldsymbol{\tau}_1 = abcab,\\
&\boldsymbol{\tau}_2 = (abcab)(acabcb)(acbcacb)(abcab)(acabcb),\\
&\quad\vdots
\end{aligned}$$

Каждая из трех компонент $abcab$, $acabcb$, $acbcacb$ морфизма h называется ***блоком***. Отметим, что любой блок $\boldsymbol{B}$ можно записать в форме $\boldsymbol{IT}$, где строки $\boldsymbol{I}$ и $\boldsymbol{T}$ имеют префиксом букву a и суффиксом строку, построенную на алфавите $\{b, c\}$. Назовем строки $\boldsymbol{I}$ и $\boldsymbol{T}$ соответственно ***инициатором*** и ***терминатором***. Отметим на будущее, что все инициаторы и терминаторы различны:

$$\{abc, ac, acbc; ab, abcb, acb\}.$$

Фактически, множество инициаторов и терминаторов в точности совпадает с множеством всех строк длиной 2 или 3, имеющих префикс a и свободный от квадратов суффикс, содержащий только буквы $\{b, c\}$.

Полезность инициаторов и терминаторов проявляется в том, что если в какой-либо строке из множества $h^*(\boldsymbol{\tau}_0)$ найдется инициатор, тогда за ним следует терминатор. И, аналогично, если в такой строке найден терминатор, то ему предшествует инициатор. Кроме того, каждая строка $\boldsymbol{\tau}_n$ $(n > 0)$ из множества $h^*(\boldsymbol{\tau}_0)$ является сочленением блоков, т.е. тех же инициаторов и терминаторов. Поэтому для любого блока $\boldsymbol{B}$ в строке $\boldsymbol{\tau}_n$ с помощью обратного морфизма h^{-1} можно определить предшествующую подстроку $h^{-1}(\boldsymbol{B})$.

Лемма 3.3.1. Строка Туе $\boldsymbol{\tau}_n$ будет свободной от квадратов только тогда, когда будет свободной от квадратов строка $\boldsymbol{\tau}_{n-1}$.

Доказательство. Очевидно, что строка $\boldsymbol{\tau}_{n-1}$ будет свободной от квадратов, если строка $\boldsymbol{\tau}_n$ свободна от квадратов. Теперь предположим, что при некотором $n \geqslant 1$ в строке $\boldsymbol{\tau}_n$ существует квадрат $\boldsymbol{u} = \boldsymbol{u}_L\boldsymbol{u}_R$.

Если $\boldsymbol{u}_R$ имеет инициатор $\boldsymbol{I}_i$ в качестве префикса для некоторого $i \in 1..3$, тогда для некоторого $k \geqslant 0$ подстрока $\boldsymbol{u}_L$ представима в виде

$$\boldsymbol{u}_L = (\boldsymbol{I}_i\boldsymbol{T}_i)(\boldsymbol{I}_{j_1}\boldsymbol{T}_{j_1})\cdots(\boldsymbol{I}_{j_k}\boldsymbol{T}_{j_k}).$$

Подстроки $\boldsymbol{u}_L$ и $\boldsymbol{u}_R$ состоят из целого числа блоков, и поэтому определено обратное отображение $h^{-1}(\boldsymbol{u}_L)$, а также $h^{-1}(\boldsymbol{u}_L\boldsymbol{u}_R)$, которое будет квадратом, принадлежащим строке $\boldsymbol{\tau}_{n-1}$.

Подобным образом, если $\boldsymbol{u}_R$ содержит терминатор $\boldsymbol{T}_i$ в качестве префикса, для некоторого $k \geqslant 0$ имеем

$$\boldsymbol{I}_i\boldsymbol{u}_L = (\boldsymbol{I}_i\boldsymbol{T}_i)(\boldsymbol{I}_{j_1}\boldsymbol{T}_{j_1})\cdots(\boldsymbol{I}_{j_k}\boldsymbol{T}_{j_k})\boldsymbol{I}_i = \boldsymbol{v}_L\boldsymbol{I}_i.$$

Тогда $\boldsymbol{v}_R = \boldsymbol{T}_i(\boldsymbol{I}_{j_1}\boldsymbol{T}_{j_1})\cdots(\boldsymbol{I}_{j_k}\boldsymbol{T}_{j_k})$ будет префиксом подстроки $\boldsymbol{u}_R$. Поэтому $\boldsymbol{v}^2 = \boldsymbol{v}_L\boldsymbol{I}_i\boldsymbol{v}_R$ будет квадратом, где $\boldsymbol{v} = \boldsymbol{v}_L$ состоит из целого числа блоков. Следовательно, существует обратное отображение $h^{-1}(\boldsymbol{v})$, которое порождает квадрат, принадлежащий строке $\boldsymbol{\tau}_{n-1}$.

Теперь предположим, что $\boldsymbol{u}_R$ не имеет буквы a в качестве префикса. Поскольку не существует подстрок-квадратов, которые не содержали бы букву a, эта буква будет хотя бы по одному разу входить как в подстроку $\boldsymbol{u}_L$, так и в подстроку $\boldsymbol{u}_R$. Поэтому или в некотором инициаторе $\boldsymbol{I}_i$ или в некотором терминаторе $\boldsymbol{T}_i$ выполняется равенство $\boldsymbol{u}_R[1] = \boldsymbol{u}_L[1]$. Далее рассуждения, подобные приведенным выше, приводят к заключению, что в строке $\boldsymbol{\tau}_n$ присутствует квадрат $\boldsymbol{v}^2$, такой, что существует $h^{-1}(\boldsymbol{v}^2)$, порождающее квадрат в строке $\boldsymbol{\tau}_{n-1}$.

Мы рассмотрели все возможные случаи присутствия квадратов в строке $\boldsymbol{\tau}_n$, которые приводят к присутствию квадратов в строке $\boldsymbol{\tau}_{n-1}$. Лемма доказана. ■

Из леммы 3.3.1 немедленно следует теорема.

Теорема 3.3.2. Если начальная строка $\boldsymbol{\tau}_0$ свободна от квадратов, то свободными от квадратов будут все строки из множества $h^*(\boldsymbol{\tau}_0)$. ■

Для окончания анализа этого типа строк Туе осталось заметить, что если бесконечная строка $\boldsymbol{\tau}_\infty$ содержит квадрат, то в этом случае квадраты должны содержать некоторые конечные строки $\boldsymbol{\tau}_n$, что противоречит теореме 3.3.2. Следовательно, справедлива такая теорема.

Теорема 3.3.3. Бесконечная строка Туе $\boldsymbol{\tau}_\infty$ является решением задачи построения $(3, 2)$-исключений. ■

Морфизм (3.3) был первым исследуемым преобразованием, который генерирует свободные от квадратов строки на алфавите из трех букв [29], но существуют и другие подобные морфизмы. Один из них задается следующими формулами [39, 75].

$$h(a) = acb, \quad h(b) = c, \quad h(c) = ab. \tag{3.13}$$

Существует очень "симпатичный" способ показать, что этот морфизм действительно порождает бесконечные строки, свободные от квадратов. Имея строку Туе

$$\boldsymbol{t}_\infty = abbabaabbaababba\cdots,$$

сформируем новую строку $\boldsymbol{u}_\infty$ на алфавите $\{0,1,2\}$ путем подсчета в строке $\boldsymbol{t}_\infty$ букв b между последовательными буквами a. Получим

$$\boldsymbol{u}_\infty = 2102012\cdots.$$

Нетрудно заметить, что если в строке $\boldsymbol{u}_\infty$ существует квадрат, тогда в строке $\boldsymbol{t}_\infty$ существует продолжаемый квадрат, что противоречит теореме 3.2.6. Следовательно, строка $\boldsymbol{u}_\infty$ свободна от квадратов. Наконец, отметим, что строка $\boldsymbol{u}_\infty$ порождена морфизмом

$$h(2) = 210, \quad h(0) = 1, \quad h(1) = 20. \tag{3.14}$$

Подстановка $\{0,1,2\} \to \{b,c,a\}$ показывает, что данный морфизм совпадает с морфизмом (3.13). В упражнении 3.3.5 показаны детали этого соответствия.

В другом морфизме, который также решает задачу построения $(3,2)$-исключений [155], отображение $h(b)$ получается из отображения $h(a)$, а $h(c)$ — из $h(b)$ путем циклической перестановки букв алфавита:

$$\begin{aligned} h(a) &= abcbacbcabcba,\\ h(b) &= bcacbacabcacb,\\ h(c) &= cabacbabcabac. \end{aligned} \tag{3.15}$$

Морфизмы на больших алфавитах, порождающие строки, свободные от квадратов, также представляют интерес. Рассмотрим, например, известную головоломку "Ханойские башни": имеется три стержня, пронумерованные от 1 до 3, и n дисков разного диаметра. Первоначально все диски собраны на стержне 1 в убывающем порядке их диаметров (наименьший диск находится на верху пирамиды). Цель головоломки заключается в том, чтобы переместить диски (по одному за один раз) на другой стержень так, чтобы они опять располагались в первоначальном порядке. Разрешается помещать только меньший диск на больший либо на свободный стержень — другие перемещения дисков запрещены. Возможно всего шесть типов перемещения: со стержня 1 на стержень 2, со стержня 2 на стержень 3, со стержня 3 на стержень 1 и в обратном порядке. Закодируем эти перемещения буквами a, b, c и $\overline{a}$, $\overline{b}$, $\overline{c}$ соответственно. На алфавите $\{a,b,c,\overline{a},\overline{b},\overline{c}\}$ построим морфизм

$$\begin{aligned} &h(a) = a\overline{c}, \quad h(b) = c\overline{b}, \quad h(c) = b\overline{a}.\\ &h(\overline{a}) = ac, \quad h(\overline{b}) = cb, \quad h(\overline{c}) = ba. \end{aligned} \tag{3.16}$$

Этот морфизм решает задачу построения $(6,2)$-исключений и, следовательно, находит минимальную последовательность перемещений дисков в нашей головоломке, поскольку, очевидно, эта минимальная последовательность не должна содержать повторяющихся подпоследовательностей. Подробно этот морфизм рассмотрен в работе [8].

Упражнения 3.3

1. Покажите, что морфизм (3.3) является кодом.
2. Разработайте алгоритм, который на основании морфизма (3.3) по начальной строке $\boldsymbol{\tau}_0$ генерирует заданное количество строк $\boldsymbol{\tau}_n$.
3. Приведите полное и подробное доказательство леммы 3.3.1.
4. Покажите, что шесть блоков $\boldsymbol{I}_i$ и $\boldsymbol{T}_i$, $i = 1,2,3$, можно попарно упорядочить 120 разными способами. Исходя из этого ответьте, сколько можно определить морфизмов, порождающих строки, свободные от квадратов?
5. Чтобы убедиться в том, что морфизм (3.13) действительно порождает строки, свободные от квадратов, докажите следующее.
   а) Если строка $\boldsymbol{u}_\infty$ содержит квадрат, тогда строка $\boldsymbol{t}_\infty$ содержит подстроку $\boldsymbol{x}^2\lambda$ с наибольшей гранью длиной 1.
   б) Морфизм (3.14) порождает строку $\boldsymbol{u}_\infty$ как предел последовательности строк из множества $h^*(0)$.
6. Покажите, что морфизм (3.13) порождает строки длиной $3\cdot 2^i$, $i \geqslant 0$, циклические сдвиги которых свободны от квадратов. Сравните эту задачу с "исследовательской задачей" из упражнения 1.4.4.
7. Херенди Тамаш (Herendi Tamás) заметил, что морфизм (3.13) не обладает одним хорошим свойством, которое присуще морфизму Туе, а именно: морфизм (3.13) не отображает *любую* строку, свободную от квадратов, в строку, также свободную от квадратов. Приведите пример, подтверждающий наблюдение Тамаша.
8. Докажите, что морфизм (3.15) решает задачу построения $(3,2)$-исключений.
9. Докажите, что морфизм (3.16) решает задачу построения $(6,2)$-исключений. (Решение приведено в работе [8].)

3.4 Строки Фибоначчи

Эту главу мы завершим изучением строк Фибоначчи, порождаемых морфизмом (3.4): $h(a) = ab$, $h(b) = a$. Как и в предыдущих разделах, введем множество $h^*(\boldsymbol{f}_0) = \{\boldsymbol{f}_0, \boldsymbol{f}_1, \boldsymbol{f}_2, \dots\}$, где $\boldsymbol{f}_n = h(\boldsymbol{f}_{n-1})$ для любого целого $n \geqslant 1$ и для

определенности положим $\boldsymbol{f}_0 = b$. Далее мы увидим, что длины $f_n = |\boldsymbol{f}_n|$ этих строк являются числами Фибоначчи. Не будет преувеличением сказать, что числа Фибоначчи "оккупировали" комбинаторику и не только — они привлекают такой научный интерес, что им посвящен отдельный журнал *The Fibonacci Quarterly*, где публикуются работы, изучающие их свойства и свойства других итерационных числовых последовательностей.

Приведем несколько первых строк Фибоначчи.

$$\begin{aligned}
\boldsymbol{f}_0 &= b,\\
\boldsymbol{f}_1 &= a,\\
\boldsymbol{f}_2 &= ab,\\
\boldsymbol{f}_3 &= aba,\\
\boldsymbol{f}_4 &= abaab,\\
\boldsymbol{f}_5 &= abaababa,\\
\boldsymbol{f}_6 &= abaababaabaab,\\
&\vdots
\end{aligned} \tag{3.17}$$

Длины этих строк равны соответственно $f_0 = 1$, $f_1 = 1$, $f_2 = 2$, $f_3 = 3$, $f_4 = 5$, $f_5 = 8$, $f_6 = 13$. Строки Фибоначчи связаны соотношением, аналогичным соотношению между числами Фибоначчи.

Лемма 3.4.1. Строки Фибоначчи удовлетворяют рекуррентному соотношению

$$\boldsymbol{f}_0 = b, \quad \boldsymbol{f}_1 = a, \quad \boldsymbol{f}_n = \boldsymbol{f}_{n-1}\boldsymbol{f}_{n-2}, \quad n \geqslant 2. \tag{3.18}$$

Доказательство. леммы легко получить методом математической индукции (см. упражнение 3.4.2). ■

Отсюда получаем, что длины строк $\boldsymbol{f}_n$ удовлетворяют соотношению

$$f_n = f_{n-1} + f_{n-2}, \quad \forall n \geqslant 2, \tag{3.19}$$

которое является рекуррентным соотношением для чисел Фибоначчи [135].

Из соотношения (3.18) следует, что, подобно строкам Туе, строки Фибоначчи $\boldsymbol{f}_n$ при $n > 0$ сохраняют префикс. Это свойство строк Фибоначчи позволяет ввести ***бесконечную строку Фибоначчи*** $\boldsymbol{f}_\infty$ как такую, префиксом которой будет любая строка $\boldsymbol{f}_n$, $n > 0$.

В разделе 3.2 мы видели, что длина строк Туе $\boldsymbol{t}_n$ равна 2^n, т.е. длина удваивалась при каждом применении отображения h. Поскольку длина строк Фибоначчи "почти" удваивается при увеличении номера строки на 1, естественно предположить, что длина f_n будет экспонентой от n. Если это действительно так, тогда

существует действительное число $\phi > 1$, такое, что $f_n \in \Omega(\phi^n)$. Чтобы доказать это, заметим, что из соотношения (3.19) следуют неравенства $f_{n-1} \geqslant \phi^{n-2}$ и $f_{n-2} \geqslant \phi^{n-3}$. Отсюда с необходимостью получаем

$$f_n \geqslant \phi^{n-2} + \phi^{n-3} = \phi^{n-3}(\phi + 1). \tag{3.20}$$

Если число ϕ удовлетворяет уравнению

$$\phi^2 = \phi + 1, \tag{3.21}$$

тогда автоматически получим неравенство $f_n \geqslant \phi^{n-1}$ и, следовательно, аналогичные неравенства для f_{n-1} и f_{n-2}. Уравнение (3.21) — это простое квадратное уравнение относительно ϕ, положительный корень которого имеет значение

$$\phi = \left(1 + \sqrt{5}\right)/2 \approx 1{,}618034. \tag{3.22}$$

Это число, интересное само по себе, часто называется "золотым значением"[1] [135]. Для проверки значения (3.22) заметим, что

$$f_0 = 1 \geqslant \phi^{-1} = \left(\sqrt{5} - 1\right)/2 \approx 0{,}618034 \quad \text{и} \quad f_1 = 1 \geqslant \phi^0 = 1.$$

Из соотношений (3.20) и (3.21) следует, что $f_2 \geqslant \phi^1$, тогда $f_3 \geqslant \phi^2$ и т.д. Отсюда по индукции получаем основной результат

$$f_n \geqslant \phi^{n-1}, \tag{3.23}$$

где число ϕ задается формулой (3.22). В упражнении 3.4.4 предложено доказать, что $f_n \leqslant \phi^n$.

Завершив изложение предварительных сведений о строках Фибоначчи, вернемся к основным свойствам этих строк. Установим два факта, касающиеся строк Фибоначчи, которые на первый взгляд кажутся противоречащими друг другу.

- В терминах троек (i, p^*, r^*), введенных в связи с задачей 2.15, покажем, что в строке Фибоначчи $\boldsymbol{f}_n$ число кратных подстрок имеет порядок $\Theta(f_n \log f_n)$. (Таким образом, строки Фибоначчи являются "наихудшими" входными данными для алгоритмов вычисления кратных подстрок в строковых последовательностях.)
- С другой стороны, мы также покажем, что *все* кратные подстроки в строках Фибоначчи можно найти за время порядка $\Theta(n)$, используя для этого кодировку (i, p^*, r^*, t). Эта кодировка основана на "сериях" кратных подстрок (см. формальное определение серий в разделе 2.3). Если говорить не формально, то серия — это кратная строка $\boldsymbol{u}[1..p^*]^{r^*}$, следующая за (возможно, пустым) собственным префиксом $\boldsymbol{u}[1..t]$ подстроки $\boldsymbol{u}[1..p^*]$.

[1] В отечественной литературе это значение обычно называется "коэффициентом золотого сечения". — *Примеч. ред.*

К этим фактам "из жизни" строк Фибоначчи мы придем кружной дорогой, "посетив по пути некоторые интересные места" свойств этих строк. Как упоминалось ранее, строки Фибоначчи должны быть решением задачи построения (2, 4)-исключений — мы покажем, что они содержат кубы, но не содержат кратных подстрок четвертого порядка.

Начнем обобщение идеи строк Фибоначчи следующим способом. Напомним, что морфизм $h : A \to A^*$ (см. формулы (3.4)) оперирует двумя буквами a и b: $h(a) = ab$, $h(b) = a$. Расширим это отображение на множество A^*, т.е. будем считать, что это отображение h действует $A^* \to A^*$, оперируя при этом двумя произвольными строками $\boldsymbol{x}$ и $\boldsymbol{y}$ из множества A^*:

$$h(\boldsymbol{x}) = \boldsymbol{xy}, h(\boldsymbol{y}) = \boldsymbol{x}. \tag{3.24}$$

Таким образом, морфизм (3.4) будет частным случаем отображения (3.24) при $\boldsymbol{x} = a$ и $\boldsymbol{y} = b$. Аналогично (3.17) можно вычислить

$$\begin{aligned} h^2(\boldsymbol{y}) &= \boldsymbol{xy}, \\ h^3(\boldsymbol{y}) &= \boldsymbol{xyx}, \\ h^4(\boldsymbol{y}) &= \boldsymbol{xyxxy}, \\ &\vdots \end{aligned}$$

Теперь определим n-ю ***обобщенную строку Фибоначчи*** как

$$\boldsymbol{f}_n(\boldsymbol{x}, \boldsymbol{y}) = h^n(\boldsymbol{y}), \tag{3.25}$$

где $h^n(\boldsymbol{y})$ — $(n+1)$-й член последовательности $h^*(\boldsymbol{y}) = \{\boldsymbol{y}, h(\boldsymbol{y}), h^2(\boldsymbol{y}), \dots\}$. Несколько первых обобщенных строк имеют вид

$$\begin{aligned} \boldsymbol{f}_0(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{y}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{x}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{xy}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{xyx}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{xyxxy}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{xyxxyxyx}, \\ \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{xyxxyxyxxyxxy}, \\ &\vdots \end{aligned}$$

В общем случае

$$\boldsymbol{f}_n(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{f}_{n-1}(\boldsymbol{x}, \boldsymbol{y})\boldsymbol{f}_{n-2}(\boldsymbol{x}, \boldsymbol{y}), \quad n \geqslant 2. \tag{3.26}$$

Определим ***бесконечную обобщенную строку Фибоначчи*** $\boldsymbol{f}_\infty(\boldsymbol{x},\boldsymbol{y})$ как строку, содержащую все строки $\boldsymbol{f}_n(\boldsymbol{x},\boldsymbol{y})$, $n \geqslant 0$, в качестве префиксов.

Поскольку $h^n(\boldsymbol{y}) = h^{n-k}(h^k(\boldsymbol{y}))$, из определения (3.25) следует, что

$$\boldsymbol{f}_n(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{f}_{n-k}(h^k(\boldsymbol{x}), h^k(\boldsymbol{y})).$$

Тогда, положив $\boldsymbol{x} = a$ и $\boldsymbol{y} = b$ и заметив, что $h^k(\boldsymbol{x}) = h^{k+1}(\boldsymbol{y})$, получим начальный результат, относящийся к обычным строкам Фибоначчи.

Лемма 3.4.2. Для любых целых k и n, таких, что $0 \leqslant k \leqslant n-1$, справедливо соотношение $\boldsymbol{f}_n = \boldsymbol{f}_{n-k}(\boldsymbol{f}_{k+1}, \boldsymbol{f}_k)$. ■

Этот результат позволяет выразить любую строку Фибоначчи $\boldsymbol{f}_n$ через выбранные строки $\boldsymbol{f}_{k+1}$ и $\boldsymbol{f}_k$. Например, строку $\boldsymbol{f}_7$ можно выразить через строки $\boldsymbol{f}_3$ и $\boldsymbol{f}_2$:

$$\boldsymbol{f}_7 = \boldsymbol{f}_5(\boldsymbol{f}_3, \boldsymbol{f}_2) = (aba)(ab)(aba)(aba)(ab)(aba)(ab)(aba).$$

Лемма 3.4.2 применима и в бесконечном случае:

$$\boldsymbol{f}_\infty = \boldsymbol{f}_\infty(\boldsymbol{f}_{n+1}, \boldsymbol{f}_n) = \boldsymbol{f}_{n+1}\boldsymbol{f}_n\boldsymbol{f}_{n+1}\boldsymbol{f}_{n+1}\boldsymbol{f}_n\boldsymbol{f}_{n+1}\boldsymbol{f}_n\boldsymbol{f}_{n+1}\cdots. \tag{3.27}$$

Это равенство используется в упражнении 3.4.11 для доказательства двух следующих лемм.

Лемма 3.4.3. Для любого целого $n \geqslant 2$ выполняется равенство $\boldsymbol{f}_n^2 = \boldsymbol{f}_{n+1}\boldsymbol{f}_{n-2}$. ■

Лемма 3.4.4. Для любого целого $n \geqslant 3$ строка $\boldsymbol{f}_n$ имеет грани $\boldsymbol{f}_i$ для $i = n-2, n-4, \ldots, 2-(n \bmod 2)$. ■

Из этих лемм и свойства сохранения префиксов в строках Фибоначчи следует, что при $n \geqslant 3$ любое вхождение строки $\boldsymbol{f}_{n+1}$ в строку $\boldsymbol{f}_\infty$ порождает в этой строке квадрат $\boldsymbol{f}_n^2$. Строка $\boldsymbol{f}_n$ может располагаться или после подстроки $\boldsymbol{f}_{n+1}\boldsymbol{f}_n$ или после подстроки $\boldsymbol{f}_{n+1}^2 = \boldsymbol{f}_{n+1}\boldsymbol{f}_n\boldsymbol{f}_{n-1}$. Повторное применение леммы 3.4.1 показывает, что

$$\boldsymbol{f}_n\boldsymbol{f}_{n+1}\boldsymbol{f}_n = \boldsymbol{f}_n^3\boldsymbol{f}_{n-3}\boldsymbol{f}_{n-2}. \tag{3.28}$$

Таким образом, любое вхождение строки $\boldsymbol{f}_n$ в выражение (3.27) возможно только в виде куба $\boldsymbol{f}_n^3$. Кроме того, для любого $n \geqslant 3$ строки $\boldsymbol{f}_n$ и $\boldsymbol{f}_n^2$ в действительности будут *оболочками* строки $\boldsymbol{f}_\infty$ в том смысле, как они (оболочки) определены в формулировке задачи 2.18. Поскольку $\boldsymbol{f}_n$ — оболочка $\boldsymbol{f}_\infty$, тогда любое другое[2]

[2]Здесь имеются в виду вхождения строки $\boldsymbol{f}_n$ в строку $\boldsymbol{f}_\infty$, отличные от квадратов и кубов. — *Примеч. ред.*

вхождение строки $\boldsymbol{f}_n$ в строку $\boldsymbol{f}_\infty$, которое не содержится в кратных подстроках выражения (3.27), должно быть некоторым циклическим сдвигом $R_j(\boldsymbol{f}_n)$, $j > 0$. Но по теореме 1.4.2 это возможно только в случае, если $\boldsymbol{f}_n$ является кратной строкой, что, конечно, не соответствует действительности. Таким образом, мы рассмотрели все возможные случаи вхождения строки $\boldsymbol{f}_n$ (и, следовательно, $\boldsymbol{f}_n^2$ и $\boldsymbol{f}_n^3$) в строку $\boldsymbol{f}_\infty$.

Зная о вхождениях $\boldsymbol{f}_n^3$ в строку $\boldsymbol{f}_\infty$, можем сделать предположение о существовании квадратов любого циклического сдвига $R_j(\boldsymbol{f}_n)$, $j > 0$. Это предположение доказано в упражнении 3.4.12.

Теперь мы можем показать, что в строке $\boldsymbol{f}_\infty$ не существует четвертых степеней от $\boldsymbol{f}_n$. На основе леммы 3.4.1 можно записать

$$\boldsymbol{f}_n^4 = \boldsymbol{f}_n^3 \boldsymbol{f}_{n-2} \boldsymbol{f}_{n-3} \boldsymbol{f}_{n-2}.$$

Сравнивая это выражение с выражением (3.28), делаем заключение, что $\boldsymbol{f}_n^4$ может существовать только тогда, когда выполняется равенство $\boldsymbol{f}_{n-3}\boldsymbol{f}_{n-2} = \boldsymbol{f}_{n-2}\boldsymbol{f}_{n-3}$. Как показано в упражнении 3.4.13, такое равенство невозможно, поэтому $\boldsymbol{f}_n^4$ не существует.

Далее встает естественный вопрос: можно ли утверждать, что *все* квадраты в строке $\boldsymbol{f}_\infty$ являются квадратами конечных строк Фибоначчи или их циклических сдвигов? Для ответа на этот вопрос расширим понятие обратного морфизма, введенного в разделе 3.1. Там обратное отображение было определено только на множестве строк $h^*(A) - A$. В случае строк Фибоначчи мы можем определить обратное отображение h^{-1} на более коротких подстроках, если учтем, что подстрока ab может появиться в строке $\boldsymbol{f}_\infty$ только как результат отображения $h : a \to ab$, тогда как подстрока aa может появиться только в том случае, если первая буква a является самым левым элементом строковой последовательности и получена в результате отображения $h : b \to a$. С учетом этих замечаний определим обратный морфизм для строк Фибоначчи следующим образом:

$$h^{-1} : ab \to a, \quad a \not\to b, \tag{3.29}$$

где запись $a \not\to b$ означает, что отображение $a \to b$ возможно только в том случае, если за буквой a не следует (справа) буква b (другими словами, отображение $a \to b$ применяется только тогда, когда за буквой a следует другая буква a либо эта буква a является самым правым элементом строки). Поскольку морфизм Фибоначчи (3.4) не может породить подстроки bb, поэтому отображение (3.29) действительно определяет алгоритм, который должен выполняться над строкой $\boldsymbol{f}_n$ слева направо, проверяя каждое вхождение буквы a и выполняя обратное отображение h^{-1} в зависимости от того, какая буква следует за данной буквой a. Поскольку этот алгоритм порождает строку, "предшествующую" строке $\boldsymbol{f}_n$, то отсюда заключаем, что морфизм Фибоначчи (3.4) является кодом.

Отметив, что подстрока a^3 не может появиться в строках Фибоначчи (см. упражнение 3.4.9), мы можем сформулировать следующее утверждение.

Теорема 3.4.5. Квадрат $\boldsymbol{u}^2$ может входить в строку $\boldsymbol{f}_\infty$ только в том случае, если строка $\boldsymbol{u}$ является циклическим сдвигом строки $\boldsymbol{f}_n$ при некотором целом $n \geqslant 0$.

Доказательство. Мы уже видели, что для любого циклического сдвига $R_j(\boldsymbol{f}_n)$ строки $\boldsymbol{f}_n$, где $j \in 0..f_n - 1$, строка $\boldsymbol{u}^2$ обязательно входит в строку $\boldsymbol{f}_\infty$. Это доказывает достаточность утверждения теоремы.

Для доказательства необходимости утверждения теоремы предположим существование в строке $\boldsymbol{f}_\infty$ квадрата $\boldsymbol{u}^2$, где строка $\boldsymbol{u}$ не будет сдвигом ни какой строки $\boldsymbol{f}_n$. Рассмотрим минимальную из таких строк $\boldsymbol{u}$. Пусть n — наименьшее целое, такое, что $\boldsymbol{u}$ является собственной подстрокой строки $\boldsymbol{f}_n$. Далее покажем, что при этих предположениях строка $h^{-1}(\boldsymbol{f}_n) = \boldsymbol{f}_{n-1}$ содержит квадрат $\boldsymbol{v}^2$, где $|\boldsymbol{v}| < |\boldsymbol{u}|$ и строка $\boldsymbol{v}$ не является сдвигом ни какой строки Фибоначчи. Это противоречие докажет утверждение теоремы.

Возможны четыре варианта строки $\boldsymbol{u}$.

- $\boldsymbol{u} = b \cdots b$. Этот вариант не реализуем, поскольку в этом случае строка $\boldsymbol{u}^2$ будет содержать подстроку bb.
- $\boldsymbol{u} = a \cdots b$. Этот вариант также не реализуем, поскольку в данном случае строка $\boldsymbol{u}$ обязана иметь вид $\boldsymbol{u} = a \cdots ab$, и тогда для строки $\boldsymbol{v} = h^{-1}(\boldsymbol{u})$ будет выполняться неравенство $|\boldsymbol{v}| < |\boldsymbol{u}|$.
- $\boldsymbol{u} = a \cdots a$. В этом случае $\boldsymbol{u}^2 = a \cdots aa \cdots a$, и поскольку подстрока a^3 невозможна, имеем $\boldsymbol{u}^2 = a \cdots baab \cdots ba$. Так как и подстрока bb невозможна, то с неизбежностью $\boldsymbol{u} = aba \cdots aba$. Тогда для строки $\boldsymbol{v} = h^{-1}(\boldsymbol{u})$ будет выполняться неравенство $|\boldsymbol{v}| < |\boldsymbol{u}|$.
- $\boldsymbol{u} = b \cdots a$. Поскольку подстрока bb невозможна, то строка $\boldsymbol{u}$ должна располагаться после буквы a. В этом случае существует подстрока

$$ab \cdots \lambda ab \cdots \lambda a = (ab \cdots \lambda)^2 a$$

некоторой буквы $\lambda \in \{a, b\}$. Таким образом, этот вариант сводится к варианту $\boldsymbol{u} = a \cdots a$ или $\boldsymbol{u} = a \cdots b$.

Теорема доказана. ■

Из этой теоремы сразу получаем, что ни один циклический сдвиг любой строки Фибоначчи не может входить в $\boldsymbol{f}_\infty$ в четвертой степени, поскольку

$$(R_j(\boldsymbol{f}_n))^4 = ((R_j(\boldsymbol{f}_n))^2)^2,$$

но, как мы видели выше, квадрат $(R_j(\boldsymbol{f}_n))^2$ не может быть строкой Фибоначчи.

Отметим, что строка $\boldsymbol{f}_7$ содержит куб $(aba)^3$. Поэтому имеет место следующая теорема.

Теорема 3.4.6. Все строки Фибоначчи $\boldsymbol{f}_n$ при $n \geqslant 7$ содержат кубы, но ни одна строка Фибоначчи не содержит кратных подстрок четвертого порядка. ■

Мы показали в разделах 3.2 и 3.3, что если $\boldsymbol{f}_\infty$ содержит кратные подстроки четвертого порядка, тогда такие же подстроки содержат и строки $\boldsymbol{f}_n$. Но по теореме 3.4.6 строки $\boldsymbol{f}_n$ не могут содержать кратные подстроки четвертого порядка, поэтому справедлива следующая теорема.

Теорема 3.4.7. Бесконечная строка Фибоначчи $\boldsymbol{f}_\infty$ является решением задачи построения $(2, 4)$-исключения. ■

Теперь мы располагаем достаточными данными для получения оценки числа кратных подстрок, содержащихся в строке $\boldsymbol{f}_n$. Чтобы получить приближенную нижнюю границу для этого числа, заметим, что для любого целого $k \in 3..n-4$ количество кратных подстрок определяется вхождениями строк $\boldsymbol{f}_k$ в строку $\boldsymbol{f}_n$. Однако при этом в конечную сумму не включаются квадраты a^2 и $(ab)^2$, что, как покажем ниже, не влияет на асимптотику оценок.

В упражнении 3.4.15 показано, что буква bf_{n-2} раз входит в строку $\boldsymbol{f}_n$. Тогда из леммы 3.4.2 следует, что строка $\boldsymbol{f}_k f_{n-k-2}$ раз входит в $\boldsymbol{f}_n$. Поскольку только одно из этих вхождений является суффиксом строки $\boldsymbol{f}_n$, значит, не менее $f_{n-k-2}-1$ вхождений строки $\boldsymbol{f}_k$ порождают кубы $\boldsymbol{f}_k^3$. Далее отметим, что каждый куб порождает по крайней мере $\boldsymbol{f}_k$ различных квадратов $(R_j(\boldsymbol{f}_k))^2$ $(j = 0, 1, \ldots, f_k - 1)$, формируемых из циклических сдвигов строки $\boldsymbol{f}_k$. Следовательно, общее число кратных подстрок, определяемых $f_{n-k-2}-1$ вхождениями кубов $\boldsymbol{f}_k^3$ в строку $\boldsymbol{f}_n$, не меньше величины $f_k(f_{n-k-2}-1)$.

Ранее мы уже показали, что $f_n \in \Theta(\phi^n)$, поэтому $f_k f_{n-k-2} \in \Theta(\phi^k \phi^{n-k-2})$ и, независимо от значения k,

$$f_k(f_{n-k-2}-1) \in \Theta(f_{n-2}). \tag{3.30}$$

Поскольку оценка (3.30) справедлива для любого $k \in 3..n-4$, то отсюда мы делаем заключение, что в строке $\boldsymbol{f}_n$ содержится не менее $\Theta((n-2)f_{n-2})$ кратных подстрок. Так как $n \approx \log_\phi f_n$ и $f_{n-2} \approx \boldsymbol{f}_n/\phi^2$, приходим к следующей теореме.

Теорема 3.4.8. Количество $q(n)$ кратных подстрок в строке $\boldsymbol{f}_n$ имеет порядок $\Omega(f_n \log f_n)$. ■

Поскольку, как упоминалось ранее, существуют алгоритмы (см. раздел 12.1), вычисляющие все кратные подстроки в любой строке длиной n за время $\Theta(n \log n)$, тогда нижняя граница из теоремы 3.4.8 будет также верхней границей и, кроме того, даст асимптотически наибольшее число кратных подстрок, которые могут содержаться в любой строке длиной n.

В начале этого раздела уже говорилось о том, что несмотря на доказанную оценку из теоремы 3.4.8, "действительное" число кратных подстрок в строках

$\boldsymbol{f}_n$ имеет линейный порядок $\Theta(f_n)$. Это не противоречие, если вспомнить обсуждения 1.3.2 и 2.3.1: кодирование информации — это ключ к эффективности строковых алгоритмов.

Вернемся к рассуждениям, приведенным выше. Очевидно, что описать все кратные подстроки способна кодировка в виде тройки (i, p^*, r^*), введенной в разделе 2.3: для любой позиции i, с которой начинает квадрат или куб подстроки $R_j(\boldsymbol{f}_k)$, присутствует одна кратная строка — $(i, f_k, 2)$ или $(i, f_k, 3)$. Но, как показано в наших рассуждениях выше, каждый куб $\boldsymbol{f}_k^3$ порождает дополнительно $f_k - 1$ кратных строк, которые, впрочем, нет необходимости описывать точно. Рассмотрим, например, кратную подстроку $\boldsymbol{f}_4^3 = (abaab)^3$, которая начинается с позиции 9 в строке

$$\boldsymbol{f}_8 = abaababa\underline{abaababaababaaba}ababaabaab.$$

Этот куб и все кратные строки, входящие в его циклические сдвиги, можно описать с помощью ***серий***, кодируемых "четверками", введенными в разделе 2.3:

$$(i, p^*, r^*, t) = (9, 5, 3, 1),$$

где конечный элемент $t = 1$ показывает общее количество циклических сдвигов строки $\boldsymbol{f}_5$, исключая саму строку $\boldsymbol{f}_5$, которая является кубом. В данном случае четверка $(9, 5, 3, 1)$ определяет кубы $(abaab)^3$ и $(baaba)^3$ и квадраты $(abaab)^2$, $(ababa)^2$ и $(babaa)^2$. *Предлагаю доказать читателю*, что с помощью четверок (i, p^*, r^*, t) можно описать серии кубов и квадратов в любой строке $\boldsymbol{f}_n$.

Итак, для определения всех вхождений кратных подстрок строки $\boldsymbol{f}_k$ в строку $\boldsymbol{f}_n$ ($k = 1, 2, \ldots, n-2$) необходимо получить значения только одной четверки (i, p^*, r^*, t). Как мы уже видели, строка $\boldsymbol{f}_k$ входит в строку $\boldsymbol{f}_n$ f_{n-k-2} раз. Поэтому для строки $\boldsymbol{f}_n$ необходимо определить следующее количество четверок:

$$q'(n) = \sum_{k=1}^{n-2} f_{n-k-2} = \sum_{k=0}^{n-3} f_k.$$

По индукции легко показать, что $\sum_{k=0}^{n-3} f_k < f_{n-1}$. Это доказывает такую теорему.

Теорема 3.4.9. Количество $q'(n)$ серий кратных подстрок в строке $\boldsymbol{f}_n$ не превышает f_{n-1}. ■

Мы снова вернулись к нашей интересной ситуации. Теорема 3.4.8 показывает, что в строках Фибоначчи очень много кратных строк. Несмотря на этот факт, теорема 3.4.9 говорит, что если эти кратные строки "запаковать" в серии, то количество серий будет иметь линейный порядок относительно длины строки. Здесь встает закономерный вопрос для любопытных: для любой ли строковой последовательности $\boldsymbol{x}$ можно представить все ее кратные подстроки в виде линейного (по объему) множества серий? Мы вернемся к этому интересному вопросу в главе 12.

Отметим, что *обе* кодировки (i, p^*, r^*) и (i, p^*, r^*, t), определяющие кратные подстроки путем указания их позиции i в строке, требуют представления строки в виде массива, как сказано в обсуждении 1.2.1.

В заключение этого раздела рассмотрим применение отображения (3.24) для точной локализации квадратов $\boldsymbol{f}_n^2$ в строке $\boldsymbol{f}_\infty$ для любого целого $n \geqslant 3$. Покажем, что для этого можно построить алгоритм, который вычисляет все кратные подстроки в строках Фибоначчи за линейное время. Положим в формулах (3.24) $\boldsymbol{x} = f_{n+1}$, $\boldsymbol{y} = f_n$ для любого неотрицательного целого n. Теперь $\boldsymbol{x}$ и $\boldsymbol{y}$ являются строками на алфавите, состоящем из чисел Фибоначчи. Рассмотрим строку $\boldsymbol{f}_\infty(f_{n+1}, f_n)$, например, для $n = 3$:

$$\boldsymbol{f}_\infty(f_4, f_3) = 535535355355353553 \cdots . \tag{3.31}$$

Обозначим через $\Sigma_{i,n}$ сумму первых i значений в строке $\boldsymbol{f}_\infty(f_{n+1}, f_n)$. Очевидно, что $\Sigma_{0,n} = 0$ для любого n, а для примера (3.31) имеем

$$\Sigma_{1,3} = 5, \quad \Sigma_{2,3} = 8, \quad \Sigma_{5,3} = 21, \quad \Sigma_{9,3} = 46.$$

Используя эти обозначения, можно указать позиции в строке $\boldsymbol{f}_\infty$, где находятся строки $\boldsymbol{f}_{n+1}$ и f_n^2. Сначала докажем следующую теорему.

Теорема 3.4.10. Для любого целого $i \geqslant 0$

а) $\boldsymbol{f}_{n+1} = \boldsymbol{f}_\infty[\Sigma_{i,n} + 1..\Sigma_{i,n} + f_{n+1}]$, $n \geqslant 2$,

б) $\boldsymbol{f}_n^2 = \boldsymbol{f}_\infty[\Sigma_{i,n} + 1..\Sigma_{i,n} + 2f_n]$, $n \geqslant 3$.

Доказательство. Поскольку строка $\boldsymbol{f}_\infty$ свободна от квадратов b^2, тогда любое вхождение строки f_n в выражение (3.27) следует за вхождением строки $\boldsymbol{f}_{n+1} = \boldsymbol{f}_{n-1}\boldsymbol{f}_{n-2}\boldsymbol{f}_{n-1}$, т.е. строка $\boldsymbol{f}_{n+1}$ присутствует при каждом вхождении строки $\boldsymbol{f}_n$ в строку $\boldsymbol{f}_\infty$. Поэтому утверждение а) теоремы следует непосредственно из леммы 3.4.2 и равенства (3.27). Вследствие леммы 3.4.3 и того факта, что при $n \geqslant 3$ строка $\boldsymbol{f}_{n-2}$ является префиксом как строки $\boldsymbol{f}_{n+1}$, так и строки $\boldsymbol{f}_n$, делаем заключение, что квадрат $\boldsymbol{f}_n^2$ встретится в тех же позициях, что и строки $\boldsymbol{f}_n$ и $\boldsymbol{f}_{n+1}$. Это доказывает утверждение б) теоремы. ■

Некоторым "недостатком" последней теоремы можно считать условие $n \geqslant 3$ в утверждении б), что не позволяет локализовать квадраты $\boldsymbol{f}_1^2 = a^2$ и $\boldsymbol{f}_2^2 = (ab)^2$. Однако следующий результат показывает, что эти квадраты также учитываются.

Лемма 3.4.11. Для $n = 1, 2$ квадраты $\boldsymbol{f}_n^2$ входят в строку $\boldsymbol{f}_\infty$ только как подстроки строк $\boldsymbol{f}_{n+2}^2$.

Доказательство этой леммы предложено дать в упражнении 3.4.16. ■

С учетом последней леммы теорема 3.4.10 позволяет эффективно локализовать все вхождения квадратов $\boldsymbol{f}_n^2$ в строку $\boldsymbol{f}_\infty$ для любого $n \geqslant 1$. Как мы видели, вхождения строк f_n в выражение (3.27) порождает кубы $\boldsymbol{f}_n^3$ и, следовательно, квадраты и кубы сдвигов $R_j(\boldsymbol{f}_k)$ ($j = 0..n-1, n \geqslant 3$). Использование для серий кодировки (i, p^*, r^*, t) позволяет вычислить все кратные подстроки в строке $\boldsymbol{f}_n$, причем за линейное время. Алгоритм, реализующий этот подход, описан в [123]. В работе [89] приведены точные формулы для количества квадратов и количества *различных* квадратов в строках Фибоначчи. Отметим работу [139], в которой доказан несколько неожиданный результат, что количество серий (см. обсуждение 2.3.3) в строках Фибоначчи меньше количества различных квадратов в этих строках.

Обсуждение 3.4.1. Строки Фибоначчи обладают многими замечательными свойствами, но, наверное, еще более замечательным фактом является то, что большинство важных свойств этих строк можно обобщить с помощью строк Штурма.

Строкой Штурма (Sturmian string) называется бесконечная строковая последовательность, в которой для любого целого $k \geqslant 0$ существует $k+1$ различных подстрок длиной k. Строки Штурма строятся на бинарном алфавите. Любой конечный префикс бесконечной строки Штурма также называется строкой Штурма. Значение $k+1$ часто называют ***сложностью*** строки. Бесконечная строка $\boldsymbol{x}$ имеет сложность k только тогда, когда она представима в виде $\boldsymbol{x}[1..k]^\infty$. Поэтому строки Штурма являются строками наименьшей возможной сложности.

Строки Штурма представляют большой интерес. Поскольку они имеют малую сложность, то естественно предположить, что они содержат много кратных подстрок. И это действительно так [92]: все конечные строки Штурма длиной n содержат $\Theta(n \log n)$ кратных подстрок, которые можно представить в виде $\Theta(n)$ серий, вычисляемых за время порядка $\Theta(n)$ (как и в случае строк Фибоначчи). Более того, для любого целого $r \geqslant 4$ существует строка Штурма, которая является решением задачи построения $(2, r)$-исключения, точно так же как строки Фибоначчи являются решением задачи построения $(2, 4)$-исключений.

Подобно строкам Фибоначчи, строки Штурма можно определить с помощью морфизма, при использовании которого можно установить различные свойства этих строк [92]. Этот морфизм также позволяет распознать за линейное время, является ли данная строка строкой Штурма [37]. В недавней работе [91] показано, что эти свойства распространяются на обобщение строк Штурма, называемых *двух-паттерными*.

Строки Штурма можно определить разными способами, которые показывают их с разных сторон; им посвящено много работ [9, 34, 33, 162, 166, 180, 178, 215]. Существуют другие классы строк с малой сложностью, изучаемые в работах [179, 10, 197]. ■

Упражнения 3.4

1. Мы упоминали строки Фибоначчи в двух предыдущих главах. Какие свойства данных строк использовались в примерах этих глав?

2. Дайте другое определение строк $\boldsymbol{f}_n$ для $n \geqslant 2$ на основе понятий "суффикс" и "префикс". Затем докажите лемму 3.4.2.

3. Предложите рекурсивный алгоритм вычисления первых 20 строк Фибоначчи, построенный на основе соотношений (3.18). Затем исключите рекурсию и напишите "обычный" итерационный алгоритм. Какой из этих алгоритмов проще для понимания?

4. Методом математической индукции докажите, что $f_n \leqslant \phi^n$. Отсюда из неравенства (3.23) будет следовать, что f_n имеет порядок $\Theta(\phi^n)$.

5. Докажите точную формулу $f_n = \left(\phi^{n+1} - (1-\phi)^{n+1}\right)/\sqrt{5}$. Эту формулу также можно переписать в виде $f_n = \left[\phi^{n+1}/\sqrt{5}\right]$, где $[x]$ обозначает целую часть числа x.

6. ***Обратные строки Фибоначчи*** определяются с помощью соотношений

$$\boldsymbol{f}'_0 = b, \quad \boldsymbol{f}'_1 = a, \quad \boldsymbol{f}'_n = \boldsymbol{f}'_{n-2}\boldsymbol{f}'_{n-1}, \qquad \forall n \geqslant 2.$$

Покажите, что морфизм $h' : a \to ba$, $b \to a$ порождает обратные строки Фибоначчи при начальной строке $\boldsymbol{f}'_0 = b$.

7. Докажите, что $h(\boldsymbol{f}_\infty) = \boldsymbol{f}_\infty$.

8. Покажите, что строка $\boldsymbol{f}_n$ не содержит собственных префиксов b, префиксов aa и суффиксов aa.

9. Покажите, что строка $\boldsymbol{f}_n$ не содержит подстрок b^2, a^3, $(ab)^3$ и $(ba)^3$.

10. Предложите алгоритм, реализующий "обратный морфизм" (3.29) для строк $\boldsymbol{f}_n$.

11. **а)** Докажите лемму 3.4.3.

б) Докажите лемму 3.4.4. Можно ли доказать более сильный результат (подобный лемме 3.2.3), что других граней, кроме указанных в условии леммы, в строках $\boldsymbol{f}_n$ нет?

в) Используя леммы 3.4.3 и 3.4.4, докажите утверждение, сделанное в тексте, что любое вхождение строки $\boldsymbol{f}_{n+1}$ (соответственно, $\boldsymbol{f}_n$) в строку $\boldsymbol{f}_\infty$ порождает квадрат $\boldsymbol{f}_n^2$ (соответственно, $\boldsymbol{f}_n^3$).

12. Покажите, что любое вхождение куба $\boldsymbol{f}_n^3$ в строку $\boldsymbol{f}_\infty$ порождает квадраты $(R_j(\boldsymbol{f}_k))^2$ для любого целого $j = 1, \ldots, f_k - 1$.

13. Докажите, что $\boldsymbol{f}_{n-1}\boldsymbol{f}_n \neq \boldsymbol{f}_n\boldsymbol{f}_{n-1}$.

14. В тексте мы "бездоказательно" утверждали, что нет строк Фибоначчи, которые были бы кратными строками. Докажите или опровергните это утверждение.

15. **а)** Докажите, что в строке $\boldsymbol{f}_n$ содержится f_{n-1} букв a и f_{n-2} букв b.

б) Докажите, что f_n будет четным числом только тогда, когда $n = 3k + 2$ для некоторого неотрицательного целого k.

в) Используя утверждения а) и б), предложите другое доказательство предыдущего упражнения.

16. Докажите лемму 3.4.11.

17. Рассуждения, ведущие к теореме 3.4.8, можно значительно сократить, если установить более точную нижнюю границу для числа кратных подстрок, присутствующих в строке f_n. Можете ли вы получить *точную* формулу для числа кратных подстрок в строке f_n? (Попробуйте использовать для этого лемму 3.4.11!)

18. Аналогично предыдущему упражнению, попробуйте усовершенствовать доказательство теоремы 3.4.9 путем вывода точной формулы для числа серий кратных подстрок, присутствующих в строке f_n.

19. Чтобы локализовать все кратные подстроки в строке $\boldsymbol{f}_\infty$, предложите алгоритм, который бы за фиксированное время находил значение $\boldsymbol{f}_\infty[i]$ для любого заданного целого $i > 0$. Можете ли вы предложить формулу для вычисления этого значения?

20. Покажите, как можно построить свободную от кратных подстрок строку длиной 2^k на алфавите из $k + 1$ букв.

21. Покажите, что "морфизм Штурма" $h' : a \to aab$, $b \to ab$ порождает строки $\boldsymbol{f}'_n = R_{n-1}(\boldsymbol{f}_n)$, n нечетно, для начальной строки $\boldsymbol{f}'_1 = a$.

22. Покажите, что если для некоторого целого $k > 0$ строка $\boldsymbol{x}$ имеет сложность k, тогда $\boldsymbol{x} = \boldsymbol{x}[1..k]^r$ для некоторого целого $r \geqslant 2$.

ГЛАВА 4

Строковые алгоритмы и тестовые данные

Используйте мягкие слова и твердые аргументы.

— Аноним

В данной книге основное внимание уделяется алгоритмам над строковыми последовательностями. Поэтому естественно перед обсуждением специальных строковых алгоритмов рассмотреть некоторые наиболее общие свойства таких алгоритмов. Этой теме посвящен раздел 4.1. Одним из обсуждаемых свойств алгоритмов является их корректность, что обычно проверяется на определенных тестовых данных. Специальные строковые последовательности, введенные в главе 3, будут прекрасными тестовыми данными для некоторых строковых алгоритмов, однако также необходимы тестовые данные и более общего вида. В разделах 4.2 и 4.3 показаны практичные и эффективные методы генерирования множеств тестовых данных, эти методы разработаны специально для доказательного тестирования определенных строковых алгоритмов. Некоторые из этих методов генерирования тестовых данных реализованы в экспериментальных пакетах программного обеспечения [161].

4.1 Хорошие строковые алгоритмы

В этом разделе мы рассмотрим следующие шесть свойств строковых алгоритмов, которые определяют их практическую полезность.

- Корректность.
- Скорость вычислений и требуемая память (временна́я и пространственная сложность алгоритма).
- Отсутствие возврата к ранее просмотренным данным.
- Независимость от алфавита.
- Возможность предварительной подготовки данных.
- Кодирование выходного результата.

Корректность

Нет сомнения, что корректность является необходимым свойством любого алгоритма, — некорректный алгоритм бесполезен. В худшем случае некорректный алгоритм, реализованный в компьютерной программе, может быть просто опасным, если, например, он управляет тормозной системой автомобиля или полетом самолета. Доказательство корректности программных продуктов — даже если есть четкое определение понятия корректности — большая нерешенная проблема "инженерии программного обеспечения". Для математических алгоритмов, представленных в книге, корректность доказывается относительно легко по сравнению с доказательствами корректности для многих других типов алгоритмов, но это не означает, что эти доказательства совсем простые. Несколько лет назад один научный работник реализовал на языке C четыре ранее опубликованных алгоритма для вычисления канонических форм петель (см. определение 1.4.3); среди этих алгоритмов был один мой. Обнаружилось, что три из этих алгоритмов, включая и мой, оказались некорректными!

Доказательство корректности алгоритма проходит два основных этапа. На первом этапе предлагается математическое "доказательство" корректности; на втором этапе выполняется проверка алгоритма на тестовых входных данных, для которых известен правильный выходной результат. Доказательства корректности, выполненные на каждом из этих этапов, ненадежны. Математические доказательства ненадежны из-за "человеческого фактора", а выполнение алгоритма на тестовых данных не может доказать "полную" корректность, поскольку тестовые данные никогда не бывают исчерпывающими. Однако лучше иметь ненадежные средства, чем не иметь средств вообще. В разделах 4.2 и 4.3 я покажу, как можно генерировать полные тестовые данные, подходящие для многих строковых алгоритмов, в виде строковых последовательностей с широким диапазоном их длин и при этом с невысокой стоимостью самого процесса генерирования данных.

Временная и пространственная сложность

Временна́я и пространственная сложность — основные характеристики любого алгоритма, и, конечно, алгоритм, время выполнения которого в наихудшем случае

имеет порядок $\Theta(n)$ на строках длиной n, предпочтительнее алгоритма с временем выполнения в наихудшем случае порядка $O(n^2)$. Но не все так очевидно и ясно с этими характеристиками. Например, простой алгоритм 2.2.1, вычисляющий вхождения частного паттерна длиной m в строку длиной n и описанный в разделе 2.2, имеет время выполнения, не лучшее, чем у какого-нибудь плохого алгоритма. Однако для этого алгоритма временна́я граница $O(mn)$ достигается только на одном патологическом варианте строковых последовательностей, который не встречается на практике, тогда как на "практических" строках время выполнения имеет порядок $O(m+n)$. Таким образом, этот алгоритм фактически является "хорошим" алгоритмом, поскольку он прост и в среднем выполняется быстрее, чем алгоритм КМП (см. раздел 7.1). Кроме того, он элегантно обходит возможные возвраты к просмотренным ранее позициям в строке для их повторной проверки.

Пример вычисления деревьев суффиксов (см. раздел 2.1), требующий времени порядка $\Theta(n \ln \alpha)$ на алфавите объемом α и памяти порядка $\Theta(n\alpha)$, показывает, что "хороший" порядок относительно n не гарантирует быстрого выполнения или малого требуемого объема памяти, поскольку константы пропорциональности в асимптотических оценках могут быть чрезмерно большими, особенно для больших алфавитов. Аналогично тщательно разработанные алгоритмы сравнения с паттернами, где сведено к теоретическому минимуму количество парных буквенных сравнений [97, 52], почти не используются на практике, поскольку они чрезвычайно сложны, вследствие чего усложняется процесс вычисления, что, в свою очередь, приводит к увеличению значений констант пропорциональности в их асимптотических оценках.

Если сравнивать временну́ю и пространственную сложности алгоритмов, то временна́я сложность считается более важным критерием при выборе алгоритма. Это отображает тот факт, что стоимость хранения единицы данных обычно ниже стоимости единицы времени процесса вычислений. Таким образом, временна́я сложность алгоритма часто "перевешивает" все другие характеристики алгоритма.

Отсутствие возврата к ранее просмотренным данным

Просмотр строковых последовательностей естественно выполнять слева направо. Поэтому чрезвычайно привлекательным свойством строковых алгоритмов считается их ***онлайновость***. Это означает, что если в процессе вычислений уже просмотрена позиция i в строковой последовательности, то алгоритм уже не вернется к этой позиции i снова. Другими словами, онлайновый алгоритм никогда ***не выполняет возврата*** к уже просмотренным позициям.

Существует также более слабое определение онлайновых алгоритмов. Например, алгоритм называется онлайновым с ***окном*** размера k, если существует такая положительная константа k, что для выполнения вычислений в текущей позиции i требуется не более k строковых позиций из интервала $i-k+1..i$. Еще одно по-

добное слабое определение позволяет считать алгоритм онлайновым, если значение, вычисляемое алгоритмом для любой позиции i, никогда не пересчитывается, даже если потребуется заново обратиться к позициям $i' < i$.

Из любого сделанного выше определения онлайновости следует, что строковый онлайновый алгоритм при решении определенной задачи для строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ эффективно решает ту же задачу для каждого префикса $\boldsymbol{x}[1..i]$ этой строки. Таким образом, можно сказать, что онлайновый алгоритм фактически решает n задач по цене одной задачи.

Онлайновым алгоритмом является алгоритм КМП, упоминавшийся выше. Отметим, что алгоритм 1.3.1 вычисления массива граней также является онлайновым (в слабом определении этого понятия): массив граней $\boldsymbol{\beta}[1..i]$ для подстроки $\boldsymbol{x}[1..i]$ вычисляется после просмотра значения $\boldsymbol{x}[i]$, и далее этот массив не пересчитывается.

Независимость от алфавита

В разделе 1.2 показано, что строковые последовательности в общем случае могут быть определены на произвольных множествах элементов. Поэтому желательно, чтобы строковые алгоритмы эффективно выполнялись на строках, определенных на любых алфавитах, т.е. чтобы они были независимы от алфавита. Таким алгоритмом является, например, алгоритм 1.3.1 вычисления массива граней — он имеет асимптотически оптимальное линейное время выполнения при условии, что сравнение (равны или не равны) двух любых букв алфавита выполняется за константное время, и при этом алгоритм не использует никакие другие свойства алфавита.

С другой стороны, часто можно уменьшить временну́ю сложность алгоритма, если использовать некоторые особые свойства алфавита. Поэтому если данное свойство алфавита часто встречается на практике, то разрабатываются специальные версии алгоритма, которые оказываются более эффективными (по сравнению с исходным алгоритмом) на строковых последовательностях, определенных на таких алфавитах. Чтобы более точно описать эту проблему, рассмотрим следующую простую задачу.

Задача 4.1 (Подсчет вхождений). Подсчитать количество вхождений каждой буквы в данную строку $\boldsymbol{x}$ длиной n. ■

Рассмотрим эту задачу для трех различных алфавитов A.

- ***Произвольный (неупорядоченный) алфавит***: для произвольных букв λ, $\mu \in A$ равенство $\lambda = \mu$ проверяется за фиксированное время (примером такого алфавита может служить множество китайских иероглифов).
- ***Упорядоченный алфавит***: для произвольных букв λ, $\mu \in A$ некоторое определенное отношение упорядочения $\lambda < \mu$ проверяется за фиксированное

время (примером такого алфавита может служить множество действительных чисел или множество слов английского языка).

- ***Индексированный алфавит***: определен массив T, такой, что для произвольной буквы $\lambda \in A$ значение $T[\lambda]$ можно получить за константное время (примером такого алфавита может служить подмножество символов ASCII).

Отметим, что индексированный алфавит A обязательно должен быть конечной мощности $\alpha = |A|$. Тогда на индексированном алфавите задачу 4.1 можно решить непосредственным подсчетом: сначала целочисленный массив $T[1..n]$ полагается равным нулю, затем для каждой позиции $i \in 1..n$ входной строки $\boldsymbol{x}$ значение $T[\boldsymbol{x}[i]]$ должно увеличиваться на единицу. Имея дополнительный список кодировки букв, ненулевые элементы массива T можно вывести за время, пропорциональное количеству различных букв в строке $\boldsymbol{x}$. Таким образом, для решения задачи требуется время порядка $\Theta(\alpha + n)$.

В случае упорядоченного алфавита можно построить сбалансированное дерево поиска (такое, как АВЛ-дерево[1] или 2-3-дерево[2]) путем просмотра входной строки $\boldsymbol{x}$ слева направо: каждый узел дерева идентифицируется парой (λ, c), где λ — буква из строки $\boldsymbol{x}$, а c — счетчик, который принимает значение 1 при создании узла и затем увеличивается на единицу при нахождении каждого нового вхождения буквы λ в строку $\boldsymbol{x}$. Время, необходимое для решения задачи 4.1, в данном случае имеет порядок $O(n \log n)$. Отметим, что если объем алфавита α значительно больше величины n, то такой алгоритм может выполняться быстрее, чем соответствующий алгоритм для строк, определенных на индексированном алфавите.

В случае неупорядоченного алфавита необходимо выполнить $\Theta(n)$ шагов, на каждом из которых следует выполнить порядка $\Theta(n)$ попарных сравнений букв. Поэтому решение задачи 4.1 в данном случае требует $O(n^2)$ времени. Однако нетрудно заметить, что решение может быть найдено проще и быстрее, если алфавит будет, например, двоичным. В этой связи отметим, что в цифровых моделях компьютерных вычислений буквы неупорядоченного алфавита можно закодировать в виде бинарных строк, для которых применимы отношения порядка. Таким образом, на практике всегда можно считать, что алфавит упорядочен.

На этой простой (и типичной) задаче показано, что выбор подходящего алгоритма очень сильно зависит от алфавита, которым определяется исследуемая строковая последовательность. Обсуждение интересного вопроса зависимости ал-

[1] АВЛ-деревом (названным так в честь его авторов Г. М. Адельсона-Вельского и Е. М. Ландиса) называется сбалансированное по высоте дерево двоичного поиска, у которого для каждого узла высоты его правого и левого поддеревьев отличаются не более чем на единицу. — *Примеч. ред.*

[2] 2-3-деревом называется дерево поиска специального вида, когда из каждого внутреннего узла выходит от двух до трех ребер и все пути от корня дерева до любого листа имеют одинаковую длина. Кроме того, во внутренние узлы обычно помещаются определенные записи с информацией об этих узлах или путях от корня до узла. — *Примеч. ред.*

горитма от алфавита на этом не заканчивается; оно будет продолжено в других разделах книги.

Необходимость предварительной подготовки данных

Как мы видели в разделе 2.1, вычисление дерева суффиксов можно рассматривать как процесс предварительной подготовки данной строки $\boldsymbol{x}$: это дерево затем используется для многократных поисков в строке $\boldsymbol{x}$, при этом для каждого такого поиска его время пропорционально длине искомого паттерна. Аналогично в алгоритме КМП (см. раздел 7.1) на этапе предварительной подготовки вычисляется массив граней паттерна, который затем используется в процессе поиска, позволяя исключить обратный просмотр. Таким образом, возможность алгоритма использовать предварительную подготовку данных можно считать дополнительным положительным качеством этого алгоритма.

Продемонстрируем преимущества предварительной подготовки на примере задачи 4.1, сформулированной в предыдущем подразделе. Предположим, что алфавит A фиксирован (что известно заранее) и конечен. Также предположим, что необходимо подсчитать вхождения букв не для одной строки, а для $m \geqslant 1$ строк, определенных на этом алфавите A. Для простоты и без потери общности можно считать, что все m строк имеют одинаковую длину n. Тогда на этапе предварительной подготовки за время $\Theta(\alpha)$ алфавит можно закодировать в виде хеш-таблицы T так, чтобы отдельные буквы были доступны за фиксированное или почти константное время. Значения хеш-таблицы $T[\lambda]$ для каждой буквы λ должны содержать счетчик строк $j \in 0..m$, показывающий номер строки, в которой последний раз встречалась буква λ. Первоначально счетчик j устанавливается равным нулю для всех λ. Значения $T[\lambda]$ также должны содержать счетчик вхождений c. После завершения этапа предварительной подготовки начинается процесс просмотра m строк. Для каждой позиции i в j-й строке $\boldsymbol{x}_j$ вычисляется $T[\boldsymbol{x}_j[i]]$; если значение строкового счетчика равно j, тогда значение счетчика вхождений c увеличивается на единицу, иначе строковый счетчик устанавливается равным j, а счетчик c — равным 1. Среднее время этапа предварительной подготовки должно "раскладываться" на все m строк. Поэтому время обработки одной строки составляет $\Theta(\alpha/m + n)$ независимо от используемого алфавита. Очевидно, такой алгоритм, использующий предварительную подготовку, во многих случаях предпочтительнее других рассмотренных выше алгоритмов для решения задачи 4.1, поскольку предполагает, что алфавит обработан заранее.

Отметим, что все рассмотренные алгоритмы для подсчета количества вхождений букв в заданные строки являются онлайновыми.

Кодирование выходного результата

Как мы видели в разделе 2.3, при вычислении кратных строк (задача 2.15) ключевую роль играет кодирование (способ представления) выходного результа-

та, которое позволяет уменьшить объем выходных данных, а также сложность алгоритма от $O(n^2)$ до $O(n \log n)$. Например, для отображения в выходных результатах кратной строки a^6 наиболее эффективно в нормальной форме $(i,\ 1,\ 6)$, при этом предполагается, что вхождения кратных подстрок a^2, a^3 и a^4 также определены.

В общем случае, как показано в обсуждениях 1.3.2 и 2.3, неясно, на основании какого критерия принимать решение об использовании кодирования выходных данных. Рассмотрим, например, массив граней $\boldsymbol{\beta}$ для некоторой строки $\boldsymbol{x}$. Обычно считается, что массив граней является подходящим способом кодирования (представления) граней. Однако, как следует из леммы 1.3.1, для того чтобы действительно иметь список всех граней, необходимо вычислить $\boldsymbol{\beta}[n], \boldsymbol{\beta}^2[n], \ldots$. Напомним: из леммы 3.2.3 следует, что строки Туе, являющиеся $(2,3)$-исключениями, а также строки Фибоначчи имеют количество граней, пропорциональное логарифму от длин строк. Поэтому в общем случае требуется $\Theta(\log n)$ времени (или даже больше) для формирования списка граней строки $\boldsymbol{x}$ на основании вычисленного массива граней $\boldsymbol{\beta}$. Скептики правы, когда утверждают, что, вычислив массив граней, мы в действительности не имеем решения задачи, поскольку требуются еще дополнительные значительные вычислительные усилия, чтобы получить окончательное решение задачи. В пользу массивов граней приводятся такие практические соображения: как правило, основной интерес представляют грани наибольшей длины (которые легко получить из массивов граней), а грани небольшой длины рассматриваются только после того, как получены грани наибольшей длины.

Упражнения 4.1

1. Разработайте в деталях четыре алгоритма вычисления количества вхождений букв в заданные строки, кратко описанные в этом разделе.

2. Оцените временну́ю сложность всех четырех алгоритмов, описанных в данном разделе. Определите классы строковых последовательностей, для которых тот или иной алгоритм предпочтительнее других.

3. Покажите, что буквы индексированного алфавита A объемом α можно однозначно отобразить за линейное время в множество натуральных чисел $N_\alpha = \{1, 2, \ldots, \alpha\}$. Отсюда будет следовать утверждение, что любой индексированный алфавит без потери общности можно считать множеством натуральных чисел N_α.

4. В тексте раздела утверждалось (но не доказывалось), что строки Туе, являющиеся $(2,3)$-исключениями, имеют $\Theta(\log n)$ граней, а строки Фибоначчи — $\Omega(\log n)$ граней, где n — длина строки. Докажите эти утверждения, используя леммы 3.2.3 и 3.4.4.

4.2 Разные паттерны

Как сделать одну строку отличной от другой? Это фундаментальный вопрос методов генерирования данных, предназначенных для тестирования строковых алгоритмов: как сгенерировать как можно больше "различных" строк, избегая при этом дублирования не "различных" строк.

Традиционный подход (согласно определениям раздела 1.2) — считать две строки различными только тогда, когда они различны хотя бы в одной позиции. Согласно этому определению, на конечном алфавите A объемом α можно построить в точности α^n различных строк длиной n, тем самым предлагая для тестирования исчерпывающие данные в виде α^n различных строк длиной n на алфавите A.

Однако такой подход не является единственным или наилучшим единственным для многих строковых алгоритмов. Рассмотрим простую задачу подсчета вхождений букв в некоторую заданную строку $\boldsymbol{x}$, описанную в разделе 4.1. На неупорядоченном алфавите алгоритмы, решающие эту задачу, просматривают каждую букву строки $\boldsymbol{x}$ по отдельности, выполняя операцию сравнения каждой буквы с буквами алфавита. Для таких алгоритмов результат выполнения будет одинаковым независимо от того, будет ли входная строка иметь вид $\boldsymbol{x} = aab$ или $\boldsymbol{y} = ddc$. Нужно, чтобы тестовые данные содержали одну из таких строк, но совсем не обязательно тестировать обе эти строки.

Этот пример порождает идею ***эквивалентности относительно паттерна***, где две строки $\boldsymbol{x}$ и $\boldsymbol{y}$ будут считаться эквивалентными, если они соответствуют некоторому "паттерну". Чтобы сформулировать эту идею более точно, введем такое определение.

Определение 4.2.1. Две строки $\boldsymbol{x}$ и $\boldsymbol{y}$ называются p-**эквивалентными**, если выполняются следующие условия:

а) для всех целых $n \geqslant 0$ $|\boldsymbol{x}| = |\boldsymbol{y}| = n$;

б) для всех целых i и j, $1 \leqslant i \leqslant j \leqslant n$, $\boldsymbol{x}[i] = \boldsymbol{x}[j] \Leftrightarrow +\boldsymbol{y}[i] = \boldsymbol{y}[j]$.

Если строки $\boldsymbol{x}$ и $\boldsymbol{y}$ не являются p-эквивалентными, то они называются p-**различными**. ■

В этом разделе, следуя работе [183], мы исследуем свойства p-различных строк. В частности, мы покажем, как подсчитать количество и как вычислить все p-различные строки длиной n. Покажем, что таких строк относительно немного. Далее будет описано, как вычислить такие строки за константное время в расчете на одну строку.

В следующем разделе мы введем понятие эквивалентности относительно граней (или b-эквивалентность), которое еще ближе обычному понятию эквивалентности, чем p-эквивалентность. Это приведет к определению b-различных строк,

которых, как будет показано, еще меньше, чем p-различных строк. Такие строки также можно вычислить за константное время в расчете на одну строку.

Таким образом, p-различные и b-различные строки могут быть тестовыми данными для алгоритмов, которым они подходят, при этом они эффективны (дают исчерпывающий набор тестовых данных) и результативны (их можно быстро сгенерировать).

Первым важным фактом, относящимся к p-эквивалентности, является то, что это отношение действительно является отношением эквивалентности между строковыми последовательностями, и поэтому оно разбивает все множество строк длиной n на классы эквивалентности p-различных строк (см. упражнение 4.2.2). Отсюда следует, что для каждого класса эквивалентности можно определить единственный представитель этого класса. Чтобы увидеть это, введем счетный бесконечный ***стандартный алфавит***

$$\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_\alpha, \ldots\} \tag{4.1}$$

и подалфавиты $\Lambda_\alpha = \{\lambda_1, \lambda_2, \ldots, \lambda_\alpha\}$ для каждого целого $\alpha \geqslant 1$. Предполагаем, что буквы алфавита Λ упорядочены естественным образом

$$\lambda_1 < \lambda_2 < \cdots < \lambda_\alpha < \ldots.$$

Вследствие такого упорядочения без потери общности можно считать, что стандартный алфавит состоит из натуральных чисел. Тогда для любой заданной строки $\boldsymbol{x}$ на алфавите Λ, определим ***p-каноническую строку*** $\boldsymbol{x}^*$, соответствующую строке $\boldsymbol{x}$, как лексикографически наименьшую строку на алфавите Λ, p-эквивалентную строке $\boldsymbol{x}$. Очевидно, что строка $\boldsymbol{x}^*$ является уникальным представителем своего класса p-эквивалентности. Нетрудно показать, что любая строка $\boldsymbol{x}^*$ удовлетворяет следующему условию.

(P) Для любого целого положительного j, если буква λ_j входит в строку $\boldsymbol{x}^*$, тогда следующая буква λ_{j+1} не может иметь другого вхождения в эту строку ранее самого первого (самого левого) вхождения буквы λ_j.

Обозначим через $p'[\alpha, n]$ количество p-канонических строк длиной n, сформированных с использованием всех букв подалфавита Λ_α (каждая буква подалфавита должна хотя бы один раз входить в каждую строку). Мы подсчитаем это количество с помощью бесконечного двухмерного массива, называемого p'-массивом.

Теорема 4.2.2. Для любых положительных целых α и n

а) $p'[1, n] = 1$;

б) если $\alpha > n$, тогда $p'[\alpha, n] = 0$;

в) $p'[\alpha, \alpha] = 1$;

г) если $\alpha \geqslant 2$ и $n \geqslant 2$, тогда $p'[\alpha, n] = p'[\alpha - 1, n - 1] + \alpha p'[\alpha, n - 1]$.

Доказательство.

а) Если $\alpha = 1$, тогда имеется только одна p-каноническая строка $\boldsymbol{x}^* = \lambda_1^n$.

б) Если $\alpha > n$, тогда на основании свойства (Р) можно утверждать, что не существует p-канонической строки, которая содержала бы букву λ_α.

в) Из свойства (Р) следует, что существует только одна p-каноническая строка, которая содержит α различных букв: $\boldsymbol{x}^* = \lambda_1\lambda_2\ldots\lambda_\alpha$.

г) Обозначим через $\pi_1 = p'[\alpha-1, n-1]$ количество различных p-канонических строк длиной $n-1$, которые содержат в точности $\alpha-1$ букв подалфавита $\Lambda_{\alpha-1}$. Обозначим множество этих строк как $S_1 = \{\boldsymbol{x}_1^*, \boldsymbol{x}_2^*, \ldots, \boldsymbol{x}_{\pi_1}^*\}$. Тогда все строки вида

$$\boldsymbol{x}_i^*\lambda_\alpha, \tag{4.2}$$

где целое i удовлетворяет неравенствам $1 \leqslant i \leqslant \pi_1$, будут различными и p-каноническими.

Подобным образом обозначим через $\pi_2 = p'[\alpha, n-1]$ количество различных p-канонических строк длиной $n-1$, которые содержат в точности α букв подалфавита Λ_α. Обозначим множество этих строк как $S_2 = \{\boldsymbol{y}_1^*, \boldsymbol{y}_2^*, \ldots, \boldsymbol{y}_{\pi_2}^*\}$. Тогда все α строк вида

$$\{\boldsymbol{y}_i^*\lambda_1, \boldsymbol{y}_i^*\lambda_2, \ldots, \boldsymbol{y}_i^*\lambda_\alpha\}, \tag{4.3}$$

где целое i удовлетворяет неравенствам $1 \leqslant i \leqslant \pi_2$, будут различными и p-каноническими. Поскольку конечные буквы в этих строках входят в них не менее двух раз, эти строки не совпадают ни с одной строкой вида (4.2). Поэтому справедливо неравенство

$$p'[\alpha, n] \geqslant p'[\alpha-1, n-1] + \alpha p'[\alpha, n-1].$$

Пусть строка $\boldsymbol{x}^*$ будет p-канонической длины n, содержащей все буквы подалфавита Λ_α. Представим строку $\boldsymbol{x}^*$ как $\boldsymbol{x}^* = \boldsymbol{y}^*\lambda_i$. Если буква λ_i входит в строку $\boldsymbol{y}^*$, тогда $\boldsymbol{y}^* \in S_2$, и поэтому строка $\boldsymbol{x}^*$ будет одной из строк вида (4.3). С другой стороны, из свойства (Р) следует, что буква λ_α не может входить в строку $\boldsymbol{y}^*$. Поэтому при $i = \alpha$ имеем $\boldsymbol{y}^* \in S_1$, а строка $\boldsymbol{x}^*$ является одной из строк вида (4.2). Эти рассуждения приводят к неравенству

$$p'[\alpha, n] \leqslant p'[\alpha-1, n-1] + \alpha p'[\alpha, n-1].$$

Из последнего неравенства и противоположного ему, полученного выше, следует утверждение *г*) теоремы. ■

Рекуррентное соотношение утверждения *г*) теоремы 4.2.2 хорошо известно в математике: с учетом начальных значений, определяемых утверждениями *а*)–*в*) этой теоремы, это соотношение определяет так называемые числа Стирлинга второго рода $\sigma_n^{(\alpha)}$ [135]. Таким образом, для всех положительных целых чисел n и α имеем

$$p'[\alpha, n] = \sigma_n^{(\alpha)}. \tag{4.4}$$

В табл. 4.1 приведены значения верхнего левого угла p'-массива. Суммы $p[n]$ значений по столбцам равны общему количеству p-канонических строк длиной n. Отметим, что числа $p[n]$ также хорошо известны — они называются числами Белла [211].

Таблица 4.1. Значения p'-массива для строк длиной $n \leqslant 9$ и $\alpha \leqslant 9$

n	1	2	3	4	5	6	7	8	9
$\sigma_n^{(1)}$	1	1	1	1	1	1	1	1	1
$\sigma_n^{(2)}$	0	1	3	7	15	31	63	127	255
$\sigma_n^{(3)}$	0	0	1	6	25	90	301	966	2025
$\sigma_n^{(4)}$	0	0	0	1	10	65	350	1701	7770
$\sigma_n^{(5)}$	0	0	0	0	1	15	140	1050	6951
$\sigma_n^{(6)}$	0	0	0	0	0	1	21	266	2646
$\sigma_n^{(7)}$	0	0	0	0	0	0	1	28	462
$\sigma_n^{(8)}$	0	0	0	0	0	0	0	1	36
$\sigma_n^{(9)}$	0	0	0	0	0	0	0	0	1
$p[n]$	1	2	5	15	52	203	877	4140	20147

На примере значений табл. 4.1 можно проиллюстрировать другую зависимость между числами Стирлинга и нашими числами p'. Числа Стирлинга $\sigma_n^{(\alpha)}$ обычно определяют как число способов разбиения множества S из n элементов на α непустых непересекающихся подмножеств, объединение которых составляет множество S. Чтобы увидеть, как это определение соответствует числам $p'[\alpha, n]$, рассмотрим случай, когда $n = 4$ и $\alpha = 2$. Если выписать для этого случая семь строк ($p'[2, 4] = \sigma_4^{(2)} = 7$) и затем образовать $\alpha = 2$ подмножества *позиций* одинаковых букв в этих строках, то увидим, что декомпозиция множества $\{1, 2, 3, 4\}$ на подмножества обладает следующими свойствами.

- Такая декомпозиция единственная (поскольку все строки различные).
- Декомпозиция непустая (поскольку в строки входят все α букв).
- Образованные подмножества не пересекаются (поскольку каждой позиции соответствует только одна буква).

Приведем эту декомпозицию.

1234	
$aaab$	$\{1,2,3\}\{4\}$
$aaba$	$\{1,2,4\}\{3\}$
$aabb$	$\{1,2\}\{3,4\}$
$abaa$	$\{1,3,4\}\{2\}$
$abab$	$\{1,3\}\{2,4\}$
$abba$	$\{1,4\}\{2,3\}$
$abbb$	$\{1\}\{2,3,4\}$

Легко видеть, что объединение в приведенном примере по отдельности левых и правых множеств порождает два непустых и непересекающихся множества. Доказательство этого факта в общем случае оставим для упражнения 4.2.4.

Теорема 4.2.2 предлагает итерационный способ вычисления чисел $p'[\alpha, n]$. Также различные формулы для этих чисел можно получить, используя связь чисел Стирлинга второго рода с биноминальными коэффициентами и с числами Стирлинга первого рода [135]. Отметим, что для любого фиксированного алфавита объемом α сумма $\sum_{i=1}^{\alpha} p'[i,n]$ равняется количеству p-различных строк длиной n, содержащих *не более* α различных букв. Поскольку для значений n, больших α, почти все такие строки содержат в точности α букв, отсюда следует, что

$$\lim_{n\to\infty}\left(\sum_{i=1}^{\alpha} p'[i,n] \Big/ \frac{\alpha^n}{\alpha!}\right) = 1. \tag{4.5}$$

Если рассматривать различные, в обычном понимании, строки длиной n, сформированные из не более α букв, то таких строк всего α^n. Формула (4.5) показывает, что p-различных строк, также сформированных из не более α букв, асимптотически в $\alpha!$ раз меньше. Например, для $\alpha = 5$ и $n = 9$ имеем $5^9 = 1\,953\,125$ обычных различных строк, тогда как, суммируя первые пять значений в 9-м столбце табл. 4.1, получаем $17\,002$ p-различных строк. Отношение $1\,953\,125$ к $17\,002$ дает значение, близкое к 115, что примерно равно $5!$.

Конечно, наибольший интерес представляет сумма $p[n] = \sum_{i=1}^{n} p'[i,n]$, которую можно вычислить различными способами, в частности с помощью рекуррентного соотношения

$$p[n] = \sum_{j=0}^{n-1} C_{n-1}^{j} p[j], \quad n \geqslant 2 \tag{4.6}$$

или как бесконечную сумму

$$p[n] = e^{-1} \sum_{j \geqslant 1} \frac{j^{n-1}}{(j-1)!}, \tag{4.7}$$

в которой фактически надо оценить только $\Theta(n)$ слагаемых, поскольку, как легко показать,

$$p[n] - 1 > e^{-1} \sum_{j=1}^{2n} \frac{j^{n-1}}{(j-1)!}. \tag{4.8}$$

Итак, можем подсчитать количество p-различных строк путем вычисления величин $p'[\alpha, n]$ и $p[n]$. Теперь покажем, как генерировать p-канонические строки, которые можно использовать в качестве тестовых данных для строковых алгоритмов. Возвращаясь к доказательству теоремы 4.2.2, видим, что

- добавление буквы λ_α к p-каноническим строкам увеличивает их количество на величину $p'[\alpha - 1, n - 1]$;
- добавление букв $\lambda_1, \lambda_2, \ldots, \lambda_\alpha$ к p-каноническим строкам увеличивает их количество на величину $p'[\alpha, n - 1]$.

Это простое замечание дает основу для итерационного алгоритма вычисления всех $p[n]$ p-канонических строк длиной n. Как показано ниже, эти строки можно генерировать за фиксированное время в расчете на одну строку путем построения дерева T_n. Фактически, мы возвращаемся к обсуждению задачи 2.4 из раздела 2.1, поскольку дерево T_n является синтаксическим деревом, но у него помечаются не ребра, а узлы.

Узлы дерева T_n помечаются парой (λ, α), где λ — буква алфавита Λ, а α — количество различных букв, которые можно найти на пути от корня дерева до текущего узла. Дерево T_1 состоит из одного корневого узла, помеченного как $(\lambda_1, 1)$, а для всех целых $n \geqslant 2$ деревья T_n формируются из деревьев T_{n-1} путем добавления к листу, конечному узлу с меткой (λ, α), новых конечных узлов с метками

$$(\lambda_1, \alpha), (\lambda_2, \alpha), \ldots, (\lambda_\alpha, \alpha), (\lambda_{\alpha+1}, \alpha + 1).$$

На рис. 4.1 показано полное дерево T_3, где буквы λ_i заменены на индексы i.

Из теоремы 4.2.2 следует, что дерево T_n имеет ровно $p[n]$ конечных узлов (листьев) и буквы, стоящие на пути от корня до этих узлов, дают все $p[n]$ p-канонические строки $\boldsymbol{x}^*$ длиной n. Таким способом генерируются эти строки на основе построенного дерева T_n. Еще раз отметим, что дерево T_n формируется из дерева T_{n-1} путем добавления $p[n]$ концевых узлов — операция, требующая фиксированного времени для одного узла или всего $\Theta(p[n])$ времени. Из формулы (4.6) следует очевидное неравенство $p[n] \geqslant 2p[n-1]$, откуда заключаем, что все дерево T_n можно построить за время $\Theta(p[n])$. Это доказывает следующую теорему.

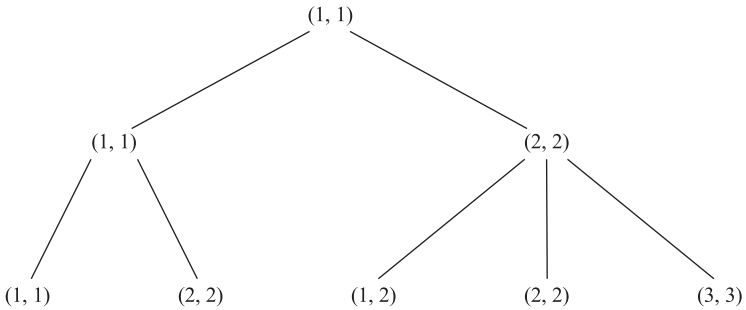


Рис. 4.1. Дерево T_3, представляющее p-различные строки длиной 3

Теорема 4.2.3. Для любого положительного целого n все p-канонические строки длиной n можно вычислить за время порядка $\Theta(p[n])$, для чего необходима память объемом $\Theta(p[n])$. ■

Отметим, что данный алгоритм генерирования p-канонических строк является онлайновым в слабом значении этого понятия, как обсуждалось в разделе 4.1, — поскольку дерево T_n строится на основе дерева T_{n-1}, для решения задачи для данного n необходимо сначала решить задачу для $n-1$. Такое онлайновое решение эквивалентно построению дерева T_n "в ширину". Это означает, что решение легко расширяемо: если возникнет необходимость в дереве T_{n+1}, то оно может быть сформировано тем же алгоритмом за один шаг из дерева T_n. Как мы увидим в следующем разделе, эффективное решение "в ширину" иногда может привести к определенным трудностям.

Результат, подобный теореме 4.2.3, можно установить и для p-канонических строк длиной n, использующих не более α букв алфавита Λ. Напомним, что таких строк всего $p'[\alpha, n]$. В этом случае для генерирования таких строк используется поддерево дерева T_n, в котором пути длиной n заканчиваются в конечных узлах, помеченных метками вида $(*, \alpha)$.

В заключение этого раздела отметим, что дерево T_n можно проходить (просматривать) различными способами, получая при этом различное упорядочение p-канонических строк. Например, при прямом прохождении T_n получаем список строк, упорядоченных в лексикографическом порядке.[3] И, соответственно, по-

[3]При прохождении дерева в прямом порядке сначала посещается корень дерева, затем в таком же порядке посещаются все узлы самого левого поддерева, далее посещаются узлы следующих слева направо поддеревьев. При прохождении дерева в обратном порядке обход начинается с самого правого листа самого правого поддерева данного дерева, затем в таком же порядке посещаются все узлы этого самого правого поддерева, далее узлы следующих справа налево поддеревьев. Последним посещается корень дерева. — *Примеч. ред.*

лучим обратный порядок строк, если проходить дерево T_n в обратном порядке. Поскольку каждую строку можно "исключать" из списка после ее генерирования, для генерирования строк на основе прохождения дерева T_n в любом порядке фактически требуется память объемом порядка $\Theta(n)$. Так как $p[n] \leqslant 2^n$ (что следует из формулы (4.6)), то этот объем памяти эквивалентен объему $\Theta(\log p[n])$.

Упражнения 4.2

1. Переформулируйте определение 4.2.1 таким образом, чтобы исключить из него требование равенства длин строк $\boldsymbol{x}$ и $\boldsymbol{y}$.
2. Отношение эквивалентности обладает свойствами рефлексивности, симметричности и транзитивности. Докажите, что отношение p-эквивалентности действительно является отношением эквивалентности.
3. Докажите свойство (P).
4. Используя "обычное" определение чисел Стирлинга, покажите, что равно числу всех p-канонических строк длиной n, включающих ровно α букв.
5. Докажите формулы (4.6)–(4.8).
6. Постройте дерево T_4 и проверьте, что оно имеет $p[4] = 15$ листьев (конечных узлов).
7. Разработайте рекурсивный алгоритм построения дерева T_n для заданного значения n.
8. Покажите, что поддерево дерева T_n, соответствующего $p'[\alpha, n]$ p-каноническим строкам длиной n, содержащим не более α букв алфавита Λ, можно построить за время $O(\alpha p'[\alpha, n])$, используя память объемом $O(\alpha p'[\alpha, n])$.
9. В тексте без доказательства утверждалось, что при прямом прохождении дерева T_n получается список p-канонических строк, упорядоченных в лексикографическом порядке. Докажите или опровергните это утверждение.

4.3 Разные грани

Как мы видели в разделах 1.2 и 1.3, основное преимущество массивов граней заключается в возможности описать периодическую структуру строковых последовательностей. Поэтому не вызывает удивления тот факт, что массивы граней играют важную роль во многих алгоритмах, которые исследуют эту структуру или выходной результат которых зависит от этой структуры. Например, алгоритм КМП (см. раздел 7.1), который находит частные паттерны в строках (задача 2.8), зависит от вычисления массивов граней для этих паттернов. Алгоритмы, вычисляющие все оболочки для данной строки (задача 2.18), также сильно зависят от массивов граней: эффективные алгоритмы требуют первоначальной замены

строки ее массивом граней, тогда последующие вычисления выполняются с этим массивом, а не с самой строкой.

Подготовка тестовых данных для этих и других алгоритмов показывает важность генерирования строк, порождающих различные массивы граней. Введем определение.

Определение 4.3.1. Две строки $\boldsymbol{x}$ и $\boldsymbol{y}$ называются **эквивалентными относительно граней** (b-**эквивалентными**), если их массивы граней совпадают. В противном случае строки $\boldsymbol{x}$ и $\boldsymbol{y}$ называются b-**различными**. ■

Чтобы пояснить это определение, рассмотрим следующие строки:

$$\boldsymbol{x} = ababb, \quad \boldsymbol{x}' = ababc, \quad \boldsymbol{x}'' = abaacb.$$

Строки $\boldsymbol{x}$ и $\boldsymbol{x}'$ p-различны, но b-эквивалентны, поскольку имеют одинаковый массив граней $\beta = 00120$. Таким образом, мы видим, что из b-эквивалентности не вытекает p-эквивалентность. С другой стороны, строки $\boldsymbol{x}$ и $\boldsymbol{x}''$ b-различны, поскольку строка $\boldsymbol{x}''$ имеет массив граней 00100, но они также и p-различны. Очевидно, что из b-различия строк с необходимостью вытекает их p-различие. Это простое наблюдение сформулируем в виде леммы.

Лемма 4.3.2. Если две строки p-эквивалентны, то они также b-эквивалентны.

Доказательство этой леммы предложено дать в упражнении 4.3.1. ■

Как следует из этой леммы, каждый класс b-эквивалентных строк состоит из одного или нескольких классов p-эквивалентных строк. Следовательно, количество классов b-эквивалентных строк меньше количества классов p-эквивалентных строк. Поскольку p-эквивалентность является отношением эквивалентности, мы снова можем определить b-канонические представители $\boldsymbol{x}^*$ для каждого класса эквивалентности как лексикографически наименьшую строку из данного класса, определенную на стандартном алфавите Λ. Согласно этому определению, каждый класс b-эквивалентных строк на алфавите Λ имеет бесконечную мощность, но с учетом леммы 4.3.2 без потери общности мы можем упростить ситуацию, если ограничим этот класс только p-каноническими строками. Тогда, например, класс S p-канонических b-эквивалентных строк, соответствующих массиву граней $\beta = 00100$, вполне обозрим:

$$S = \{\lambda_1\lambda_2\lambda_1\lambda_3\lambda_2, \lambda_1\lambda_2\lambda_1\lambda_3\lambda_3, \lambda_1\lambda_2\lambda_1\lambda_3\lambda_4\},$$

Для этого класса b-каноническим элементом будет строка $\boldsymbol{x}^* = \lambda_1\lambda_2\lambda_1\lambda_3\lambda_2$.

Для того чтобы сделать вычисление b-канонических строк эффективным, надо ввести структуры, подобные деревьям, используемым в разделе 4.2, тогда канонические строки длиной n можно будет вычислить на основе канонических строк

длиной $n-1$. Для этого необходимо понять, как элемент $\beta[n]$ массива граней $\boldsymbol{\beta}$ можно вычислить, зная только предшествующие ему элементы $\boldsymbol{\beta}[1..n-1]$. Напомним, что из леммы 1.3.1 следует, что для любой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ возможные значения $\boldsymbol{\beta}[n]$ являются элементами убывающей последовательности

$$\left\langle 1\boldsymbol{\beta}^1[n-1]+1, \boldsymbol{\beta}^2[n-1]+1, \ldots, \boldsymbol{\beta}^k[n-1]+1, 0 \right\rangle, \tag{4.9}$$

где $\boldsymbol{\beta}^1[n-1] \equiv \boldsymbol{\beta}[n-1]$, $\boldsymbol{\beta}^i[n-1] = \boldsymbol{\beta}[\boldsymbol{\beta}^{i-1}[n-1]]$ ($i \in 2..k$) и k — наименьшее целое, такое, что $\boldsymbol{\beta}^k[n-1] = 0$. Докажем более строгий результат о том, что множество значений, действительно принимаемых величиной $\boldsymbol{\beta}[n]$, не зависит от префикса $\boldsymbol{x}[1..n-1]$.

Лемма 4.3.3. Для любого целого $n \geqslant 2$ значения, принимаемые $\boldsymbol{\beta}[n]$, зависят только от $\boldsymbol{\beta}[1..n-1]$ и размера алфавита.

Доказательство. Предположим, что есть две строки $\boldsymbol{x}$ и $\boldsymbol{y}$ длиной $n-1$, определенные на алфавите размером α и имеющие одинаковый массив граней $\boldsymbol{\beta}[1..n-1]$. Предположим далее, что для некоторой буквы λ и некоторого целого m строка $\boldsymbol{x}' = \boldsymbol{x}\lambda$ имеет массив граней $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n] = \boldsymbol{\beta}[1..n-1]m$ и при этом не существует такой буквы μ, что строка $\boldsymbol{y}' = \boldsymbol{y}\mu$ имела бы тот же самый массив граней $\boldsymbol{\beta}$. Тогда число m должно принимать одно из значений множества (4.9).

Сначала рассмотрим случай, когда $m = \boldsymbol{\beta}^i[n-1]+1$ для некоторого $i \in 1..k$. Поскольку $\boldsymbol{\beta}^i[n-1] = m-1$, то отсюда следует, что

$$\boldsymbol{y}[1..m-1] = \boldsymbol{y}[n+1-m..n-1].$$

Так как $\boldsymbol{\beta}^i[n] \neq m$, то, положив $\boldsymbol{y}[n] = \boldsymbol{y}[m]$, для некоторого $m' > m$, получим

$$\boldsymbol{y}[1..m'] = \boldsymbol{y}[n+1-m'..n].$$

Но это означает, что

$$\boldsymbol{y}[1..m'-1] = \boldsymbol{y}[n+1-m'..n-1].$$

Поэтому $\boldsymbol{\beta}^i[n-1] = m'-1 > m-1$, что противоречит условию $\boldsymbol{\beta}^i[n-1] = m-1$. Таким образом, лемма справедлива для любого целого $m = \boldsymbol{\beta}^i[n-1]+1$.

Теперь пусть $m = 0$. Тогда для любой из α букв $\boldsymbol{y}'[n] = \mu$ получаем значение $\boldsymbol{\beta}[n] > 0$, тогда как в случае $\boldsymbol{x}'[n] = \lambda$ имеем $\boldsymbol{\beta}[n] = 0$. Следовательно, существует целое $m' > 0$, такое, что для $\boldsymbol{y}'\boldsymbol{\beta}[n] = m'$, но для $\boldsymbol{x}'\boldsymbol{\beta}[n] \neq m'$, что противоречит уже рассмотренному случаю.

Отсюда следует вывод, что массив $\boldsymbol{\beta}$ будет массивом граней для некоторой строки $\boldsymbol{x}'$ только тогда, когда он будет массивом граней и для строки $\boldsymbol{y}'$. ■

Этот фундаментальный результат дает возможность (см. работы [90, 79]) вычислить массив $\boldsymbol{\beta}[1..n]$ на основе массива $\boldsymbol{\beta}[1..n-1]$ без непосредственного обращения к исходной строке. Как можно использовать этот результат, мы покажем далее, а сейчас докажем, что любой префикс b-канонической строки $\boldsymbol{x}_n^* = \boldsymbol{x}^*[1..n]$ также будет b-канонической строкой.

Лемма 4.3.4. Для любого $n \geqslant 1$ в b-канонической строке $x_n^* = x_{n-1}^*\lambda$, где λ — некоторая буква стандартного алфавита, префикс x_{n-1}^* также является b-канонической строкой.

Доказательство. Предположим, что строка $\boldsymbol{x}_n^* = \boldsymbol{x}_{n-1}\lambda$ имеет массив граней $\boldsymbol{\beta}_n = \boldsymbol{\beta}[1..n]$ и префикс $\boldsymbol{x}_{n-1}$ не является b-канонической строкой. Обозначим через $\boldsymbol{\beta}_{n-1} = \boldsymbol{\beta}[1..n-1]$ массив граней строки $\boldsymbol{x}_{n-1}$. Поскольку $\boldsymbol{x}_{n-1}$ не является b-канонической строкой, то существует строка $\boldsymbol{y}_{n-1} < \boldsymbol{x}_{n-1}$ с тем же массивом граней $\boldsymbol{\beta}_{n-1}$. Тогда вследствие леммы 4.3.3, существует строка $\boldsymbol{y}_n = \boldsymbol{y}_{n-1}\lambda'$ с массивом граней $\boldsymbol{\beta}_n$, но при этом $\boldsymbol{y}_n < \boldsymbol{x}_n^*$. Последнее неравенство противоречит условию, что строка $\boldsymbol{x}_n^*$ является b-канонической. Лемма доказана. ■

Из этой леммы вытекает, что *все* b-канонические строки $\boldsymbol{x}_n^*$ можно сформировать из b-канонических строк $\boldsymbol{x}_{n-1}^*$. Это дает нам надежду, что, как и в случае с p-каноническими строками, можно сгенерировать b-канонические строки $\boldsymbol{x}_n^*$ на основе древовидной структуры, построенной для генерирования строк $\boldsymbol{x}_{n-1}^*$. Но прежде чем перейти к построению такой структуры, нам необходимо выяснить, как на основе массива граней $\boldsymbol{\beta}_{n-1}$ можно вычислить различные массивы $\boldsymbol{\beta}_n$.

Лемма 4.3.5. Пусть b-канонической строке $\boldsymbol{x}_{n-1}^*$ на стандартном алфавите Λ соответствует массив граней $\boldsymbol{\beta}_{n-1}$. Тогда массив $\boldsymbol{\beta}_{n-1}$ будет префиксом в точности k различных массивов граней $\boldsymbol{\beta}_n$ только в том случае, когда строка $\boldsymbol{x}_{n-1}^*\lambda_k$ будет b-канонической с массивом граней $\boldsymbol{\beta}_{n-1}0$.

Доказательство. Сначала предположим, что строка $\boldsymbol{x}_{n-1}^*\lambda_k$ является b-канонической и имеет только пустую грань. Поскольку для каждой b-канонической строки соответствующий ей массив граней лексикографически наименьший, поэтому не существует буквы λ_i, $i < k$, такой, что строка $\boldsymbol{x}_{n-1}^*\lambda_i$ имеет только пустую грань. Следовательно, для всех $i \in 1..k-1$ строки $\boldsymbol{x}_{n-1}^*\lambda_i$ имеют различные непустые грани.

Теперь предположим, что для некоторого $i > k$ b-каноническая строка $\boldsymbol{x}_{n-1}^*\lambda_i$ имеет наибольшую грань длиной $m > 0$, тогда $\boldsymbol{\beta}_n = \boldsymbol{\beta}_{n-1}m$. (Фактически, $m \geqslant 3$, поскольку $m \geqslant i > k \geqslant 2$.) Из леммы 4.3.4 следует, что строка $\boldsymbol{x}_{n-1}^*$ имеет b-канонический префикс $\boldsymbol{x}_m^* = \boldsymbol{x}_{m-1}^*\lambda_i$, где $\boldsymbol{x}_{m-1}^*$ также b-каноническая строка. Далее, поскольку строка $\boldsymbol{x}_{n-1}^*\lambda_k$ имеет только пустую грань, то и префикс $\boldsymbol{x}_{m-1}^*\lambda_k$ также имеет только пустую грань. Тогда для любого положительного целого $k' \leqslant k$ строка $\boldsymbol{x}_{m-1}^*\lambda_k'$ является b-канонической с непустой гранью. Таким образом, мы свели

первоначальную задачу с конечными положительными целыми числами $n-1$ и k к такой же задаче с другими конечными числами $m-1$ и k'. Такое сведение теоретически можно продолжать бесконечно, что убеждает нас в невозможности существования такого $i > k$, что строка $\boldsymbol{x}^*_{n-1}\lambda_i$ имеет непустую грань. Следовательно, существует ровно k различных массивов граней $\boldsymbol{\beta}_n$, что и доказывает достаточность утверждения леммы.

Для доказательства необходимости утверждения леммы предположим, что существует ровно k различных массивов граней $\boldsymbol{\beta}_n$. Но тогда один из них обязательно должен иметь вид $\boldsymbol{\beta}_{n-1}0$, а этому массиву, как мы уже убедились, должна соответствовать строка $\boldsymbol{x}^*_{n-1}\lambda_k$. ■

Отметим, что лемма 4.3.5 не обязана выполняться для конечного алфавита Λ_α — возможно, она будет выполняться, если алфавит достаточно большой. Например, на алфавите $\Lambda_3 = \{\lambda_1, \lambda_2, \lambda_3\}$ b-каноническая строка $\boldsymbol{x}^*_7 = \lambda_1\lambda_2\lambda_1\lambda_3\lambda_1\lambda_2\lambda_1$ имеет массив граней $\beta_7 = 0010123$, но не существует строки $\boldsymbol{x}^*_8 = \boldsymbol{x}^*_7\lambda$ с массивом граней вида $\boldsymbol{\beta}_8 = 00101230$.

Леммы 4.3.3–4.3.5 составляют основу алгоритма генерирования b-канонических строк длиной n: к каждой b-канонической строке $\boldsymbol{x}^*_j$ ($j = 1, 2, \ldots, n-1$) добавляется по одной букве λ стандартного алфавита $\{\lambda_1, \lambda_2, \ldots\}$ до тех пор, пока для некоторого k получим строку $\boldsymbol{x}^*_j\lambda_k$, имеющую только пустую грань. Строки $\boldsymbol{x}^*_j\lambda_1$, $\boldsymbol{x}^*_j\lambda_2$, …, $\boldsymbol{x}^*_j\lambda_k$ составляют полное множество b-канонических строк, которые можно получить на основе строки $\boldsymbol{x}^*_j$.

Для реализации этого алгоритма надо уметь генерировать синтаксические деревья T'_n, подобные деревьям T_n, используемым для аналогичных целей в разделе 4.2. В дереве T'_n узлы также будут помечены парой (λ, β), где $\lambda \in \Lambda$, а β — элемент массива граней для буквы λ в строке, определяемой метками узлов на пути от корня дерева до текущего узла. Дерево T'_1 состоит только из корневого узла с меткой $(\lambda_1, 0)$, а для $n \geqslant 2$ дерево T'_n формируется из дерева T'_{n-1} путем добавления к каждому листу (конечному узлу) этого дерева новых конечных узлов с метками (λ_1, β_1), (λ_2, β_2), …, (λ_k, β_k).[4] Таким образом, каждый узел дерева T'_n определяет b-каноническую строку совместно с ее массивом граней. Если обозначить через $b[n]$ количество b-канонических строк длиной n, то легко видеть, что количество листьев в дереве T'_n в точности равно $b[n]$. Итак, все $b[n]$ b-канонических строк (и соответствующих им массивов) можно получить, добавляя $b[n]$ первые потомки (сыновья[5]) к листьям дерева T'_{n-1}.

[4] Здесь важно отметить, что для каждого листа дерева T'_{n-1} значение k будет свое, не обязательно совпадающее с аналогичными значениями для других листьев. — *Примеч. ред.*

[5] "Детско-родительская" терминология широко применяется в теории графов, особенно при описании деревьев, и не должна вызывать затруднений у читателя. Напомним, что в предисловии автор предполагает, что читатель знаком с теорией графов и алгоритмами, выполняемыми над деревьями (алгоритмами прохождения и построения деревьев и т.п.). — *Примеч. ред.*

Обсудим, стоит ли строить дерево T'_n методом построения "в ширину" (так же как строится дерево T_n из предыдущего раздела), когда для вычисления $b[n]$ конечных узлов дерева T'_n предварительно уже построено дерево T'_{n-1}. Однако при построении дерева T'_n таким способом возникает проблема: чтобы вычислить сына данного узла N_j надо выполнить вычисления над массивом граней, для чего необходимо иметь доступ к строке x^*_j и ее массиву граней β_j, что, в свою очередь, требует просмотра всех узлов на пути от корня дерева до узла N_j. На такой просмотр потребуется время порядка $\Theta(j)$. Эта дополнительная работа неприемлема, если мы хотим вычислить сынов узла N_j за константное время в расчете на одного сына — вместо этого каждый из k сынов узла N_j вычисляется в среднем за время порядка $\Theta(j/k)$.

Чтобы избежать просмотра пути до каждого узла N_j, можно строить дерево T'_n методом построения "в глубину". В этом случае правый брат[6] текущего узла N_j добавляется к дереву T'_n только в том случае, когда в это дерево введены все сыновья узла N_j. В этом случае до завершения всех вычислений в узле N_j требуется хранить в памяти строки $\boldsymbol{x}^*$ и соответствующие им массивы β , необходимые для этих вычислений. Например, при построении дерева T'_3, показанного на рис. 4.2, узлы вводятся в порядке $ABDECFG$, а строки $\boldsymbol{x}^*$ и массивы β принимают соответственно такие значения:

$$\boldsymbol{x}^* = \lambda_1, \lambda_1\lambda_1, \lambda_1\lambda_1\lambda_1, \lambda_1\lambda_1\lambda_2, \lambda_1\lambda_2, \lambda_1\lambda_2\lambda_1, \lambda_1\lambda_2\lambda_2;$$
$$\beta = 0, 01, 012, 010, 00, 001, 000.$$

В этом методе построения дерева "в глубину" реализуется симметричный способ прохождения дерева, когда каждый его узел посещается не более двух раз. Подробности этого метода построения дерева предложено показать в упражнении 4.3.5.

Напомним, что при доказательстве теоремы 1.3.3 мы показали, что для любой строки длиной n внутренний цикл **while** в алгоритме 1.3.1 (это алгоритм вычисления массива граней) выполняется не более $n-2$ раза. В контексте построения дерева T'_n это означает, что любой путь длиной n в этом дереве обязательно заканчивается листом. Поэтому для вычисления массива граней для n строк (префиксов), соответствующих этому пути, необходимо время, не превышающее $\Theta(n)$. Тогда для вычисления каждого узла на этом пути *в среднем* затрачивается $\Theta(n)/n$ времени, т.е. константное (фиксированное) время. Поскольку в дереве T'_n содержится $b[n]$ конечных узлов, на вычисление каждого из которых требуется константное время, то отсюда получаем следующее утверждение, аналогичное утверждению теоремы 4.2.3.

[6] "Братьями" называются узлы дерева, имеющие общего "родителя". — *Примеч. ред.*

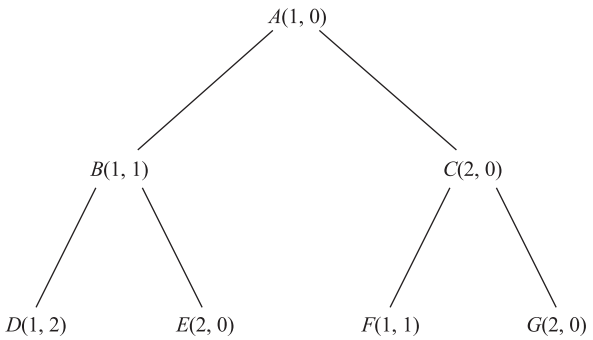


Рис. 4.2. Дерево T'_3 для вычисления всех b-различных строк, длина которых не превышает 3

Теорема 4.3.6. Для любого положительного целого n все b-канонические строки длиной n можно вычислить за время порядка $\Theta(b[n])$, для чего необходима память объемом $\Theta(b[n])$. ■

Простая модификация описанного алгоритма позволит вычислить все b-канонические строки длиной n, определенные на *конечном* алфавите Λ_α. Вместо условия введения новых сынов некоторого узла до тех пор, пока последний элемент массива граней будет равен нулю, применяется новое условие — новые сыны некоторого узла вводятся до тех пор, пока *или* последний элемент массива граней будет равен нулю *или* будут использованы все α букв. Как показано в работе [90], такой алгоритм можно легко модифицировать для того, чтобы он генерировал специальные деревья, определяемые заданным массивом граней. Однако такой алгоритм имеет существенный недостаток — он не является онлайновым, т.е. дерево T'_{n+1} нельзя легко сгенерировать на основании ранее построенного дерева T'_n. От этого недостатка данного алгоритма недавно удалось избавиться путем введения некоторых усовершенствований и упрощений в "механике" алгоритма [79].

Далее рассмотрим b'-массив, аналогичный p'-массиву из раздела 4.2. Для положительных целых чисел α и $nb'[\alpha, n]$ обозначает количество b-канонических строк длиной n, сформированных с использованием всех α букв алфавита Λ_α. Обозначим через $b[n]$ сумму элементов в n-м столбце b'-массива, т.е. $b[n] = \sum_{\alpha \geqslant 1} b'[\alpha, n]$. В общем случае найти замкнутую формулу для вычисления элементов b'-массива труднее, чем для p'-массива. Следующая теорема содержит информацию, которая поможет установить верхние границы для значений $b'[\alpha, n]$ и $b[n]$.

Теорема 4.3.7. Для любых положительных целых α и n

а) если $\alpha > \lceil \log_2(n+1) \rceil$, тогда $b'[\alpha, n] = 0$;

б) $b'[1, n] = b'[\alpha, 2^{\alpha-1}] = 1$;

в) $b'[2,n] = p'[2,n] = 2^{\alpha-1} - 1$;

г) для любого положительного целого j и $\alpha > 2$

$$b'[\alpha, 2^{\alpha-1} + j] < p'[\alpha, \alpha + j].$$

Доказательство.

а) Доказательство этого утверждения проведем методом математической индукции. Сначала покажем, что утверждение выполняется для $n = 1$. Так как одна буква λ_1 является b-канонической строкой, то $p'[1,1] = 1$. Тогда для любого $\alpha > 1$ $p'[\alpha, 1] = 0$, поскольку не существует строк длиной 1, сформированных из $\alpha > 1$ букв.

Теперь предположим, что утверждение справедливо для некоторого положительного целого α и для всех n, удовлетворяющих неравенствам $2^{\alpha-1} \leqslant \leqslant n \leqslant 2^{\alpha} - 1$. Покажем, что в этом случае утверждение справедливо и для всех значений n', удовлетворяющих неравенствам $2^{\alpha} \leqslant n' \leqslant 2^{\alpha+1} - 1$.

Из определения b'-массива следует, что предположение индукции эквивалентно утверждению о том, что для всех n, удовлетворяющих неравенствам $2^{\alpha-1} \leqslant n \leqslant 2^{\alpha} - 1$, для формирования b-канонических строк $\boldsymbol{x}_n$, соответствующих любому массиву граней β_n, используется не более α букв λ_1, λ_2, $\ldots$, λ_α (перечислены в возрастающем порядке). Поэтому в любой b-канонической строке $x^*_{n'}$, $n' \geqslant 2^{\alpha}$, буква $\lambda_{\alpha+1}$ не может встретиться в позиции, номер которой меньше 2^{α}.

Нам необходимо показать, что для любого n', удовлетворяющего неравенствам $2^{\alpha} \leqslant n' \leqslant 2^{\alpha+1} - 1$, не существует b-канонической строки $\boldsymbol{x}^*_{n'}$, содержащей букву $\lambda_{\alpha+2}$. Предположим противное — некоторая строка $x^*_{n'}$ содержит букву $\lambda_{\alpha+2}$ в качестве завершающей строку, т.е. $\boldsymbol{x}^*_{n'} = \boldsymbol{x}^*_{n'-1}\lambda_{\alpha+2}$. Такая строка может существовать только в том случае, если каждая из строк

$$\{\boldsymbol{x}^*_{n'-1}\lambda_1, \boldsymbol{x}^*_{n'-1}\lambda_2, \ldots, \boldsymbol{x}^*_{n'-1}\lambda_{\alpha+1}\}$$

будет b-канонической с непустой гранью. Рассмотрим строку $\boldsymbol{y} = \boldsymbol{x}^*_{n'-1}$ $\lambda_{\alpha+1}$. Пусть буква $\lambda_{\alpha+1}$ в этой строке впервые встречается в позиции j. По предположению индукции $j \geqslant 2^{\alpha}$, следовательно, длина наибольшей грани строки $\boldsymbol{y}$ должна превышать $n'/2$. Но тогда $\boldsymbol{y}[j - (n' - j)] = \lambda_{\alpha+1}$, что противоречит тому, что буква $\lambda_{\alpha+1}$ впервые встречается в позиции j. Отсюда следует, что строка $\boldsymbol{x}^*_{n'-1}\lambda_{\alpha+1}$ не может иметь непустую грань. Поэтому на основании леммы 4.3.5 делаем заключение, что не существует b-канонической строки $\boldsymbol{x}^*_{n'}$, содержащей букву $\lambda_{\alpha+2}$, что и требовалось доказать.

б) Значение $b'[1,n] = 1$ соответствует строке λ_1^n, значение $b'[\alpha, 2^{\alpha-1}] = 1$ — строкам

$$X = \{\lambda_1, \lambda_1\lambda_2, \lambda_1\lambda_2\lambda_1\lambda_3, \lambda_1\lambda_2\lambda_1\lambda_3\lambda_1\lambda_2\lambda_1\lambda_4, \ldots\}.$$

Обозначим через $\boldsymbol{x}_\alpha$ произвольную строку из множества X, $\alpha = 1, 2, \ldots$, и представим ее как $\boldsymbol{x}_\alpha = \boldsymbol{x}'_\alpha \lambda_\alpha$. Нетрудно заметить, что любую строку из множества X можно генерировать последовательно с помощью соотношения $\boldsymbol{x}_\alpha = \boldsymbol{x}_{\alpha-1}\boldsymbol{x}'_{\alpha-1}\lambda_\alpha$, где по определению $\boldsymbol{x}_0 = \boldsymbol{x}'_0 = \varepsilon$. Все строки $\boldsymbol{x}_\alpha$ имеют только пустые грани. Кроме того, при $\alpha \geqslant 2$ они обладают тем свойством, что строки $\boldsymbol{x}'_\alpha \lambda_i$, $1 \leqslant i \leqslant \alpha$, имеют непустые грани. Из леммы 4.3.5 следует, что b-канонические строки длиной $n = 2^{\alpha-1}$ должны заканчиваться буквой λ_α. Но существует только одна b-каноническая строка с таким свойством. Поэтому $b'[\alpha, 2^{\alpha-1}] = 1$.

в) Заметим, что b-канонические строки $\boldsymbol{x}^*_n$ формируются из строк $\boldsymbol{x}^*_{n-1}$ путем присоединения к последним не менее двух различных букв. Присоединение к $\boldsymbol{x}^*_{n-1}$ буквы всегда приводит к появлению грани максимальной длины $\beta^* \geqslant 1$. Из леммы 4.3.5 следует, что только присоединение буквы λ_2 приведет к построению строки $\boldsymbol{x}^*_n$, имеющей пустую грань. Но при $\alpha = 2$ к строкам $\boldsymbol{x}^*_{n-1}$ можно присоединить *не более* двух букв. Поэтому в данном случае каждая p-каноническая строка является также b-канонической. Поскольку лемма 4.3.2 утверждает, что каждая b-каноническая строка с необходимостью является p-канонической, то отсюда получаем требуемый результат.

г) Для $\alpha \geqslant 3$ рассмотрим одну из b-канонических строк $\boldsymbol{x}_\alpha$ длиной $n = 2^{\alpha-1}$, введенных при доказательстве утверждения *б*) данной теоремы. Такая строка является p-канонической и в дереве T_n порождает α p-канонических строк путем присоединения букв λ_1, λ_2, $\ldots$, λ_α. Однако только две b-канонические строки можно сформировать на основе $\boldsymbol{x}_\alpha$: $\boldsymbol{x}_\alpha\lambda_1$ и $\boldsymbol{x}_\alpha\lambda_2$. Но тогда поддерево с корнем в узле, соответствующем, например, строке $\boldsymbol{x}_\alpha\lambda_3$, будет принадлежать дереву T_n, но не будет входить в дерево T'_n. Опять принимая во внимание тот факт, что каждая строка является p-канонической, приходим к выводу, что для любого $j > 1$ выполняется неравенство $b'[\alpha, n+j] < p'[\alpha, \alpha+j]$. Утверждение доказано. ■

В табл. 4.2 приведены значения верхнего левого угла b'-массива. Чтобы еще раз оценить преимущества от использования p- или b-канонических строк в качестве тестовых данных вместо различных в обычном понимании строк, отметим, например, что на 4-буквенном алфавите существует $4^9 = 262\,144$ различных строк, тогда как $p'[4, 9] = 7770$ (см. табл. 4.1) и из табл. 4.2 имеем $b'[4, 9] = 4$.

В общем случае мы уже видели (равенство (4.5)), что сумма $\sum_{i=1}^{\alpha} p'[i, n]$ асимптотически меньше величины α^n в $\alpha!$ (или больше) раз. Утверждение *г*) теоремы 4.3.7 говорит о том, что значения $b'[\alpha, n]$ в свою очередь значительно меньше значений $p'[\alpha, n]$. Фактически, это утверждение дает верхнюю границу для $b[n]$, выраженную посредством элементов p'-массива: для любого положительного це-

лого n имеем

$$b[n] \leqslant \sum_{k=1}^{k^*} \sigma^{(k)}_{n-2^{k-1}+k}, \tag{4.10}$$

где $k^* = \lceil \log_2(n+1) \rceil$. Это неравенство будет строгим уже при $n \geqslant 3$.

До настоящего времени не найдена замкнутая формула для $b[n]$, но в работе [G01] доказан впечатляющий результат, что $b[n] \leqslant f_{2n+1} < \phi^{2n}$, где f_{2n+1} — длина строк Фибоначчи $\boldsymbol{f}_{2n+1}$, а $\phi = (1+\sqrt{5})/2 \approx 1{,}62$ — отношение золотого сечения [135].

Таблица 4.2. Значения b'-массива для строк длиной $n \leqslant 10$ и $\alpha \leqslant 4$

n	1	2	3	4	5	6	7	8	9	10
$b'[1,n]$	1	1	1	1	1	1	1	1	1	1
$b'[2,n]$	0	1	3	7	15	31	63	127	255	511
$b'[3,n]$	0	0	0	1	4	15	46	134	370	997
$b'[4,n]$	0	0	0	0	0	0	0	1	4	16
$b[n]$	1	2	4	9	20	47	110	263	630	1525

Упражнения 4.3

1. Докажите лемму 4.3.2.
2. Докажите, что отношение b-эквивалентности действительно является отношением эквивалентности.
3. Постройте дерево T'_5 и убедитесь, что оно содержит 20 узлов.
4. Докажите по индукции, что $b[n] \geqslant \sum_{1 \leqslant i \leqslant n} b[i]$.
5. Напишите рекурсивный алгоритм построения дерева T'_n с константным временем для каждого узла.
6. Докажите неравенство (4.10).

ЧАСТЬ II

Вычисление внутренних паттернов

И Слово стало плотью и обитало с нами, полное благодати и истины; и мы видели славу Его…

– Иоанн 1.14

ГЛАВА 5

Деревья для строковых последовательностей

. . . все слова,
но нет свершений!

– Филип Мессинджер (1583–1640). Парламент любви

Мы начинаем изучение внутренних паттернов с рассмотрения деревьев двух типов — деревьев граней и деревьев суффиксов, которые естественны для представления строковых последовательностей и возникают во многих приложениях. Мы отложим до раздела 13.1 рассмотрение другого внутреннего паттерна, дерева оболочек, который, подобно дереву граней, содержит информацию о периодических свойствах строк. В этой главе мы также рассмотрим две древовидные структуры суффиксов: ориентированные ациклические графы слов и массивы суффиксов.

5.1 Деревья граней

Как мы видели в разделе 1.2, наибольшие грани, присутствующие в строках, позволяют представить строки в нормальной форме. В разделе 1.3 показано, что все грани префиксов, т.е. массив граней, можно вычислить за линейное время, а сам массив используется для определения периодов и нормальных форм этих префиксов. В главе 1 понятие граней использовалось в доказательстве леммы периодичности (теорема 1.3.5) и для определения циклических сдвигов некратных

строк, что в главе 3 пригодилось при исследовании различных типов строк. В главе 4 эти понятия помогли в процессе разработки данных для тестирования строковых алгоритмов. Мы увидим, что грани естественны в алгоритмах вычисления частных паттернов (главы 7 и 8), кратных подстрок (подраздел 12.1.2) и оболочек (раздел 13.1).

Таким образом, массивы граней для внутренних паттернов представляют несомненный интерес, а как указывалось в разделе 2.1, эти массивы естественно представлять в виде древовидных структур. Здесь мы ограничимся только примером такой структуры для строк Фибоначчи, которая уже использовалась при пояснении определения 2.3.1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\boldsymbol{f}_7 =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8

На рис. 5.1 показан массив граней β для данного примера, представленный в виде дерева. Отметим, что грани строк $\boldsymbol{f}_7[1..i]$ определяются предками узлов i на этом дереве. Например, гранями строки $\boldsymbol{f}_7[1..18]$ являются подстроки $\boldsymbol{f}_7[1..10]$, $\boldsymbol{f}_7[1..5]$, $\boldsymbol{f}_7[1..2]$ и $\boldsymbol{f}_7[1..0] = \varepsilon$.

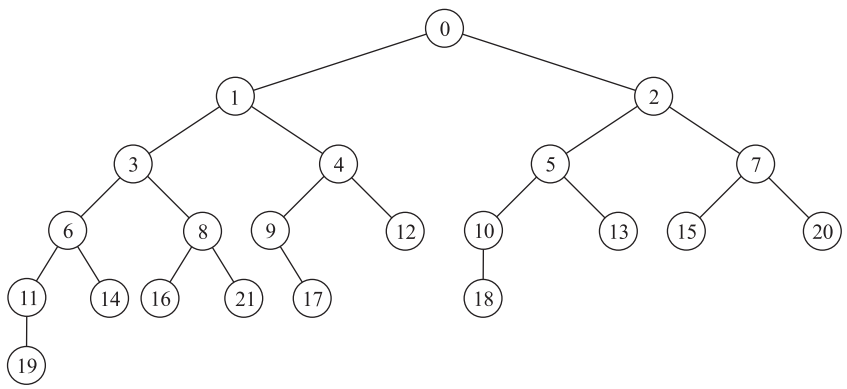


Рис. 5.1. Дерево граней для $\boldsymbol{f}_7$

Упражнения 5.1

1. Эффективное прохождение дерева граней требует линейного времени, если проходить дерево снизу вверх, т.е. от сынов к родителям. Разработайте экономичную (по критерию объема памяти) структуру данных для представления этого дерева, которая дополнительно позволяла бы для любого родителя получать доступ к его сынам за константное время в расчете на

одного сына. ("Экономичность по критерию объема памяти" здесь обозначает, что общая требуемая память не должна превышать величины $3(n+1)$.) Затем измените алгоритм 1.3.1 с учетом новой структуры данных таким образом, чтобы время выполнения алгоритма было линейным относительно длины строк.

2. Покажите, что дерево граней для строк Фибоначчи всегда будет бинарным, т.е. каждый узел будет иметь не более двух сынов.

3. Покажите, что в дереве граней поддерево с корнем в узле $i \geqslant 1$ определяет все раппорты подстрок $\boldsymbol{x}[1..i]$ в строке $\boldsymbol{x}$.

5.2 Деревья суффиксов

Как указывалось в разделе 2.1, деревья суффиксов имеют многочисленные применения в задачах со строковыми последовательностями [12]. Наиболее часто они используются на "подготовительном этапе" для того, чтобы заменить исходную входную строку $\boldsymbol{x}$ на дерево суффиксов $T_{\boldsymbol{x}}$. Дерево $T_{\boldsymbol{x}}$ можно эффективно применить для поиска вхождений заданной подстроки $\boldsymbol{u}$, определенной на алфавите A, в строку $\boldsymbol{x}$ с временны́ми затратами порядка $O(|\boldsymbol{u}| \log \alpha)$, что особенно эффективно, когда $|\boldsymbol{u}|$ значительно меньше $|\boldsymbol{x}|$, $\alpha = |A|$ мало, а строка $\boldsymbol{x}$ фиксированна. Такие условия выполняются, например, в последовательностях ДНК.

Как показано в разделе 2.1 (и упражнении 2.1.7), для хранения дерева $T_{\boldsymbol{x}}$ необходима память объемом $\Theta(nf(\alpha))$, где $f(\alpha)$ — некоторая функция от α или малая по величине константа, зависящая от природы алфавита. В этом разделе мы покажем, что дерево $T_{\boldsymbol{x}}$ можно построить за время порядка $O(n \log n)$, когда алфавит упорядочен, и за время $\Theta(n)$, если алфавит индексирован (вернитесь к разделу 4.1 для объяснения такого различия).

В этом разделе представлены три алгоритма построения дерева суффиксов, которые будут описаны после предварительного подраздела, где поговорим о деревьях суффиксов "в общем". Два из этих алгоритмов, разработанные Мак-Крейтом (McCreight, [174]) и Укконеном (Ukkonen, [225]), выполняются на упорядоченных алфавитах, а алгоритм Фарача (Farach, [86]) работает на индексированных алфавитах. Существует еще один алгоритм построения дерева суффиксов — в разделе 12.1.1 мы обнаружим, что такое дерево появляется как побочный продукт алгоритма вычисления непродолжаемых вправо раппортов. Алгоритм Мак-Крейта построен на основе более раннего алгоритма Винера [230]. Эти алгоритмы имеют одинаковое время выполнения, порядка $O(n \log \alpha)$, однако алгоритм Винера более сложный и требует большего объема памяти, поэтому здесь он рассматриваться не будет. Однако отметим, что модифицированная версия алгоритма Винера [49] позволяет производить поиск обратных подстрок в заданной строке $\boldsymbol{x}$. В недав-

ней работе [11] предложен алгоритм построения особого типа дерева суффиксов, который находит применение в анализе текста и последовательностей ДНК.

В этом и следующем разделах предполагается, что объем α алфавита конечен, но это не ограничивает возможностей применения описанных алгоритмов: если алфавит бесконечен, то в качестве значения α можно принять число различных букв, фактически принимающих участие в формировании данной исследуемой строки $\boldsymbol{x}$. Однако подсчет количества различных букв, входящих в строку $\boldsymbol{x}$, также требует определенных затрат, что показано в разделе 4.1 на примере задачи 4.1.

5.2.1 Предварительные сведения о деревьях суффиксов

Алгоритмы, описанные далее в этом подразделе, используют специальные способы эффективного прохождения дерева суффиксов после его построения. В этом подразделе мы покажем способы обхода дерева суффиксов и представим предварительные соображения об их роли в этих алгоритмах.

Начнем с определения ***наибольшего общего префикса*** $\boldsymbol{u}$ для двух данных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ — это наибольшая строка $\boldsymbol{u}$, которая является префиксом обеих строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Для наибольшего общего префикса введем обозначение $\boldsymbol{u} = \mathrm{LCP}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.[1] Пусть строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ являются суффиксами некой строки $\boldsymbol{x}$, т.е. $\boldsymbol{x}_1 = \boldsymbol{x}[i..n]$ и $\boldsymbol{x}_2 = \boldsymbol{x}[j..n]$. В этом случае наибольший общий префикс для таких строк будем обозначать как $\mathrm{LCP}(i, j)$. Например, для строки Фибоначчи

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f}_6 =$	a	b	a	a	b	a	b	a	a	b	a	a	b

имеем $\mathrm{LCP}(baababaabaab, baabaab) = \mathrm{LCP}(2, 7) = baaba$, тогда как $\mathrm{LCP}(3, 7) = \varepsilon$ и $\mathrm{LCP}(5, 7) = ba$.

Этот простой пример позволит сделать нам элементарное, но очень важное наблюдение. Пусть строка $\boldsymbol{u}$ определяет узел дерева $T_{\boldsymbol{x}}$ и пусть этот узел является корнем минимального поддерева, содержащего суффиксы $\boldsymbol{x}[i..n]$ и $\boldsymbol{x}[j..n]$ строки $\boldsymbol{x}$. В этом случае $\boldsymbol{u}$ будем называть ***наименьшим общим предком*** этих суффиксов и будем обозначать как $\boldsymbol{u} = \mathrm{LCA}(i, j)$.[2] Поскольку строка $\boldsymbol{u}$ определяет наибольший путь от корня дерева $T_{\boldsymbol{x}}$ до соответствующего узла, отсюда немедленно получаем определяющее свойство дерева суффиксов: для любой пары суффиксов $\boldsymbol{x}[i..n]$ и $\boldsymbol{x}[j..n]$ выполняется равенство

$$\mathrm{LCA}(i, j) = LCP(i, j). \tag{5.1}$$

Мы покажем практическое применение этого соотношения при описании алгоритма Фарача (подраздел 5.2.4).

[1] LCP обозначает Longest Common Prefix, т.е. наибольший общий префикс. — *Примеч. ред.*

[2] LCA обозначает Lowest Common Ancestor, т.е. наименьший общий предок. — *Примеч. ред.*

После построения дерева суффиксов для строки $\boldsymbol{x}$ необходимо для каждого суффикса $\boldsymbol{x}[i..n]\$$ определить узлы $\mathrm{LCP}(i,j)$, в которых путь по дереву $T_{\boldsymbol{x}}$, определяющий префикс $\boldsymbol{u}$, ветвится на пути к суффиксу $\boldsymbol{x}[i..n]\$$ и на пути к другим суффиксам $\boldsymbol{x}[j..n]\$$. В алгоритме Мак-Крейта, где суффиксы вычисляются в естественном порядке $i = 1, 2, \ldots, n$, наибольший общий префикс вычисляется на множестве *наибольших* суффиксов $\boldsymbol{x}[j..n]\$$, $j < i$, как строка, имеющая *максимальную* длину. В этом случае $\mathrm{LCP}(i,j)$ определяет самый низкий узел в дереве $T_{\boldsymbol{x}}$, в котором путь к суффиксу $\boldsymbol{x}[i..n]\$$ имеет ответвление к любому ранее вычисленному суффиксу. Поэтому разобьем суффикс, определяющий конечный узел i ($1 \leqslant i \leqslant n+1$) дерева $T_{\boldsymbol{x}}$, на две части:[3]

$$\boldsymbol{x}[i..n]\$ = \mathrm{head}(i)\,\mathrm{tail}(i),$$

где $\mathrm{head}(1) = \varepsilon$ и для $i \geqslant 2$ $\mathrm{head}(i) = \mathrm{LCP}(i,j)$ максимальной длины, вычисленный по всем $j : 1 \leqslant j \leqslant i-1$.[4] Как упоминалось в разделе 2.1 и как видно на рис. 5.2, маркером окончания строки является символ \$, показывающий, что подстрока $\mathrm{tail}(i)$ не пустая. Это соглашение выполняется и в случае $i = n+1$ — здесь строка $\boldsymbol{x}[n+1..n]\$$ дает $\mathrm{head}(n+1) = \varepsilon$ и $\mathrm{tail}(n+1) = \$$.

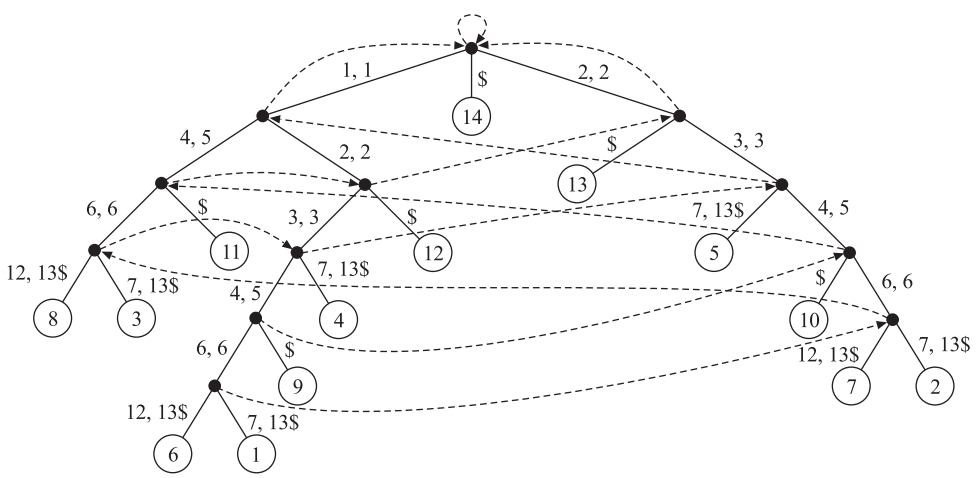


Рис. 5.2. Связное дерево суффиксов для строки Фибоначчи $\boldsymbol{f}_6$

[3] Здесь и далее автор часто отождествляет узел и подстроку, соответствующую этому узлу. Поэтому можно встретить выражения, подобные тому, что один узел является префиксом другого узла или что одна подстрока является предком другой подстроки и т.п. Надеемся, что такая "взаимоподмена" понятий не вызовет у читателя затруднений (хотя вполне обоснованное раздражение вызвать может). — *Примеч. ред.*

[4] Название "head" означает "голова" или "передняя часть", а "tail" — "хвост" или "задняя часть". — *Примеч. пер.*

Заметим, что подстрока $\mathrm{head}(i)$ никогда не будет соответствовать конечному узлу. Интуитивно ясно, что $\mathrm{head}(i)$ является отправной точкой (префиксом), от которой начинается поиск в $T_{\boldsymbol{x}}$ суффикса $\boldsymbol{x}[i..n]\$$. В табл. 5.1 приведены значения подстрок head(i) для суффиксов строки Фибоначчи $\boldsymbol{f}_6$ (см. также рис. 5.2).

Таблица 5.1. Значения подстрок $\mathrm{head}(i)$ для суффиксов строки Фибоначчи f_6

i	$\mathbf{head}(i)$	i	$\mathbf{head}(i)$
1	ε	8	$aaba$
2	ε	9	$abaab$
3	a	10	$baab$
4	aba	11	aab
5	ba	12	ab
6	$abaaba$	13	b
7	$baaba$	14	ε

Очевидно, что подстроке $\mathrm{tail}(i)$ в дереве $T_{\boldsymbol{x}}$ соответствует путь от внутреннего узла, определяемого $\mathrm{head}(i)$, до конечного узла. В упражнении 5.2.1 предложено показать, что $\mathrm{tail}(i)$ *всегда* соответствует конечному узлу. Например, на рис. 5.2 видно, что подстроке $\mathrm{tail}(5) = baabaab\$$ соответствует конечный узел 7.

Следующая лемма, используемая в алгоритме Мак-Крейта, показывает, как на основе известного значения $\mathrm{head}(i)$ можно локализовать в дереве $T_{\boldsymbol{x}}$ строку $\mathrm{head}(i+1)$. Если говорить неформально, то эта лемма утверждает, что наибольший собственный суффикс строки $\mathrm{head}(i)$ является префиксом строки $\mathrm{head}(i+1)$ и поэтому будет предком строки $\mathrm{head}(i+1)$ в дереве $T_{\boldsymbol{x}}$.

Лемма 5.2.1. Пусть $\mathrm{head}(i) = \boldsymbol{x}[i..i+h]$ для некоторых целых i, $1 \leqslant i \leqslant n$, и h, $-1 \leqslant h \leqslant n-i$. Тогда подстрока $\boldsymbol{x}[i+1..i+h]$ является префиксом строки $\mathrm{head}(i+1)$.

Доказательство. Утверждение леммы тривиально, когда $i+h \leqslant i$, поскольку в этом случае строка $\boldsymbol{x}[i+1..i+h] = \varepsilon$ является префиксом любой строки, в том числе и строки $\mathrm{head}(i+1)$. Поэтому рассмотрим случай, когда $i+h > i$. Обозначим $\lambda = \boldsymbol{x}[i]$ и $\boldsymbol{u} = \boldsymbol{x}[i+1..i+h]$, тогда $\mathrm{head}(i) = \lambda\boldsymbol{u}$. По определению подстрок head существует номер $j \in 1..i-1$ такой, что строка $\lambda\boldsymbol{u}$ является префиксом как суффикса $\boldsymbol{x}[i..n]$, так и суффикса $\boldsymbol{x}[j..n]$. Но тогда строка $\boldsymbol{u}$ также является префиксом обоих суффиксов $\boldsymbol{x}[i+1..n]$ и $\boldsymbol{x}[j+1..n]$ и поэтому должна быть префиксом строки $\mathrm{head}(i+1)$. ■

Отметим, что все подстроки $\mathrm{head}(i)$ для строки $\boldsymbol{f}_6$, приведенные в табл. 5.1, удовлетворяют лемме 5.2.1.

На основе доказанной леммы можно ввести другой способ обхода дерева $T_{\boldsymbol{x}}$, который будет востребован в алгоритмах Мак-Крейта и Укконена. Определим ***суффиксную функцию*** s для любого узла $\boldsymbol{u}$ дерева $T_{\boldsymbol{x}}$ следующим образом:

- для $\boldsymbol{u} = \varepsilon$, $s(\boldsymbol{u}) = \varepsilon$;
- для $\boldsymbol{u} \neq \varepsilon$ (в этом случае $\boldsymbol{u} = \lambda \boldsymbol{v}$ с непустой буквой $\lambda \in A$) $s(\boldsymbol{u}) = \boldsymbol{v}$.

Суффиксная функция является указателем от любого непустого (конечного или внутреннего) узла $\boldsymbol{u}$ дерева $T_{\boldsymbol{x}}$ к узлу $\boldsymbol{v} = s(\boldsymbol{u})$, который является наибольшим собственным суффиксом строки $\boldsymbol{u}$. В упражнении 5.2.3 предложено доказать, что если u является узлом дерева $T_{\boldsymbol{x}}$, то узлом этого дерева будет и $s(\boldsymbol{u})$. Отметим, что если $\boldsymbol{u}$ является конечным узлом $i \leqslant n$, тогда $s(\boldsymbol{u})$ также является конечным узлом $i+1$. Если же $\boldsymbol{u}$ — конечный узел $n+1$, тогда $s(\boldsymbol{u}) = \varepsilon$. Когда $\boldsymbol{u} = \mathrm{head}(i)$, то, вследствие леммы 5.2.1, $s(\boldsymbol{u})$ будет префиксом $\mathrm{head}(i+1)$.

Дерево суффиксов, включающее связи между узлами, определяемые суффиксной функцией s, называется ***связным деревом суффиксов***. На рис. 5.2 представлено связное дерево суффиксов для строки Фибоначчи $\boldsymbol{f}_6$ (показаны связи только для внутренних узлов дерева).

Теперь рассмотрим механизм поиска непустой строки $\boldsymbol{v}$, при этом поиск начинается из заданного узла $\boldsymbol{u}$ дерева $T_{\boldsymbol{x}}$ (узел $\boldsymbol{u}$ назовем ***префиксным узлом***). Другими словами, для данного префиксного узла $\boldsymbol{u}$, который уже локализован в дереве $T_{\boldsymbol{x}}$, необходимо найти префикс $\boldsymbol{uv}$ строки $\boldsymbol{x}$. Такая ситуация представлена на рис. 5.3, где узел $\boldsymbol{u}$ имеет k различных сынов $\boldsymbol{uu}_i$, $i = 1, 2, \ldots, k$, и $\boldsymbol{u}_i = \lambda_i \boldsymbol{u}'_i$ с попарно различными буквами λ_i. Мы ищем совпадение $\boldsymbol{u}_i$ со строкой $\boldsymbol{v} = \lambda \boldsymbol{v}'$.

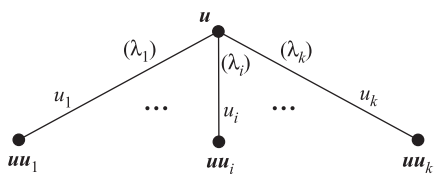


Рис. 5.3. Сканирование дерева суффиксов

Если мы не знаем, является ли в действительности $\boldsymbol{uv}$ префиксом строки $\boldsymbol{x}$, то в этом случае поочередно просматриваются метки ребер λ_i, и при обнаружении совпадения $\lambda = \lambda_i$ продолжается побуквенное сравнение $\boldsymbol{v}'$ с метками ребер, соответствующих строке u'_i. Такой процесс назовем ***медленным сканированием***.

Если известно, что $\boldsymbol{uv}$ действительно является префиксом строки $\boldsymbol{x}$, то в таком случае надо только определить точную позицию $\boldsymbol{uv}$ в строке $\boldsymbol{x}$ (или ее положение в дереве $T_{\boldsymbol{x}}$). Для этого не обязательно производить полное побуквенное сравнение $\boldsymbol{v}'$ и u'_i, поскольку после нахождения совпадения $\lambda = \lambda_i$ и определения $\boldsymbol{u}_i$ имеются две возможности.

1. $|\boldsymbol{v}| > |\boldsymbol{u}_i|$. Тогда $\boldsymbol{v} = \boldsymbol{u}_i\boldsymbol{v}''$ для некоторой непустой строки $\boldsymbol{v}''$, и поиск совпадений продолжается рекурсивно для $\boldsymbol{v}''$ с новым префиксным узлом $\boldsymbol{u}\boldsymbol{u}_i$.
2. $|\boldsymbol{v}| \leqslant |\boldsymbol{u}_i|$. Тогда $\boldsymbol{v} = \boldsymbol{u}_i[1..|\boldsymbol{v}|]$, и поиск заканчивается.

Поскольку значение $|\boldsymbol{u}_i|$ можно получить из метки ребра, поэтому определить, какая из двух описанных возможностей в действительности имеет место, можно за константное время. Таким образом, время поиска строки $\boldsymbol{v}$ не пропорционально $|\boldsymbol{v}|$, а значительно меньше и зависит от количества просмотренных узлов. Поэтому такой процесс назовем ***быстрым сканированием***. Как увидим далее, использование быстрого сканирования является ключевым фактором уменьшения времени выполнения алгоритмов Мак-Крейта и Укконена.

5.2.2 Алгоритм Мак-Крейта

Алгоритм Мак-Крейта (будем для краткости называть его алгоритмом МК) начинается с построения дерева T_1', содержащего только два узла: корень ε и конечный узел 1, соответствующий суффиксу $\boldsymbol{x}\$ = \boldsymbol{x}[1..n]\$$. Эти узлы связаны ребром, помеченным $\boldsymbol{x}\$$. Дерево T_1' также содержит суффиксную связь для корня $s(\varepsilon) = \varepsilon$.

Далее для любого $i = 1, 2, \ldots, n$ алгоритм МК вычисляет новое дерево T_{i+1}' на основе уже построенного дерева T_i' путем добавления суффикса $\boldsymbol{x}[i+1..n]\$$. Эти суффиксы добавляются в убывающем порядке их длин, и поскольку конечных узлов $i+2, i+3, \ldots, n+1$ *нет* в дереве T_{i+1}', полное дерево суффиксов $T_{\boldsymbol{x}}$ будет построено только на шаге $n+1$, т.е. $T_{\boldsymbol{x}} = T_{n+1}'$.

На i-й итерации в алгоритме МК на основе дерева T_i' выполняются следующие операции.

1. Добавляется конечный узел $i+1$.
2. Добавляется внутренний узел $\mathrm{head}(i+1)$.
3. Вычисляется метка $\mathrm{tail}(i+1)$ для ребра, идущего от узла $\mathrm{head}(i+1)$ до конечного узла $i+1$.
4. Устанавливается суффиксная связь $s(\mathrm{head}(i))$.

Из последнего пункта видно, что каждое дерево T_{i+1}' является связным, но суффиксные связи вычисляются только для внутренних узлов. После завершения дерева T_{i+1}' суффиксная связь может быть не определена только для внутреннего узла $\mathrm{head}(i+1)$.

Представленные выше операции 1 и 3 тривиальны, если локализован узел $\mathrm{head}(i+1)$, операция 4, как увидим далее, может быть реализована в процессе вычисления $\mathrm{head}(i+1)$. Поэтому основное внимание уделим выполнению операции 2. Но сначала напомним следующие факты.

- Строка $s(\mathrm{head}(i))$ является префиксом (предком в дереве $T_{\boldsymbol{x}}$) строки $\mathrm{head}(i+1)$ (лемма 5.2.1).
- Обозначим через $\mathrm{parent}(\boldsymbol{u})$ родителя в дереве $T_{\boldsymbol{x}}$ непустого узла $\boldsymbol{u}$, тогда или
$$\mathrm{parent}(\boldsymbol{u}) = s(\boldsymbol{u}) = \varepsilon, \tag{5.2}$$
или $s(\mathrm{parent}(\boldsymbol{u}))$ является собственным префиксом строки $s(\boldsymbol{u})$ (доказано в упражнении 5.2.3).
- В дереве T'_{i+1} для непустой строки $\mathrm{head}(i)$ вычислена суффиксная связь $s(\mathrm{parent}(\mathrm{head}(i)))$.

На основе этих фактов заключаем, что можем использовать узел $s(\mathrm{parent}(\mathrm{head}(i)))$ в качестве отправной точки для поиска $\mathrm{head}(i+1)$. Из этого могут быть два исключения.

- Если $\boldsymbol{x}[i]$ является самым левым (первым) вхождением некоторой буквы λ в строку $\boldsymbol{x}$, тогда $\mathrm{head}(i) = \varepsilon$. В этом случае отправной точкой для поиска $\mathrm{head}(i+1)$ будет корень дерева, при этом, очевидно, $\mathrm{parent}(\mathrm{head}(i)) = \varepsilon$.
- Если $\mathrm{head}(i) \neq \varepsilon$, но тем не менее выполняется равенство (5.2), тогда $\mathrm{head}(i) = \lambda^k$ для некоторого положительного целого k. В данном случае отправной точкой для поиска $\mathrm{head}(i+1)$ будет узел λ^k. (Заметим, что при $k = 1$ отправной точкой опять будет корень дерева.)

Из дальнейшего рассмотрения эти два случая исключаем. Обозначим $\boldsymbol{u} = \mathrm{parent}(\mathrm{head}(i))$ и $\boldsymbol{v} = \mathrm{label}(\boldsymbol{u}, \mathrm{head}(i))$ — метка (label) ребра, проходящего от узла $\boldsymbol{u}$ до узла $\mathrm{head}(i)$. В этих обозначениях можно записать $\mathrm{head}(i) = \boldsymbol{uv}$, так что строка $\boldsymbol{w} = s(\boldsymbol{u})\boldsymbol{v}$ должна быть префиксом строки $\mathrm{head}(i+1)$. Тогда, даже если строка $\mathrm{head}(i+1)$ не определена как узел дерева T'_i, подстрока $\boldsymbol{v}$ должна быть просмотрена на некотором нисходящем пути из $s(\boldsymbol{u})$. Заметим, что здесь выполняются условия, необходимые для быстрого сканирования, приведенные в разделе 5.2.1. Поэтому можем формально выразить поиск строки $\boldsymbol{v}$, который начинается от префиксного узла $s(\boldsymbol{u})$, как

$$\boldsymbol{w} \leftarrow fastscan(s(\boldsymbol{u}), \boldsymbol{v}),$$

где быстрое сканирование рассматривается как функция $fastscan$, которая определяет положение в дереве T'_{i+1} строки $\boldsymbol{w} = s(\boldsymbol{u})\boldsymbol{v}$.

Возможны две ситуации, в зависимости от того, является или нет строка $\boldsymbol{w}$ узлом дерева T'_i. Если строка $\boldsymbol{w}$ уже представлена узлом дерева T'_i, тогда на нисходящем пути от узла $\boldsymbol{w}$ существует непустой префикс подстроки $\boldsymbol{x}[|\boldsymbol{w}|+1..n]$, при этом наибольшим таким префиксом будет строка $\mathrm{head}(i+1)$, которую можно вычислить с помощью медленного сканирования. Эту операцию представим в виде функции $slowscan$ (медленное сканирование):

$$\mathrm{head}(i+1) \leftarrow slowscan(\boldsymbol{w}, \mathrm{tail}(i)).$$

Если после вычислений окажется, что строка $\mathrm{head}(i+1)$ еще не представлена узлом дерева, то такой узел надо создать. Отметим, что возможна ситуация, когда $\mathrm{head}(i+1) = \boldsymbol{w}$.

Теперь рассмотрим случай, когда в дереве T_i' нет узла, представляющего строку $\boldsymbol{w}$. Сначала предположим, что $\mathrm{head}(i+1) \neq \boldsymbol{w}$, тогда ребро, выходящее из узла $\boldsymbol{w}$, должно вести к непустому префиксу строки $\mathrm{tail}(i)$. Но, по определению $\mathrm{head}(i)$, из узла, представляющего $\mathrm{head}(i)$, должно выходить не менее двух ребер, ведущих к различным префиксам, одним из которых будет $\mathrm{tail}(i)$. Поскольку $\boldsymbol{w} = s(\mathrm{head}(i))$, тогда из узла $\boldsymbol{w}$ также должно выходить не менее двух ребер, ведущих к различным префиксам. Получили противоречие. Отсюда заключаем, что возможно только равенство $\mathrm{head}(i+1) = \boldsymbol{w}$, которое не требует дальнейшего поиска.

Основываясь на представленных рассуждениях, запишем в явном виде алгоритм Мак-Крейта.

Алгоритм 5.2.1 (Алгоритм Мак-Крейта)

▷ *Вычисление связного дерева суффиксов* $T_{\boldsymbol{x}}$
Построение дерева T_1'
for $i \leftarrow 1$ **to** n **do** ▷ *построение дерева* T_{i+1}' *на основе дерева* T_i'
  $\boldsymbol{u} \leftarrow \mathrm{parent}(\mathrm{head}(i))$ ▷ $\boldsymbol{u} \leftarrow \varepsilon$, *если* $\mathrm{head}(i) = \varepsilon$
  $\boldsymbol{v} \leftarrow \mathrm{label}(\boldsymbol{u}, \mathrm{head}(i))$ ▷ $\boldsymbol{v} \leftarrow \varepsilon$, *если* $\mathrm{head}(i) = \varepsilon$
  if $\boldsymbol{u} \neq \varepsilon$ **then**
    $\boldsymbol{w} \leftarrow fastscan(s(\boldsymbol{u}), \boldsymbol{v})$
  else
    $\boldsymbol{w} \leftarrow \boldsymbol{v}[1..|\boldsymbol{v}| - 1]$
  if $\boldsymbol{w}$ — новый узел **then**
    $\mathrm{head}(i+1) \leftarrow \boldsymbol{w}$
    добавить $\mathrm{head}(i+1)$ в дерево T_i'
  else
    $\mathrm{head}(i+1) \leftarrow slowscan(\boldsymbol{w}, \mathrm{tail}(i))$
    if $\mathrm{head}(i+1)$ — новый узел **then**
      добавить $\mathrm{head}(i+1)$ в дерево T_i'
  $s(\mathrm{head}(i)) \leftarrow \boldsymbol{w}$
  добавить конечный узел $i+1$ в дерево T_i'
  $\mathrm{label}(\mathrm{head}(i+1), i+1) \leftarrow \mathrm{tail}(i+1)$

Теорема 5.2.2. Алгоритм 5.2.1 корректно вычисляет связное дерево суффиксов $T_{\boldsymbol{x}}$.

Доказательство корректности основывается на следующих свойствах алгоритма:

- для непустой строки $\boldsymbol{x}$ в дереве T_{n+1}' существует невнутренний узел с менее чем двумя сыновьями;

- ребра, выходящие из внутренних узлов, ведут к разным префиксам;
- суффиксная связь для любого внутреннего узла устанавливается корректно;
- в дереве T'_{n+1} для любого $i = 1, 2, \ldots, n+1$ существует конечный узел i, соответствующий суффиксу $\boldsymbol{x}[i..n]\$$. ■

Установить временну́ю сложность алгоритма МК является более сложным делом.

Теорема 5.2.3. Алгоритм 5.2.1 вычисляет связное дерево суффиксов $T_{\boldsymbol{x}}$ для заданной строки $\boldsymbol{x}[1..n]$, определенной на упорядоченном алфавите A размером α, за время $O(n \log \alpha)$ с использованием памяти объемом $O(n\alpha)$.

Доказательство. За исключением операций быстрого и медленного сканирования, каждый шаг алгоритма 5.2.1 выполняется за константное время с использованием стандартных методов обхода дерева и его представления. Поэтому, если не учитывать операции сканирования, для выполнения алгоритма необходимо время порядка $\Theta(n)$.

Рассмотрим операцию быстрого сканирования. Обозначим через B максимальное (по всем узлам дерева) время, необходимое для выбора правильного пути из текущего узла. Тогда суммарное время, затрачиваемое на быстрое сканирование при выполнении n шагов алгоритма, составляет $O(nB)$. Но необходимо также учесть возможный рекурсивный вызов функции быстрого сканирования вследствие существования промежуточных узлов на нисходящем пути из узла $s(\mathrm{parent}(\mathrm{head}(i+1))) = s(\boldsymbol{u})$ в узел $\boldsymbol{uv} = s(\mathrm{head}(i+1))$, которые не имеют "двойников" на пути от $\boldsymbol{u}$ до $\boldsymbol{uv}$. Каждый такой узел может потребовать времени для операции быстрого сканирования порядка $O(B)$. При выполнении n шагов алгоритма может встретиться m таких узлов, поэтому на их обработку потребуется время порядка $O(m\mathrm{B})$. Но поскольку в дереве $T_{\boldsymbol{x}}$ нет пути, длиннее n, то $m < n$. Следовательно, общее время, затрачиваемое на операцию быстрого сканирования при выполнении алгоритма, составляет $O(nB)$.

Теперь рассмотрим медленное сканирование. Заметим, что поскольку $s(\mathrm{head}(i))$ является префиксом строки $\mathrm{head}(i+1)$, то

$$|\,\mathrm{head}(i+1)| - |\,\mathrm{head}(i)| + 1 \geqslant 0. \tag{5.3}$$

Фактически, последнее неравенство позволяет получить точное количество букв, просматриваемых в процессе медленного сканирования для локализации $\mathrm{head}(i+1)$, $i = 1, 2, \ldots, n$. Для этого неравенство (5.3) надо просуммировать по всем n шагам алгоритма. Получим

$$|\,\mathrm{head}(n+1)| - |\,\mathrm{head}(1)| + n = n.$$

Таким образом, общее время операции медленного сканирования составляет $O(nB)$.

Теперь оценим значение времени B. Поскольку алфавит A упорядочен, в каждом узле дерева суффиксов этот алфавит можно представить в виде упорядоченного массива или дерева поиска. Такая структура содержит не более $\Theta(\alpha)$ элементов, поиск по ней требует времени порядка $O(\log \alpha)$. Отсюда получаем оценки, приведенные в утверждении теоремы. ■

В упражнении 5.2.5 предлагается провести дальнейший анализ временно́й и пространственной сложности алгоритма МК.

5.2.3 Алгоритм Укконена

Подобно алгоритму МК, алгоритм Укконена (для краткости будем называть его алгоритмом Укк) выполняется итерационно, начиная с исходного дерева и заканчивая полным деревом суффиксов $T\boldsymbol{x}$. Но алгоритм Укк применяет другую стратегию, отличную от стратегии алгоритма МК, — в алгоритме МК в промежуточных деревьях последовательно представляются суффиксы $\boldsymbol{x}[i..n]\$$, $i = 1, 2, \ldots, n+1$, тогда как в алгоритме Укк в промежуточных деревьях последовательно представляются *префиксы* $\boldsymbol{x}[1..i]\$$, $i = 1, 2, \ldots, n$, заканчивая префиксом $\boldsymbol{x}[1..n]\$$ (т.е. заданной строкой). Таким образом, алгоритм Укк является онлайновым алгоритмом (см. раздел 4.1) — каждое построенное промежуточное дерево T_i фактически является законченным связным деревом суффиксов для строки $\boldsymbol{x}[1..i]$.

Однако в этом алгоритме метка \$ конца строки добавляется в дерево только на последней итерации построения дерева $T_{\boldsymbol{x}}$. В промежуточных деревьях T_i ($1 \leqslant i \leqslant n$) конечные узлы (т.е. те узлы, которые представляют суффиксы) не обязательно являются листьями этих деревьев. Если префикс $\boldsymbol{x}[1..i]$ имеет два суффикса $\boldsymbol{u}$ и $\boldsymbol{uv}$, тогда узел, соответствующий суффиксу $\boldsymbol{u}$, может быть внутренним узлом на пути от корня дерева до узла, представляющего суффикс $\boldsymbol{uv}$. Более того, если строка $\boldsymbol{u}$ является *только префиксом* строки $\boldsymbol{uv}$, тогда строка $\boldsymbol{u}$ вообще может не иметь узла в дереве T_i, ее представляющего.

Чтобы понять, как алгоритм Укк "рисует" деревья, рассмотрим пример построения дерева суффиксов для строки $\boldsymbol{f}_4 = abaab$, показанного на рис. 5.4 (здесь *только для пояснения* ребра помечены значениями подстрок, а не парой чисел ("i_1", "i_2"), определяющих позиции этих подстрок). Отметим, что здесь, как и в алгоритме МК, суффиксные связи определяются только для внутренних узлов (узлов ветвления).

На рис. 5.4 видно, что в деревьях T_3 и T_5 некоторые суффиксы не представлены узлами дерева, и только дерево T_6 представляет дерево суффиксов в привычной форме. Однако важно отметить, что преобразование дерева T_5 в дерево T_6 (в общем случае дерева T_n в дерево T_{n+1}) происходит по тем же правилам алгоритма, что и преобразование дерева T_i в дерево T_{i+1}, $1 \leqslant i \leqslant n-1$.

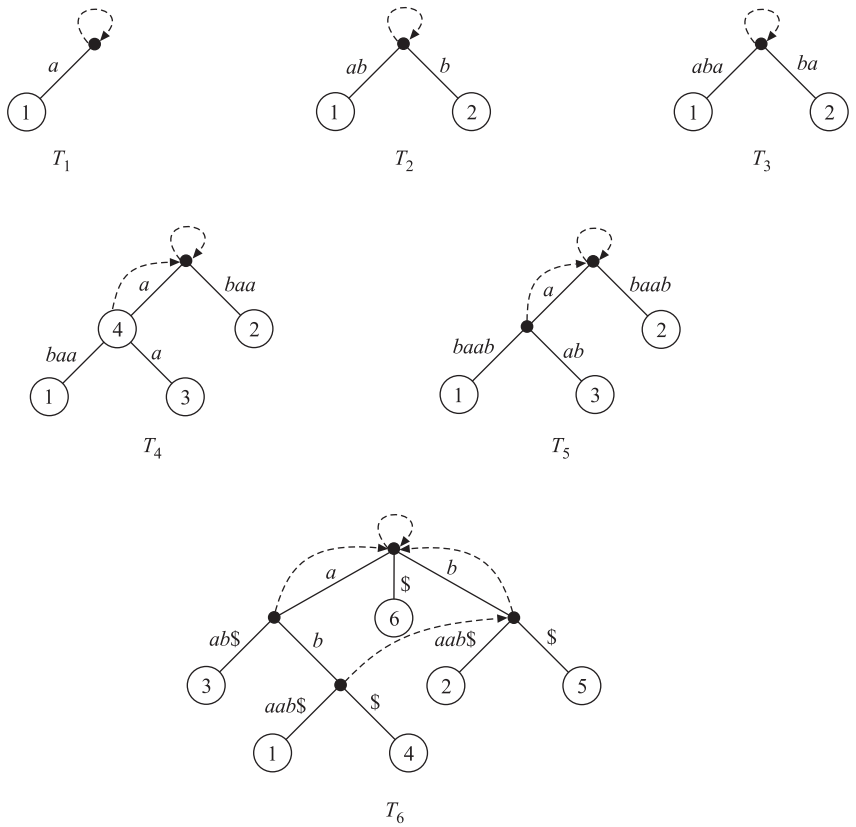


Рис. 5.4. Связное дерево суффиксов для $\boldsymbol{f}_4$, формируемое алгоритмом Укк

Поскольку алгоритм Укк является онлайновым, за один шаг он обрабатывает по одной букве строки $\boldsymbol{x}$, поэтому при описании данного алгоритма основное внимание уделим преобразованию дерева T_i в дерево T_{i+1}, $1 \leqslant i \leqslant n$, при этом предполагая, что дерево T_1 всегда имеет такой вид, как показано на рис. 5.4. Чтобы немного упростить описание алгоритма, введем символ \$ в исходную строку, так что теперь $\boldsymbol{x}[n+1] = \$$.

Если в дереве T_i определены i суффиксов $\boldsymbol{x}[j..i]$ ($j \in 1..i$), то задача преобразования дерева T_i в дерево T_{i+1} будет сведена к определению $i+1$ суффиксов $\boldsymbol{x}[j..i+1]$ в дереве T_{i+1}, при этом теперь $j \in 1..i+1$. При преобразовании дерева T_i в дерево T_{i+1} выполняются такие операции.

1. Новый j-й суффикс в дереве T_{i+1} получен из "старого" j-го суффикса дерева T_i добавлением к последнему одной буквы $\boldsymbol{x}[i+1]$. Если "старый" суффикс уже представлен в дереве T_i листом, то все, что требуется сделать в этой ситуации, — это добавить к метке ребра, ведущего к этому листу, позицию

$i+1$. Если же "старый" суффикс не представлен в дереве T_i листом, тогда необходимо "расширить" существующий в дереве T_i путь к суффиксу $\boldsymbol{x}[j..i]$ до суффикса $\boldsymbol{x}[j..i+1]$. Такое расширение может потребовать или создание нового пути (и, следовательно, создание нового листа), или создание нового узла на уже существующем пути. Последний вариант возможен в случае, когда новый суффикс является префиксом уже существующего суффикса.

2. Новый $(i+1)$-й суффикс определяется как новый сын корня дерева T_i. Если буква $\boldsymbol{x}[i+1]$ ранее не встречалась в строке $\boldsymbol{x}$, тогда $(i+1)$-й суффикс становится листом дерева T_{i+1}. В противном случае этот суффикс представлен на одном из существующих путей, исходящих от корня, а выполняемые при этом действия являются специфическим случаем операции, описанной в первом пункте, когда начальной точкой будет корень дерева.

Следующая лемма составляет основу операций, выполняемых при преобразовании дерева T_i в дерево T_{i+1}.

Лемма 5.2.4. Пусть число j определяет суффикс $\boldsymbol{x}[j..n]$ строки $\boldsymbol{x}$.

а) Если $j > 1$ и представляет лист дерева $T_{\boldsymbol{x}}$, тогда суффикс $j-1$ также будет листом.

б) Если в дереве $T_{\boldsymbol{x}}$ есть путь, начинающийся с некоторой буквы λ и проходящий через суффикс j ($j \leqslant n$), тогда существует путь, начинающийся с буквы λ и проходящий через суффикс $j+1$.

Доказательство леммы предложено дать в упражнении 5.2.6. ■

Утверждения леммы справедливы для любого дерева суффиксов $T_{\boldsymbol{x}}$, поэтому она применима к связным деревьям T_i, которые строятся в ходе выполнения алгоритма Укк. Поскольку в каждом дереве T_i ($i = 1, 2, \ldots, n+1$) суффиксу 1 всегда соответствует узел-лист, утверждение *а*) леммы говорит о том, что существует такое целое число $j_L \in 1..i$, что любой суффикс $j \in 1..j_L$ является листом, тогда как суффиксы $j \in j_L + 1..i$ таковыми не являются. Аналогично утверждение *б*) леммы говорит о том, что существует такое целое число $j_\lambda \in 2..i+1$, что любому суффиксу $j \in j_\lambda..i$ в дереве T_i соответствует путь в этом дереве, который начинается в узле с буквой λ, тогда как суффиксам $j \in 1..j_\lambda - 1$ соответствуют пути, который начинаются в узлах, не совпадающих с буквой λ. Очевидно, что для любой буквы $\lambda j_L < j_\lambda$. На рис. 5.4 видно, что в дереве T_4 $j_L = 3$ и $j_a = j_b = 4$, тогда как в дереве T_6 $j_L = 3$ и $j_a = 4$, но $j_b = 6$ (поскольку нет суффикса, который следовал бы за последней буквой b).

Возвращаясь к преобразованию дерева T_i в дерево T_{i+1}, мы можем, основываясь на приведенных выше результатах и наблюдениях, разбить множество суффиксов на три непересекающихся класса (здесь $\lambda = \boldsymbol{x}[i+1]$, т.е. буква, добавляемая на i-м шаге алгоритма):

- $1 \leqslant j \leqslant j_L$ (листья дерева T_i);
- $j_L + 1 \leqslant j \leqslant j_{\boldsymbol{x}_{[i+1]}} - 1$;
- $j_{\boldsymbol{x}_{[i+1]}} \leqslant j \leqslant i$ (суффиксы, являющиеся префиксами подстроки $\boldsymbol{x}[j..i+1]$).

Суффиксы первого и третьего классов не требуют перестройки дерева T_i, а требуют только внесения изменений в метки ребер. Рассмотрим сначала первый класс суффиксов. Напомним замечание, сделанное выше, что для суффикса j, являющегося листом, необходимо только добавить букву $\boldsymbol{x}[i+1]$ к пути, ведущему к этому префиксу. Но поскольку данный лист дерева T_i останется листом в дереве T_{i+1}, добавление всей последовательности букв $\boldsymbol{x}[i+1], \boldsymbol{x}[i+2], \ldots, \boldsymbol{x}[n+1]$ можно реализовать, просто добавив метку "∞" к ребру, ведущему к узлу j. Таким образом, если узел-лист j является сыном некоего узла $\boldsymbol{u} = \boldsymbol{x}[1..h]$, то ребро от узла $\boldsymbol{u}$ до узла j помечается как $(h+1, \infty)$, что указывает на суффикс $\boldsymbol{x}[h+1..n+1]$, присоединенный к этому ребру.

Для третьего класса суффиксов также нет необходимости изменять дерево T_i, поскольку уже существует путь от суффикса j, который начинается с буквы $\lambda = \boldsymbol{x}[i+1]$, — для преобразования дерева T_i в дерево T_{i+1} достаточно суффикс j переместить по дереву на одну позицию вниз для обозначения строки $\boldsymbol{x}[j..i+1]$, которая уже должна быть представлена в дереве T_i. В последнем случае надо эффективно локализовать суффикс j, для чего можно использовать метод быстрого сканирования, описанный в разделе 5.2.1.

Таким образом, в процессе преобразования дерева T_i в дерево T_{i+1} следует основное внимание уделить суффиксам "второго класса" $j \in j_L + 1..j_{\boldsymbol{x}_{[i+1]}} - 1$, которые не являются листьями в дереве T_i и при этом не встречались ранее в строке $\boldsymbol{x}[1..i]$ в качестве префиксов подстроки $\boldsymbol{x}[j..i+1]$. Рассмотрим пример строки, в которой присутствуют суффиксы всех трех классов.

	1	2	3	4	5	6	7	8	9	10	
$\boldsymbol{x} =$	a	a	b	b	a	c	b	b	a	a	$\ldots$

Для $i = 9$ имеем $j_L = 6$, поскольку

- буква c в строке $\boldsymbol{x}$ впервые встречается в позиции 6;
- подстрока $\boldsymbol{x}[7..9]$ встречалась ранее в строках $\boldsymbol{x}[1..i]$ как $\boldsymbol{x}[3..5]$.

Также имеем $j_{\boldsymbol{x}_{[i+1]}} = j_a = 9$, поскольку подстрока $\boldsymbol{x}[9..10] = aa$ встречалась ранее, а подстрока $\boldsymbol{x}[8..10] = baa$ не встречалась. Для $i = 9$ имеем следующие три класса суффиксов $j : \{1..6,\ 7..8,\ 9..9\}$.

При преобразовании дерева T_i в дерево T_{i+1} алгоритм Укк начинает с проверки первого суффикса $j_L + 1$, который не является листом в дереве T_i. Если суффикс $j_L + 1$ не встречается в строке $\boldsymbol{x}[1..i]$ после буквы $\boldsymbol{x}[i+1]$, тогда в дереве T_{i+1} создается лист $j_L + 1$, значение j_L увеличивается на единицу и далее проверяется

следующий суффикс. Когда или $j_L > i$ или суффикс $j_L + 1$ следует за буквой $\boldsymbol{x}[i+1]$, преобразование дерева T_i заканчивается.

Поиск суффикса $j_L + 1$ начинается от префиксного узла $\boldsymbol{u}$. Первоначально, в дереве T_1, узел $\boldsymbol{u}$ является корнем дерева, в последующих деревьях узел $\boldsymbol{u}$ определяется как ближайший предок суффикса $j_L + 1$, для которого существует суффиксная связь.

Для манипуляций с узлом $\boldsymbol{u}$ используем измененный вариант быстрого сканирования, который назовем ***интеллектуальным сканированием***, а соответствующую функцию, которая будет реализовать такое сканирование, назовем *smartscan*. Подобно быстрому сканированию, функция $smartscan(\boldsymbol{u}, \boldsymbol{v})$ определяет в дереве T_i строку $\boldsymbol{w} = \boldsymbol{uv}$, а также заменяет префиксный узел $\boldsymbol{u}$ ближайшим предком строки $\boldsymbol{w}$ (если узел $\boldsymbol{w}$ уже существует, то происходит формальная замена $\boldsymbol{w}$ на $\boldsymbol{w}$). Таким образом, при интеллектуальном сканировании вычисляется пара строк:

$$(\boldsymbol{u}, \boldsymbol{w}) \leftarrow smartscan(\boldsymbol{u}, \boldsymbol{v}).$$

Как и в алгоритме МК, здесь используется тот факт, что если $\boldsymbol{u}$ является префиксным узлом для суффикса j_L, тогда суффиксная связь $s(\boldsymbol{u})$ указывает на префиксный узел j_L+1. Каждое дерево T_i создается как связное дерево суффиксов с суффиксными связями, вычисленными для каждого внутреннего узла. Эти связи облегчают начальный поиск для суффикса $j_L + 1$, и любой последующий поиск для того же суффикса использует в интеллектуальном сканировании измененный префиксный узел.

Приведем в формальной форме алгоритм Укк.

Алгоритм 5.2.2 (Алгоритм Укконена)

▷ *Онлайновый алгоритм вычисления связного дерева суффиксов* $T_{\boldsymbol{x}}$
построение дерева T_1 ▷ *с суффиксной связью* $s(\text{корень}) = \text{корень}$
$j_L \leftarrow 1$ ▷ *в соответствии с леммой 5.2.4 узлы-листья*
▷ *добавляются в порядке* $1, 2, \ldots$
$\boldsymbol{u} \leftarrow \varepsilon$ ▷ *начальным префиксным узлом является корень*
for $i \leftarrow 1$ **to** n **do** ▷ *преобразование* T_i *в* T_{i+1}
▷ *в цикле repeat формируется последний узел ветвления,*
▷ *его суффиксная связь изменяется в зависимости от*
▷ *значения переменной* $exit$
$\boldsymbol{w}_{\text{пред}} \leftarrow \varepsilon$
$exit \leftarrow$ `FALSE`
repeat ▷ *создание, если необходимо, новых листов*
▷ *локализация в* T_i *суффикса* $\boldsymbol{w} = \boldsymbol{uv} = \boldsymbol{x}[j_L + 1..i]$
▷ *и, если необходимо, замена префиксного узла* $\boldsymbol{u}$
$(\boldsymbol{u}, \boldsymbol{w}) \leftarrow smartscan(\boldsymbol{u}, \boldsymbol{v})$

if не существует пути от $\boldsymbol{w}$, помеченного $\boldsymbol{x}[i+1]$ **then**
▷ *формируется новый лист (лемма 5.2.4)*
$j_L \leftarrow j_L + 1$; **if** $j_L > i$ **then** $exit \leftarrow$ TRUE
if узел $\boldsymbol{w}$ в T_i не существует **then**
создание узла $\boldsymbol{w}$
помечается ребро, ведущее к узлу $\boldsymbol{w}$
помечается ребро, выходящее из узла $\boldsymbol{w}$
добавляется лист j_L, помечается **?** ребро от $\boldsymbol{w}$ до j_L
▷ *вычисляется связь для предыдущего узла*
if $\boldsymbol{w}_{\text{пред}} \neq \varepsilon$ **then** $s(\boldsymbol{w}_{\text{пред}}) \leftarrow \boldsymbol{w}$
▷ *для следующего выполнения smartscan*
▷ *определяется префиксный узел*
$\boldsymbol{u} \leftarrow s(\boldsymbol{u})$
else
▷ *суффикс $j_L + 1$ в T_i расширяется до суффикса $j_L + 1$*
▷ *в T_{i+1}, аналогичные действия выполняются*
▷ *для всех последующих суффиксов (лемма 5.2.4)*
▷ *вычисляется суффиксная связь для последнего узла*
if $\boldsymbol{w}_{\text{пред}} \neq \varepsilon$ **then**
$s(\boldsymbol{w}_{\text{пред}}) \leftarrow \boldsymbol{w}$ ▷ *для $\boldsymbol{w}$ существует узел*
$exit \leftarrow$ TRUE
▷ *выход из цикла, если $j_L > i$ или нет возможности*
▷ *добавить новые листья*
until $exit$

Осталось прояснить еще одно "темное место" данного алгоритма. Если в дереве T_i создана последовательность листьев, суффиксную связь $s(\boldsymbol{w})$ родителя последнего листа $j_L + h$ в этой последовательности можно установить только тогда, когда следующий суффикс $j_L + h + 1$ *не является* листом. В этом случае $s(\boldsymbol{w})$ будет указывать на строку $\boldsymbol{w}' = \boldsymbol{x}[j_L + h + 1..i]$, которой соответствует узел в дереве T_i. Это утверждение основывается на следующих рассуждениях: строка $\boldsymbol{w} = \boldsymbol{x}[j_L + h..i]$ порождает путь, помеченный, например, меткой $\boldsymbol{\pi}$, который не начинается с буквы $\boldsymbol{x}[i+1]$. Однако узел $\boldsymbol{w}'$ имеет не менее двух нисходящих путей: путь $\boldsymbol{\pi}$ и еще один путь, который начинается с буквы $\boldsymbol{x}[i+1]$. Эти рассуждения доказывают следующую теорему.

Теорема 5.2.5. Алгоритм 5.2.2 корректно вычисляет связные деревья суффиксов T_i, $i = 1, 2, \ldots, n+1$. ■

Доказательство оценки временно́й сложности алгоритма 5.2.2 повторяет схему доказательства теоремы 5.2.3.

Теорема 5.2.6. Алгоритм 5.2.2 вычисляет связное дерево суффиксов $T_{\boldsymbol{x}}$ для заданной строки $\boldsymbol{x}[1..n]$, определенной на упорядоченном алфавите A размером α, за время $O(n \log \alpha)$ с использованием памяти объемом $O(n\alpha)$.

Доказательство. Сначала заметим, что любая операция в алгоритме 5.2.2 выполняется не более $2n - 1$ раз: один раз для каждого значения i, плюс не более $n - 1$ раз при создании листьев в цикле **repeat**.

За исключением, возможно, операций интеллектуального сканирования и определения пути от узла $\boldsymbol{w}$, каждый шаг алгоритма можно выполнить за константное время, используя для этого стандартные методы обхода дерева и его представления. Путь от узла $\boldsymbol{w}$ определяется за время $O(B)$, где, как и в доказательстве теоремы 5.2.3, B обозначает максимальное (по всем узлам дерева) время, необходимое для выбора правильного пути из текущего узла. Таким образом, если не учитывать операцию сканирования, то алгоритм выполняется за время $O(nB)$.

Теперь рассмотрим операцию интеллектуального сканирования. Для каждого значения j_L функция *smartscan* вызывается несколько раз, но после ее первого вызова выполняется только повторное вычисление префиксного узла $\boldsymbol{u}$, для чего необходимо фиксированное время. Первоначально строка $\boldsymbol{u}$ полагается равной пустой строке, затем в процессе сканирования она изменяет свое значение, при этом сканирование не уменьшает ее длину, тогда как вычисление суффиксной функции s может уменьшить ее длину, но не более чем на единицу. Поэтому после всех сканирований длина строки $\boldsymbol{u}$ не может уменьшиться более чем на n. Следовательно, общее количество промежуточных узлов, просмотренных в процессе сканирования, также не превышает n. Таким образом, функция *smartscan* обрабатывает не более $2n$ узлов, при этом на каждый из них тратит не более $O(B)$ времени. Теперь осталось привлечь оценку времени B из доказательства теоремы 5.2.3 и получить требуемое утверждение данной теоремы. ■

В упражнении 5.2.9 предложено провести детальное сравнение алгоритмов МК и Укк. На практике алгоритм МК выполняется немного быстрее, чем алгоритм Укк [102]. С другой стороны, онлайновая природа алгоритма Укк позволяет динамически строить дерево суффиксов $T_{\boldsymbol{x}_1\boldsymbol{x}_2\ldots\boldsymbol{x}_k}$ по мере поступления на обработку строк $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$.

5.2.4 Алгоритм Фарача

Алгоритм Фарача (будем его для краткости обозначать как алгоритм Ф) представляет собой значительное открытие: это первый алгоритм построения дерева суффиксов, которое действительно выполняется за время $\Theta(n)$, при этом даже не требуется выполнения условия конечности алфавита. Такая эффективность достигается за счет того, что строковые последовательности определяются на индексированном алфавите (см. раздел 4.1) или, что эквивалентно, на целочисленном

алфавите $A = \{1, 2, \ldots, \alpha\}$, при этом накладывается дополнительное условие, что $\alpha \in O(n)$. Такие алфавиты часто встречаются на практике.

Основу алгоритма Ф составляет алгоритм, описанный в работе [85], он кардинально отличается от других алгоритмов построения деревьев суффиксов. Мы опишем алгоритм Ф в виде пяти выполняемых шагов, что несколько отличается от оригинального описания метода в статье [86]. Чтобы сделать описание алгоритма более понятным, используем в качестве примера строку $\boldsymbol{g} = 121112212221$, определенную на алфавите $A = \{1, 2\}$ (в этом примере $n = 12$).

Шаг I. Построение нечетного дерева $T_{\boldsymbol{x}}^{\text{нч}}$.

Нечетное дерево $T_{\boldsymbol{x}}^{\text{нч}}$ является деревом суффиксов для строки $\boldsymbol{x}$, узлы-листья которого ограничены нечетными позициями $1, 3, 5, \ldots$ строки $\boldsymbol{x}\$$. Дерево $T_{\boldsymbol{g}}^{\text{нч}}$ показано на рис. 5.5.

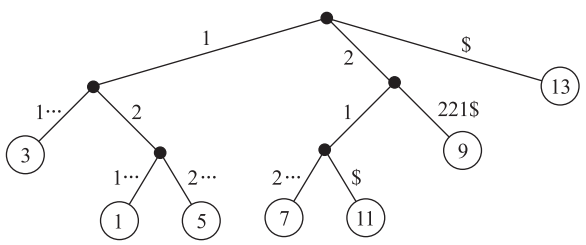


Рис. 5.5. Нечетное дерево для строки $\boldsymbol{g}$

Основой эффективного построения дерева $T_{\boldsymbol{x}}^{\text{нч}}$ за время $\Theta(n)$ служит поразрядная сортировка всех упорядоченных пар $\pi_i = (\boldsymbol{x}[2i-1], \boldsymbol{x}[2i])$, $i = 1, 2, \ldots, \lceil n/2 \rceil$, где для нечетного n последней парой будет $(\boldsymbol{x}[n], \$)$ (символ \$ принимается как $(\alpha+1)$-я буква алфавита). Далее формируется новая строка $\boldsymbol{x}'$ из ***рангов*** $\rho(\pi_i)$ пар π_i. В нашем примере пары $(1,\ 2)$, $(1,\ 1)$, $(1,\ 2)$, $(2,\ 1)$, $(2,\ 2)$, $(2,\ 1)$ имеют ранги соответственно 2, 1, 2, 3, 4, 3. Поэтому $\boldsymbol{g}' = 212343$.

Для строки $\boldsymbol{x}'$ строится дерево суффиксов $T_{\boldsymbol{x}'}$. Между этим деревом и деревом $T_{\boldsymbol{x}}^{\text{нч}}$ существуют такие соответствия.

- Листьям i дерева $T_{\boldsymbol{x}'}$ соответствуют листья $2i-1$ в дереве $T_{\boldsymbol{x}}^{\text{нч}}$.
- Внутренним узлам дерева $T_{\boldsymbol{x}'}$, представляющим подстроки длиной k, соответствуют внутренние узлы дерева $T_{\boldsymbol{x}}^{\text{нч}}$, представляющие подстроки длиной $2k$.

Дерево $T_{\boldsymbol{g}'}$ показано на рис. 5.6.

На основе приведенных соответствий нетрудно показать, что дерево $T_{\boldsymbol{x}'}$ можно преобразовать в дерево $T_{\boldsymbol{x}}^{\text{нч}}$ за время $\Theta(n)$. Если обозначить через

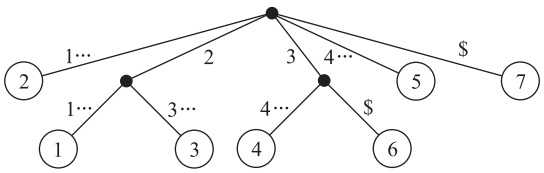


Рис. 5.6. Дерево суффиксов для строки $\boldsymbol{g}'$

$t(n)$ максимальное (по всем возможным строкам $\boldsymbol{x}$ длиной n) время вычисления дерева $T_{\boldsymbol{x}}$ данным алгоритмом (выполнив все его пять шагов), тогда дерево $T_{\boldsymbol{x}}^{\text{нч}}$ можно вычислить за время, не превышающее

$$t(n/2) + \Theta(n), \tag{5.4}$$

путем предварительного вычисления дерева $T_{\boldsymbol{x}'}$ и последующего его преобразования в дерево $T_{\boldsymbol{x}}^{\text{нч}}$. Если, кроме того, мы сможем показать, что время, требуемое для выполнения других четырех шагов алгоритма Ф, имеет порядок $O(n)$, тогда

$$t(n) - t(n/2) \in \Theta(n) \tag{5.5}$$

и, следовательно, $t(n) \in \Theta(n)$.

Шаг II. Построение четного дерева $T_{\boldsymbol{x}}^{\text{чет}}$ на основе дерева $T_{\boldsymbol{x}}^{\text{нч}}$.

Заметим следующее: если в строке $\boldsymbol{x}$ известен лексикографический порядок суффиксов, которые начинаются с нечетных позиций $2i+1$, $i = 1, 2, \ldots, \lceil n/2 \rceil - 1$, тогда можно определить лексикографический порядок суффиксов, которые начинаются с четных позиций $2i$, путем поразрядной сортировки первых элементов пар $(\boldsymbol{x}[2i], 2i+1)$, которую можно выполнить за $\Theta(n)$ время.

В нашем примере мы получим упорядочение 3, 1, 5, 7, 11, 9, 13 нечетных позиций в строке $\boldsymbol{g}$ путем обхода в прямом порядке дерева $T_{\boldsymbol{x}}^{\text{нч}}$. Исключая позицию 1 и добавляя к этим числам значения $\boldsymbol{g}[2i]$, $i = 1, 2, \ldots, 6$, получим набор пар (2, 3), (1, 5), (2, 7), (2, 11), (1, 9), (1, 13), который надо отсортировать по первым элементам. После сортировки имеем (1, 5), (1, 9), (1, 13), (2, 3), (2, 7), (2, 11). Вычитая из вторых элементов пар первые, получим набор 4, 8, 12, 2, 6, 10, который является лексикографическим порядком четных суффиксов $\boldsymbol{g}[2i..n+1]$ и, следовательно, порядком листьев в дереве $T_{\boldsymbol{x}}^{\text{чет}}$.

Как будет пояснено далее, теперь можно за время $\Theta(n)$ вычислить длины наименьших общих префиксов смежных листьев $2i$ и $2j$ в дереве $T_{\boldsymbol{x}}^{\text{чет}}$:

$$\text{lcp}(2i, 2j) = |\,\text{LCP}(2i, 2j)|.$$

Теперь построить дерево $T_{\boldsymbol{x}}^{\text{чет}}$ не представляет труда. Подобное дерево для строки $\boldsymbol{g}$ показано на рис. 5.7.

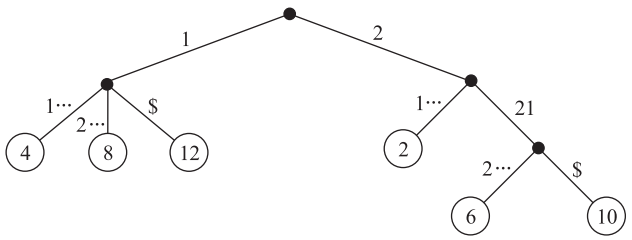


Рис. 5.7. Четное дерево для строки $\boldsymbol{g}$

Шаг III. Слияние деревьев $T_{\boldsymbol{x}}^{\text{нч}}$ и $T_{\boldsymbol{x}}^{\text{чет}}$.

Далее необходимо найти эффективный способ слияния нечетного и четного деревьев в одно дерево $T_{\boldsymbol{x}}$. Слияние будем производить начиная с корня. Предположим, что для каждого узла деревьев $T_{\boldsymbol{x}}^{\text{нч}}$ и $T_{\boldsymbol{x}}^{\text{чет}}$ выходящие из них ребра занесены в специальные списки, где они упорядочены в возрастающем лексикографическом порядке подстрок, которые представляют эти ребра. Алгоритм слияния деревьев просматривает только первые буквы подстрок, представленных ребрами деревьев $T_{\boldsymbol{x}}^{\text{нч}}$ и $T_{\boldsymbol{x}}^{\text{чет}}$, пусть это будут буквы $\lambda^{\text{нч}}$ и $\lambda^{\text{чет}}$. Тогда,

- если $\lambda^{\text{нч}} \neq \lambda^{\text{чет}}$, определяется поддерево, соответствующее меньшей из этих букв, и без изменений присоединяется к узлу-родителю;
- если $\lambda^{\text{нч}} = \lambda^{\text{чет}}$ и длины подстрок, представленных соответствующими ребрами, равны, в дерево слияния к текущему узлу добавляются два сына: один — из четного дерева, другой — из нечетного;
- если $\lambda^{\text{нч}} = \lambda^{\text{чет}}$ и длины подстрок, представленных соответствующими ребрами, различны, в дерево слияния к текущему узлу добавляются два узла, находящиеся на одном нисходящем пути, при этом ближайший узел будет соответствовать более короткой подстроке.

Если начать эту процедуру для корней нечетного и четного деревьев, далее она рекурсивно выполняется для корней всех поддеревьев, которые, возможно, уже содержат узлы из нечетного и четного деревьев, поскольку ранее мог быть реализован случай $\lambda^{\text{нч}} = \lambda^{\text{чет}}$. Так как время манипулирования с любым ребром этих деревьев фиксированно, то общее время слияния деревьев составит $\Theta(n)$.

В результате описанных действий получится дерево $M_{\boldsymbol{x}}$, в котором будут присутствовать поддеревья, которые прошли процедуру слияния и которые ее избежали (т.е. были перенесены в дерево $M_{\boldsymbol{x}}$ без изменений). Теперь

встает задача преобразовать полученное дерево $M_{\boldsymbol{x}}$ в конечное дерево $T_{\boldsymbol{x}}$. На рис. 5.8 показано полученное дерево слияния для строки $\boldsymbol{g}$, где толстыми линиями выделены ребра, которые "прошли обработку" в процессе слияния. Отметим, что в результате слияния может получиться так, что некоторые суффиксы (как суффиксы 4 и 8 на рис. 5.8) будут представлены внутренними узлами дерева $M_{\boldsymbol{x}}$.

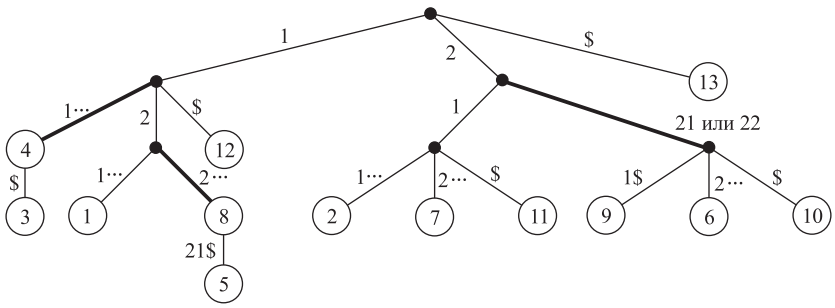


Рис. 5.8. Дерево слияния M_g для строки $\boldsymbol{g}$

Шаг IV. Построение LCP-дерева.

В качестве прелюдии к преобразованию дерева $M_{\boldsymbol{x}}$ представим некоторые факты и простые утверждения, которые помогут нам в дальнейшем в построении дерева $T_{\boldsymbol{x}}$. (Далее будем называть "слитой частью дерева $M_{\boldsymbol{x}}$" ту часть этого дерева, которая прошла обработку в процессе слияния нечетного и четного деревьев, и "неслитой частью" — необработанную часть этого дерева.)

O1. В процессе построения дерева $M_{\boldsymbol{x}}$ каждый его узел формируется на основе как дерева $T_{\boldsymbol{x}}^{\text{нч}}$, так и дерева $T_{\boldsymbol{x}}^{\text{чет}}$.

O2. Узел $\boldsymbol{u}$ принадлежит слитой части дерева $M_{\boldsymbol{x}}$ только тогда, когда он имеет потомков $2i$ и $2j-1$ (возможно, один из них совпадает с самим узлом $\boldsymbol{u}$), которые являются листьями в деревьях $T_{\boldsymbol{x}}^{\text{чет}}$ и $T_{\boldsymbol{x}}^{\text{нч}}$ соответственно.

O3. Пусть в дереве $M_{\boldsymbol{x}}\boldsymbol{u} = \text{LCA}(2i, 2j-1)$, а в дереве $T_{\boldsymbol{x}}$ есть узел $\boldsymbol{u}' = \\ = \text{LCA}(2i, 2j-1)$, который в дереве $M_{\boldsymbol{x}}$ принадлежит поддереву с корнем $\boldsymbol{u}$.[5] Тогда $\boldsymbol{u}'$ является префиксом $\boldsymbol{u}$. Следовательно, любая пара $(2i, 2j-1)$ с наименьшим общим предком $\boldsymbol{u}$ в дереве $M_{\boldsymbol{x}}$ имеет наименьшим общим предком подстроку $\boldsymbol{u}'$ в дереве $T_{\boldsymbol{x}}$.

[5]Напомним, что LCA обозначает "наименьший общий предок" (см. раздел 5.2.1). — *Примеч. ред.*

O4. Если узел $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$ в дереве $M_{\boldsymbol{x}}$ не является корнем этого дерева, тогда $\mathrm{LCA}(2i, 2j-1) \neq \varepsilon$ и поэтому $\boldsymbol{u}' = \mathrm{LCA}(2i, 2j-1)$ не является корнем дерева $T_{\boldsymbol{x}}$.

O5. Пусть (i, i') — пара листьев дерева $M_{\boldsymbol{x}}$, которые оба четные или нечетные. Тогда $\mathrm{LCA}(i, i') = \mathrm{LCP}(i, i')$, т.е. в этом случае выполняется равенство (5.1).

O6. С учетом фактов O2, O3 и O5 равенство $M_{\boldsymbol{x}} = T_{\boldsymbol{x}}$ будет выполняться только тогда, когда для любого внутреннего узла $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$ дерева $M_{\boldsymbol{x}}$ выполняется условие $\boldsymbol{u} = \mathrm{LCP}(2i, 2j-1)$.

Утверждение O6 говорит о том, что для корректного преобразования дерева $M_{\boldsymbol{x}}$ в дерево $T_{\boldsymbol{x}}$ необходимо проверить равенство (5.1) для каждой пары $(2i, 2j-1)$ четных-нечетных листьев дерева $M_{\boldsymbol{x}}$. Другими словами, надо вычислить $\mathrm{LCP}(2i, 2j-1)$ или, что то же самое, найти префикс строки $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$ в дереве $M_{\boldsymbol{x}}$ длиной $\mathrm{lcp}(2i, 2j-1)$, который определит $\boldsymbol{u}' = \mathrm{LCA}(2i, 2j-1)$ в дереве $T_{\boldsymbol{x}}$. Заметим, что

$$\mathrm{lcp}(2i, 2j-1) = \mathrm{lcp}(2i+1, 2j) + 1, \quad \text{если } \boldsymbol{x}[2i] = \boldsymbol{x}[2j-1], \tag{5.6}$$

и $\mathrm{lcp}(2i, 2j-1) = 0$ в противном случае. Но, согласно утверждению O4, $\mathrm{lcp}(2i, 2j-1) = 0$ только в том случае, когда $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$ является корнем дерева $M_{\boldsymbol{x}}$. Таким образом, для *всех* четных-нечетных пар $(2i, 2j-1)$ в поддереве с корнем $\boldsymbol{u}$ (если $\boldsymbol{u}$ не является корнем дерева $M_{\boldsymbol{x}}$) $\mathrm{lcp}(2i, 2j-1)$ можно вычислить на основе значения $\mathrm{lcp}(2i+1, 2j)$. Тогда, если в дереве $M_{\boldsymbol{x}}$ найдено $\boldsymbol{v} = \mathrm{LCA}(2i+1, 2j)$, можно определить дерево, построенное на внутренних узлах дерева $M_{\boldsymbol{x}}$, такое, что в этом дереве $\boldsymbol{v}$ будет родителем $\boldsymbol{u}$. Если $\boldsymbol{v}$ не является корнем дерева $M_{\boldsymbol{x}}$, то можно найти его родителя в дереве, подобном тому, в котором искался родитель $\boldsymbol{u}$. Этот процесс можно продолжать до тех пор, пока родителем не окажется корень дерева $M_{\boldsymbol{x}}$. Назовем дерево, которое определяется в процессе поиска родителей, ***LCP-деревом***. Заметим, что $\mathrm{lcp}(2i, 2j-1)$ для всех четных-нечетных пар $(2i, 2j-1)$ равняется глубине вершины $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$ в LCP-дереве.

Ниже будет показано, что LCP-дерево можно вычислить за время $\Theta(n)$. LCP-дерево для строки $\boldsymbol{g} = 121112212221\$$ приведено на рис. 5.9. Здесь, например, $\mathrm{lcp}(5, 8) = 3$, поскольку узел 8 в этом дереве находится на глубине 3. Подобным образом вычисляются

$$\mathrm{lcp}(3, 4) = \mathrm{lcp}(1, 8) = \mathrm{lcp}(2, 7) = \mathrm{lcp}(2, 11) = \mathrm{lcp}(6, 9) = \mathrm{lcp}(9, 10) = 2.$$

Шаг V. Построение дерева $T_{\boldsymbol{x}}$ на основе дерева $M_{\boldsymbol{x}}$ и LCP-дерева.

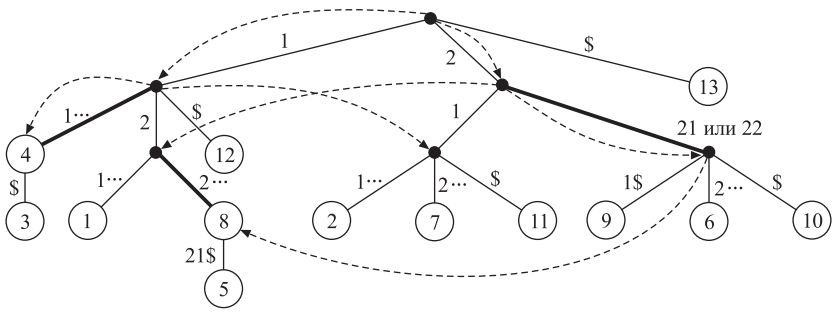


Рис. 5.9. Дерево M_g и LCP-дерево для строки $\boldsymbol{g}$

Чтобы преобразовать дерево $M_{\boldsymbol{x}}$ в дерево $T_{\boldsymbol{x}}$, сначала надо для каждого узла $\boldsymbol{u}$ в LCP-дереве проверить, будет ли его глубина $d(\boldsymbol{u})$ удовлетворять условию

$$d(\boldsymbol{u}) = |\boldsymbol{u}|, \tag{5.7}$$

где $|\boldsymbol{u}|$ обозначает длину подстроки, определяемой путем в дереве $M_{\boldsymbol{x}}$ от корня до узла $\boldsymbol{u}$. Глубину каждого узла можно определить во время обхода LCP-дерева, а $|\boldsymbol{u}|$ для каждого узла можно вычислить в процессе формирования дерева $M_{\boldsymbol{x}}$ на основе нечетного и четного деревьев. Поскольку дерево $M_{\boldsymbol{x}}$ формируется путем слияния этих деревьев, то $d(\boldsymbol{u}) \leqslant |\boldsymbol{u}|$.

Предположим, что $\boldsymbol{v}$ — вершина, такая, что для всех ее предков $\boldsymbol{u}$ в LCP-дереве выполняется равенство (5.7). Если, кроме того, $d(\boldsymbol{v}) = |\boldsymbol{v}|$, то проверка закончена. Если же $d(\boldsymbol{v}) < |\boldsymbol{v}|$, то выполняются следующие действия.

P1. Обозначим через $p(\boldsymbol{v})$ родителя узла $\boldsymbol{v}$ в дереве $M_{\boldsymbol{x}}$. Новый узел $\boldsymbol{v}'$ вставляется в дерево $M_{\boldsymbol{x}}$ как сын узла $p(\boldsymbol{v})$, если $|\boldsymbol{v}'| = d(\boldsymbol{v})$. Фактически, $\boldsymbol{v}' = \mathrm{LCP}(2i, 2j-1)$ для каждой четной-нечетной пары $(2i, 2j-1)$, принадлежащей поддереву дерева $M_{\boldsymbol{x}}$ с корнем в узле $\boldsymbol{v}$. Поэтому в дереве $T_{\boldsymbol{x}}\boldsymbol{v}'$ становится родителем поддеревьев из четного и нечетного деревьев с корнем $\boldsymbol{v}$, которые сливаются на шаге III алгоритма, а узел $\boldsymbol{v}$ при этом удаляется.

P2. Узлы, подобные узлу $\boldsymbol{v}$, представленному выше, обязательно появляются при слиянии четных и нечетных поддеревьев. Все четные и нечетные поддеревья, имеющие корнем узел $\boldsymbol{v}$ и слитые в дереве $M_{\boldsymbol{x}}$, необходимо снова разделить. Такое разделение начинается с узла $\boldsymbol{v}'$. Целесообразно применить итерационный процесс разделения, который несложно организовать, если в каждом узле дерева $M_{\boldsymbol{x}}$ хранить указатели на четные и нечетные поддеревья, которые слиты в этом узле. Тогда "правильное" поддерево с корнем в $p(\boldsymbol{v})$ можно сформировать так, как показано на

рис. 5.10. Здесь $\ell^{\text{чет}}$ (соответственно $\ell^{\text{нч}}$) — длина исходящего ребра[6], ведущего к четному (нечетному) поддереву $T_{\boldsymbol{x}}^{\text{чет}}$ (соответственно, $T_{\boldsymbol{x}}^{\text{нч}}$), которое слито в дереве $M_{\boldsymbol{x}}$ с первоначальным корнем $\boldsymbol{v}$.

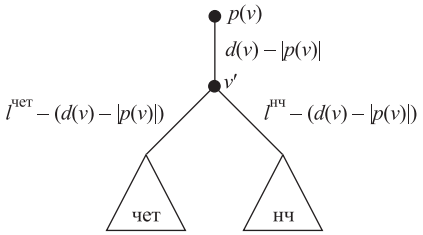


Рис. 5.10. Формирование поддерева с корнем $p(\boldsymbol{v})$

Чтобы лучше понять, как выполняются действия P1 и P2, рассмотрим для строки $\boldsymbol{g} = 121112212221\$$ преобразование дерева слияния M_g, показанного на рис. 5.9, в дерево T_g, приведенное на рис. 5.11.

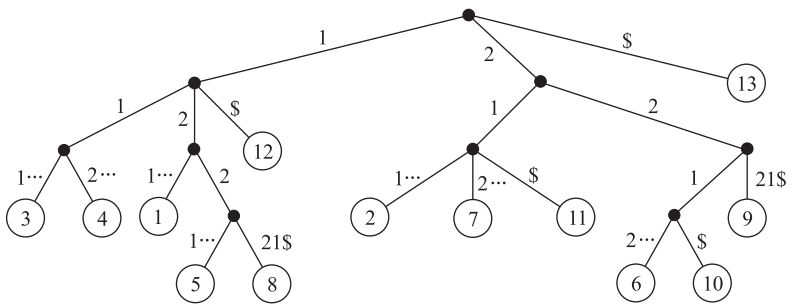


Рис. 5.11. Законченное дерево суффиксов T_g

Обозначим через $\boldsymbol{v}$ родителя узлов 9, 6 и 10 в дереве M_g; как и выше, $p(\boldsymbol{v})$ обозначает родителя узла $\boldsymbol{v}$. Поскольку в данном случае

$$2 = d(\boldsymbol{v}) < |\boldsymbol{v}| = 3,$$

вводится новый узел $\boldsymbol{v}'$ как сын узла $p(\boldsymbol{v})$ с ребром длиной

$$d(\boldsymbol{v}) - |p(\boldsymbol{v})| = 1. \tag{5.8}$$

Последнее значение определяет длину префикса, который необходимо удалить из меток исходящих ребер сливаемых поддеревьев $T_{\boldsymbol{x}}^{\text{чет}}$ и $T_{\boldsymbol{x}}^{\text{нч}}$. Возвра-

[6]Здесь и далее длиной ребра называется длина метки этого ребра. — *Примеч. ред.*

щаясь к рис. 5.7 и 5.5, находим эти поддеревья (они отдельно показаны на рис. 5.12).

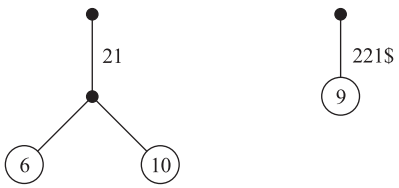


Рис. 5.12. Фрагменты четного и нечетного поддеревьев

В данном случае $\ell^{\text{чет}} = 2$, а $\ell^{\text{нч}} = 4$. Как видно на рис. 5.11, из меток исходящих ребер, согласно равенству (5.8), удаляется первая буква (буква 2), т.е. $\ell^{\text{чет}}$ и $\ell^{\text{нч}}$ уменьшаются на единицу. Поддеревья должны вставляться в лексикографическом порядке первых букв меток (подстрок) ребер нисходящего пути от узла $\boldsymbol{v}'$. Поэтому на рис. 5.10 четное дерево расположено слева, а нечетное — справа.

Может случиться, что $\ell^{\text{чет}} = 1$ или $\ell^{\text{нч}} = 1$ (одновременно эти равенства выполняться не могут). В этом случае соответствующие исходящие ребра удаляются и корнем соответствующего поддерева становится узел $\boldsymbol{v}'$.

Мы закончили поверхностный обзор алгоритма Ф — внимательный читатель обязательно найдет в этом обзоре много "темных мест". Поэтому некоторые детали мы рассмотрим отдельно в следующем подразделе, но они будут, в основном, относиться к реализации алгоритма.

Реализация

Рассмотрим некоторые подробности выполнения алгоритма Ф, опущенные при общем его описании.

Шаг I. Построение нечетного дерева $T_{\boldsymbol{x}}^{\text{нч}}$.

В описании этого шага остались неясными некоторые детали процесса сортировки, а также преобразования дерева $T_{\boldsymbol{x}'}$ в дерево $T_{\boldsymbol{x}}^{\text{нч}}$.

Чтобы выполнить сортировку целочисленных пар $\pi_i = (\boldsymbol{x}[2i-1], \boldsymbol{x}[2i])$, необходимо дважды использовать поразрядную сортировку. Сначала происходит сортировка по вторым буквам $\boldsymbol{x}[2i]$, а затем по первым буквам $\boldsymbol{x}[2i-1]$. Каждая из таких сортировок требует времени порядка $\Theta(\alpha+n/2)$.

Как показано в упражнении 5.2.13, упорядочение рангов $\rho_i = \rho(\pi_i)$ этих отсортированных пар выполняется по дереву суффиксов строки $\boldsymbol{x}' = \rho_1\rho_2\ldots\rho_{\lceil n/2\rceil}$ и соответствует упорядочению листьев $2i-1$ дерева $T_{\boldsymbol{x}}^{\text{нч}}$. Аналогично каждый внутренний узел $\boldsymbol{u}$ дерева $T_{\boldsymbol{x}'}$, определяющий подстроку $\boldsymbol{x}'$

длиной $|\boldsymbol{u}|$, соответствует узлу дерева $T_{\boldsymbol{x}}^{\text{нч}}$, определяющему подстроку длиной $|2\boldsymbol{u}|$, — отдельные буквы подстроки $\boldsymbol{x}'$ (эти буквы являются рангами) заменяются на соответствующие подстроки $\boldsymbol{x}[2i-1..2i]$ строки $\boldsymbol{x}$. Реализация такой замены на примере дерева $T_{\boldsymbol{g}'}$ (см. рис. 5.6) приводит к дереву $T_{\boldsymbol{g}'\text{зм}}$, показанному на рис. 5.13.

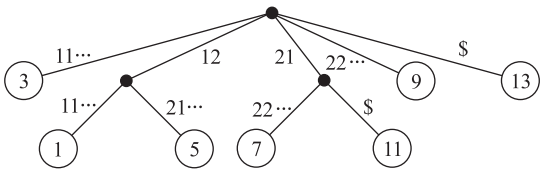


Рис. 5.13. Дерево $T_{\boldsymbol{g}'\text{зм}}$

Как видно из этого примера, дерево $T_{\boldsymbol{x}'\text{зм}}$ не обязательно будет синтаксическим деревом, поскольку в процессе замены может случиться, что выходящие из некоторого узла ребра будут заканчиваться на узлах, представляющих одинаковые буквы. В этом случае необходимо вернуться на предыдущий шаг. Но с другой стороны, одинаковыми могут быть только первые буквы на двух путях, начинающихся из одного узла, — если вторые буквы также совпадают, тогда в дереве $T_{\boldsymbol{x}'}$ должно быть два пути с одинаковой первой буквой ρ, а это противоречит тому, что данное дерево является деревом суффиксов.

Обратное преобразование дерева $T_{\boldsymbol{x}'\text{зм}}$ в дерево $T_{\boldsymbol{x}}^{\text{нч}}$ просто: в каждом внутреннем узле, предполагая, что исходящие из него ребра представлены в лексикографическом порядке, сравнивается первая буква метки ребра с первой буквой метки предшествующего ему ребра. И если эти буквы совпадают, вводится новое ребро с меткой длиной 1 (и, соответственно, новый узел на конце этого ребра). Как легко проверить, с помощью данной процедуры дерево $T_{\boldsymbol{g}'\text{зм}}$ (рис. 5.13) действительно преобразуется в дерево $T_{\boldsymbol{g}\text{нч}}$ (рис. 5.5).

Построение дерева $T_{\boldsymbol{x}'\text{зм}}$ требует одного обхода дерева $T_{\boldsymbol{x}'}$ с операцией замены во внутренних узлах. Замена выполняется за константное время, поэтому на такое преобразование необходимо время порядка $\Theta(n)$.

Шаг II. Построение четного дерева $T_{\boldsymbol{x}}^{\text{чет}}$ на основе дерева $T_{\boldsymbol{x}}^{\text{нч}}$.

Напомним, что выполнение второй поразрядной сортировки четных суффиксов строки $\boldsymbol{x}$ основано на известном лексикографическом порядке нечетных суффиксов этой строки. Для суффиксов четных номеров $2i$ и $2j$ можно получить аналог формулы (5.6):

$$\text{lcp}(2i, 2j) = \text{lcp}(2i+1, 2j+1) + 1, \quad \text{если } \boldsymbol{x}[2i] = \boldsymbol{x}[2j], \qquad (5.9)$$

и $\mathrm{lcp}(2i, 2j) = 0$ в противном случае. Тогда, зная $\mathrm{lcp}(2i+1, 2j+1)$, значение $\mathrm{lcp}(2i, 2j)$ можно вычислить за константное время. Поскольку дерево $T_{\boldsymbol{x}}^{\text{нч}}$ является деревом суффиксов, для которого справедливо равенство (5.1), то $\mathrm{LCA}(2i+1, 2j+1) = \mathrm{LCP}(2i+1, 2j+1)$ для любой пары листьев. Известный алгоритм [112] позволяет найти наименьший общий предок для любых узлов в любом дереве за константное время, основываясь на предварительном линейном упорядочении номеров узлов дерева. Используя этот алгоритм, можно вычислить $\mathrm{lcp}(2i, 2j)$ за константное время для каждой пары смежных суффиксов в лексикографически упорядоченном дереве $T_{\boldsymbol{x}}^{\text{чет}}$. Но в этом дереве опять выполняется равенство (5.1), поэтому значение $\mathrm{lcp}(2i, 2j)$ определяет $\mathrm{LCA}(2i, 2j)$, т.е. имеем всю необходимую информацию для построения дерева $T_{\boldsymbol{x}}^{\text{чет}}$.

Итак, используя $\Theta(n)$ времени для предварительной обработки дерева $T_{\boldsymbol{x}}^{\text{нч}}$, дерево $T_{\boldsymbol{x}}^{\text{чет}}$ можно построить за время $\Theta(n)$.

Шаг III. Слияние деревьев $T_{\boldsymbol{x}}^{\text{нч}}$ и $T_{\boldsymbol{x}}^{\text{чет}}$.

Этот шаг выполняется за время $\Theta(n)$.

Шаг IV. Построение LCP-дерева.

На этом шаге мы можем снова использовать алгоритм [112], вычисляющий наименьший общий предок для любой пары узлов за константное время. В качестве побочного продукта процесса построения дерева слияния $M_{\boldsymbol{x}}$ мы уже имеем соседние четные-нечетные (или нечетные-четные) пары $(2i, 2j-1)$, которые являются листьями в деревьях $T_{\boldsymbol{x}}^{\text{чет}}$ и $T_{\boldsymbol{x}}^{\text{нч}}$ соответственно. Для каждой такой пары за константное время можно вычислить $\boldsymbol{u} = \mathrm{LCA}(2i, 2j-1)$. Как указывалось ранее, родителем $\boldsymbol{u}$ в LCP-дереве является $\boldsymbol{v} = \mathrm{LCA}(2i+1, 2j)$, что также вычисляется за константное время. Таких смежных неоднородных (четные-нечетные или нечетные-четные) пар, как показано в упражнении 5.2.15, всего n. Поэтому LCP-дерево строится за время, пропорциональное n.

В нашем примере в дереве $M_{\boldsymbol{x}}$ имеется $n+1 = 13$ узлов, которые в дереве $M_{\boldsymbol{x}}$ появляются в следующем порядке: 4, 3, 1, 8, 5, 12, 2, 7, 11, 9, 6, 10, 13. Имеем 12 смежных неоднородных пар $(4,\ 3)$, $(4,\ 1)$, $(1,\ 8)$, $(8,\ 5)$, $(5,\ 12)$, $(5,\ 2)$, $(2,\ 7)$, $(2,\ 11)$, $(2,\ 9)$, $(9,\ 6)$, $(9,\ 10)$, $(10,\ 13)$, для которых надо вычислить наименьшие общие предки.

Шаг V. Построение дерева $T_{\boldsymbol{x}}$ на основе дерева $M_{\boldsymbol{x}}$ и LCP-дерева.

Как мы видели, обработка LCP-дерева сверху вниз позволяет определить узлы, исходящие ребра которых не удовлетворяют равенству (5.7). В таких узлах выполняются операции P1 и P2, которые требуют только ограниченного числа замены указателей на узлы. Таким образом, время выполнения шага V пропорционально количеству узлов в LCP-дереве, т.е. имеет порядок $O(n)$.

Корректность и эффективность

Итак, из предыдущего подраздела видно, что время $t(n)$, необходимое для выполнения первого шага алгоритма Ф, действительно имеет верхнюю границу (5.4), а другие шаги этого алгоритма требуют времени порядка $\Theta(n)$. Поэтому справедлива оценка (5.5), и мы имеем право сформулировать следующий формальный результат.

Теорема 5.2.7. Алгоритм Фарача вычисляет дерево суффиксов $T_{\boldsymbol{x}}$ для заданной строки $\boldsymbol{x}[1..n]$, определенной на индексированном алфавите, за время порядка $\Theta(n)$. ■

Несмотря на асимптотическую оптимальность алгоритма Ф, остается неясным, когда наиболее целесообразно использовать его на практике. Хотя, как мы видели, основная идея алгоритма проста, в целом алгоритм весьма сложен для реализации, по сравнению с другими алгоритмами построения деревьев суффиксов, и, кроме того, требует значительного объема памяти. С другой стороны, как мы увидим на примере алгоритмов глав 7 и 8, хорошие алгоритмы могут порождать еще лучшие алгоритмы.

5.2.5 Применение и реализация

После того как мы рассмотрели в подробностях построение дерева суффиксов, будет более понятным, как это дерево можно использовать. Определить, является ли данная строка $\boldsymbol{u}$ подстрокой строки $\boldsymbol{x}$, можно с помощью медленного сканирования (см. подраздел 5.2.1). Найти первое вхождение строки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ несложно, если предварительно ребра дерева суффиксов пометить целочисленными метками и если все суффиксы строки $\boldsymbol{x}$ являются листьями в дереве $T_{\boldsymbol{x}}$. Эффективно найти *все* вхождения строки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ можно, используя стандартные способы обхода поддерева с корнем $\boldsymbol{u}$ (если $\boldsymbol{u}$ является узлом в дереве $T_{\boldsymbol{x}}$) или обхода поддеревьев с корнями в сынах узлов, на которых оканчиваются ребра, определяющие префиксы строки $\boldsymbol{u}$ (если $\boldsymbol{u}$ не является узлом в дереве $T_{\boldsymbol{x}}$). Если же заранее известно, что строка $\boldsymbol{u}$ является подстрокой строки $\boldsymbol{x}$, то для нахождения первого вхождения строки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ можно применить быстрое сканирование. В упражнениях 5.2.16 и 5.2.17 показаны детали подобного применения дерева суффиксов.

До сих пор мы не упоминали одно из основных свойств деревьев суффиксов, которое часто определяет возможность их использования: всем различным подстрокам строки $\boldsymbol{x}$ соответствуют свои пути в дереве $T_{\boldsymbol{x}}$, начинающиеся в корне этого дерева. Поэтому реестр всех различных подстрок строки $\boldsymbol{x}$ можно получить как список всех подстрок, определенных на каждом таком пути. Даже если метки ребер имеют вид "i_1, i_2", что соответствует подстроке $\boldsymbol{x}[i_1, i_2]$, в некоторых случаях получение реестра всех подстрок может потребовать времени порядка

$\Theta(n^2)$, поскольку, как показано в упражнении 1.2.10, строка $\boldsymbol{x}$ в общем случае может иметь C_n^2 различных подстрок. Но предположим, что требуется подсчитать только количество таких подстрок. Вследствие указанного свойства дерева суффиксов, количество различных подстрок, соответствующих любому пути в дереве $T_{\boldsymbol{x}}$, равно количеству букв в метках ребер, составляющих этот путь, т.е. равно сумме величин $i_2 - i_1 + 1$, где сумма берется по всем меткам "i_1, i_2" этого пути. Для каждого ребра, имеющего метку "i_1, i_2", назовем величину $i_2 - i_1 + 1$ ***шириной*** этого ребра, т.е. ширина ребра равна количеству букв, представленных его меткой. Тогда общее количество различных подстрок в строке $\boldsymbol{x}$ равно сумме значений ширины всех ребер дерева $T_{\boldsymbol{x}}$. Используя стандартные способы обхода деревьев, эту сумму можно вычислить за время, пропорциональное количеству ребер в дереве $T_{\boldsymbol{x}}$, т.е. пропорционально n. Таким образом, можем утверждать, что *если дерево суффиксов $T_{\boldsymbol{x}}$ построено*, то количество различных подстрок в строке $\boldsymbol{x}$ можно вычислить за время $\Theta(n)$.

Покажем еще одно применение дерева суффиксов. Пусть даны две строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ и необходимо вычислить наибольший общий фактор (подстроку) $\mathrm{LCF}(\boldsymbol{x}_1, \boldsymbol{x}_2)$. (Эта задача поставлена в упражнении 2.2.17.) Используя свойства дерева суффиксов, которые показаны в процессе его построения, нетрудно показать, что данную задачу можно решить за время $\Theta(|\boldsymbol{x}_1| + |\boldsymbol{x}_2|)$, однако снова предполагая, что дерево суффиксов уже построено. Подробнее эта задача рассмотрена в упражнении 5.2.20.

Впечатляющая скорость выполнения алгоритмов, связанных с деревьями суффиксов, достигается не бесплатно, а требует определенных затрат, которые выражаются в необходимых объемах памяти. Как следует из теорем 5.2.3 и 5.2.6, в каждом узле дерева суффиксов время поиска составляет $O(\log \alpha)$, но только при условии, что в каждом узле хранятся данные, содержащие $O(\alpha)$ указателей и, возможно, другие дополнительные указатели и данные. Для индексированного алфавита поиск в каждом узле требует константного времени, но опять в каждом узле должна храниться информация объемом $O(\alpha)$. Кроме того, может потребоваться в каждом узле хранить указатель на родителя этого узла и указатели, необходимые для быстрого обхода дерева. Также необходимо $2 \log n$ бит для хранения меток каждого ребра. Таким образом, даже для малых α реализация дерева суффиксов невозможна без существенного, иногда недопустимо большого, объема памяти. Для очень малых алфавитов (например, $\alpha \leqslant 4$) дерево $T_{\boldsymbol{x}}$ можно эффективно реализовать как бинарное дерево, что исключает большие потребности в памяти. Как указывалось в обсуждении 2.1, использование деревьев суффиксов наиболее приемлемо в ситуациях, когда дерево $T_{\boldsymbol{x}}$ (или строка $\boldsymbol{x}$) не изменяется, но используется часто, а объем алфавита α мал — этим условиям удовлетворяют, например, последовательности ДНК.

И последнее замечание: в принципе, для очень малых алфавитов на практике можно достичь теоретической эффективности алгоритмов, но такая эффек-

тивность все равно будет иллюзорной. Даже если для хранения каждой буквы тратится всего два бита (как в последовательности ДНК), то для хранения строки длиной n необходим массив объемом $n/4$ байт. Кроме того, для хранения дерева суффиксов, если в каждом узле сохраняются не менее двух указателей и дополнительная служебная информация, плюс метки ребер, потребуется не менее $10n$ байт. В действительности, "экономичная" реализация алгоритма Мак-Крейта [143] потребовала $20n$ байт памяти. На практике, особенно для больших значений n, время выполнения алгоритмов с деревьями суффиксов и требуемый для их реализации объем памяти могут значительно (в десятки раз) отличаться от *теоретических оценок* как в сторону увеличения, так и в сторону уменьшения, что обусловлено как обрабатываемыми данными, так и конкретной реализацией алгоритма под управлением конкретной операционной системы. Исследование этих вопросов проведено в работе [71].

Несколько пессимистические рассуждения этого раздела ведут к необходимости построения структур суффиксов, отличных от деревьев суффиксов. Две такие структуры описаны в следующем разделе.

Упражнения 5.2

1. Покажите, что для любого целого $i \in 1..n+1$ $\mathrm{tail}(i)$ реализуется в дереве $T\boldsymbol{x}$ как конечный узел j, где $j = i + |\,\mathrm{head}(i)|$.

2. Можно заметить, что для всех элементов табл. 5.1 выполняется неравенство

$$|\,\mathrm{head}(i+1)| \geqslant |\,\mathrm{head}(i)| - 1.$$

Докажите это неравенство в общем случае (т.е. для любого положительного целого i). Докажите подобное неравенство для $\mathrm{tail}(i)$.

3. Пусть $\boldsymbol{u}$ — узел дерева $T_{\boldsymbol{x}}$ (но не корень этого дерева) и пусть s обозначает суффиксную функцию. Покажите, что

а) $s(\boldsymbol{u})$ также является узлом дерева $T_{\boldsymbol{x}}$;

б) $s(\boldsymbol{u})$ не является узлом поддерева с корнем в узле $\boldsymbol{u}$;

в) если в дереве $T_{\boldsymbol{x}}$ узел $\boldsymbol{v}$ является родителем узла $\boldsymbol{u}$, то или $s(\boldsymbol{v}) = s(\boldsymbol{u}) = \varepsilon$ или $s(\boldsymbol{v})$ является предком $s(\boldsymbol{u})$.

4. ***Источником*** называется узел $\boldsymbol{v}$ дерева $T_{\boldsymbol{x}}$, такой, что для любого узла $\boldsymbol{u}$ этого дерева $\boldsymbol{v} \neq s(\boldsymbol{u})$. Покажите, что

а) конечный узел 1 всегда будет источником;

б) если $|\boldsymbol{x}| > 1$, тогда существует источник, который не является конечным узлом;

в) если α' — количество различных букв в строке $\boldsymbol{x}$ (не считая символа \$), тогда существует ровно α' источников, которые не являются конечными узлами.

5. Приведите пример строки $\boldsymbol{x}$, для которой алгоритм МК построит дерево $T_{\boldsymbol{x}}$ за время $\Theta(n^2)$, если в этом алгоритме, вместо быстрого сканирования, использовать медленное сканирование.

6. Докажите лемму 5.2.4.

7. Для строки $\boldsymbol{x} = aabbacbbaa$ с помощью алгоритма Укк постройте дерево T_9 и опишите действия, необходимые для построения дерева T_{10}.

8. В доказательствах теорем 5.2.3 и 5.2.6 предполагалось, что в каждом узле дерева используются для хранения букв упорядоченный массив или подходящее дерево поиска. Обсудите эти и другие структуры данных, которые можно применить для хранения информации в узлах дерева, и укажите ситуации, в которых их можно применить. Можно ли уменьшить требуемый объем памяти (соответственно, время выполнения) до величины порядка $\Theta(n)$, если алфавит неупорядочен, и если можно, то как это отразится на времени выполнения (соответственно, объеме памяти)?

9. Очевидно, что алгоритм Укк более сложный в "интеллектуальном" смысле, чем алгоритм МК. Но каково соотношение их вычислительных сложностей? Предполагая, что оба алгоритма строят одинаковое дерево суффиксов и используют одинаковые структуры данных для представления информации в каждом узле дерева, что можно сказать о *реальном* времени их выполнения? Приведите примеры ситуаций, в которых свойство онлайновости алгоритма Укк является решающим преимуществом.

10. Объясните, почему для доказательства эффективности алгоритма Ф необходимо условие, что $\alpha \in O(n)$.

Совет. Рассмотрите, как работает поразрядная сортировка.

11. Используя соотношение (5.5), покажите, что для алгоритма Ф $t(n) \in O(n)$.

12. Покажите, как две поразрядные сортировки можно использовать для сортировки в возрастающем порядке упорядоченных целочисленных пар $(\boldsymbol{x}[2i-1], \boldsymbol{x}[2i])$.

13. Покажите, что порядок суффиксов $2i-1$, заданный деревом $T^{\text{нч}}_{\boldsymbol{x}}$, совпадает с порядком рангов ρ_i, заданным деревом суффиксов строки $\rho_1\rho_2\ldots\rho_{\lceil n/2\rceil}$.

14. Покажите, что дерево суффиксов $T_{\boldsymbol{x}}$ можно построить, зная лексикографический порядок суффиксов строки $\boldsymbol{x}$ и порядок наименьших общих предков смежных пар.

15. Покажите, что количество смежных четных-нечетных пар в последовательности из $n+1$ целых чисел в точности равно n только тогда, когда эти числа не однородные (т.е. не все четные или не все нечетные).

16. Предложите способ определения реберных меток "i_1, i_2" в процессе построения дерева суффиксов, который облегчал бы вычисление первого вхождения любой подстроки $\boldsymbol{u}$ в заданную строку $\boldsymbol{x}$. Затем объясните, как быстрое сканирование находит первое вхождение подстроки $\boldsymbol{u}$, если известно, что подстрока $\boldsymbol{u}$ действительно входит в строку $\boldsymbol{x}$. Определите время, необходимое для выполнения быстрого сканирования.

17. Объясните, как, используя подходящие метки ребер и способ обхода дерева суффиксов, найти все вхождения заданной подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$. Определите необходимые для этого время и объем памяти.

18. Автор утверждал, что для алфавита размером в 4 буквы использование деревьев суффиксов может потребовать памяти объемом $20n$ байт (или даже $40n$ байт), где n — длина исходной строки. Поскольку он не доказал свое утверждение, оно может показаться пессимистическим или даже пораженческим. Для строк длиной $n = 100$ Кбайт, 1 Мбайт, 5 Мбайт и алфавита размером $\alpha = 4$ оцените минимальную требуемую память для подходящего представления дерева $T_{\boldsymbol{x}}$.

19. ***Деревом позиций*** $P_{\boldsymbol{x}}$ [230] называется дерево суффиксов $T_{\boldsymbol{x}}$, измененное таким образом, что в нем лист i представляет самый короткий префикс суффикса $\boldsymbol{x}[i..n]$, который при этом не совпадает с другими подстроками строки $\boldsymbol{x}$.

а) Покажите, что i является листом дерева $P_{\boldsymbol{x}}$ только в том случае, когда $i \leqslant j_L$, другими словами, только в том случае, когда i является листом дерева $T_{\boldsymbol{x}}$.

б) Покажите, что дерево $P_{\boldsymbol{x}}$ имеет n листьев только тогда, когда буква $\boldsymbol{x}[n]$ входит в строку $\boldsymbol{x}$ только один раз.

в) Предложите простой алгоритм построения дерева $P_{\boldsymbol{x}}$ на основе дерева $T_{\boldsymbol{x}}$.

20. Предложите алгоритм на основе дерева суффиксов для вычисления наибольшего общего фактора для двух заданных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ за время, пропорциональное $|\boldsymbol{x}_1| + |\boldsymbol{x}_2|$. Покажите, что ваш алгоритм можно расширить для обработки $k \geqslant 2$ строк.

Совет. Сформируйте строку $\boldsymbol{x} = \boldsymbol{x}_1\$_1\boldsymbol{x}_2\$_2$ с использованием разных символов окончания строк $\$_1$ и $\$_2$. Заметьте, что $\mathrm{LCF}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ в дереве $T_{\boldsymbol{x}}$ соответствует узел $\boldsymbol{u}$ наибольшей длины, такой, что поддерево с корнем в узле $\boldsymbol{u}$ содержит символ $\$_1$.

5.3 Альтернативные структуры для представления суффиксов

В этом разделе мы рассмотрим две структуры, альтернативные деревьям суффиксов. Обе эти структуры имеют перед деревьями суффиксов некоторые теоретические преимущества, которые должны проявиться на практике. Хотя описываемые здесь структуры различны между собой, они имеют общее свойство, которое заключается в том, что обе они строятся на основе деревьев суффиксов.

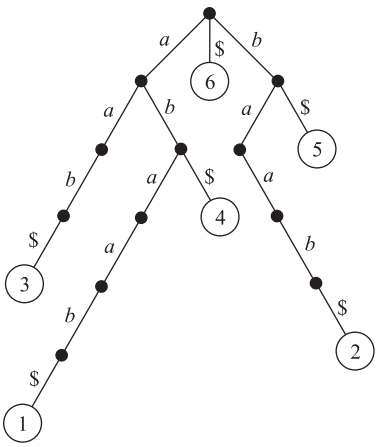
5.3.1 Ориентированные ациклические графы слов

Необходимость введения структур, отличных от деревьев суффиксов, вытекает из простого наблюдения, которое читатель уже должен был сделать: дерево суффиксов представляет по отдельности все одинаковые повторяющиеся подстроки, что кажется излишним, — узлам, представляющим повторяющиеся подстроки, соответствуют ребра с одинаковыми метками, и различие между ними проявляется только в метках узлов-листьев. Например, на рис. 5.2 поддерево $T'_{\boldsymbol{v}}$, соответствующее подстроке $\boldsymbol{v} = aababaabaab\$$, встречается дважды, присоединенное к узлам b и ab; поддерево $T'_{\boldsymbol{w}}$, соответствующее подстроке $\boldsymbol{w} = abaabaab\$$, встречается три раза, присоединенное к узлам $aaba$, $baaba$ и $abaaba$. Подобным образом на рис. 5.4 в дереве суффиксов T_6 поддерево $T'_{aab\$}$ встречается дважды, присоединенное к узлам b и ab. Поэтому ориентированные ациклические графы слов (сокращенно, ОАГС) вводятся как средство для избежания таких излишеств [35, 70].

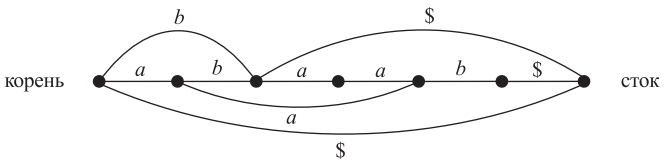
Но прежде чем приступить к непосредственному описанию ОАГС, укажем способ определения наличия в дереве суффиксов $T_{\boldsymbol{x}}$ одинаковых поддеревьев. В описании алгоритма Укконена для простоты мы предполагали, что $\boldsymbol{x} = \boldsymbol{x}[1..n+1]$, где $\boldsymbol{x}[n+1] = \$$. Поскольку, как будет показано далее, ОАГС можно легко получить путем преобразования "неплотных" деревьев суффиксов, мы на время предположим, что каждая подстрока $\boldsymbol{u}$ строки $\boldsymbol{x}$ представляется в дереве $T_{\boldsymbol{x}}$ отдельным узлом, даже если такой узел имеет только одного сына. Пример неплотного дерева суффиксов для строки $\boldsymbol{x} = abaab\$$ показан на рис. 5.14, *а*.

Для каждой непустой подстроки $\boldsymbol{u}$ строки $\boldsymbol{x}$ определим множество $\Pi_{\boldsymbol{u}}$ всех позиций в строке $\boldsymbol{x}$, где оканчиваются подстроки $\boldsymbol{u}$. Назовем такое множество ***множеством окончаний***. По определению положим $\Pi_{\varepsilon} = \{1, 2, \ldots, n+1\}$. С помощью множества окончаний можно отобразить структуру дерева суффиксов $T_{\boldsymbol{x}}$. Пусть $\boldsymbol{u}$ — непустая подстрока строки $\boldsymbol{x}$, которая входит в строку $\boldsymbol{x}$ k раз в позициях $j_1, j_2, \ldots, j_k$. Эти номера позиций являются метками листьев в дереве $T_{\boldsymbol{x}}$ и представляют префиксы подстроки $\boldsymbol{u}$. Множество окончаний для подстроки $\boldsymbol{u}$ вычисляется следующим образом:

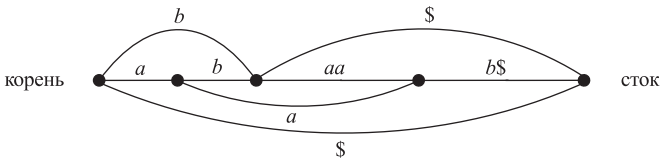
$$\Pi_{\boldsymbol{u}} = \{j_1 + |\boldsymbol{u}| - 1, j_2 + |\boldsymbol{u}| - 1, \ldots, j_k + |\boldsymbol{u}| - 1\}.$$



а) Неплотное дерево суффиксов



б) Граф D_x (7 узлов, 10 дуг)



в) Граф D_x (5 узлов, 8 дуг)

Рис. 5.14. Преобразование дерева суффиксов в ОАГС для строки $\boldsymbol{x} = abaab\$$

В случае, когда $\boldsymbol{u} = \varepsilon$, множество Π_ε является множеством меток листьев, которые представляют префиксы ε, т.е. множеством меток всех листьев. Отметим, что в случае $\boldsymbol{u} \neq \varepsilon$ множество $\Pi_{\boldsymbol{u}}$ является подмножеством множества Π_ε.

Рассмотрим несколько примеров. Для одиночной буквы λ Π_λ представляет множество всех позиций в строке $\boldsymbol{x}$, в которых встречается эта буква (и, конечно, это — множество меток листьев, имеющих префикс λ). Если $\boldsymbol{u} = \boldsymbol{x}[j..n+1]$ для всех j, $1 \leqslant j \leqslant n+1$, тогда $\Pi_{\boldsymbol{u}} = \{n+1\}$, показывая тем самым, что подстроки

строки $\boldsymbol{x}$ могут быть префиксами только одного суффикса $\boldsymbol{x}[j..n+1]$. В общем случае, если $\boldsymbol{x}[j]$ — буква, которая входит в строку $\boldsymbol{x}$ только один раз, тогда для произвольной подстроки $\boldsymbol{u}$, содержащей эту букву, $|\Pi_{\boldsymbol{u}}| = 1$. В последнем примере подстроке $\boldsymbol{u}$ соответствует или лист дерева $T_{\boldsymbol{x}}$, или узел, имеющий только одного сына.

Для построения ориентированных ациклических графов слов практический интерес представляют подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$, имеющие одинаковые множества окончаний. В этом случае эти подстроки в дереве $T_{\boldsymbol{x}}$ представимы узлами, которые являются корнями одинаковых поддеревьев. Если $\Pi_{\boldsymbol{u}_1} = \Pi_{\boldsymbol{u}_2}$, то в этом случае будем говорить, что подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ П-***эквивалентны***. В упражнении 5.3.1 доказывается, что П-эквивалентность действительно является отношением эквивалентности.

Лемма 5.3.1. Пусть $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ — две непустые подстроки строки $\boldsymbol{x}$, такие, что $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$. Тогда подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ будут П-эквивалентными только тогда, когда подстрока $\boldsymbol{u}_1$ встречается в строке $\boldsymbol{x}$ только как суффикс подстроки $\boldsymbol{u}_2$.

Доказательство. Если $\Pi_{\boldsymbol{u}_1} = \Pi_{\boldsymbol{u}_2}$, тогда по определению множеств окончаний подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ оканчиваются в одинаковом множестве позиций. И поскольку $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$, то $\boldsymbol{u}_1$ является суффиксом $\boldsymbol{u}_2$. Обратно: если $\boldsymbol{u}_1$ встречается в строке $\boldsymbol{x}$ только как суффикс $\boldsymbol{u}_2$, тогда множество окончаний для $\boldsymbol{u}_1$ должно совпадать с аналогичным множеством подстроки $\boldsymbol{u}_2$. ■

Можно интерпретировать этот результат в терминах знакомой нам суффиксной функции, введенной в разделе 5.2.1. Пусть s^k ($k \geqslant 1$) обозначает последовательное применение k раз функции s, т.е. если $s(\boldsymbol{u}_3) = \boldsymbol{u}_2$ и $s(\boldsymbol{u}_2) = \boldsymbol{u}_1$, тогда $s^2(\boldsymbol{u}_3) = \boldsymbol{u}_1$. Лемма 5.3.1 говорит о том, что если подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ П-эквивалентны и $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$, тогда существует такое целое положительное число k, что $s^k(\boldsymbol{u}_2) = \boldsymbol{u}_1$. Таким образом, с помощью суффиксной функции можно определить число вхождений П-эквивалентных подстрок в строку $\boldsymbol{x}$.

Лемма 5.3.2. Пусть $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ — две непустые П-эквивалентные подстроки строки $\boldsymbol{x}$, такие, что $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$. Тогда для некоторого целого положительного числа k $s^k(\boldsymbol{u}_2) = \boldsymbol{u}_1$ и подстрока $\boldsymbol{u}_1$ будет П-эквивалентной всем подстрокам $s^i(\boldsymbol{u}_2)$, $i = 1, 2, \ldots, k$.

Доказательство. Пусть $\boldsymbol{u} = s^i(\boldsymbol{u}_2)$ для некотрого $i \in 1..k$. Вследствие леммы 5.3.1, подстрока $\boldsymbol{u}_1$ может входить в строку $\boldsymbol{x}$ только как суффикс подстроки $\boldsymbol{u}_2$. Тогда подстрока $\boldsymbol{u}$ также должна входить в строку $\boldsymbol{x}$ только как суффикс подстроки $\boldsymbol{u}_2$. Но тогда, вследствие леммы 5.3.1, подстрока $\boldsymbol{u}$ будет П-эквивалентной подстроке $\boldsymbol{u}_2$ и, следовательно, П-эквивалентной подстроке $\boldsymbol{u}_1$. ■

Далее покажем, что множества окончаний не могут *частично* перекрываться. Отсюда будет следовать, что эти множества соответствуют структуре дерева.

Лемма 5.3.3. Пусть $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ — две непустые подстроки строки $\boldsymbol{x}$, такие, что $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$. Тогда может выполняться только одно из следующих утверждений:

а) $\Pi_{\boldsymbol{u}_2} \cap \Pi_{\boldsymbol{u}_1} = \varnothing$;

б) $\Pi_{\boldsymbol{u}_2} \subset \Pi_{\boldsymbol{u}_1}$.

Доказательство тривиально, когда $\boldsymbol{u}_1 = \varepsilon$. Поэтому предположим, что подстрока $\boldsymbol{u}_1$ не пустая. Пусть множества Π_{u_1} и $\Pi_{\boldsymbol{u}_2}$ имеют общий элемент. Этот элемент определяет конечную позицию как подстроки $\boldsymbol{u}_1$, так и подстроки $\boldsymbol{u}_2$. Поэтому подстрока $\boldsymbol{u}_1$ является суффиксом подстроки $\boldsymbol{u}_2$. Но тогда каждый элемент множества $\Pi_{\boldsymbol{u}_2}$ должен входить в множество $\Pi_{\boldsymbol{u}_1}$. ■

Теперь можно "связать" понятие П-эквивалентности с наличием одинаковых (изоморфных) поддеревьев в дереве $T_{\boldsymbol{x}}$.

Лемма 5.3.4. Пусть $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ — узлы дерева суффиксов $T_{\boldsymbol{x}}$. Поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$ дерева $T_{\boldsymbol{x}}$ будут изоморфны тогда и только тогда, когда подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ будут П-эквивалентны.

Доказательство тривиально, когда $\boldsymbol{u}_1 = \boldsymbol{u}_2$. Поэтому полагаем, что $\boldsymbol{u}_1 \neq \boldsymbol{u}_2$. Поскольку пустая строка ε П-эквивалентна сама себе, также полагаем, что подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ не пусты.

Пусть $\Pi_{\boldsymbol{u}_2} = \Pi_{\boldsymbol{u}_1}$. Без потери общности можно считать, что $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$. Тогда, вследствие леммы 5.3.1, подстрока $\boldsymbol{u}_1$ может входить в строку $\boldsymbol{x}$ только как суффикс подстроки $\boldsymbol{u}_2$. В таком случае поддерево $T'_{\boldsymbol{u}_2}$ с корнем в узле $\boldsymbol{u}_2$ является поддеревом для суффикса $\boldsymbol{u}_1$ подстроки $\boldsymbol{u}_2$ и поэтому должно совпадать с поддеревом $T'_{\boldsymbol{u}_1}$.

Теперь пусть совпадают поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$. Тогда в строке $\boldsymbol{x}$ существует суффикс $\boldsymbol{u}_1\boldsymbol{x}[j..n+1]$ только в том случае, когда существует суффикс $\boldsymbol{u}_2\boldsymbol{x}[j..n+1]$. Для каждого такого целого j подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ заканчиваются в одной и той же позиции $j-1$. Поскольку эти подстроки не имеют окончаний в других позициях, отличных от позиций j, то отсюда следует, что $\Pi_{\boldsymbol{u}_2} = \Pi_{\boldsymbol{u}_1}$. ■

Теперь мы можем дать четкое определение ориентированному ациклическому графу слов для данной строки $\boldsymbol{x}$ (будем обозначать такой граф как $D_{\boldsymbol{x}}$) и показать, как он строится на основании "неплотного" дерева суффиксов $T_{\boldsymbol{x}}$. Иллюстрацией к процессу построения графа $D_{\boldsymbol{x}}$ послужит рис. 5.14. Сначала опишем этапы построения графа $D_{\boldsymbol{x}}$.

1. Для всех П-эквивалентных узлов $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$ произведем слияние поддеревьев $T'_{\boldsymbol{u}_i}$, $i = 1, 2, \ldots, k$ в единый подграф. Эту операцию можно выполнить путем слияния узлов в каждом поддереве.

Поскольку, как мы видели, $\Pi_{\boldsymbol{x}[j..n+1]} = \{n+1\}$ для любого листа $j \in 1..n+1$, поэтому все листья П-эквивалентны. Вследствие леммы 5.3.4, на этом шаге все листья сольются в один узел, который назовем ***стоком***.

На рис. 5.14 можно найти четыре множества П-эквивалентных узлов: множество всех листьев, $\{aa, baa, abaa\}$, $\{ba, aba\}$ и $\{b, ab\}$.

2. Дуги графа ориентированы от корня в направлении, соответствующем отношениям "родитель-сын" в дереве $T_{\boldsymbol{x}}$. На рис. 5.14 ориентация дуг не показана: она совпадает с направлением слева направо.

Дополнительный третий шаг, показанный на рис. 5.14, *в*, производит "зачистку" графа $D_{\boldsymbol{x}}$ — удаляются узлы, имеющие единичные полустепени входа и исхода.[7] Отметим, что для хранения даже первоначального графа $D_{\boldsymbol{x}}$ хватает линейной (относительно n) памяти.

Отметим, что построенный ОАГС по буквенно представляет те же $n+1$ суффиксы, что и соответствующее дерево суффиксов. И так же, как в дереве суффиксов, в графе метки на исходящих из какого-либо узла дугах попарно различны. ОАГС имеет в точности $n+1$ путей от корня до стока длиной $1, 2, \ldots, n+1$, которые можно проверить теми же способами, что и пути в дереве суффиксов. Очевидно, что граф $D_{\boldsymbol{x}}$ ацикличен, поскольку все пути от корня до стока ориентированы согласно ориентации "родитель-сын" в дереве $T_{\boldsymbol{x}}$.

Это определение ОАГС, данное в полуалгоритмической форме, на самом деле не является алгоритмом. Алгоритм построения ОАГС обязательно будет зависеть от метода "разуплотнения" дерева $T_{\boldsymbol{x}}$, который может потребовать памяти порядка $\Theta(n^2)$, что может быть нежелательным. Мы оставим "на потом" вопрос об эффективном построении ОАГС, сначала разберемся с требованиями, предъявляемыми к объему памяти, необходимому для хранения ОАГС.

Теорема 5.3.5. Для строки $\boldsymbol{x} = \boldsymbol{x}[1..n]\$$ длиной $n+1$ граф $D_{\boldsymbol{x}}$ имеет $N \leqslant 2n+1$ узлов и не более $N + n - 1 \leqslant 3n$ дуг.

Доказательство. Каждый узел графа $D_{\boldsymbol{x}}$ соответствует одному или нескольким множествам окончаний подстрок, определяемых узлами дерева $T_{\boldsymbol{x}}$. Лемма 5.3.3 говорит о том, что эти множества не могут частично пересекаться. Также очевидно, что для корня $\Pi_\varepsilon = \{1, 2, \ldots, n+1\}$. Узлы графа $D_{\boldsymbol{x}}$ должны соответствовать узлам поддеревьев с корнями в узлах $\{1, 2, \ldots, n+1\}$, у которых потомки соответствуют непустым непересекающимся множествам, объединение которых включается в узел родителя. В упражнении 5.3.4 предложено показать, что такое дерево содержит не более $2n+1$ узлов.

[7] Полустепенью входа (исхода) узла (вершины) ориентированного графа называется количество дуг, входящих в данный узел (выходящих из данного узла). — *Примеч. ред.*

Теперь рассмотрим остовное дерево[8] $ST_{\boldsymbol{x}}$ графа $D_{\boldsymbol{x}}$, который показывает путь от корня до стока наибольшей длины, помеченный буквами строки $\boldsymbol{x}$, как показано на рис. 5.15. Остовное дерево $ST_{\boldsymbol{x}}$ имеет N узлов и $N-1$ ребер. Теперь в это остовное дерево будем по одной добавлять дуги графа $D_{\boldsymbol{x}}$. Поскольку добавление каждой такой дуги также добавляет не менее одного нового пути, а всего в графа $D_{\boldsymbol{x}}$ путей ровно n, то отсюда следует, что общее количество дуг в графе $D_{\boldsymbol{x}}$ не более $(N-1)+n$. ■

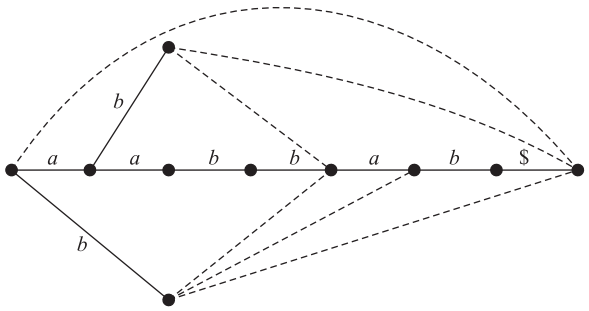


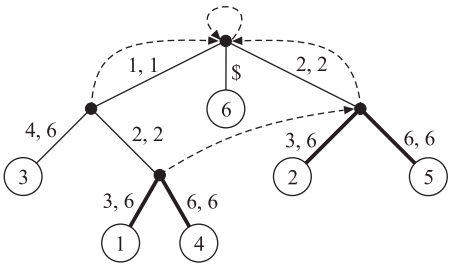
Рис. 5.15. Остовное дерево графа $D_{\boldsymbol{x}}$ для $\boldsymbol{x} = aabbab\$$ ($N = 10$, $n+1 = 7$)

Из последней теоремы вытекает искомый результат, что для хранения ОАГС $D_{\boldsymbol{x}}$ требуется память, объем которой пропорционален длине строки $\boldsymbol{x}$. Труднее разработать алгоритм построения графа $D_{\boldsymbol{x}}$, который выполнялся бы за время, пропорциональное $|\boldsymbol{x}|$. В решении последней проблемы может помочь суффиксная функция s, обладающая полезными свойствами, установленными в лемме 5.3.2. Эти свойства позволяют установить изоморфизмы поддеревьев в дереве суффиксов $T_{\boldsymbol{x}}$ и, следовательно, П-эквивалентность соответствующих подстрок.

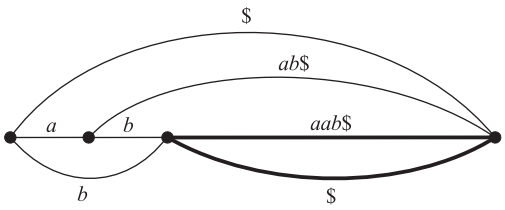
Предложим следующий алгоритм преобразования дерева $T_{\boldsymbol{x}}$ в граф $D_{\boldsymbol{x}}$, основанный на использовании суффиксной функции. Кратко этот алгоритм можно представить в виде двух последовательных шагов.

1. На первом шаге на основе суффиксных связей дерева $T_{\boldsymbol{x}}$ строится ***ориентированный ациклический граф*** (сокращенно, — ОАГ; этот граф может не совпадать с ОАГС). Процесс создания ОАГ для строки $\boldsymbol{x} = abaab\$$ показан на рис. 5.16, *а* и *б*.
2. На втором шаге минимизируется ширина (см. определение в разделе 5.2.5) входящих дуг для каждого узла ОАГ. Этот процесс показан на рис. 5.16, *а* и *б*.

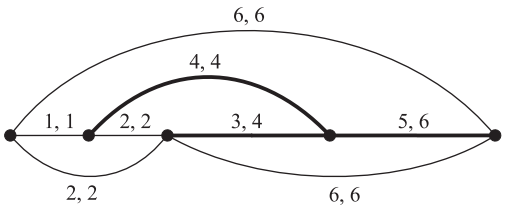
[8] *Остовным деревом* графа называется минимальный (по количеству ребер) связный ациклический граф, содержащий все вершины исходного графа. — *Примеч. ред.*



а) Суффиксные связи в дереве суффиксов
(**толстыми линиями** обозначены совпадающие поддеревья)



б) ОАГ (**толстыми линиями**
обозначены слитые поддеревья)



в) ОАГС (толстыми линиями
обозначены измененные дуги)

Рис. 5.16. Преобразование дерева суффиксов в ОАГС (см. рис. 5.14)

Следующая лемма обосновывает первый шаг алгоритма.

Лемма 5.3.6. Пусть $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ — два узла дерева суффиксов $T_{\boldsymbol{x}}$, такие, что $|\boldsymbol{u}_1| \leqslant |\boldsymbol{u}_2|$. Тогда поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$ изоморфны (будут совпадать), если выполняются условия:

а) эти поддеревья содержат одинаковое количество листьев;

б) $s^k(\boldsymbol{u}_2) = \boldsymbol{u}_1$ для некоторого целого положительного числа k.

Доказательство тривиально, когда $\boldsymbol{u}_1 = \varepsilon$, поскольку в этом случае $T'_\varepsilon = T_{\boldsymbol{x}}$. Поэтому будем считать, что $\boldsymbol{u}_1 \neq \varepsilon$.

Предположим, что поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$ изоморфны. Тогда, вследствие леммы 5.3.4, подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ П-эквивалентны и, как результат леммы 5.3.2, существует целое число k, такое, что $s^k(\boldsymbol{u}_2) = \boldsymbol{u}_1$. Тогда, как показано в упражнении 5.3.3, поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$ имеют одинаковое количество листьев. Это доказывает достаточность утверждения леммы.

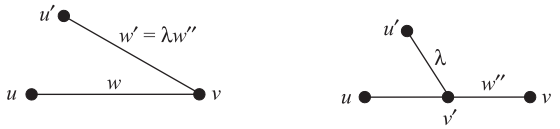
Теперь предположим, что выполняются условия *а*) и *б*) леммы. Обозначим через $v_{\boldsymbol{u}}$ количество листьев в поддереве $T'_{\boldsymbol{u}}$. Заметим, что $v_{\boldsymbol{u}}$ равно количеству вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$. По условию *б*) леммы $\boldsymbol{u}_1$ является суффиксом подстроки $\boldsymbol{u}_2$, т.е. каждый лист в поддереве $T'_{\boldsymbol{u}_2}$ соответствует листу в поддереве $T'_{\boldsymbol{u}_1}$. По условию *а*) леммы $v_{\boldsymbol{u}_2} = v_{\boldsymbol{u}_1}$, поэтому подстрока $\boldsymbol{u}_1$ может входить в строку $\boldsymbol{x}$ только как суффикс подстроки $\boldsymbol{u}_2$. Тогда, вследствие леммы 5.3.1, $\Pi_{\boldsymbol{u}_2} = \Pi_{\boldsymbol{u}_1}$, теперь из леммы 5.3.4 следует, что поддеревья $T'_{\boldsymbol{u}_1}$ и $T'_{\boldsymbol{u}_2}$ совпадают.■

Подчеркнем, что лемма 5.3.6 справедлива только в случае, когда $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ являются листьями дерева суффиксов. Поэтому процесс слияния одинаковых поддеревьев включает слияние всех листьев дерева $T_{\boldsymbol{x}}$ в узел-сток графа $D_{\boldsymbol{x}}$. Все другие изоморфные поддеревья можно локализовать, если следовать суффиксным связям в дереве $T_{\boldsymbol{x}}$. Такие одинаковые поддеревья будут сливаться только в том случае, когда они содержат одинаковое количество листьев. Процесс слияния можно реализовать за время, пропорциональное величине n, если выполнить предварительную обработку дерева $T_{\boldsymbol{x}}$, как показано ниже в алгоритме 5.3.1. (Эту процедуру предварительной обработки можно выполнить как "побочный продукт" процесса построения самого дерева $T_{\boldsymbol{x}}$.)

Алгоритм 5.3.1

▷ *Преобразование дерева суффиксов $T_{\boldsymbol{x}}$ в ОАГСD$_x$*
▷ *Предварительная обработка дерева $T_{\boldsymbol{x}}$*
обход дерева $T_{\boldsymbol{x}}$:
  а) сохранение в каждом узле $\boldsymbol{u}$ количества листьев в поддереве T'_u
  б) сохранение узла-источника для каждой цепочки суффиксных связей
▷ *Шаг 1: построение ОАГ из дерева $T_{\boldsymbol{x}}$*
for каждый сохраненный узел-источник $\boldsymbol{u}$ **do**
  ▷ *Проверка цепочки суффиксных связей до корня*
  while $s(\boldsymbol{u}) \neq \varepsilon$ **do**
    if $T'_{\boldsymbol{u}}$ и $T'_{s(\boldsymbol{u})}$ имеют одинаковое количество листьев **then**
      слияние $T'_{\boldsymbol{u}}$ и $T'_{s(\boldsymbol{u})}$
    $\boldsymbol{u} \leftarrow s(\boldsymbol{u})$
  ▷ *Шаг 2: построение $D_{\boldsymbol{x}}$ на основе ОАГ*
for каждая вершина $\boldsymbol{v}$ **do**

определение входящей дуги $(\boldsymbol{u}, \boldsymbol{v})$ наибольшей ширины $|\boldsymbol{w}|$
for каждая входящая дуга $(\boldsymbol{u}', \boldsymbol{v})$ ширины $1 < |\boldsymbol{w}'| < |\boldsymbol{w}|$ **do**
    удаление дуги $(\boldsymbol{u}', \boldsymbol{v})$
    создание дуги $(\boldsymbol{u}', \boldsymbol{v}')$ ▷ *как показано на рис. 5.17*



а) Дуга w' ОАГ б) Дуга λ ОАГС

Рис. 5.17. Преобразование входящих дуг ОАГ в дуги ОАГС

Как видно на рис. 5.16, *б* и *в*, граф ОАГ может не быть графом ОАГС: ОАГ может иметь дуги (подобные дуге с меткой $ab\$$ на рис. 5.16, *б*), метки которых можно определить как суффиксы меток на других дугах (на том же рисунке метка $aab\$$ имеет суффикс $ab\$$). Эта проблема порождена тем, что используется дерево $T_{\boldsymbol{x}}$, представленное в компактной форме, поэтому некоторые изоморфные поддеревья (такие, как ребро $b\$$ на рис. 5.14, *а*) могут быть не найдены.

На втором шаге алгоритма 5.3.1 находятся "пропавшие" суффиксы путем просмотра входящих дуг для каждого узла $\boldsymbol{v}$ ОАГ. Сначала определяется дуга $(\boldsymbol{u}, \boldsymbol{v})$ с меткой $\boldsymbol{w}$ наибольшей длины и затем одна за другой просматриваются другие дуги с метками $\boldsymbol{w}'$ длиной $2 \leqslant |\boldsymbol{w}'| \leqslant |\boldsymbol{w}|$. Если $\boldsymbol{w}'$ является суффиксом $\boldsymbol{w}$, то такая дуга удаляется, а вместо нее добаляется дуга $(\boldsymbol{u}', \boldsymbol{v}')$ с меткой единичной длины, как показано на рис. 5.17.

Приведенные выше рассуждения позволяют сформулировать следующую теорему.

Теорема 5.3.7. Алгоритм 5.3.1 корректно вычисляет ОАГС $D_{\boldsymbol{x}}$ для данной строки $\boldsymbol{x}$ на основе ранее построенного дерева суффиксов $T_{\boldsymbol{x}}$. ■

Очевидно, что время выполнения алгоритма 5.3.1 должно каким-то образом зависеть от размера алфавита, поскольку этот размер влияет на построение дерева суффиксов. Но поскольку мы предполагали, что дерево суффиксов уже построено, преобразование дерева $T_{\boldsymbol{x}}$ в граф $D_{\boldsymbol{x}}$ можно выполнить за время, линейное относительно $|\boldsymbol{x}|$.

Теорема 5.3.8. Алгоритм 5.3.1 преобразует дерево суффиксов $T_{\boldsymbol{x}}$ для данной строки $\boldsymbol{x}$ в ОАГС $D_{\boldsymbol{x}}$ за время порядка $\Theta(n)$.

Доказательство. В алгоритме 5.3.1 предварительная обработка дерева суффиксов требует посетить каждый узел дерева $T_{\boldsymbol{x}}$ для вычисления количества листьев

в поддереве T'_u, для чего можно использовать стандартный способ обхода дерева в обратном порядке. Во время обхода дерева $T_{\boldsymbol{x}}$ можно сформировать список узлов, которые или являются листьями или указывают на суффиксную связь. Поскольку дерево $T_{\boldsymbol{x}}$ имеет $\Theta(n)$ узлов (метки ребер не исследуются на этом этапе), то при использовании стандартного способа обхода дерева предварительная обработка потребует времени порядка $\Theta(n)$.

На первом шаге алгоритма проверяется каждый узел каждой цепочки суффиксных связей от источника до корня. Если на этапе предварительной обработки получен список всех узлов, от которых начинается цепочка суффиксных связей, то "слияние" поддеревьев требует только константного времени для переустановки указателей. Поскольку количество узлов с суффиксными связями имеет порядок $\Theta(n)$, выполнение первого шага алгоритма также требует времени порядка $\Theta(n)$.

На втором шаге обработка каждой дуги графа ОАГ также требует только константного времени. Количество дуг в ОАГ равно количеству дуг в ОАГС, по теореме 5.3.5 дуг в ОАГС всего $\Theta(n)$. Поэтому общее время выполнения второго шага также имеет порядок $\Theta(n)$. ■

Отметим, что в недавних работах [68, 127] описаны алгоритмы, которые вычисляют ОАГС непосредственно, без предварительного вычисления дерева суффиксов.

Недавние исследования показали [143, 127, 62], что при применении соответствующих методов ОАГС требует значительно меньше памяти для его хранения, чем соответствующие деревья суффиксов. Но основное преимущество ОАГС заключается не в этом, а в том, что они предлагают другой взгляд на структуру суффиксов строковых последовательностей, что показывают леммы 5.3.1–5.3.4 и 5.3.6. В следующем разделе рассмотрим еще одну структуру суффиксов, которая также более экономична по сравнению с деревьями суффиксов и, кроме того, не зависит от размера алфавита.

5.3.2 Массивы суффиксов

Как обычно, предполагаем, что задана строковая последовательность $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Как и выше во всех многочисленных обсуждениях деревьев суффиксов, полагаем, что каждый суффикс $\boldsymbol{x}[j..n]$ строки $\boldsymbol{x}$ ($j = 1, 2, \ldots, n$) определяется просто номером j листа в дереве $T_{\boldsymbol{x}}$. Тогда ***массив суффиксов*** (или *суффиксная строка*!) $\boldsymbol{\sigma} = \boldsymbol{\sigma}_{\boldsymbol{x}}$ — это просто массив n целых чисел $j = \boldsymbol{\sigma}[i]$, определяющих суффиксы, упорядоченные в возрастающем лексикографическом порядке (т.е. i — номер суффикса $\boldsymbol{x}[j..n]$ в этом упорядоченном множестве всех суффиксов строки $\boldsymbol{x}$) [171, 172]. Например, строка

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{g} = & a & b & a & a & b & a & a & b, \end{array}$$

введенная в разделе 2.1, имеет следующие суффиксы, упорядоченные в лексикографическом порядке.

6	aab	1	$abaabaab$
3	$aabaab$	8	b
7	ab	5	$baab$
4	$abaab$	2	$baabaab$

Поэтому для данной строки массив суффиксов имеет вид $\boldsymbol{\sigma}_g = 63741852$.

Отметим одно почти очевидное, но очень важное свойство массива суффиксов: все суффиксы строки $\boldsymbol{x}$, имеющие одинаковый префикс, располагаются последовательно в массиве $\boldsymbol{\sigma}$. Это свойство незаменимо для поиска всех вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ — всем этим вхождениям в массиве $\boldsymbol{\sigma}$ соответствует ряд последовательных элементов. Если в нашем примере положить $\boldsymbol{u} = ab$, то этой подстроке будут соответствовать элементы массива $\boldsymbol{\sigma}[3..5]$. Если $\boldsymbol{u} = baa$, тогда этой подстроке соответствуют элементы $\boldsymbol{\sigma}[7..8]$.

Имея дерево суффиксов $T_{\boldsymbol{x}}$, массив $\boldsymbol{\sigma}_{\boldsymbol{x}}$ можно вычислить путем обхода дерева $T_{\boldsymbol{x}}$ в глубину, когда для каждого узла просматриваются в лексикографическом порядке первые буквы меток ребер, исходящих из данного узла. Если символу \$ окончания строки присвоить наименьшее лексикографическое значение, тогда в процессе такого обхода дерева $T_{\boldsymbol{x}}$ все листья этого дерева будут посещаться в лексикографическом порядке. Таким образом, мы почти доказали следующую теорему.

Теорема 5.3.9. Массив суффиксов $\boldsymbol{\sigma}_{\boldsymbol{x}}$ для данной строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$, определенной на алфавите A размером α, можно вычислить на основе ранее построенного дерева суффиксов $T_{\boldsymbol{x}}$ за время порядка $O(n \log \alpha)$. ■

В зависимости от вида структуры данных, реализованной в каждом узле $T_{\boldsymbol{x}}$, время вычисления массива $\boldsymbol{\sigma}_{\boldsymbol{x}}$ в определенных ситуациях действительно может достигать верхней границы $\Theta(n)$. Более подробно эта тема рассмотрена в упражнении 5.3.7. В работе [172] показано, что среднее (по всем возможным строкам $\boldsymbol{x}$) время вычисления массива $\boldsymbol{\sigma}_{\boldsymbol{x}}$ составляет $\Theta(n)$, причем без использования дерева суффиксов. Этот прямой метод вычисления массива суффиксов требует значительно меньшего объема памяти, чем необходимо для построения дерева суффиксов, но за счет небольшого увеличения времени вычислений [172, 151, 31, 200].

Теперь, зная, что такое массив суффиксов и как его вычислить, мы основное внимание в оставшейся части данного раздела уделим эффективному использованию этих массивов для решения основной задачи поиска: найти в строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$ все $k \geqslant 0$ вхождений подстроки $\boldsymbol{u} = \boldsymbol{u}[1..m]$. Мы представим три алгоритма решения этой задачи, назовем их условно ***наивным***, ***простым*** и ***сложным*** алгоритмами соответственно.

Наивный и простой алгоритмы используют только массив суффиксов $\boldsymbol{\sigma}$, n элементов которого требуют $\log n$ бит для хранения каждого значения-элемента. Они выполняются за время, не превышающее теоретической верхней границы. На практике простой алгоритм выполняется быстрее наивного, а также, согласно работе [172], быстрее сложного алгоритма. Сложный алгоритм, кроме массива суффиксов, использует также дополнительный целочисленный массив объемом, не превышающим $3n$ (этот дополнительный массив можно получить как побочный продукт в процессе вычисления массива $\boldsymbol{\sigma}$). Этот алгоритм *теоретически* в самом худшем случае выполняется за время, близкое к времени решения данной задачи с помощью деревьев суффиксов и ОАГС, но *на практике* он выполняется значительно быстрее. В табл. 5.2 приведены основные временны́е и пространственные характеристики этих алгоритмов в предположении, что массив суффиксов уже вычислен (на этапе предварительной обработки), а дерево суффиксов, если оно использовалось для вычисления массива $\boldsymbol{\sigma}$, удалено из памяти.

Таблица 5.2. Характеристики алгоритмов (на основе массивов суффиксов) нахождения всех вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$

Алгоритм	Требуемый объем памяти (байты)	Теоретическая верхняя граница времени выполнения
Наивный	$n\log(n/8)$	$O(m(\log n + k))$
Простой	$n\log(n/8)$	$O(m(\log n + k))$
Сложный	$4n\log(n/8)$	$O(m + \log n + k)$

Основная идея всех трех алгоритмов элементарна и известна любому студенту, изучавшему курс компьютерных наук, — это бинарный поиск! Наивный алгоритм не использует ничего, кроме бинарного поиска, который использует $\boldsymbol{\sigma}$ как индексный массив для выбора M-го лексикографически наибольшего суффикса. Как показано в листинге алгоритма 5.3.2, он (алгоритм) возвращает ненулевое значение, если подстрока $\boldsymbol{u} = \boldsymbol{u}[1..m]$ входит в строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$, при этом возвращаемое значение равно целому числу j, такому, что $\boldsymbol{u} = \boldsymbol{x}[j..j+m-1]$. Алгоритм выполняет $O(\log n)$ сравнений, каждое из которых требует $O(m)$ времени. Поэтому все сравнения выполняются за время $O(m\log n)$. Как упоминалось выше, все k вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ индексированы соседними элементами массива $\boldsymbol{\sigma}$. Таким образом, чтобы найти остальные $k-1$ вхождений, надо просмотреть в массиве $\boldsymbol{\sigma}$ соседние позиции из интервала $[L, R]$, который содержит M-ю позицию первого найденного вхождения. Это требует дополнительно $O(mk)$ времени, что доказывает оценку времени выполнения, приведенную в табл. 5.2.

Алгоритм 5.3.2. (Наивный Алгоритм)

▷ *Наивное использование массива суффиксов для локализации* $\boldsymbol{u}$ *в* $\boldsymbol{x}$
$j \leftarrow 0$; $L \leftarrow 1$; $R \leftarrow n$
repeat
  $M \leftarrow \lceil (R+L)/2 \rceil$
  if $\boldsymbol{u} = \boldsymbol{x}[\boldsymbol{\sigma}[M]..\boldsymbol{\sigma}[M]+m-1]$ **then**
    $j \leftarrow \boldsymbol{\sigma}[M]$
  else if $\boldsymbol{u} > \boldsymbol{x}[\boldsymbol{\sigma}[M]..\boldsymbol{\sigma}[M]+m-1]$ **then**
    $L \leftarrow M$
  else
    $R \leftarrow M$
until $L = M$ или $j \neq 0$

Теперь представим простой и сложный алгоритмы. Простой алгоритм можно рассматривать как усовершенствование наивного алгоритма, где очевидна необходимость повысить эффективность обоих операторов **if**. Один способ такого усовершенствования показан в упражнении 5.3.8. Основой более значительных изменений может служить ранее описанное свойство массивов суффиксов: если суффиксы $\boldsymbol{\sigma}[L]$ и $\boldsymbol{\sigma}[R]$ имеют одинаковый префикс, тогда такой же префикс имеют суффиксы $\boldsymbol{\sigma}[K]$, где $K = L, L+1, \ldots, R$. Назовем это свойство массивов суффиксов ***кластерным***. Используя это свойство мы с неизбежностью возвращаемся к еще одному понятию, а именно к понятию *наибольшего общего префикса* (LCP), введенному в разделе 5.2.1. Напомним, что $\mathrm{lcp} = |\,\mathrm{LCP}\,|$ обозначает длину такого префикса. Для модификации алгоритма 5.3.2 введем величины $P_L = \mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[L])$, $P_R = \mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[R])$ и $P = \min\{P_L, P_R\}$. В соответствии с кластерным свойством сравнение $\boldsymbol{u}$ с префиксами $\boldsymbol{\sigma}[K]$ необходимо выполнять только для $\boldsymbol{u}[P+1..m]$, поскольку префикс $\boldsymbol{u}[1..P]$ одинаковый у всех этих суффиксов. Приведем листинг простого алгоритма.

Алгоритм 5.3.3. (Простой Алгоритм)

▷ *Простое использование массива суффиксов для локализации* $\boldsymbol{u}$ *в* $\boldsymbol{x}$
$j \leftarrow 0$; $L \leftarrow 1$; $R \leftarrow n$
$P_L \leftarrow 0$; $P_R \leftarrow 0$
repeat
  $P \leftarrow \min\{P_L, P_R\}$.
  $M \leftarrow \lceil (R+L)/2 \rceil$
  ▷ *Вычисление* $\mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[M])$
  $P_M \leftarrow P + \mathrm{lcp}\,(\boldsymbol{u}[P+1..m], \boldsymbol{x}[\boldsymbol{\sigma}[M]+P..n])$
  if $P_M = m$ **then**
    $j \leftarrow \boldsymbol{\sigma}[M]$
  else if $\boldsymbol{u}[P_M+1] > \boldsymbol{x}[\boldsymbol{\sigma}[M]+P_M]$ **then**

$L \leftarrow M$; $P_L \leftarrow P_M$
else
$R \leftarrow M$; $P_R \leftarrow P_M$
until $L = M$ или $j \neq 0$

Для выполнения каждого оператора этого алгоритма, за исключением вычисления lcp, требуется константное время, поэтому справедлива верхняя оценка временно́й сложности алгоритма, приведенная в табл. 5.2. Нетрудно привести примеры строк $\boldsymbol{u}$ и $\boldsymbol{x}$, для которых алгоритм выполняется за время, равное верхней оценке. Поэтому простой алгоритм не является улучшением наивного алгоритма в теоретическом плане. Однако на практике, поскольку P монотонно не убывает, операция сравнения букв в среднем требует значительно меньше времени, чем в наивном алгоритме. По-видимому, простой алгоритм является наиболее быстрым среди всех известных последовательных алгоритмов нахождения вхождений заданной подстроки $\boldsymbol{u}$ в заданную строку $\boldsymbol{x}$.

Теперь рассмотрим сложный алгоритм. Ключевая идея этого алгоритма — заблаговременно определить все возможные интервалы поиска $[L, R]$ ($L < R$), которые могут возникнуть в процессе бинарного поиска, а также предварительно вычислить $\mathrm{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[R])$ для этих интервалов. Как мы увидим далее, общее количество таких интервалов равно $2n - 3$. Все предварительно вычисленные значения lcp поместим в новый массив (или строку) $\boldsymbol{\pi} = \boldsymbol{\pi}_{\boldsymbol{x}}$, длина которого не превысит $3n - 8$ при $n \geqslant 5$. Мы также покажем, что массив $\boldsymbol{\pi}$ можно получить в процессе первоначального вычисления массива суффиксов $\boldsymbol{\sigma}$ на основе дерева суффиксов $T_{\boldsymbol{x}}$. Для этого дополнительно потребуются время и объем памяти, линейные относительно $|\boldsymbol{x}|$. После создания массива $\boldsymbol{\pi}$ вычисление величины $\mathrm{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[R])$ можно будет выполнить за константное время просто путем выбора соответствующей позиции в массиве $\boldsymbol{\pi}_{\boldsymbol{x}}$. В этом случае общее время бинарного поиска можно уменьшить до величины порядка $O(m + \log n)$.

Из всего, что сказано выше, наиболее просто доказать, что количество интервалов поиска $[L, R]$ действительно равно $2n - 3$. Для примера на рис. 5.18 показано бинарное дерево для $n = 7$, где каждому из 11 узлов соответствует свой интервал поиска. Общий случай построения бинарного дерева поиска и соответствующих интервалов предложено рассмотреть в упражнении 5.3.12. На рис. 5.18 номера, расположенные внутри узлов дерева, представляют позиции в массиве $\boldsymbol{\pi}$. Эти номера определяются согласно "правилу кучи", поэтому интервал, значение lcp которого хранится в $\boldsymbol{\pi}[i]$, разбивается на два подынтервала, значения lcp которых хранятся в $\boldsymbol{\pi}[2i]$ и $\boldsymbol{\pi}[2i + 1]$. Читателю не составит труда проверить, что при такой схеме хранения максимальная длина массива $\boldsymbol{\pi}$ при $n \geqslant 5$ не превышает $3n - 8$.

Далее, предполагая, что массив $\boldsymbol{\pi}$ уже вычислен, опишем выполнение сложного алгоритма, основное внимание уделив шагу обработки интервала $[L, R]$. Пред-

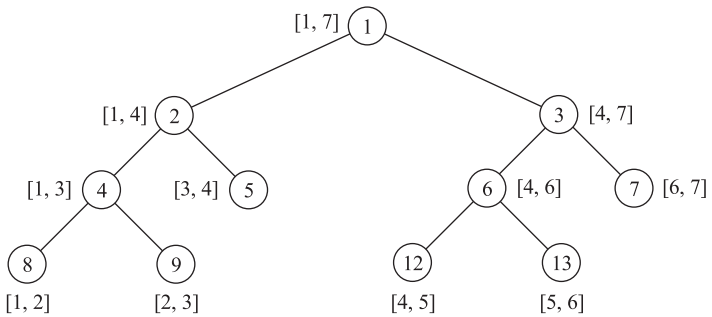


Рис. 5.18. Бинарное дерево интервалов $[L, R]$

положим, как и в простом алгоритме, что доступны значения P_L и P_R и каким-либо способом вычислено M. Также предположим, что вычислено значение i, такое, что

$$P_{LR} \equiv \mathrm{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[R]) = \boldsymbol{\pi}[i].$$

Для выполнения первого шага алгоритма должны быть известны все значения (L, R, P_L, P_R, i). Далее мы покажем, как эти величины вычисляются на последующих шагах алгоритма за время, пропорциональное величине изменения $\max\{P_L, P_R\}$, плюс константа.

Сначала рассмотрим случай $P_L = P_R$. Можно вычислить P_{M} таким же способом, как и в простом алгоритме, используя равенство $P = P_L$. Вычисление значения lcp требует $P_{\mathrm{M}} - P_L + 1$ сравнений букв и выполняется, как показано ранее, за константное время.

Теперь пусть $P_L \neq P_R$. Для определенности положим $P_L > P_R$. (Случай $P_L < P_R$ является симметричным рассматриваемому сейчас.) Возможны три ситуации, в зависимости от значения $P_{LR} = \boldsymbol{\pi}[2i]$.

1. $P_{LM} > P_L$. Для каждого $K \in L..M$ $\boldsymbol{\sigma}[K]$ имеет префикс длиной $P_{LM} > P_L = \mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[L])$. Поэтому, согласно кластерному свойству, $\boldsymbol{u}$ не может входить в интервал $[L, R]$, но, возможно, $\boldsymbol{u}$ входит в интервал $[\mathrm{M}, R]$. Однако проверять последнюю возможность нет необходимости: поскольку уже $\boldsymbol{\sigma}[M]$ имеет префикс длиной P_L, подстановки $L \leftarrow M$, $i \leftarrow 2i + 1$ позволяют определить все параметры, необходимые для следующего шага, без изменения величины $\max\{P_L, P_R\}$.

2. $P_{LM} < P_L$. В этом случае выполняется неравенство $\mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[L]) > \mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[M])$. Предположим, что $\boldsymbol{u}$ входит в строку $\boldsymbol{x}$ в некоторой позиции K, принадлежащей интервалу $[M, R]$. Тогда $\boldsymbol{\sigma}[K]$ имеет префикс длиной P_L (как и $\boldsymbol{\sigma}[L]$), но в промежуточных позициях $M \in L..K$ $\boldsymbol{\sigma}[M]$, согласно кластерному свойству, не имеет префиксов длиной P_L. Поэтому $\boldsymbol{u}$ не может

входить в строку $\boldsymbol{x}$ в позициях из интервала $[M, R]$, а если все же входит в строку $\boldsymbol{x}$, то в позициях из интервала $[L, M]$. Тогда подстановки $R \leftarrow M$, $P_R \leftarrow P_{LM}$, $i \leftarrow 2i$ позволяют определить все параметры, необходимые для следующего шага, и опять без изменения величины $\max\{P_L, P_R\}$.

3. $P_{LM} = P_L$. В этом случае $\mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[L]) = \mathrm{lcp}(\boldsymbol{u}, \boldsymbol{\sigma}[M])$. Снова, используя равенство $P = P_L$, вычисляем P_{M} тем же способом, что и в простом алгоритме. Затем вычисляем (L, P_L, i) или (R, P_R, i) за константное время.

Теперь покажем, что на каждом шаге сложного алгоритма пять параметров (L, R, P_L, P_R, i) вычисляются за время, пропорциональное величине изменения $\max\{P_L, P_R\}$, плюс константа. Поскольку величина $\max\{P_L, P_R\}$ монотонно не возрастает, а сам интервал $[L, R]$ монотонно сходится к конечному интервалу $[L, L+1]$, то алгоритм определяет вхождение $\boldsymbol{u}$ (если оно существет) либо в некоторой позиции M, либо в конечном интервале $[L, L+1]$. Тогда, используя немного модифицированное условие завершения поиска для конечного интервала, алгоритм найдет вхождение подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ за время $O(m + \log n)$, как указано в табл. 5.2. Мы оставим для упражнения 5.3.16 представление сложного алгоритма в алгоритмической форме. В упражнении 5.3.17 предложено показать, как с помощью массива $\boldsymbol{\pi}$ можно найти все k вхождений подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$ за дополнительное время $\Theta(k)$.

Нам осталось показать, как вычислить массив $\boldsymbol{\pi}$ в процессе вычисления массива $\boldsymbol{\sigma}$. Для вычисления массива $\boldsymbol{\pi}$ применяется следующий двухшаговый процесс.

1. При лексикографическом обходе в глубину дерева $T_{\boldsymbol{x}}$ путь от суффикса $\boldsymbol{\sigma}[i]$ до суффикса $\boldsymbol{\sigma}[i+1]$ проходит через узел $\boldsymbol{v}$, который является корнем поддерева минимального размера, содержащего как $\boldsymbol{\sigma}[i]$, так и $\boldsymbol{\sigma}[i+1]$. Как отмечалось в разделе 5.2.1, $\mathrm{lcp}(\boldsymbol{\sigma}[i], \boldsymbol{\sigma}[i+1]) = |\boldsymbol{v}|$. Тогда для всех $i = 1, 2, \ldots, n-1$ значения lcp для интервала $[i, i+1]$ можно вычислить при обходе в глубину дерева $T_{\boldsymbol{x}}$. В упражнении 5.3.14 показано, как вычислить позицию j в массиве $\boldsymbol{\pi}$, соответствующую интервалу $[i, i+1]$. Значения lcp для этого интервала можно сохранить путем присвоения $\boldsymbol{\pi}[j] \leftarrow |\boldsymbol{v}|$.

2. Как показано в упражнении 5.3.10, для любого интервала $[L, R]$, $R \geqslant L$, выполняется равенство

$$\mathrm{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[R]) = \min\{\mathrm{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[M]), \mathrm{lcp}(\boldsymbol{\sigma}[M], \boldsymbol{\sigma}[R])\} \qquad (5.10)$$

для всех $M \in L..R$. Создавая дерево интервалов от листьев до корня (т.е. в порядке заполнения массива $\boldsymbol{\pi}$), это равенство позволяет вычислить все элементы массива $\boldsymbol{\pi}$ за время $O(n)$.

Этим заканчивается наше исследование массивов суффиксов, возможно, наиболее привлекательной суффиксной структуры. Они вычисляются с относительно небольшими временны́ми и пространственными затратами и не зависят напрямую

от размера алфавита. Но, конечно, опосредованно массивы суффиксов зависят от размера алфавита, поскольку вычисляются на основе деревьев суффиксов. Однако, как упоминалось ранее, можно исключить деревья суффиксов из процесса вычисления массивов за счет некоторого увеличения времени вычислений [172]. Фактически, большинство современных алгоритмов непосредственного вычисления массивов суффиксов [151, 31, 200] появилось в результате конкуренции по критерию времени выполнения с алгоритмом Мак-Крейта построения дерева суффиксов [200].

В разделе 13.2 мы используем массивы суффиксов для построения эффективных решений некоторых задач вычисления повторяющихся подстрок строки $\boldsymbol{x}$.

Упражнения 5.3

1. Докажите, что отношение П-эквивалентности, определенное на подстроках строки $\boldsymbol{x}$, является отношением эквивалентности.
2. Если $\boldsymbol{v}$ — подстрока строки $\boldsymbol{x}$, то обязано ли поддерево $T'_{\boldsymbol{v}}$ с корнем в узле $\boldsymbol{v}$ быть деревом суффиксов?
3. Пусть подстроки $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ П-эквивалентны и выполняется неравенство $|\boldsymbol{u}_1| < |\boldsymbol{u}_2|$. Покажите, что j будет меткой листа в поддереве $T'_{\boldsymbol{u}_1}$ только в том случае, когда в поддереве $T'_{\boldsymbol{u}_2}$ есть лист с меткой $j + |\boldsymbol{u}_2| - |\boldsymbol{u}_1|$.
4. Пусть T — дерево, узлы которого соответствуют непустым множествам положительных целых чисел. Предположим, что корню дерева соответствует множество $\{1, 2, \ldots, n\}$, и пусть сынам каждого узла соответствуют непересекающиеся множества, объединение которых входит в множество родителя. Покажите, что такое дерево может иметь не более $2n - 1$ узлов.
5. Основываясь на исходном определении ОАГС, докажите, что для любого узла только одна входящая в него дуга имеет ширину, превышающую 1.
6. На этапе предварительной обработки алгоритма 5.3.1 для выполнения вычислений совершался обход дерева $T_{\boldsymbol{x}}$ в определенном порядке. Объясните, как организовать обход дерева так, чтобы исключить зависимость алгоритма от размера алфавита, и как при этом выполнить вычисления за время, пропорциональное только $|\boldsymbol{x}|$.
7. В тексте сделано утверждение, что поиск в глубину по дереву $T_{\boldsymbol{x}}$ для вычисления $\boldsymbol{\sigma}_{\boldsymbol{x}}$ можно "в определенных случаях" выполнить за время, пропорциональное только n. Приведите эти "определенные случаи" и опишите соответствующий процесс поиска.
8. Наивный алгоритм выполняет повторные проверки позиций подстроки $\boldsymbol{u}$ в случае, если она имеет общий префикс с листом $\boldsymbol{\sigma}[M]$. Измените этот алгоритм таким образом, чтобы исключить повторные проверки.

9. Объясните пессимистические замечания автора об алгоритме 5.3.2 и покажите изменения, которые необходимо сделать в алгоритме для усовершенствования бинарного поиска.

10. Докажите, что для произвольных позиций L и R в строке $\boldsymbol{x}$ и произвольной строке $\boldsymbol{u}$ справедливо равенство

$$\min\{|\boldsymbol{u}|, \operatorname{lcp}(\boldsymbol{\sigma}[L], \boldsymbol{\sigma}[R])\} = \min\{P_L, P_R\}.$$

Приведите примеры классов строк $\boldsymbol{x}$ и $\boldsymbol{u}$, для которых простой алгоритм будет выполняться за время $\Omega(m \log n)$.

11. Пусть алгоритм 5.3.2 выполняется над массивом длиной n. Покажите, что

а) для $n \geqslant 2$ количество возможных интервалов $[L, R]$ $(L < R)$ равно в точности $2n - 3$;

б) для $n \geqslant 5$ максимальный размер массива $\boldsymbol{\pi}$, если он вычисляется так, как описано в тексте, не превышает $3n - 8$.

Охарактеризуйте те значения n, для которых достигается верхняя оценка утверждения *б*).

12. Вычислите массив $\boldsymbol{\pi}$ для строки $\boldsymbol{g} = abaabaab$.

13. Запишем число n как $n = 2^k + h$ для $k \geqslant 1$ и $h \in 0..2^k - 1$. Получите формулы для вычисления размера массива $\boldsymbol{\pi}$ как функции от k и h. Затем покажите, как найти в массиве $\boldsymbol{\pi}$ позицию, соответствующую интервалу $[L, L + 1]$, $L \in 1..n - 1$.

14. Вместо массива $\boldsymbol{\pi}$ с $3n-8$ элементами можно использовать бинарное дерево с $2n - 3$ узлами. Обсудите эту возможность и объясните, почему первый вариант (массив $\boldsymbol{\pi}$) предпочтительнее.

15. Основываясь на описании в тексте сложного алгоритма, запишите его в алгоритмической форме.

16. Пусть с помощью сложного алгоритма найдено первое вхождение подстроки $\boldsymbol{u}$ в строку $\boldsymbol{x}$. Покажите, что все другие вхождения подстроки $\boldsymbol{u}$ можно локализовать за время, пропорциональное количеству этих вхождений.

Совет. Используйте кластерное свойство и тот факт, что массив $\boldsymbol{\pi}$ содержит позиции, соответствующие всем интервалам $[L, L + 1]$, $L = 1, 2, \ldots, n - 1$.

17. Докажите равенство (5.10).

18. Запишите второй шаг алгоритма вычисления массива $\boldsymbol{\pi}$ и затем покажите, что он выполняется за время $\Theta(n)$.

19. В упражнении 5.2.18 предлагалось оценить необходимый объем памяти для деревьев суффиксов для строк длиной $n = 100$ Кбайт, 1 Мбайт и 5 Мбайт

в случае $\alpha = 4$. На основе табл. 5.2 получите аналогичные оценки для массивов суффиксов и выведите коэффициент отношения этих оценок к объему памяти, необходимому для хранения только строки $\boldsymbol{x}$. Какие заключения можно сделать на основании этого коэффициента? Какой эффект произведет на дерево суффиксов и массив суффиксов увеличение или удвоение размера алфавита?

ГЛАВА 6

Декомпозиция строковых последовательностей

> "Всегда" и "никогда" — вот два слова, которые вы всегда должны помнить, чтобы никогда их не произносить.
>
> — У. А. Л. Джонсон (1906–1965)

В разделе 1.4 мы изучили специальные строковые последовательности, которые назвали линдонскими словами, и некоторые их свойства. Там же показали, что линдонские слова могут стать основой внутреннего паттерна, который называется линдонской декомпозицией, причем такой паттерн единственный для любой строковой последовательности. Однако в главе 1 мы не показали, как для данной строки $\boldsymbol{x}$ эффективно вычислить линдонскую декомпозицию $\boldsymbol{w}_1^{q_1}\boldsymbol{w}_2^{q_2}\cdots\boldsymbol{w}_k^{q_k}$. Поэтому в разделе 6.1 опишем два разработанных Дювалем (Duval, [80]) элегантных алгоритма, которые выполняются за линейное время. В следующем разделе покажем два применения этих алгоритмов. Одно из этих применений уже описано в разделе 1.4 — линдонская декомпозиция строки $\boldsymbol{x}$ непосредственно ведет к канонической форме соответственной строковой петли $C(\boldsymbol{x})$; таким способом получаем эффективное решение задачи, которая часто возникает в вычислительной геометрии.

В разделе 6.3 описан другой тип декомпозиции, полностью отличной от линдонской, которая называется s-факторизацией [156] и которая также выполняется за линейное время. Этот тип декомпозиции первоначально разрабатывался как

алгоритм сжатия данных [235], однако s-факторизация имеет также важные приложения для вычисления кратных подстрок строки $\boldsymbol{x}$ (см. раздел 12.2).

В этой главе, как и в предыдущей, предполагается, что алфавит A упорядочен. Для линдонской декомпозиции мы покажем, что все алгоритмы, ее вычисляющие, не зависят от размера алфавита — вычислить линдонскую декомпозицию на неограниченных алфавитах не труднее, чем на бинарном алфавите. Однако таким замечательным свойством не обладает s-факторизация, для эффективного вычисления которой используется дерево суффиксов.

6.1 Линдонская декомпозиция: алгоритм Дюваля

Сначала рассмотрим первый алгоритм Дюваля, который за линейное время обрабатывает строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$ по одной букве слева направо, используя при этом некоторое количество возвратов к предыдущим позициям строки $\boldsymbol{x}$. Затем опишем второй алгоритм Дюваля, который является модификацией первого алгоритма и уменьшает количество обратных просмотров за счет использования дополнительной памяти объемом $\Theta(n)$. Эти алгоритмы основаны на интересной взаимосвязи двух внешне совершенно различных понятий: нормальной формы и линдонских слов.

Перед описанием алгоритмов Дюваля введем следующие обозначения.

- Пусть L обозначает множество всех линдонских слов.
- В честь Чена, Фокса и Линдона (Chen, Fox, Lyndon, [48]) линдонскую декомпозицию строки $\boldsymbol{x}$ будем обозначать как $\mathrm{CFL}(\boldsymbol{x})$.
- Через $\mathrm{NF}(\boldsymbol{x})$ обозначим нормальную форму $\boldsymbol{u}^r\boldsymbol{u}'$ строки $\boldsymbol{x}$, где $\boldsymbol{u}$ — образующая подстрока строки $\boldsymbol{x}$ минимальной длины, $\boldsymbol{u}'$ — собственный префикс подстроки $\boldsymbol{u}$ и $r \geqslant 1$. В этом случае будем говорить, что подстрока $\boldsymbol{u}$ ***порождает*** строку $\boldsymbol{x}$. Будем обозначать как L^* множество всех строк, порожденных линдонскими словами.

Напомним, что лемма 1.4.5 утверждает, что любое линдонское слово примитивно. Поскольку линдонское слово порождает само себя при $r = 1$ и $\boldsymbol{u}' = \varepsilon$, то, очевидно, $L \subset L^*$. Но, конечно, не каждая строка с примитивной порождающей подстрокой принадлежит множеству L^*. Например, $\mathrm{NF}(baaba) = (baa)(ba)$ с примитивной порождающей подстрокой baa, но поскольку $baa \notin L$, то и $baaba \notin L^*$. С другой стороны, из теоремы 1.4.2 следует, что любая строка, не являющаяся кратной, имеет единственный сдвиг, который будет линдонским словом. Так $aab = R_1(baa) \in L$, и поэтому, например, $(aab)^r aa \in L^*$ для любого $r \geqslant 1$.

На каждом шаге первого алгоритма Дюваля исходная строка $\boldsymbol{x}$ рассматривается как конкатенация (сцепление) трех факторов (подстрок) $\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3$, где

- подстрока $\boldsymbol{x}_1$ является префиксом строки $\boldsymbol{x}$, для которой $\mathrm{CFL}(\boldsymbol{x}_1)$ уже вычислена;
- подстрока $\boldsymbol{x}_2$ уже просмотрена алгоритмом, но CFL для нее еще не вычислена;
- подстрока $\boldsymbol{x}_3$ является суффиксом строки $\boldsymbol{x}$, которая еще не обрабатывалась алгоритмом.

На каждом шаге алгоритма подстрока $\boldsymbol{x}_1\boldsymbol{x}_2$ увеличивается справа на одну букву (пусть это будет буква μ), тогда как подстрока $\boldsymbol{x}_3$ уменьшается слева на эту же букву. В алгоритме на любом шаге должно выполняться условие $\boldsymbol{x}_2 \in \in L^*$. Тогда на следующем шаге это условие выполнится, если $\boldsymbol{x}_2\mu \in L^*$. Если же $\boldsymbol{x}_2\mu \notin L^*$, тогда $\boldsymbol{x}_2\mu$ разбивается на два фактора $\boldsymbol{x}'_1\boldsymbol{x}'_2$, таких, что можно вычислить $\mathrm{CFL}(\boldsymbol{x}'_1)$, а $\boldsymbol{x}'_2 \in L^*$. В последнем случае новым префиксом строки $\boldsymbol{x}$, для которого необходимо вычислить CFL, становится подстрока $\boldsymbol{x}_1\boldsymbol{x}'_1$.

Теперь рассмотрим детали первого алгоритма Дювала совместно с их обоснованием. Основой этих обоснований послужат следующие три леммы, которые, в свою очередь, основываются на леммах из раздела 1.4. Первая из этих лемм показывает, как можно классифицировать подстроки $\boldsymbol{x}_2\mu$.

Лемма 6.1.1. Пусть $\boldsymbol{x}_2 \in L^*$ и $\mathrm{NF}(\boldsymbol{x}_2) = \boldsymbol{w}^r\boldsymbol{w}'$ для некоторого $\boldsymbol{w} \in L$ и пусть $\boldsymbol{w} = \boldsymbol{w}'\lambda\boldsymbol{w}''$, где $\lambda \in A$. Тогда для $\mu \in A$

а) $\mu > \lambda \Rightarrow \boldsymbol{x}_2\mu \in L$;

б) $\mu = \lambda \Rightarrow \boldsymbol{x}_2\mu \in L^* - L$;

в) $\mu < \lambda \Rightarrow \boldsymbol{x}_2\mu \notin L^*$.

Доказательство. Чтобы доказать утверждение *а*), сначала покажем, что $\boldsymbol{w}'\mu \in L$. Если $\boldsymbol{w}' = \varepsilon$, то это выполняется по определению. Поэтому пусть $\boldsymbol{w}'$ является непустой строкой. Поскольку $\boldsymbol{w} \in L$, из леммы 1.4.6 следует, что

$$\boldsymbol{w} = \boldsymbol{w}'\lambda\boldsymbol{w}'' < \mathrm{suff}(\boldsymbol{w}')\lambda\boldsymbol{w}''$$

для любого непустого собственного суффикса $\mathrm{suff}(\boldsymbol{w}')\lambda$ подстроки $\boldsymbol{w}'\lambda$. Так как $\lambda\boldsymbol{w}'' < \mu$, то

$$\boldsymbol{w}'\lambda\boldsymbol{w}'' < \mathrm{suff}(\boldsymbol{w}')\lambda\boldsymbol{w}'' < suff(\boldsymbol{w}')\mu,$$

и поэтому $\boldsymbol{w}' < \mathrm{suff}(\boldsymbol{w}')\mu$. Следовательно, для любого непустого собственного суффикса $\mathrm{suff}(\boldsymbol{w}')\mu$ подстроки $\boldsymbol{w}'\mu$

$$\boldsymbol{w}'\mu < \mathrm{suff}(\boldsymbol{w}')\mu.$$

Тогда из леммы 1.4.6 вытекает, что $\boldsymbol{w}'\mu \in L$.

По индукции, основанной на лемме 1.4.7, доказываем утверждение *а*): поскольку $\boldsymbol{w} \in L$ и $\boldsymbol{w}'\mu \in L$ и при этом $\boldsymbol{w} < \boldsymbol{w}'\mu$, то отсюда следует, что $\boldsymbol{w}\boldsymbol{w}'\mu \in L$.

В общем случае, если $\boldsymbol{w} \in L$ и $\boldsymbol{w}^s\boldsymbol{w}'\mu \in L$ для некоторого неотрицательного целого s и при этом выполняется неравенство $\boldsymbol{w} < \boldsymbol{w}^s\boldsymbol{w}'\mu$, то отсюда вытекает, что $\boldsymbol{w}^{s+1}\boldsymbol{w}'\mu \in L$.

Для доказательства утверждения *б*) заметим, что подстрока $\boldsymbol{x}_2\mu = \boldsymbol{x}_2\lambda$ имеет непустую грань $\boldsymbol{w}'\lambda$. Поэтому, вследствие леммы 1.4.5, $\boldsymbol{x}_2\mu \notin L$. Однако подстрока $\boldsymbol{x}_2\mu = \boldsymbol{w}^r\boldsymbol{w}'\lambda$ порождена подстрокой $\boldsymbol{w}$, и поэтому $\boldsymbol{x}_2\mu \in L^*$.

Для доказательства утверждения в) имеем

$$\lambda > \mu \Rightarrow \boldsymbol{w}'\lambda > \boldsymbol{w}'\mu \Rightarrow \boldsymbol{x}_2\mu > \boldsymbol{w}'\mu,$$

поэтому строка $\boldsymbol{x}_2\mu$ лексикографически больше, чем ее собственный префикс. Тогда, вследствие леммы 1.4.6, $\boldsymbol{x}_2\mu \notin L$. Предположим, что строка $\boldsymbol{x}_2\mu$ порождена подстрокой $\overline{\boldsymbol{w}} \in L$. Поскольку $\boldsymbol{x}_2\mu \notin L$, подстрока $\overline{\boldsymbol{w}}$ должна быть собственным префиксом строки $\boldsymbol{x}_2\mu$ и, следовательно, префиксом $\boldsymbol{x}_2$. Но любой префикс строки $\boldsymbol{x}_2$ длиной больше длины подстроки $\boldsymbol{w}$ не может быть примитивной строкой. Поэтому на основании леммы 1.4.5 заключаем, что $|\overline{\boldsymbol{w}}| \leqslant |\boldsymbol{w}|$. Но строгого неравенства $|\overline{\boldsymbol{w}}| < |\boldsymbol{w}|$ быть не может, поскольку тогда $\boldsymbol{x}_2$ должна порождаться $\overline{\boldsymbol{w}}$, что противоречит требованию нормальной формы о том, что $\boldsymbol{w}$ является порождающей $\boldsymbol{x}_2$ подстрокой минимальной длины. Поэтому остается только возможность, когда $\overline{\boldsymbol{w}} = \boldsymbol{w}$. Но эта ситуация также не имеет места, поскольку $\mu \neq \lambda$ и $\boldsymbol{w}'\mu$ не является префиксом $\boldsymbol{w}$. Отсюда заключаем, что порождающая $\boldsymbol{x}_2\mu$ подстрока не принадлежит множеству L, и поэтому $\boldsymbol{x}_2\mu \notin L^*$. ■

Чтобы пояснить доказанную лемму, рассмотрим пример, где $\boldsymbol{x}_2 = (aabb)^2aab$ и $\boldsymbol{w} = aabb \in L$, $r = 2$, $\boldsymbol{w}' = aab$, $\lambda = b$. Если следующая буква $\mu = c > \lambda$, тогда строка $\boldsymbol{x}_2c$, очевидно, будет лексикографически меньше любого собственного сдвига и поэтому $\boldsymbol{x}_2c \in L$, как предсказывает утверждение *а*) леммы. Для случая $\mu = b = \lambda$ строка $\boldsymbol{x}_2b = (aabb)^3$ принадлежит множеству $L^* - L$, что соответствует утверждению *б*) леммы. Если же $\mu = a < \lambda$, тогда $\mathrm{NF}(\boldsymbol{x}_2a) = (aabbaabbaab)a$. Поскольку порождающая подстрока $(aabb)^2aab$ не является примитивной, то $\boldsymbol{x}_2a \notin L^*$, как предсказывает утверждение *в*) леммы.

Следующие две леммы дают основу для построения линдонской декомпозиции.

Лемма 6.1.2. Пусть $\boldsymbol{x} = \boldsymbol{w}\boldsymbol{x}'$, где $\boldsymbol{w}$ — наибольший префикс строки $\boldsymbol{x}$, который является линдонским словом. Тогда

$$\mathrm{CFL}(\boldsymbol{x}) = \boldsymbol{w}\,\mathrm{CFL}(\boldsymbol{x}').$$

Доказательство. Пусть $\mathrm{CFL}(\boldsymbol{x}') = \boldsymbol{w}'_1\boldsymbol{w}'_2\ldots\boldsymbol{w}'_k$. Поскольку $\boldsymbol{w}$ — наибольший префикс, то $\boldsymbol{w}\boldsymbol{w}' \notin L$. Поэтому, вследствие леммы 1.4.7, $\boldsymbol{w} \geqslant \boldsymbol{w}'_1$. Тогда $\boldsymbol{w}\boldsymbol{w}'_1\boldsymbol{w}'_2\ldots\boldsymbol{w}'_k$ по определению является линдонской декомпозицией строки $\boldsymbol{x}$, причем единственной (теорема 1.4.9). Следовательно, $\mathrm{CFL}(\boldsymbol{x}) = \boldsymbol{w}\,\mathrm{CFL}(\boldsymbol{x}')$. ■

Следующая лемма показывает, как вычисляются компоненты декомпозиции $\mathrm{CFL}(\boldsymbol{x})$ в случае, когда очередная буква μ "мала", т.е. когда выполняется утверждение *в*) леммы 6.1.1.

Лемма 6.1.3. Пусть $\boldsymbol{x} = \boldsymbol{x}_2\mu\boldsymbol{x}'$, где $\boldsymbol{x}_2 \in L^*$ и $\mu \in A$. Пусть также $\mathrm{NF}(\boldsymbol{x}_2) = \boldsymbol{w}^r\boldsymbol{w}'$, где $\boldsymbol{w} = \boldsymbol{w}'\lambda\boldsymbol{w}''$ и $\lambda \in A$. Если $\mu < \lambda$, тогда

$$\mathrm{CFL}(\boldsymbol{x}) = \boldsymbol{w}^r\,\mathrm{CFL}(\boldsymbol{x}'\mu\boldsymbol{x}').$$

Доказательство. Отметим, что поскольку $\boldsymbol{x}_2 \in L^*$, то $\boldsymbol{w} \in L$. Покажем, что $\boldsymbol{w}$ — наибольший префикс строки $\boldsymbol{x}$, являющийся линдонским словом. Предположим противное, что существует префикс $\overline{\boldsymbol{w}} \in L$ строки $\boldsymbol{x}$, такой, что $|\overline{\boldsymbol{w}}| > |\boldsymbol{w}|$. Но если $|\overline{\boldsymbol{w}}| \leqslant |\boldsymbol{x}_2|$, тогда $\overline{\boldsymbol{w}}$ имеет непустую грань и, следовательно, не является примитивной строкой. Но, вследствие леммы 1.4.5, $\overline{\boldsymbol{w}}$ не может быть линдонским словом. Поэтому возможен только случай, когда $|\overline{\boldsymbol{w}}| > |\boldsymbol{x}_2|$. Тогда $\overline{\boldsymbol{w}} = \boldsymbol{x}_2\lambda\overline{\boldsymbol{x}}$ для некоторого, возможно пустого, префикса $\overline{\boldsymbol{x}}$ строки $\boldsymbol{x}'$. Следовательно, $\overline{\boldsymbol{w}}$ имеет суффикс $\boldsymbol{w}'\mu\overline{\boldsymbol{x}} < \boldsymbol{w}'\lambda$, где $\boldsymbol{w}'\lambda$ — префикс подстроки $\overline{\boldsymbol{w}}$. Поэтому

$$\overline{\boldsymbol{w}} \geqslant \boldsymbol{w}'\lambda > \boldsymbol{w}'\mu\overline{\boldsymbol{x}} = \mathrm{suff}(\boldsymbol{w}),$$

что противоречит лемме 1.4.6. Отсюда заключаем, что не существует префикса $\overline{\boldsymbol{w}} \in L$ строки $\boldsymbol{x}$, такого, что $|\overline{\boldsymbol{w}}| > |\boldsymbol{w}|$.

Теперь можно применить лемму 6.1.2 к строке $\boldsymbol{x} = \boldsymbol{w}^r\boldsymbol{w}'\mu\boldsymbol{x}'$, откуда получаем, что

$$\mathrm{CFL}(\boldsymbol{x}) = \boldsymbol{w}\,\mathrm{CFL}(\boldsymbol{w}^{r-1}\boldsymbol{w}'\mu\boldsymbol{x}'),$$

где или $r - 1 = 0$ или $\boldsymbol{w}$ является наибольшим префиксом строки $\boldsymbol{w}^{r-1}\boldsymbol{w}'\mu\boldsymbol{x}'$ и линдонским словом. Это доказывает утверждение леммы. ■

Возвращаясь к примеру $\boldsymbol{x}\mu = (aabb)^2(aab)a$, мы видим, что в случае $\mu = a$

$$\mathrm{CFL}(\boldsymbol{x}\mu) = (aabb)^2\,\mathrm{CFL}(aaba).$$

Отметим, что лемму 6.1.3 можно применить и в случае $\mu = \$$, где $\$$ — символ окончания строки, который лексикографически меньше любой буквы алфавита. Этот факт используется в алгоритме Дюваля.

Теперь можно описать работу алгоритма Дюваля: на каждом шаге этого алгоритма в процессе просмотра букв слева направо строка $\boldsymbol{x}$ представляется в виде $\boldsymbol{x} = \boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3$, где подстрока $\boldsymbol{x}_1$ уже представима в форме линдонской декомпозиции, подстрока $\boldsymbol{x}_2 \in L^*$ представима в нормальной форме, а начальный символ μ подстроки $\boldsymbol{x}_3$ используется для управления нормальной формой $\mathrm{NF}(\boldsymbol{x}_2)$ и для определения того, когда префикс $\boldsymbol{w}^r$ подстроки $\boldsymbol{x}_2$ присоединится к $\mathrm{CFL}(\boldsymbol{x}_1)$. Подробно алгоритм показан в приведенном ниже листинге, в котором переменные h, i, j указывают позиции правых окончаний $\boldsymbol{x}_1$, λ и μ соответственно. Отметим, что присвоение $i \leftarrow h + 1$ соответствует случаю *а*) леммы 6.1.1, а присвоение $i \leftarrow i + 1$ — случаю *б*) этой леммы.

Алгоритм 6.1.1. (Первый алгоритм Дюваля)

▷ *Для входной строки* $\boldsymbol{x}\$$ *выводятся упорядоченные позиции*
▷ *концов всех линдонских слов в* $\mathrm{CFL}(\boldsymbol{x})$
$h \leftarrow 0 - h = |\boldsymbol{x}_1|$, текущая длина уже вычисленной части CFL
while $h < n$ **do** ▷ *цикл выполняется до тех пор, пока* $|\mathrm{CFL}(\boldsymbol{x}_1)| < n$
  $i \leftarrow h + 1 - \boldsymbol{x}[i] = \lambda$
  $j \leftarrow h + 2 - \boldsymbol{x}[j] = \mu$
  ▷ *следующий цикл выполняется до тех пор, пока* $\boldsymbol{x}_2 \in L^*$
  while $\boldsymbol{x}[j] \geqslant \boldsymbol{x}[i]$ **do** ▷ *выполняется проверка* $\mu \geqslant \lambda$
    if $\boldsymbol{x}[j] > \boldsymbol{x}[i]$ **then**
      $i \leftarrow h + 1$
    else
      $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  ▷ *для* $\mu < \lambda$ *(на основе леммы 6.1.3) этот цикл выводит все* r
  ▷ *вхождений текущей порождающей строки* $\boldsymbol{w}$
  ▷ *длиной* $j - i + 1$
repeat
  $h \leftarrow h + (j - i)$; **output** h
until $h \geqslant i$

Отметим, что в случае $\mu < \lambda$ алгоритм выполняет возврат назад, когда после вывода r копий линдонского слова $\boldsymbol{w}$ алгоритм возвращается к началу строки $\boldsymbol{w}'\mu$ с помощью операторов $i \leftarrow h + 1$ и $j \leftarrow h + 2$. Поскольку $\boldsymbol{w}'$ является префиксом $\boldsymbol{w}$, то для $\boldsymbol{w}'$ повторяются уже ранее выполненные действия всякий раз, когда $|\boldsymbol{w}'| > 1$. В примере $\boldsymbol{x}\mu = (aabb)^2(aab)a$, приведенном выше, строка $\boldsymbol{w}' = aab$ будет обрабатываться еще раз, хотя она уже обрабатывалась как префикс строки $\boldsymbol{w} = aabb$ в начале выполнения внешнего цикла **while**. Как мы покажем далее, второй алгоритм Дюваля исключает повторную обработку путем введения записей о периодичности строки $\boldsymbol{w}$.

Сформулируем основной результат данного раздела.

Теорема 6.1.4. Алгоритм 6.1.1 корректно вычисляет линдонскую декомпозицию за время порядка $\Theta(n)$ с использованием памяти фиксированного объема.

Доказательство. Корректность алгоритма вытекает из лемм 6.1.1 и 6.1.3, как показано в приведенных выше обсуждениях. Ограниченность используемой памяти следует из того факта, что для выполнения алгоритма не требуется никакой дополнительной памяти, кроме памяти для хранения переменных h, i и j.

Для анализа времени выполнения алгоритма заметим, что каждый вывод состоит из $r \geqslant 1$ копий строки $\boldsymbol{w} \in L$ и что повторные вычисления выполняются

только с собственным префиксом $\boldsymbol{w}'$ строки $\boldsymbol{w}$. Поскольку $\boldsymbol{w}$ является порождающей строкой для текущей подстроки $\boldsymbol{x}_2 = \boldsymbol{w}^r\boldsymbol{w}'$, то ее длина $p = |\boldsymbol{w}|$ является периодом подстроки $\boldsymbol{x}_2$. Поэтому требуется $rp + |\boldsymbol{w}'| - 1$ итераций внутреннего цикла **while**, чтобы присоединить r строк общей длиной rp к текущей (уже вычисленной) декомпозиции. В лучшем случае ($|\boldsymbol{w}'| = 0$) необходимо $rp - 1$ итераций внутреннего цикла **while** для вывода каждого множества, состоящего из rp букв. В самом худшем случае ($r = 1$ и $|\boldsymbol{w}'| = p - 1$) требуется $2p - 2$ таких итераций для вывода каждого множества из p букв. Поскольку вычисления на каждой итерации выполняются за константное время, общее время вывода результата составит величину порядка $O(n)$. Отсюда следует, что алгоритм 6.1.1 выполняется за время $\Theta(n)$. ■

Прежде чем перейти к рассмотрению второго алгоритма Дюваля, обсудим взаимосвязь между нормальной формой и линдонскими словами, которая проявилась в первом алгоритме Дюваля. Этот алгоритм можно рассматривать как алгоритм, который вычисляет нормальную форму максимальных элементов множества L^*, которые встречаются в данной строке $\boldsymbol{x}$. При этом кажется, что нормальная форма вычисляется без использования массива граней. Однако это не совсем так, как мы сейчас покажем.

Для переменных h, i и j из первого алгоритма Дюваля введем обозначения $i' = i - h$ и $j' = j - h$. Поскольку на любом шаге алгоритма $h = |\boldsymbol{x}_1|$, тогда $\lambda = \boldsymbol{x}[i] = \boldsymbol{x}_2[i']$ и $\mu = \boldsymbol{x}[j] = \boldsymbol{x}_2[j']$. Заметим, что на каждом шаге в начале внешнего цикла **while** для переменных i' и j' справедливо такое утверждение: i' — наибольшее целое, меньшее j', и при этом выполняется равенство

$$\boldsymbol{x}_2[1..i' - 1] = \boldsymbol{x}_2[j' - i' + 1..j' - 1]. \tag{6.1}$$

Другими словами, $\boldsymbol{x}_2[1..i' - 1]$ является наибольшей гранью подстроки $\boldsymbol{x}_2[1..j' - 1]$ или, в обозначениях раздела 1.3, $\boldsymbol{\beta}_2[1..j' - 1] = i' - 1$, где $\boldsymbol{\beta}_2$ — массив граней строки $\boldsymbol{x}_2$. Значения переменных i и j изменяются только во внутреннем цикле **while**, при этом переменная h там не изменяется. Поэтому равенство (6.1) останется неизменным во внутреннем цикле **while** в следующих ситуациях.

- Если $\boldsymbol{x}_2[j'] = \boldsymbol{x}_2[i']$. Тогда значения i' и j' возрастут на единицу, поэтому равенство (6.1) сохранит силу.
- Если $\boldsymbol{x}_2[j'] > \boldsymbol{x}_2[i']$. Тогда $i' \leftarrow 1$, а значение j' увеличится на единицу. Из утверждения *а*) леммы 6.1.1 вытекает, что в этом случае новая строка $\boldsymbol{x}_2$ является линдонским словом и поэтому примитивна. Следовательно, ее наибольшей гранью будет ε. Поэтому равенство (6.1) также будет иметь место.

Таким образом, во время выполнения первого алгоритма Дюваля равенство (6.1) будет выполняться всегда, что доказывает следующую лемму.

Лемма 6.1.5. Во внутреннем цикле **while** алгоритма 6.1.1 выполняется $\boldsymbol{\beta}_2[j'-1] = i'-1$, где $i' = i-h$, $j' = j-h$ и $\boldsymbol{\beta}_2$ — массив граней строки $\boldsymbol{x}_2[1..j']$. ■

Основываясь на этом результате, мы можем иметь массив $\boldsymbol{\beta}_2$ для каждого нового обрабатываемого значения j', содержащего записи соответствующих значений i', для которых уже проведено сравнение букв $\boldsymbol{x}_2[j']$ и $\boldsymbol{x}_2[i']$. На этом основан второй алгоритм Дюваля. Здесь для уменьшения вычислений сохраняется $n/2$ позиций массива $\boldsymbol{\beta}_2$ в несколько измененном виде, как показано ниже.

Алгоритм 6.1.2. (Второй Алгоритм Дюваля)

```
▷ Для входной строки x$ выводятся упорядоченные позиции
▷ концов всех линдонских слов в CFL(x)
h ← 0        ▷ h = |x_1|, текущая длина уже вычисленной части CFL
j ← 2; β'[2] ← 1        ▷ инициализация j и β'[j]
while h < n do        ▷ цикл выполняется до тех пор, пока |CFL(x_1)| < n
    i ← h + β'[j − h]        ▷ x[i] = λ
    ▷ следующий цикл выполняется до тех пор, пока x_2 ∈ L*
    while x[j] ⩾ x[i] do        ▷ выполняется проверка μ ⩾ λ
        if x[j] > x[i] then
            i ← h + 1
        else
            i ← i + 1
        j ← j + 1
        ▷ поскольку i и j изменены, то
        ▷ β'[j − h] = i − h ⇔ +β_2[j − h − 1] = i − h − 1
        if j − h ⩽ n/2 then
            β'[j − h] ← i − h
        ▷ для μ < λ (на основании леммы 6.1.3) этот цикл выводит все r
        ▷ вхождений текущей порождающей строки w
        ▷ длиной j − i + 1
    repeat
        h ← h + (j − i); output h
    until h ⩾ i
    ▷ если w' = ε, то μ перемещается на одну позицию вправо
    if h = j − 1 then
        j ← j + 1
```

Сравнивая первый и второй алгоритмы Дюваля, мы видим, что в процессе вычислений переменные h, i и j принимают одинаковые значения, да и структура циклов совпадает. Различия между этими алгоритмами связаны только с повтор-

ным вычислением i после того, как выведено множество r линдонских слов $\boldsymbol{w}$. Во втором алгоритме для этих вычислений используется массив $\boldsymbol{\beta'}$, изменения в который вносятся во внутреннем цикле **while**. Изменение массива $\boldsymbol{\beta'}$ эквивалентно созданию массива граней $\boldsymbol{\beta}_2$ для строки $\boldsymbol{x}_2$. Так, записи в массиве $\boldsymbol{\beta'}$ для любого $j'-1$ соответствуют граням $i'-1=\boldsymbol{\beta}_2[j'-1]=\boldsymbol{\beta'}[j']$. Поэтому значения массива $\boldsymbol{\beta'}$ можно использовать при обработке суффикса $\boldsymbol{w'}$ строки $\boldsymbol{x}_2$ после вывода $\boldsymbol{w}^r$. Осталось прояснить еще две небольшие неясности во втором алгоритме Дюваля.

1. Изменения в массив $\boldsymbol{\beta'}$ вносятся только в случае, когда $j-h\leqslant n/2$. Напомним, что $j-h=|\boldsymbol{x}_2|$, где $\boldsymbol{x}_2=\boldsymbol{w}^r\boldsymbol{w'}$. Элементы массива $\boldsymbol{\beta'}$ используются только для определения позиций в строке $\boldsymbol{w'}$ после вывода $\boldsymbol{w}^r$. Но

$$|\boldsymbol{w'}|<|\boldsymbol{x}_2|/2=(j-h)/2\leqslant(n+1)/2.$$

Поэтому могут потребоваться только элементы массива $\boldsymbol{\beta'}$, определяющие позиции, не превосходящие $n/2$.

2. Если $j=h+1$, тогда вывод $\boldsymbol{w}^r$ завершается и значение j увеличивается на единицу. В этом случае $\boldsymbol{w'}=\varepsilon$. Для следующего шага устанавливаются значения $j=h+2$, $i=h+\boldsymbol{\beta'}[2]=h+1$.

Очевидно, что второй алгоритм Дюваля имеет такое же асимптотическое время выполнения, как и первый алгоритм. Поэтому справедлива следующая теорема.

Теорема 6.1.6. Алгоритм 6.1.2 корректно вычисляет линдонскую декомпозицию за время порядка $\Theta(n)$ с использованием памяти объемом $\Theta(n)$. ■

Мы закончим изучение представленных алгоритмов вопросом о более точной оценке времени их выполнения в самом худшем случае. Второй алгоритм Дюваля уменьшает объем обработки непустых суффиксов $\boldsymbol{w'}$, но остается вопрос о том, насколько велико это уменьшение и "сколько оно стоит". Критерием, используемым для измерения эффективности строковых алгоритмов, часто выступает количество сравнений букв в строке. Существует обширная литература [52, 96, 97, 50], посвященная определению верхних и нижних границ количества буквенных сравнений, необходимых в самом худшем случае для нахождения заданного паттерна $\boldsymbol{u}$ в данной строке $\boldsymbol{x}$. Чтобы поддержать эту традицию, рассмотрим число буквенных сравнений в алгоритмах Дюваля.

Чтобы сделать такой анализ осмысленным, сначала надо определить, что мы понимаем под "сравнением букв". В этой книге мы будем использовать этот термин в смысле ***двоичного буквенного сравнения***, т.е. как сравнение букв, результатом которого могут быть два логических значения `TRUE` (Истина) или `FALSE` (Ложь). Например, требуется только одно сравнение букв μ и λ, чтобы определить, будет ли $\mu=\lambda$ или $\mu\geqslant\lambda$. Но чтобы различить три возможных случая $\mu>\lambda$, $\mu=\lambda$, $\mu<\lambda$, требуется уже два сравнения этих букв. Это определение, очевидно, согласуется с обычной двоичной логикой цифровых компьютеров.

Напомним, что в первом алгоритме Дюваля, как показано в доказательстве теоремы 6.1.4, для вывода p букв требуется не более $2p - 2$ итераций внутреннего цикла **while**. Чтобы "перевести" это число в количество буквенных сравнений, заметим, что на каждой итерации выполняется два сравнения: одно в операторе **while** и другое в операторе **if**. Поэтому во внутреннем цикле **while** всего выполняется $4p - 4$ буквенных сравнений. Кроме того, еще одно сравнение используется для проверки $\mu < \lambda$, завершающей цикл. Таким образом, в первом алгоритме Дюваля в самом худшем случае требуется не более $4n - 3$ буквенных сравнений для вывода всех линдонских слов строки $\boldsymbol{x}$.

Во втором алгоритме Дюваля во внутреннем цикле **while** выполняется не более одной итерации, требующей не более двух буквенных сравнений для каждого значения $j = 2, 3, \ldots, n$. Поэтому всего выполняется не более $2n - 2$ буквенных сравнений. Еще одно сравнение (во внутреннем операторе **while**) требуется для каждого вывода подстроки $\boldsymbol{w}'$ — это случай, когда префикс $\boldsymbol{w}'$ линдонского слова содержит одно или несколько линдонских слов, формирующих часть линдонской декомпозиции строки $\boldsymbol{x}$. Если $\boldsymbol{w}' = \varepsilon$, то значение j увеличивается на единицу. Поэтому такое дополнительное сравнение может потребоваться не более чем для половины всех значений j. Таким образом, второй алгоритм в самом худшем случае требует не более $2{,}5n - 2$ буквенных сравнений, что значительно меньше, чем в первом алгоритме.

Поэтому второй алгоритм лучше, чем первый. Но так ли это на самом деле?

Обсуждение 6.1.1. Конечно, и с практической, и с теоретической точки зрения особый интерес представляют наиболее точные границы времени выполнения строковых алгоритмов. Но на практике алгоритмы, требующие малые или минимальные количества буквенных сравнений, могут быть не самыми эффективными. Может случиться так, что дополнительные вычисления, позволяющие исключить или уменьшить количество буквенных сравнений, потребуют бóльших временны́х затрат, чем сами сравнения. Например, как отмечалось в разделе 4.1, простой алгоритм 2.2.1 для нахождения паттернов в среднем может выполняться быстрее, чем алгоритмы (такие, как алгоритм 7.1.1 КМП), где количество буквенных сравнений для самого худшего случая сведено к минимуму; подобно другим "примитивным созданиям", основной добродетелью простого алгоритма является его простота.

Подобные суждения применимы и для алгоритмов Дюваля. Хотя мы привели оценки количеств буквенных сравнений для одного и другого алгоритмов, остается все-таки не ясным вопрос, когда *на практике* второй алгоритм предпочтительнее первого? Приведем некоторые суждения по этому поводу.

- Второй алгоритм требует дополнительной памяти объемом $n/2$ и соответствующей дополнительной обработки каждой новой буквы μ и каждого множества одинаковых линдонских слов. Будут ли эти затраты давать преимущества времени выполнения в среднем?

- Как показано в упражнении 6.1.3, время вывода любого линдонского слова, найденного в строке $\boldsymbol{x}$, можно уменьшить до константного времени. Будут ли эти простые изменения в алгоритме более эффективными, чем попытка уменьшить количество буквенных сравнений?
- Верхние границы для количества буквенных сравнений в алгоритмах Дюваля, приведенные выше, не очень точны (см. упражнение 6.1.5). Даже без рассмотрения этих оценок "в среднем", может оказаться, что в самом худшем случае различия между этими алгоритмами не определяются полученными верхними оценками. Поэтому без более точного анализа опрометчиво делать заключения об эффективности алгоритмов Дюваля только на основании имеющихся верхних оценок числа буквенных сравнений.
- Второй алгоритм Дюваля имеет преимущества перед первым алгоритмом только в случае, когда $|\boldsymbol{w}'| > 1$. Для случайно выбранных строк, даже определенных на малых алфавитах, такое событие крайне редкое. Получается, что второй алгоритм придуман для решения проблемы, которой практически не существует!
- С общей точки зрения: что более важно для алгоритмов с временем выполнения $O(n)$ — поведение этих алгоритмов в среднем или их выполнение в самом худшем случае? Различие между двумя алгоритмами с временем выполнения $\Theta(n)$ в самом худшем случае может быть относительно невелико, поэтому повышается значимость их поведения в среднем.
- Необходимо также учитывать влияние размера алфавита. Даже если алфавит очень мал, например $\alpha = 2$ или 4, то для хранения строки $\boldsymbol{x}$ требуется $n/8$ или $n/4$ байт соответственно. С другой стороны, во втором алгоритме Дюваля в новом массиве $\boldsymbol{\beta}'$ содержится $n/2$ элементов, являющихся целыми числами из интервала от 0 до $n/2$. Поэтому всего требуется памяти объемом не менее $\lceil n' \log n' \rceil / 8$ байт, где $n' = n/2+1$. Если n велико, то такой большой объем памяти может быть неприемлемым.
- Необходимо также учитывать характеристики компьютера, на котором реализуется алгоритм. Например, возможность компьютера производить относительно эффективно сравнение битовых строк или сравнение байтов (или даже полных машинных слов) может значительно повлиять на то, какой из алгоритмов Дюваля будет выполняться быстрее, поскольку даже для бинарного алфавита надо производить как сравнение одиночных битов, так и обработку длинных строк, состоящих из нескольких байтов (элементы массива $\boldsymbol{\beta}'$).

Все эти соображения высказаны не с целью приуменьшить роль теоретических компьютерных наук, предлагающих новые и все более элегантные и эффективные алгоритмы. Они высказаны с целью показать многочисленные факторы, которые

влияют на поведение алгоритмов, когда они *реализованы* в виде компьютерных программ, при этом влияние таких факторов зачастую можно оценить только экспериментальным путем. ■

Упражнения 6.1

1. Покажите, что строка $\boldsymbol{x}$ будет линдонским словом только тогда, когда в массиве суффиксов $\boldsymbol{\sigma}_{\boldsymbol{x}}[1] = 1$.
2. Покажите, что на стандартном алфавите (4.1) любое линдонское слово будет p-канонической строкой (терминология раздела 4.1). Справедливо ли обратное утверждение?
3. Пусть в лемме 1.4.7 целочисленная переменная r равна количеству вхождений выходного линдонского слова $\boldsymbol{w}$. Выразите r через переменные h, i и j, используемые в алгоритме 6.1.1. (Заметьте, что при $r > 1$ выводятся кратные строки.) Далее примените кодирование, которое уже многократно использовалось в этой книге, для того чтобы представить все r вхождений строки $\boldsymbol{w}$ в виде числового кортежа. Сделайте соответствующую модификацию алгоритма 6.1.1. Какова асимптотическая временна́я сложность модифицированного алгоритма?
4. Для алфавитов размером $\alpha = 2, 4, 8$ и строк длиной $n = 100$ Кбайт, 1 Мбайт и 5 Мбайт оцените объемы памяти, необходимые для первого и второго алгоритмов Дюваля. Далее без проведения обширных экспериментов проанализируйте соотношение "стоимость–эффективность использования" каждого алгоритма Дюваля для приведенных 9 вариантов "размер алфавита–длина строки".
5. Приведите примеры самых худших (или хотя бы "плохих") строк для обоих алгоритмов Дюваля. На основе этих примеров сделайте заключение о приведенных в тексте границах количеств буквенных сравнений для этих алгоритмов.
6. Во втором алгоритме Дюваля массив $\boldsymbol{\beta}'$ изменяется таким образом, чтобы гарантировать выполнение неравенства $j - h \leqslant n/2$. Поэтому данный массив содержит только $n/2$ элементов. Имеется возможность "освободить" массив $\boldsymbol{\beta}'$ от гарантии выполнения этого неравенства. Но в этом случае он будет содержать n элементов. Обсудите все "за" и "против" этих двух вариантов массива $\boldsymbol{\beta}'$.
7. Предположим, что при сравнении букв μ и λ равновероятны три исхода $>$, $=$ и $<$. Определите нижнюю и верхнюю границы *ожидаемого* количества буквенных сравнений в каждом алгоритме Дюваля.
8. Является ли второй алгоритм Дюваля онлайновым?

9. В статье [80] введено множество P как множество всех префиксов линдонских слов, определенных на данном алфавите A. Если алфавит A содержит максимальную букву ω, то множество P' — это множество всех кратных строк с буквой ω. (Если в алфавите A нет буквы ω, то $P' = \varnothing$.) Докажите, что $L^* = P \cup P'$.

Совет. Используйте лемму 6.1.1, в частности утверждение *a*) этой леммы.

6.2 Применения линдонской декомпозиции

В этом разделе покажем, как первый алгоритм Дюваля можно изменить так, чтобы он за время порядка $\Theta(n)$ вычислял следующие внутренние паттерны для заданной строки $\boldsymbol{x}$:

- лексикографически минимальный непустой суффикс $s_{\min}(\boldsymbol{u})$,
- лексикографически максимальный суффикс $s_{\max}(\boldsymbol{u})$,

определяемые для любого непустого префикса $\boldsymbol{u}$ строки $\boldsymbol{x}$. Конечно же, подобные изменения можно внести и во второй алгоритм Дюваля.

Основой вычисления обоих паттернов послужат приведенные ниже две леммы. В первой лемме рассматривается разбиение линдонской декомпозиции строки $\boldsymbol{x}$, подобное применяемому в алгоритме Дюваля:

$$\boldsymbol{x}[1..j] = \boldsymbol{x}_1\boldsymbol{x}_2[1..j-h],$$

где $|\boldsymbol{x}_1| = h$ и декомпозиция $\mathrm{CFL}(\boldsymbol{x}_1)$ считается уже вычисленной. В этой лемме утверждается, что

$$s_{\min}(\boldsymbol{x}[1..j]) = s_{\min}(\boldsymbol{x}_2[1..j-h]).$$

Это означает, что вычисление $s_{\min}$ для позиций подстроки $\boldsymbol{x}_2$ не зависит от префикса $\boldsymbol{x}_1$.

Лемма 6.2.1. Пусть $\boldsymbol{x} = \boldsymbol{x}_1\boldsymbol{w}$, где $\boldsymbol{x}_1$ и $\boldsymbol{w}$ — непустые строки. Тогда равенство $\mathrm{CFL}(\boldsymbol{x}) = \mathrm{CFL}(\boldsymbol{x}_1)$ выполняется только в том случае, когда $s_{\min}(\boldsymbol{x}) = \boldsymbol{w}$.

Доказательство. Если $\mathrm{CFL}(\boldsymbol{x}) = \mathrm{CFL}(\boldsymbol{x}_1)\boldsymbol{w}$, тогда $\boldsymbol{w}$ является линдонским словом и мы можем записать

$$CFL(\boldsymbol{x}) = \boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_k,$$

где $\boldsymbol{w}_k = \boldsymbol{w}$ и $\boldsymbol{w}_1 \geqslant \boldsymbol{w}_2 \geqslant \cdots \geqslant \boldsymbol{w}_k$. Без потери общности можно положить, что $s_{\min}(x) = \boldsymbol{w}'_i\boldsymbol{w}_{i+1}\ldots\boldsymbol{w}_k$ для некоторого целого $i \in 1..k$, где $\boldsymbol{w}'_i$ — непустой суффикс подстроки $\boldsymbol{w}_i$. Вследствие леммы 1.4.6, $\boldsymbol{w}'_i \geqslant \boldsymbol{w}_i$, и поскольку $\boldsymbol{w}_i \geqslant \boldsymbol{w}_k$, то отсюда следует, что $i = k$ и $s_{\min}(\boldsymbol{x}) = \boldsymbol{w}_k = \boldsymbol{w}$.

Теперь докажем в обратную сторону: предположим, что $s_{\min}(\boldsymbol{x}) = \boldsymbol{w}$, но $\mathrm{CFL}(\boldsymbol{x}) = \mathrm{CFL}(\boldsymbol{x}_1)\overline{\boldsymbol{w}}$, где $\overline{\boldsymbol{w}} \neq \boldsymbol{w}$. Тогда, повторяя приведенные выше аргументы, получим $s_{\min}(\boldsymbol{x}) = \overline{\boldsymbol{w}} \neq \boldsymbol{w}$, т.е. пришли к противоречию. ■

Вторая лемма структурно похожа на лемму 6.1.1 и поэтому также ведет к алгоритму, который структурно похож на первый алгоритм Дюваля.

Лемма 6.2.2. Пусть $\boldsymbol{x}_2 \in L^*$ и $\mathrm{NF}(\boldsymbol{x}_2) = \boldsymbol{w}^r\boldsymbol{w}'$ для некоторого $\boldsymbol{w} \in L$ и пусть $\boldsymbol{w} = \boldsymbol{w}'\lambda\boldsymbol{w}''$, где $\lambda \in A$. Пусть также $\boldsymbol{s}'\lambda = s_{\min}(\boldsymbol{w}'\lambda)$. Тогда для $\mu \in A$

а) $\mu > \lambda \Rightarrow s_{\min}(\boldsymbol{x}_2\mu) = \boldsymbol{x}_2\mu$;

б) $\mu = \lambda \Rightarrow s_{\min}(\boldsymbol{x}_2\mu) = \boldsymbol{s}'\mu$;

в) $\mu < \lambda \Rightarrow$ для любой строки $\boldsymbol{x}'s_{\min}(\boldsymbol{x}_2\mu\boldsymbol{x}') = s_{\min}(\boldsymbol{s}'\mu\boldsymbol{x}')$.

Доказательство. утверждения *а*): из леммы 6.1.1 следует $\boldsymbol{x}_2\mu \in L$, поэтому на основании леммы 1.4.6 можем утверждать, что $\boldsymbol{x}_2\mu$ меньше любого собственного суффикса.

Для доказательства утверждения *б*) достаточно заметить, что из леммы 6.1.2 следует $\mathrm{CFL}(\boldsymbol{x}_2\lambda) = \boldsymbol{w}^r\,\mathrm{CFL}(\boldsymbol{w}'\lambda)$, где $\boldsymbol{w} \geqslant \boldsymbol{w}'\lambda$ и $\boldsymbol{w} < R_j(\boldsymbol{w})$ для всех $j \in \in 1..n-1$.

Доказательство утверждения *в*): из леммы 1.4.7 следует, что

$$\mathrm{CFL}(\boldsymbol{x}_2\mu\boldsymbol{x}') = \boldsymbol{w}^r CFL(\boldsymbol{w}'\mu\boldsymbol{x}').$$

Тогда (как и в утверждении *б*)) $s_{\min}(\boldsymbol{x}_2\mu\boldsymbol{x}') = s_{\min}(\boldsymbol{w}'\mu\boldsymbol{x}')$.

Обозначим через $\boldsymbol{S}'$ суффикс строки $\boldsymbol{w}'$, такой, что $|\boldsymbol{S}'| > |\boldsymbol{s}'|$. Так как $\boldsymbol{s}'\lambda = = s_{\min}(\boldsymbol{w}'\lambda)$, то $\boldsymbol{s}'\lambda < \boldsymbol{S}'\lambda$. Поскольку $|\boldsymbol{s}'\lambda| \leqslant |\boldsymbol{S}'|$, тогда $\boldsymbol{s}'\lambda \leqslant \boldsymbol{S}'$ и $\boldsymbol{s}'\lambda < \boldsymbol{S}'\mu\boldsymbol{x}'$. Поэтому $\boldsymbol{s}'\mu\boldsymbol{x}' < \boldsymbol{s}'\lambda < \boldsymbol{S}'\mu\boldsymbol{x}'$ и, следовательно, $S'\mu\boldsymbol{x}'$ не может быть минимальным суффиксом строки $\boldsymbol{w}'\mu\boldsymbol{x}'$. Отсюда заключаем, что любой минимальный суффикс $\boldsymbol{S}'$ должен удовлетворять неравенству $|\boldsymbol{S}'| \leqslant |\boldsymbol{s}'|$ и поэтому должен быть суффиксом строки $\boldsymbol{s}'$. ■

Леммы 6.2.1 и 6.2.2 содержат всю информацию, необходимую для вычисления функции $s_{\min}$ для любого префикса строки $\boldsymbol{x}$. Как указывалось выше, лемма 6.2.1 позволяет не рассматривать строку $\boldsymbol{x}_1$, т.е. префикс, для которого линдонская декомпозиция уже вычислена (и для которого уже вычислена функция $s_{\min}$). Утверждение *а*) леммы 6.2.2 является просто частным случаем леммы 6.2.1: линдонское слово является собственным минимальным суффиксом. Из утверждений *б*) и *в*) леммы 6.2.2 следует, что $s_{\min}(\boldsymbol{x}_2\mu)$ полностью определяется предварительно вычисленной величиной $s_{\min}(\boldsymbol{w}'\lambda)$, где $\boldsymbol{w}'\lambda$ — префикс строки $\boldsymbol{w}$ и собственный префикс строки $\boldsymbol{x}_2\mu$.

Алгоритм вычисления минимального суффикса показан в приведенном ниже листинге как алгоритм 6.2.1. Чтобы упростить вывод, выводятся не сами найденные минимальные суффиксы, а целочисленный массив $\boldsymbol{\sigma}' = \boldsymbol{\sigma}'[1..n]$, где для каждого $j \in 1..n$ $s_{\min}(\boldsymbol{x}[1..j]) = \boldsymbol{x}[\boldsymbol{\sigma}'[j]..j]$. Приведем пример такого массива для строки Фибоначчи $\boldsymbol{f}_6$ (пробелами отделены линдонские слова).

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f}_6 =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$\boldsymbol{\sigma}' =$	1	1	3	4	3	6	3	8	9	8	11	12	11

Алгоритм 6.2.1 (Алгоритм вычисления минимальных суффиксов)

▷ *Для входной строки $\boldsymbol{x}$\$ выводится массив $\boldsymbol{\sigma}'[1..n]$, такой, что*
▷ *подстрока $\boldsymbol{x}[1..j]$ имеет непустой суффикс $\boldsymbol{x}[i..j]$ только тогда,*
▷ *когда $\boldsymbol{\sigma}'[j] = i$*
$h \leftarrow 0$; $\boldsymbol{\sigma}'[1] \leftarrow 1$
while $h < 0$ **do**
  $i \leftarrow h + 1$ ▷ $\boldsymbol{x}[i] = \lambda$
  $j \leftarrow h + 2$ ▷ $\boldsymbol{x}[j] = \mu$
  while $\boldsymbol{x}[j] \geqslant \boldsymbol{x}[i]$ **do** ▷ *выполняется проверка* $\mu \geqslant \lambda$
    if $\boldsymbol{x}[j] > \boldsymbol{x}[i]$ **then**
      ▷ *случай* $\mu > \lambda$, *тогда* $s_{\min}(\boldsymbol{x}_2\mu) = \boldsymbol{x}_2\mu$
      ▷ $\boldsymbol{\sigma}'[j] \leftarrow h + 1$; $i \leftarrow h + 1$
    else
      ▷ *случай* $\mu = \lambda$, *тогда* $s_{\min}(\boldsymbol{x}_2\mu) = s_{\min}(\boldsymbol{w}'\mu)$
      $\boldsymbol{\sigma}'[j] \leftarrow \boldsymbol{\sigma}'[i] + (j - i)$; $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  ▷ *случай* $\mu < \lambda$, *вычисление* h
  $r \leftarrow \lceil (i - h)/(j - i) \rceil$; $h \leftarrow h + r \times (j - i)$
  if $h = j - 1$ **then**
    $\boldsymbol{\sigma}'[j] \leftarrow h + 1$

Теорема 6.2.3. Алгоритм 6.2.1 корректно вычисляет массив $\boldsymbol{\sigma}'[1..n]$ (минимальные непустые суффиксы для всех непустых префиксов) для заданной строки $\boldsymbol{x}$\$ за время порядка $\Theta(n)$.

Доказательство. Заметим, что ход вычислений и использование переменных h, i и j в алгоритме 6.2.1 такие же, как и в первом алгоритме Дюваля, за исключением вычисления h для случая $\mu < \lambda$. Но в последнем случае для вычисления h используется переменная r, которая дает то же самое значение, что и вычисления в цикле **repeat** в алгоритме Дюваля (см. упражнение 6.2.1). Увеличение значения h (если $h = j - 1$) просто указывает на особый случай: когда одна оставшаяся буква $\boldsymbol{x}[j]$ оказывается наименьшей буквой подстроки $\boldsymbol{x}[1..j]$, тогда $s_{\min}(\boldsymbol{x}[1..j]) = \boldsymbol{x}[j]$ (т.е. это тот самый случай, когда $h = 0$ и $j = 1$). Таким образом, алгоритм 6.2.3 выполняется практически так же, как первый алгоритм Дюваля, и поэтому время его выполнения составляет $\Theta(n)$.

Теперь рассмотрим четыре изменения, производимые в массиве $\boldsymbol{\sigma}'$. Мы уже рассмотрели случай, когда $h = j - 1$. Также тривиально изменение $\boldsymbol{\sigma}'[1] \leftarrow 1$. Два

других изменения, которые вызваны ситуациями, когда $\mu > \lambda$ и $\mu = \lambda$, соответствуют утверждениям *а*) и *б*) леммы 6.2.2. Следовательно, алгоритм корректно вычисляет массив $\boldsymbol{\sigma}'$. ■

Теперь рассмотрим вторую задачу, сформулированную в начале этого раздела: вычисление лексикографически максимального суффикса $s_{\max}(\boldsymbol{u})$ любого непустого префикса $\boldsymbol{u}$ строки $\boldsymbol{x}$. Очевидный подход к решению данной задачи — заменить отношение лексикографического порядка $<$ на обратный лексикографический порядок $<_{\mathcal{R}}$ в надежде, что использование этого нового порядка в алгоритме 6.2.1 позволит получить максимальные, а не минимальные суффиксы. Итак, сначала определим новый лексикографический порядок для пар букв λ и μ алфавита A:

$$\mu <_{\mathcal{R}} \lambda \Leftrightarrow \lambda < \mu. \tag{6.2}$$

Далее следует распространить новый порядок на все строки, определенные на алфавите A. Но здесь нас подстерегают сложности: например, для строк $\boldsymbol{u} = a$ и $\boldsymbol{v} = ab$ выполняется неравенство $\boldsymbol{u} < \boldsymbol{v}$, но $\boldsymbol{v} \not<_{\mathcal{R}} \boldsymbol{u}$. Однако имеет место следующее утверждение.

Лемма 6.2.4. Для любых строк $\boldsymbol{u}$ и $\boldsymbol{v}$, определенных на упорядоченном алфавите A, одновременное выполнение неравенств $\boldsymbol{u} < \boldsymbol{v}$ и $\boldsymbol{v} <_{\mathcal{R}} \boldsymbol{u}$ возможно только в том случае, когда $\boldsymbol{u}$ является префиксом $\boldsymbol{v}$.

Доказательство этой леммы предложено дать в упражнении 6.2.6. ■

Таким образом, отношение порядка $<_{\mathcal{R}}$ распространяется на строки только в том случае, если одна из них является префиксом другой. Рассмотрим применение леммы 6.2.4 к строкам $\boldsymbol{x} = cab$ и $\boldsymbol{y} = bab$, у обеих этих строк максимальными суффиксами являются они сами. Для строки $\boldsymbol{x}$ нет проблем: эта строка является линдонским словом, она примитивна и лексикографически меньше, в соответствии с отношением порядка $<_{\mathcal{R}}$, чем любой из ее суффиксов, а в соответствии с отношением порядка$<$ она больше, чем любой ее суффикс. Для строки $\boldsymbol{y}$ ситуация иная: эта строка не является линдонским словом, она не примитивна и лексикографически не меньше, в соответствии с отношением порядка $<_{\mathcal{R}}$, чем любой из ее суффиксов (поскольку $bab \not<_{\mathcal{R}} b$), но в соответствии с отношением порядка $<$ она больше, чем любой ее суффикс (так как $bab > b > ab$). Поэтому применение алгоритма 6.2.1 с заменой отношения $<$ на отношение $<_{\mathcal{R}}$ приведет к желаемому результату для строки $\boldsymbol{x} = cab$, но не для строки $\boldsymbol{y} = bab$. В общем случае лемма 6.2.4 показывает, что такое применение алгоритма 6.2.1 будет некорректным для любых строк с непустой гранью.

Таким образом, такой наивный подход, как замена отношения $<$ на отношение $<_{\mathcal{R}}$, задачу нахождения максимального суффикса не решает. Однако в введении

обратного отношения порядка $<_{\mathcal{R}}$ есть рациональное зерно (например, для строки $\boldsymbol{x} = cab$, как показано выше, получается правильный результат). Поэтому рассмотрим данное отношение подробнее. Напомним, что L^* обозначает множество всех строк, порожденных линдонскими словами. Обозначим через $L^{\mathcal{R}}$ множество всех строк $\boldsymbol{x}$, таких, что, в соответствии с отношением порядка $<_{\mathcal{R}}$, $s_{\max}(\boldsymbol{x}) = \boldsymbol{x}$. Следующая цепочка лемм покажет, что $L^{\mathcal{R}} = L^*$.

Лемма 6.2.5. Пусть $\boldsymbol{x} \in L^{\mathcal{R}}$ и $\boldsymbol{x}'$ — непустой префикс строки $\boldsymbol{x}$. Тогда $\boldsymbol{x}' \in L^{\mathcal{R}}$.

Доказательство. Пусть $\boldsymbol{x} = \boldsymbol{x}'\boldsymbol{y}$ и пусть $\boldsymbol{u}'$ — собственный префикс строки $\boldsymbol{x}'$. Поскольку $\boldsymbol{x} \in L^{\mathcal{R}}$, то $\boldsymbol{u}' <_{\mathcal{R}} \boldsymbol{x} = \boldsymbol{x}'\boldsymbol{y}$. Но так как $|\boldsymbol{u}'| < |\boldsymbol{x}'|$, то $\boldsymbol{u}' <_{\mathcal{R}} \boldsymbol{x}'$ и, в соответствии с отношением порядка $<_{\mathcal{R}}$, $s_{\max}(\boldsymbol{x}') = \boldsymbol{x}'$. Следовательно, $\boldsymbol{x}' \in L^{\mathcal{R}}$, что и требовалось доказать. ■

Лемма 6.2.6. Пусть $\boldsymbol{x} \in L^{\mathcal{R}}$ и $\boldsymbol{x}'$ — префикс строки $\boldsymbol{x}$. Тогда $\mathrm{x}\boldsymbol{x}' \in L^{\mathcal{R}}$.

Доказательство. Утверждение тривиально, если $\boldsymbol{x}' = \varepsilon$, поэтому положим, что $\boldsymbol{x}' \neq \varepsilon$. Пусть $\boldsymbol{u}$ — собственный суффикс строки $\mathrm{x}\boldsymbol{x}'$. Возможны два случая.

1. $\boldsymbol{u} = \boldsymbol{x}''\boldsymbol{x}'$, где $\boldsymbol{x}''$ — непустой собственный суффикс строки $\boldsymbol{x}$.
   Поскольку $\boldsymbol{x} \in L^{\mathcal{R}}$, то $\boldsymbol{x}'' <_{\mathcal{R}} \boldsymbol{x}$. Из неравенства $|\boldsymbol{x}''| < |\boldsymbol{x}|$ следует, что $\boldsymbol{u} = \boldsymbol{x}''\boldsymbol{x}' <_{\mathcal{R}} \boldsymbol{x}\mathrm{x}'$.
2. $\boldsymbol{u}$ является суффиксом $\boldsymbol{x}'$.
   Если $\boldsymbol{u} = \boldsymbol{x}'$, тогда $\boldsymbol{u}$ — собственный префикс строки $\boldsymbol{x}\boldsymbol{x}'$ и, следовательно, $\boldsymbol{u} <_{\mathcal{R}} \boldsymbol{x}\boldsymbol{x}'$. Если $\boldsymbol{u}$ — собственный суффикс строки $\boldsymbol{x}'$, то, на основании леммы 6.2.5, $\boldsymbol{x}' \in L^{\mathcal{R}}$. Тогда $\boldsymbol{u} <_{\mathcal{R}} \boldsymbol{x}' <_{\mathcal{R}} \boldsymbol{x}\boldsymbol{x}'$.

В обоих случаях $\mathrm{x}\boldsymbol{x}' \in L^{\mathcal{R}}$. ■

Лемма 6.2.7. Если $\boldsymbol{x} \in L$, тогда $\boldsymbol{x} \in L^{\mathcal{R}}$.

Доказательство. Вследствие леммы 1.4.5 строка $\boldsymbol{x}$ примитивна, поэтому любой ее непустой собственный суффикс $\boldsymbol{u}$ гарантированно не будет ее префиксом. Из леммы 1.4.6 вытекает, что $\boldsymbol{x} < \boldsymbol{u}$, и, вследствие леммы 6.2.4, $\boldsymbol{u} <_{\mathcal{R}} \boldsymbol{x}$. Но тогда, в соответствии с отношением $<_{\mathcal{R}}$, $s_{\max}(\boldsymbol{x}) = \boldsymbol{x}$, что и требовалось доказать. ■

Лемма 6.2.8. Если $\boldsymbol{x} \in L^*$, тогда $\boldsymbol{x} \in L^{\mathcal{R}}$.

Доказательство непосредственно следует из лемм 6.2.6 и 6.2.7. ■

Лемма 6.2.9. $L^{\mathcal{R}} = L^*$.

Доказательство. Из леммы 6.2.8 следует, что $L^* \subset L^{\mathcal{R}}$. Докажем включение в обратную сторону. Пусть $\boldsymbol{x} \in L^{\mathcal{R}}$, $\boldsymbol{u}$ — ее наибольший префикс, который принадлежит L^*. Тогда $\mathrm{NF}(\boldsymbol{u}) = \boldsymbol{w}^r\boldsymbol{w}'$ для некоторой строки $\boldsymbol{w} = \boldsymbol{w}'\lambda\boldsymbol{w}'' \in L$.

Предположим, что $\boldsymbol{u} \neq \boldsymbol{x}$. Тогда $\boldsymbol{x} = \boldsymbol{u}\mu\boldsymbol{x}'$ для некоторого непустого суффикса $\boldsymbol{x}'$. Поскольку $\boldsymbol{u}\mu \notin L^*$, из леммы 6.1.1 следует, что $\mu < \lambda$. Поэтому $\lambda <_{\mathcal{R}} \mu$, тогда $\boldsymbol{w}'\lambda <_{\mathcal{R}} \boldsymbol{w}'\mu$ и $\boldsymbol{x} = \boldsymbol{u}\mu\boldsymbol{x}' <_{\mathcal{R}} \boldsymbol{u}\mu\boldsymbol{x}'$, что противоречит предположению о том, что $\boldsymbol{x} \in L^{\mathcal{R}}$. Отсюда заключаем, что $\boldsymbol{u} = \boldsymbol{x} \in L^*$. ■

Чтобы иметь возможность применить полученные результаты к алгоритмам Дюваля, надо сначала разобраться, что же следует из этих результатов. Лемма 6.2.9 говорит, что строка $\boldsymbol{u}$ принадлежит множеству L^* только в том случае, когда строка $\boldsymbol{u}$ сама является своим максимальным суффиксом в соответствии с отношением $<_{\mathcal{R}}$. Поскольку $(<_{\mathcal{R}})_{\mathcal{R}} \equiv <$, обратное утверждение также имеет место: строка $\boldsymbol{u} \in$ $\in L^*$ в соответствии с отношением $<_{\mathcal{R}}$ только в том случае, когда $\boldsymbol{u}$ является своим максимальным суффиксом в соответствии с отношением $<$. Поэтому, выполняя алгоритм Дюваля с отношением $<_{\mathcal{R}}$, мы не потеряли надежду каким-то образом определить строки $\boldsymbol{u}$, такие, что $\boldsymbol{u} = s_{\max}(\boldsymbol{u})$.

Эти рассуждения подводят нас к критическому замечанию: строка $\boldsymbol{u}$ будет максимальным суффиксом строки $\boldsymbol{x}$ и линдонским словом только в том случае, когда строка $\boldsymbol{u}$ будет *наибольшим* суффиксом строки $\boldsymbol{x}$, таким, что $\boldsymbol{u} = s_{\max}(\boldsymbol{u})$. Но вследствие леммы 6.2.9 каждая строка $\boldsymbol{u}$, являющаяся своим максимальным суффиксом и вычисленная на основании одного отношения порядка, принадлежит множеству L^*, вычисленному с помощью обратного отношения порядка. Таким образом, приходим к алгоритму, где сначала на основании отношения порядка $<_{\mathcal{R}}$ вычисляются элементы множества L^* (т.е. суффиксы строки $\boldsymbol{x}$), затем, в соответствии с отношением порядка$<$, в множестве L^* выбирается максимальная строка — это и будет максимальный суффикс $\boldsymbol{u}$ строки $\boldsymbol{x}$. Но это в точности описание алгоритма Дюваля: для каждого префикса $\boldsymbol{x}[1..j]$ определяются элементы множества L^*, которые заканчиваются в позиции j, т.е. вычисляются суффиксы префикса $\boldsymbol{x}[1..j]$. Выбрать для каждого j наибольший элемент из этого множества просто, поскольку алгоритм Дюваля просматривает строку $\boldsymbol{x}$ слева направо — надо взять самый левый элемент. Таким образом, приходим к следующему алгоритму вычисления максимального суффикса.

Алгоритм 6.2.2 (Алгоритм вычисления максимальных суффиксов)

▷ *Для входной строки $\boldsymbol{x}$\$ выводится массив $\boldsymbol{\sigma}''[1..n]$, такой, что*
▷ *подстрока $\boldsymbol{x}[1..j]$ имеет максимальный суффикс $\boldsymbol{x}[i..j]$ только тогда,*
▷ *когда $\boldsymbol{\sigma}''[j] = i$*
for $j \in 1..n$ **do** $\boldsymbol{\sigma}''[j] \leftarrow 0$
$h \leftarrow 0$; $\boldsymbol{\sigma}''[1] \leftarrow 1$
while $h < 0$ **do**
    $i \leftarrow h + 1 - \boldsymbol{x}[i] = \lambda$
    $j \leftarrow h + 2 - \boldsymbol{x}[j] = \mu$
    while $\boldsymbol{x}[j] >_{\mathcal{R}} \boldsymbol{x}[i]$ **do** ▷ *выполняется проверка $\mu \leqslant \lambda$*

```
    if x[j] > x[i] then
        i ← h + 1
    else
        i ← i + 1
    if σ''[j] = 0 then
        σ''[j] ← h + 1
    j ← j + 1
▷ случай μ > λ, вычисление h
r ← ⌈(i − h)/(j − i)⌉; h ← h + r × (j − i)
if h = j − 1 then
    if σ''[j] = 0 then
        σ''[j] ← h + 1
```

Как и в случае алгоритма 6.2.1, здесь для компактного представления вычисленных данных используется целочисленный массив $\boldsymbol{\sigma}''[1..n]$, где для любого целого $j \in 1..n$ $\boldsymbol{\sigma}''[j] = i$ только в том случае, если $s_{\max}(\boldsymbol{x}[1..j]) = \boldsymbol{x}[i..j]$. Другими словами, $\boldsymbol{\sigma}''[j] = i$ только тогда, когда $\boldsymbol{x}[i..j]$ является наибольшим суффиксом строки $\boldsymbol{x}[1..j]$ и при этом $s_{\max}(\boldsymbol{x}[i..j]) = \boldsymbol{x}[i..j]$. Таким образом, для каждой позиции j, т.е. для каждого префикса $\boldsymbol{x}[1..j]$, находится наименьшее значение i, такое, что $\boldsymbol{x}[i..j] \in L^*$ в соответствии с отношением $<_{\mathcal{R}}$. Поскольку позиция $h+1$ всегда обозначает начало элемента множества L^*, который обрабатывается в данный момент, то очевидно предназначение оператора $\boldsymbol{\sigma}''[j] \leftarrow h+1$. Но так как позиция j может встретиться в нескольких перекрывающихся словах множества L^*, то для того, чтобы использовалось наименьшее подходящее значение переменной h, данный оператор выполняется только после проверки условия $\boldsymbol{\sigma}''[j] = 0$, как показано в листинге.

Покажем пример вычисления максимальных суффиксов для строки $\boldsymbol{f}_6$, которая использовалась выше в примере вычисления минимальных суффиксов. Отметим, что в этом примере пробелы отделяют линдонские слова в соответствии с отношением порядка $<_{\mathcal{R}}$. Эти же пробелы отделяют слова в линдонской декомпозиции (в соответствии с отношением<) сопряженной строки $\boldsymbol{f}'_6$, полученной из строки $\boldsymbol{f}_6$ путем взаимной замены букв a и b.

1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f}_6 = a$	b	a	a	b	a	b	a	a	b	a	a	b
$\boldsymbol{f}'_6 = b$	a	b	b	a	b	a	b	b	a	b	b	a
$\boldsymbol{\sigma}' = 1$	2	2	2	2	2	5	5	5	5	5	5	5

Теорема 6.2.10. Алгоритм 6.2.2 корректно вычисляет массив $\boldsymbol{\sigma}''[1..n]$ (максимальные суффиксы для всех непустых префиксов) для заданной строки $\boldsymbol{x}$ за время порядка $\Theta(n)$.

Доказательство. Алгоритмы 6.2.1 и 6.2.2 отличаются только вычислением массивов $\boldsymbol{\sigma}'$ и $\boldsymbol{\sigma}''$, поэтому асимптотические оценки времени их выполнения совпадают. Корректность вычисления массива $\boldsymbol{\sigma}''$ следует из леммы 6.2.9 и суждений, приведших к данному алгоритму. ■

В этом разделе мы рассмотрели несколько вариантов алгоритмов Дюваля, но, как часто бывает с интересными строковыми задачами, существуют и другие алгоритмы, которые вычисляют линдонскую декомпозицию для заданной строки $\boldsymbol{x}$ или решают близкую к этой задаче задачу построения канонической формы для петли $C(\boldsymbol{x})$ с определением минимальной начальной точки $\mathrm{MSP}(\boldsymbol{x})$ (см. раздел 1.4). Алгоритмы [36, 208] вычисляют $\mathrm{MSP}(\boldsymbol{x})$, используя для этого обобщение массива граней. Другой алгоритм [126] вычисления линдонской декомпозиции конкурирует с алгоритмами Дюваля по времени выполнения в среднем, но в самом худшем случае выполняется за время порядка $\Theta(n \log n)$. Специальный алгоритм предложен в работе [207] для сравнения петель $C(\boldsymbol{x}_1)$ и $C(\boldsymbol{x}_2)$, который можно непосредственно применить для решения задачи о конгруэнтности многоугольников, которая рассматривалась в разделе 1.4.

Возможно, наиболее интересный алгоритм вычисления канонических форм предложен в работе [13]. В этой работе также показано, что первый алгоритм Дюваля можно применить для вычисления $\mathrm{MSP}(\boldsymbol{x})$, причем без увеличения числа буквенных сравнений. Второй алгоритм Дюваля можно сделать онлайновым алгоритмом вычисления минимальных начальных точек для всех префиксов строки $\boldsymbol{x}$ за время $\Theta(n)$ (и при этом потребуется всего в полтора раза больше буквенных сравнений, чем в первом алгоритме Дюваля). Этот алгоритм может вычислить минимальные начальные точки для всех *подстрок* строки $\boldsymbol{x}$ за время $\Theta(n^2)$ при дополнительной памяти $\Theta(n)$.

Упражнения 6.2

1. Покажите, что в алгоритме 6.2.1 операторы $r \leftarrow \lceil (i-h)/(j-i) \rceil$; $h \leftarrow h + + r \times (j-i)$ можно заменить одним оператором $h \leftarrow \boldsymbol{\sigma}'[i] + (j-i+1)$.
2. Необходим ли для алгоритма 6.2.1 во входной строке символ окончания строки \$? Покажите, как следует изменить этот алгоритм, чтобы можно было не использовать этот символ.
3. Покажите, как надо изменить алгоритм 6.1.2, чтобы он вычислял массив $\boldsymbol{\sigma}'$.
4. Охарактеризуйте линдонскую декомпозицию строк Фибоначчи $\boldsymbol{f}_n$.
5. Укажите необходимые и достаточные условия для справедливости следующих выражений.
   а) $\boldsymbol{\sigma}'[j] = 1$;
   б) $\boldsymbol{\sigma}'[j] = j$;

в) $\boldsymbol{\sigma}'[j+1] = \boldsymbol{\sigma}'[j] + 1$.

6. Докажите лемму 6.2.4.

7. Докажите лемму 6.2.8.

8. Покажите, как можно изменить алгоритм 6.1.2, чтобы он вычислял массив $\boldsymbol{\sigma}''$.

9. Докажите, что $\boldsymbol{\sigma}''[j+1] \geqslant \boldsymbol{\sigma}''[j]$ для любого $j \in 1..n-1$.

10. В упражнении 5.2.19 введено дерево позиций. Покажите, как дерево позиций можно использовать для вычисления канонических форм петель $C(\boldsymbol{x})$.

6.3 s-факторизация

В этом разделе рассмотрим другую, отличную от линдонской, декомпозицию строк $\boldsymbol{x} = \boldsymbol{x}[1..n]$, которая зависит от повторяемости подстрок внутри строки $\boldsymbol{x}$. В отличие от линдонской, данная декомпозиция не зависит от отношения лексикографического порядка и поэтому может применяться в случае неупорядоченных алфавитов.

Определение 6.3.1. Декомпозиция $\boldsymbol{x} = \boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_k$ называется s-**факторизацией**, если каждая подстрока $\boldsymbol{w}_j$ ($j = 1, 2, \ldots, k$) является или

а) буквой, которая не встречается в подстроке $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{j-1}$, или

б) подстрокой наибольшей длины, которая по крайней мере один раз встречалась в $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{j-1}$. ■

Для строки $\boldsymbol{x}$ s-факторизация строится при просмотре этой строки слева направо, последовательно вычисляя текущий фактор $\boldsymbol{w}_j$ на основе уже вычисленных факторов $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{j-1}$. По определению $\boldsymbol{w}_1 = \boldsymbol{x}[1]$. На j-м шаге, если буква, стоящая в позиции

$$p_j = |\boldsymbol{w}_1| + |\boldsymbol{w}_2| + \cdots + |\boldsymbol{w}_{j-1}| + 1,$$

еще не встречалась в подстроке $\boldsymbol{x}[1..p_j - 1]$, то в этом случае применяется часть *а*) определения 6.3.1. Если же эта буква уже встречалась в подстроке $\boldsymbol{x}[1..p_j - 1]$, тогда, согласно части *б*) определения, надо найти наибольшую подстроку, которая начинается в позиции p_j и которая уже имеется в подстроке $\boldsymbol{x}[1..p_j - 1]$.

Рассмотрим пример s-факторизации для строки Фибоначчи

$$\boldsymbol{f}_6 = abaababaabaab.$$

Эта факторизация состоит из таких факторов: $\boldsymbol{w}_1 = a$, $\boldsymbol{w}_2 = b$, $\boldsymbol{w}_3 = a$, $\boldsymbol{w}_4 = aba$, $\boldsymbol{w}_5 = baaba$, $\boldsymbol{w}_6 = ab$. Кратко s-факторизацию можно записать как $a/b/a/aba/baaba/ab$.

Заметим, что в случае *б*) определения 6.3.1 допускается перекрытие предыдущего вхождения фактора $\boldsymbol{w}_j$ с суффиксом этого же фактора. Например, s-факторизацией строки a^n будет a/a^{n-1}.

Интересно отметить, что хотя определение s-факторизации не зависит от природы алфавита, но известные эффективные алгоритмы вычисления s-факторизации используют деревья суффиксов и поэтому зависят от алфавита, который предполагается упорядоченным и, желательно, не слишком большим (см. раздел 2.1).

Чтобы показать, как вычисляется s-факторизация для заданной строки $\boldsymbol{x}$, предположим, что дерево суффиксов $T_{\boldsymbol{x}}$ уже построено (см. раздел 5.2). Далее предположим, что каждый внутренний узел i дерева $T_{\boldsymbol{x}}$ помечен числовой меткой, совпадающей с наименьшей числовой меткой конечных узлов, являющихся потомками данного узла i. Присвоение меток внутренним узлам можно выполнить с помощью стандартной процедуры обхода дерева $T_{\boldsymbol{x}}$ в обратном порядке, когда каждый из $\Theta(n)$ узлов посещается только один раз [135, 5]. Такие метки внутренних узлов являются номерами самых левых позиций в строке $\boldsymbol{x}$ подстрок, представленных путем от корня дерева до данного внутреннего узла. По соглашению корень дерева помечается меткой 0. Помеченное дерево суффиксов для строки Фибоначчи $\boldsymbol{f}_6 = abaababaabaab\$$ показано на рис. 6.1.

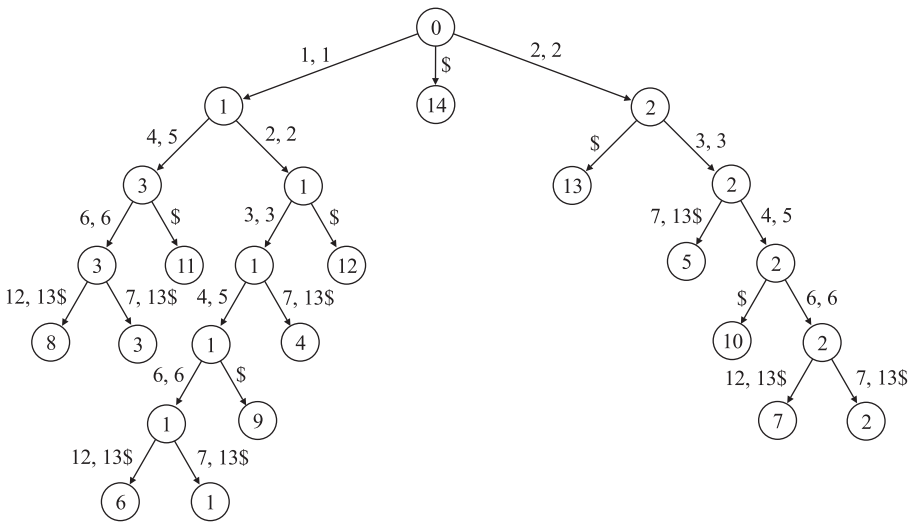


Рис. 6.1. Помеченное дерево суффиксов

Чтобы более детально показать процесс построения s-факторизации, обозначим каждый фактор $\boldsymbol{w}_j$ парой чисел (i_L, ℓ), где i_L — начальная позиция в строке $\boldsymbol{x}$ первого вхождения повторяющейся подстроки $\boldsymbol{w}_j$, $\ell = |\boldsymbol{w}_j|$ — длина подстро-

ки $\boldsymbol{w}_j$. Пусть i_0 — текущая позиция строки $\boldsymbol{x}$, т.е. позиция, следующая за префиксом $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{j-1}$. Для значения i_0 корректное значение ℓ можно вычислить путем просмотра дерева $T_{\boldsymbol{x}}$ от корня до конечного узла i_0. Для этого сначала надо положить $i \leftarrow i_0$, затем проверяется буква $\boldsymbol{x}[i]$, при этом i увеличивается при прохождении нисходящих ребер дерева, ведущих к узлу с меткой i_0. Если в конечной точке пути узел имеет ненулевую метку, например v, то тогда $i_L \leftarrow v$ и ℓ полагается равной длине подстроки, представленной узлом v. Если же последним узлом пути будет корень дерева (он помечен меткой 0), то полагаем $(i_L, \ell) \leftarrow (i_0, 0)$, что указывает на то, что в позиции i_L обнаружена новая буква. Алгоритм s-факторизации представлен в приведенном ниже листинге и основан на функции сравнения *match*, подробно описать которую предложено в упражнении 6.3.2. Этот алгоритм для строки $\boldsymbol{f}_6$ должен дать следующую последовательность пар (i_L, ℓ):

$$(1,0),(2,0),(1,1),(1,3),(2,5),(1,2). \tag{6.3}$$

Алгоритм 6.3.1 (Алгоритм Зива-Лемпела)

▷ *Вычисление s-факторизации заданной строки $\boldsymbol{x}$*
▷ *как последовательности пар (i_L, ℓ)*
▷ *на основе помеченного дерева суффиксов $T_{\boldsymbol{x}}$*
$i_0 \leftarrow 1$
while $i_0 \leqslant n$ **do**
  $(i_L, \ell) \leftarrow match(i_0, T_{\boldsymbol{x}})$
  output (i_L, ℓ)
  $i_0 \leftarrow i_0 + \ell$

Используя алгоритм Укконена (см. раздел 5.2.3) для построения дерева суффиксов, алгоритм 6.3.1 можно реализовать как онлайновый, т.е. s-факторизацию строки $\boldsymbol{x}$ можно вычислять в процессе построения дерева $T_{\boldsymbol{x}}$ [108, 235].

Чтобы использовать алгоритм 6.3.1 для сжатия данных, его следует немного изменить. Например, чтобы облегчить восстановление данных, первое вхождение каждой буквы необходимо указать точно, а для отделения факторов удобно использовать разделители. С учетом этих "пожеланий", s-факторизацию строки Фибоначчи $\boldsymbol{f}_6 = abaababaabaab$, полученную в виде последовательности (6.3), следует переписать как

$$a/b/1{,}1/1{,}3/2{,}5/1{,}2 \tag{6.4}$$

Но в таком случае возрастает количество используемых символов от 13 (количество букв в исходной строк $\boldsymbol{f}_6$) до 19 в представлении (6.4) (если считать разделители и запятые). В общем случае чем длиннее исходная строка, тем больше проявляется эффект сжатия данных. Например, для следующей строки Фибоначчи

$$\boldsymbol{f}_7 = abaababaabaababaababa$$

длиной 21 s-факторизация

$$a/b/1{,}1/1{,}3/2{,}5/4{,}8/2{,}2 \tag{6.5}$$

имеет длину 23, что не на много больше длины факторизации (6.4).

Метод сжатия данных Зива-Лемпена [235] и его модификации основаны на s-факторизации. Один из вариантов этого метода реализован в системе Unix в виде процедур *compress* (сжатие) и *uncompress* (восстановление). Обычно процедура *compress* сжимает длинный текст на английском языке в два или три раза по сравнению с исходным объемом текста. В упражнении 6.3.5 предложено детализировать алгоритм восстановления исходной строковой последовательности на основе имеющейся s-факторизации, подобной (6.5).

В разделе 12.2 мы рассмотрим применение s-факторизации для нахождения кратных подстрок в данной строке $\boldsymbol{x}$.

Упражнения 6.3

1. Докажите, что любая строка $\boldsymbol{x}$ имеет только одну s-факторизацию.

2. Запишите функцию *match*, которая используется в алгоритме 6.3.1. Затем докажите корректность этого алгоритма.

3. Постройте дерево суффиксов $T_{\boldsymbol{x}}$ для строки $\boldsymbol{x} = abaababcabab$ и затем вычислите s-факторизацию для этой строки.

4. Запишите s-факторизацию для строки $\boldsymbol{x} = (ab)^n$, $n \geqslant 2$, в форме (6.4). Каков коэффициент сжатия в данном случае, т.е. чему равно отношение количества символов в s-факторизации к длине строки $2n$?

Совет. Предположите, что любой символ как в строке $\boldsymbol{x}$, так и в ее s-факторизации требует для хранения полбайта (четыре бита).

5. Разработайте алгоритм процедуры *uncompress*, выполняющий за линейное время восстановление строки $\boldsymbol{x}$ на основании ее s-факторизации.

6. Иногда используется другая форма s-факторизации, где в определении 6.3.1 условие *б*) заменено следующим условием *б*):

б) лексикографически максимальной подстрокой, которая по крайней мере один раз встречалась в $\boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_{j-1}$.

Перепишите функцию *match* и алгоритм процедуры *uncompress* в соответствии с новым определением s-факторизации. Какой коэффициент сжатия в этом случае будет для строки $\boldsymbol{x} = (ab)^n$?

ЧАСТЬ III

Вычисление частных паттернов

Не хлебом одним будет жить человек, но всяким словом, исходящим из уст Божиих

— Матфей 4.4

Компьютерная наука, интеллектуальная научная дисциплина, без сомнения, восходит к Алану Тьюрингу (1935 год), но распознать эту науку в общем случае несложно: как заметил Винни-Пух, время определяют вещи. С такой точки зрения вполне уверенно можно утверждать, что наша дисциплина сформировалась примерно 40 лет назад. Прилагательное "классический" (и его производные), которое часто используется в таких науках и искусствах, как физика, музыка или математика, и которое указывает на достижения, достигнутые в этих науках и искусствах известными и прославленными первопроходцами сотни, а иногда и тысячи лет назад, кажется совсем неуместным применительно к компьютерной науке. В этом случае фразы "классический подход" или "классический алгоритм" будут относиться к работам, выполненным менее 25 лет назад людьми, которые до сих пор живы и продолжают плодотворно работать! То же самое можно сказать о той области компьютерной науки, которая изучается в данной книге: самые ранние работы по вычисляемым паттернам строковых последовательностей относятся к 60-м годам, интерес к вычисляемым частным паттернам (в виде "классического подхода" сравнения с паттерном) проявился в 70-х годах, в частности, после публикаций в 1977 году "классических" алгоритмов Бойера–Мура и Кнута–Морриса–Пратта, авторы которых продолжают творить среди нас.

После такого разъяснения представим материал данной части как "классический": задачи сравнения с паттернами были первыми изученными задачами этой области компьютерной науки и сегодня продолжают оставаться в поле пристального внимания исследователей. То, что эта область продолжает интенсивно развиваться, является ее преимуществом, а не недостатком, поскольку расширяется сфера применения данной области. В главах 7 и 8 рассмотрим алгоритмы локализации точных единичных паттернов в строковых последовательностях. В главах 9 и 10 эти алгоритмы будут "расширены" для выполнения сравнения с "приближенными" паттернами (как это определено в разделе 2.2). Наконец, в главе 11 рассмотрим применение конечных автоматов для сравнения с паттернами, в частности для сравнения регулярных выражений и для одновременной локализации в строке нескольких различных паттернов, причем как для точного, так и приближенного сравнения.

ГЛАВА 7

Базовые алгоритмы

Слова, без сомнения, наиболее мощный наркотик, который использует человечество.

— Редьярд Киплинг (1865–1936).
The Times, 15 февраля 1923

В этой главе мы рассмотрим четыре различных "классических" подхода к вычислению всех вхождений заданного непустого паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ в заданную непустую строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Первые два подхода (правда, второй после небольшой модификации) гарантируют выполнение соответствующих алгоритмов за время порядка $O(n + m)$. Другие два не гарантируют такой скорости вычислений, однако обладают другими полезными свойствами, которые делают их конкурентоспособными в определенных ситуациях. В следующей главе мы систематизируем алгоритмы точного сравнения с паттернами, где будет представлено несколько десятков разработанных в последние 20 лет вариантов базовых алгоритмов, описанных в данной главе. Превосходный обзор алгоритмов сравнения с паттернами имеется на Web-узле [47], который включает краткое их описание, программы на языке C и аплеты Java.

7.1 Алгоритм Кнута–Морриса–Пратта

Основная идея алгоритма КМП (так для краткости будем называть алгоритм Кнута–Морриса–Пратта, Knuth–Morris–Pratt, [137]) заключается в последовательном сдвиге паттерна $\boldsymbol{p}$ вдоль строковой последовательности $\boldsymbol{x}$ и сравнении его

с просматриваемым участком последовательности $\boldsymbol{x}$ так, как это делается в простом алгоритме 2.2.1 (см. раздел 2.2). При этом паттерн $\boldsymbol{p}$ сдвигается на следующую букву вдоль строки $\boldsymbol{x}$ только в том случае, когда $\boldsymbol{p}$ не совпадает с текущим участком последовательности $\boldsymbol{x}$.

Сначала рассмотрим этот алгоритм на примере, который уже использовался в разделе 2.2: $\boldsymbol{x} = a^n$, $\boldsymbol{p} = a^{m-1}b$, $2 \leqslant m \leqslant n$. В этом примере после сравнения $a^{m-1}b$ с подстрокой $\boldsymbol{x}[1..m]$ мы будем располагать информацией, что

$$\boldsymbol{x}[1..m-1] = \boldsymbol{p}[1..m-1] = a^{m-1}.$$

Чтобы проверить, будет ли паттерн $\boldsymbol{p}$ совпадать с подстрокой $\boldsymbol{x}[2..m+1]$, нет необходимости проводить сравнения в подстроке $\boldsymbol{x}[2..m-1]$, поскольку уже известно, что $\boldsymbol{x}[2..m-1] = a^{m-2}$. Поэтому следует провести только одно сравнение $\boldsymbol{x}[m] : \boldsymbol{p}[m-1]$, и в случае равенства этих букв, мы фиксируем несовпадение

$$\boldsymbol{x}[m+1] \neq b = \boldsymbol{p}[m].$$

Применяя эту стратегию в каждой позиции $i \leqslant n - m + 1$, можно показать (см. упражнение 7.1.4), что выполняется всего $2n - m$ буквенных сравнений, а не $m(n - m + 1)$, как можно было ожидать.

На этом примере показана основная особенность алгоритма КМП [184, 137]: сравнения

$$\boldsymbol{x}[i..i+h-1] = \boldsymbol{p}[1..h], 1 \leqslant h \leqslant m,$$

можно (частично или полностью) использовать для избежания дальнейшей проверки подстроки $\boldsymbol{u} = \boldsymbol{x}[i..i+h-1]$. Если паттерн $\boldsymbol{p}$ входит в строку $\boldsymbol{x}$ в некоторой позиции $i' \in i..i+h-1$, тогда строка $\boldsymbol{u} = \boldsymbol{p}[1..h]$ имеет непустой собственный префикс $\boldsymbol{p}[1..h']$, $h' = i + h - i'$, а также суффикс $\boldsymbol{x}[i'..i+h-1]$. Другими словами, подстрока $\boldsymbol{p}[1..h']$ должна быть *гранью* строки $\boldsymbol{u} = \boldsymbol{p}[1..h]$!. Эта ситуация проиллюстрирована на рис. 7.1.

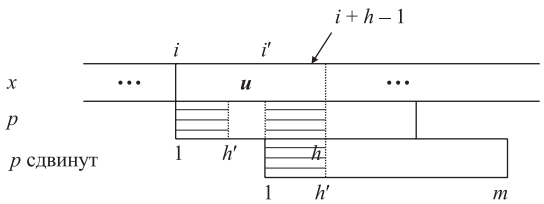


Рис. 7.1. Грань строки $\boldsymbol{p}[1..h]$

Таким образом, с помощью граней строки $\boldsymbol{u}$ надо определить только те позиции i', в которых может встретиться $\boldsymbol{p}$. Но поскольку $\boldsymbol{u}$ — это подстрока $\boldsymbol{p}[1..h]$ паттерна $\boldsymbol{p}$ для некоторого $h \in 1..m$, то нет необходимости вычислять грани строки

$\boldsymbol{u}$ в процессе выполнения алгоритма — можно вычислить их *заранее* и сохранить в массиве граней $\boldsymbol{\beta} = \boldsymbol{\beta}[1..m]$. Поэтому алгоритм КМП имеет предварительный этап, на котором вычисляется массив $\boldsymbol{\beta}$. Такие вычисления можно выполнить за время $\Theta(m)$, если использовать для этого алгоритм 1.3.1.

Фактически надо предварительно вычислить не сам массив граней, а его более слабый вариант. Обозначим через j позицию в паттерне $\boldsymbol{p}$ первого несовпадения. Тогда префикс паттерна $\boldsymbol{p}$, который не требует проверки при следующем сравнении, — это наибольшая грань подстроки $\boldsymbol{p}[1..j-1]$. Поэтому следующей буквой для сравнения будет $\boldsymbol{p}[\boldsymbol{\beta}[j-1]+1]$, где $\boldsymbol{\beta}[j-1]$ — длина наибольшей грани подстроки $\boldsymbol{p}[1..j-1]$, при этом, конечно, $\boldsymbol{\beta}[j-1] < j-1$. Следовательно, в действительности для всех $j \in 2..m+1$ надо вычислить $\boldsymbol{\beta}'[j] = \boldsymbol{\beta}[j-1]+1$, положив при этом $\boldsymbol{\beta}'[1] = 0$. Отметим, что для $j \geqslant 2$ $\boldsymbol{\beta}[j-1]+1 < (j-1)+1 = j$, поэтому $\boldsymbol{\beta}'[j] < j$. Если $j = 1$, тогда сравнивать нечего, и в этом случае необходимо изменить значение i, т.е. сдвинуть паттерн вдоль строки $\boldsymbol{x}$. После этих пояснений мы можем записать алгоритм КМП (см. следующий листинг). Заметим, что оператор присвоения $j \leftarrow \boldsymbol{\beta}'[j]$, выполняемый в результате несовпадения $\boldsymbol{p}[j] \neq \boldsymbol{x}[i]$, соответствует сдвигу вправо вдоль строки $\boldsymbol{x}$ паттерна p на $j - \boldsymbol{\beta}'[j]$ позиций.

Алгоритм 7.1.1 (Алгоритм Кнута–Морриса–Пратта)

▷ *Нахождение всех вхождений паттерна* $\boldsymbol{p}$ *в строку* $\boldsymbol{x}$
$i \leftarrow 1$; $j \leftarrow 1$
while $i \leqslant n - m + j$ **do**
  ▷ $(i, j) \leftarrow match(i, j, m)$
  if $j = m + 1$ **then output** $i - m$
  if $j = 1$ **then** $i \leftarrow i + 1$
  else $j \leftarrow \boldsymbol{\beta}'[j]$

Отметим, что в этом алгоритме процедура *match* (сравнить) выполняется так же, как и ее двойник в алгоритме 2.2.1, т.е. она выполняет сравнение "буква за буквой" слева направо. Однако есть между ними и небольшое отличие: в алгоритме 2.2.1 процедура *match* необязательно начинает сравнение с первой буквы $\boldsymbol{p}[1]$, тогда как в алгоритме КМП она начинает сравнение с буквы $\boldsymbol{x}[i]$ с заданной позиции j, $1 \leqslant j \leqslant m$, т.е. с буквы $\boldsymbol{p}[j]$. Кроме того, здесь она возвращает как значение i, так и значение j, показывающее позицию несовпадения букв в $\boldsymbol{x}$ и $\boldsymbol{p}$.

Обсуждение алгоритма КМП мы подытоживаем в следующей теореме.

Теорема 7.1.1. Алгоритм 7.1.1 корректно вычисляет все вхождения заданного непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$. ■

Чтобы оценить сложность алгоритма КМП, подсчитаем количество буквенных сравнений, выполняемых в процедуре *match*. Если результат сравнения показывает равенство $\boldsymbol{x}[i] = \boldsymbol{p}[j]$, тогда значения переменных i и j увеличиваются на еди-

ницу, при этом паттерн $\boldsymbol{p}$ не сдвигается относительно $\boldsymbol{x}$. Если результат сравнения показывает неравенство $\boldsymbol{x}[i] \neq \boldsymbol{p}[j]$, тогда для случая $j = 1$ значение i увеличивается на единицу и паттерн $\boldsymbol{p}$ сдвигается вправо на одну позицию, а в случае $j > 1$ значение i не меняется, паттерн $\boldsymbol{p}$ сдвигается вправо на $j - \boldsymbol{\beta}'[j] \geqslant 1$ позиций. Заметим, что переменная i может возрастать не более n раз, тогда как паттерн $\boldsymbol{p}$ сдвигается вправо не более $n - m + 1$ раз. Однако эти две верхние границы не могут достигаться одновременно, поскольку цикл **while** закончится как только выполнится хотя бы одно из этих условий. Следовательно, общее количество буквенных сравнений не может превысить величины

$$n + (n - m + 1) - 1 = 2n - m.$$

Поскольку все другие операции в цикле **while** выполняются за константное время, мы приходим к следующей теореме.

Теорема 7.1.2. Алгоритм 7.1.1 выполняется за время порядка $O(n)$ с использованием дополнительной памяти объемом $\Theta(m)$, при этом выполняется не более $2n - m$ сравнений с элементами строки $\boldsymbol{x}$. ■

Пример с $\boldsymbol{x} = a^n$ и $\boldsymbol{p} = a^{m-1}b$ показывает, что верхняя граница числа буквенных сравнений наименьшая из возможных. Если к времени выполнения алгоритма 7.1.1 добавить время $\Theta(m)$ выполнения предварительного этапа, то получим полное время порядка $\Theta(n+m)$, необходимое для поиска всех вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$. В контексте обсуждения 1.2.1 заметим, что эта верхняя временна́я граница зависит от возможности вывода каждого вхождения $\boldsymbol{p}$ за константное время, используя для локализации $\boldsymbol{p}$ в строке $\boldsymbol{x}$ только целые числа.

Рассмотрим также поучительный пример с $\boldsymbol{x} = a^n$ и $\boldsymbol{p} = a^m$, где паттерн $n - m + 1$ раз входит в строку $\boldsymbol{x}$. Поскольку паттерн $\boldsymbol{p}$ имеет грань a^{m-1}, то $\boldsymbol{\beta}'[m+1] = m$, и поэтому, для того чтобы определить каждое вхождение паттерна $\boldsymbol{p}$ после первого, требуется всего одно буквенное сравнение $\boldsymbol{p}[m] : \boldsymbol{x}[i]$. Этот пример показывает замечательное свойство алгоритма КМП: необходимо только n буквенных сравнений для вывода $n - m + 1$ локализаций вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$ — другими словами, если значение n растет значительно быстрее m, то имеем константное время нахождения одного вхождения паттерна. Как мы увидим в следующем разделе, этот пример (периодический паттерн с малым периодом входит много раз в строковую последовательность) доставит большие трудности алгоритму Бойера–Мура, основному конкуренту алгоритма КМП.

Описав общий алгоритм КМП, можно приступать к его усовершенствованию, точнее, к усовершенствованию вычисления вспомогательного массива $\boldsymbol{\beta}'$. Напомним, что в алгоритме 7.1.1 ситуация, когда $\boldsymbol{x}[i] \neq \boldsymbol{p}[j]$, $j > 1$, ведет к выполнению оператора $j \leftarrow \boldsymbol{\beta}'[j]$. Но выполнение этого оператора бесполезно, если $\boldsymbol{p}[j] = \boldsymbol{p}[\boldsymbol{\beta}'[j]]$, поскольку тогда на следующей итерации по-прежнему будет выполняться неравенство $\boldsymbol{x}[i] \neq \boldsymbol{p}[j]$! В действительности нас интересует наибольшая

грань подстроки $\boldsymbol{p}[1..j-1]$, за которой *не следует* буква $\boldsymbol{p}[j]$ (если, конечно, такая грань существует). Другими словами, надо найти наибольшее целое число j' (если оно существует), такое, что

- подстрока $\boldsymbol{p}[1..j'-1]$ является гранью строки $\boldsymbol{p}[1..j-1]$;
- подстрока $\boldsymbol{p}[1..j']$ *не является* гранью строки $\boldsymbol{p}[1..j]$.

Это подводит нас к построению нового вспомогательного массива $\boldsymbol{\beta}''$, j-й элемент которого вычисляется по следующим правилам.

> Если $j = m+1$, то $\boldsymbol{\beta}''[j] \leftarrow \boldsymbol{\beta}'[j]$. Для значений $j \in 1..m$ $\boldsymbol{\beta}''[j] \leftarrow j'$, где $j'-1$ — длина наибольшей грани подстроки $\boldsymbol{p}[1..j-1]$, такой, что $\boldsymbol{p}[j'] \neq \boldsymbol{p}[j]$. Если такой грани не существует, то $\boldsymbol{\beta}''[j] \leftarrow 0$.

Напомним (см. раздел 1.3), что грани подстроки $\boldsymbol{p}[1..j-1]$ можно определить посредством их длин, которые являются элементами убывающего множества

$$S_{j-1} = \{\boldsymbol{\beta}[j-1], \boldsymbol{\beta}^2[j-1], \ldots, \boldsymbol{\beta}^k[j-1]\}, \tag{7.1}$$

где k — целое число, такое, что $\boldsymbol{\beta}^k[j-1] = 0$ (число k определяется единственным образом). Тогда для любого $j \in 2..m$ ненулевое значение j' элемента $\boldsymbol{\beta}''[j]$ возможно только в том случае, если существует значение (длина) $j'-1 \in S_{j-1}$, которое удовлетворяет условиям, перечисленным выше.

Отметим, что приведенное выше определение элементов массива $\boldsymbol{\beta}''$ применимо даже тогда, когда длина $j'-1$ наибольшей подходящей грани равна нулю. В этом случае для некоторого $j < m+1$ и *любой* грани длиной $j'-1$ выполняется равенство $\boldsymbol{p}[j'] = \boldsymbol{p}[j]$. Особое значение $\boldsymbol{\beta}''[j] = 0$ показывает, что следующее сравнение будет выполняться между буквами $\boldsymbol{x}[i+1]$ и $\boldsymbol{p}[1]$, т.е. это тот случай, когда процедура *match* возвращает $j = 1$. Детали, связанные с минимальными изменениями в алгоритме 7.1.1, необходимые для преобразования массива $\boldsymbol{\beta}$ (или массива $\boldsymbol{\beta}'$) в массив $\boldsymbol{\beta}''$, показаны в упражнениях 7.1.9 и 7.1.10. Назовем таким образом модифицированный алгоритм 7.1.1 алгоритмом КМП*. В упражнении 7.1.11 предложено показать, что алгоритм КМП* в самом худшем случае выполняется так же, как и алгоритм КМП. Поэтому справедлива такая теорема.

Теорема 7.1.3. Алгоритм КМП* корректно вычисляет все вхождения заданного непустого паттерна $\boldsymbol{p}[1..m]$ в заданную строку $\boldsymbol{x}[1..n]$ за время порядка $O(n)$ с использованием дополнительной памяти объемом $\Theta(m)$, при этом выполняется не более $2n - m$ буквенных сравнений. ■

Интересно посмотреть, какой вид имеют массивы $\boldsymbol{\beta}$, $\boldsymbol{\beta}'$ и $\boldsymbol{\beta}''$ для строк Фибоначчи. Мы увидим, что здесь различия между массивами $\boldsymbol{\beta}'$ и $\boldsymbol{\beta}''$ весьма значительны. В частности, $\boldsymbol{\beta}''[j] = 0$ для всех позиций j вхождения буквы a, следующей за буквой b, поскольку каждая строка Фибоначчи начинается с буквы a

и не содержит подстрок bb. Для строки Фибоначчи $\boldsymbol{f}_7$ массивы $\boldsymbol{\beta}$, $\boldsymbol{\beta}'$ и $\boldsymbol{\beta}''$ имеют следующий вид.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$\boldsymbol{f}_7 =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	
$\boldsymbol{\beta} =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	
$\boldsymbol{\beta}' =$	0	1	1	2	2	3	4	3	4	5	6	7	5	6	7	8	9	10	11	12	8	9
$\boldsymbol{\beta}'' =$	0	1	0	2	1	0	4	0	2	1	0	7	1	0	4	0	2	1	0	12	0	9

Алгоритмы КМП и КМП* просты, элегантны и интересны, поэтому они стимулировали многочисленные исследования методов вычисления внутренних паттернов в строковых последовательностях. Однако эти алгоритмы, к сожалению, не эффективны: показано, как теоретически [22], так и практически [212, 76], что алгоритм КМП* в среднем лишь чуть быстрее простого алгоритма 2.2.1, хотя в самом худшем случае он, конечно, выполняется значительно быстрее. В следующем разделе мы покажем другие подходы построения алгоритмов сравнения с паттернами, которые эффективны в среднем.

Упражнения 7.1

1. Напишите процедуру *match* для алгоритма КМП и докажите ее корректность.
2. Предположим, что строки $\boldsymbol{x}$ и $\boldsymbol{p}$ заканчиваются специальными буквами (которые называются ***сигнальными метками*** (sentinels)) $\$_1$ и $\$_2$ соответственно. Эти буквы различны между собой и не совпадают ни с одной буквой алфавита. Перепишите процедуру *match* так, чтобы она могла распознавать и использовать сигнальные метки. Оцените эффективность новой процедуры *match* по сравнению с аналогичной процедурой из предыдущего упражнения.
3. Предположим, что алгоритм КМП применяется к строке $\boldsymbol{x} = a^n$ и паттерну $\boldsymbol{p} = a^m$, $1 \leqslant m \leqslant n$, с использованием

   а) процедуры *match* из предыдущего упражнения,

   б) процедуры *match* из первого упражнения.

   В тексте показано, что в случае *б*) для решения задачи необходимо n буквенных сравнений. Оцените количество буквенных сравнений, необходимых в случае *а*). На этом основании сделайте заключение об использовании минимального количества буквенных сравнений как критерия эффективности алгоритмов сравнения с паттернами.
4. Пусть алгоритм КМП (с исходной процедурой *match*) применяется для нахождения всех вхождений паттерна $\boldsymbol{p} = a^{m-1}b$ в строку $\boldsymbol{x} = a^n$, $2 \leqslant m \leqslant n$.

Покажите, что в этом случае алгоритм выполнит ровно $2n - m$ буквенных сравнений.

5. Докажите утверждение, сделанное в тексте, что в алгоритме КМП не могут одновременно достигаться верхняя граница увеличения переменной i и верхняя граница величины сдвига паттерна.

6. В тексте ничего не сказано о *нижней границе* количества буквенных сравнений, выполняемых в алгоритме КМП. Покажите, что эта нижняя граница равна $n - m + 1$. Что вы можете сказать о *среднем* количестве буквенных сравнений, выполняемых в алгоритме КМП, с учетом предварительной обработки паттерна $\boldsymbol{p}$?

7. Сколько буквенных сравнений потребуется алгоритму КМП для нахождения всех вхождений паттерна $\boldsymbol{p} = (\lambda_1\lambda_2 \ldots \lambda_k)^{m/k}$ в строку $\boldsymbol{x} = (\lambda_1\lambda_2 \ldots \lambda_k)^{n/k}$, если все буквы $\lambda_1, \lambda_2, \ldots, \lambda_k$ различны?

8. Докажите, что смещение $\boldsymbol{\beta}''[j]$, соответствующее несовпадению сравниваемых букв в позиции j, равно смещению $\boldsymbol{\beta}''[\boldsymbol{\beta}'[j]]$, соответствующему несовпадению букв в позиции $\boldsymbol{\beta}'[j]$, только в том случае, если $\boldsymbol{p}[j] = \boldsymbol{p}[\boldsymbol{\beta}'[j]]$. Отсюда будет следовать, что $\boldsymbol{\beta}''[j] = \boldsymbol{\beta}'[j]$ или $\boldsymbol{\beta}''[j] = \boldsymbol{\beta}''[\boldsymbol{\beta}'[j]]$.

9. Предположим, что к паттерну $\boldsymbol{p}$ справа добавлена уникальная буква \$. Модифицируйте алгоритм 1.3.1 таким образом, чтобы он за время порядка $\Theta(n)$ вычислял массив $\boldsymbol{\beta}'$. Другая модификация этого алгоритма за такое же время должна вычислять массив $\boldsymbol{\beta}''$. Докажите корректность новых алгоритмов, основываясь на предположении о корректности алгоритма 1.3.1.

Совет. Алгоритм для вычисления массива $\boldsymbol{\beta}'$ должен быть проще, чем алгоритм 1.3.1. На основе этого алгоритма, используя соотношения из предыдущего упражнения, несложно построить алгоритм для вычисления массива $\boldsymbol{\beta}''$. Можно ли вычислить массив $\boldsymbol{\beta}''$ без предварительного вычисления (и сохранения) массива $\boldsymbol{\beta}'$?

10. Измените алгоритм КМП так, чтобы он использовал массив $\boldsymbol{\beta}''$, а не массив $\boldsymbol{\beta}'$.

11. Вычислите массив $\boldsymbol{\beta}''$ для паттерна $\boldsymbol{p} = a^{m-1}b$. Затем покажите, что в самом худшем случае алгоритм КМП* требует не более $2n - m$ буквенных сравнений.

12. Вычислите массивы $\boldsymbol{\beta}'$ и $\boldsymbol{\beta}''$ для

а) $\boldsymbol{p} = abcdabce$;

б) $\boldsymbol{p} = abcabcacab$;

в) $\boldsymbol{p} = abacabadacabae$.

Существуют ли строки, для которых $\boldsymbol{\beta}'' = \boldsymbol{\beta}'$?

7.2 Алгоритм Бойера–Мура

Подобно алгоритму КМП, алгоритм БМ (так для краткости будем называть алгоритм Бойера–Мура, Boyer–Moore, [38]) сдвигает паттерн $\boldsymbol{p}$ вдоль текстовой строки $\boldsymbol{x}$ слева направо. Однако, в отличие от алгоритма КМП, в алгоритме БМ буквы в паттерне $\boldsymbol{p}$ сравниваются *справа налево*, т.е. в порядке $\boldsymbol{p}[m]$, $\boldsymbol{p}[m-1]$, $\ldots$, $\boldsymbol{p}[1]$. Как мы увидим далее, это, на первый взгляд, незначительное изменение коренным образом изменит стратегию определения, в случае несовпадения сравниваемых букв, следующего положения паттерна $\boldsymbol{p}$ относительно строки $\boldsymbol{x}$. Как и алгоритм КМП, данный алгоритм зависит от предварительно вычисленных массивов, которые позволяют определить каждый сдвиг паттерна $\boldsymbol{p}$ за константное время. Далее мы увидим, что в алгоритме БМ такие массивы тесно связаны с массивом $\boldsymbol{\beta}''$, описанным в предыдущем разделе.

Прежде чем детально описывать алгоритм БМ, рассмотрим присущие ему ограничения. На практике встречается много задач сравнения с паттерном, которые чувствительны к этим ограничениям.

- Один из предварительно вычисляемых массивов алгоритма БМ (обычно этот массив обозначается как $\boldsymbol{\delta}_1$) зависит от знания вхождений в паттерн $\boldsymbol{p}$ каждой буквы алфавита строки $\boldsymbol{x}$. Поэтому алфавит строки $\boldsymbol{x}$ должен быть конечным и известным заранее.
- Доступ к элементам массива $\boldsymbol{\delta}_1$ основан не на индексах элементов, а на буквах, т.е. к элементам этого массива обращаются как $\boldsymbol{\delta}_1[\lambda]$. Для того чтобы время доступа к этим элементам было константное, необходимы предварительные вычисления для определения буквы λ. Это означает, что алфавит должен быть *индексированным* (терминология раздела 4.1). Напомним, что если алфавит упорядочен, но не индексированный, то может потребоваться время порядка $\Theta(\log m)$ для определения буквы λ в паттерне $\boldsymbol{p}[1..m]$ и такое же время для доступа к элементам массива $\boldsymbol{\delta}_1$. Иногда это может быть существенным ограничением для применения алгоритма БМ.
- Поскольку паттерн $\boldsymbol{p}$ просматривается справа налево, то необходимо организовать доступ к элементам паттерна в обратном порядке. Как указывалось в обсуждении 1.2.1, может потребоваться линейное время для копирования элементов паттерна $\boldsymbol{p}$ в новый массив в обратном порядке.

Теперь приведем листинг алгоритма БМ, а затем рассмотрим этот алгоритм подробно.

Алгоритм 7.2.1 (Алгоритм Бойера–Мура)

▷ *Нахождение всех вхождений паттерна* $\boldsymbol{p}$ *в строку* $\boldsymbol{x}$
$i \leftarrow m$
while $i \leqslant n$ **do**
  $(i, j) \leftarrow hctam(i, m)$

if $j = 0$ **then output** $i + 1$
$i \leftarrow i + \max\{\boldsymbol{\delta}_1[\boldsymbol{x}[i]], \boldsymbol{\delta}_2[j]\}$

Надеюсь, что читатель простит мне некоторую вольность за присвоение имени *hctam*[1] процедуре, которая сравнивает паттерн $\boldsymbol{p}$ с буквами строки $\boldsymbol{x}$, выполняя это сравнение справа налево. Отметим, что здесь сравнение с $\boldsymbol{p}$ заканчивается, когда переменная j, первоначально равная m, уменьшается до нуля, тогда как в алгоритме КМП переменная j возрастала до значения $m + 1$ начиная с некоторого начального значения — с этой точки зрения процедура *hctam* немного проще процедуры *match*, поскольку не зависит от предыдущего значения j. Как видно, алгоритм БМ прост и корректен, если, конечно, корректно работает процедура *hctam* и корректно вычисляется величина сдвига паттерна (что требует корректного вычисления массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$). Обобщим это утверждение в виде теоремы.

Теорема 7.2.1. При условии корректной реализации процедуры *hctam* и массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$ алгоритм 7.2.1 корректно вычисляет все вхождения данного непустого паттерна $\boldsymbol{p}[1..m]$ в заданную строку $\boldsymbol{x}[1..n]$. ■

Основная идея массива $\boldsymbol{\delta}_1$ проста. Мы предполагаем, что процедура *hctam* находит совпадение (возможно, на пустых подстроках)

$$\boldsymbol{x}[i + 1..i + h] = \boldsymbol{p}[m - h + 1..m]$$

для некоторого $i \in 0..n - h$ и некоторого $h \in 0..m$ и затем возвращает пару чисел $(i, j) = (i, m-h)$. По соглашению положим, что строка $\boldsymbol{x}$ и паттерн $\boldsymbol{p}$ начинаются с сигнальных меток, т.е. $\boldsymbol{x}[0] = \$_1$ и $\boldsymbol{p}[0] = \$_2$. Тогда можно утверждать, что для любого случая, включая $i = 0$ и $j = 0$, несовпадение

$$\lambda = \boldsymbol{x}[i] \neq \boldsymbol{p}[j] = \mu \tag{7.2}$$

будет найдено. Сделаем очевидное наблюдение: если какая-либо подстрока строки $\boldsymbol{x}$, содержащая позицию i, совпадает с паттерном $\boldsymbol{p}$, тогда в $\boldsymbol{p}$ входит буква $\lambda = \boldsymbol{x}[i]$. Далее, если мы знаем позицию (обозначим ее j') самого правого вхождения буквы λ в паттерн $\boldsymbol{p}$, тогда мы можем определить *минимальный* сдвиг $\boldsymbol{p}$, удовлетворяющий условию, что $\boldsymbol{x}[i] = \lambda$. Как показано на рис. 7.2, в этом случае $j' < j$, что ведет к сдвигу паттерна на $j - j'$ позиций. Тогда при следующем вызове процедуры *hctam* она начнет работу со сравнения букв $\boldsymbol{p}[m]$ и $\boldsymbol{x}[i + (m - j')]$. Таким образом, в этом случае надо изменить значение i:

$$i \leftarrow i + (m - j'). \tag{7.3}$$

[1] Если читатель еще не догадался, то *hctam* — это перевернутое слово *match*. — *Примеч. пер.*

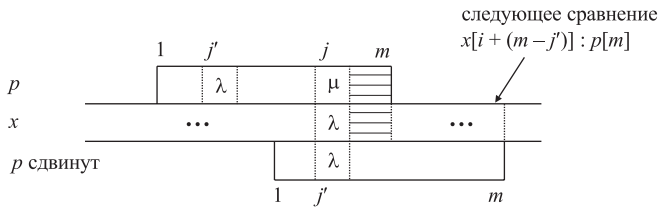


Рис. 7.2. Паттерн сдвинут так, чтобы буква λ в строке $\boldsymbol{x}$ совпала с самым правым вхождением этой буквы в паттерн

Итак, очевидно, что для каждой буквы λ (включая $\$_1$) алфавита A необходимо вычислить позицию j' ее самого правого вхождения в паттерн $\boldsymbol{p}$, положив $j' = 0$, если данная буква λ не входит в $\boldsymbol{p}$. Тогда на предварительном этапе массив $\boldsymbol{\delta}_1$ вычисляется как

$$\boldsymbol{\delta}_1[\lambda] \leftarrow m - j'. \tag{7.4}$$

Теперь, если обнаружится несовпадение (7.2) и при этом *выполняется неравенство* $j' < j$ (т.е. $\boldsymbol{\delta}_1[\lambda] > m - j$), следующую позицию для сравнения можно вычислить так, как это сделано в алгоритме 7.2.1: $i \leftarrow i + \boldsymbol{\delta}_1[\lambda]$.

Использование массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$

"Все это хорошо, — подумает с некоторым сомнением внимательный читатель, — но что делать в случае $\boldsymbol{\delta}_1[\lambda] < m - j$?" Ответ прост: такая ситуация возможна только для $j < m$, но тогда *обязательно* $j \leqslant 1$. Все, что мы можем сделать в этой ситуации, — это сдвинуть паттерн $\boldsymbol{p}$ вправо на одну позицию так, чтобы следующее сравнение начиналось со сравнения буквы $\boldsymbol{p}[m]$ с буквой $\boldsymbol{x}[i + (m - j) + 1]$. Это эквивалентно присваиванию

$$i \leftarrow i + (m - j) + 1. \tag{7.5}$$

Сравнивая присваивания (7.3) и (7.5), находим, что в случае $j' > j$ (т.е. в случае $\boldsymbol{\delta}_1[\lambda] < m - j$) выполняется неравенство $(m - j) + 1 > m - j'$. Но, как мы покажем ниже, для всех j справедливо неравенство

$$\boldsymbol{\delta}_2[j] \geqslant (m - j) + 1. \tag{7.6}$$

Поэтому присвоение (7.5) теряет необходимость, если использовать присвоение

$$i \leftarrow i + \max\{\boldsymbol{\delta}_1[\boldsymbol{x}[i]], \boldsymbol{\delta}_2[j]\}$$

из алгоритма 7.2.1.

Подобно массиву $\boldsymbol{\delta}_1$, массив $\boldsymbol{\delta}_2 = \boldsymbol{\delta}_2[0..m]$ "вступает в игру" после того, как определено совпадение подстрок

$$\boldsymbol{x}[i + 1..i + h] = \boldsymbol{p}[m - h + 1..m]$$

для некоторого $i \in 0..n-h$ и некоторого $h \in 0..m$ и затем определено несовпадение (7.2). (Здесь, как и ранее, $j = m - h$.) Но с другой стороны, вычисление массива $\boldsymbol{\delta}_2$ имеет много общего с вычислением массива $\boldsymbol{\beta}''$ из алгоритма КМП*, поскольку этот массив так или иначе связан с вычислением граней, но с привлечением дополнительных ограничений на грани. Если быть более точным, то после определения совпадения подстрок необходимо найти наибольшее целое j' ($j' < j$), такое, чтобы выполнялось совпадение подстрок

$$\boldsymbol{p}[j'+1..j'+h] = \boldsymbol{p}[j+1..m]$$

и при этом $\mu' = \boldsymbol{p}[j'] \neq \boldsymbol{p}[j] = \mu$. Если такое j' существует, то происходит присвоение

$$i \leftarrow i + (m - j'), \tag{7.7}$$

как показано на рис. 7.3. Конечно, величину сдвига $\boldsymbol{\delta}_2[j] = m - j'$ можно вычислить заранее для любого $j \in 1..m$. Обращаем ваше внимание на определенную схожесть рис. 7.2 и 7.3.

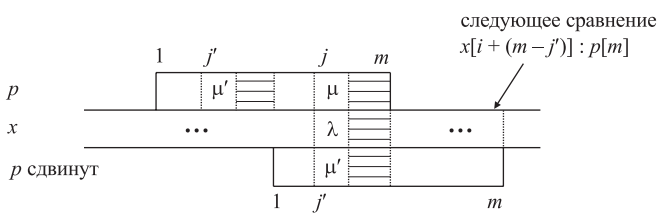


Рис. 7.3. Сдвиг первого типа

Если такого j', удовлетворяющего перечисленным условиям, не существует (включая случай $j = 0$), тогда ищется такое наибольшее целое j', что подстрока $\boldsymbol{p}[1..j']$ является суффиксом строки $\boldsymbol{p}[j+1..m]$, т.е. $\boldsymbol{p}[1..j']$ должна быть наибольшей гранью паттерна $\boldsymbol{p}$, при этом $j' \leqslant h$. В этом случае, как показано на рис. 7.4, необходимо выполнить присвоение

$$i \leftarrow i + (m - j) + (m - j'). \tag{7.8}$$

Снова величины $\boldsymbol{\delta}_2[j] = (m-j)+(m-j')$ можно вычислить заранее для любого j.

Необходимо потребовать, чтобы вычисление величин $\boldsymbol{\delta}_2[j]$, выполненное согласно рис. 7.3 и 7.4, удовлетворяло следующим двум условиям.

- Вычисление $\boldsymbol{\delta}_2[j]$ должно быть *осторожным*, т.е. сдвиг паттерна должен быть минимальным при выполнении условия, что собственная подстрока паттерна совпадает с подстрокой $\boldsymbol{x}[i..i+h-1]$ (рис. 7.3) или что собственный префикс паттерна совпадает с суффиксом подстроки $\boldsymbol{x}[i..i+h-1]$ (рис. 7.4). Минимум сдвига гарантирован тем, что, в соответствии с присвоениями (7.7) и (7.8), выбирается наибольшее значение j'.

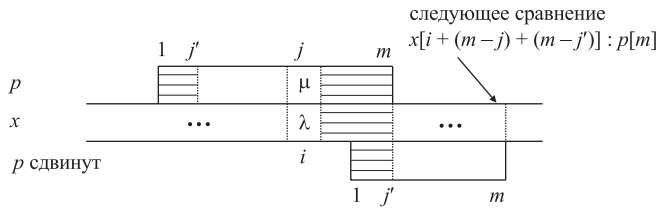


Рис. 7.4. Сдвиг второго типа

- Для любого j должно выполняться неравенство (7.6) $\boldsymbol{\delta}_2[j] \geqslant (m-j)+1$. Отсюда вытекает присвоение (7.7) при $j' < j$ и присвоение (7.8) при $m-j' \geqslant 1$. Поэтому достаточно сделать сдвиг на величину $\max\{\boldsymbol{\delta}_1[\lambda], \boldsymbol{\delta}_2[j]\}$. Отметим, что такая формула сдвига гарантирует, что на каждом шаге алгоритма БМ паттерн будет сдвигаться не менее, чем на одну позицию, т.е. алгоритм обязательно завершится.

Мы показали, как используются массивы $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$, и убедились, что при таком их использовании алгоритм 7.2.1 будет выполняться корректно.

Вычисление массива δ_2

Теперь вернемся к вычислению массива $\boldsymbol{\delta}_2$. Здесь существенно, что позиция j' вычисляется как функция от заданной позиции $j \in 0..m$. Эти вычисления разбиваются на две части. Первая часть вычислений, которая соответствует сдвигу ***первого типа***, показанного на рис. 7.3, выполняется только для тех значений j, для которых существует соответствующее значение j' (другими словами, эта часть вычислений выполняется тогда, когда существует соответствующая грань, которой предшествует буква μ', не равная μ). Вторая часть вычислений, которая соответствует сдвигу ***второго типа***, показанного на рис. 7.4, выполняется для тех значений j, соответствующие значения j' которых не могут быть вычислены в первой части вычислений. Чтобы понять различие между этими частями вычислений, предположим, что в результате вычислений первого типа получаем массив $\boldsymbol{g}_I = \boldsymbol{g}_I[1..m]$, где $j' = \boldsymbol{g}_I[j]$ — значение, соответствующее значению j (если, конечно, такое значение j' существует), а в результате вычислений второго типа получаем массив $\boldsymbol{g}_{II} = \boldsymbol{g}_{II}[1..m]$. Вычисление массива $\boldsymbol{g}_{II}$ проще, поэтому начнем рассмотрение с этого массива.

Предположим, что с помощью алгоритма 1.3.1 за линейное время вычислен массив граней $\boldsymbol{\beta} = \boldsymbol{\beta}[1..m]$ паттерна $\boldsymbol{p}[1..m]$. Для данного значения $j \in 1..m$, как ясно из рис. 7.4, значение $j' = \boldsymbol{g}_{II}[j]$ равно длине наибольшей грани паттерна, причем эта длина не должна превышать $h = m - j$. Чтобы определить это значение, надо просто иметь доступ к элементам убывающей последовательности

$$S_m = \langle \boldsymbol{\beta}[m], \boldsymbol{\beta}^2[m], \ldots, \boldsymbol{\beta}^k[m] = 0 \rangle,$$

определенной в разделе 1.3. Тогда для каждого j, такого, что $0 < j \leqslant m - \boldsymbol{\beta}[m]$, положим $\boldsymbol{g}_{II}[j] = \boldsymbol{\beta}[m]$. Для значений j из интервала $m - \boldsymbol{\beta}[m] < j \leqslant m - \boldsymbol{\beta}^2[m]$ положим $\boldsymbol{g}_{II}[j] = \boldsymbol{\beta}^2[m]$ и т.д. Для $j = 0$ (если имеется совпадение с паттерном) положим $\boldsymbol{g}_{II}[j] = \boldsymbol{\beta}[m]$. Очевидно, что массив $\boldsymbol{g}_{II}$ можно вычислить за линейное время.

Для вычисления значений $\boldsymbol{g}_I[j]$, $1 \leqslant j \leqslant m$, необходимо найти наибольшее целое j' (если оно существует и меньше j), такое, что

- подстрока $\boldsymbol{p}[j'+1..j'+(m-j)]$ является гранью строки $\boldsymbol{p}[j'+1..m]$,
- подстрока $\boldsymbol{p}[j'..j'+(m-j)]$ *не является* гранью строки $\boldsymbol{p}[j'..m]$.

Если такое значение j' существует, тогда $\boldsymbol{g}_I[j] \leftarrow j'$, иначе $\boldsymbol{g}_I[j] \leftarrow 0$.

Чтобы лучше понять эти вычисления, рассмотрим грани суффиксов паттерна $\boldsymbol{p}$. Читатель с хорошей памятью, конечно же, помнит упражнение 1.3.10, где введено понятие "правого массива граней". Обозначим этот массив как $\boldsymbol{\rho} = \boldsymbol{\rho}[1..m]$, где $\boldsymbol{\rho}[j]$ — длина наибольшей грани подстроки $\boldsymbol{p}[j..m]$. Напомним (также из упражнения 1.3.10), что если $\boldsymbol{\beta}_T$ — массив граней транспонированной (обратной) строки

$$\boldsymbol{p}_T = \boldsymbol{p}[m]\boldsymbol{p}[m-1]\ldots p[1],$$

тогда $\boldsymbol{\beta}_T[m-i+1] = \boldsymbol{\rho}[i]$ для всех $i \in 1..m$. Следовательно, длины граней суффиксов $\boldsymbol{p}[i..m]$ являются элементами убывающей последовательности

$$S^T_{m-i+1} = \left\langle \boldsymbol{\beta}_T[m-i+1], \boldsymbol{\beta}^2_T[m-i+1], \ldots, \boldsymbol{\beta}^k_T[m-i+1] = 0 \right\rangle.$$

Тогда для вычисления элемента $\boldsymbol{g}_I[j]$ необходимо наибольшее целое число j' такое, что $j' < j$ и число $m-j$ принадлежит последовательности $S^T_{m-j'}$, но число $m-j+1$ не является элементом последовательности $S^T_{m-j'+1}$. Отметим, что если b^* — наибольший элемент массива $\boldsymbol{\beta}_T$, тогда для любого j, такого, что $m-j > b^*$, $\boldsymbol{g}_I[j] \leftarrow 0$.

Вычисление массива $\boldsymbol{g}_I$ выполняется аналогично, но не идентично, вычислению массива $\boldsymbol{\beta}''$ из раздела 7.1. Вместе с тем, используя те же аргументы, что и в разделе 7.1, можно показать, что вычисление массива $\boldsymbol{g}_I$ выполняется за время порядка $\Theta(m)$. Детали этих вычислений оставим для рассмотрения в упражнении 7.2.8, а здесь сформулируем основной результат этого подраздела.

Теорема 7.2.2. Для данного паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ массив $\boldsymbol{\delta}_2 = \boldsymbol{\delta}_2[0..m]$ можно вычислить за время порядка $\Theta(m)$. ■

Мы отсылаем читателя, заинтересовавшегося этими сложными вычислениями, к работе [119], где впервые доказано, что массив $\boldsymbol{\delta}_2$ можно корректно вычислить за линейное время, а также к недавней работе [60], где предложено более простое доказательство этих фактов.

Примеры применения алгоритма БМ

Чтобы лучше понять, как вычисляются и используются массивы $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$ в алгоритме БМ, рассмотрим пример строки Фибоначчи $\boldsymbol{f}_7$, представленный в предыдущем разделе.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\boldsymbol{f}_7 =$	\$$_2$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
$\boldsymbol{\beta} =$		0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8
$\boldsymbol{\rho} =$		8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1	3	2	1	0	0
$\boldsymbol{\beta}_T =$		0	0	1	2	3	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
$\boldsymbol{g}_I =$		0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	16	0	16	20
$\boldsymbol{g}_{II} =$	8	8	8	8	8	8	8	8	8	8	8	8	8	8	3	3	3	3	3	1	1	0
$\boldsymbol{\delta}_2 =$	34	33	32	31	30	29	28	27	26	25	24	23	22	13	25	24	23	22	5	22	5	1

Вычисление массива $\boldsymbol{\delta}_1$ просто: $\boldsymbol{\delta}_1[a] = 0$, $\boldsymbol{\delta}_1[b] = 1$ и для любой буквы λ, не равной a или b, $\boldsymbol{\delta}_1[\lambda] = 21$. Однако для поиска вхождения в строки Фибоначчи текстовых строк, состоящих только из букв a и b, массив $\boldsymbol{\delta}_1$ не играет никакой роли, поскольку в этом случае для всех $j \in 0..m$

$$\max\{\boldsymbol{\delta}_1[\lambda], \boldsymbol{\delta}_2[j]\} = \boldsymbol{\delta}_2[j]. \tag{7.9}$$

Но если паттерн будет содержать и другие буквы, отличные от букв a и b, тогда массив $\boldsymbol{\delta}_1$ необходим.

Чтобы проиллюстрировать важность обоих массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$, рассмотрим пример, приведенный в исходной работе [38], где впервые описан алгоритм БМ. В этом примере паттерн $p =$ `AT-THAT` сравнивается с текстовой строкой

$$\boldsymbol{x} = \cdots\texttt{WHICH-FINALLY-HALTS.--AT-THAT-POINT}\cdots .$$

В этом случае $\boldsymbol{\delta}_2[0..7] = 12\ 11\ 10\ 9\ 8\ 7\ 4\ 1$, тогда как $\boldsymbol{\delta}_1[\texttt{A}] = 1$, $\boldsymbol{\delta}_1[\texttt{T}] = 0$, $\boldsymbol{\delta}_1[\texttt{-}] = 4$, $\boldsymbol{\delta}_1[\texttt{H}] = 2$ и $\boldsymbol{\delta}_1[\lambda] = 7$ для любых других букв λ. Использование значений массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$ показано на приведенной ниже схеме выполнения алгоритма БМ, где вертикальными стрелками обозначены начальные позиции буквенных сравнений для процедуры *hctam*. Отметим, что здесь для локализации паттерна потребовалось всего 15 буквенных сравнений.

```
      ↓       ↓  ↓    ↓      ↓
WHICH-FINALLY-HALTS.--AT-THAT-POINT
AT-THAT                                  δ1[F] = 7, δ2[7] = 1
      AT-THAT                            δ1[-] = 4, δ2[7] = 1
          AT-THAT                        δ1[L] = 7, δ2[6] = 4
                AT-THAT                  δ1[-] = 4, δ2[5] = 7
                     AT-THAT             так быстро!
```

Эффективность алгоритма БМ

В отличие от алгоритма КМП, алгоритм БМ в значительно большей мере отличается в теоретическом плане от простого алгоритма 2.2.1 и, как показывает предыдущий пример, на практике выполняется также значительно быстрее. Практические [212] и теоретические [25, 28] исследования показывают, что среднее количество выполняемых алгоритмом БМ буквенных сравнений является сублинейной функцией от длины текстовой строки, даже если сюда приплюсовать буквенные сравнения, выполняемые на этапе вычисления массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$. Обычно для текста на естественном языке среднее количество буквенных сравнений составляет примерно $0{,}3n$, включая предварительную обработку паттерна, если длина паттерна больше 10, но намного меньше n. Для сравнения — алгоритм КМП в этой ситуации требует в среднем $1{,}0n$ буквенных сравнений (см. упражнение 7.1.6).

Чтобы среднее количество буквенных сравнений было сублинейным, для этого важную роль в алгоритме БМ играет использование массива $\boldsymbol{\delta}_1$, особенно для текстов на английском или на других языках, имеющих большие алфавиты. Однако для анализа самого худшего случая эффектом от использования массива $\boldsymbol{\delta}_1$ можно пренебречь, поскольку этот самый "худший случай" реализуется на бинарном алфавите. Тогда, как показывает пример со строкой Фибоначчи в предыдущем подразделе и пример со строкой $\boldsymbol{x} = (a^k b)^r$, рассмотренный ниже, всегда выполняется равенство (7.9). Поэтому в анализе самого худшего случая мы будем игнорировать использование массива $\boldsymbol{\delta}_1$.

К сожалению, в самом худшем случае, независимо от того, используется или нет массив $\boldsymbol{\delta}_1$, поведение алгоритма БМ не лучше, чем поведение простого алгоритма 2.2.1 — алгоритм БМ также может потребовать $m(n-m+1)$ буквенных сравнений. Этот факт может подтвердить пример $\boldsymbol{x} = a^n$, $\boldsymbol{p} = a^m$, который успешно решается алгоритмом КМП, но в алгоритме БМ для каждого из $n-m+$ $+\,1$ вхождений паттерна $\boldsymbol{p}$ требуется провести m буквенных сравнений, чтобы определить одно вхождение. Этот феномен отражает основное различие между алгоритмами КМП и БМ: если паттерн является кратной строкой, то в алгоритме КМП сравнения выполняются только на одном периоде паттерна, тогда как в алгоритме БМ сравнения производятся для каждой буквы паттерна. В разделе 8.4 будет показана модификация алгоритма БМ, которая гарантированно находит все вхождения даже периодических паттернов за линейное время. Но сейчас мы исследуем временну́ю сложность исходного алгоритма БМ.

Анализ поведения алгоритма БМ в самом худшем случае, даже если исключить из рассмотрения периодические паттерны и проигнорировать массив $\boldsymbol{\delta}_1$, является очень сложной задачей. В работе [137] в результате тяжелого доказательства показано, что этот алгоритм выполняет не более $7n$ буквенных сравнений для нахождения всех вхождений апериодического паттерна. Несколько лет спустя результатом другого трудного доказательства [104] стала верхняя граница в $4n$ буквенных сравнений. И только в 1994 году эта верхняя граница смогла

опуститься до величины примерно $3n$ [53]. Доказательство последнего результата также технически сложно и трудно. Поэтому мы представим собственное доказательство облегченного результата (которое, вместе с тем, будет основываться на доказательстве из работы [53]), что для апериодических паттернов количество буквенных сравнений не превышает $4n$.

Но прежде чем приступить непосредственно к доказательству этого утверждения, рассмотрим класс примеров, для которых требуется примерно $3n$ буквенных сравнений. Рассмотрим текстовую строку $\boldsymbol{x} = (a^k b)^r$, где k ($k \geqslant 2$) и r ($r \geqslant 1$) — целые числа. Применим алгоритм БМ для сравнения этой строки с паттерном $\boldsymbol{p} = a^{k-1}ba^{k-1}$. Этот паттерн входит ровно $r-1$ раз в строку $\boldsymbol{x}$, при этом для определения каждого вхождения требуется $2k-1$ буквенных сравнений. В упражнении 7.2.9 предложено показать, что при $r \geqslant 3$ алгоритм БМ выполнит ровно $(3k-1)(r-1)$ буквенных сравнений. Если k и r устремить к бесконечности, то отношение

$$\frac{(3k-1)(r-1)}{n} = \frac{(3k-1)(r-1)}{(k+1)r}$$

будет стремиться к 3. Следовательно, для любого действительного числа $\varepsilon > 0$ существуют задачи сравнения с паттернами текстовых строк длиной $n = n(\varepsilon)$, для решения которых алгоритм БМ выполнит более $3n - \varepsilon$ буквенных сравнений. Эти задачи, с учетом результата статьи [53], составляют существенную часть задач для "самого плохого случая" поведения алгоритма БМ.

Теперь рассмотрим "самый плохой случай" для апериодических паттернов. Но сначала введем некоторые новые обозначения. Обозначим через $\boldsymbol{t}$ подстроку строки $\boldsymbol{x}$, которую закончила обрабатывать процедура *hctam* на некотором шаге алгоритма БМ. Фактически $\boldsymbol{t} = \boldsymbol{p}[j+1..m]$ — суффикс паттерна $\boldsymbol{p}$ длиной $t = |\boldsymbol{t}| = m - j$. Заметим, что количество буквенных сравнений, выполненных при данном вызове процедуры *hctam*, равно $t+1$ (t сравнений, при которых произошло совпадение букв, и одно сравнение, конечное, когда сравниваемые буквы не совпали). Далее обозначим через s величину сдвига паттерна, вычисляемую только с использованием массива $\boldsymbol{\delta}_2$. Тогда или $s = (m-j') - (m-j) = j - j'$ (рис. 7.3), или $s = m - j'$ (рис. 7.4). Суффикс паттерна $\boldsymbol{p}$ длиной s обозначим как $\boldsymbol{s}$.

Рассмотрим строку $\boldsymbol{ts}$. При сдвиге первого типа (рис. 7.3) строка $\boldsymbol{ts} = \boldsymbol{p}[j'+1..m]$ имеет грань $\boldsymbol{p}[m-j]$ и, следовательно, период $s = (m-j') - (m-j)$. При сдвиге второго типа (рис. 7.4) строка $\boldsymbol{ts}$ также имеет грань длиной $m-j$, и поскольку $|\boldsymbol{ts}| = (m-j') - (m-j)$, то $\boldsymbol{ts}$ имеет период $s = m - j'$.

Обозначим через f количество позиций строки $\boldsymbol{x}$, обрабатываемых во время текущего вызова процедуры *hctam*, но такие, что буквы, стоящие в этих позициях, не встречались процедуре *hctam* ранее при ее предыдущих вызовах. Поскольку на каждом шаге алгоритма БМ паттерн сдвигается не менее, чем на одну позицию, то $f \geqslant 1$. Теперь покажем, что при каждом вызове процедуры *hctam* количество

$t + 1$ буквенных сравнений удовлетворяет неравенству

$$t + 1 \leqslant f + 3s \tag{7.10}$$

для случая, когда в нормальной форме паттерна $\boldsymbol{p} = \boldsymbol{u}^k\boldsymbol{u}'$ показатель степени $k < 3$. (В этом случае будем говорить, что паттерн ***p* не является 3-периодическим***.) Сколько в алгоритме БМ не производилось бы сравнений, ни сумма Σf, ни сумма Σs не могут превосходить n. Тогда для случая не 3-периодического паттерна на основании неравенства (7.10) мы сможем сделать вывод, что общее количество буквенных сравнений, выполненных во время всех вызовов процедуры *hctam*, не превысит величины $n + 3n = 4n$.

Далее будем различать ***длинный сдвиг*** паттерна, когда $3s \geqslant t$, и ***короткий сдвиг***, когда $3s < t$. Нетрудно заметить, что при сравнениях, приводящих к длинному сдвигу, неравенство (7.10) выполняется. Поэтому необходимо рассмотреть только те сравнения, выполняемые во время вызова процедуры *hctam*, которые приводят к короткому сдвигу паттерна.

Поскольку при сдвиге второго типа (рис. 7.4) предполагается, что паттерн $\boldsymbol{p}$ имеет период $m - j'$, короткий сдвиг требует выполнения условия

$$m - j' = s < t/3 = (m - j)/3 \leqslant m/3.$$

Поэтому паттерн $\boldsymbol{p}$ с необходимостью должен быть 3-периодическим, но для этого случая неравенство (7.10) мы не рассматриваем. Таким образом, для доказательства неравенства (7.10) надо исследовать только сдвиг $s = j - j'$ первого типа (рис. 7.3) при дополнительном условии, что $3(j - j') < m - j$. Такая ситуация короткого сдвига первого типа показана на рис. 7.5. Здесь мы можем убедиться, что суффикс $\boldsymbol{ts} = \boldsymbol{p}[j' + 1..m]$ имеет грань длиной $m - j$ и, следовательно, период $s = j - j'$.

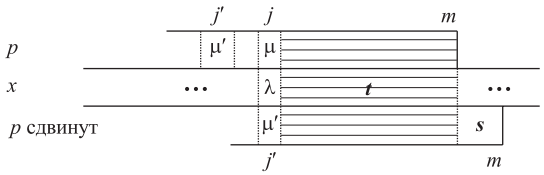


Рис. 7.5. Короткий сдвиг первого типа

Без потери общности мы можем предположить, что $\boldsymbol{s} = \boldsymbol{v}^k$, $k \geqslant 1$, где $\boldsymbol{v}$ — *некратная* строка. Тогда подстрока $\boldsymbol{t}$ строки $\boldsymbol{x}$ и суффикс $\boldsymbol{ts}$ паттерна $\boldsymbol{p}$ имеют период $v = |\boldsymbol{v}|$. Теперь можно записать правую нормальную форму (см. упражнение 1.2.24) для строки $\boldsymbol{ts}$: $\boldsymbol{ts} = \boldsymbol{v}'\boldsymbol{v}^r$, где $r = \lfloor (s+t)/v \rfloor$ и $\boldsymbol{v}'$ — возможно, пустой собственный суффикс строки $\boldsymbol{v}$, поскольку $\boldsymbol{t} = \boldsymbol{v}'\boldsymbol{v}^{r-k}$.

Имея полную характеристику текущего сравнения подстроки $\boldsymbol{t}$, рассмотрим сравнения, выполняемые процедурой *hctam* на предыдущих шагах алгоритма БМ.

Лемма 7.2.3. Если текущий сдвиг является сдвигом первого типа, тогда во время предыдущих выполнений процедуры *hctam*

а) буква $\boldsymbol{p}[m]$ не сравнивалась ни с одним вхождением $\boldsymbol{v}[v]$ в строке $\boldsymbol{t}$, т.е. не сравнивалась ни с одной буквой в позициях $\boldsymbol{t}[t-iv]$, $i=0,1,\ldots,r-k-1$;

б) между позициями паттерна $\boldsymbol{p}$ и строки $\boldsymbol{t}$ имело место не более v сравнений;

в) буква $\boldsymbol{p}[m]$ имела сравнения только с некоторыми буквами или в префиксе $\boldsymbol{t}[1..v-1]$ или в подстроке $\boldsymbol{t}[t-(v-1)..t-1]$.

Доказательство.

а) Предположим, что такое сравнение имело место. Поскольку при текущем вызове процедуры *hctam* $\boldsymbol{p}[m]$ сравнивается с $\boldsymbol{t}[t]$, самым правым вхождением $\boldsymbol{v}[v]$ в $\boldsymbol{t}$, то можно предположить, что буква $\boldsymbol{p}[m]$ ранее сравнивалась с $\boldsymbol{t}[t-iv]$ для некоторого целого $i \in 1..r-k-1$ и при этом было получено совпадение

$$\boldsymbol{p}[m-(t-iv)+1..m] = \boldsymbol{t}[1..t-iv],$$

а также несовпадение $\boldsymbol{p}[m-(t-iv)] = \mu \neq \lambda$, где, как и ранее, λ — буква, которая в строке $\boldsymbol{x}$ предшествует подстроке $\boldsymbol{t}$.

Зададим вопрос: какой сдвиг может быть в результате такого несовпадения? Определенно, это короткий сдвиг первого типа (рис. 7.5), поскольку достижимо значение j', используемое для текущего сравнения. При этом можно найти такое большое значение j', что $j' < j$. Однако отметим (см. рис. 7.5), что такой выбор значения j' ведет к такому сдвигу паттерна, что $\boldsymbol{p}[m]$ должно лежать правее $\boldsymbol{t}$. Но тогда текущие совпадения букв не будут иметь место. Получили противоречие. Делаем заключение, что буква $\boldsymbol{p}[m]$ не сравнивалась ранее с любым вхождением $\boldsymbol{v}[v]$ в строку $\boldsymbol{t}$.

б) Предположим, что $\boldsymbol{p}[m]$ выравнивалась для сравнения с некоторой буквой $\boldsymbol{t}[q]$, $q \in 1..t$. Как следует из утверждения *а*), позиция q не совпадает ни с одним вхождением $\boldsymbol{v}[v]$ в строку $\boldsymbol{t}$. Тогда может быть не более v сравнений паттерна $\boldsymbol{p}$ со строкой $\boldsymbol{t}$, причем последнее сравнение дало несовпадение сравниваемых букв, иначе строка $\boldsymbol{v}$ равна циклическому сдвигу самой себя и, следовательно, по теореме 1.4.2 является кратной строкой, что противоречит предположению о том, что строка $\boldsymbol{v}$ не кратна.

в) Предположим далее, что $q \notin 1..t$. Тогда q должно быть позицией в некоторой подстроке $\boldsymbol{v}''$ строки $\boldsymbol{v}$. Заметим, что после обнаружения несовпадения, $\boldsymbol{p}[m]$ сдвигается так, чтобы провести сравнение с $\boldsymbol{v}''[v]$. Тогда через конечное число выполнений процедуры *hctam* $\boldsymbol{p}[m]$ будет иметь сравнение с $\boldsymbol{v}[v]$, что противоречит утверждению *а*).

Лемма доказана. ■

Отметим, что этот результат не зависит от предположений о периодичности паттерна $\boldsymbol{p}$ — лемма справедлива для всех сдвигов первого типа. Мы используем эту лемму и предыдущее обсуждение для доказательства основного результата данного подраздела.

Теорема 7.2.4. Если паттерн $\boldsymbol{p}$ не является 3-периодическим, тогда алгоритм 7.2.1 выполняет не более $4n$ буквенных сравнений.

Доказательство. Как указывалось выше, мы можем ограничиться только рассмотрением сдвигов первого типа, где при сравнении произошло t ($t < m$) совпадений. В этом случае применима лемма 7.2.3. В частности, из утверждений *б*) и *в*) следует, что большинство позиций $[1..v-1]$ и $[t-(2v-2)..t-1]$ подстроки $\boldsymbol{t}$ вычислено ранее на предыдущих этапах алгоритма — всего таких позиций $3v-3 \leqslant 3s-3$. Тогда количество f позиций, обрабатываемых впервые, удовлетворяет неравенству

$$f + 3s \geqslant t - (3s-3) + 3s = t + 3.$$

Следовательно, выполняется неравенство (7.10) и поэтому количество буквенных сравнений, выполняемых в алгоритме БМ, как показано ранее, не превышает $4n$.■

Как отмечалось выше, безусловная верхняя граница $3n$ числа буквенных сравнений устанавливается аналогично, но технически более сложным способом.

На этом мы на время прекращаем рассмотрение алгоритма Бойера–Мура. Но в следующей главе мы рассмотрим несколько вариантов этого алгоритма, включая один (раздел 8.4), для которого можно доказать линейное время выполнения даже для периодических паттернов.

Упражнения 7.2

1. Опишите структуру данных, способствующую вычислению значений $\boldsymbol{\delta}_1(\lambda)$ для букв λ упорядоченного (но не индексированного) алфавита A размером α за время $O(\log \alpha)$.
2. Основываясь на предположении, что текстовая строка $\boldsymbol{x}$ (и паттерн $\boldsymbol{p}$) начинается в нулевой позиции с символа $\$_1$ (паттерн, соответственно, с символа $\$_2$), напишите для алгоритма БМ процедуру *hctam* и докажите ее корректность.
3. При обсуждении массива $\boldsymbol{\delta}_1$ рассматривались случаи $j' < j$ и $j' > j$, но не случай $j' = j$. Восполните этот пробел.
4. Для заданного индексированного алфавита A и заданного паттерна $\boldsymbol{p}$ предложите алгоритм вычисления массива $\boldsymbol{\delta}_1$. Оцените временну́ю сложность этого алгоритма.

5. Выполните предыдущее упражнение в предположении, что алфавит не индексирован, но конечен и упорядочен, для чего используйте "пред-предварительный" этап для преобразования данного алфавита A в индексированный алфавит A' и для преобразования паттерна $\boldsymbol{p}$ в строку, определенную на алфавите A'.

6. Вычислите массив $\boldsymbol{\delta}_1$ для индексированного алфавита $A = \{a, b, c, d\}$ и паттерна $\boldsymbol{p} = abcabacad$. Каково значение $\boldsymbol{\delta}_1[\boldsymbol{p}[m]]$ для произвольного паттерна $\boldsymbol{p}$? Приведите пример строки $\boldsymbol{x}$, такой, что использование для определения сдвига только массива $\boldsymbol{\delta}_1$ всегда ведет к сдвигу паттерна $\boldsymbol{p}$ *влево* вдоль строки $\boldsymbol{x}$.

7. Покажите, что "идею" массива $\boldsymbol{\delta}_1$ можно адаптировать для применения в алгоритме КМП. Опишите подробно механизм вычисления нового массива $\boldsymbol{\delta}_1$ и требования, накладываемые на этот массив для того, чтобы его можно было использовать в алгоритме КМП.

8. Приведите алгоритм вычисления за время $\Theta(m)$ массива $\boldsymbol{\delta}_2$ для заданного паттерна $\boldsymbol{p}$. Докажите корректность этого алгоритма и оцените его временну́ю сложность. Кроме памяти объемом $\Theta(m)$, необходимой для хранения паттерна $\boldsymbol{p}$ и массива $\boldsymbol{\delta}_2$, сколько еще памяти может потребоваться для данного алгоритма? От каких параметров (объем алфавита, длина паттерна и т.п.) может зависеть или не зависеть объем дополнительной памяти?

9. Покажите, что для нахождения всех вхождений паттерна $\boldsymbol{p} = a^{k-1}ba^{k-1}$ в строку $\boldsymbol{x} = (a^kb)^r$, где k ($k \geqslant 2$) и r ($r \geqslant 3$) — целые числа, алгоритм БМ выполнит ровно $(3k-1)(r-1)$ буквенных сравнений. Сколько потребуется буквенных сравнений в случае $r = 1$ и $r = 2$? Предположите, что строка $\boldsymbol{x}$ и паттерн $\boldsymbol{p}$ используют начальные символы $\$_1$ и $\$_2$ соответственно.

10. Сколько буквенных сравнений потребуется алгоритму БМ для нахождения всех вхождений паттерна $\boldsymbol{p} = (\lambda_1\lambda_2\ldots\lambda_k)^{m/k}$ в строку $\boldsymbol{x} = (\lambda_1\lambda_2\ldots\lambda_k)^{n/k}$, если все буквы $\lambda_1, \lambda_2, \ldots, \lambda_k$ различны?

7.3 Алгоритм Карпа–Рабина

Почти все из нескольких десятков разработанных после 1970 года алгоритмов сравнения с паттернами являются вариантами или алгоритма КМП или алгоритма БМ либо их обоих. Есть одно примечательное исключение из этого правила — это алгоритм Карпа–Рабина, который использует подход, совершенно отличный от подходов, использующих сдвиг паттерна вдоль просматриваемой текстовой строки.

Основная идея алгоритма КР (так для краткости мы будем называть алгоритм Карпа–Рабина, Karp–Rabin) — использование ***сигнатур***. Сигнатура — это

определенное "замещающее" представление данной строки, которое должно удовлетворять следующим требованиям.

1. Должна быть малая вероятность того, что сигнатуры разных строк имеют одинаковую сигнатуру.
2. Сигнатуру можно эффективно вычислить.
3. Сигнатуру можно вычислить на основе другой сигнатуры.

Сигнатура играет такую же роль при сравнении с паттерном, что и ***хеш-функция*** в структурах данных [5]. Фактически, как и предложено в основополагающей работе [131], сигнатура σ строки $\boldsymbol{x}$ — это целочисленная хеш-функция, вычисленная на основе букв строки $\boldsymbol{x}$. В алгоритме КР сигнатура $\sigma(\boldsymbol{p})$ паттерна $\boldsymbol{p}$ обычно вычисляется заранее, как и сигнатуры $\sigma(\boldsymbol{x}[i..i+m-1])$ для всех $i \in 1..n-m+1$. Если для некоторого значения i выполняется равенство $\sigma(\boldsymbol{x}[i..i+m-1]) = \sigma(\boldsymbol{p})$, тогда вступает в действие процедура *match*(i, m) из простого алгоритма 2.2.1 (раздел 2.2), которая проверяет, будет ли совпадение $\boldsymbol{x}[i..i+m-1] = \boldsymbol{p}$.

При "хорошем" выборе функции σ, она, конечно, должна удовлетворять первому и третьему требованиям, перечисленным выше. Тогда только изредка возможно ***ложное совпадение***, т.е. выполняется совпадение сигнатур, но не совпадение строк:

$$\sigma(\boldsymbol{x}[i..i+m-1]) = \sigma(\boldsymbol{p}), \quad \text{но} \quad \boldsymbol{x}[i..i+m-1] \neq \boldsymbol{p}.$$

Таким образом, выполнив простое сравнение двух целых чисел, для чего требуется только константное время, можно определить совпадение паттерна с подстрокой строки $\boldsymbol{x}$. В работе [131] показано, как сигнатуры текстовых строк можно эффективно вычислять, т.е. второе требование для сигнатур также выполнимо.

Для дальнейшего обсуждения данного подхода немного упростим запись сигнатур: сигнатуру $\sigma(\boldsymbol{x}[i..i+m-1])$ будем обозначать как $\sigma(i)$. Конечно, если на каждом шаге алгоритма необходимо полностью вычислять $\sigma(i)$, то такой подход не практичен, поскольку тогда надо обеспечить доступ к каждой позиции подстроки $\boldsymbol{x}[i..i+m-1]$ и, следовательно, для вычисления каждой сигнатуры $\sigma(i)$ может потребоваться время порядка $\Theta(m)$, а всего для нахождения всех вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$ может потребоваться время порядка $\Theta(m(n-m+1))$.

Но в алгоритме КР предложен способ *последовательного* вычисления $\sigma(i)$, т.е. сигнатура $\sigma(i+1)$ вычисляется на основе сигнатуры $\sigma(i)$. Сделаем упрощающее предположение, что и паттерн $\boldsymbol{p}$ и строку $\boldsymbol{x}$ можно трактовать как двоичные последовательности, состоящие из 0 и 1: при необходимости буквы любого алфавита можно закодировать в виде двоичных строк. Тогда имеет смысл рассмотреть следующие суммы:

$$\Sigma(i) = \sum_{j=1}^{m} 2^{m-j} \boldsymbol{x}[i+j-1]; \quad \Sigma(p) = \sum_{j=1}^{m} 2^{m-j} \boldsymbol{p}[j]. \tag{7.11}$$

Эти суммы — просто другая запись двоичных строк длиной m в виде целых двоичных чисел такой же длины (учитывая ведущие нули). Поскольку равенство $\Sigma(i) = \Sigma(\boldsymbol{p})$ возможно только в том случае, когда $\boldsymbol{x}[i..i+m-1] = \boldsymbol{p}$, алгоритм локализации паттерна $\boldsymbol{p}$ в строке $\boldsymbol{x}$ будет состоять только из вычисления и сравнения сумм Σ. Однако для больших паттернов (например, при $m > 32$) значения Σ также становятся слишком большими для того, чтобы их можно было бы быстро и просто сравнить. Поэтому в работе [131] предложено определять сигнатуры как вычеты по модулю q, где q — простое (достаточно большое) число:

$$\sigma(i) = \Sigma(i) \bmod q; \quad \sigma(\boldsymbol{p}) = \Sigma(\boldsymbol{p}) \bmod q. \tag{7.12}$$

Такая хеш-функция отображает 2^m всех возможных двоичных строк длиной m на множество целых чисел $0..q-1$. Тогда каждому такому числу будет соответствовать в среднем $2^m/q$ двоичных строк или, другими словами, с вероятностью $1/q$ сигнатуры двух различных строк будут совпадать.

Чтобы показать, как вычисляются сигнатуры $\sigma(i)$, введем операции $+_q$, $-_q$ и $\times_q$, определяющие сложение, вычитание и умножение по модулю q соответственно. Оставим для упражнений 7.3.1 и 7.3.2 исследование свойств этих операций. Здесь отметим, что эти операции позволяют вычислить сигнатуру $\sigma(i+1)$ на основании значения $\sigma(i)$, $i \geqslant 1$, за константное время по следующей формуле.

Лемма 7.3.1. $\sigma(i+1) = 2\times_q \big(\sigma(i) -_q (2^{m-1} \bmod q) \times_q \boldsymbol{x}[i]\big) +_q \boldsymbol{x}[i+m]$.

Доказательство этой формулы предложено дать в упражнении 7.3.3. ■

Поскольку значения $2^{m-1} \bmod q$ могут быть вычислены заранее, вычисление сигнатуры $(i+1)$ на основании значения $\sigma(i)$ требует всего четырех операций, которые можно эффективно реализовать на компьютере. С учетом этого замечания запишем алгоритм КР.

Алгоритм 7.3.1 (Алгоритм Карпа–Рабина)

```
▷ Нахождение всех вхождений паттерна p в строку x
выбрать q
вычислить σ(p)                ▷ используются формулы (7.12)
for i ← 1 to n − m + 1 do
    вычислить σ(i)            ▷ используется формула леммы 7.3.1
    if σ(i) = σ(p) then j ← match(i, m)
    if j = m + 1 then output i
```

Нет сомнения, что приведенный алгоритм КР корректен, поскольку он сводится к алгоритму 2.2.1 в случае $\sigma(i) = \sigma(\boldsymbol{p})$, а в случае $\sigma(i) \neq \sigma(\boldsymbol{p})$ по определению $\boldsymbol{x}[i..i+m-1] \neq \boldsymbol{p}$. Поэтому имеет место следующая теорема.

Теорема 7.3.2. Алгоритм 7.3.1 корректно вычисляет все вхождения непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$. ■

Очевидно, что эффективность алгоритма КР очень сильно зависит от выбора числа q. При плохом выборе числа q процедура $match(i, m)$ может вызываться $n - m + 1$ раз (и при этом совпадения сигнатур будут ложными). Тогда для выполнения алгоритма КР потребуется время порядка $\Theta(nm)$, что не лучше, чем у простого алгоритма 2.2.1. С другой стороны, при удачном выборе числа q не будет ложных совпадений сигнатур, и поэтому алгоритм КР будет выполнен за время $O(n + km)$, где k — количество вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$. В работе [131] теоретически показано, что если простое число q выбрано случайным образом из интервала целых чисел $1..mn^2$ и ложные совпадения сигнатур также случайны, тогда ожидаемое время выполнения алгоритма КР практически сводится до минимума $O(n + km)$. Это ожидаемое время выполнения алгоритма КР почти сравнимо с временем выполнения алгоритма БМ. Однако заметим, что в алгоритме КР время выполнения зависит от количества k вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$. Поэтому если паттерн часто встречается в строке $\boldsymbol{x}$, например $k \in \Theta(n)$, то время выполнения алгоритма имеет порядок $\Theta(nm)$.

На практике, как утверждается в статье [131], алгоритм КР будет наиболее подходящим алгоритмом сравнения с паттерном в случае больших паттернов, например, при $m \geqslant 200$. Если сравнивать алгоритм КР с алгоритмами КМП и БМ, то алгоритм КР требует дополнительной памяти только фиксированного объема. Для такой реализации данного алгоритма можно использовать накапливающие регистры для быстрого доступа к данным независимо от величины m, тогда как в алгоритмах КМП и БМ доступ к данным для больших значений m замедляется, поскольку данные выбираются из массива размером $\Theta(m)$. Но с другой стороны, на машинах определенной архитектуры (например, при использовании кеш-памяти) такая задержка доступа к данным может быть незначительной, а алгоритм КР может выполняться медленнее из-за интенсивного использования целочисленной арифметики [24, 103].

Как указывается в работе [131], алгоритм КР можно адаптировать для работы с двухмерными паттернами, даже в случае их неравномерной формы, что ведет к эффективной реализации параллельных вычислений.

Упражнения 7.3

1. Обозначим через $\bullet$ одну из операций $+$, $-$, $\times$, а через $\bullet_q$ — соответствующую операцию $+_q$, $-_q$ или $\times_q$. Покажите, что для любых целых положительных чисел q, r и s выполняется равенство

$$(r \bullet s) \bmod q = (r \bmod q) \bullet_q (s \bmod q).$$

2. Для больших значений m вычисления по формулам (7.12) могут вызвать переполнение машинного слова, если суммы (7.11) вычисляются ранее, чем вычеты по модулю q. Используя операции $+_q$ и $\times_q$, разработайте эффективную итерационную процедуру вычисления σ, которая будет сохранять частные суммы, ограниченные величиной порядка $O(q)$.

3. Докажите лемму 7.3.1.

4. Пусть $m \leqslant 32$. Разработайте эффективный алгоритм, подобный алгоритму КР, который будет использовать суммы Σ вместо сигнатур σ. Какова верхняя граница выполнения такого алгоритма в самом худшем случае? Какова нижняя граница его выполнения в самом лучшем случае? Какова средне ожидаемая его временна́я сложность?

5. Предположим, что $\boldsymbol{p}$ и $\boldsymbol{x}$ определены на целочисленном алфавите $\{0, 1, \ldots, t\}$. Предложите наиболее приемлемые в этом случае суммы типа (7.11). Сформулируйте и докажите обобщение леммы 7.3.1.

7.4 Алгоритм Демелки–Бейза-Ятса–Гоннета

Прежде чем описывать сам алгоритм, немного остановимся на его истории. Его недавняя история началась в 1992 году, когда Бейза-Ятс и Гоннет (Baeza-Yates and Gonnet) в своей знаменитой работе [24] предложили метод, основанный на бинарных отображениях; этот метод будет описан далее в этом разделе. Эта работа инспирировала появление десятков, если не сотен других работ, которые разрабатывают самые разнообразные варианты этого метода. Однако у этой истории существует неожиданная предыстория: в работе Дёмёлки 1968 года (Dömölki, [77]) описывается практически тот же самый алгоритм, и, более того, автор ссылается на статью [78] 1965 года, в которой "этот метод описан более подробно". Так получилось, что алгоритм Дёмёлки, опубликованный в Англии в популярном научном компьютерном журнале, остался забытым и не востребованным на 27 лет, пока его заново не открыл кто-то другой!

На этот необычный факт обратили мое внимание [190] во время моих лекций, прочитанных в университете Дебрецена в октябре-ноябре 2001 года. В честь первооткрывателя данного метода будем называть описываемый алгоритм алгоритмом Демелки–Бейза-Ятса–Гоннета (сокращенно — алгоритм ДБГ), а не алгоритмом Бейза-Ятса–Гоннета, как принято в настоящее время.

Алгоритм ДБГ подобен алгоритму КР, рассмотренному в предыдущем разделе, и также, как и алгоритм КР, отличен от алгоритмов КМП и БМ. Здесь основное внимание уделяется не наиболее быстрому способу сдвига паттерна вдоль строки $\boldsymbol{x}$, а оптимизации вычислений на каждом шаге алгоритма, выполняющих сравнение с паттерном для каждой позиции строки $\boldsymbol{x}$. Другими словами, алгоритм ДБГ (как и алгоритм КР) пытается минимизировать вычисления на каждом шаге,

но не пытается минимизировать количество шагов (как это делается в алгоритмах КМП и БМ). Фактически, в алгоритме ДБГ найден способ сравнения текущей буквы $\boldsymbol{x}[i]$ со всеми m буквами паттерна $\boldsymbol{p}$ с помощью одной операции, которая выполняется за константное время!

Ключевую роль в алгоритме ДБГ играет бинарный массив (строка) $\boldsymbol{s} = \boldsymbol{s}[1..m]$, который описывает текущее ***состояние*** вычислений для текущей позиции i ($i \in \in 1..n$) в строке $\boldsymbol{x}$ [24]. В этом массиве для любого значения $j \in 1..m$ выполняется следующее соглашение:

$$\boldsymbol{s}[j] = 0 \Leftrightarrow +\boldsymbol{x}[i-j+1..i] = \boldsymbol{p}[1..j], \tag{7.13}$$

где i — текущая позиция в строке $\boldsymbol{x}$. Очевидно, что если подстрока $\boldsymbol{x}[i-m+1..i]$ совпадает с паттерном $\boldsymbol{p}$, то $\boldsymbol{s}[m] = 0$. Кажется разумным до начала вычисления массива $\boldsymbol{s}$ положить все его элементы равными 1 (т.е. положить $\boldsymbol{s} = 1^m$), это будет отражать тот факт, что на данном шаге алгоритма пока не производились сравнения с паттерном. Далее надо показать, как изменить значения массива $\boldsymbol{s}$ так, чтобы выполнялось соглашение (7.13).

Но сначала введем еще один массив (подобный массиву $\boldsymbol{\delta}_1$ из алгоритма БМ), предварительно сделав предположения о том, что алфавит строки $\boldsymbol{x}$ конечен, известен заранее и индексирован (см. раздел 4.1). Тогда без потери общности (см. упражнение 4.1.3) можно считать, что алфавит состоит из натуральных чисел $N_\alpha = \{1, 2, \ldots, \alpha\}$. Теперь определим двухмерный массив $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$, такой, что для всех $h \in 1..\alpha$ и $j \in 1..m$

$$\boldsymbol{t}[h, j] = 0, \text{ если } \boldsymbol{p}[j] = h, \text{ и } \boldsymbol{t}[h, j] = 1 \text{ в противном случае.} \tag{7.14}$$

При таком определении массива $\boldsymbol{t}$ для любой буквы $h \in N_\alpha$ строка $\boldsymbol{t}[h, 1..m]$ элементов этого массива будет состоять из нулей и единиц, причем нули будут стоять в тех позициях, в которых встречается буква h в паттерне $\boldsymbol{p}$. Следующая лемма показывает использование массива $\boldsymbol{t}$ для вычисления массива (состояния) $\boldsymbol{s}$ в позиции i на основании известного состояния в позиции $i-1$.

Лемма 7.4.1. Обозначим через $\boldsymbol{s}^{(i)}$ строку $\boldsymbol{s}$, определяющую состояние, соответствующее позиции i текстовой строки $\boldsymbol{x}$. Положим, что начальное состояние описывается как $\boldsymbol{s}^{(0)}[0..m] = 0(1^m)$. Тогда для всех $i \in 1..n$ и $j \in 1..m$

$$\boldsymbol{s}^{(i)}[j] = \boldsymbol{s}^{(i-1)}[j-1] \vee \boldsymbol{t}[\boldsymbol{x}[i], j], \tag{7.15}$$

где $\vee$ — знак логической операции OR (логическое сложение ИЛИ). (Напомним, что результат логической операции OR равен 1, если хотя бы один операнд равен 1, и 0 — в противном случае.)

Доказательство. Пусть i — текущая позиция в строке $\boldsymbol{x}$. (Отметим, что, согласно формуле (7.15), в вычислениях не участвует значение $\boldsymbol{s}^{(i-1)}[m]$.) Сначала предположим, что для некоторого $j \in 1..m \boldsymbol{s}^{(i-1)}[j-1] = 0$, тогда, в соответствии с соглашением (7.13), выполняется равенство

$$\boldsymbol{x}[i-j+1..i-1] = \boldsymbol{p}[1..j-1].$$

Если $\boldsymbol{t}[\boldsymbol{x}[i], j] = 0$, тогда, согласно (7.14), $\boldsymbol{x}[i] = \boldsymbol{p}[j]$, и поэтому полагаем $\boldsymbol{s}^{(i)}[j] \leftarrow$ $\leftarrow 0$. Если же $\boldsymbol{t}[\boldsymbol{x}[i], j] = 1$, тогда $\boldsymbol{x}[i] \neq \boldsymbol{p}[j]$, и в этом случае полагаем $\boldsymbol{s}^{(i)}[j] \leftarrow 1$. Каждое из этих действий совпадает с результатом выполнения операции OR.

Аналогично рассматривается случай $\boldsymbol{s}^{(i-1)}[j-1] = 1$. ■

Чтобы увидеть, как работает формула (7.15) в вычислении состояний $\boldsymbol{s}^{(i)}$, $i \in$ $\in 1..n$, рассмотрим простой пример с паттерном $\boldsymbol{p} = 212$ и строкой $\boldsymbol{x} = 12112121$, определенными на алфавите $A = \{1, 2\}$. Здесь $m = 3$, $n = 8$ и $\boldsymbol{t} = \left(\begin{smallmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{smallmatrix}\right)$. Используя начальное состояние $\boldsymbol{s}^{(0)} = 0111$, вычисляем

$$\begin{aligned} \boldsymbol{s}^{(1)}[1] &= 0 \vee \boldsymbol{t}[1,1] = 1, \\ \boldsymbol{s}^{(1)}[2] &= 1 \vee \boldsymbol{t}[1,2] = 1, \\ \boldsymbol{s}^{(1)}[3] &= 1 \vee \boldsymbol{t}[1,3] = 1. \end{aligned}$$

Аналогично выполняются вычисления для $i = 2, 3, \ldots, 8$: $\boldsymbol{s}^{(2)} = 011$, $\boldsymbol{s}^{(3)} = 101$, $\boldsymbol{s}^{(4)} = 111$, $\boldsymbol{s}^{(5)} = 011$, $\boldsymbol{s}^{(6)} = 101$, $\boldsymbol{s}^{(7)} = 010$, $\boldsymbol{s}^{(8)} = 101$. Значение $\boldsymbol{s}^{(7)}[3] = 0$ определяет совпадение $\boldsymbol{p}$ и $\boldsymbol{x}[5..7]$.

Приведем листинг алгоритма ДБГ.

Алгоритм 7.4.1 (Алгоритм Демелки–Бейза-Ятса–Гоннета)

```
▷ Нахождение всех вхождений паттерна p в строку x
s[0..m] ← 0(1^m)        ▷ задание начального состояния
for i ← 1 to n do
    ▷ Сдвиг на одну позицию вправо в массиве s с помощью
    ▷ процедуры rightshift, затем вычисляется новое состояние,
    ▷ оставляя без изменения s[0] и выполняя
    ▷ для остальных элементов массива s логическую
    ▷ операцию OR с операндами s[1..m] и t[x[i], 1..m]
    s ← rightshift(s, 1) ∨ t[x[i], 1..m]
    if s[m] = 0 then output i − m + 1
```

Алгоритм 7.4.1 основан на доказанной формуле леммы 7.4.1, поэтому мы имеем формальное право сформулировать следующее утверждение.

Теорема 7.4.2. Алгоритм 7.4.1 корректно вычисляет все вхождения непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$. ■

Чтобы оценить временну́ю и пространственную сложность алгоритма ДБГ, необходимо ввести в рассмотрение еще один параметр, а именно, длину w машинного слова. Как видно из алгоритма 7.4.1, если $m \leqslant w$, то каждая из строк $\boldsymbol{s}[1..m]$ и $\boldsymbol{t}[h, 1..m]$ будет занимать не более одного машинного слова. Поэтому вычисление массива $\boldsymbol{s}$ может потребовать всего две операции, выполняемые за константное время: операцию сдвига на один разряд и логическую операцию OR в регистре, содержащем одно машинное слово. В случае произвольного m строки $\boldsymbol{s}[1..m]$ и $\boldsymbol{t}[h, 1..m]$ будут занимать по $\lceil m/w \rceil$ машинных слов, и поэтому для вычисления массива $\boldsymbol{s}$ потребуется время порядка $\Theta(\lceil m/w \rceil)$. Следовательно, имеет место такая теорема.

Теорема 7.4.3. Алгоритм 7.4.1 вычисляет все вхождения непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$ за время порядка $\Theta(\lceil m/w \rceil)$, при этом используя память объемом $\Theta((\alpha + 1)\lceil m/w \rceil)$. ■

Из этой теоремы следует, что в случае больших паттернов алгоритм ДБГ выполняется за время $\Theta(nm)$, а в случае больших алфавитов требует памяти объемом $\Theta(nm)$. Поэтому в таких ситуациях данный алгоритм не применяется. Однако в случае коротких паттернов и малых алфавитов алгоритм выполняется очень быстро и эффективно. Как мы увидим в главе 10, алгоритм ДБГ можно очень легко адаптировать для вычисления приближенных паттернов — это очень полезный и гибкий алгоритм для решения задач такого класса [233].

Относительно массива $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$ можно сказать, что его целесообразно вычислить на предварительном этапе, и такая операция не займет много времени.

Теорема 7.4.4. Массив $\boldsymbol{t}$ можно вычислить за время $\Theta(\alpha\lceil m/w \rceil + m)$.

Доказательство этой теоремы предложено дать в упражнении 7.4.2. ■

Напомним обсуждение 1.3.1, в котором рассматривался вопрос об отношении между размером задачи n и длиной машинного слова w и где показано, что на практике при $w \geqslant 32$ вполне обоснованно можно положить $\log_2 n/w \leqslant 2$ (или, другими словами, $w \in \Omega(\log_2 n)$). Тогда, как следует из теоремы 7.4.3, время выполнения алгоритма ДБГ составит $O(mn/\log n)$ для любых текстовых строк $\boldsymbol{x}[1..n]$ — это лучшая верхняя граница времени выполнения среди всех известных алгоритмов сравнений с паттерном!

Упражнения 7.4

1. Завершите доказательство леммы 7.4.1.
2. Предложите алгоритм вычисления массива $\boldsymbol{t}$, используемого в алгоритме ДБГ. Затем докажите теорему 7.4.4.

3. Вычислите массив $\boldsymbol{t}$ для алфавита $A = \{a, b, c, d, e\}$ и паттерна

$$\boldsymbol{p} = abacabadabacabae.$$

4. Можно ли переформулировать задачу данного раздела таким образом, чтобы алгоритм ее решения использовал логическую операцию AND (логическое умножение И) вместо операции OR?

7.5 Заключение

Из четырех описанных в этой главе алгоритмов наибольшим спектром достоинств обладает, без сомнения, алгоритм Бойера–Мура. На практике, конечно, учитываются все параметры задачи (длина текстовой строки, длина паттерна, количество найденных вхождений паттерна в строку и т.д.). Алгоритм БМ при прочих равных условиях для периодических паттернов гарантирует, в теории, не более $3n$ буквенных сравнений и время выполнения порядка $\Theta(n)$. Более того, как мы видели, среднее количество буквенных сравнений в алгоритме БМ составляет только примерно одну треть от среднего количества буквенных сравнений, выполняемых алгоритмом КМП. Вместе с тем не иссякает интерес к алгоритму КМП, поскольку он гарантирует не более $2n$ буквенных сравнений в самом худшем случае и может обрабатывать множественные вхождения периодических паттернов. Если не брать во внимание периодические паттерны, то алгоритм БМ теоретически имеет единственный недостаток, который сформулирован в виде ограничений применения этого алгоритма в начале раздела 7.2. Фактически, на сегодняшний день алгоритм БМ является лучшим для точного сравнения с паттерном, даже несмотря на то что его многочисленные "потомки" подчас демонстрируют эффективность, превосходящую эффективность своего знаменитого "предка". В следующей главе мы рассмотрим несколько таких перспективных "потомков". В частности, мы покажем, что небольшая модификация алгоритма БМ позволяет находить множественные вхождения периодических паттернов с такой же эффективностью, как и при использовании алгоритма КМП.

ГЛАВА 8

Наследники Бойера–Мура

> Тот, кто только говорит, но ничего не делает,
> похож на сад, заросший сорняками.
>
> — Аноним

В этой главе мы рассмотрим несколько вариантов алгоритма Бойера–Мура (алгоритм БМ, раздел 7.2), которые разработаны в последнюю четверть минувшего столетия и которые имеют или некоторые теоретические усовершенствования (уменьшают количество буквенных сравнений), или практические достоинства (уменьшают время выполнения). Фактически обширную литературу по этой теме можно четко разделить на теоретические и практические работы: те, которые устанавливают все более точные оценки границ числа буквенных сравнений в самом худшем случае, и те, которые предлагают все более быстрые и универсальные алгоритмы. В этой главе основное внимание уделяется именно последним работам и алгоритмам. И не потому, что я считаю теоретические работы не интересными и не важными, а просто потому, что хочу проследить эволюцию идей, ведущих к современному теоретическому пониманию проблемы сравнения текстовых строк, близкому к тому подходу, который пропагандируется в этой книге. Цель этой главы — дать "качественное" представление основных идей (многие из них являются простыми эвристиками), которые повышают практическую эффективность алгоритма БМ. И, как мы увидим, некоторые из этих идей также ведут к теоретическим улучшениям, т.е. уменьшают количество буквенных сравнений. В разделе 8.6 мы рассмотрим интересный вариант алгоритма КМП (об этом алгоритме см. раздел 7.1), который уменьшает количество буквенных сравнений

в самом худшем случае с величины $2n-m$ до величины $1{,}5n-0{,}5m$. В последнем разделе кратко рассмотрим теоретические аспекты описанных алгоритмов.

8.1 Алгоритм БМ — цикл с перескоками

Мы начнем с алгоритма, который был представлен в исходной работе Бойера и Мура [38], но который был забыт на несколько лет. Основная идея этого алгоритма (его часто называют ***циклом с перескоками*** (skip loop)) заключается в том, что паттерн $\boldsymbol{p}$ сдвигается вдоль строки $\boldsymbol{x}$ путем больших "перескоков", при этом длина перескоков определяется с помощью легко проверяемого условия, которое *позволяет надеяться*, что паттерн $\boldsymbol{p}$ не встречается в "перепрыгнутой" части строки $\boldsymbol{x}$.

Для практической реализации данного алгоритма используется знакомый нам слегка измененный массив $\boldsymbol{\delta}_1$ (здесь его будем обозначать как $\boldsymbol{\delta}'_1$), с помощью которого получим условия перескока. Изменения в массиве $\boldsymbol{\delta}_1$ просты: для любой буквы λ положим $\boldsymbol{\delta}'_1[\lambda] = \boldsymbol{\delta}_1[\lambda]$, *за исключением* случая $\lambda = \boldsymbol{p}[m]$, когда вместо присвоения $\boldsymbol{\delta}_1[\boldsymbol{p}[m]] \leftarrow 0$ положим $\boldsymbol{\delta}'_1[p[m]] \leftarrow N$, где N — некоторое фиксированное число, такое, что $N > m+n$. Далее в алгоритме БМ (см. алгоритм 7.2.1) первый оператор в цикле **while** заменяется на следующий оператор цикла

$$\textbf{repeat } i \leftarrow i + \boldsymbol{\delta}'_1[x[i]] \textbf{ until } i > n$$

Пусть i_0 — значение i, которое предшествует выполнению перескока. Если после перескока мы имеем $i > N$, то отсюда можно сделать следующие заключения:

1. $\boldsymbol{x}[i-N] = \boldsymbol{p}[m]$;
2. не существует позиции $i' \in i_0..(i-N)-1$, такой, что $\boldsymbol{x}[i'] = \boldsymbol{p}[m]$.

Таким образом, $i-N$ — это первая позиция в строке $\boldsymbol{x}$, которая находится правее позиции i_0-1 и в которой присутствует буква $\boldsymbol{p}[m]$. В этом случае можно вернуться к обычному процессу алгоритма БМ (т.е. можем выполнить процедуру *hctam*$(i, m-1)$), предварительно выполнив оператор

$$\textbf{if } i > N \textbf{ then } i \leftarrow (i-N)-1.$$

Если же на выходе перескока $i \leqslant N$, тогда $i > n$, и мы снова можем утверждать, что не существует позиции i', такой, что $i_0 \leqslant i' \leqslant n$ и $\boldsymbol{x}[i'] = \boldsymbol{p}[m]$. Другими словами, паттерн $\boldsymbol{p}$ не может совпасть с подстрокой $\boldsymbol{x}[i-m+1..n]$, и поэтому поиск можно закончить.

Даже это краткое описание позволит без особого труда изменить алгоритм БМ таким образом, чтобы он реализовал идею цикла с перескоками. В результате получим алгоритм, который обычно называют BMFAST (быстрый алгоритм БМ), поскольку на практике в среднем он выполняется быстрее, чем исходный алгоритм

БМ [121]. Свойства алгоритма BMFAST и ситуации, когда он выполняется быстрее (или медленнее), чем алгоритм БМ, показаны в упражнениях 8.1.1–8.1.5.

Чтобы получить качественное представление о преимуществах последнего алгоритма, заметим, что в алгоритме BMFAST буквенное сравнение $\boldsymbol{x}[i_0] : \boldsymbol{p}[m]$ заменяется одним или несколькими перескоками.

- Если в действительности $\boldsymbol{x}[i_0] = \boldsymbol{p}[m]$, тогда в цикле с перескоками выполняется больше операторов, чем выполнялось бы буквенных сравнений. Поэтому в данном случае алгоритм BMFAST не эффективен.
- Если $\boldsymbol{x}[i] = \boldsymbol{p}[m]$ для некоторого $i > i_0$, тогда нахождение такого i в общем случае в алгоритме BMFAST происходит быстрее, чем в алгоритме БМ.
- Если не существует такого $i > i_0$, что $\boldsymbol{x}[i] = \boldsymbol{p}[m]$, то в этом случае алгоритм BMFAST также более эффективен, чем алгоритм БМ, особенно на больших алфавитах.

Предложены другие варианты циклов с перескоками, в частности в [119] предложен алгоритм, построенный на этой основе, для поиска одной буквы паттерна $\boldsymbol{p}$ в строке $\boldsymbol{x}$, который особенно эффективен, если известно, что эта буква в строке $\boldsymbol{x}$ встречается нечасто. Этот алгоритм в общем случае выполняется быстрее, чем алгоритм БМ, но в среднем медленнее, чем алгоритм BMFAST [121].

Упражнения 8.1

1. Объясните подробно, почему имеют место утверждения 1 и 2.
2. Запишите алгоритм BMFAST и докажите его корректность в предположении, что корректен алгоритм БМ.
3. Охарактеризуйте ситуации, в которых алгоритм BMFAST будет выполняться быстрее, чем алгоритм БМ. Аналогично опишите ситуации, где этот алгоритм будет более медленным, чем алгоритм БМ. Не забудьте рассмотреть случай $\boldsymbol{x} = a^n$, $\boldsymbol{p} = a^m$!
4. В разделе 7.2 был показан пример $\boldsymbol{x} = (a^k b)^r$, $\boldsymbol{p} = a^{k-1}ba^{k-1}$, который считается "плохим" для выполнения алгоритма БМ. Оцените количество буквенных сравнений, выполняемых алгоритмом BMFAST для этого примера. Будет ли в действительности в этом случае алгоритм BMFAST быстрее, чем алгоритм БМ?
5. Всегда ли алгоритм BMFAST требует меньшего количества буквенных сравнений, чем алгоритм БМ? Обоснуйте свой ответ.
6. Запишите алгоритм, упоминавшийся в конце раздела, который реализует идею цикла с перескоками для поиска вхождения в строку одной заданной буквы.

8.2 Алгоритм Бойера–Мура–Хоспула

В предыдущем разделе показано, как небольшое изменение массива $\boldsymbol{\delta}_1$ (точнее, только одного значения $\boldsymbol{\delta}_1[\boldsymbol{p}[m]]$) можно использовать для реализации длинного "перескока" паттерна вдоль исследуемой строки. В этом разделе рассмотрим еще один способ вычисления значения $\boldsymbol{\delta}_1[\boldsymbol{p}[m]]$, который позволит реализовать алгоритм БМ без использования массива $\boldsymbol{\delta}_2$. Это поможет избежать трудностей, связанных с вычислением массива $\boldsymbol{\delta}_2$ и описанных в разделе 7.2 (и которые снова проявят себя в разделе 8.4). Преимущества такого подхода очевидны — новый алгоритм на практике в среднем эффективнее исходного алгоритма БМ, а количество выполняемых буквенных сравнений, как показано в разделе 7.2, за исключением некоторых редких случаев, пропорционально mn.

Напомним, что путем использования массива $\boldsymbol{\delta}_2$ достигаются две цели: во-первых, алгоритм БМ никогда не возвращается назад, т.е. любое смещение паттерна выполняется только вправо; во-вторых, за исключением случая 3-периодических паттернов, алгоритм БМ выполняется за время порядка $O(n)$, т.е. достигает теоретической верхней границы времени выполнения, что невозможно сделать без использования массива $\boldsymbol{\delta}_2$. Хоспул (Horspool, [119]) показал, что по крайней мере первую цель можно достичь, используя только массив $\boldsymbol{\delta}_1$.

В разделе 7.2 определено, что для любой буквы λ из алфавита A $\boldsymbol{\delta}_1[\lambda] = m - j'$, где j' — самая правая позиция буквы λ в паттерне $\boldsymbol{p}$, и $\boldsymbol{\delta}_1[\lambda] = 0$, если λ не встречается в паттерне. Основная идея алгоритма Бойера–Мура–Хоспула (алгоритма БМХ) заключается в том, что когда обнаружено несовпадение паттерна с очередной подстрокой строки $\boldsymbol{x}$, паттерн $\boldsymbol{p}$ сдвигается далее не в соответствии с обнаруженной позицией несовпадения, а в соответствии с самой правой позицией в паттерне буквы, вызвавшей несовпадение. Другими словами, пусть имеется совпадение

$$\boldsymbol{x}[i+1..i+h] = \boldsymbol{p}[m-h+1..m]$$

для некоторого $h \in 0..m$, которое заканчивается несовпадением

$$\lambda = \boldsymbol{x}[i] \neq \boldsymbol{p}[m-h] = \mu.$$

Вместо сдвига $i \leftarrow i + \boldsymbol{\delta}_1[\boldsymbol{x}[i]]$, предусмотренного в алгоритме БМ, в алгоритме БМХ выполняется сдвиг

$$i \leftarrow (i+h) + \boldsymbol{\delta}_1[\boldsymbol{x}[i+h]].$$

Таким образом, $\boldsymbol{p}[m]$ далее сравнивается с буквой в позиции i строки $\boldsymbol{x}$, которая в текущем сравнении отстояла на $\boldsymbol{\delta}_1[\boldsymbol{x}[i+h]]$ позиций справа от позиции буквы $\boldsymbol{p}[m]$. Такая стратегия решает одну из проблем в алгоритме БМ, порождаемую сдвигом, определяемым только на основе массива $\boldsymbol{\delta}_1$, — теперь невозможен сдвиг паттерна *влево*, так как все значения в массиве $\boldsymbol{\delta}_1$ неотрицательны. С другой

стороны, новый подход обостряет проблему нулевого сдвига (когда $\boldsymbol{\delta}_1[\lambda] = 0$), который возможен при совпадении $\lambda = \boldsymbol{p}[m]$, поскольку тогда (для $h > 0$) должно выполняться совпадение $\boldsymbol{p}[m] = \boldsymbol{x}[i+h]$, что приводит к значению

$$\boldsymbol{\delta}_1[\boldsymbol{x}[i+h]] = m - m = 0.$$

Как указывалось в начале этого раздела, алгоритм БМХ решает эту проблему путем переопределения значения $\boldsymbol{\delta}_1[\lambda]$ для случая $\lambda = \boldsymbol{p}[m]$ (новый массив будем обозначать как $\boldsymbol{\delta}_1''$). Сначала рассмотрим ситуацию, когда совпадение $\boldsymbol{p}[m] = \boldsymbol{x}[i+h]$ порождено единственным вхождением λ в паттерн $\boldsymbol{p}$. В этом случае можно не проводить сравнение $\boldsymbol{p}$ и $\boldsymbol{x}$ до тех пор, пока паттерн $\boldsymbol{p}$ не сдвинут полностью на позицию $i+h$ в строке $\boldsymbol{x}$. Поэтому при следующем вызове процедуры *hctam* можно выполнять сравнения, начиная со сравнения $\boldsymbol{p}[m]$ с буквой $\boldsymbol{x}[(i+h)+m]$. В таком случае удобно положить $\boldsymbol{\delta}_1''[\lambda] \leftarrow m$. В общем случае, обозначив через j' ($j' < m$) позицию самого правого вхождения буквы λ в паттерн $\boldsymbol{p}$, имеет смысл положить $\boldsymbol{\delta}_1''[\lambda] \leftarrow m - j'$. Таким способом в алгоритме БМХ создается массив $\boldsymbol{\delta}_1''$, который совпадает с массивом $\boldsymbol{\delta}_1$ для всех букв, за исключением буквы $\lambda = \boldsymbol{p}[m]$, когда значение $\boldsymbol{\delta}_1''[\lambda]$ вычисляется так, как только что показано. Алгоритм БМХ легко получается из алгоритма БМ путем внесения минимальных изменений в процедуру *hctam* и в вычисление массива $\boldsymbol{\delta}_1$. Мы оставим описание этих изменений для упражнения 8.2.1. В упражнении 8.2.3 предложено определить такие строки $\boldsymbol{p}$ и $\boldsymbol{x}$, для которых алгоритм БМХ требует времени выполнения порядка $O(mn)$.

Относительная простота алгоритма БМХ (по сравнению с алгоритмом БМ) не только делает его привлекательным для применения на практике, но и послужила причиной многочисленных исследований его поведения. В работах [23, 25, 28] показано, что на алфавите размером α для сравнения текстовых строк длиной n и m ($n > m$) количество буквенных сравнений примерно равно (или больше) оценке $(n - m + 1)/\alpha$, которая становится все более точной при возрастании значений m и α. Так, для больших алфавитов алгоритм БМХ (несмотря на то, что в самом худшем случае он работает плохо) в среднем выполняется значительно быстрее, чем стандартный алгоритм БМ. Напомним (см. раздел 7.2), что алгоритм БМ на больших алфавитах в среднем выполняет около $0{,}3n$ буквенных сравнений.

Упражнения 8.2

1. В соответствии с описанием, данным в этом разделе, запишите алгоритм БМХ, указав изменения в процедуре *hctam* и в вычислении массива $\boldsymbol{\delta}_1$.
2. Докажите корректность алгоритма БМХ, записанного в предыдущем упражнении.
3. Опишите бесконечные классы строк $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и $\boldsymbol{x}[1..n]$, таких, что алгоритм БМХ для их сравнения потребует времени порядка $\Theta(mn)$.

4. В первых двух разделах этой главы описаны различные модификации алгоритма БМ, основанные на вычисленном особым образом значении $\boldsymbol{\delta}_1[\boldsymbol{p}[m]]$. Можно ли скомбинировать эти модификации в единый алгоритм, который не использовал бы массив $\boldsymbol{\delta}_2$, но выполнял бы и перескок, и сдвиг на основе измененного массива $\boldsymbol{\delta}_1$? Запишите такой алгоритм, докажите его корректность, оцените его асимптотическую сложность и эффективность в практическом применении.

Замечание. Автору этой книги известно, что гибридный алгоритм, предложенный в этом упражнении, никогда ранее не изучался. Ваши исследования этого алгоритма могут привести к новым открытиям! (Так уже не раз бывало, например, см. [129].)

8.3 Частота встречаемости букв и алгоритмы Бойера–Мура–Санди

Хотя все алгоритмы БМ сканируют паттерн $\boldsymbol{p}$ справа налево, нет обязательного правила именно такого порядка просмотра этих элементов. Независимо от порядка их просмотра, целью должно быть наискорейшее нахождение несовпадения паттерна с просматриваемой подстрокой строки $\boldsymbol{x}$ и кратчайшее (по количеству раз) перемещение паттерна вдоль строки $\boldsymbol{x}$ (с помощью перескоков или сдвигов). В этом разделе рассмотрим несколько вариантов алгоритма БМ, которые применимы в ситуациях, когда известны оценки частот встречаемости букв алфавита в строке $\boldsymbol{x}$ и когда значения этих частот значимо различны для разных букв. В таких ситуациях особый интерес представляют те буквы паттерна $\boldsymbol{p}$, которые наиболее редко встречаются в строке $\boldsymbol{x}$. Такие буквы, присутствующие в паттерне, с наименьшей вероятностью будут совпадать с буквами строки $\boldsymbol{x}$, и на этой основе можно организовать большие перескоки или сдвиги паттерна $\boldsymbol{p}$ вдоль строки $\boldsymbol{x}$. В частности, используя такой подход, можно значительно ускорить сравнение текстов на естественных языках, у которых частота встречаемости букв, как правило, известна.

8.3.1 Сравнение со стражем

Предположим, что на предварительном этапе алгоритма найдена позиция $j_{\text{стр}}$, такая, что буква $\boldsymbol{p}[j_{\text{стр}}]$ из паттерна $\boldsymbol{p}$ — это буква, с наименьшей частотой встречаемая в текстовых строках, определенных на некотором алфавите. В этом случае процедуру *hctam* следует применять только тогда, когда найдено совпадение ***стража*** (guard) $\boldsymbol{p}[j_{\text{стр}}]$ с соответствующей буквой в строке $\boldsymbol{x}$. Другими словами, в этом случае можно в алгоритме БМ использовать следующий фрагмент кода.

if $\boldsymbol{x}[i-(m-j_{\text{стр}})] = \boldsymbol{p}[j_{\text{стр}}]$ **then**

$(i, j) \leftarrow hctam(i, m)$
if $j = 0$ **then output** $i + 1$
else
$j \leftarrow j_{\text{стр}}$
$i \leftarrow i + \max\{\boldsymbol{\delta}_1[\boldsymbol{x}[i]], \boldsymbol{\delta}_2[j]\}$

Доказательство корректности такого варианта алгоритма БМ мы оставим для упражнения 8.3.1 (доказательство не очевидно).

Идея "стража", по-видимому, впервые предложена в работе [142], затем она была независимо сформулирована в работе [121]. Согласно тестовым проверкам, результаты которых представлены в [121], использование стража приводит к явному увеличению скорости сравнения текстов на естественных языках.

8.3.2 Наиболее эффективные перескоки

Из сказанного выше следует вывод, что наиболее быстрый поиск в строке $\boldsymbol{x}$ буквы $\boldsymbol{p}[j]$ будет тогда, когда $j = j_{\text{стр}}$. Также представляет интерес предположение, сделанное в упражнении 8.3.5, что алгоритм со стражем можно эффективно реализовать как цикл с перескоками, где значение $j_{\text{стр}}$ будет использоваться для того, чтобы исключить проверку всех позиций в строке $\boldsymbol{x}$ во время перескока.

Здесь мы рассмотрим еще один подход к организации перескоков, описанный в [121], — это модификация алгоритма BMFAST, где "основная" буква $\boldsymbol{p}[m]$ заменена на наиболее редко встречаемую букву $\boldsymbol{p}[j]$.

Предположим, нам надо найти вхождения паттерна $\boldsymbol{p} = \textit{marzipan}$ в тексте $\boldsymbol{x}$ на английском языке, набранном в нижнем регистре и из которого, для простоты, удалены пробелы и знаки препинания. Алгоритм BMFAST выполняет перескоки длиной $\boldsymbol{\delta}'_1[x[i]]$, если текущая буква $\boldsymbol{x}[i]$ совпадает с $\boldsymbol{p}[m] = \boldsymbol{p}[8] = n$. Длина перескоков может быть от 1 (если $\boldsymbol{x}[i] = a$) до 8 (если в паттерне $\boldsymbol{p}$ нет буквы $\boldsymbol{x}[i]$) или до N (если $\boldsymbol{x}[i] = n$). Поскольку в английском языке буква n шестая по частоте встречаемости, то перескоки длиной N будут совершаться относительно часто. Так как такие перескоки расточительны (значительно менее эффективны, чем просто распознать равенство $\boldsymbol{x}[i] = \boldsymbol{p}[m] = n$), то желательно уменьшить часто́ты таких перескоков.

Один из возможных способов уменьшить количество перескоков длиной N — это искать в строке $\boldsymbol{x}$ букву, совпадающую с буквой $\boldsymbol{p}[1] = m$, но еще лучше выбрать для сравнения букву, которая расположена ближе к правому краю паттерна и которая имеет низкую частоту появления в строке $\boldsymbol{x}$. "Кандидатами" для такого сорта букв могут быть буква $\boldsymbol{p}[6] = p$ и, особенно, буква $\boldsymbol{p}[4] = z$. (Мы не рассматриваем буквы $\boldsymbol{p}[7] = a$ и $\boldsymbol{p}[5] = i$, поскольку эти буквы имеют еще более высокую частоту встречаемости в английском языке, чем буква n.) Для организации сравнения с этими буквами можно вычислить массив $\boldsymbol{\delta}'_1$ так,

как будто паттерн $\boldsymbol{p}$ содержит только 6 (если использовать букву $\boldsymbol{p}[6] = p$) или 4 буквы (при выборе буквы $\boldsymbol{p}[4] = z$). Конечно, в этом случае средняя длина перескоков уменьшится (поскольку исключены перескоки длиной N), а длина наибольшего перескока составит только 6 (соответственно, 4) позиций. Таким образом, возникает необходимость оценить и сравнить "стоимость" уменьшения длины перескоков и "стоимость" уменьшения количества перескоков длиной N. В работе [121] проведен соответствующий анализ и предложена методика выбора для сравнения такой буквы паттерна $\boldsymbol{p}$, которая минимизирует (по вероятности) количество перескоков. Для выполнения предварительных действий по выбору такой буквы требуется время порядка $\Theta(m)$, но это все равно приводит к относительно небольшому уменьшению общего времени сравнения паттерна и текстовой строки — уменьшение составляет всего 3–5% [121].

8.3.3 Первый алгоритм Бойера–Мура–Санди

Как показано в упражнении 8.3.5, использование стража в действительности скрывает перескоки. Все, что мы рассмотрели в этом разделе ранее, — это просто набор способов перескока, основанных на частоте встречаемости букв. Теперь рассмотрим более революционную идею: переопределение в алгоритме БМ массива $\boldsymbol{\delta}_1$ таким образом, чтобы полностью исключить необходимость выполнения буквенных сравнений с паттерном $\boldsymbol{p}$ в порядке справа налево. (Этот новый алгоритм назовем БМС1, т.е. первый алгоритм Бойера–Мура–Санди.)

Подобно всем другим модификациям алгоритма БМ, обобщающая идея алгоритма БМС1 также проста [217]. Предположим, что при текущем сравнении паттерна $\boldsymbol{p}$ и строки $\boldsymbol{x}$ буква $\boldsymbol{p}[m]$ выровнена по букве $\boldsymbol{x}[i_0]$. Тогда после обнаружения несовпадения букв следующая попытка сравнения не сможет достигнуть успеха, пока буква $\boldsymbol{x}[i_0 + 1] = \lambda$ будет совпадать с такой же буквой λ, входящей в паттерн $\boldsymbol{p}$. Следовательно, если j' — *самая правая* позиция буквы λ в паттерне $\boldsymbol{p}$, то возможен "безопасный" сдвиг паттерна вдоль строки $\boldsymbol{x}$ на $m - j' + 1$ позиций. Это наблюдение подводит к мысли вычислить на предварительном этапе алгоритма БМС1 массив $\boldsymbol{\Delta}_1$ следующим образом: для каждой буквы $\lambda \in A$ положим

$$\boldsymbol{\Delta}_1[\lambda] \leftarrow m - j' + 1,$$

где j' — самая правая позиция буквы λ в паттерне $\boldsymbol{p}$, и $\boldsymbol{\Delta}_1[\lambda] \leftarrow 0$, если буквы λ нет в паттерне $\boldsymbol{p}$. Отметим, что для любой буквы λ

$$\boldsymbol{\Delta}_1[\lambda] = \boldsymbol{\delta}_1[\lambda] + 1.$$

Нетрудно также заметить, что если при текущем сравнении, когда буква $\boldsymbol{p}[m]$ выравнивалась по букве $\boldsymbol{x}[i_0]$, обнаружено несовпадение, то величина $\boldsymbol{\Delta}_1[\boldsymbol{x}[i_0 + + 1]]$ определяет сдвиг паттерна $\boldsymbol{p}$ вправо для следующего сравнения, когда буква $\boldsymbol{p}[m]$ выравнивается по букве $\boldsymbol{x}\,[i_0 + \boldsymbol{\Delta}_1\,[\boldsymbol{x}\,[i_0 + 1]]]$.

Как показано в работе [S90], массив $\boldsymbol{\Delta}_1$ имеет ряд преимуществ по сравнению с массивом $\boldsymbol{\delta}_1$.

1. Подобно элементам массива $\boldsymbol{\delta}_1''$ из алгоритма БМХ, элементы $\boldsymbol{\Delta}_1[\lambda]$ всегда больше нуля (если λ входит в $\boldsymbol{p}$), и поэтому сдвиг паттерна вдоль строки $\boldsymbol{x}$ всегда будет ненулевым.
2. Можно ожидать, что в среднем величины сдвигов, рассчитанные на основе массива $\boldsymbol{\Delta}_1$, будут значительно большими, чем рассчитанные на основе массива $\boldsymbol{\delta}_1$. Напомним, что в алгоритме БМ обнаружение несовпадения в позиции i ведет к сдвигу паттерна $\boldsymbol{p}$ вдоль строки $\boldsymbol{x}$ на величину

$$\boldsymbol{\delta}_1[\boldsymbol{x}[i]] - (m-j) = \boldsymbol{\Delta}_1[\boldsymbol{x}[i]] - 1 - (m-j), \tag{8.1}$$

   где $m-j$ — длина совпадающих подстрок перед обнаружением несовпадения в позиции i. Поскольку в среднем

$$\boldsymbol{\Delta}_1[\boldsymbol{x}[i_0+1]] = \boldsymbol{\Delta}_1[\boldsymbol{x}[i]],$$

   то из (8.1) следует, что сдвиг на основе массива $\boldsymbol{\Delta}_1$ будет превышать сдвиг на основе массива $\boldsymbol{\delta}_1$ на величину $m-j+1$, где $m-j$ — средняя длина совпадающих подстрок.
3. Основное преимущество применения массива $\boldsymbol{\Delta}_1$: поскольку теперь величина сдвига паттерна не зависит от позиции обнаруженного несовпадения (при текущем сравнении), следовательно, величина сдвига не зависит и от порядка, в котором будут сравниваться буквы паттерна. Таким образом мы освободились от зависимости алгоритма БМ от порядка выполнения сравнений справа налево!

Как видно из этого списка "достоинств", алгоритм БМС1 можно легко реализовать так, чтобы сдвиг определялся только значениями массива $\boldsymbol{\Delta}_1$. Кроме того, можно ожидать, что этот алгоритм будет эффективнее алгоритма БМХ, поскольку средняя величина сдвига в алгоритме БМС1 больше, чем в БМХ. Первые два пункта списка "достоинств" показывают, что в простейшей форме алгоритм БМС1 может поддерживать просмотр паттерна справа налево. Однако третий пункт дает свободу в выборе порядка просмотра паттерна. Поэтому в принципе можно найти такой порядок, который приведет к более быстрому выполнению алгоритма, хотя бы в среднем. Это заставляет опять вспомнить о частоте встречаемости букв. Рассмотрим более эффективную версию алгоритма БМС1, где используется знание о частоте букв.

В наиболее общей версии алгоритма БМС1 применяется перестановка позиций $1..m$ паттерна $\boldsymbol{p}$, которая хранится в массиве $\boldsymbol{\pi} = \boldsymbol{\pi}[1..m]$. Хотя возможна *любая* перестановка позиций $1..m$, рационально применить перестановку, основанную на частоте встречаемости букв. Такая перестановка определяется следующим образом: для каждого $j \in 1..m$ определяем $\boldsymbol{\pi}[j] = j'$, где j' — позиция буквы

$\boldsymbol{p}[j']$, имеющей j-ю по величине частоту встречаемости. Здесь предполагается, что позиции в паттерне считаются справа налево. Например, для паттерна

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{p} = & m & a & r & z & i & p & a & n \end{array}$$

получаем $\boldsymbol{\pi} = 4\ 6\ 1\ 3\ 8\ 5\ 7\ 2$.

Для использования массива $\boldsymbol{\pi}$ в алгоритме БМС1 и в соответствующей процедуре сравнения достаточно просто $\boldsymbol{p}[j]$ заменить на $\boldsymbol{p}[\boldsymbol{\pi}[j]]$. Эти изменения в алгоритме предлагается сделать в упражнении 8.3.7. Отметим, что теперь сравнение паттерна $\boldsymbol{p}$ и строки $\boldsymbol{x}$ можно всегда проводить в естественном порядке $j = 1, 2, \ldots, m$, поскольку для выполнения сканирования паттерна слева направо или справа налево можно применить перестановки $\boldsymbol{\pi} = 1\ 2 \ldots n$ или $\boldsymbol{\pi} = nn - 1 \ldots 1$ соответственно.

Алгоритм БМС1 особенно привлекателен для сравнения паттернов со строками, определенными на алфавитах, для которых известно распределение частот встречаемости букв, и особенно если это распределение явно асимметрично. Его также можно скомбинировать со способами вычисления эффективных перескоков, подобными тому, который используется в алгоритме BMFAST. Такой гибридный алгоритм не только быстрый, но и относительно простой. Однако алгоритм БМС1 имеет и обратную сторону: возрастает время выполнения (по сравнению с алгоритмом БМ), поскольку используется непрямой доступ к элементам паттерна $\boldsymbol{p}$ посредством отображения $\boldsymbol{p}[\boldsymbol{\pi}[j]]$. Это дополнительное время вычислений увеличивается, когда распределение частот встречаемости букв алфавита не имеет сильно выраженной асимметрии. Кроме того, как показано в упражнении 8.3.6, требуется не менее $\Theta(n)$ дополнительного времени для предварительного вычисления массива $\boldsymbol{\pi}$, что может составить значительную величину при больших значениях m.

8.3.4 Второй алгоритм Бойера–Мура–Санди

В этом подразделе мы покажем, как освободить алгоритм БМ от требования сканировать паттерн справа налево, используя для этого модификацию массива $\boldsymbol{\delta}_2$. В результате получим второй алгоритм Бойера–Мура–Санди (назовем его для краткости БМС2), который в некоторых случаях выполняется даже быстрее, чем алгоритм БМС1.

Сначала заметим, что массив $\boldsymbol{\Delta}_1$ из алгоритма БМС1 можно скомбинировать с массивом $\boldsymbol{\delta}_2$ из стандартного алгоритма БМ. Оставляя пока в стороне вопрос о вычислении массива $\boldsymbol{\Delta}_1$, заменим в алгоритме БМ присвоения $i_0 \leftarrow i \leftarrow m$, выполняемые в начале каждого шага сравнения, на присвоения

$$i_0 \leftarrow i \leftarrow \max\left\{i_0 + \boldsymbol{\Delta}_1\left[\boldsymbol{x}\left[i_0 + 1\right]\right], i + \boldsymbol{\delta}_2[j]\right\}.$$

Как указывается в работе [219], это должно улучшить алгоритм в случае, когда известно распределение частот встречаемости букв алфавита. Можно ввести массив $\boldsymbol{\Delta}_2$, который будет обобщением массива $\boldsymbol{\delta}_2$, если использовать $\boldsymbol{\pi}[j]$ вместо j. Если, как и выше, i_0 будет обозначать позицию в строке $\boldsymbol{x}$, с которой выравнивается позиция m в паттерне $\boldsymbol{p}$, тогда для всех $h \in 1..m\boldsymbol{\pi}'\ [h] = i_0 - (m - \boldsymbol{\pi}[h])$ будет обозначать позицию в строке $\boldsymbol{x}$, с которой выравнивается позиция $\boldsymbol{p}[\boldsymbol{\pi}[h]]$. Допуская некоторую небрежность в записи, кратко можно записать ($\forall h \in 0..m$)

$$\boldsymbol{p}[\boldsymbol{\pi}[1..h]] = \boldsymbol{p}[\boldsymbol{\pi}[1]]\boldsymbol{p}[\boldsymbol{\pi}[2]] \ldots \boldsymbol{p}[\boldsymbol{\pi}[h]],$$
$$\boldsymbol{x}[\boldsymbol{\pi}'[1..h]] = \boldsymbol{x}[\boldsymbol{\pi}[1..h] + (i_0 - m)] = \boldsymbol{x}[\boldsymbol{\pi}'[1]]\boldsymbol{x}[\boldsymbol{\pi}'[2]] \ldots \boldsymbol{x}[\boldsymbol{\pi}'[h]].$$

Тогда каждое частичное совпадение паттерна $\boldsymbol{p}$ и строки $\boldsymbol{x}$ можно выразить как равенство $\boldsymbol{x}[\boldsymbol{\pi}'[1..j-1]] = \boldsymbol{p}[\boldsymbol{\pi}[1..j-1]]$ для некоторого $j \in 0..m+1$, при котором обнаруживается несовпадение

$$\boldsymbol{x}[\boldsymbol{\pi}'[1..j]] \neq \boldsymbol{p}[\boldsymbol{\pi}[1..j]]. \tag{8.2}$$

Так же, как и в алгоритме БМ, мы можем считать, что $\boldsymbol{x}$ и $\boldsymbol{p}$ заканчиваются специальными несовпадающими символами $\$_1$ и $\$_2$ соответственно. Тогда при $j = m + 1$ (в этом случае паттерн $\boldsymbol{p}$ совпадает с соответствующей подстрокой строки $\boldsymbol{x}$) выполняются равенства

$$\boldsymbol{x}[\boldsymbol{\pi}'[m+1]] = \$_1, \quad \boldsymbol{p}[\boldsymbol{\pi}[m+1]] = \$_2.$$

Следовательно, и в таком случае справедливо условие (8.2).

Используя эти обозначения и соглашения, опишем далее вычисление массива $\boldsymbol{\Delta}_2$ как обобщение вычислений, соответствующих сдвигам первого и второго типов в алгоритме БМ.

- **Сдвиг первого типа.** Если существуют наибольшее положительное целое число $j' < \boldsymbol{\pi}[j]$ и смещение $d = \boldsymbol{\pi}[j] - j'$, такие, что

  $$\boldsymbol{p}[\boldsymbol{\pi}[1..j-1] - d] = \boldsymbol{p}[\boldsymbol{\pi}[1..j-1]],$$

  и при этом $\boldsymbol{p}[j'] \neq \boldsymbol{p}[\boldsymbol{\pi}[j]]$, тогда сдвиг, порождаемый несовпадением в позиции $\boldsymbol{\pi}[j]$, определяет значение d j-го элемента массива $\boldsymbol{\Delta}_2$, т.е. $\boldsymbol{\Delta}_2[j] \leftarrow d$.

- **Сдвиг второго типа.** Если не существует целого j', такого, как описано в предыдущем пункте, тогда находится наибольшая грань $\boldsymbol{p}[\boldsymbol{\pi}[1..j']]$ строки $\boldsymbol{p}[\boldsymbol{\pi}[1..m]]$ ($0 \leqslant j' \leqslant m-1$), удовлетворяющая следующим условиям: $j' < j$, $d = \boldsymbol{\pi}[m] - \boldsymbol{\pi}[j'] > 0$ и

  $$d = \boldsymbol{\pi}[m-1] - \boldsymbol{\pi}[j'-1] = \boldsymbol{\pi}[m-2] - \boldsymbol{\pi}[j'-2] = \cdots = \boldsymbol{\pi}[m-h'+1] - \boldsymbol{\pi}[1].$$

  Используя полученное таким образом значение d, снова полагаем $\boldsymbol{\Delta}_2[j] \leftarrow$ $\leftarrow d$.

Поскольку возможна ситуация, когда $j' = 0$ (т.е. строка $\boldsymbol{p}[\boldsymbol{\pi}[1..m]]$ имеет только пустую грань), необходимо положить $\boldsymbol{\pi}[0] = 0$ для того, чтобы приведенные выше условия всегда имели смысл. В действительности, как показано в упражнении 8.3.9, если перестановка $\boldsymbol{\pi}$ соответствует порядку возрастания частот встречаемости букв алфавита, то почти всегда будет выполняться равенство $j' = 0$, т.е. в этом случае будем иметь максимальный возможный сдвиг $d = \boldsymbol{\pi}[m]$.

Нетрудно заметить, что вычисление массива $\boldsymbol{\Delta}_2$, выполненное для "переставленного" паттерна $\boldsymbol{p}[\boldsymbol{\pi}[1..m]]$, в точности соответствует вычислению массива $\boldsymbol{\delta}_2$ для исходного паттерна $\boldsymbol{p}[1..m]$. Основная трудность в вычислении массива $\boldsymbol{\Delta}_2$ возникает из-за того, что здесь позиции $\boldsymbol{\pi}[h]$ и $\boldsymbol{\pi}[h+1]$ не обязаны быть смежными, и поэтому необходимо искать величину смещения d. Вследствие этих дополнительных вычислений и непрямого доступа к буквам паттерна, предварительное вычисление массива $\boldsymbol{\Delta}_2$ требует бóльших затрат, чем вычисление массива $\boldsymbol{\delta}_2$, хотя порядок сложности вычисления массива $\boldsymbol{\Delta}_2$ остается на уровне $\Theta(m)$. Таким образом, несмотря на то что вычисление массива $\boldsymbol{\Delta}_2$ требует бóльших затрат, ничего принципиально нового в этих вычислениях нет. Поэтому подробно мы их рассматривать не будем.

Когда массив $\boldsymbol{\Delta}_2$ вычислен, сдвиг в алгоритме БМС2 можно определить по формуле

$$i_0 \leftarrow i_0 + \max\{\boldsymbol{\Delta}_1[\boldsymbol{x}[i_0+1]], \boldsymbol{\Delta}_2[\boldsymbol{\pi}[j]]\}$$

или по упрощенной формуле (предложенной в [S90])

$$i_0 \leftarrow i_0 + \boldsymbol{\Delta}_2[\boldsymbol{\pi}[j]].$$

Как показано в статье [121], даже значительно усовершенствованные версии алгоритма БМС2 конкурентоспособны только в случае длинных паттернов, когда возрастающая стоимость предварительных вычислений становится ограничительным фактором для любых алгоритмов. Эти исследования вызывают сомнения в высокой эффективности алгоритма БМС2 и приводят к разработке значительно более простого алгоритма [121], который мы назовем БМС2*.

В этом алгоритме для определения величины перескока выбирается буква $\boldsymbol{p}[j]$ или как самая правая буква паттерна (тогда $j = m$, так делается в алгоритме BMFAST, см. раздел 8.1), или на основе частоты встречаемости букв (см. выше подраздел "Наиболее эффективные перескоки"). После выполнения перескока алгоритм либо завершается, если перескок вышел за пределы строки $\boldsymbol{x}$, либо буква $\boldsymbol{p}[j]$ выравнивается по букве $\boldsymbol{x}[i_0 - (m-j)]$ и начинается побуквенное сравнение. (Здесь, как и ранее, считается, что на предыдущем шаге позиция m паттерна выравнивалась по позиции i_0 в строке $\boldsymbol{x}$.) Затем, после того как процедура *hctam* найдет несовпадение, в алгоритме БМС2* выполняется сдвиг $i_0 \leftarrow i_0 + \boldsymbol{\Delta}[j]$, где $\boldsymbol{\Delta}[j]$ определяется следующим образом:

- если существует наибольшее целое число $j' < j$, такое, что $\boldsymbol{p}[j'] = \boldsymbol{p}[j]$, тогда $\boldsymbol{\Delta}[j] = j - j'$;
- в противном случае $\boldsymbol{\Delta}[j] = j$.

Как видно из этого описания, алгоритм БМС2* очень прост с несложным предварительным этапом. Но несмотря на свою простоту этот алгоритм оказался очень эффективным на практике [121], особенно в случае использования стража в процедуре сравнения и эффективной реализации перескоков. Детали этого алгоритма оставляем для упражнения 8.3.11.

Упражнения 8.3

1. Докажите корректность алгоритма со стражем, приведенного в подразделе "Сравнение со стражем", основываясь на следующих фактах.

- В алгоритме используется значение $\boldsymbol{\delta}_1[\boldsymbol{x}[i]]$ независимо от того, установлено равенство $\boldsymbol{x}[i] = \boldsymbol{p}[m]$ или нет.
- Значение $\boldsymbol{\delta}_2[j]$ используется независимо от того, установлено равенство

$$\boldsymbol{x}[i - (m - j) + 1..i] = \boldsymbol{p}[j + 1..m]$$

или нет.

2. Профессор Шрдлу (Etaoin Shrdlu) вычислил массив вероятностей $\boldsymbol{prob}[\lambda]$, с которыми буквы λ некоторого алфавита A встречаются в произвольном месте произвольного текста, основанного на этом алфавите. Запишите алгоритм, вычисляющий $j_{\text{стр}}$ на основе массива $\boldsymbol{prob}$, и определите его сложность.

3. Рассмотрите модификацию алгоритма со стражем, где, кроме замены $j \leftarrow j_{\text{стр}}$, используется замена $i \leftarrow i - (m - j_{\text{стр}})$. Сравните эффективность модифицированного алгоритма с исходным алгоритмом.

4. Измените алгоритм КМП так, чтобы он использовал страж.

5. Измените алгоритм БМ со стражем таким образом, чтобы сравнение с $\boldsymbol{p}[j_{\text{стр}}]$ выполнялось внутри цикла, определяющего перескок. Будет ли эта версия алгоритма работать быстрее, чем алгоритм со стражем?

6. Разработайте алгоритм, который использовал бы массив $\boldsymbol{prob}$ профессора Шрдлу для вычисления массива перестановок $\boldsymbol{\pi}[1..m]$ (см. алгоритм БМС1). Какова временна́я сложность полученного алгоритма? Как следует расположить элементы в массиве $\boldsymbol{prob}$, чтобы уменьшить временну́ю сложность алгоритма?

7. Запишите алгоритм БМС1 с применением массива перестановок $\boldsymbol{\pi}$ и соответствующую процедуру сравнения. Какова временна́я сложность алгоритма БМС1 в самом худшем случае?

8. Модифицируйте алгоритм БМС1 так, чтобы он использовал перескоки, как описано в разделе 8.1. Какие трудности необходимо преодолеть в этой модификации, которые не встречались в алгоритме BMFAST?

9. Покажите, что если перестановка $\boldsymbol{\pi}$ букв паттерна $\boldsymbol{p}$ упорядочивает их в порядке возрастания частоты встречаемости этих букв, тогда строка $\boldsymbol{p}[\boldsymbol{\pi}[1..m]]$ имеет пустую грань, за исключением случая, когда $\boldsymbol{p} = \lambda^m$ для некоторой буквы λ. Докажите более общий результат: данное утверждение справедливо, если какая-либо буква входит в паттерн только подряд (без перебивки другими буквами).

10. Опишите изменения в вычислениях алгоритма КМП, где вместо массива $\boldsymbol{\beta}'$ будет использоваться его аналог массив $\mathbf{B}'$, основанный на перестановке $\boldsymbol{\pi}$ позиций паттерна $\boldsymbol{p}$. Запишите новый алгоритм КМП и процедуру *match*. Как вы думаете, будет ли новый алгоритм более эффективным на практике и в теоретическом плане, чем исходный алгоритм КМП?

11. Запишите алгоритм БМС2* для случаев, когда для определения величины перескока выбирается буква $\boldsymbol{p}[j]$ как самая правая буква паттерна (тогда $j = m$) и на основе частоты встречаемости букв (тогда $j = j_{\text{стр}}$). Для этих случаев запишите процедуру *hctam* и алгоритм вычисления массива $\boldsymbol{\Delta}$. Какова временна́я сложность алгоритма БМС2* в самом худшем случае?

8.4 Алгоритм Бойера–Мура–Гелила

В предыдущих разделах этой главы рассматривались, в основном, простые (хотя и интересные) модификации массива $\boldsymbol{\delta}_1$ из алгоритма БМ. В этом разделе рассмотрим модификации массива $\boldsymbol{\delta}_2$.

Напомним читателю, что в разделе 7.2 возможная периодичность паттерна $\boldsymbol{p}$ мешала установить безусловную верхнюю границу в $4n$ количества буквенных сравнений, выполняемых в алгоритме БМ. Здесь мы опишем другую, очень простую модификацию алгоритма БМ, которую назовем алгоритмом Бойера–Мура–Гелила (сокращенно, алгоритм БМГ). Затем сделаем простую модификацию алгоритма БМГ и получим алгоритм, который назовем, возможно нескромно, алгоритмом Бойера–Мура–Гелила–Смита (сокращенно, алгоритм БМГС). Последний алгоритм позволяет установить верхнюю границу буквенных сравнений без всяких условий, накладываемых на паттерн $\boldsymbol{p}$. В результате, с помощью алгоритма БМГС получим доказательство более строгого варианта теоремы 7.2.4.

Основная идея алгоритма БМГ [100] заключается в том, что при сравнении паттерна $\boldsymbol{p}$ со строкой $\boldsymbol{x}$ можно использовать наше знание о периоде паттерна для исключения лишних сравнений. Так, если при очередном сравнении паттерна со строкой обнаружены совпадения $\boldsymbol{x}[i+1..i+m] = \boldsymbol{p}[1..m]$ и несовпадение $\boldsymbol{x}[i] \neq \boldsymbol{p}[0]$, то осуществляем сдвиг вправо на величину $\boldsymbol{\delta}_2[0] = m - \boldsymbol{\beta}[m]$, т.е. на

период паттерна (см. раздел 1.2). После выполнения сдвига мы знаем, что префикс $\boldsymbol{p}[1..\boldsymbol{\beta}[m]]$ совпадает с соответствующей подстрокой строки $\boldsymbol{x}$, выровненной по этому префиксу, и поэтому нет необходимости проводить здесь сравнения — достаточно проверить совпадение со строкой $\boldsymbol{x}$ суффикса $\boldsymbol{p}[\boldsymbol{\beta}[m]+1..m]$ длиной $m-\boldsymbol{\beta}[m]$.

Вариант Гелила (Galil) алгоритма БМ чрезвычайно прост: после завершения очередного шага сравнения значение $\boldsymbol{\beta}[m]$ наибольшей грани паттерна $\boldsymbol{p}$ используется для определения последних $m-\boldsymbol{\beta}[m]$ позиций паттерна, которые будут проверяться при следующем вызове процедуры *hctam*. Напомним факт из раздела 7.2, что значение $\boldsymbol{\beta}[m]$ можно вычислить на предварительном этапе алгоритма как побочный продукт вычисления массива $\boldsymbol{\delta}_2$. В представленном ниже алгоритме 8.4.1 видны все изменения, которые необходимо сделать в алгоритме БМ, основное из которых будет введение новой переменной $jump$ (скачок), значение которой равно длине префикса паттерна, исключаемого из сравнения при следующем вызове процедуры *hctam*. В этом алгоритме предполагается, что процедура *hctam* при завершении сравнений возвращает значение $j=0$ независимо от того, равняется или нет переменная $jump$ нулю. Соответствующую модификацию процедуры *hctam* и доказательство корректности этого алгоритма предложено дать в упражнениях 8.4.1 и 8.4.2.

Алгоритм 8.4.1 (Алгоритм Бойера–Мура–Гелила)

```
▷ Нахождение всех вхождений паттерна p в строку x
i ← m; jump ← 0
while i ⩽ n do
    (i, j) ← hctam(i, jump, m)
    if j = 0 then
        output i + 1; jump ← β[m]
    else
        jump ← 0
    i ← i + max{δ1[x[i]], δ2[j]}
```

На большинстве тестовых задачах алгоритм БМГ выполнялся медленнее, чем алгоритм БМ, поскольку в основном цикле выполняется еще обработка переменной $jump$. Достоинства этого алгоритма проявляются, только когда большой период паттерна $\boldsymbol{p}$ часто встречается в строке $\boldsymbol{x}$. Однако алгоритм БМГ полезен в теоретическом плане: в работе [100] на его основе доказана верхняя граница в $4n$ буквенных сравнений в самом худшем случае для произвольного паттерна, независимо от того, периодический он или нет. Далее мы покажем, как небольшое изменение в этом алгоритме позволяет получить данную границу.

Переход от алгоритма БМГ к алгоритму БМГС естественен и понятен: вместо постоянного использования переменной $jump$ следует использовать ее толь-

ко в случае сдвига второго типа (см. рис. 7.4). Сдвиг этого типа легко распознается благодаря тому факту, что в этом случае выполняется неравенство $\boldsymbol{\delta}_2[j] \geqslant m$. Поэтому если после очередного шага сравнения выполняется неравенство $\boldsymbol{\delta}_2[j] \geqslant \boldsymbol{\delta}_1[\boldsymbol{x}[i]]$, то значение переменной $jump$ для следующего сравнения должно устанавливаться следующим образом: если $\boldsymbol{\delta}_2[j] \geqslant m$, тогда $jump \leftarrow j'$, в противном случае $jump \leftarrow 0$, где, как показано на рис. 7.4, j' — это длина наибольшей грани подстроки $\boldsymbol{p}[j+1..m]$. Поскольку для сдвига второго типа справедливо равенство $\boldsymbol{\delta}_2[j] = (m-j) + (m-j')$, отсюда получаем формулу для вычисления значения j':

$$j' = (m-j) - (\boldsymbol{\delta}_2[j] - m).$$

Подробности реализации модифицированного алгоритма БМГ (или, другими словами, алгоритма БМГС) оставим для упражнения 8.4.3. Отметим только, что алгоритм БМГС не намного сложнее алгоритма БМГ, но имеет некоторое преимущество (уменьшает количество буквенных сравнений) в случае реализации сдвига второго типа. Фактически, в алгоритме БМГС гарантировано, что в случае сдвига второго типа на следующем шаге число буквенных сравнений будет не более, чем на предыдущем. Во всех других случаях количество буквенных сравнений будет не больше, чем в алгоритме БМГ (и, следовательно, не больше, чем в алгоритме БМ). Таким образом, можно ожидать, что на практике алгоритм БМГС в среднем будет выполняться быстрее, чем алгоритм БМГ. В теоретическом плане алгоритм БМГС приводит к доказательству теоремы 7.2.4 для случая произвольного паттерна.

Напомним, что вследствие леммы 7.2.3 можно утверждать, что для *любого* паттерна $\boldsymbol{p}$ (т.е. независимо от того, периодический он или нет) в случае сдвига первого типа (см. рис. 7.3) всегда имеем

$$t + 1 \leqslant f + 3s, \tag{8.3}$$

где $t = m - j$ — длина текущего совпадающего суффикса $\boldsymbol{p}$, f — количество совпадающих позиций и s — длина сдвига. Мы также видели, что неравенство (8.3) справедливо во всех случаях, когда результаты сравнения порождают "длинные" сдвиги (тогда $3s \geqslant t$). Таким образом, для доказательства теоремы 7.2.4 необходимо только рассмотреть сдвиги второго типа (см. рис. 7.4) для 3-периодического паттерна $\boldsymbol{p}$, который в правой нормальной форме можно представить как $\boldsymbol{p} = \boldsymbol{v}'\boldsymbol{v}^k$, $k \geqslant 3$, где $\boldsymbol{v}$ — некратная строка. Как и в разделе 7.2 длину строки $\boldsymbol{v}$ будем обозначать как v, т.е. $v = |\boldsymbol{v}|$.

После того как мы освежили свою память фактами из главы 7, можно приступать к основному результату данного раздела.

Теорема 8.4.1. Выполнение алгоритма БМГС требует не более $4n$ буквенных сравнений.

Доказательство. Как мы видели, утверждение теоремы выполняется для всех паттернов, которые не являются 3-периодическими, и в случае длинных сдвигов и сдвигов первого типа. Поэтому предположим, что имеется период $v \leqslant m/3$ и короткий сдвиг s второго типа, порождаемый текущим сравнением. Поскольку сдвиг короткий, то $3s < t$, и так как это сдвиг второго типа, то, при найденном несовпадении символа $\boldsymbol{p}[m-t]$ с соответствующей буквой строки $\boldsymbol{x}$, этот сдвиг должен быть не менее $m-t$, т.е. $s \geqslant m-t$. На основе этих двух неравенств мы заключаем, что для выполненного сравнения 3-периодического паттерна, которое порождает короткий сдвиг второго типа, справедливы неравенства

$$t \geqslant 3m/4 \geqslant 9v/4. \tag{8.4}$$

Теперь рассмотрим сравнение, которое непосредственно предшествовало текущему сравнению. Если такого предыдущего сравнения не было или оно порождает сдвиг второго типа, то выполнение алгоритма БМГС гарантирует, что в *текущем* сравнении будут проверяться только те позиции строки $\boldsymbol{x}$, которые уже проверялись на предыдущем сравнении. Другими словами, в этом случае для текущего сравнения выполняется равенство $t = f$, так что неравенство (8.3) будет выполняться. Поэтому надо рассмотреть только предыдущее сравнение, порождающее сдвиг первого типа, как показано на рис. 8.1.

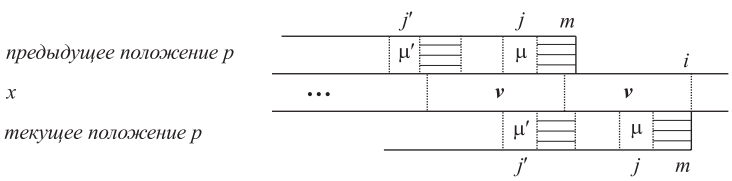


Рис. 8.1. Предыдущий сдвиг первого типа, при текущем сравнении $t \geqslant 9v/4$

Предположим, что на предыдущем шаге выполнено $t_{\text{пред}} = m - j$ буквенных сравнений и после этого рассчитан сдвиг $s_{\text{пред}} = j - j'$. Если, кроме того, предположить, что $t_{\text{пред}} \geqslant v$, тогда должна существовать подстрока

$$\boldsymbol{u} = \boldsymbol{p}[j'+1..j'+(m-j)]$$

длиной не менее v, такая, что $\boldsymbol{u} = \boldsymbol{p}[j+1..m] = \boldsymbol{u}'\boldsymbol{v}$ для некоторой строки $\boldsymbol{u}'$. Поэтому подстрока $\boldsymbol{u}$ должна иметь суффикс $\boldsymbol{v}$. Но поскольку буква $\boldsymbol{p}[j'] = \mu'$ не совпадает с буквой μ, суффикс $\boldsymbol{v}$ не может выравниваться ни с каким из k вхождений подстроки $\boldsymbol{v}$ в паттерн $\boldsymbol{p} = \boldsymbol{v}'\boldsymbol{v}^k$. Поэтому $\boldsymbol{v}$ совпадает со своим циклическим сдвигом $R_j(\boldsymbol{v})$, $0 < j \leqslant n-1$ (см. раздел 1.4). Но по теореме 1.4.2 это возможно только тогда, когда $\boldsymbol{v}$ является кратной строкой, что противоречит

определению данной строки $\boldsymbol{v}$. На основе этого противоречия делаем заключение, что неверно предположение о том, что $t_{\text{пред}} \geqslant v$. Поэтому далее считаем, что $t_{\text{пред}} = |\boldsymbol{u}| < v$.

Теперь сосредоточим внимание на значении $s_{\text{пред}}$. Сначала заметим, что подстрока $\boldsymbol{u}$ не может быть суффиксом строки $\boldsymbol{v}$, поскольку она не может находиться левее несовпадения $\mu' \neq \mu$. Следовательно, строка $\mu'\boldsymbol{u}$ будет или подстрокой строки $\boldsymbol{v}$ (тогда она при каждом вхождении в строку $\boldsymbol{v}$ располагается левее подстроки $\boldsymbol{p}[j+1..m]$), или подстрокой некоторого циклического сдвига строки $\boldsymbol{v}$ (тогда она также будет подстрокой строки $\boldsymbol{v}^2$). В обоих случаях самая правая позиция $j' + (m-j)$ в строке $\mu'\boldsymbol{u}$ будет находиться от позиции m в $\boldsymbol{p}$ на меньшем расстоянии, чем v. Таким образом, для предыдущего сдвига имеем

$$s_{\text{пред}} = m - (j' + (m-j)) = j - j' < v.$$

Из неравенств (8.4) следует, что подстрока $\boldsymbol{v}^2$ строки $\boldsymbol{x}$ совпадает с суффиксом $\boldsymbol{v}^2$ паттерна $\boldsymbol{p}$ при текущем выравнивании (позиция i на рис. 8.1). Поскольку $s_{\text{пред}} < v$, то подстрока $\boldsymbol{v}$ строки $\boldsymbol{x}$ должна быть левее крайней правой позиции паттерна при предыдущем его положении, как показано на рис. 8.1. Вследствие корректной работы алгоритма БМ (и, следовательно, алгоритма БМГС) такая ситуация невозможна (так как это спровоцировало бы сдвиг паттерна влево). Отсюда делаем вывод, что на предыдущем шаге не мог быть сдвиг первого типа. ■

Читатель может сравнить это доказательство с доказательством леммы 7.2.3 — и здесь и там используются одни и те же идеи.

Как будет показано в следующем разделе, алгоритм БМГС можно рассматривать как упрощенную версию более общего алгоритма, так называемого алгоритма Турбо-БМ, который в действительности требует всего $2n$ буквенных сравнений.

Кроме алгоритма Турбо-БМ, алгоритм БМГ инспирировал разработку по крайней мере еще одного алгоритма (назовем его БМГАГ, см. [16]). В этом алгоритме сохраняется информация о нескольких предыдущих этапах сравнения, что дает возможность сравнить паттерн $\boldsymbol{p}$ со строкой $\boldsymbol{x}$ путем использования не более $2n - m + 1$ буквенных сравнений. К сожалению, этот результат достигается алгоритмом БМГАГ только при условии выполнения дополнительных вычислений порядка $9n$ "небуквенных" сравнений и других вычислений на предварительном этапе алгоритма. С учетом этих обстоятельств данный алгоритм не может быть конкурентом алгоритму Турбо-БМ ни в практическом, ни в теоретическом планах. Однако дальнейшее совершенствование алгоритма БМГАГ, предложенное в работах [65, 60], позволило значительно сократить и упростить вычисления предварительного этапа, а также уменьшить верхнюю границу буквенных сравнений до $1{,}5n$.

Упражнения 8.4

1. Перепишите процедуру *hctam* для ее использования в алгоритме БМГ.
2. Докажите корректность алгоритма БМГ на основе доказанной корректности алгоритма БМ.
3. Запишите алгоритм БМГС и докажите его корректность.
4. Применив небольшую изобретательность, можно попытаться уменьшить количество буквенных сравнений в алгоритме БМГС, выполняемых в случае, когда на предыдущем шаге имел место сдвиг первого типа. Основная трудность, которую надо преодолеть, заключается в том, что если в случае реализации на предыдущем шаге сдвига второго типа известно, что подстрока совпадающих символов является префиксом паттерна $\boldsymbol{p}$, то в случае сдвига первого типа можно только утверждать, что эта подстрока входит в $\boldsymbol{p}$. Здесь не обойтись только одной дополнительной переменной $jump$ — надо еще хранить значение самой правой позиции подстроки совпадающих символов.

   Запишите этот новый алгоритм (назовем его БМГС*) и процедуру *hctam* для этого алгоритма. Очевидно, что алгоритм БМГС* никогда не будет использовать больше буквенных сравнений, чем алгоритм БМГС. Но будет ли на практике этот алгоритм более эффективным, чем алгоритм БМГС?

8.5 Алгоритм Турбо-БМ

Как сказано в предыдущем разделе, в алгоритме БМГС можно уменьшить количество буквенных сравнений (см. упражнение 8.4.4 и краткое описание алгоритма БМГС*). В этом разделе мы покажем, как можно модифицировать алгоритм БМГС путем организации хранения информации о предыдущем этапе сравнения. В этом случае не только уменьшается количество используемых буквенных сравнений, но иногда и увеличивается длина сдвига паттерна $\boldsymbol{p}$ вдоль строки $\boldsymbol{x}$. Этот новый алгоритм назовем алгоритмом Турбо-БМ, а увеличенные сдвиги — ***турбо-сдвигами*** [73].

Как показано на рис. 8.2 и 8.3, после частичного совпадения подстроки $\boldsymbol{t} = \boldsymbol{p}[j+1..m]$ с буквами строки $\boldsymbol{x}$ происходит соответственно сдвиг первого или второго типов, и далее наблюдается следующее совпадение подстроки $\boldsymbol{t}'$ с буквами строки $\boldsymbol{x}$. Здесь, как и ранее, используем обозначения $t = |\boldsymbol{t}|$, $s = |\boldsymbol{s}|$ и $t' = |\boldsymbol{t}'|$.

Случай $t' = s$ в алгоритме БМГС* рассматривается особо, поскольку тогда или суффикс $\boldsymbol{ts}$ паттерна $\boldsymbol{p}$ совпадает с буквами строки $\boldsymbol{x}$ и остается проверить только подстроку $\boldsymbol{p}[1..j']$ (если предыдущий сдвиг был сдвигом первого типа), или весь паттерн $\boldsymbol{p}$ совпадает с буквами строки $\boldsymbol{x}$ (если предыдущий сдвиг был сдвигом второго типа).

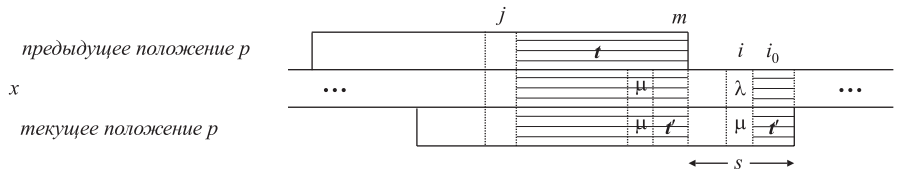


Рис. 8.2. Предыдущий сдвиг первого типа

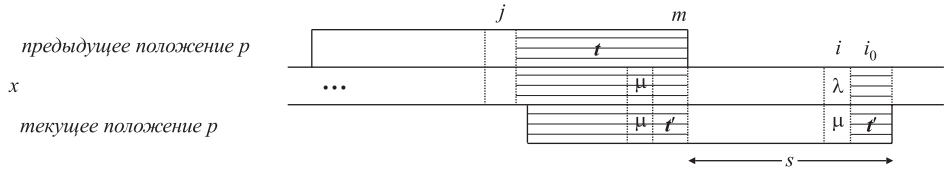


Рис. 8.3. Предыдущий сдвиг второго типа

Чтобы познакомиться с турбо-сдвигами, рассмотрим случай $t' < s$, когда зафиксировано несовпадение

$$\boldsymbol{p}[m - t'] = \mu \neq \lambda = \boldsymbol{x}[i_0 - t']. \tag{8.5}$$

Здесь строка $\mu\boldsymbol{t}'$ является суффиксом строки $\boldsymbol{s}$. Привлекая дополнительное условие $t' < t$, мы видим, что строка $\mu\boldsymbol{t}'$ является также суффиксом строки $\boldsymbol{t}$, совпадающей с подстрокой строки $\boldsymbol{x}$, и, в частности, для "общей" буквы μ имеем

$$\boldsymbol{p}[m - t'] = \boldsymbol{p}[(m - s) - t'] = \mu = \boldsymbol{x}[(i_0 - s) - t']. \tag{8.6}$$

Сравнивая (8.5) и (8.6), видим, что буквы μ и λ в строке $\boldsymbol{ts}$ отделены друг от друга на s позиций. Напомним (см. раздел 7.2), что строка $\boldsymbol{ts}$ с необходимостью имеет период s. Поэтому буквы μ и λ, стоящие в этих позициях, не могут совпасть в $\boldsymbol{ts}$. Следовательно, можно без опаски сдвинуть паттерн вправо на $i - s$ позиций, где находится буква μ.

На рис. 8.2 видно, что в случае, когда предыдущее сравнение порождает сдвиг первого типа, букву $\boldsymbol{p}[m - s - t]$ можно сдвинуть вправо на $t - t'$ позиций, тогда для следующего сравнения буква $\boldsymbol{p}[m]$ выравнивается с буквой $\boldsymbol{x}[i + t]$. Это соответствует турбо-сдвигу

$$i \leftarrow i + t. \tag{8.7}$$

С другой стороны, на рис. 8.3 видно, что в случае, когда предыдущее сравнение порождает сдвиг второго типа, букву $\boldsymbol{p}[1]$ можно сдвинуть вправо на $m - s - t'$ позиций, т.е. для следующего сравнения $\boldsymbol{p}[m]$ выравнивается с $\boldsymbol{x}[i + m - s]$. Это соответствует турбо-сдвигу

$$i \leftarrow i + (m - s). \tag{8.8}$$

Отметим, что для сдвига первого типа $t \leqslant m - s$, тогда как для сдвига второго типа $m - s \leqslant t$. Поскольку $t = m - j$, присваивания

$$s \leftarrow \boldsymbol{\delta}_2[j] - (m - j)$$
$$jump \leftarrow m - \max\{j, s\}$$

для каждого типа сдвига определят число позиций на расстоянии s от правого конца паттерна $\boldsymbol{p}$ (выровненного для следующего сравнения), не требующих проверки. Это число хранится в переменной $jump$.

Теперь пришло время записать весь алгоритм Турбо-БМ.

Алгоритм 8.5.1 (Алгоритм Турбо-Бойера–Мура)

▷ *Нахождение всех вхождений паттерна* $\boldsymbol{p}$ *в строку* $\boldsymbol{x}$
$i \leftarrow m$; $jump \leftarrow 0$
while $i \leqslant n$ **do**
  $(i, j) \leftarrow hctam(i, s, jump, m)$
  if $j = 0$ **then output** $i + 1$
  $shift \leftarrow \max\{jump, \boldsymbol{\delta}_1[\boldsymbol{x}[i]]\}$
  $i \leftarrow i + \max\{\boldsymbol{\delta}_2[j], shift\}$
  if $\boldsymbol{\delta}_2[j] < shift$ **then**
    $jump \leftarrow 0$
  else
    $s \leftarrow \boldsymbol{\delta}_2[j] - (m - j)$; $jump \leftarrow m - \max\{j, s\}$

Отметим, что в данном алгоритме не требуется этап предварительных вычислений, который присутствовал в алгоритме БМ, — для хранения дополнительной нужной информации используются переменные s и $jump$. Верхняя граница в $4n$ буквенных сравнения, показанная на алгоритме БМГС*, конечно, справедлива и для алгоритма Турбо-БМ. Но фактически, как отмечалось ранее, с небольшими усилиями можно показать [73], что этот алгоритм требует не более $2n$ буквенных сравнений. Теоретически в самом худшем случае алгоритм Турбо-БМ выполняет не больше буквенных сравнений, чем алгоритм КМП (см. раздел 7.1), но в среднем, как отмечено ранее (раздел 7.5), его реализация должна быть более быстрой.

В работе [73] также предлагается модификация алгоритма Турбо-БМ, где используется ориентированный ациклический граф слов (см. раздел 5.3.1) паттерна $\boldsymbol{p}$ для определения более эффективных значений переменных s и $jump$. Такой алгоритм на практике выполняется более быстро, чем алгоритм Турбо-БМ, но он требует предварительного этапа для построения графа слов.

Упражнения 8.5

1. Перепишите процедуру *hctam* для ее использования в алгоритме Турбо-БМ (и убедитесь, что это очень простая процедура!).

8.6 Наследники алгоритма КМП

После того как мы рассмотрели несколько вариантов алгоритма БМ, в этом разделе опишем одну модификацию алгоритма КМП (см. раздел 7.1), которая значительно (от $2n$ до $1{,}5n$) уменьшает количество буквенных сравнений, выполняемых в самом худшем случае в процессе решения задачи нахождения всех вхождений паттерна $\boldsymbol{p}$ в текстовую строку $\boldsymbol{x}$. Хотя при описании этого алгоритма мы будем основываться на идеях, высказанных в работе [13], впервые он был представлен в тезисах к докторской диссертации Ханката (Hancart, [110]). Поэтому данный алгоритм будем называть алгоритмом Кнута–Морриса–Пратта–Ханката (сокращенно, КМПХ).

Основная идея алгоритма КМПХ вытекает из наблюдения, сделанного в начале раздела 7.1, где в первый раз был затронут вопрос о сложности алгоритма КМП. Напомним, что сравнение паттерна $\boldsymbol{p} = a^{m-1}b$ со строкой $\boldsymbol{x} = a^n$ требует $2n - m$ буквенных сравнений, т.е. столько же, сколько требуется в алгоритме КМП в самом худшем случае, как показано в разделе 7.1 позднее. Чтобы исключить такие случаи, в алгоритме КМПХ паттерн $\boldsymbol{p}$ разбивается на две части, которые затем обрабатываются более-менее самостоятельно:

- префикс максимальной длины вида $\lambda^{m'-1}$, где $1 < m' < m$, λ — некоторая буква алфавита;
- непустой суффикс $\boldsymbol{p}[m'..m]$, где $\boldsymbol{p}[m'] \neq \lambda$.

В случае, когда $\boldsymbol{p} = \lambda^m$, полагаем $m' \leftarrow 1$, т.е. здесь префикс пустой и суффикс $\boldsymbol{p}[m'..m] = \lambda^m$.

Используем обозначение в виде тройки (i, j_L, j_R) для отслеживания частичных совпадений обеих частей паттерна (левой и правой частей) с подстрокой $\boldsymbol{x}[i+1..i+m]$, как показано на рис. 8.4:

- j_L — позиция в префиксе паттерна $\boldsymbol{p}$, такая, что $j_L \in 1..m'$ и

$$\boldsymbol{p}[1..j_L - 1] = \boldsymbol{x}[i+1..i+(j_L-1)];$$

- j_R — позиция в суффиксе паттерна $\boldsymbol{p}$, такая, что $j_R \in m'..m+1$ и

$$\boldsymbol{p}[m'..j_R - 1] = \boldsymbol{x}[i+m'..i+(j_R-1)].$$

Начальное значение тройки (i, j_L, j_R) на момент старта алгоритма КМПХ полагается равным $(0, 1, m')$, отражая тот факт, что в этот момент не зафиксировано каких-либо совпадений с символами строки $\boldsymbol{x}$ ни префикса $\boldsymbol{p}[1..m'-1]$, ни суффикса $\boldsymbol{p}[m'..m]$.

Выполнение алгоритма КМПХ подобно выполнению алгоритма ПМК: паттерн также сдвигается вдоль строки $\boldsymbol{x}$ слева направо без возможности вернуться назад, на каждом этапе сравнение паттерна $\boldsymbol{p}$ со строкой $\boldsymbol{x}$ выполняется в окне

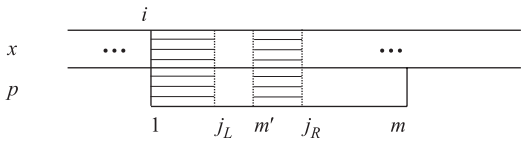


Рис. 8.4. Частичные совпадения $\boldsymbol{p}[1..j_L-1]$ и $\boldsymbol{p}[m'..j_R-1]$ левой и правой частей паттерна соответственно

длиной m, которое начинается от позиции $i+1$ строки $\boldsymbol{x}$. Как и в алгоритме КМП, в данном алгоритме сравнение букв происходит слева направо, но по отдельности в двух "подокнах", определяемых значением m'. Подробнее эти вычисления можно описать как процесс изменения значений троек $t=(i,j_L,j_R)$, при этом естественно выделить три различных случая в зависимости от значения j_R.

1. $j_R = m'$ (это означает, что на текущем этапе сравнения ни одна из букв суффикса $\boldsymbol{p}[m'..m]$ не совпадает с буквами строки $\boldsymbol{x}$).

В этом случае сравниваем $\boldsymbol{p}[j_R]$ с $\boldsymbol{x}[i+j_R]$. Если обнаружено совпадение, то получаем новую тройку $t=(i,j_L,j_R+1)$. Если же совпадения нет, то тройка имеет вид $t=(i+1,j'_L,m')$, где $j'_L=\max\{1,j_L-1\}$. Последняя тройка соответствует сдвигу паттерна на одну позицию вправо, при этом после сдвига выполняется сравнение в левом подокне с $\boldsymbol{p}[1..j'_L-1]=\lambda^{j'_L-1}$.

2. $m' < j_R < m+1$ (т.е. на текущем этапе сравнения подстрока $\boldsymbol{p}[m'..j_R-1]$ совпадает с буквами строки $\boldsymbol{x}$).

Далее сравниваем $\boldsymbol{p}[j_R]$ с $\boldsymbol{x}[i+j_R]$. Если обнаружено совпадение, то получаем новую тройку $t=(i,j_L,j_R+1)$ (такую же, как в п. 1).

Теперь предположим, что $\boldsymbol{p}[j_R] \neq \boldsymbol{x}[i+j_R]$. В таком случае следует выполнить сдвиг паттерна вправо на максимально возможное число позиций. Поскольку

$$\boldsymbol{x}[i+m'] = \boldsymbol{p}[m'] \neq \boldsymbol{p}[j] = \lambda, \quad \forall j \in 1..m'-1,$$

этот сдвиг не может иметь длину менее m'.

Напомним, что в алгоритме КМП далее вычисляется $\boldsymbol{\beta}''[j]$, при этом или $\boldsymbol{\beta}''[j]-1$ равно длине наибольшей грани подстроки $\boldsymbol{p}[1..j-1]$, такой, что $\boldsymbol{p}[\boldsymbol{\beta}''[j]] \neq \boldsymbol{p}[j]$, или $\boldsymbol{\beta}''[j]=0$, если такой грани не существует. Обозначим через q период подстроки $\boldsymbol{p}[1..j_R-1]$. Тогда (см. раздел 1.2) $(j_R-1)-q$ определяет длину наибольшей грани подстроки $\boldsymbol{p}[1..j_R-1]$. Так как $j_R-1-q \geqslant \boldsymbol{\beta}''[j_R]-1$, то

$$j_R - \boldsymbol{\beta}''[j_R] \geqslant q.$$

Но из определения m' следует, что $q \geqslant m'$. Поэтому

$$j_R - \boldsymbol{\beta}''[j_R] \geqslant m'. \tag{8.9}$$

Так же как и в стандартном алгоритме КМП, сдвиг будет равен $j_R - \boldsymbol{\beta}''[j_R]$ позициям. После сдвига сначала со строкой $\boldsymbol{x}$ будет сравниваться префикс паттерна длиной $\boldsymbol{\beta}''[j_R] - 1$. Если $\boldsymbol{\beta}''[j_R] - 1 < m'$, то $\boldsymbol{p}[1..\boldsymbol{\beta}''[j_R] - 1] = \lambda^{\boldsymbol{\beta}''[j_R]-1}$, и новая тройка будет иметь вид

$$t = \left(i + (j_R - \boldsymbol{\beta}''[j_R]), \max\{\boldsymbol{\beta}[j_R], 1\}, m'\right).$$

Если же $\boldsymbol{\beta}''[j_R] - 1 < m'$, то новая тройка вычисляется как

$$t = (i + (j_R - \boldsymbol{\beta}''[j_R]), m', \boldsymbol{\beta}''[j_R]).$$

3. $j_R = m + 1$ (суффикс $\boldsymbol{p}[m'..m]$ полностью совпадает с буквами строки $\boldsymbol{x}$). Если, кроме того, $j_L = m'$, то это означает, что паттерн $\boldsymbol{p} = \boldsymbol{p}[1..m' - 1]\ \boldsymbol{p}[m'..m]$ полностью совпадает с подстрокой строки $\boldsymbol{x}$, которая начинается с позиции $i+1$ (т.е. найдено одно из вхождений паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$). Если конец строки $\boldsymbol{x}$ не достигнут, то для следующего этапа сравнений тройка t вычисляется так же, как в п. 2 для случая $\boldsymbol{p}[j_R] \neq \boldsymbol{x}[i + j_R]$.

Если $j_L < m'$, тогда выполняется сравнение $\boldsymbol{p}[j_L]$ с $\boldsymbol{x}[i + j_L]$. Если обнаружено совпадение этих букв, то новая тройка равна $t = (i, j_L + 1, j_R)$. В противном случае t вычисляется так же, как в п.2.

Вычисления, описанные в этих пунктах, используются в следующем алгоритме (процедура $match$ здесь похожа на аналогичную процедуру в алгоритме КМП, ее оставим для упражнения 8.6.2).

Алгоритм 8.6.1 (Алгоритм Кнута–Морриса–Пратта–Ханката)

```
▷ Нахождение всех вхождений паттерна p в строку x
(i, j_L, j_R) ← (0, 1, m')
while i ⩽ n − m do
    ▷ Поиск совпадений подстроки x с суффиксом p[m'..m]
    j_R ← match(i, j_R, m)
    ▷ Если p[m'..m] полностью совпадает с подстрокой x,
    ▷ то ищутся совпадения с префиксом p[1..m' − 1]
    if i = m + 1 then
        j_L ← match(i, j_L, m' − 1)
        if j_L = m' then output i + 1
    ▷ Переопределение троек t в соответствии с пп. 1–3
    i ← i + (j_R − β''[j_R])
    if j_R = m' then
```

$j_L \leftarrow \max\{1, j_L - 1\}$
else if $\boldsymbol{\beta}''[j_R] \leqslant m'$ **then**
$(j_L, j_R) \leftarrow \bigl(\max\{\boldsymbol{\beta}''[j_R], 1\}, m'\bigr)$
else
$(j_L, j_R) \leftarrow (m', \boldsymbol{\beta}''[j_R])$

Отметим, что в этом алгоритме переменная i на каждом шаге возрастает не менее, чем на 1, поэтому алгоритм обязательно должен завершиться. Предполагая, что тройки t пересчитываются корректно, можно утверждать, что справедлива следующая теорема.

Теорема 8.6.1. Алгоритм 8.6.1 корректно вычисляет все вхождения заданного непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$. ■

Чтобы вычислить количество буквенных сравнений, выполняемых в алгоритме КМПХ, введем величину C_i, равную количеству буквенных сравнений, которые выполняются на текущем i-м шаге цикла **while**. Если обозначить через (i, j_L, j_R) значения тройки t в начале цикла, а через (i, j_L, j_R^*) — значения после выполнения процедуры $match(i, j_R, m)$, тогда вычислить C_i можно следующим образом.

1. Если $j_R^* = m'$, тогда на текущем шаге цикла **while** имеет место только одно буквенное сравнение в процедуре *match*. Поэтому здесь $C_i = 1$.
2. Если $j_R^* \in m' + 1..m$, тогда $C_i = j_R^* - j_R + 1$.
3. Если $j_R^* = m + 1$, то это означает, что найдено полное совпадение суффикса $\boldsymbol{p}[m'..m]$ с соответствующей подстрокой строки $\boldsymbol{x}$. Поэтому процедура *match* вызывается также для проверки префикса $\boldsymbol{p}[1..m' - 1]$ как *match*$(i, j_L, m' - 1)$. В этом случае, если не используется специальный символ $\boldsymbol{p}[m + 1] = \$$,

$$C_i \leqslant (m - j_R + 1) + (m' - j_L).$$

Для удобства последующих вычислений, введем еще одну величину

$$Q_i = 3i/2 + j_L/2 + j_R, \tag{8.10}$$

изменение ΔQ_i которой за время выполнения одного шага цикла **while** можно вычислить по следующим формулам.

1. Если $j_R^* = m'$, тогда, как нетрудно заметить, $i^* = i + 1$ и $j_R^* = j_R$, при этом значение j_L может уменьшиться не более, чем на 1. Поэтому

$$\Delta Q_i \geqslant (3 \times 1)/2 - 1/2 = 1.$$

2. В случае, когда $j_R^* > m'$ и $\boldsymbol{\beta}''[j_R^*] \leqslant m'$, паттерн $\boldsymbol{p}$ сдвигается вправо на $j_R^* - \boldsymbol{\beta}''[j_R^*]$ позиций, тогда как j_L изменится не более, чем на $\max\{\boldsymbol{\beta}''[j_R^*], 1\}$, а j_R уменьшится до величины m'. Следовательно,

$$\begin{aligned}\Delta Q_i &\geqslant 3(j_R^* - \boldsymbol{\beta}''[j_R^*])/2 + (\boldsymbol{\beta}''[j_R^*] - j_L)/2 + (m' - j_R) = \\ &= (j_R^* - \boldsymbol{\beta}''[j_R^*]) + (j_R^* - j_L)/2 + (m' - j_R).\end{aligned}$$

3. Наконец, если $j_R^* > m'$ и $\boldsymbol{\beta}''[j_R^*] > m'$, тогда находим

$$\begin{aligned}\Delta Q_i &= 3(j_R^* - \boldsymbol{\beta}''[j_R^*])/2 + (m' - j_L)/2 + (\boldsymbol{\beta}''[j_R^*] - j_R) = \\ &= (j_R^* - j_R) + (j_R^* - \boldsymbol{\beta}''[j_R^*])/2 + (m' - j_L)/2.\end{aligned}$$

Теперь покажем, что количество буквенных сравнений, выполняемых на любом шаге цикла **while**, не превосходит величины ΔQ_i.

Лемма 8.6.2. В цикле **while** алгоритма 8.6.1 для любого i выполняется неравенство $C_i \leqslant \Delta Q_i$.

Доказательство. Рассмотрим пять случаев, составленных из трех возможных вариантов вычисления величины C_i и трех вариантов вычисления величины ΔQ_i. Обозначим эти случаи как (11), (22), (23), (32) и (33), где первая цифра обозначает вариант вычисления величины C_i, а вторая — величины ΔQ_i.

(11) В этом случае мы немедленно получаем $\Delta Q_i \geqslant 1 = C_i$.

(22) В этом случае

$$\Delta Q_i - C_i \geqslant (m' - \boldsymbol{\beta}''[j_R^*] - 1) + (j_R^* - j_L)/2 \geqslant -1/2,$$

поскольку $j_R^* > m' \geqslant j_L$ и $\boldsymbol{\beta}''[j_R^*] \leqslant m'$. Но так как разность $\Delta Q_i - C_i$ должна быть целым числом, отсюда получаем $\Delta Q_i - C_i \geqslant 0$.

(23) Здесь, поскольку $j_R^* - \boldsymbol{\beta}''[j_R^*] \geqslant 1$ и $m' - j_L \geqslant 0$, имеем

$$\Delta Q_i - C_i \geqslant (j_R^* - \boldsymbol{\beta}''[j_R^*])/2 + (m' - j_L)/2 - 1 \geqslant -1/2.$$

Поэтому, как и в случае (22), $\Delta Q_i - C_i \geqslant 0$.

(32) В этом случае $j_R^* = m + 1$, поэтому

$$\Delta Q_i - C_i \geqslant (m+1)/2 - -\boldsymbol{\beta}''[m+1] = ((m+1) - \boldsymbol{\beta}''[m+1])/2 - \boldsymbol{\beta}''[m+1]/2.$$

Далее используем неравенство (8.9), которое справедливо и в случае $j_R^* = m + 1$, и напомним, что в варианте 2 вычисления ΔQ_i выполняется неравенство $\boldsymbol{\beta}''[m+1] \leqslant m'$. Поэтому $\Delta Q_i - C_i \geqslant (m' - m')/2 = 0$, что и требовалось доказать.

(33) В данном случае имеем

$$\Delta Q_i - C_i \geqslant ((m+1) - \boldsymbol{\beta}''[m+1])/2 - (m' - j_L)/2.$$

Снова применяя неравенство (8.9), получаем $\Delta Q_i - C_i \geqslant (m' - m' + j_L)/2 \geqslant \geqslant 0$.

Лемма доказана. ■

Теперь можем сформулировать и доказать основной результат данного раздела.

Теорема 8.6.3. Алгоритм 8.6.1 вычисляет все вхождения непустого паттерна $\boldsymbol{p}$ в заданную строку $\boldsymbol{x}$ за время порядка $O(n-m)$, при этом требуется $\Theta(m)$ дополнительной памяти и выполняется не более $\lfloor (3n-m)/2 \rfloor$ сравнений с элементами строки $\boldsymbol{x}$.

Доказательство. Очевидно, что, просуммировав по всем шагам цикла **while**, получим значение $C = \Sigma_i C_i$ общего количества буквенных сравнений, выполняемых в алгоритме 8.6.1. Из леммы 8.6.2 следует, что $C \leqslant \Sigma_i \Delta Q_i = Q_{\text{кон}} - Q_{\text{нач}}$, где $Q_{\text{кон}}$ — значение Q в конце последнего шага цикла **while**, $Q_{\text{нач}}$ — значение Q в начале первого шага цикла. Поскольку начальная тройка имеет вид $t = (0, 1, m')$, то из определения (8.10) следует, что $Q_{\text{нач}} = 1/2 + m'$. Значение $Q_{\text{кон}}$ удовлетворяет неравенству

$$Q_{\text{кон}} \leqslant 3(n-m)/2 + m'/2 + m.$$

Отсюда получаем требуемую оценку

$$C \leqslant (3n-m)/2 - (m'+1)/2.$$

■

Упражнения 8.6

1. Запишите алгоритм с временем выполнения порядка $O(m)$, вычисляющий m' для паттерна $\boldsymbol{p}[1..m]$.
2. Перепишите процедуру *match* для ее использования в алгоритме КМПХ так, чтобы минимизировать количество используемых буквенных сравнений (т.е. вопросы эффективности кода должны стоять на втором плане после минимизации числа буквенных сравнений).

8.7 Создайте собственный алгоритм!

Даже не очень внимательный читатель мог заметить, что наше обсуждение возможностей "наследников" алгоритмов БМ и КМП является далеко не исчерпывающим, даже если принять во внимание только вопросы, относящиеся к их

использованию на практике. Вне поля нашего внимания остались, например, такие вопросы. Как изменятся алгоритмы Турбо-БМ и КМПХ, если в них ввести перескоки? Как сравнить алгоритм Турбо-БМ с алгоритмом БМХ? Как сравнить алгоритмы, основанные на частоте встречаемости букв, с алгоритмами, где используются стратегии поиска, основанные на значениях массива $\boldsymbol{\delta}_2$? Быстрее ли в среднем выполняется алгоритм Турбо-БМ по сравнению с алгоритмом БМГС*? Если ответ на последний вопрос утвердительный (как предполагается в работе [121]), то означает ли это, что алгоритмы с одной (но хорошей) стратегией определения сдвига (например, только на основе массива $\boldsymbol{\Delta}_1$) предпочтительнее алгоритмов, которые используют несколько стратегий определения сдвига? Как влияют на эффективность алгоритмов их программная реализация и выбор аппаратной и программной платформ?

На самом деле при изучении алгоритмов точного сравнения паттернов со строками можно сформулировать десятки подобных вопросов, связанных с эффективностью их выполнения. К счастью, часть сизифова труда по получению ответов на такие вопросы взяли на себя авторы работы [121], предложив классификацию алгоритмов точного сравнения паттернов со строками, в рамках которой можно представить и сравнить многие, в том числе наиболее важные алгоритмы этой категории. Позднее большой экспериментальный материал был опубликован в работах [153, 154, 177], а также представлен на Web-узле [47]. В этом разделе мы покажем классификацию алгоритмов, предложенную в [121]. Согласно этой классификации, алгоритмы локализации паттернов $\boldsymbol{p}$ в строках $\boldsymbol{x}$ можно достаточно точно описать на основании трех основных параметров:

- тип перескока;
- порядок, в котором выполняется сравнение букв в паттерне;
- способ определения сдвига.

В табл. 8.1–8.3 представлены обозначения и соответствующие описания значений этих параметров, на основе которых можно определить многие (но не все) алгоритмы, описанные в этой и предыдущей главах.

Таблица 8.1. Типы перескоков

Тип	Описание
нет	
$\boldsymbol{\delta}'_1$	как в алгоритме BMFAST
редкий	перескок к следующей редко встречающейся букве [119]
мин.ст	перескок к букве с минимальной стоимостью [121]

Таблица 8.2. Порядок сравнения

Порядок	Описание
впр	слева направо, как в алгоритме КМП
обр	справа налево, как в алгоритме БМ
$\boldsymbol{\pi}$	использование массива перестановок, основанного на частоте встречаемости букв
впр + *стр*	*впр* совместно со стражем (см. раздел 8.3)
обр + *стр*	*обр* совместно со стражем
$\boldsymbol{\pi}$ + *стр*	массив перестановок совместно со стражем

Таблица 8.3. Способы определения сдвигов

Способ	Описание
$+1$	постоянный сдвиг паттерна на одну позицию вправо
$\boldsymbol{\beta}'$	сдвиг определяет грань подстроки совпадающих букв (как в алгоритме КМП)
$\boldsymbol{\beta}''$	для вычисления сдвига используется грань и определенная позиция (как в алгоритме КМП)
$\boldsymbol{\delta}_1\boldsymbol{\delta}_2$	сдвиг, как в исходном алгоритме БМ
$\boldsymbol{\delta}_2$	сдвиг, как в алгоритме БМ, но только на основе $\boldsymbol{\delta}_2$
$\boldsymbol{\delta}_1''$	сдвиг на основе модифицированного массива $\boldsymbol{\delta}_1$ (алгоритм БМХ)
$\boldsymbol{\Delta}_1$	сдвиг на основе модифицированного массива $\boldsymbol{\delta}_1$ (подход Санди)
$\boldsymbol{\Delta}_2$	сдвиг на основе модифицированного массива $\boldsymbol{\delta}_2$ с использованием массива $\boldsymbol{\pi}$ (подход Санди)
$\boldsymbol{\Delta}$	сдвиг на основе перескока (подход Санди)
турбо	сдвиг на основе массивов $\boldsymbol{\delta}_1$, $\boldsymbol{\delta}_2$ и турбо-сдвига

Используя эти обозначения, можно охарактеризовать алгоритмы сравнения с паттерном в терминах троек

<перескок, порядок сравнения, сдвиг>,

где каждый параметр тройки принимает одно из значений, приведенных в соответствующих таблицах. Например, простой алгоритм из раздела 2.2 можно описать в виде тройки $<$*нет*, *впр*, $+1>$, а одна из характеризаций алгоритма BMFAST имеет вид $<\boldsymbol{\delta}_1', $ *обр*, $\boldsymbol{\delta}_1\boldsymbol{\delta}_2>$, тогда как алгоритм БМС2* можно представить как $<$*наим.ст*, *обр*, $\boldsymbol{\Delta}>$.

Отметим, что общее количество различных троек (следовательно, и количество различных алгоритмов), которые можно составить на основе значений пара-

метров, данных в этих трех таблицах, составляет $240 = 4 \times 6 \times 10$. Даже с учетом того, что некоторые тройки невозможны (например, $<$*нет*, *обр*, $\beta'>$), эта классификация предлагает к рассмотрению и изучению огромное количество возможных алгоритмов. На этой основе вы можете попытаться "сконструировать" собственный алгоритм. Успехов вам!

Упражнения 8.7

1. На основе приведенной классификации разработайте собственный алгоритм и определите его эффективность в среднем и в самом худшем случае.
2. Покажите, что алгоритм КМПХ, описанный в предыдущем разделе, не охватывается приведенной классификацией. Как можно включить его в эту классификацию?

8.8 Точные оценки сложности алгоритмов точного сравнения с паттерном

В последних двух главах мы изучали, в основном с практической точки зрения, алгоритмы вычисления всех вхождений заданного паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ в заданную текстовую строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Основываясь на результатах экспериментальных исследований, было показано, что все варианты алгоритма Бойера–Мура в среднем ведут себя "почти линейно" — среднее количество буквенных сравнений, используемых этими алгоритмами, составляет cn, где c, как правило, равно $0{,}3$ или меньше, а в алгоритме БМХ равно $1/\alpha$, где α — размер алфавита. Более того, мы видели, что по крайней мере один из вариантов алгоритма БМ (а именно, алгоритм Турбо-БМ) требует не более $2n$ буквенных сравнений в самом худшем случае, тогда как другой алгоритм (алгоритм БМГАГ, [65]) — не более $1{,}5n$ буквенных сравнений. Но в то же время вариант алгоритма КМП (алгоритм КМПХ) требует только $(3n - m)/2$ буквенных сравнений. Таким образом, одинаковую временну́ю сложность в самом худшем случае, по крайней мере измеренную как количество буквенных сравнений, имеют как варианты алгоритма БМ, так и варианты алгоритма КМП. Из этих наблюдений естественным образом возникают следующие фундаментальные теоретические вопросы.

1. Какое минимальное число буквенных сравнений требуется в самом худшем случае алгоритмам сравнения с паттерном?
2. Существуют ли алгоритмы, на которых достигается эта минимальная граница?
3. Какое минимальное число буквенных сравнений требуется *в среднем* алгоритмам сравнения с паттерном?

Первоначально ответ на первый вопрос был дан в тот же год, когда были опубликованы алгоритмы КМП и БМ: как показано в работе [196], для любого фиксированного паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и любого алгоритма, который корректно определяет, является ли паттерн $\boldsymbol{p}$ подстрокой произвольных текстовых строк $\boldsymbol{x} = \boldsymbol{x}[1..n]$, существует такая строка $\boldsymbol{x}^*$, в которой необходимо проверить не менее $n - m + 1$ позиций. Однако не существует алгоритма, который гарантировал бы, что ему требуется просмотреть *не более* $n - m + 1$ позиций. Поэтому остается открытым вопрос, насколько точна эта нижняя граница. Далее рассмотрим вопрос о нижней границе временно́й сложности алгоритмов, которые только определяют, входит ли заданный паттерн $\boldsymbol{p}$ в строку $\boldsymbol{x}$, но не обязательно находят все вхождения паттерна в строку.

В последнюю четверть минувшего столетия и позднее основные усилия исследователей были направлены на поиск ответов на вопросы 1 и 2, сформулированные выше. Значительный вклад в этот поиск был сделан в работах [54, 55, 96, 97, 52, 56]. Вариантами ответов на эти вопросы является поиск оптимального соотношения между временно́й и пространственной сложностью алгоритмов [99, 66]. В частности, в статье [52] представлен алгоритм (назовем его алгоритмом CH), который вычисляет все вхождения паттерна $\boldsymbol{p}$ в строку $\boldsymbol{x}$ с проверкой не более

$$\left(1 + \frac{8}{3m+3}\right) n \tag{8.11}$$

букв строки $\boldsymbol{x}$. Более того, этот алгоритм является ***онлайновым***, хотя и в особом понимании этого термина, — буквы текстовой строки могут появляться только в окне длиной m, которое монотонно перемещается слева направо без возврата назад. Если эффективность алгоритмов измерять как количество используемых буквенных сравнений, то алгоритм CH на сегодняшний день наиболее эффективен. Он имеет только один недостаток: необходимость этапа предварительных вычислений с временем выполнения порядка $O(m^2)$.

Позднее в работе [50] в качестве более точного ответа на первый вопрос было показано, что временна́я сложность алгоритма CH очень близка к теоретической нижней границе числа буквенных сравнений. Для онлайновых алгоритмов, в которых применяются два вида сравнения (паттерн-текст и текст-текст), нижняя граница составляет

$$\left(1 + \frac{9}{4m+4}\right) n, \tag{8.12}$$

причем эта оценка справедлива только для тех паттернов, длина которых удовлетворяет условию $m = 36k + 35$, $k \geqslant 0$. Для онлайновых алгоритмов, использующих только сравнения типа паттерн-текст, доказана оценка нижней границы, слегка превышающая приведенную. Таким образом, все онлайновые алгоритмы

должны выполнять количество буквенных сравнений, не меньшее, чем указано в (8.12).

Для общих алгоритмов, использующих сравнения как типа паттерн-текст, так и типа текст-текст, доказана оценка нижней границы в самом худшем случае

$$\left(1+\frac{2}{m+1}\right)n, \tag{8.13}$$

которая справедлива для всех паттернов длиной $m = 2k+1$, $k \geqslant 2$.

На основе приведенных оценок делаем вывод, что асимптотически (т.е. для достаточно больших m и n) так называемая "точная оценка сложности алгоритмов сравнения с паттерном", по крайней мере в терминах буквенных сравнений в самом худшем случае, установлена практически точно. Из формул (8.11)–(8.13) следует, что наилучший алгоритм сравнения с паттерном должен выполнять

$$(1+d/m)\times n \tag{8.14}$$

буквенных сравнений в самом худшем случае, где для онлайновых алгоритмов $9/4 \leqslant d \leqslant 8/3$ и для произвольных алгоритмов $2 \leqslant d \leqslant 8/3$.

Теперь вернемся к "забытому" третьему вопросу. Поскольку анализ поведения алгоритмов в среднем кажется более сложным, чем анализ поведения в самом худшем случае, может показаться неожиданным, что в настоящее время известен точный ответ на этот вопрос. Сначала в работе [137] был описан простой алгоритм сравнения с паттерном, который в среднем требовал

$$O\left(\frac{\log_\alpha m}{m}n\right) \tag{8.15}$$

буквенных сравнений, где $\alpha = |A|$ — размер алфавита и среднее берется по всем возможным строкам, построенным на алфавите A. Два года спустя в статье [234] было показано, что для достаточно больших m и $n > 2m$ верхняя граница (8.15) также является нижней границей минимального количества буквенных сравнений, которые используются в среднем любым алгоритмом сравнения с паттерном. В [152] описан вариант алгоритма БМ, на котором достигается эта граница.

Сравнивая оценку (8.15) с оценкой (8.14), нетрудно заметить, что

$$\frac{\log_\alpha m}{m} < \frac{m+d}{m}.$$

Отсюда вытекает, что поведение в среднем наилучшего возможного (в среднем) алгоритма асимптотически лучше (примерно на величину $\log_\alpha m/m$) поведения в самом худшем случае наилучшего возможного (в самом худшем случае) алгоритма. Впрочем, этот результат был интуитивно ожидаем.

Упражнения 8.8

1. Объясните, почему оценки (8.12) и (8.13) справедливы только для определенных значений m и почему эти оценки нельзя использовать как нижние границы для всех паттернов.

2. Выведите оценку (8.14) из оценок (8.11)–(8.13).

ГЛАВА 9

Алгоритмы вычисления расстояния между строками

Человек думает, что чем тверже он произносит слова, тем тверже его позиция.

— Герман Мелвилл (1819–1891)

В двух предыдущих главах мы рассмотрели алгоритмы точного сравнения с паттернами. Теперь приступим к изучению алгоритмов *приближенного* сравнения заданного паттерна $\boldsymbol{p}$ с заданной строкой $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Однако к этой теме мы подойдем непрямым путем. Чтобы найти приближенное совпадение паттерна $\boldsymbol{p}$ с подстрокой $\boldsymbol{u}$ строки $\boldsymbol{x}$, сначала надо определиться с понятием "приближенное совпадение". Как показано в разделе 2.2, существует, по крайней мере, три подхода к определению этого понятия.

- Приближенность совпадения между паттерном $\boldsymbol{p}$ и подстрокой $\boldsymbol{u}$ измеряется с помощью "расстояния" $d(\boldsymbol{p}, \boldsymbol{u})$. В разделе 2.2 определено несколько разновидностей такого расстояния: расстояние Хемминга $d = d_H$, расстояние преобразования $d = d_E$, расстояние Левенштейна $d = d_L$ и взвешенное расстояние $d = d_W$. Конечно, возможны и другие типы расстояния (некоторые из них упоминались в разделе 2.2).
- Паттерн может содержать метасимволы, которые представляют не некоторые конкретные буквы, а определенные классы букв. Таким образом паттерн сам становится приближенным. В разделе 2.2 описаны такие метасимволы,

как символ подстановки •, представляющий произвольную одиночную букву, и символ подстановки $*$, представляющий произвольную строку букв.

- Еще более "приближенными" будут паттерны, являющиеся регулярными выражениями, которые содержат метасимволы | и *. Эти метасимволы позволяют проводить сравнение с альтернативными подстроками и кратными подстроками соответственно (см. раздел 2.2).

Исходя из первого подхода к определению "приближенности", для нахождений совпадений необходимо уметь вычислять расстояния $d(\boldsymbol{p}, \boldsymbol{u})$ для произвольной подстроки $\boldsymbol{u}$ строки $\boldsymbol{x}$. В этом случае задача вычисления расстояний между строками становится естественной подзадачей общей проблемы приближенного сравнения с паттерном.

Поэтому в данной главе в качестве прелюдии к алгоритмам приближенного сравнения с паттерном изучим алгоритмы вычисления расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ между заданными строками $\boldsymbol{x}_1 = \boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2 = \boldsymbol{x}_2[1..n_2]$. Если $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ являются последовательностями ДНК или РНК, тогда, как мы увидим далее, вычислив расстояние $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$, можно также определить то, что называется ***оптимальным выравниванием*** (optimal alignment) двух строк. Мы также обнаружим, что некоторые описываемые в этой главе алгоритмы выполняют двойные роли: они вычисляют расстояния между строками и *одновременно* выполняют приближенное сравнение с паттерном.

Как показано в работе [120], для расстояния, определяемого произвольной матрицей весов W, нижняя граница временно́й сложности задачи вычисления расстояний между строками равна $\Omega(n \log n)$. Эта же граница применима к алгоритмам приближенного сравнения с паттерном (раздел 13.3). В данной и следующей главах приведем несколько алгоритмов, на которых достигается эта нижняя граница, однако при этом будут использованы некоторые специальные свойства расстояний Хемминга и преобразования.

Далее в этой главе детально рассмотрим четыре алгоритма, два из которых универсальны и непосредственно вычисляют расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ для произвольных расстояний d. Два других алгоритма вычисляют (не обязательно единственную) наибольшую общую подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$, на основании которой, как показано в разделе 2.2, можно вычислить расстояние Левенштейна. Все эти алгоритмы существенно используют методы динамического программирования, что открывает широкий простор для различных теоретических и практических исследований.

В следующей главе мы покажем, как универсальные алгоритмы вычисления расстояний, описанные ниже, можно преобразовать в алгоритмы приближенного сравнения с паттерном.

9.1 Базисная рекурсия

Как предложено показать в упражнении 9.1.1, существует алгоритм прямого вычисления расстояния Хемминга $d_{\mathrm{H}}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ между двумя строками $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ одинаковой длины n за время порядка $\Theta(n)$. Поэтому в данном разделе исследуем только расстояния Левенштейна, преобразования и взвешенное. Поскольку алгоритмы, описываемые далее, применимы ко всем этим трем типам расстояний, то естественно обозначать расстояние просто как d, предполагая при этом, что это расстояние удовлетворяет условиям метрики (2.2)–(2.5). Из этих условий можно получить некоторые свойства операций редактирования, выполняемых на одиночных (не пустых) буквах λ и μ.

- Вставка (замена пустой строки ε на букву λ): $d(\varepsilon, \lambda) > 0$.
- Удаление (замена буквы λ на пустую строку ε): $d(\lambda, \varepsilon) > 0$.
- Подстановка (замена буквы λ на букву μ): $d(\lambda, \mu) > 0$ тогда и только тогда, когда $\lambda \neq \mu$.

Также напомним, что если $d = d_L$ или d_E, то $d(\varepsilon, \lambda) = d(\lambda, \varepsilon) = 1$ для любой буквы λ; тогда как для $d = d_L d(\lambda, \mu) = 2$ и для $d = d_E$ $d(\lambda, \mu) = 1$.

Отметим также, что расстояние между отдельными буквами можно рассматривать как ***стоимость*** выполнения соответствующих операций редактирования. В общем случае расстояние между двумя строками является минимальной стоимостью преобразования одной строки в другую.

Все алгоритмы, представленные в этой главе, являются алгоритмами ***динамического программирования***. Это означает, что они сравнивают входные строки $\boldsymbol{x}_1 = \boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2 = \boldsymbol{x}_2[1..n_2]$ слева направо, вычисляя для каждой пары позиций i и j минимальную стоимость преобразования подстроки $\boldsymbol{x}_1[1..j]$ в подстроку $\boldsymbol{x}_2[1..j]$, основываясь при этом на уже вычисленных стоимостях (расстояниях) $d(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j-1])$, $d(\boldsymbol{x}_1[1..i-1], \boldsymbol{x}_2[1..j])$ и $d(\boldsymbol{x}_1[1..j-1], \boldsymbol{x}_2[1..j-1])$. Эта базисная рекурсия объясняется ниже.

Для дальнейших вычислений удобно ввести двухмерный ***массив стоимостей*** $\boldsymbol{c} = \boldsymbol{c}[0..n_1, 0..n_2]$, в котором для всех $i \in 0..n_1$ и $j \in 0..n_2$ $\boldsymbol{c}[i, j] = d(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j])$. Начальные значения для этого массива определяются следующим образом.

- $\boldsymbol{c}[0, 0] = 0$ — это минимальная стоимость преобразования пустой строки в пустую строку.
- $\boldsymbol{c}[0, j] = \sum_{1 \leqslant h \leqslant j} d(\varepsilon, \boldsymbol{x}_2[h])$ — минимальная стоимость вставки первых j букв строки $\boldsymbol{x}_2$ в пустую строку.
- $\boldsymbol{c}[i, 0] = \sum_{1 \leqslant h \leqslant i} d(\boldsymbol{x}_1[h], \varepsilon)$ — минимальная стоимость удаления первых i букв строки $\boldsymbol{x}_1$ таким образом, чтобы сформировать пустую строку.

Для большего понимания массива стоимостей рассмотрим пример массива $\boldsymbol{c}[0..5, 0..6]$, соответствующего вычислению расстояния Левенштейна $d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)$,

где $\boldsymbol{x}_1 = rests$, $\boldsymbol{x}_2 = stress$.[1] Как видно из табл. 9.1, искомое расстояние равно

$$\boldsymbol{c}[5,6] = d_L(\boldsymbol{x}_1[1..5], \boldsymbol{x}_2[1..6]) = 3.$$

Таблица 9.1. Массив стоимостей для $d_L(rests, stress)$

	$\boldsymbol{j}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**
i		ε	s	t	r	e	s	s
0	ε	0	1	2	3	4	5	6
1	r	1	2	3	2	3	4	5
2	e	2	3	4	3	2	3	4
3	s	3	2	3	4	3	2	3
4	t	4	3	2	3	4	3	4
5	s	5	4	3	4	5	4	3

Отметим, что в этом примере минимальная стоимость преобразования $rest \to$ $\to stre$, т.е. $\boldsymbol{c}[4,4]$, равна 4. Эта стоимость получается путем прибавления стоимости одного удаления (буквы t из $rest$) к минимальной стоимости $\boldsymbol{c}[3,4] = 3$ преобразования $res \to stre$. Иначе стоимость $\boldsymbol{c}[4,4]$ можно получить путем прибавления стоимости одной вставки (буквы e в $stre$) к минимальной стоимости $\boldsymbol{c}[4,3] = 3$ преобразования $rest \to str$. Таким образом,

$$\boldsymbol{c}[4,4] = \boldsymbol{c}[3,4] + 1 = \boldsymbol{c}[4,3] + 1.$$

Мы также видим, что стоимость $\boldsymbol{c}[5,6] = 3$ полного преобразования $rests \to$ $\to stress$ можно получить из минимальной стоимости $\boldsymbol{c}[4,5] = 3$ преобразования $rest \to stres$, поскольку стоимость преобразования конечных букв $s \to s$ равна нулю. Таким образом, $\boldsymbol{c}[5,6] = \boldsymbol{c}[4,5] = 0$.

Нетрудно заметить, что фактически минимальную стоимость можно получить тремя различными путями. Например, для преобразования $rest \to stress$ имеем

$$\boldsymbol{c}[4,6] = \boldsymbol{c}[3,6] + 1 = \boldsymbol{c}[4,5] + 1 = \boldsymbol{c}[3,5] + 2.$$

Последние равенства являются частным случаем отношений базисной рекурсии, которая позволяет вычислить любой элемент массива стоимостей $\boldsymbol{c}$ на основе начальных значений $\boldsymbol{c}[0,0]$, $\boldsymbol{c}[0,j]$, $j = 1, 2, \ldots, n_2$ и $\boldsymbol{c}[i,0]$, $i = 1, 2, \ldots, n_1$. Имеет место следующая лемма.

[1] Здесь шутка автора: $rest$ — покой, сон, $stress$ — стресс, напряжение. — *Примеч. пер.*

Лемма 9.1.1. Для всех $i \in 1..n_1$, $j \in 1..n_2$

$$\begin{aligned}\boldsymbol{c}[i,j] = \min\{&\boldsymbol{c}[i-1,j] + d(\boldsymbol{x}_1[i], \varepsilon),\\ &\boldsymbol{c}[i,j-1] + d(\varepsilon, \boldsymbol{x}_2[j]),\\ &\boldsymbol{c}[i-1,j-1] + d(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j])\}.\end{aligned}$$

Доказательство. Основой доказательства служит факт, установленный в упр. 2.2.13: во время преобразования строк операции вставки и удаления можно выполнять в любом порядке. Этот факт порождает три возможности выполнения операций вставки и удаления при манипулировании буквами $\boldsymbol{x}_1[i]$ и $\boldsymbol{x}_2[j]$.

- Первая возможность — удаляется буква $\boldsymbol{x}_1[i]$, стоимость операции равна $d(\boldsymbol{x}_1[i], \varepsilon)$, которая прибавляется к стоимости $\boldsymbol{c}[i-1, j]$.
- Вторая возможность — вставляется буква $\boldsymbol{x}_2[j]$, стоимость операции равна $d(\varepsilon, \boldsymbol{x}_2[j])$, которая прибавляется к стоимости $\boldsymbol{c}[i, j-1]$.
- Третья возможность — подстановка, т.е. замена буквы $\boldsymbol{x}_1[i]$ буквой $\boldsymbol{x}_2[j]$ (буква $\boldsymbol{x}_1[i]$ удаляется, буква $\boldsymbol{x}_2[j]$ вставляется). Стоимость этой операции $d(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j])$ прибавляется к стоимости $\boldsymbol{c}[i-1, j-1]$. Конечно, если $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$, то $d(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j]) = 0$.

Поскольку не существует других возможных манипуляций буквами $\boldsymbol{x}_1[i]$ и $\boldsymbol{x}_2[j]$ (с учетом "несимметричности" этих букв), то минимальная стоимость $\boldsymbol{c}[i, j]$ должна быть минимумом этих трех перечисленных возможностей. ■

Повторим, что основой доказательства этой леммы служит факт, что операции вставки и удаления можно выполнять в любом порядке. Прилежный читатель, который выполнил упражнение 2.2.13, мог заметить, что это свойство операций вставки и удаления *не зависит* от условия (2.4) симметричности метрики. Оно имеет место даже в том случае, когда для некоторых строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ $d(\boldsymbol{x}_1, \boldsymbol{x}_2) \neq$ $\neq d(\boldsymbol{x}_2, \boldsymbol{x}_1)$. Таким образом, лемма 9.1.1 справедлива даже тогда, когда не выполняется условие симметричности. Это замечание подводит нас к мысли, что стоимость преобразований по своей природе не является полным аналогом расстояния. Поскольку мы определили стоимость в терминах преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$, то возможны случаи, когда стоимость "обратного" преобразования строки $\boldsymbol{x}_2$ в строку $\boldsymbol{x}_1$ не будет совпадать со стоимостью "прямого" преобразования. Такая ситуация возникает, например, в приложениях молекулярной биологии, где матрица весов W в некоторых случаях может быть несимметричной, что часто порождает неравенства $d_W(\boldsymbol{x}_1, \boldsymbol{x}_2) \neq d_W(\boldsymbol{x}_2, \boldsymbol{x}_1)$.

Наконец, отметим, что массив стоимостей и схема рекурсии для его вычисления, данная в лемме 9.1.1, очень просты и естественны. Поэтому они имеют большое применение не только для вычисления расстояний между строками (а значит, и LCS), но и, как основа почти всех алгоритмов, для вычисления наименьшей

общей сверхстроки для множества строк и для нахождения "наилучшего" выравнивания набора строк. Эти NP-полные задачи весьма важны для вычислительной биологии.

Упражнения 9.1

1. Приведите алгоритм вычисления расстояния Хемминга между двумя строками длиной n, выполняемый за время порядка $\Theta(n)$.
2. Докажите формулы для вычисления начальных значений $\boldsymbol{c}[0,0]$, $\boldsymbol{c}[0,j]$ и $\boldsymbol{c}[i,0]$ массива стоимостей.

9.2 Алгоритм Вагнера–Фишера

Алгоритм Вагнера–Фишера (Wagner–Fischer) вычисляет массив стоимостей и, следовательно, расстояние между двумя заданными строками $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$. Он, конечно, основывается на лемме 9.1.1. Его временна́я и пространственная сложности имеют порядок $\Theta(n_1 n_2)$ (доказательство этого утверждения оставим для упр. 9.2.1). Отметим, что этот алгоритм в определенном смысле онлайновый (раздел 4.1), поскольку в процессе вычисления расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ также вычисляются расстояния между каждой парой префиксов строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. В разделе 9.3 покажем простой прием, который позволит в алгоритме Вагнера–Фишера уменьшить требуемый объем памяти до порядка $\Theta(\min\{n_1, n_2\})$.

Как указывается в работе [202], алгоритм вычисления расстояния между строками, основанный на массиве стоимостей, неоднократно независимо переоткрывался при проведении исследований в таких областях, как распознавание речи, автоматическая проверка орфографии, молекулярная биология и др. [228, 192, 227, 201, 203, 195, 113, 229]. Среди всех этих вариантов одного, по сути, алгоритма мы выбрали работу Вагнера и Фишера [229] для представления здесь данного алгоритма вследствие того, что алгоритм Вагнера–Фишера, кроме основного назначения, выполняет также "трассировку", позволяя точно отображать последовательность выполняемых операторов редактирования, а также $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

Предположим, что для заданных строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$ уже вычислен массив стоимостей $\boldsymbol{c}[0..n_1, 0..n_2]$. Для произвольной позиции $q = [i,j]$, $1 \leqslant i \leqslant n_1$, $1 \leqslant j \leqslant n_2$, элемента массива $\boldsymbol{c}$ определим *перемещение* $v : q \to q'$ как функцию отображения позиций q в предыдущую позицию q' согласно следующим правилам.

- Если $\boldsymbol{c}[i,j] = \boldsymbol{c}[i-1,j] + d(\boldsymbol{x}_1[i], \varepsilon)$, то $q' = [i-1, j]$;
- иначе, если $\boldsymbol{c}[i,j] = \boldsymbol{c}[i,j-1] + d(\varepsilon, \boldsymbol{x}_2[j])$, то $q' = [i, j-1]$;
- в противном случае $q' = [i-1, j-1]$.

Таким образом, перемещение определяет одну из трех возможных позиций в массиве $\boldsymbol{c}$, исходя из которой с помощью одной операции редактирования получено текущее значение $\boldsymbol{c}[i,j]$, при этом операторы редактирования выбираются согласно порядку

$$<\textit{удаление}, \textit{вставка}, \textit{подстановка}>. \tag{9.1}$$

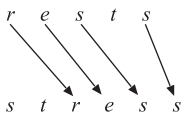
Поскольку функция v уменьшает на единицу по крайней мере одно из значений i или j, то существует такое целое число $k \in 1..i+j$, что суперпозиция $v^k([i,j])$ приведет к позиции $[i',j']$ (т.е. $v^k([i,j]) = [i',j']$), где $\min\{i',j'\} = 0$. В примере из предыдущего раздела (см. табл. 9.1) мы можем отследить последовательность

$$v([5,6]) = [4,5]; v([4,5]) = [3,5]; v([3,5]) = [2,4]; v([2,4]) = [1,3]; v([1,3]) = [0,2].$$

В этом случае $k = 5$ и $v^5([5,6]) = [0,2]$. Теперь, если для каждой операции перестановки (но не операций удаления и вставки) из этой последовательности перемещений записать ее текущее значение $[i,j]$, то в результате получим так называемый *след* (trace), который обычно обозначается буквой τ. В нашем примере получаем след

$$\tau([5,6]) = \{[5,6],[3,5],[2,4],[1,3]\},$$

который соответствует подстановкам



В этом примере каждая "подстановка" просто замещает одну букву на саму себя. Вследствие этого стоимость таких подстановок равна нулю. Но, как показано в упр. 9.2.3, это не является обязательным свойством следа — след может содержать позиции, на которых выполняются подстановки с ненулевой стоимостью.

В общем случае след $\tau([n_1,n_2])$ определяет все подстановки (включая все подстановки нулевой стоимости), которые содержатся в последовательности операторов редактирования минимальной стоимости, необходимых для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$. Отметим, что если позиция i, соответствующая букве $\boldsymbol{x}_1[i]$, *не входит* в след τ, то это означает, что данная буква удаляется в процессе преобразования $\boldsymbol{x}_1 \to \boldsymbol{x}_2$. Но если в следе τ не встречается позиция j, соответствующая букве $\boldsymbol{x}_2[j]$, то тогда эта буква вставляется в процессе преобразования. Таким образом след можно использовать для прямого и полного определения всей последовательности операторов редактирования минимальной стоимости, выполняемых в процессе преобразования $\boldsymbol{x}_1 \to \boldsymbol{x}_2$. В нашем примере отсутствующим в следе позициям соответствуют буква $\boldsymbol{x}_1[4] = t$, которая удаляется, и подстрока $\boldsymbol{x}_2[1..2] = st$, которая вставляется. Эту ситуацию, если использовать символ

пробела Δ, наглядно можно представить в виде следующей таблицы.

$$\begin{array}{cccccccc} \Delta & \Delta & r & e & s & t & s \\ s & t & r & e & s & \Delta & s \end{array}$$

Это представление показывает оптимальное (минимальной стоимости) выравнивание строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. (Как указывалось ранее, задача нахождения наилучшего выравнивания строковых последовательностей весьма важна для вычислительной биологии.) Мы оставим для упр. 9.2.6 детальное описание алгоритмов, находящих последовательность операторов редактирования и вычисляющих след.

Далее отметим, что в следе подстановки нулевой стоимости определяют общую подпоследовательность строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$, хотя, как показано в упр. 9.2.3, эта общая подпоследовательность не обязана быть *наибольшей* общей подпоследовательностью $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Но если мы ограничимся только расстоянием Левенштейна $d_L = d$, для которого $d(\lambda, \mu) = 2$ на любой паре несовпадающих букв λ и μ, то нетрудно видеть, что для любых строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$ выполняется равенство

$$\boldsymbol{c}[n_1, n_2] = n_1 + n_2 - 2|\,\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)|. \tag{9.2}$$

Отсюда следует, что при условии $d = d_L$ длину $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ можно вычислить по формуле

$$|\,\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)| = (n_1 + n_2 - \boldsymbol{c}[n_1, n_2])/2. \tag{9.3}$$

Как показано в упр. 9.2.4, саму подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ несложно вычислить из соответствующего следа, если $d = d_L$.

В работе [229] приведено еще одно интересное применение массива стоимостей, а именно для проверки орфографии текстов. Здесь используется расстояние $d = d_W$ с матрицей весов W, в которой отображен тот факт, что при печати на стандартной клавиатуре ошибочные буквы (опечатки) чаще всего замещают наиболее вероятные буквы (т.е. те, которые наиболее часто встречаются в текстах на данном языке), при этом ошибочные буквы, как правило, на клавиатуре располагаются рядом с "правильными" буквами. Подобная ситуация имеет место в матрицах весов для последовательностей ДНК. Здесь наиболее часто встречается пара нуклеотидов C и G, которая может замещать более редкие пары C и T или G и A. В обоих этих приложениях становится важной операция подстановки специального типа: перестановка соседних символов. Как показано в [229], если перестановка включена в набор основных операторов редактирования, то расстояние между строками $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ можно вычислить за время порядка $\Theta(n_1 n_2)$, предполагая при этом, что стоимость перестановки несовпадающих букв λ и μ не меньше, чем $(d(\varepsilon, \lambda) + d(\mu, \varepsilon))/2$.

Упражнения 9.2

1. Приведите алгоритм вычисления массива стоимостей для двух заданных строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$, выполняемый за время порядка $\Theta(n_1 n_2)$.
2. Докажите, что если для расстояния d выполняется условие симметричности, то тогда массив стоимостей для преобразования строки $\boldsymbol{x}_2$ в строку $\boldsymbol{x}_1$ совпадает с транспонированным массивом стоимостей для преобразования строки $\boldsymbol{x}_1$ в строку $\boldsymbol{x}_2$. Проверьте свое доказательство путем вычисления массива $\boldsymbol{c}$ для строк $\boldsymbol{x}_1 =$ *stress* и $\boldsymbol{x}_2 =$ *rests*.
3. Для расстояния преобразования $d = d_E$ вычислите массив стоимостей и след, которые соответствуют строкам $\boldsymbol{x}_1 =$ *sowsear* и $\boldsymbol{x}_2 =$ *silkpurse*. Покажите, что в данном примере не все позиции в следе соответствуют подстановкам нулевой стоимости. Также покажите, что общая подпоследовательность, определяемая на основе следа, не является подпоследовательностью $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.
4. Для расстояния Левенштейна $d = d_L$ вычислите массив стоимостей и след, которые соответствуют строкам $\boldsymbol{x}_1 =$ *sowsear* и $\boldsymbol{x}_2 =$ *silkpurse*. Покажите, что в данном примере общая подпоследовательность, определяемая на основе следа, является подпоследовательностью $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.
5. Для чего необходимо было устанавливать порядок выполнения операторов редактирования (9.1)? Влияет ли этот порядок на правильность вычислений следа и соответствующей последовательности выполняемых операторов редактирования?
6. Предложите алгоритм, который вычислял бы след на основе ранее вычисленного массива стоимостей $\boldsymbol{c}[0..n_1, 0..n_2]$ за время порядка $O(n_1 + n_2)$. Предложите также алгоритм с временем выполнения $\Theta(n_1 + n_2)$ для вычисления последовательности операторов редактирования.
7. Докажите равенство (9.2).
8. Предложите алгоритм, который вычислял бы $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ на основании ранее вычисленного следа для расстояния $d = d_L$ за время порядка $O(\min\{n_1, n_2\})$.
9. Разработайте матрицу весов W для алфавита английского языка (только для строчных букв) таким образом, чтобы учесть возрастающую вероятность замены одной буквы на другую, если этим буквам соответствуют рядом расположенные клавиши на стандартной клавиатуре пишущей машинки. Например, можно положить

$$W[d, e] = W[d, r] = W[d, s] = W[d, f] = W[d, x] = W[d, c] = 1/2,$$

тогда как $W[d, \lambda] = 1$ для всех других букв λ. Затем вычислите массив стоимостей для $\boldsymbol{x}_1 =$ *sowsear* и $\boldsymbol{x}_2 =$ *silkpurse* при условии, что $d = d_W$.

9.3 Алгоритмы Хешберга

Хешберг (Hirschberg) [117] предложил алгоритм непосредственного вычисления $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ за время порядка $\Theta(n_1 n_2)$ без явного вычисления массива стоимостей $\boldsymbol{c}$. Он также показал, что этот алгоритм требует память объемом только $\Theta(\min\{n_1, n_2\})$.

Для описания этого алгоритма сначала определим массив $\boldsymbol{\gamma}[0..n_1, 0..n_2]$ (аналог массива стоимостей), в котором для всех $i \in 0..n_1$ и $j \in 0..n_2$

$$\boldsymbol{\gamma}[i,j] = |\,\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j])|.$$

Таким образом, $\boldsymbol{\gamma}[i,j]$ равняется длине наибольшей общей подпоследовательности подстрок $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..j]$. Поскольку пустая строка ε имеет LCS нулевой длины с любой другой строкой, то очевидно, что $\boldsymbol{\gamma}[i,0] = \boldsymbol{\gamma}[0,j] = 0$. Другие элементы массива $\boldsymbol{\gamma}$ вычисляются на основе следующей леммы (аналог леммы 9.1.1).

Лемма 9.3.1. Для расстояния Левенштейна $d = d_L$ и для всех $i \in 1..n_1$ и $j \in 1..n_2$

- $\boldsymbol{\gamma}[i,j] = \boldsymbol{\gamma}[i-1, j-1] + 1$, если $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$,
- $\boldsymbol{\gamma}[i,j] = \max\{\boldsymbol{\gamma}[i-1,j], \boldsymbol{\gamma}[i,j-1]\}$ в противном случае.

Доказательство. Для доказательства достаточно заметить, что для расстояния $d = d_L$ выполняются равенства (9.2) и (9.3), если в них n_1 и n_2 заменить соответственно на $i \in 1..n_1$ и $j \in 1..n_2$, а затем следует применить лемму 9.1.1 для случая $d = d_L$. ■

Алгоритм 9.3.1 (Алгоритм H1)

```
▷ Вычисление длины LCS(x1, x2)
▷ Элементам первой строки и первого столбца
▷ массива γ присваиваются нулевые значения
for i ← 0 to n1 do
    γ[i, 0] ← 0
for j ← 0 to n2 do
    γ[0, j] ← 0
▷ Вычисление массива γ
for i ← 1 to n1 do
    for j ← 1 to n2 do
        if x1[i] = x2[j] then
            γ[i, j] ← γ[i − 1, j − 1] + 1
        else
            γ[i, j] ← max{γ[i − 1, j], γ[i, j − 1]}
```

Алгоритм H1 является простой другой формой записи леммы 9.3.1. Мы привели этот тривиальный алгоритм не для того, чтобы бросить вызов бесспорному

интеллекту читателя, а для того, чтобы позднее можно было сравнить его с другой улучшенной версией этого алгоритма, также основанной на лемме 9.3.1.

Хешберг заметил, что в алгоритме H1 для определения элементов строки i массива γ необходимо знать только элементы строки $i-1$. Поэтому в любой момент выполнения алгоритма достаточно хранить в памяти только две строки этого массива. Для этого можно задать массив $\gamma'[0..1, 0..n_2]$ размерностью $2 \times n_2$, где хранились бы текущая (i-я) и предыдущая ($(i-1)$-я) строки массива γ. Введем ***триггерную*** (переключающую) ***переменную*** $i' \in 0..1$, с помощью которой осуществлялся бы доступ к элементам $\gamma'[i', j]$ текущей строки, тогда как значение $1 - i'$ определяло бы элементы $\gamma'[1 - i', j]$ предыдущей. В результате этих нововведений получим новый алгоритм 9.3.2, который возвращает последнюю строку $\gamma[n_1, 0..n_2]$ массива γ. Этот алгоритм запишем как функцию H2.

Алгоритм 9.3.2 (Алгоритм H2)

```
▷ Вычисление длины LCS(x1, x2)
function H2(n1, x1, n2, x2)
    ▷ Присвоение значения 0 элементу γ'[1,0] и элементам строки γ'[0..1, 0..n2]
    γ'[1,0] ← 0
    for j ← 0 to n2 do
        γ'[0,j] ← 0
    ▷ Инициализация триггерной переменной
    i' ← 0
    ▷ Вычисление γ'[1 − i', j]
    for i ← 1 to n1 do
        i' ← 1 − i'
        for j ← 1 to n2 do
            if x1[i] = x2[j] then
                γ'[i,j] ← γ'[1 − i', j − 1] + 1
            else
                γ'[i,j] ← max{γ'[1 − i', j], γ'[i', j − 1]}
    ▷ γ'[i,j] = |LCS(x1, x2[1..j])|, j = 0, 1, ..., n2
    return γ'[i', 0..n2]
```

Отметим, что использование триггерной переменной делает алгоритм H2 значительно более эффективным (по сравнению с алгоритмом H1), поскольку позволяет уменьшить требуемый объем памяти до величины порядка $\Theta(n_2)$. Однако тот же результат можно получить, если подобные вычисления выполняются не относительно строк, а относительно столбцов массива γ. Поэтому можно говорить о том, что в данном алгоритме требуемый объем памяти составляет величину порядка $\Theta(\min\{n_1, n_2\})$. В упр. 9.3.3 предложено применить подобный подход

к алгоритму Вагнера–Фишера для того, чтобы в этом алгоритме уменьшить требуемый объем памяти до величины такого же порядка.

До сих пор все было хорошо. Однако напомним, мы пока можем вычислить только *длину* наибольшей общей подпоследовательности $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$, а не саму эту подпоследовательность. Для решения этой задачи можно предложить алгоритм, где рекурсивно вызывалась бы функция H2. Для описания такого алгоритма необходимо ввести еще несколько понятий. Обозначим через $\boldsymbol{\Gamma}_{ij}$ подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j])$, а через $\boldsymbol{\Gamma}^*_{ij} - \mathrm{LCS}(\boldsymbol{x}_1[i+1..n_1], \boldsymbol{x}_2[j+1..n_2])$. Тогда $\boldsymbol{\gamma}[i,j] = |\boldsymbol{\Gamma}_{ij}|$ и $\boldsymbol{\gamma}^*[i,j] = |\boldsymbol{\Gamma}^*_{ij}|$. Кроме того, обозначим

$$\gamma_i^{\max} = \max_{0 \leqslant j \leqslant n_2} \{\boldsymbol{\gamma}[i,j] + \boldsymbol{\gamma}^*[i,j]\}.$$

Далее мы докажем, что для всех $i \in 0..n_1$ $\gamma_i^{\max} = \gamma[n_1, n_2]$. Отсюда следует, что наибольшую общую последовательность строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ можно найти путем разделения строки $\boldsymbol{x}_1$ на префикс $\boldsymbol{x}_1[1..i]$ и суффикс $\boldsymbol{x}_1[i+1..n_1]$ (позиция i выбирается произвольно) и последующего вычисления функций $\mathrm{H2}(i, \boldsymbol{x}_1[1..i], n_2, \boldsymbol{x}_2)$ и $\mathrm{H2}(n_1 - i, \boldsymbol{x}_1[i+1..n_1], n_2, \boldsymbol{x}_2)$, что позволит найти $\boldsymbol{\gamma}[i, n_2]$ и $\boldsymbol{\gamma}^*[i, 0]$.

Эту схему вычислений можно применить рекурсивно. Тогда, если на каждом шаге рекурсии выбирать $i = \lfloor n_1/2 \rfloor$, после $\log_2 n_1$ шагов вычисление $\boldsymbol{\gamma}[n_1, n_2]$ можно свести к суммированию нулей и единиц. При этом для каждой полученной единицы можно проверить, найдена ли буква, входящая в LCS. Более четко и детально этот алгоритм мы опишем ниже, а пока докажем следующую лемму.

Лемма 9.3.2. Для всех $i \in 0..n_1$ $\gamma_i^{\max} = \boldsymbol{\gamma}[n_1, n_2]$.

Доказательство. Для произвольного номера i определим j_0, такое, что $\gamma_i^{\max} = \boldsymbol{\gamma}[i, j_0] + \boldsymbol{\gamma}^*[i, j_0]$ для соответствующих LCS $\boldsymbol{\Gamma}_{ij_0}$ и $\boldsymbol{\Gamma}^*_{ij_0}$. Тогда строка $\boldsymbol{\Gamma} = \boldsymbol{\Gamma}_{ij_0}\boldsymbol{\Gamma}^*_{ij_0}$ будет наибольшей общей подпоследовательностью строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ длиной

$$\gamma_i^{\max} \leqslant \boldsymbol{\gamma}[n_1, n_2]. \tag{9.4}$$

С другой стороны, если $\boldsymbol{\Gamma}_{n_1 n_2}$ — LCS строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$, то $\boldsymbol{\Gamma}_{n_1 n_2}$ является подпоследовательностью строки $\boldsymbol{x}_2$, и ее можно записать в форме $\boldsymbol{\Gamma}_1\boldsymbol{\Gamma}_2$, где $\boldsymbol{\Gamma}_1$ — подпоследовательность подстроки $\boldsymbol{x}_1[1..i]$, $\boldsymbol{\Gamma}_2$ — подпоследовательность подстроки $\boldsymbol{x}_1[i+1..n_1]$, а значение i такое же, как в неравенстве (9.4). Тогда существует номер j_1, такой, что $\boldsymbol{\Gamma}_1$ будет подпоследовательностью подстроки $\boldsymbol{x}_2[1..j_1]$, а $\boldsymbol{\Gamma}_2$ — подпоследовательностью подстроки $\boldsymbol{x}_2[j_1+1..n_2]$. В этом случае

$$|\boldsymbol{\Gamma}_1| \leqslant |\,\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j_1])| = \boldsymbol{\gamma}[i, j_1].$$
$$|\boldsymbol{\Gamma}_2| \leqslant |\,\mathrm{LCS}(\boldsymbol{x}_1[i+1..n_1], \boldsymbol{x}_2[j_1+1..n_2])| = \boldsymbol{\gamma}^*[i, j_1].$$

Следовательно, мы вправе записать

$$\boldsymbol{\gamma}[n_1, n_2] = |\boldsymbol{\Gamma}_{n_1 n_2}| = |\boldsymbol{\Gamma}_1| + |\boldsymbol{\Gamma}_2| \leqslant \boldsymbol{\gamma}[i, j_1] + \boldsymbol{\gamma}^*[i, j_1] \leqslant \gamma_i^{\max}. \qquad (9.5)$$

Из неравенств (9.4) и (9.5) следует, что при любом выборе i выполняется равенство $\gamma_i^{\max} = \boldsymbol{\gamma}[n_1, n_2]$, что и требовалось доказать. ■

Основываясь на результате доказанной леммы, можно записать алгоритм вычисления наибольшей общей последовательности (алгоритм 9.3.3) и доказать его корректность. Этот алгоритм снова реализуется в виде функции, которая возвращает искомую LCS.

Алгоритм 9.3.3 (Алгоритм H3)

▷ *Вычисление* $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$
function H3$(n_1, \boldsymbol{x}_1, n_2, \boldsymbol{x}_2)$
    ▷ *Вычисление* LCS *для строк длиной* 0 *и* 1
    if $\min\{n_1, n_2\} = 0$ **then return** $\boldsymbol{\Gamma} \leftarrow \varepsilon$
    else if $n_1 = 1$ **then**
        if $\exists j \in 1..n_2$, такое, что $\boldsymbol{x}_1[1] = \boldsymbol{x}_2[j]$ **then**
            return $\boldsymbol{\Gamma} \leftarrow \boldsymbol{x}_1[1]$
        else
            return $\boldsymbol{\Gamma} \leftarrow \varepsilon$
        ▷ *Шаг рекурсии:* $\boldsymbol{x}_1$ *разбивается пополам*
    else
        $i \leftarrow \lfloor n_1/2 \rfloor$
        $\boldsymbol{\gamma}'_1[1..n_2] \leftarrow$ H2$(i, \boldsymbol{x}_1[1..i], n_2, \boldsymbol{x}_2[1..n_2])$
        $\boldsymbol{\gamma}'_2[1..n_2] \leftarrow$ H2$(n_1 - i, \boldsymbol{x}_1[i+1..n_1], n_2, \boldsymbol{x}_2[1..n_2])$
        Нахождение j_0, при котором достигается минимум $\boldsymbol{\gamma}'_1[j] + \boldsymbol{\gamma}'_2[j]$
        ▷ *Рекурсивный вызов функции* H3 *для решения двух подзадач*
        $\boldsymbol{\Gamma}_1 \leftarrow$ H3$(i, \boldsymbol{x}_1[1..i], j_0, \boldsymbol{x}_2[1..j_0])$
        $\boldsymbol{\Gamma}_2 \leftarrow$ H3$(n_1 - i, \boldsymbol{x}_1[i+1..n_1], n_2 - j_0, \boldsymbol{x}_2[j_0+1..n_2])$
        ▷ *Формирование окончательного решения на основе решений*
        ▷ *двух подзадач*
        return $\boldsymbol{\Gamma} \leftarrow \boldsymbol{\Gamma}_1\boldsymbol{\Gamma}_2$

Теорема 9.3.3. Алгоритм 9.3.3 корректно вычисляет $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

Доказательство. Если n_1 или n_2 равно нулю, то в этом случае LCS является пустой строкой. Если $n_1 = 1$, тогда LCS не будет пустой строкой только тогда, когда $\boldsymbol{x}_1[1]$ совпадает с какой-либо буквой строки $\boldsymbol{x}_2[1..n_2]$.

Если $n_1 > 1$, значение $\boldsymbol{\gamma}[n_1, n_2] = |\,\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)|$, получаемое после двух вызовов функции H2, вычисляется корректно вследствие леммы 9.3.2. Эти вычисления определяют позицию $j_0 \in 0..n_2$, которая разбивает строку $\boldsymbol{x}_2$ на две части.

К каждой части рекурсивно применяется функция H3 для вычисления LCS $\mathbf{\Gamma}_1$ и $\mathbf{\Gamma}_2$, соответствующих этим частям строки $\boldsymbol{x}_2$. Окончательный результат получается путем объединения $\mathbf{\Gamma}_1$ и $\mathbf{\Gamma}_2$. ■

Чтобы определить временну́ю сложность алгоритма H3, рассмотрим $T(n_1, n_2)$ — время выполнения этого алгоритма. Поскольку алгоритм H2 выполняется за время порядка $\Theta(n_1, n_2)$, существует константа K, такая, что

$$T(n_1, n_2) \leqslant Kn_1n_2 + T(n_1/2, j_0) + T(n_1/2, n_2 - j_0).$$

(Здесь сделано допущение, что j_0 вычисляется за время порядка $\Theta(n_2)$, а время порядка $O(n_2)$, необходимое для нахождения j, такого, что $\boldsymbol{x}_1[1] = \boldsymbol{x}_2[j]$, учтено в константе K.) Предположим, что для всех $i \in 1..n_1 - 1$ и $j \in 1..n_2 - 1$ существует другая константа K', такая, что $T(i, j) < K'ij$. Тогда

$$T(n_1, n_2) \leqslant Kn_1n_2 + K'(n_1/2)(j_0 + n_2 - j_0) = (K + K'/2)n_1n_2,$$

и, если положим $K' > 2K$, получим $T(n_1, n_2) < K'n_1n_2$. Отсюда по индукции мы заключаем, что алгоритм H3 выполняется за время $\Theta(n_1n_2)$.

Для оценки объема памяти, требуемого для алгоритма H3, заметим, что этот алгоритм можно организовать таким образом, что для реализации всех рекурсивных вызовов будет достаточно памяти, где хранятся строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$, а также два вектора $\boldsymbol{\gamma}'_1$ и $\boldsymbol{\gamma}'_2$. Таким образом, мы доказали такую теорему.

Теорема 9.3.4. Алгоритм 9.3.3 вычисляет наибольшую общую подпоследовательность для строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$ за время порядка $\Theta(n_1n_2)$ с использованием памяти объемом $\Theta(n_2)$. ■

Хешберг разработал три алгоритма вычисления LCS, первый из которых опубликован в 1977 году [114], два других опубликованы двумя годами позднее [115]. Первый алгоритм описан выше и является в общем случае самым медленным из трех. Два других работают быстрее, когда длина $\mathrm{lcs} = |\operatorname{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)|$ наибольшей общей подпоследовательности мала либо, наоборот, достаточно большая. Чтобы быть более точным, предположим, что $n_1 < n_2$, и пусть $\ell = \mathrm{lcs}(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Тогда второй алгоритм Хешберга вычисляет LCS за время порядка $O(\ell n_2 + n_2 \log \alpha')$, где α' $(\alpha' \leqslant \alpha)$ — количество различных букв в строке $\boldsymbol{x}_2$. Третий алгоритм Хешберга выполняется за время порядка $O((n_1 + 1 - \ell)\ell \log n_2)$. Второй алгоритм эффективен, когда LCS короткая и алфавит не велик, но может потребовать в самом худшем случае времени порядка $\Theta(n_1n_2)$. Третий алгоритм эффективен, когда LCS или очень короткая или очень длинная и может потребовать в самом худшем случае времени порядка $\Theta(n_1^2 \log n_2)$.

Упражнения 9.3

1. Докажите лемму 9.3.1.
2. Опишите изменения, которые необходимо сделать в алгоритме H2 для того, чтобы вычисления производились относительно столбцов (а не относительно строк, как в алгоритме 9.3.2).
3. Предложите усовершенствования в алгоритме Вагнера–Фишера, аналогичные алгоритму H2.
4. При доказательстве неравенства $T(n_1, n_2) < K' n_1 n_2$ для алгоритма H2 сказано "по индукции мы заключаем...". Запишите эту индукцию в явном виде.
5. Покажите по индукции, что общее количество вызовов функции H3 в точности равно $2n_1 - 1$.
6. Пусть k и k' — фиксированные действительные числа, удовлетворяющие условию $0 < k < k' < 1$. Рассмотрите класс $\mathcal{C}$ всех пар $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ строковых последовательностей, таких, что

$$kn \leqslant \mathrm{lcs}(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant k'n,$$

где $n = \min\{|\boldsymbol{x}_1|, |\boldsymbol{x}_2|\}$. Охарактеризуйте сложность второго и третьего алгоритмов Хешберга, выполняемых на строках класса $\mathcal{C}$.

9.4 Алгоритм Ханта–Шиманского

В этом разделе мы представим еще один алгоритм, который непосредственно вычисляет наибольшую общую подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ для строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$ без явного вычисления расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Конечно, если LCS найдена, то расстояние Левенштейна d_L можно вычислить на основании равенства (9.2).

Если алфавит упорядочен, алгоритм Ханта–Шиманского (Hunt–Szymanski, сокращенно — алгоритм ХШ) [122] имеет временну́ю сложность $O((n + r)\log n)$ и пространственную $O(n + r)$, где $n = \max\{n_1, n_2\}$, r — общее количество совпадений $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$, $i \in 1..n_1$, $j \in 1..n_2$. В самом худшем случае (например, если $\boldsymbol{x}_1 = \boldsymbol{x}_2 = a^n$) алгоритм ХШ выполняется за время порядка $\Theta(n^2 \log n)$ и требует памяти порядка $\Theta(n^2)$. Однако обычно $r \in O(n)$, и если алфавит упорядочен, то алгоритм выполняется за время порядка $\Theta(n \log n)$ и требует памяти порядка $\Theta(n)$, что значительно лучше того, что могут предложить алгоритмы из предыдущего раздела.

Как и многие другие алгоритмические новшества, этот алгоритм основан на очень простой идее [2]: LCS определяется с помощью "наибольшей" строго убы-

вающей и непрерывной ломаной линии, проходящей через узлы сетки, представляющие совпадения $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$. На рис. 9.1 иллюстрируется эта идея на примере $\mathrm{LCS}(rests, stress) = ress$. Здесь прочерченная непрерывная линия, определяющая LCS, содержит наибольшее возможное число диагональных "строго убывающих" сегментов. На этом рисунке можно найти *все* возможные общие подпоследовательности строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Так строки $rests$ и $stress$ имеют еще общую подпоследовательность sts. Отметим также, что хотя строки $rests$ и $stress$ и имеют относительно длинную LCS, но значение r относительно мало, удовлетворяя неравенствам $r < n_1 + n_2 < 2n$.

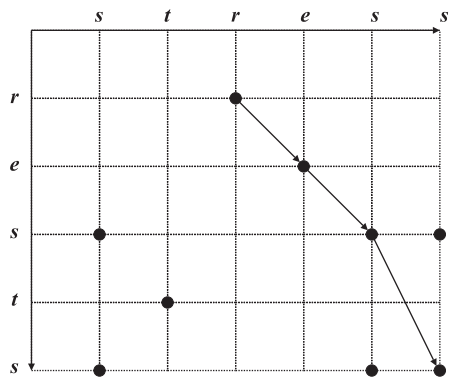


Рис. 9.1. Линия, формирующая LCS

Конечно, если работать непосредственно с сеткой размером $n_1 \times n_2$, то не получим улучшения нижней временной границы $\Omega(n_1 n_2)$ предыдущих алгоритмов. Вместо этого используется двухмерный массив $\boldsymbol{j}$, в котором элемент $\boldsymbol{j}[i,h]$ равен длине самого короткого префикса строки $\boldsymbol{x}_2$, который имеет LCS длиной h с подстрокой $\boldsymbol{x}_1[1..i]$. Другими словами, $\boldsymbol{j}[i,h]$ равен наименьшему целому числу j^*, такому, что

$$|\,\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j^*])| = h.$$

Например, для $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$ имеем $\boldsymbol{j}[4,1] = 1$, $\boldsymbol{j}[4,2] = 2$, $\boldsymbol{j}[4,3] = = 5$, при этом элементы $\boldsymbol{j}[4,h]$ не определены для $h > 3$. Аналогично $\boldsymbol{j}[5,1] = 1$, $\boldsymbol{j}[5,2] = 2$, $\boldsymbol{j}[5,3] = 5$, $\boldsymbol{j}[5,4] = 6$ и $\boldsymbol{j}[5,h]$ не определены для $h > 4$. В общем случае элементы $\boldsymbol{j}[i,h]$ определены только тогда, когда $h \leqslant |\,\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2)| \leqslant \leqslant i$. Для всех позиций $[i,h]$, где $\boldsymbol{j}[i,h]$ не определенны, "принудительно" положим $\boldsymbol{j}[i,h] = n_2 + 1$. Для полной определенности также положим $\boldsymbol{j}[i,0] = 0$ для всех $i \in 1..n_1$, $\boldsymbol{j}[0,0] = 0$ и $\boldsymbol{j}[0,h] = n_2 + 1$ для $h \in 1..n_1$. Отметим, что значения элементов массива $\boldsymbol{j}$ легко получить из рисунка, подобного рис. 9.1.

Установим некоторые свойства массива $\boldsymbol{j}$, которые будут использованы в алгоритме Ханта–Шиманского.

Лемма 9.4.1. Для всех целых $i \in 1..n_1$ и $h \in 1..i$ выполняются неравенства

$$\boldsymbol{j}[i-1, h-1] + 1 \leqslant \boldsymbol{j}[i, h] \leqslant \boldsymbol{j}[i-1, h].$$

Доказательство тривиально при $i = 1$, поэтому будем считать, что $i > 1$.

Заметим, что если элемент $\boldsymbol{j}[i-1, h]$ определен, то также определены $\boldsymbol{j}[i-1, h-1]$ и $\boldsymbol{j}[i, h]$. Если подстроки $\boldsymbol{x}_1[1..i-1]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i-1, h]]$ имеют LCS длиной h, тогда (по определению массива $\boldsymbol{j}$) подстроки $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i-1, h]]$ имеют общую подпоследовательность длиной не менее h. Отсюда следует, что $\boldsymbol{j}[i, h] \leqslant$ $\leqslant \boldsymbol{j}[i-1, h]$.

Далее, поскольку элемент $\boldsymbol{j}[i, h]$ определен, подстроки $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i, h]]$ имеют общую подпоследовательность длиной h. Удалив самые правые буквы из этих подстрок, можно уменьшить длину общей подпоследовательности не более, чем на единицу. Поэтому подстроки $\boldsymbol{x}_1[1..i-1]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i, h]-1]$ имеют общую подпоследовательность и, следовательно, LCS длиной не менее $h-1$. Это означает, что $\boldsymbol{j}[i-1, h-1] \leqslant \boldsymbol{j}[i, h]$. Что и требовалось доказать. ■

Мы используем этот результат для того, чтобы как вычисляется значение $\boldsymbol{j}[i, h]$, если уже известны значения $\boldsymbol{j}[i-1, h-1]$ и $\boldsymbol{j}[i-1, h]$. Этим будет построена основа рекурсии для алгоритма ХШ.

Лемма 9.4.2. Для всех целых $i \in 1..n_1$ и $h \in 1..|\operatorname{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2)|$ значение $\boldsymbol{j}[i, h]$ вычисляется по следующему правилу:

$$\boldsymbol{j}[i, h] \text{ равняется наименьшему целому } j^* \in \boldsymbol{j}[i-1, h-1] + 1..j[i-1, h], \quad (9.6)$$

такому, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j^*]$ (если такое j^* существует);

$$\boldsymbol{j}[i, h] = \boldsymbol{j}[i-1, h], \text{ если такого } j^* \text{ не существует.}$$

Доказательство. Сначала рассмотрим, как более простой, второй вариант вычисления $\boldsymbol{j}[i, h]$, когда такого j^* не существует. Поскольку $\boldsymbol{j}[i, h]$ определяет *наименьший* префикс строки $\boldsymbol{x}_2$, то самым правым элементом в LCS для подстрок $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i, h]]$ будет буква $\boldsymbol{x}_2[\boldsymbol{j}[i, h]]$. Вследствие леммы 9.4.1 $\boldsymbol{j}[i, h] \in$ $\in \boldsymbol{j}[i-1, h-1] + 1..\boldsymbol{j}[i-1, h]$. Но по нашему предположению не существует j^* из этого интервала, такого, что $\boldsymbol{x}_2[j^*] = \boldsymbol{x}_1[i]$, поэтому $\boldsymbol{x}_1[i] \neq \boldsymbol{x}_2[\boldsymbol{j}[i, h]]$. Следовательно, та же самая LCS будет общей подпоследовательностью и для подстрок $\boldsymbol{x}_1[1..i-1]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i, h]]$ — в таком случае $\boldsymbol{j}[i, h] \geqslant \boldsymbol{j}[i-1, h]$. Вследствие леммы 9.4.1 справедливо также обратное неравенство. Отсюда делаем заключение, что $\boldsymbol{j}[i, h] = \boldsymbol{j}[i-1, h]$.

Теперь предполагаем, что существует наименьшее целое j^* из интервала $\boldsymbol{j}[i-1, h-1] + 1..\boldsymbol{j}[i-1, h]$, такое, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j^*]$. Из этого предположения вытекает, что подстроки $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..j^*]$ имеют общую подпоследовательность,

состоящую из $\mathrm{LCS}(\boldsymbol{x}_1[1..i-1], \boldsymbol{x}_2[1..\boldsymbol{j}[i-1,h-1]])$ длиной $h-1$ и буквы $\boldsymbol{x}_2[j^*]$. Отсюда заключаем, что $\boldsymbol{j}[i,h]$ не превышает j^*.

Теперь покажем, что $\boldsymbol{j}[i,h]$ не меньше j^*. Предположим, что имеет место строгое неравенство $\boldsymbol{j}[i,h] < j^*$. Тогда, вследствие леммы 9.4.1, $\boldsymbol{j}[i-1,h-1] < \boldsymbol{j}[i,h] < j^*$. Поскольку j^* — это *наименьшее* целое из интервала $\boldsymbol{j}[i-1,h-1]+1..\boldsymbol{j}[i-1,h]$, такое, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j^*]$, то отсюда заключаем, что $\boldsymbol{x}_1[i] \neq \boldsymbol{x}_2[\boldsymbol{j}[i,h]]$. Так как по условию $|\mathrm{LCS}(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..\boldsymbol{j}[i,h]])| = h$, поэтому подстроки $\boldsymbol{x}_1[1..i-1]$ и $\boldsymbol{x}_2[1..\boldsymbol{j}[i,h]]$ также имеют общую подпоследовательность длиной h. Следовательно, $\boldsymbol{j}[i-1,h] \leqslant \boldsymbol{j}[i,h+1]$, а лемма 9.4.1 уточняет, что $\boldsymbol{j}[i-1,h] = \boldsymbol{j}[i,h+1]$. Но это невозможно, поскольку, как мы видели, $\boldsymbol{j}[i,h+1] < j^* < \boldsymbol{j}[i-1,h]$. Это противоречие и доказывает, что $\boldsymbol{j}[i,h]$ не меньше j^*. Имея также доказанным утверждение, что $\boldsymbol{j}[i,h]$ не превышает j^*, делаем вывод, что $\boldsymbol{j}[i,h] = j^*$. Теорема доказана. ■

Лемма 9.4.2 фактически устанавливает для массива $\boldsymbol{j}$ такой же результат, что и лемма 9.3.1 для массива $\boldsymbol{\gamma}$ значения в строке i можно вычислить, зная только значения в строке $i-1$. В алгоритмах H2 и H3 этот результат привел к уменьшению требуемого объема памяти от величины порядка $\Theta(n_1 n_2)$ до величины порядка $\Theta(n_2)$. Здесь этот результат также позволяет уменьшить требуемый объем памяти и, кроме того, время выполнения алгоритма, по крайней мере в среднем.

Напомним, что в алгоритме H2 был введен массив $\boldsymbol{\gamma}'[1..2, 1..n_2]$ для замены массива $\boldsymbol{\gamma}$. В алгоритме ХШ также заменяем массив $\boldsymbol{j}$ одномерным массивом $\boldsymbol{j}'[0..n_2]$. Этот алгоритм можно описать в виде последовательности таких шагов.

Шаг 1. Вычисление массива MATCHLIST

Из леммы 9.4.2 следует, что для вычисления значений $\boldsymbol{j}[i+1, 0..n_1]$ на основе значений $\boldsymbol{j}[i, 0..n_1]$ надо для каждого значения $i \in 1..n_1$ знать такую позицию j в строке $\boldsymbol{x}_2$, для которой $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$. Эту информацию в подходящей форме будем получать из массива MATCHLIST, в котором каждая позиция i является указателем на список значений j, таких, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$. Для дальнейших вычислений на шаге 3 желательно, чтобы эти значения j были упорядочены в убывающем порядке.

Отметим, что если строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ определены на упорядоченном алфавите, то массив MATCHLIST можно вычислить за время порядка $O(n \log n + r)$ и с объемом памяти порядка $\Theta(n_1 + r)$, для чего можно использовать эффективные алгоритмы сортировки или поиска по дереву. (Здесь, как и выше, $n = \max\{n_1, n_2\}$.) Но если алфавит общего вида (не упорядочен), то вычисление массива MATCHLIST в самом худшем случае может потребовать времени порядка $O(n^2)$. Детали вычисления этого массива оставим для упр. 9.4.1. (См. также задачу 4.1 из раздела 4.1.)

Шаг 2. Начальное задание значений массива $\boldsymbol{j}'$

Полагаем $\boldsymbol{j}'[0] \leftarrow 0$ и для всех $h \in 1..n_2$ полагаем $\boldsymbol{j}'[h] \leftarrow n_2 + 1$ (тем самым показываем, что элементы $\boldsymbol{j}'[h]$ не определены).

Шаг 3. Вычисление значений $\boldsymbol{j}[i,h]$ на основе значений $\boldsymbol{j}[i-1,h]$, $i = 1, 2, \ldots, n_1$

Эти вычисления выполняются путем замены значений в предыдущем массиве $\boldsymbol{j}[0..n_1]$, соответствующем номеру $i-1$, на аналогичные значения, соответствующие номеру i. Напомним (лемма 9.4.2), что $\boldsymbol{j}[i,h] = \boldsymbol{j}[i-1,h]$ (тогда значение $\boldsymbol{j}'[h]$ не изменяется) только в том случае, если не существует номера j^* из интервала $\boldsymbol{j}[i-1,h-1]+1..\boldsymbol{j}[i-1,h]$, такого, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j^*]$. Поэтому для текущего значения i и для каждого значения $j^* \in \mathrm{MATCHLIST}[i]$ по значениям массива $\boldsymbol{j}'$ надо найти такое значение h^*, что

$$\boldsymbol{j}'[h^*-1]+1 \leqslant j^* \leqslant \boldsymbol{j}'[h^*].$$

Тогда, согласно лемме 9.4.2,

$$\boldsymbol{j}'[h^*] \leftarrow j^*. \tag{9.7}$$

Каждое такое присвоение, которое изменяет значение $\boldsymbol{j}'[h^*]$, соответствует одной из r точек сетки (как показано на рис. 9.1), где $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j^*]$. Это присвоение можно интерпретировать как создание конечного узла (листа) $(i, j^*, \mathrm{LINK}[h^*-1])$ на синтаксическом дереве (см. раздел 2.1), где $\mathrm{LINK}[h^*-1]$ — указатель на узел, созданный ранее при присвоении (9.7) для $\boldsymbol{j}'[h^*-1]$. (Для $h^* = 1$ полагаем, что $\mathrm{LINK}[h^*-1] =$ `NULL`.) Когда лист создан, указатель на него хранится в $\mathrm{LINK}[h^*]$. После обработки всех $i \in 1..n_1$ наибольшее значение h, такое, что $\boldsymbol{j}'[h] < n_2+1$, равняется длине $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$, а $\mathrm{LINK}[h]$ указывает путь длиной h по дереву от листа до корня. Этот путь и определяет LCS.

Шаг 4. Запись $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ в обратном порядке

На дереве находится первый узел, который соответствует наибольшему значению h, такому, что $\boldsymbol{j}'[h] < n_2+1$. Затем записывается путь от этого узла до корня дерева — это и будет $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$, записанная в обратном порядке.

Более подробно каждый описанный шаг показан в алгоритме 9.4.1.

Алгоритм 9.4.1 (Алгоритм Ханта–Шиманского)

▷ *Вычисление* $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$
▷ *Шаг 1: Вычисление массива* MATCHLIST
Для каждого $i \in 1..n$ вычисляется $\mathrm{MATCHLIST}[i]$ — указатель на список монотонно убывающих значений j, таких, что $\boldsymbol{x}_1[i] = \boldsymbol{x}_2[j]$
▷ *Шаг 2: Начальное задание значений массива* $\boldsymbol{j}'$
$\boldsymbol{j}'[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n_1 **do**

$\boldsymbol{j}'[i] \leftarrow n_2 + 1$
LINK$[0] \leftarrow$ NULL
▷ *Шаг 3: Вычисление значений* $\boldsymbol{j}[i,h]$ *на основе значений* $\boldsymbol{j}[i-1,h]$, $1 \leqslant i \leqslant n_1$
for $i \leftarrow 1$ **to** n_1 **do**
  for all $j^* \in$ MATCHLIST$[i]$ **do**
    поиск h^*, такого, что $\boldsymbol{j}'[h^*-1]+1 \leqslant j^* \leqslant \boldsymbol{j}'[h^*]$
    if $j^* < \boldsymbol{j}'[h^*]$ **then**
      $\boldsymbol{j}'[h^*] \leftarrow j^*$
      LINK$[h^*] \leftarrow$ *новый_узел*$(i, j^*,$ LINK$[h^*-1])$
▷ *Шаг 4: Запись* LCS$(\boldsymbol{x}_1, \boldsymbol{x}_2)$ *в обратном порядке*
поиск наибольшего h, такого, что $\boldsymbol{j}'[h] < n_2 + 1$
ПУТЬ $\leftarrow$ LINK$[h]$
while ПУТЬ $\neq$ NULL **do**
  output пара (i,j) из узла, указанного указателем ПУТЬ
  продвижение указателя ПУТЬ из текущего узла далее к корню дерева
  в соответствии с указателем LINK$[h-1]$

Чтобы пояснить действия, выполняемые алгоритмом ХШ, рассмотрим наш предыдущий пример, где $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$. Здесь, как видно из рис. 9.1,

$$\begin{aligned}
&\text{MATCHLIST}[1] = <3>\\
&\text{MATCHLIST}[2] = <4>\\
&\text{MATCHLIST}[3] = <6,5,1>\\
&\text{MATCHLIST}[4] = <2>\\
&\text{MATCHLIST}[5] = <6,5,1>
\end{aligned}$$

Значения элементов массива $\boldsymbol{j}'$ после i-й итерации следующие.

$$\begin{aligned}
\text{Начальные значения} &: 077777\\
i = 1 &: 0\underline{\mathbf{3}}7777\\
i = 2 &: 03\underline{\mathbf{4}}777\\
i = 3 &: 014\underline{\mathbf{5}}77\\
i = 4 &: 012577\\
i = 5 &: 0125\underline{\mathbf{6}}7
\end{aligned}$$

Подчеркнутые значения в массиве $\boldsymbol{j}'$ определяют LCS в обратном порядке:

$$(i=5, j^*=6), (i=3, j^*=5), (i=2, j^*=4), (i=1, j^*=3).$$

Теорема 9.4.3. Для двух заданных строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$, определенных на упорядоченном алфавите, алгоритм 9.4.1 корректно вычисляет LCS$(\boldsymbol{x}_1, \boldsymbol{x}_2)$ за

время порядка $O((n+r)\log n)$ с использованием памяти объемом порядка $O(n+$ $+r)$, где $n=\max\{n_1,n_2\}$, r — общее число пар (i,j), таких, что $\boldsymbol{x}_1[i]=\boldsymbol{x}_2[j]$.

Доказательство корректности алгоритма 9.4.1 непосредственно следует из утверждений лемм 9.4.1 и 9.4.2.

Для анализа времени выполнения алгоритма и объема требуемой памяти рассмотрим каждый шаг алгоритма в отдельности. Как мы уже отмечали, если алфавит упорядочен, то на шаге 1 требуется время порядка $O(n\log n+r)$ и память порядка $\Theta(n+r)$. Шаг 2 выполняется за время порядка $O(n_1)$ и требует $\Theta(n_1)$ памяти для хранения массива $\boldsymbol{j}'$.

На шаге 3 для выполнения цикла **for** требуется время порядка $O(n_1)$, плюс необходимо время для выполнения таких операций:

а) вычисление r значений массива MATCHLIST,

б) поиск значения h^* в массиве $\boldsymbol{j}'$,

в) не более r модификаций массива $\boldsymbol{j}'$ и создание не более r узлов дерева.

Для выполнения операции *а*) требуется время порядка $\Theta(r)$, для выполнения операции *б*) можно применить алгоритм бинарного поиска со временем выполнения порядка $O(r\log n_1)$, для операции *в*) необходимо $O(r)$ времени. Таким образом, общее время выполнения шага 3 не превышает величины порядка $O(n_1+r\log n_1)$ с использованием памяти объемом порядка $O(r)$.

Шаг 4 требует $O(n_1)$ времени для определения позиции h в массиве $\boldsymbol{j}'$ и $O(r)$ времени для вывода LCS. Суммируя время и объемы памяти, необходимые для выполнения каждого шага алгоритма, получаем требуемый результат. ■

В работе [17] предложен вариант алгоритма ХШ, который выполняется за время порядка $O(n_1\log n_2+t\log(2n_1n_2/t))$, где t — количество "главных" совпадений между буквами строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Поскольку $t\leqslant r$, максимальное время выполнения этого алгоритма составляет величину порядка $\Theta(n_1n_2)$, тогда как минимальное время — величину порядка $\Theta(n_1\log n_2)$.

Упражнения 9.4

1. Предложите алгоритм вычисления массива MATCHLIST в алгоритме ХШ, предполагая, что

а) алфавит упорядочен,

б) алфавит не упорядочен.

2. В каждом варианте алгоритма укажите его асимптотическую сложность в самом худшем и самом лучшем случаях и приведите примеры строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$, на которых достигаются эти границы.

3. В алгоритме ХШ, как и в других алгоритмах вычисления LCS, строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ можно менять местами. На основе этого покажите, что временну́ю и пространственную границы сложности этого алгоритма, данные в теореме 9.4.1, можно немного усилить.

9.5 Алгоритм Укконена–Майерса

В этом разделе мы возвращаемся к основной идее алгоритма Вагнера–Фишера: итерационному процессу вычисления массива стоимостей $\boldsymbol{c} = \boldsymbol{c}[0..n_1, 0..n_2]$, основанному на рекуррентных соотношениях леммы 9.1.1. Однако здесь мы представим массив $\boldsymbol{c}$ в виде ориентированного ациклического графа G с одной вершиной-источником, соответствующей позиции $[0, 0]$ массива $\boldsymbol{c}$, и вершинами-стоками, соответствующими позициям $[n_1, j]$ и $[i, n_2]$ ($1 \leqslant i \leqslant n_1$, $1 \leqslant j \leqslant n_2$) этого массива, включая позицию $[n_1, n_2]$. Мы снова используем лемму 9.1.1 для нахождения пути наименьшей стоимости ("кратчайшего" пути) от $\boldsymbol{c}[0, 0]$ до $\boldsymbol{c}[n_1, n_2]$. Одновременно с этим используем само значение расстояния $d = d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ как "инструмент" для распознавания тех вершин и дуг графа G, которые невозможно эффективно вычислить. В результате применения этих соображений получим алгоритм, который назовем алгоритмом Укконена–Майерса (сокращенно, УМ). Этот алгоритм, независимо предложенный Укконеном (Ukkonen) [223] и Майерсом (Myers) [189], вычисляет d за время порядка $O(nd)$, где $n = \min\{n_1, n_2\}$. Поскольку обычно значение d значительно меньше значения n, то алгоритм УМ представляет несомненный интерес.

Существенно, что этот же алгоритм можно использовать в других часто встречающихся на практике ситуациях, где значение d представляет верхнюю границу (или пороговое значение) неизвестного значения расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ — можно модифицировать алгоритм таким образом, чтобы он возвращал или значение $d(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant d$, или значение ∞ (если это расстояние вычислить невозможно).

Как уже указывалось, вершины графа $G = (V, E)$[2] соответствуют позициям $[i, j]$ массива $\boldsymbol{c}$ для всех $i \in 0..n_1$ и $j \in 0..n_2$. Каждая вершина $[i, j] \in V$ имеет не более трех входящих дуг $([i', j'], [i, j])$:

$$\begin{aligned}
([i-1, j], [i, j]) \in E, \quad &\text{если} \quad \boldsymbol{c}[i, j] - \boldsymbol{c}[i-1, j] = d(\boldsymbol{x}_1[i], \varepsilon); \\
([i, j-1], [i, j]) \in E, \quad &\text{если} \quad \boldsymbol{c}[i, j] - \boldsymbol{c}[i, j-1] = d(\varepsilon, \boldsymbol{x}_2[j]); \\
([i-1, j-1], [i, j]) \in E, \quad &\text{если} \quad \boldsymbol{c}[i, j] - \boldsymbol{c}[i-1, j-1] = d(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j]).
\end{aligned} \tag{9.8}$$

Для любой вершины $[i, j] \in V$ в соответствии с леммой 9.1.1 существует хотя бы одна входящая дуга только в том случае, если $[i, j] \neq [0, 0]$. Припишем каждой дуге $([i', j'], [i, j])$ вес $\boldsymbol{c}[i, j] - \boldsymbol{c}[i', j']$.

[2] Здесь использовано стандартное обозначение графа $G = (V, E)$ как совокупность двух множеств: V — множество вершин, E — множество дуг графа. — *Примеч. ред.*

Назовем граф G ***графом зависимостей*** массива $\boldsymbol{c}$. Нетрудно заметить, что

- $|E| \leqslant 3(|V| - 1)$;
- граф G является подграфом графа, структура которого в точности соответствует структуре массива $\boldsymbol{c}$ (в графе G отсутствуют дуги, веса которых не равны минимальной стоимости);
- существуют ориентированные пути от вершины $[0, 0]$ до каждой вершины $[i, j] \in V$;
- сумма весов дуг, составляющих ориентированный путь от вершины $[0, 0]$ до вершины $[i, j]$, равна стоимости последовательности операций редактирования минимальной стоимости, необходимых для преобразования подстроки $\boldsymbol{x}_1[1..i]$ в подстроку $\boldsymbol{x}_2[1..j]$;
- граф G имеет одну вершину-источник $[0, 0]$ и не более $n_1 + n_2 - 1$ вершин-стоков, все из которых соответствуют или элементам строки n_1, или элементам столбца n_2 массива $\boldsymbol{c}$.

На рис. 9.2 показан граф зависимостей для примера $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$, где использовано расстояние Левенштейна.

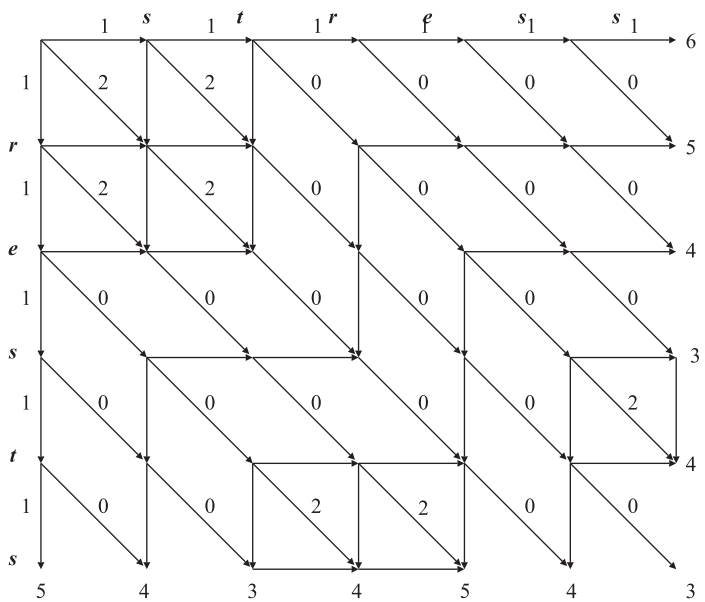


Рис. 9.2. Ориентированный граф зависимостей G для $d_L(rests, stress)$

Теперь покажем, как можно эффективно удалить из графа G вершины и дуги, которые не лежат на пути минимальной стоимости, проходящем от вершины $[0, 0]$

до вершины $[n_1, n_2]$. Покажем, что эти вычисления можно выполнить за время порядка $\Theta(nd)$.

Сначала заметим, что для каждой вершины $[i', j']$, лежащей на пути от вершины $[0, 0]$ до вершины [i,j], выполняется неравенство $\boldsymbol{c}[i, j] \geqslant \boldsymbol{c}[i', j']$. Следовательно, на пути минимальной стоимости к вершине $[n_1, n_2]$ не может находиться вершины $[i', j']$, для которых $\boldsymbol{c}[i', j'] > \boldsymbol{c}[i, j]$, и поэтому все такие вершины и все пути, которые ведут только к таким вершинам, можно удалить из графа G. Справедливо и обратное утверждение: все вершины и все дуги, которые не удовлетворяют условию $\boldsymbol{c}[i', j'] > \boldsymbol{c}[i, j]$, должны лежать на пути к вершине $[n_1, n_2]$ и поэтому *должны* представлять некоторую последовательность операторов редактирования минимальной стоимости для расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Как показано на рис. 9.3 для нашего примера, это простое наблюдение может значительно упростить граф G.

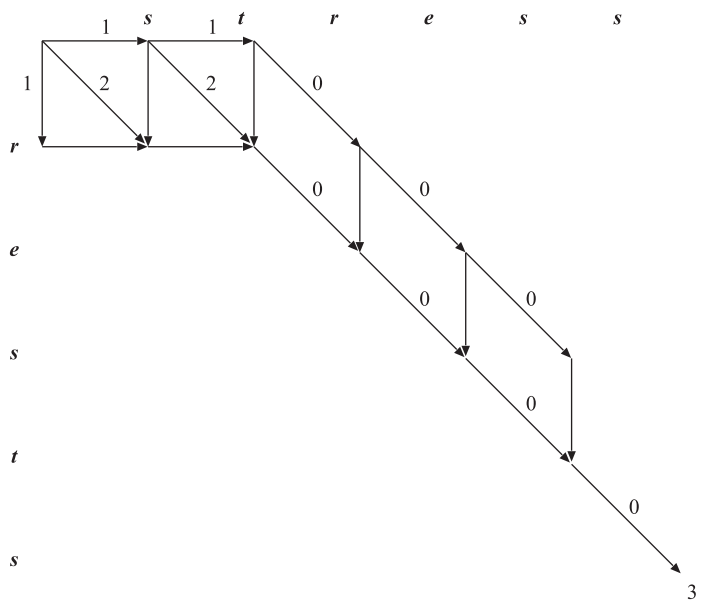


Рис. 9.3. Редуцированный граф зависимостей для $d_L(rests, stress)$

Если внимательно рассмотреть рис. 9.3, то нетрудно заметить, что в редуцированном графе все операции вставки и удаления имеют стоимость, в точности равную 1. Также можно заметить, что в этом графе нет вершин, отстоящих более чем на $\lceil d/2 \rceil = 2$ "шага" от диагональной линии, которая начинается в вершине $[0, 1]$ и заканчивается в вершине $[n_1, n_2] = [5, 6]$. Более точно: в редуцированном графе не может быть более $\lfloor (n_1 - n_2 + d)/2 \rfloor$ вершин, лежащих ниже этой диагональной линии, и не может быть более такого же числа вершин, лежащих выше этой линии.

Поскольку сама диагональная линия состоит не более чем из n_1 дуг, то отсюда следует, что в редуцированном графе $O(n_1 d)$ вершин (и столько же дуг). Поэтому задачу построения редуцированного графа можно решить за время порядка $O(n_1 d)$ с использованием объема памяти такого же порядка.

В оставшейся части этого раздела предыдущие нестрогие рассуждения выразим в строгой математической форме, что также даст нам возможность оценить функцию расстояния посредством положительных значений стоимости операций вставки и удаления. Это в свою очередь приведет нас к желаемому алгоритму. Для упрощения выкладок далее в этом разделе будем предполагать, что $n_2 \geqslant n_1$. Для функции расстояния d, если она удовлетворяет условиям метрики, это предположение не является ограничением, поскольку в таком случае $d(\boldsymbol{x}_1, \boldsymbol{x}_2) = d(\boldsymbol{x}_2, \boldsymbol{x}_1)$ для любой пары строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$.

Пусть d^* — минимальная стоимость операций удаления и вставки, выполняемых по всем буквам алфавита A, т.е.

$$d^* = \min_{\lambda \in A} \{d(\lambda, \varepsilon), d(\varepsilon, \lambda)\} > 0.$$

Обозначим через $\Delta_{i,j}$ диагональ (идущую слева направо), на которой лежит вершина $[i, j]$. Думаю, что читатель не будет протестовать, если $\Delta_{i,j}$ припишем значение $j - i$. Таким образом, $\Delta_{i,j}$ может принимать целочисленные значения из интервала от $-n_1$ до n_2.

Теперь рассмотрим ориентированный путь π на графе G от вершины $[i', j']$ до вершины $[i, j]$. Если $\Delta_{i,j} - \Delta_{i',j'} \geqslant 0$, тогда путь π содержит не менее $\Delta_{i,j} - \Delta_{i',j'}$ операций удаления. Если же $\Delta_{i,j} - \Delta_{i',j'} \leqslant 0$, тогда путь π содержит не менее $\Delta_{i',j'} - \Delta_{i,j}$ операций вставки. Поэтому, учитывая соотношения (9.8), можно записать

$$\boldsymbol{c}[i, j] - \boldsymbol{c}[i', j'] \geqslant |\Delta_{i,j} - \Delta_{i',j'}| d^* = |(j - i) - (j' - i')| d^*. \tag{9.9}$$

В частности, полагая $[i', j'] = [0, 0]$, получаем неравенство $\boldsymbol{c}[i, j] \geqslant |j - i| d^*$, справедливое для любой вершины $[i, j]$, лежащей на пути от $[0, 0]$ до $[n_1, n_2]$. Поскольку $\boldsymbol{c}[i, j] \leqslant \boldsymbol{c}[n_1, n_2]$ для любой такой вершины $[i, j]$ и имеет место равенство $d = d(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{c}[n_1, n_2]$, то отсюда следует, что

$$d/d^* \geqslant |j - i|, \tag{9.10}$$

поэтому $|\Delta_{i,j}| \leqslant \lfloor d/d^* \rfloor$. В нашем примере $\boldsymbol{x}_1 = rests$, $\boldsymbol{x}_2 = stress$ и $d^* = 1$, поэтому первоначальное множество диагоналей $\{-3, -2, -1,\ 0, 1, 2,\ 3\}$ графа G должно быть уменьшено по крайней мере вдвое, чтобы получить редуцированный граф, показанный на рис. 9.3.

Однако не все так просто. Рассмотрим произвольный путь на графе G от вершины $[0, 0]$ до вершины $[n_1, n_2]$, проходящий через некоторую промежуточную

вершину $[i,j]$. Применяя дважды неравенство (9.9), сначала к пути $[i,j]-[n_1,n_2]$, а затем к пути $[0,0]-[i,j]$, и учитывая, что $\boldsymbol{c}[0,0]=0$, получим усиление неравенства (9.10):

$$d/d^* \geqslant |j-i|+|(n_2-n_1)-(j-i)|. \tag{9.11}$$

Рассмотрим два возможных случая.

- $j \leqslant i$ (в этом случае $\Delta_{i,j} \leqslant 0$)

  В данной ситуации неравенство (9.11) перепишется как $d/d^* \geqslant (n_2-n_1)+$ $+2(i-j)$, поскольку мы положили, что $n_2 \geqslant n_1$. Поскольку здесь $i-j$ — неотрицательное целое число, последнее неравенство можно переписать в виде

  $$q \geqslant i-j \geqslant 0, \tag{9.12}$$

  где

  $$q = \lfloor (d/d^* - (n_2-n_1))/2 \rfloor .$$

  Неравенство (9.12) справедливо для всех промежуточных вершин $[i,j]$, лежащих в графе G на пути $[0,0]-[n_1,n_2]$.

- $j \geqslant i$ (в этом случае $\Delta_{i,j} \geqslant 0$)

  Здесь надо исследовать две ситуации, которые зависят от того, будет ли выполняться неравенство $n_2-n_1 \leqslant j-i$. Если $n_2-n_1 > j-i$, то неравенство (9.11) запишется в виде $d/d^* \geqslant n_2-n_1$. Отсюда для величины q, определяемой формулой (9.13), получаем оценку

  $$q \geqslant 0. \tag{9.13}$$

  С другой стороны, для ситуации $n_2-n_1 \leqslant j-i$ находим, что

  $$d/d^* \geqslant 2(j-i)-(n_2-n_1),$$

  откуда следует, что

  $$q+(n_2-n_1) \geqslant j-i \geqslant 0. \tag{9.14}$$

Итак, в результате наших небольших исследований получено три неравенства (9.12), (9.14) и (9.15). Неравенство (9.14) в данный момент не представляет для нас интереса, поскольку оно выражает тривиальное условие, что значение d/d^* должно быть равно не менее половины разности длин строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Неравенства (9.12) и (9.15) объединим как

$$-q \leqslant j-i \leqslant q+(n_2-n_1). \tag{9.15}$$

Еще раз подчеркнем, что эти неравенства справедливы для любой промежуточной вершины $[i,j]$, лежащей в графе G на произвольном пути от вершины $[0,0]$ до

вершины $[n_1, n_2]$. Поскольку $j - i = \Delta_{i,j}$, то неравенства (9.16) можно рассматривать как ограничения на промежуточные вершины $[i, j]$, т.е. эти вершины должны находиться в графе G между диагоналями $-q$ и $q + (n_2 - n_1)$. Если применить неравенства (9.16) к нашему примеру, где $\boldsymbol{x}_1 = rests$, $\boldsymbol{x}_2 = stress$ и $d^* = 1$, то получим, что $q = 1$, а эти неравенства запишутся как $-1 \leqslant j - i \leqslant 2$, что соответствует редуцированному графу на рис. 9.3, где все вершины располагаются на диагоналях $\{-1, 0, 1, 2\}$.

Чтобы создать алгоритм на основе неравенств (9.16), предварительно рассмотрим такую задачу: для некоторого заданного положительного целого числа D надо определить, будет ли выполняться неравенство $d(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant D$. Для решения этой задачи на основе формулы (9.13) можно сначала вычислить пробное значение

$$Q = \lfloor (D/d^* - (n_2 - n_1))/2 \rfloor$$

и затем найти минимальную сумму весов на путях от вершины [0,0] до вершины $[n_1, n_2]$, которые лежат между диагоналями, задаваемые неравенствами (9.16):

$$-Q \leqslant \Delta_{i,j} \leqslant Q + (n_2 - n_1).$$

Если эта минимальная сумма, которая "по идее" должна быть равна $\boldsymbol{c}[n_1, n_2]$, больше значения D, то это означает, что не все вершины [i,j] включены в рассчитываемую сумму для того, чтобы получить минимальное значение $\boldsymbol{c}[n_1, n_2]$. В этом случае необходимо включить в рассмотрение дополнительные диагонали и, соответственно, вершины, которые лежат на них. А для этого нужно увеличить значение Q и, следовательно, значение D. Если же вычисленная сумма не превышает D, то это означает, что $\boldsymbol{c}[n_1, n_2] \leqslant D$. Эти вычисления реализуем в виде функции TRIAL_DISTANSE (пробное расстояние), которая во всех случаях возвращает значение $\boldsymbol{c}[n_1, n_2]$, но это значение будет считаться "правильным" только тогда, когда $\boldsymbol{c}[n_1, n_2] \leqslant D$.

```
▷ Вычисление c[n1, n2] на основе заданного значения D
function TRIAL_DISTANCE(D)
    ▷ На основе формулы (9.13) вычисляется Q
    ▷ и затем интервал диагоналей
    Q ← ⌊(D/d* − (n2 − n1))/2⌋
    ▷ Вычисление c[n1, n2]
    for i ← 1 to n1 do
        for j ← max{1, i − Q} to min{n1, i + Q + (n2 − n1)} do
            c[i, j] ← min{c[i − 1, j] + d(x1[i], ε),
                c[i, j − 1] + d(ε, x2[j]),
                c[i − 1, j − 1] + d(x1[i], x2[j])}
    return c[n1, n2]
```

Теперь все, что остается сделать алгоритму Укконена–Майерса (представлен ниже как алгоритм 9.5.1), так это инициализировать массив $\boldsymbol{c}$ и затем предложить подходящие значения D для функции TRIAL_DISTANCE. По причинам, которые будут понятны после обсуждения сложности алгоритма УМ, начальное значение D берется равным 1, а затем удваивается до тех пор, пока не будет выполнено неравенство $\boldsymbol{c}[n_1, n_2] \leqslant D$. На основе обсуждений, проведенных выше для обоснования алгоритма, мы можем утверждать, что алгоритм УМ выполняется корректно.

Алгоритм 9.5.1 (Алгоритм Укконена–Майерса)

▷ *Вычисление* $d(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{c}[n_1, n_2]$
▷ *Присвоение значений элементам нулевых столбца и строки массива* $\boldsymbol{c}$
$\boldsymbol{c}[0,0] \leftarrow 0$
for $i \leftarrow 1$ **to** n_1 **do**
    $\boldsymbol{c}[i,0] \leftarrow \boldsymbol{c}[i-1,0] + d(\boldsymbol{x}_1[i], \varepsilon)$
for $j \leftarrow 1$ **to** n_2 **do**
    $\boldsymbol{c}[0,j] \leftarrow \boldsymbol{c}[0,j-1] + d(\varepsilon, \boldsymbol{x}_2[j])$
▷ *Инициализация* D
$D \leftarrow 1$
▷ *Удвоение значения* D *до тех пор,*
▷ *пока не будет достигнуто неравенство* $D \geqslant$ TRIAL_DISTANCE(D)
while $D <$ TRIAL_DISTANCE(D) **do**
    $D \leftarrow 2D$
output $\boldsymbol{c}[n_1, n_2]$

Для оценки сложности нового алгоритма рассмотрим сначала функцию TRIAL_DISTANCE, в частности двойной цикл **for**. Эти циклы выполняются не более $2Q + +(n_2 - n_1) + 1$ раз, поскольку, вследствие формулы (9.13), справедливо неравенство

$$2Q + (n_2 - n_1) + 1 \leqslant D/d^* + 1.$$

Таким образом, общее время выполнения функции TRIAL_DISTANCE имеет порядок $O(n_1 D)$. В алгоритме 9.5.1 эта функция выполняется $h + 1$ раз для значений $D = 1, 2, \ldots, 2^h$, где h — наименьшее положительное целое, такое, что $d = d(\boldsymbol{x}_1, \boldsymbol{x}_2) \leqslant 2^h$. Следовательно, общее время выполнения функции TRIAL_DISTANCE в алгоритме 9.5.1 составит

$$O(1 + 2 + \cdots + 2^h)n_1) = O(2^{h+1} n_1) = O(4 n_1 d) = O(n_1 d).$$

Наши выкладки доказывают следующую теорему.

Теорема 9.5.1. Для заданных строк $\boldsymbol{x}_1 = \boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2 = \boldsymbol{x}_2[1..n_2]$, $n_2 \geqslant n_1$, алгоритм 9.5.1 корректно вычисляет $d = d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ за время порядка $O(n_1 d)$ с использованием памяти объемом порядка $\Theta(n_1 n_2)$. ■

Как показано в упр. 9.5.2, объем памяти, необходимый для выполнения алгоритма УМ, можно уменьшить до величины порядка $O(n_1 d)$. В упр. 9.5.3 предложено модифицировать этот алгоритм таким образом, чтобы и время его выполнения имело порядок $O(n_1 d)$. Однако этого можно добиться только тогда, когда известна верхняя граница d для расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

Для специальных случаев расстояний преобразования и Левенштейна в работе [144] предложена "пошаговая" версия алгоритма УМ, которая позволяет вычислить $d(\boldsymbol{x}_1, \lambda\boldsymbol{x}_2)$ и соответствующий массив стоимостей на основе известных расстояния $d(\boldsymbol{x}_1, \lambda\boldsymbol{x}_2)$ и его массива стоимостей за время порядка $O(d)$, где d — снова известная верхняя граница для расстояния, а λ — произвольная буква. Если эта верхняя граница не известна, то время выполнения такого алгоритма возрастает до величины порядка $O(n_1 + n_2)$. Поскольку $d(\boldsymbol{x}_1, \boldsymbol{x}_2\lambda)$ можно приближенно вычислить на основе массива стоимостей для $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$, алгоритм [144] можно использовать для вычисления искомого расстояния последовательно, начиная с $d(\varepsilon, \varepsilon)$ путем добавления к начальным строкам новых букв, на каждом шаге по одной букве (при этом не имеет значения, в начало или конец строк добавляются новые буквы). Если верхняя граница d для расстояния определяется на основе подобных вычислений, то время выполнения данного алгоритма имеет порядок $O(d(n_1 + n_2))$, т.е. такой же порядок, что и алгоритм УМ. Однако пошаговый алгоритм позволяет эффективно решать другие задачи приближенного сравнения с паттерном, в частности, задачи P1 и P2, сформулированные в разделе 13.4. Алгоритм [144] в вычислительном плане весьма сложен, поэтому более подробно мы его рассматривать не будем.

Более простую реализацию пошаговая идея нашла в алгоритме [133], который позволяет вычислить $d(\boldsymbol{x}_1, \lambda\boldsymbol{x}_2)$ или $d(\boldsymbol{x}_1, \boldsymbol{x}_2\lambda)$ за время порядка $O(n_1 + n_2)$ независимо от того, известна ли верхняя граница d для расстояния. Этот алгоритм предпочтительнее других, если верхняя граница d неизвестна или велика, например $d \in \Theta(n_1)$.

Наконец отметим, что в алгоритме УМ, подобно алгоритму Вагнера–Фишера, наибольшую общую подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ можно восстановить, двигаясь по графу G в обратном порядке от вершины $[n_1, n_2]$ до вершины $[0, 0]$.

Упражнения 9.5

1. В теореме 9.5.1 утверждается, что алгоритм УМ выполняется за время $O(n_1 d)$, хотя только для инициализации массива $\boldsymbol{c}$ необходимо время порядка $\Omega(n_2)$. Чтобы развеять все сомнения в корректности этой теоремы, докажите, что $n_2 \in O(n_1 d)$.

2. В теореме 9.5.1 утверждается, что для выполнения алгоритма УМ необходим объем памяти порядка $\Theta(n_1 n_2)$. Предложите усовершенствования в этом

алгоритме, которые позволили бы сократить необходимый объем памяти до величины порядка $O(n_1 d)$.

3. Покажите, как можно модифицировать алгоритм УМ для вычисления $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ в случае, когда известна верхняя граница d этого расстояния.

9.6 Заключение

В этой главе мы рассмотрели четыре основных алгоритма, которые для заданных строк $\boldsymbol{x}_1 = \boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2 = \boldsymbol{x}_2[1..n_2]$ вычисляют или расстояние $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$, или наибольшую общую подпоследовательность $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

- Алгоритм Вагнера–Фишера вычисляет расстояние $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ для произвольной функции расстояния d за время порядка $\Theta(n_1 n_2)$ с использованием памяти объемом такого же порядка. Если $d = d_L$ (расстояние Левенштейна), то этот алгоритм позволяет также найти $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ в качестве "сопутствующего продукта" вычисления расстояния.
- Алгоритм Хешберга (алгоритм H3) вычисляет $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ (следовательно, и расстояние Левенштейна $d_L(\boldsymbol{x}_1, \boldsymbol{x}_2)$) за время порядка $\Theta(n_1 n_2)$, но использует память объемом порядка $\Theta(n)$, где $n = \min\{n_1, n_2\}$.
- Алгоритм Ханта–Шиманского вычисляет $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ за время, которое обычно имеет порядок $\Theta(n \log n)$, с использованием памяти объемом порядка $\Theta(n)$, где $n = \max\{n_1, n_2\}$. Временна́я сложность этого алгоритма превышает $\Theta(n \log n)$ только тогда, когда строки $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ имеют количество совпадающих букв, превышающее величину порядка $\Theta(n)$ (например, когда размер алфавита мал по отношению к длине строк).
- Алгоритм Укконена–Майерса вычисляет расстояние $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ за время порядка $\Theta(dn)$ с использованием памяти объемом порядка $\Theta(n_1 n_2)$, где $n = \min\{n_1, n_2\}$. Как и в алгоритме Вагнера–Фишера, этот алгоритм позволяет эффективно вычислить и $\mathrm{LCS}(\boldsymbol{x}_1, \boldsymbol{x}_2)$, если используется функция расстояния Левенштейна d_L. Этот алгоритм более предпочтителен, по сравнению с другими алгоритмами, если известна верхняя граница d для расстояния $d(\boldsymbol{x}_1, \boldsymbol{x}_2)$ и эта граница не слишком велика (например, $d \in O(\log n)$ или даже $d \in O(n/\log n)$). Такая ситуация часто встречается на практике: обычно вычисление расстояния между строками представляет интерес только тогда, когда это расстояние относительно мало по сравнению с длиной строк.

  Для расстояния преобразования d_E, как показано в работе [189], если входные строки выбираются случайным образом, то алгоритм УМ *в среднем* выполняется за время порядка $O(n + k^2)$, а в работах [189, 149] доказано, что с использованием дерева суффиксов этот алгоритм *в самом худшем случае* выполняется за время порядка $O(n \log \alpha + k^2)$, где α — размер алфавита.

Пошаговые версии алгоритма УМ применимы для расстояний преобразования и Левенштейна [144, 133].

Существуют другие алгоритмы, решающие те же задачи, что и рассмотрены в этой главе, или близкие к ним. В работе [173] показано, как "четыре русских" метода [20] можно применить для разбиения массива стоимостей на перекрывающиеся квадратные подматрицы так, чтобы время вычисления расстояния имело порядок $O(n^2/\log n)$. Однако эти алгоритмы становятся предпочтительнее алгоритма УМ только для очень больших строк (когда $n \geqslant 300\,000$). В статье [128] предложены алгоритмы для "тяжелых" общих подпоследовательностей, а в статье [204] — для больших "возрастающих" подпоследовательностей. Несколько недавних алгоритмов [63, 64, 125] предлагают алгоритмы для вычисления LCS, построенные на основе использования бинарных векторов [77, 7, 24], которые обсуждаются в разделе 7.4 и главе 10. В работе [63] показано, как можно реализовать алгоритмы Хешберга и Ханта–Шиманского с использованием таких векторов. В общем случае эти алгоритмы выполняются за время порядка $O(n_1 n_2/w)$, где w — длина машинного слова. Такие алгоритмы показали себя очень быстрыми на практике. Также быстрым является алгоритм Эллисона и Дикса (Allison and Dix) [7], который использует бинарные векторы для вычисления *длины* LCS.

ГЛАВА 10

Приближенное сравнение с паттерном

> Газета состоит из одинакового количества слов независимо от того, содержит она новости или нет.
>
> — Генри Филдинг (1707–1754)

Количество различных алгоритмов приближенного сравнения с паттернами, предложенных в период с 80-х годов и до настоящего времени, сравнимо с количеством алгоритмов точного сравнения с паттернами (см. главы 7 и 8). На момент написания этой книги можно было без труда заметить, что для построения всего множества всех наиболее эффективных и востребованных алгоритмов приближенного сравнения используются всего две идеи, с которыми мы уже имели удовольствие познакомиться.

1. В разделе 7.4 был описан алгоритм Демелки–Бейза-Ятса–Гоннета (сокращенно, алгоритм ДБГ) для вычисления точного совпадения с паттерном. В этом алгоритме использовались бинарный вектор $\boldsymbol{s} = \boldsymbol{s}[1..m]$, который описывает текущее "состояние" вычислений, и "кодирующая" бинарная матрица $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$, ставящая в соответствие элементам паттерна $\boldsymbol{p}[1..m]$ буквы $1..\alpha$ индексированного алфавита. В этом случае можно эффективно реализовать бинарные операции на бинарных массивах $\boldsymbol{s}$ и $\boldsymbol{t}$ для пересчета вектора состояний $\boldsymbol{s}$ с целью поиска совпадения с паттерном следующей буквы $\boldsymbol{x}[i+1]$, входящей в заданную текстовую строку $\boldsymbol{x} = \boldsymbol{x}[1..n]$.

В разделах 10.2–10.4 мы покажем три применения этой идеи для решения задачи приближенного сравнения с паттерном.

2. В разделе 9.1 был введен двухмерный массив стоимостей $\boldsymbol{c}$, элементы которого определялись равенством $\boldsymbol{c}[i,j] = d(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j])$, т.е. равны расстоянию между префиксами строк $\boldsymbol{x}_1 = \boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2 = \boldsymbol{x}_2[1..n_2]$. Отношения базовой рекурсии между смежными элементами массива $\boldsymbol{c}$ дали возможность в главе 9 построить два алгоритма прямого вычисления расстояния между строками (алгоритмы Вагнера–Фишера и Укконена–Майерса) и два алгоритма вычисления наибольшей общей подпоследовательности LCS для двух строк (алгоритмы Хешберга и Ханта–Шиманского).

   В разделе 10.1 мы немного изменим определение элементов массива $\boldsymbol{c}$, что позволит адаптировать алгоритм Вагнера–Фишера для приближенного сравнения с паттерном. В разделах 10.3 и 10.4 рассмотрим несколько вариаций реализации и применения массива $\boldsymbol{c}$ и его возможного "сотрудничества" с бинарными массивами алгоритма ДБГ. Наконец, в разделе 10.5 дан краткий обзор теоретических исследований временно́й сложности недавних алгоритмов сравнения с паттерном, использующих функции расстояния преобразования и Левенштейна.

Отметим, что в работе [191] дан современный обзор алгоритмов приближенного сравнения с паттерном, включая описание методов, которые не рассматриваются в данной книге. Кроме того, в этой работе приведены результаты экспериментов по оценке относительной эффективности алгоритмов.

10.1 Алгоритм для произвольной функции расстояния

Мы начали главу 9 с обсуждения массива стоимостей $\boldsymbol{c}$ и его свойств. Здесь мы пойдем таким же путем. Переопределим массив стоимостей $\boldsymbol{c} = \boldsymbol{c}[0..n_1, 0..n_2]$ из главы 9 как массив $\boldsymbol{c} = \boldsymbol{c}[0..n, 1..m]$ с элементами, которые определяются соотношением

$$c[i,j] = \min_{0 \leqslant i' \leqslant i} d(x[i'..i], p[1..j]), \tag{10.1}$$

справедливым для всех $i \in 0..n$, $j \in 1..m$. Здесь, как и ранее, d — произвольная функция расстояния, удовлетворяющая условиям (2.2)–(2.5) метрики, включая функцию взвешенного расстояния (раздел 2.2). Минимум по i' в выражении (10.1) необходим вследствие того, что, как показано в упражнении 10.1.1, здесь необязательно выполнение равенства $i - i' + 1 = j$, т.е. не исключаются случаи, когда $|\boldsymbol{x}[i'..i]| \neq |\boldsymbol{p}[1..j]|$. Таким образом, $\boldsymbol{c}[i,j]$ теперь равняется расстоянию между префиксом паттерна $\boldsymbol{p}[1..j]$ и "ближайшим" суффиксом подстроки $\boldsymbol{x}[1..i]$.

В частности, $\boldsymbol{c}$[i,m] теперь равно минимальному расстоянию между паттерном $\boldsymbol{p}$ и подстроками строки $\boldsymbol{x}$, которые оканчиваются в позиции i.

Из соотношения (10.1) следует, что начальные значения для элементов нулевой строки массива $\boldsymbol{c}$ здесь вычисляются точно так же, как и в разделе 9.1:

- для всех $j \in 1..m \boldsymbol{c}[0,j] = d(\varepsilon, \boldsymbol{p}[1..j])$ — суммарная стоимость операций вставки для префикса $\boldsymbol{p}[1..j]$ паттерна $\boldsymbol{p}$.

Однако начальные значения для первого столбца массива $\boldsymbol{c}$ вычисляются способом, отличным от того, как это делается в разделе 9.1, поскольку здесь фактически находится минимум расстояний между буквой $\boldsymbol{p}[1]$ и подстроками $\boldsymbol{x}[i'..i]$, а не вычисляется расстояние между $\boldsymbol{p}[1]$ и единственным префиксом $\boldsymbol{x}[1..i]$. Конечно, если $\boldsymbol{x}[i] = \boldsymbol{p}[1]$, тогда мы положим

$$\boldsymbol{c}[i,1] \leftarrow 0, \tag{10.2}$$

но если $\boldsymbol{x}[i] \neq \boldsymbol{p}[1]$, то в этом случае необходимо применить формулу

$$\boldsymbol{c}[i,1] \leftarrow \min\{\boldsymbol{c}[i-1,1] + d(\boldsymbol{x}[i], \varepsilon), d(\boldsymbol{x}[i], \boldsymbol{p}[1])\}. \tag{10.3}$$

Отметим, что различия в инициализации массивов стоимостей $\boldsymbol{c}$ здесь и в разделе 9.1 — это *единственное* различие в вычислении этих массивов. Если значения в нулевой строке и первом столбце массива $\boldsymbol{c}$ вычислены корректно, тогда, поскольку для произвольной функции расстояния операции вставки и удаления можно упорядочить любым способом (упражнение 2.2.14), справедливы утверждения леммы 9.1.1, задающие отношения базисной рекурсии для вычисления элементов массива $\boldsymbol{c}$. Поэтому, как и ранее, для всех $i \in 1..n$ и $j \in 2..m$ элементы $\boldsymbol{c}[i,j]$ вычисляются по формуле

$$\begin{aligned} \boldsymbol{c}[i,j] \leftarrow \min\{&\boldsymbol{c}[i-1,j] + d(\boldsymbol{x}[i], \varepsilon), \\ &\boldsymbol{c}[i,j-1] + d(\varepsilon, \boldsymbol{p}[j]), \\ &\boldsymbol{c}[i-1,j-1] + d(\boldsymbol{x}[i], \boldsymbol{p}[j])\}. \end{aligned} \tag{10.4}$$

Используем пример из главы 9, где вычислялось расстояние между строками *rests* и *stress*, для того чтобы показать различие между массивами $\boldsymbol{c}$, вычисленным в главе 9 и вычисленным по формулам данной главы. Обозначим через $\boldsymbol{c}_{\text{dist}}$ массив стоимостей, вычисленный на основе значений расстояний (как в главе 9), а через $\boldsymbol{c}_{\text{best}}$ — массив, вычисленный на основе значений наилучшего (best) приближенного совпадения с паттерном всех суффиксов, которые оканчиваются в позиции i. Если для вычисления массива стоимостей используется расстояние Левенштейна, то получим массив $\boldsymbol{c}_{\text{dist}}$, который представим в табл. 10.1. Массив $\boldsymbol{c}_{\text{best}}$ для $\boldsymbol{p} =$ *stress* и $\boldsymbol{x} =$ *rests* показан в табл. 10.2. Отметим, что все различия в этих массивах определяются только различными значениями в столбце, соответствующего $j = 1$.

Таблица 10.1. Массив $\boldsymbol{c}_{\text{dist}}$ для $d_L(rests, stress)$

	$\boldsymbol{j}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**
i		ε	s	t	r	e	s	s
0	ε	0	1	2	3	4	5	6
1	r	1	2	3	2	3	4	5
2	e	2	3	4	3	2	3	4
3	s	3	2	3	4	3	2	3
4	t	4	3	2	3	4	3	4
5	s	5	4	3	4	5	4	3

Таблица 10.2. Массив $\boldsymbol{c}_{\text{best}}$ для $\boldsymbol{p} = stress$ и $\boldsymbol{x} = rests$

	$\boldsymbol{j}$	**1**	**2**	**3**	**4**	**5**	**6**
i		s	t	r	e	s	s
0	ε	1	2	3	4	5	6
1	r	2	3	2	3	4	5
2	e	2	3	3	2	3	4
3	s	0	1	2	3	2	3
4	t	1	0	1	2	3	4
5	s	0	1	2	3	2	3

Для любого неотрицательного целого числа k по столбцу j массива $\boldsymbol{c}_{\text{best}}$ можно определить такую позицию i, для которой существует подстрока $\boldsymbol{u} = \boldsymbol{x}[i'..i]$, что $d(\boldsymbol{u}, \boldsymbol{p}[1..j]) \leqslant k$. В этом случае будем говорить, что в позиции i строки $\boldsymbol{x}$ ***имеет место*** k***-приближенное совпадение*** с префиксом $\boldsymbol{p}[1..j]$ паттерна $\boldsymbol{p}$. Например, взяв $j = 5$ и $k = 2$ из табл. 10.2, находим, что в позициях $i = 3$ и $i = 5$ имеют место 2-приближенные совпадения с $\boldsymbol{p}[1..5] = stres$. Аналогично для $j = 3$ и $k = 1$ в позиции $i = 4$ имеет место 1-приближенное совпадение с $\boldsymbol{p}[1..3] = str$. Отметим, что совпадающие подстроки $\boldsymbol{u}$ не обязательно будут единственными.

Очевидно, что алгоритм Вагнера–Фишера (см. раздел 9.2), если в нем для инициализации массива стоимостей применить формулы (10.2) и (10.3), можно использовать для решения задачи приближенного сравнения с паттерном. Но, как отмечено выше, длина $i - i' + 1$ совпадающей подстроки $\boldsymbol{u}$ может не совпадать с длиной m паттерна. Поэтому, если необходимо определить хотя бы одну подстроку $\boldsymbol{u} = \boldsymbol{x}[i'..i]$, частично совпадающую с паттерном, то также необходимо указать и значение i'. Это нетрудно сделать, если знать последовательность перемещений v, введенных в разделе 9.2. Эти перемещения определяют последо-

вательность операторов редактирования, необходимых для преобразования подстроки $\boldsymbol{u}$ в паттерн $\boldsymbol{p}$ (или, что то же самое, для преобразования $\boldsymbol{p} \to \boldsymbol{u}$). Зная перемещения, можно вычислить значение i'. Как показано в упражнении 10.1.3, количество перемещений не превышает $2|\boldsymbol{p}| = 2m$, тогда как количество k-приближенных совпадений в любой позиции i не превышает n. Поэтому общее время вычисления всех возможных значений i' имеет порядок $O(mn)$, т.е. не превышает время, необходимое для вычисления массива $\boldsymbol{c}$. Следовательно, справедлива такая теорема.

Теорема 10.1.1. Для заданных текстовой строки $\boldsymbol{x}[1..n]$, паттерна $\boldsymbol{p}[1..m]$ и неотрицательного целого числа k модифицированный алгоритм Вагнера–Фишера вычисляет все k-приближенные совпадения строки $\boldsymbol{x}$ с паттерном $\boldsymbol{p}$ за время порядка $O(mn)$ с использованием памяти, объем которой имеет такой же порядок. ■

Еще раз подчеркнем, что модифицированный алгоритм Вагнера–Фишера вычисляет *все* k-приближенные совпадения строки $\boldsymbol{x}$ с любым префиксом $\boldsymbol{p}[1..j]$ $(1 \leqslant j \leqslant m)$ паттерна $\boldsymbol{p}$.

Конечно, алгоритм, описанный в этом разделе, очень затратный как по используемому времени, так и по используемому объему памяти. Однако, как было показано в упражнении 9.3.3 и как будет показано в упражнении 10.1.4, объем используемой памяти можно легко уменьшить, если заметить, что в массиве $\boldsymbol{c}$ значения в строке i зависят только от значений в строке $i-1$. Кроме того, данный алгоритм имеет одно несомненное достоинство: он применим для работы с любыми функциями расстояния, которые удовлетворяют аксиомам (2.2)–(2.5) метрики. В следующем разделе мы покажем, как путем наложения различного рода ограничений на область применения алгоритмов приближенного сравнения можно значительно уменьшить их временну́ю сложность.

Упражнения 10.1

1. Для расстояния Левенштейна d_L приведите пример паттерна $\boldsymbol{p}[1..m]$ и текстовой строки $\boldsymbol{x}$, таких, что в формуле (10.1) элемент $\boldsymbol{c}[i,m]$ вычислялся бы при значении i', которое удовлетворяет соотношениям

   а) $i + i' - 1 < m$;

   б) $i + i' - 1 = m$;

   в) $i + i' - 1 > m$.

   Затем покажите, что все эти три соотношения могут иметь место тогда, когда стоимость некоторой операции подстановки превышает стоимость операций вставки и удаления.

2. Докажите, что для расстояния преобразования d_E без потери общности в выражении (10.1) можно считать, что $i + i' - 1 \leqslant m$.

3. Покажите, что максимальное число перемещений v, которые преобразуют совпадающую подстроку $\boldsymbol{u}$ в паттерн $\boldsymbol{p}$, равно $2|\boldsymbol{p}|$.

4. Запишите алгоритм (модификацию алгоритма Вагнера–Фишера, соответствующую теореме 10.1.1), который вычислял бы все целые i и соответствующие целые i', такие, что $d(\boldsymbol{x}[i'..i], \boldsymbol{p}) \leqslant k$. Покажите, что для реализации этого алгоритма достаточно памяти объемом порядка $\Theta(\min\{mn, m^2\})$. Далее покажите, что если не требуется выводить значения i', то объем памяти можно уменьшить до величины порядка $\Theta(m)$.

10.2 Алгоритм, вычисляющий несовпадения

В этом разделе мы покажем, как уже известный нам алгоритм ДБГ из раздела 7.4 можно легко модифицировать так, чтобы он вычислял k-приближенные совпадения, рассчитанные на основе расстояния Хемминга (см. раздел 2.2). В действительности, это только первое из нескольких возможных применений идеи алгоритма ДБГ к различным типам задач приближенного сравнения с паттерном. Здесь представим идею этого алгоритма в более общем контексте, чем мы делали это ранее. Вместе с тем в этом разделе мы будем применять только расстояние Хемминга, поэтому задачу, решаемую с помощью алгоритма данного раздела, назовем ***задачей k-несовпадений***.

Далее будем считать, что задано некоторое целое число $k \geqslant 0$, а также заданы паттерн $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и текстовая строка $\boldsymbol{x} = \boldsymbol{x}[1..n]$, которые определены на алфавите A. Как и в алгоритме ДБГ, здесь мы требуем, чтобы алфавит A был конечен, индексирован и известен заранее. В этом случае можно считать (см. упражнение 4.1.3), что $A = \{1, 2, \ldots, \alpha\}$.

В исходном алгоритме ДБГ использовался ***вектор состояний*** $\boldsymbol{s} = \boldsymbol{s}[1..m]$, в котором элемент $\boldsymbol{s}[j]$ указывал, есть ли совпадение текущей подстроки $\boldsymbol{x}[i - j + 1..i]$ строки $\boldsymbol{x}$ с префиксом $\boldsymbol{p}[1..j]$ паттерна $\boldsymbol{p}$. Поскольку в разделе 7.4 мы рассматривали точное совпадение с паттерном, поэтому элементы $\boldsymbol{s}[j]$ принимали только два значения: $\boldsymbol{s}[j] = 0$, если имело место совпадение, и $\boldsymbol{s}[j] = 1$ в случае несовпадения. Здесь мы рассматриваем k-приближенное сравнение с паттерном на основе расстояния Хемминга, поэтому естественно сейчас положить, что элементы $\boldsymbol{s}[j]$ будут равны ***количеству*** несовпадающих позиций при сравнении $\boldsymbol{p}[1..j]$ и текущей подстрокой строки $\boldsymbol{x}$. Поскольку число несовпадающих позиций может быть от 0 до m, то для хранения этого числа необходимо не менее $\lceil \log_2(m+1) \rceil$ бит на каждый элемент вектора $\boldsymbol{s}$, а не один бит, как в алгоритме ДБГ. Обозначим через B число $\lceil \log_2(m+1) \rceil$ — количество битов, требуемых для хранения каждого элемента вектора $\boldsymbol{s}$ (ниже мы покажем, как можно уменьшить это число).

Далее вместо формулы (7.13) для вычисления $\boldsymbol{s}[j]$ в исходном алгоритме ДБГ будем использовать формулу

$$\boldsymbol{s}[j] = d_H(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]). \tag{10.5}$$

Эта формула применима для любого $j \in 1..m$, тогда как i последовательно принимает значения $1, 2, \ldots, n$.

Напомним, что в алгоритме ДБГ также использовался массив $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$, где для каждой буквы $h \in 1..\alpha$ и для каждой позиции $j \in 1..m$

$$\boldsymbol{t}[h,j] = 0, \text{ если } \boldsymbol{p}[j] = h, \text{ и } \boldsymbol{t}[h,j] = 1 \text{ в противном случае.} \tag{10.6}$$

Здесь мы также используем массив $\boldsymbol{t}$, но теперь каждый элемент этого массива будет не одиночным битом, а будет содержать $B \geqslant 1$ бит. При этом $\boldsymbol{t}[h,j]$ будет также равняться 0 или 1, но перед ними будет стоять $B-1$ значащих нулей. Эти дополнительные нули необходимы для того, чтобы можно было просто и эффективно вычислить сумму $\boldsymbol{s} + \boldsymbol{t}[h, 1..m]$, аналог операции логического сложения $\boldsymbol{s} \vee \boldsymbol{t}[h, 1..m]$ в алгоритме ДБГ. Как предложено показать в упражнении 10.2.2, массив $\boldsymbol{t}$ можно эффективно вычислить на предварительном этапе алгоритма.

Используя новые формулы (10.5) и (10.6) для вычисления вектора $\boldsymbol{s}$ и массива $\boldsymbol{t}$, можем сформулировать и доказать результат, аналогичный лемме 7.4.1 (здесь операция логического сложения заменяется на обычную операцию сложения).

Лемма 10.2.1. Пусть $\boldsymbol{s}^{(i)}$ — вектор состояний, соответствующий позиции i строки $\boldsymbol{x}$. Начальные состояния определяются как $\boldsymbol{s}^{(0)}[0..m] = 0^B 0^B \ldots 0^B$. Тогда для всех $i \in 1..n$ и $j \in 1..m$ справедлива формула

$$\boldsymbol{s}^{(i)}[j] = \boldsymbol{s}^{(i-1)}[j-1] + \boldsymbol{t}[\boldsymbol{x}[i], j]. \tag{10.7}$$

■

Доказательство в точности повторяет схему доказательства леммы 7.4.1, оставим его для упражнения 10.2.1. ■

Отметим, что возможны другие леммы, подобные леммам 10.2.1 и 7.4.1, если предложить оператор ($+$, $\vee$ или какой-либо другой), который на основе значений для позиции $j-1$ предыдущего состояния корректно пересчитывал бы значения для позиции j текущего состояния. Другими словами, леммы 10.2.1 и 7.4.1 можно рассматривать как частные случаи более общей леммы, где определен некий оператор (назовем его оператором $\boldsymbol{op}$), удовлетворяющий этому условию. В дальнейшем при использовании идеи алгоритма ДБГ мы будем учитывать, что возможна такая общая лемма.

Подобно тому, как в разделе 7.4 на основе леммы 7.4.1 был создан алгоритм ДБГ, здесь на основе леммы 10.4.1 создадим аналогичный алгоритм для при-

ближенного сравнения с паттерном и назовем его алгоритмом ПДБГ ("приближенный" алгоритм ДБГ). В этом новом алгоритме на каждом шаге выполняются вычисления

$$\boldsymbol{s} \leftarrow \mathit{rightshift}(\boldsymbol{s}, B) + \boldsymbol{t}[\boldsymbol{x}[i], 1..m], \tag{10.8}$$

где

- сначала вектор состояний $\boldsymbol{s}$ с помощью процедуры *rightshift* сдвигается вправо на B бит (с учетом B нулей, входящих в значение $\boldsymbol{s}[0]$), тем самым осуществляется переход от состояния $j-1$ к состоянию j ($j \in 1..m$);
- затем для всех m позиций в сдвинутом векторе $\boldsymbol{s}$ выполняется пересчет путем прибавления соответствующих m элементов массива $\boldsymbol{t}$, определяемых текущей буквой $\boldsymbol{x}[i]$ строки $\boldsymbol{x}$.

Реализация алгоритма ПДБГ очень похожа на реализацию алгоритма ДБГ, поэтому оставим ее для упражнения 10.2.3.

Чтобы оценить сложность нового алгоритма, сначала заметим, что для хранения вектора $\boldsymbol{s}[1..m]$ требуется mB бит, а массива $\boldsymbol{t}$ — αmB бит. Так же как и в алгоритме ДБГ, предположим, что машинное слово имеет длину w бит. Тогда для хранения вектора $\boldsymbol{s}[1..m]$ требуется $\lceil mB/w \rceil$ машинных слов, а массива $\boldsymbol{t}$ — $\alpha\lceil mB/w \rceil$ машинных слов. Далее предположим, что арифметические операции (такие, как сдвиг и сложение) на отдельных машинных словах могут выполняться за константное время. Тогда для каждого значения i операцию (10.8) можно выполнить за время порядка $\Theta(\lceil mB/w \rceil)$. На этом основании можем сформулировать аналог теоремы 7.4.3.

Теорема 10.2.2. Для заданных текстовой строки $\boldsymbol{x}[1..n]$, паттерна $\boldsymbol{p}[1..m]$ и неотрицательного целого числа k алгоритм ПДБГ вычисляет все k-приближенные совпадения строки $\boldsymbol{x}$ с паттерном $\boldsymbol{p}$ за время порядка $\Theta(\lceil mB/w \rceil n)$ с использованием памяти объемом порядка $\Theta(\alpha\lceil mB/w \rceil)$. ■

Поскольку w — константа, а $B \in \Theta(\log m)$, то отсюда делаем заключение, что алгоритм ПДБГ имеет время выполнения порядка $\Theta(mn \log m)$. Но это еще не та эффективность, которую хотелось бы иметь. Конечно, если длина паттерна m мала настолько, что величина $\lceil mB/w \rceil$ ограничена небольшим целым числом, то в этом случае можно утверждать, что алгоритм ПДБГ на практике будет иметь время выполнения порядка $\Theta(n)$. Но чтобы быть уверенным в том, что величина mB/w действительно невелика, следует уменьшить B до величины порядка $\Theta(\log k)$.

Сделаем два почти очевидных наблюдения. Во-первых, мы можем считать, что всегда $k < m$, поскольку для $k \geqslant m$ задача k-приближенного сравнения тривиальна. Во-вторых, значение $\boldsymbol{s}[j]$ нет необходимости вычислять точно, если это значение превышает k. Действительно, когда $\boldsymbol{s}[j]$ впервые превышает k, то достаточно просто зафиксировать этот факт, так как несовпадение с паттерном $\boldsymbol{p}[1..m]$ уже имеет место и не важно, будут ли зафиксированы еще другие совпадения

или несовпадения с суффиксами $\boldsymbol{p}[j+1..m]$ паттерна. Эти простые наблюдения позволяют утверждать, что для хранения необходимой информации достаточно $\lceil \log_2(k+1) \rceil$ бит плюс один бит переполнения (фиксирующий, что значение $\boldsymbol{s}[j]$ превышает k) для каждого $m+1$ элемента вектора $\boldsymbol{s}$ и, вследствие этого, также для каждого элемента массива $\boldsymbol{t}$. Модифицированный таким образом алгоритм ПДБГ удовлетворяет теореме 10.2.2 с константой $B = \lceil \log_2(k+1) \rceil + 1$ и асимптотически имеет время выполнения порядка $\Theta(mn \log k)$. Как отмечалось в разделе 7.4, фактически $w \in \Omega(\log n)$, поэтому порядок времени выполнения алгоритма можно записать как $O(\lceil m \log k / \log n \rceil n)$. Отметим, что в этом случае для $k = 1$, $B = 1$ алгоритм ПДБГ ведет себя так же, как и алгоритм БДГ (что и ожидалось).

Для объяснения вычислительного механизма модифицированного алгоритма ПДБГ рассмотрим пример, взятый из работы [24] и представленный в табл. 10.3. Здесь рассматривается задача нахождения 2-приближенных совпадений паттерна $\boldsymbol{p} = ababc$ со строкой $\boldsymbol{x} = abdababababc$ на основе расстояния Хемминга. Поскольку здесь $k = 2$, необходимо $B = 3$ бит для каждого элемента вектора $\boldsymbol{s}$ (предполагая, что значения этих элементов не превысят 3) плюс один бит переполнения (который положим равным 1, если количество несовпадений превысит 3). Для удобства вычислений в момент времени, когда бит переполнения становится равным 1, его поместим в отдельный вектор, который обозначим $\boldsymbol{s}'$. Вертикальные стрелки в табл. 10.3 указывают на две позиции в строке $\boldsymbol{x}$, где достигается 2-приближенное совпадение с паттерном; точнее, это две позиции, где выполняется условие $\boldsymbol{s}[5] + 2^2\boldsymbol{s}'[5] \leqslant 2$.

Таблица 10.3. Сравнение паттерна $\boldsymbol{p} = ababc$ со строкой $\boldsymbol{x} = abdababababc$, $k = 2$, с помощью модифицированного алгоритма ПДБГ

$\boldsymbol{x}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{d}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{c}$
$\boldsymbol{t}[\boldsymbol{x}[i]]$	01011	10101	11111	01011	10101	01011	10101	01011	10101	11110
$\boldsymbol{s}$	01011	10202	12131	02220	10323	02003	10301	02001	10301	12100
$\boldsymbol{s}'$	01111	00111	00011	00001	00000	00010	00001	00010	00001	00010
								↑		↑

Бит переполнения в элементе $\boldsymbol{s}[j]$ используется в следующей процедуре для проверки, превышает ли $\boldsymbol{s}[j]$ значения k. Если бит переполнения равен 0, то он просто игнорируется. Если же он равен 1, то его надо переписать в вектор $\boldsymbol{s}'$ в ту же позицию j, а затем в векторе $\boldsymbol{s}$ его следует вернуть в состояние 0. После выполнения такой процедуры величина $\boldsymbol{s}[j] + 2^{B-1}\boldsymbol{s}'[j]$ будет нижней границей количества несовпадений в позиции j паттерна $\boldsymbol{p}$ (т.е. количество несовпадений в позиции j не меньше указанной величины). Описанную процедуру можно реализовать с помощью оператора логического умножения $\boldsymbol{op} = \text{AND}$ с использованием шаблона $0^B10^B1 \ldots 0^B1$ для извлечения битов переполнения из вектора $\boldsymbol{s}$ и инверсного шаблона $1^B01^B0 \ldots 1^B0$ для возврата битов переполнения в нуле-

вое состояние. (Напомним, что бинарный логический оператор AND возвращает значение 1 только в том случае, если оба его операнда имеют значение 1.)

Из табл. 10.3 видно, что векторы $\boldsymbol{s}$ и $\boldsymbol{s}'$ на каждом шаге алгоритма должны сдвигаться вправо одновременно, что обеспечит синхронизацию позиций j в этих векторах. Это позволит проверить выполнение неравенства

$$\boldsymbol{s}[m] + 2^{B-1}\boldsymbol{s}'[m] \leqslant k,$$

для того чтобы определить, имеет ли место k несовпадений паттерна $\boldsymbol{p}[1..m]$ в текущей позиции строки $\boldsymbol{x}$.

Детали реализации модифицированного алгоритма ПДБГ оставим для упражнения 10.2.3. Отметим только, что объем памяти, необходимый для хранения векторов $\boldsymbol{s}$ и $\boldsymbol{s}'$ и соответствующих шаблонов, ограничен величиной порядка $O(\lceil mB/w \rceil)$, что и требуется для выполнения условий теоремы 10.2.2.

Рассмотрим пару небольших примеров, чтобы лучше разобраться в пространственных и временны́х требованиях алгоритма ПДБГ. Предположим, что длина машинного слова $w = 32$. Для аппроксимационной константы $k \leqslant 7$ значение B равно 4. Поэтому для любого паттерна длиной $m \leqslant 8$ величина $\lceil mB/w \rceil = 1$. В этом случае для хранения одного элемента вектора $\boldsymbol{s}[1..8]$ (следовательно, и вектора $\boldsymbol{s}'[1..8]$) достаточно одного машинного слова, а алгоритм ПДБГ будет выполняться за время порядка $\Theta(n)$. Если же $k \leqslant 3$, тогда значение $B = 3$, и для любого паттерна длиной $m \leqslant 10$ величина $\lceil mB/w \rceil$ снова равна 1. Продолжение подобного анализа оставим для упражнения 10.2.4.

Другие алгоритмы

Первым практическим алгоритмом для решения задачи k-несовпадений, по-видимому, был алгоритм, предложенный Ландау и Вишкиным (Landau, Vishkin) [146, 147]. Этот алгоритм использует двухмерный массив "несогласованности паттерна", который вычисляется на предварительном этапе алгоритма. Затем алгоритм вычисляет массив "несогласованности текста", на основе которого определяется k-приближенное совпадение паттерна и заданной текстовой строки. Включая предварительный этап, этот алгоритм выполняется за время порядка $O(k(n + m\log m))$. Как показано в упражнении 10.2.5, время выполнения алгоритма Ландау–Вишкина в самом худшем случае превышает верхнюю границу времени выполнения алгоритма ПДБГ на достаточно широком множестве входных данных. Гелилом и Джейнкарло (Galil, Giancarlo) [95] время выполнения алгоритма Ландау–Вишкина в самом худшем случае было улучшено до величины порядка $O(kn + m\log m)$ путем использования дерева суффиксов. Однако и этот алгоритм показал на практике медленную работу [24]. Другой алгоритм, использующий бинарные операции, описан в работе [23]. Этот алгоритм подсчитывает длины частичных совпадений с паттерном $\boldsymbol{p}$ для каждой позиции i строки $\boldsymbol{x}$. Если в алфавите размером α все буквы в паттерне $\boldsymbol{p}$ и строке $\boldsymbol{x}$ могут появиться с одинаковой

вероятностью, то *математическое ожидание* времени выполнения этого алгоритма составит величину порядка $\Theta(mn/\alpha)$, что является некоторым улучшением по сравнению с алгоритмом ПДБГ в случае больших значений α (например, если $\alpha \in \Omega(m)$ или $\alpha \in \Omega(n)$).

Подход, использованный при построении алгоритма Бойера–Мура (раздел 7.2), также можно использовать для решения задачи k-несовпадений. В работе [218] описана обобщенная версия алгоритма Бойера–Мура–Хоспула (раздел 8.2), которая вычисляет все k-приближенные совпадения паттерна $\boldsymbol{p}$ и строки $\boldsymbol{x}$ за среднее время порядка $O\left(k\left(\frac{1}{m-k}+\frac{k}{\alpha}\right)n\right)$. Таким образом, этот алгоритм может конкурировать с алгоритмом ПДБГ в случае, если значение k мало по сравнению с m и α и значение m велико по сравнению с длиной машинного слова w. Представляет большой интерес предложенное Эль-Мабруком и Крочемором (El-Mabrouk, Crochemore) [82] объединение алгоритма ПДБГ и алгоритма Бойера–Мура для вычисления точного совпадения с паттерном. На практике для больших значений m и малых значений k этот гибридный алгоритм выполняется немного быстрее, чем алгоритм ПДБГ.

Даже на основе такого краткого знакомства с другими возможными алгоритмами для решения задачи k-несовпадений, можно сделать вывод, что на сегодняшний день алгоритм ПДБГ и его варианты, по-видимому, показывают наилучшие результаты в решении этой задачи.

Упражнения 10.2

1. Докажите лемму 10.2.1.

2. Запишите алгоритм, который вычислял бы массив $\boldsymbol{t}$ за время порядка $\Theta(\alpha mB/w)$.

3. Запишите исходный алгоритм ПДБГ (когда $B = \lceil \log_2(m+1) \rceil$) и его модификацию (когда $B = \lceil \log_2(k+1) \rceil + 1$).

4. Составьте таблицу, где бы для $k = 0, 3, 7, 15, 31, 63$ и $\lceil mB/w \rceil \in 1..10$ приводилась максимальная возможная длина паттерна m, с которым при этих условиях может работать алгоритм ПДБГ. Рассмотрите два варианта таблицы: когда длина машинного слова $w = 31$ и когда $w = 64$.

5. Для положительных целых чисел $n \geqslant m \geqslant k \geqslant 2$ докажите, что при выполнении неравенства $m \log_2 m/n \leqslant 2$ также справедливо неравенство

$$mn \log_2 k / \log_2 n < k(n + m \log_2 m).$$

Используя доказанное неравенство, сравните асимптотическую эффективность алгоритмов ПДБГ и Ландау–Вишкина.

10.3 Алгоритмы, вычисляющие разности

В этом разделе рассмотрим алгоритмы приближенного сравнения с паттерном, основанные на расстоянии преобразования. Назовем задачи, решаемые такими алгоритмами, ***задачами k-разностей***. Напомним, что для расстояния преобразования стоимости всех однобуквенных операций удаления и вставки и операций подстановки положительны и равны между собой, т.е.

$$d_E(\varepsilon, \lambda) = d_E(\lambda, \varepsilon) = d_E(\lambda, \mu) > 0 \tag{10.9}$$

для любой буквы λ из алфавита A и для любых букв $\lambda \neq \mu$. Конечно, если $\lambda = \mu$, то $d_E(\lambda, \lambda) = 0$. Поскольку стоимость всех однобуквенных операций одинаковая, то без потери общности мы можем положить эту стоимость равной 1. На основании формул (10.2), (10.3) и леммы 9.1.1 нетрудно показать, что для расстояния преобразования массив стоимостей $\boldsymbol{c}$ можно представить в виде табл. 10.4. Здесь элементы δ_{i1} в столбце 1 вычисляются следующим образом (лемма 9.1.1):

$$\delta_{i1} = 0, \text{ если } \boldsymbol{x}[i] = \boldsymbol{p}[1], \text{ и } \delta_{i1} = 1, \text{ если } \boldsymbol{x}[i] \neq \boldsymbol{p}[1]. \tag{10.10}$$

Таблица 10.4. Массив стоимостей для расстояния преобразования

	$\boldsymbol{j}$	**0**	**1**	**2**	**...**	$\boldsymbol{m}$
i		ε	$\boldsymbol{p}[1]$	$\boldsymbol{p}[2]$	...	$\boldsymbol{p}[m]$
0	ε	0	1	2	...	m
1	$\boldsymbol{x}[1]$	0	δ_{11}	...	...	...
2	$\boldsymbol{x}[2]$	0	δ_{21}	...	...	...
		...	...	...	...	...
n	$\boldsymbol{x}[n]$	0	δ_{n1}	...	...	...

Теперь докажем важный результат, который первоначально в более общей форме был установлен в работе [173], а затем в форме, представленной ниже, приведен в статье [226].

Лемма 10.3.1. Для любого массива стоимостей $\boldsymbol{c} = \boldsymbol{c}[0..n, 0..m]$, вычисленного с использованием расстояния преобразования, справедливы соотношения ($i \in 1..n, j \in 1..m$):

а) $\boldsymbol{c}[i,j] - \boldsymbol{c}[i,j-1] \in \{-1, 0, 1\}$;

б) $\boldsymbol{c}[i,j] - \boldsymbol{c}[i-1,j] \in \{-1, 0, 1\}$;

в) $\boldsymbol{c}[i,j] - \boldsymbol{c}[i-1,j-1] \in \{0, 1\}$.

Доказательство. Сначала заметим, что вследствие леммы 9.1.1 и равенств (10.9) значение любого элемента $\boldsymbol{c}[i,j]$ массива $\boldsymbol{c}$ не может превышать более чем на 1 значения элементов $\boldsymbol{c}[i-1,j]$, $\boldsymbol{c}[i,j-1]$ и $\boldsymbol{c}[i-1,j-1]$. Поскольку все значения массива $\boldsymbol{c}$ являются целыми числами, то остается только доказать, что разности в утверждении леммы не могут быть меньше -1 (соотношения a) и b) или 0 (соотношение c)).

Докажем соотношение a) по индукции, заметив в табл. 10.4, что это соотношение выполняется для $i = 1$. Предположим, что оно справедливо для некоторого $i \geqslant 1$ и для всех j. Теперь предположим, что для $i+1$ и для некоторого наименьшего j соотношение a не выполняется, т.е. имеет место неравенство

$$\boldsymbol{c}[i+1,j] - \boldsymbol{c}[i+1,j+1] < -1.$$

Но тогда $\boldsymbol{c}[i+1,j-1] > \boldsymbol{c}[i+1,j] + 1$ и, следовательно,

$$\begin{aligned}\boldsymbol{c}[i+1,j] &= \min\{\boldsymbol{c}[i,j], \boldsymbol{c}[i,j-1], \boldsymbol{c}[i+1,j-1]\} + 1 = \\ &= \min\{\boldsymbol{c}[i,j], \boldsymbol{c}[i,j-1]\} + 1.\end{aligned}$$

Отсюда вытекает, что

$$\boldsymbol{c}[i+1,j-1] > \min\{\boldsymbol{c}[i,j], \boldsymbol{c}[i,j-1]\} + 2.$$

С другой стороны, мы знаем, что для расстояния преобразования должно выполняться неравенство

$$\boldsymbol{c}[i,j-1] \geqslant \boldsymbol{c}[i+1,j-1] - 1 \tag{10.11}$$

и поэтому должно выполняться неравенство

$$\boldsymbol{c}[i+1,j-1] \geqslant \boldsymbol{c}[i,j] + 2. \tag{10.12}$$

Из неравенств (10.11) и (10.12) вытекает, что

$$\boldsymbol{c}[i-1,j] - \boldsymbol{c}[i,j] \geqslant \boldsymbol{c}[i+1,j-1] - 1 - \boldsymbol{c}[i,j] > 1.$$

Последнее неравенство противоречит лемме 9.1.1 для расстояния преобразования. Это противоречие и доказывает, что не существует таких значений j, для которых соотношение a не выполняется.

Соотношение b следует из соотношения a вследствие симметрии формулы (10.4) относительно i и j. Доказательство соотношения c оставим для упражнения 10.3.1. ■

Вооружившись этой леммой, теперь можем представить два алгоритма, решающих задачу k-разностей. Первый алгоритм предложен Укконеном [226]. В этом алгоритме скомбинированы результаты леммы 10.3.1 и подход, реализованный

в алгоритме Укконена–Майерса (раздел 9.5). Он выполняется в среднем за время порядка $O(kn)$. Второй алгоритм — это новый алгоритм, предложенный Майерсом [187]. Этот алгоритм основан на той же идее, что и алгоритм ДБГ (раздел 7.4), и использует в вычислениях бинарные операции. Его время выполнения в самом худшем случае имеет порядок $O(mn/w)$, т.е. не зависит от k. Идеи алгоритма Укконена можно применить для модификации алгоритма Майерса, после чего его время выполнения в среднем будет иметь порядок $O(kn/w)$. Алгоритм Ву–Менбера, третий алгоритм для решения задачи k-разностей, описан в разделе 10.4. Хотя этот алгоритм требует времени выполнения в самом худшем случае порядка $O(kmn/w)$, его область применения значительно шире, чем области применения алгоритмов Укконена и Майерса.

10.3.1 Алгоритм Укконена

Алгоритм Укконена–Майерса, используемый для вычисления расстояния между строками (раздел 9.5), в самом худшем случае выполняется за время порядка $O(dn)$, где d — фактическое расстояние между строками. Такой результат достигается путем уменьшения количества элементов массива $\boldsymbol{c}$, участвующих в вычислениях (учитываются только те элементы массива $\boldsymbol{c}$, которые располагаются на определенных диагоналях этого массива, а число таких диагоналей имеет порядок $O(d)$, подробнее см. раздел 9.5). Подобную стратегию использует алгоритм Укконена решения задачи k-разностей, причем время выполнения этого алгоритма в среднем составляет величину порядка $O(kn)$. Основой данного алгоритма является соотношение *в*) леммы 10.3.1, которое фактически показывает, что необходимые для вычислений диагональные элементы массива $\boldsymbol{c}$ (вычисленного на основе расстояния преобразования) образуют неубывающую последовательность (не по значениям, а по местоположению). Если обозначить через j' самую правую позицию в строке i массива $\boldsymbol{c}$, такую, что $\boldsymbol{c}[i, j'] \leqslant k$, тогда для всех элементов в строке $i+1$ в позициях $j = j'+2, j'+3, \ldots, m$ справедливо неравенство $\boldsymbol{c}[i+1, j] > k$. Поэтому $m - j' - 1$ самых правых элементов, стоящих в строке $i+1$ массива $\boldsymbol{c}$, можно исключить из вычислений, поскольку они не могут определять k-приближенное совпадение с паттерном $\boldsymbol{p}$. В общем случае получаем такое правило: во всех строках массива $\boldsymbol{c}$, номер которых превышает i, можно не рассматривать элементы, стоящие на диагоналях, располагающихся правее диагонали $\Delta_{i,j'}$. Эта простая стратегия составила основу следующего алгоритма.

Алгоритм 10.3.1. (Алгоритм Укконена)

▷ *Вычисление всех k-приближенных совпадений*
▷ *паттерна $\boldsymbol{p}[1..m]$ со строкой $\boldsymbol{x}[1..n]$*
инициализация элементов в нулевых строке и столбце массива $\boldsymbol{c}$ (см. табл. 10.4)
▷ *В строке 0 число k находится в позиции $j' = k$*

```
j' ← k
for i ← 1 to n do
    j'' ← 0        ▷ для случая, когда k = 0
for j ← 1 to min{j' + 1, m} do
    вычисление c[i, j] на основе формулы (10.4)
    if c[i, j] ⩽ k then j'' ← j
▷ Вывод результата
if j'' = m then output i
▷ Сохранение самой правой позиции j', такой, что c[i, j'] ⩽ k
j' ← j''
```

Очевидно, что временна́я сложность алгоритма Укконена пропорциональна сумме n значений переменной j', вычисленных во время выполнения алгоритма. Конечно, в самом худшем случае, когда все j' равны m (в этом случае в каждой строке $i \in 1..n\boldsymbol{c}[i,m] \leqslant k$), алгоритм Укконена выполняется за время порядка $\Theta(mn)$. Но даже если, например, условие $\boldsymbol{c}[i,m/2] \leqslant k$ выполняется для числа строк порядка $\Theta(n)$, то и в этом случае алгоритм выполняется за время порядка $\Theta(mn)$.

Чтобы определить временну́ю сложность алгоритма Укконена в среднем, необходимо оценить математическое ожидание $M[j']$ значения j' для каждой строки i массива $\boldsymbol{c}$, соответствующего префиксу $\boldsymbol{x}[1..i]$ строки $\boldsymbol{x}$. Математическое ожидание $M[j']$ можно оценить путем вычисления математического ожидания значения расстояния преобразования между префиксом $\boldsymbol{p}[1..j]$ паттерна $\boldsymbol{p}$ и случайно выбранной строки $\boldsymbol{x}$ длиной не более j. В работе [226] утверждается: "вполне очевидно", что $M[j'] \in O(k)$. Однако оказалось, что это утверждение "не очевидно" и оно не было доказано до 1992 года, когда появилась работа Чанга и Лэмпи (Chang, Lampe) [44] с доказательством этого факта.

Доказательство оценки $O(kn)$ временно́й сложности в среднем алгоритма Укконена мы начнем с доказательства двух лемм, первая из которых устанавливает одно простое свойство расстояния преобразования, а вторая лемма чисто техническая.

Лемма 10.3.2. Для всех целых $j \in 1..m$ и $i \in j..n$ справедливо неравенство

$$\boldsymbol{c}[i,j] \geqslant d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j])/2.$$

Доказательство. Пусть $d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) = d$. Напомним, что существует такое целое число $i' \in i-j+1..i$, что

$$\boldsymbol{c}[i,j] = d_E(\boldsymbol{x}[i'..i], \boldsymbol{p}[1..j]) = d' \leqslant d.$$

На основе неравенства треугольника (2.5) получаем

$$d - d' \leqslant d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{x}[i'..i]). \tag{10.13}$$

Очевидно следующее утверждение: расстояние преобразования между двумя строками длиной n_1 и n_2 (пусть для определенности $n_2 \geqslant n_1$) не меньше $n_2 - n_1$; минимум $n_2 - n_1$ этого расстояния достигается тогда, когда первая строка является суффиксом второй. На основании этого утверждения получаем неравенство

$$d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{x}[i'..i]) \leqslant d_E(\boldsymbol{x}[i'..i], \boldsymbol{p}[1..j]) = d'. \tag{10.14}$$

Комбинируя неравенства (10.13) и (10.14), получаем требуемый результат. ■

Лемма 10.3.3. Пусть $p_{j,r}$ — вероятность события, что две случайные строки длиной $j > 0$ имеют общую подпоследовательность длиной $\lceil rj \rceil$, $r < 1$. Тогда для произвольного значения j и значения r, удовлетворяющего неравенствам $7/8 \leqslant r \leqslant 1$, существуют положительные константы s и t ($t < 1$), такие, что $p_{r,j} < st^j/j$.

Доказательство см. в [44]. ■

Чтобы использовать результаты последних лемм, введем величину

$$j'' = 2k/(1-r),$$

где $7/8 \leqslant r \leqslant 1$, как того требует лемма 10.3.3. Отметим, что для всех целых $j \geqslant j''$ выполняется неравенство $j - 2k \geqslant rj$. Поскольку $j'' > 2$, математическое ожидание $M[j']$ ограничено сверху величиной

$$M[j'] < (j''-1) + \sum_{j \geqslant j''} \left(j \times P\{\boldsymbol{c}[i,j] \leqslant k\}\right), \tag{10.15}$$

где $P\{\boldsymbol{c}[i,j] \leqslant k\}$ — вероятность того, что $\boldsymbol{c}[i,j] \leqslant k$. В этом выражении условие $\boldsymbol{c}[i,j] \leqslant k$ имеет следствием (по лемме 10.3.2) неравенство

$$d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) \leqslant'' 2k.$$

Однако, как показано в упражнении 10.3.7,

$$d_E(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) \geqslant j - |\operatorname{LCS}(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j])|. \tag{10.16}$$

Поэтому из условия $\boldsymbol{c}[i,j] \leqslant k$ также вытекает, что

$$\operatorname{LCS}(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) \geqslant j - 2k,$$

где, если выполняется условие $j \geqslant j''$, $j - 2k \geqslant rj$. Следовательно, при выполнении неравенства $j \geqslant j''$ и на основании леммы 10.3.3 при соответствующем выборе констант s и t имеем

$$\begin{aligned} P\{\boldsymbol{c}[i,j] \leqslant k\} &\leqslant P\{\operatorname{LCS}(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) \geqslant rj\} \leqslant \\ &\leqslant P\{LCS(\boldsymbol{x}[i-j+1..i], \boldsymbol{p}[1..j]) \geqslant \lceil rj \rceil\} < st^j/j. \end{aligned} \tag{10.17}$$

Подставив (10.17) в выражение (10.15), получаем

$$M[j'] < (j'' - 1) + \sum_{j \geqslant j''} (jst^j/j) = (j'' - 1) + O(1) \in O(k).$$

Отсюда следует, что алгоритм Укконена выполняется в среднем за время порядка $O(kn)$. Также заметим, что, как и в алгоритме Вагнера–Фишера, для реализации алгоритма Укконена достаточно хранить в памяти только две строки массива $\boldsymbol{c}$. Таким образом, мы доказали следующую теорему.

Теорема 10.3.4. Алгоритм 10.3.1 решает задачу k-разностей для паттерна $\boldsymbol{p}[1..m]$ и текстовой строки $\boldsymbol{x}[1..n]$ в среднем за время порядка $O(kn)$ с использованием памяти объемом порядка $\Theta(m)$. ■

Приведенная верхняя граница времени выполнения в среднем алгоритма Укконена не кажется вполне удовлетворительной, однако на практике алгоритм выполняется достаточно быстро. В упражнениях 10.3.4 и 10.3.5 показано, как можно изменить этот алгоритм, чтобы он работал и с использованием расстояния Левенштейна.

Через три года после появления первоначального варианта алгоритма Укконена, он был усовершенствован на основе применения дерева суффиксов так, что его время выполнения в самом худшем случае имело порядок $O(k^2n)$ с использованием памяти объемом порядка $O(n)$ [149]. Позднее были предложены варианты этого алгоритма с временем выполнения в самом худшем случае порядка $O(kn)$ и с использованием памяти объемом порядка $O(m^2)$ [148, 98]. В 90-х годах проводилось дальнейшее совершенствование данного алгоритма [44, 222, 232], кульминацией этого процесса стал алгоритм Майерса [187], который мы рассмотрим в следующем разделе.

10.3.2 Алгоритм Майерса

В этом разделе представим алгоритм, который для достижения очень высокой эффективности (в самом худшем случае время вычислений не зависит от аппроксимационной константы k) использует бинарные векторы оригинальным способом, основываясь на особых свойствах расстояния преобразования (лемма 10.3.1). Как и в других алгоритмах, использующих бинарные векторы, будем предполагать, что алфавит индексирован (это требование обычно выполняется на практике), т.е. будем считать, что имеем алфавит $A = \{1, 2, \ldots, \alpha\}$.

Матрицы разностей

Сначала покажем, как в алгоритме Майерса массив стоимостей $\boldsymbol{c}$, вычисленный для расстояния преобразования, можно заменить матрицей разностей $\boldsymbol{\Delta}$, точнее, двумя матрицами разностей $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$. Начнем с определения векторов строк

$\boldsymbol{\Delta}_i = \boldsymbol{\Delta}_i[1..m]$ ***горизонтальных разностей*** ($i = 1, 2, \ldots, n$), где для каждого $j \in 1..m$

$$\boldsymbol{\Delta}_i[j] = \boldsymbol{c}[i, j] - \boldsymbol{c}[i, j-1], \tag{10.18}$$

и определения векторов строк $\Delta'_i = \Delta'_i[1..m]$ ***вертикальных разностей*** ($i = 1, 2, \ldots, n$), где для каждого $j \in 1..m$

$$\Delta'_i[j] = \boldsymbol{c}[i, j] - \boldsymbol{c}[i-1, j]. \tag{10.19}$$

Напомним (см. табл. 10.4), что нам известны значения элементов массива стоимостей, которые стоят в нулевых строке и столбце:

$$\boldsymbol{c}[0, j] = j \text{ для всех } j \in 0..m, \tag{10.20}$$

$$\boldsymbol{c}[i, 0] = 0 \text{ для всех } i \in 0..n. \tag{10.21}$$

Теперь на основании формул (10.18) и (10.21) мы можем вычислить все элементы ***матрицы разностей*** $\boldsymbol{\Delta} = \boldsymbol{\Delta}[1..n, 1..m]$:

$$\Delta[i, j] = \boldsymbol{\Delta}_i[j] \text{ для всех } i \in 1..n \text{ и } j \in 0..m. \tag{10.22}$$

Аналогично на основании формул (10.19) и (10.20) вычисляются элементы матрицы $\boldsymbol{\Delta}'$:

$$\boldsymbol{\Delta}'[i, j] = \boldsymbol{\Delta}'_i[j] \text{ для всех } i \in 1..n \text{ и } j \in 0..m. \tag{10.23}$$

Для удобства дальнейших вычислений расширим определение матриц $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$, включив в них нулевые строки и нулевые столбцы, положив при этом

$$\boldsymbol{\Delta}_0[j] = \boldsymbol{\Delta}'_0[j] = 0 \text{ для всех } j \in 0..m, \tag{10.24}$$

$$\boldsymbol{\Delta}_i[0] = \boldsymbol{\Delta}'_i[0] = 0 \text{ для всех } i \in 1..n. \tag{10.25}$$

Из приведенных формул видно, что векторы строк (10.18) и (10.19) можно вычислить последовательно $\boldsymbol{\Delta}_{i-1} \to \boldsymbol{\Delta}_i$ и $\boldsymbol{\Delta}'_{i-1} \to \boldsymbol{\Delta}'_i$ ($i = 1, 2, \ldots, n$). Поэтому таким же способом можно вычислить и все элементы (10.22) и (10.23) матриц $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$. Однако напомним, что наша цель заключается в том, чтобы для каждого значения i вычислить $\boldsymbol{c}[i, m]$ и проверить выполнение неравенства $\boldsymbol{c}[i, m] \leqslant k$. Для этого на основании формулы (10.19) можем записать

$$\boldsymbol{c}[i, m] = \boldsymbol{c}[0, m] + \sum_{i'=1}^{i} \boldsymbol{\Delta}'_{i'}[j] = m + \sum_{i'=1}^{i} \boldsymbol{\Delta}'_{i'}[j]. \tag{10.26}$$

Таким образом, вычисление массива $\boldsymbol{c}$ можно заменить вычислением матриц разностей $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$ (точнее, вычислением векторов строк $\boldsymbol{\Delta}_i$ и $\boldsymbol{\Delta}'_i$).

Преимуществом такого способа вычислений является то, что в силу леммы 10.3.1 нам надо знать значения только тех элементов матриц $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$, которые

стоят на диагоналях $\{-1, 0, 1\}$. Если мы найдем способ эффективного вычисления этих значений (а значит, и $\boldsymbol{c}[i, m]$), то тем самым полностью используем специфические свойства расстояния преобразования, формализованные в лемме 10.3.1, что, в свою очередь, приведет к уменьшению общего времени решения задачи k-разностей. Эта идея реализована в алгоритме Майерса.

Чтобы лучше понять отношения между массивом стоимостей $\boldsymbol{c}$ и соответствующими матрицами разностей $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$ в случае расстояния преобразования, предлагаем сравнить табл. 10.5 и 10.6, где они приведены. (Напомним, что массив $\boldsymbol{c}$ для этого примера в случае расстояния Левенштейна приведен в табл. 10.2.)

Таблица 10.5. Массив стоимостей $\boldsymbol{c}$ для $\boldsymbol{x} = rests$ и $\boldsymbol{p} = stress$ на основе расстояния преобразования

	$\boldsymbol{j}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**
i		ε	s	t	r	e	s	s
0	ε	0	1	2	3	4	5	6
1	r	0	1	2	2	3	4	5
2	e	0	1	2	3	2	3	4
3	s	0	0	1	2	3	2	3
4	t	0	1	0	1	2	3	3
5	s	0	0	1	1	2	2	3

Таблица 10.6. Матрицы разностей для $\boldsymbol{x} = rests$ и $\boldsymbol{p} = stress$

	$\boldsymbol{j}$	**1**	**2**	**3**	**4**	**5**	**6**
i		s	t	r	e	s	s
1	r	1	1	0	1	1	1
2	e	1	1	1	-1	1	1
3	s	0	1	1	1	-1	1
4	t	1	-1	1	1	1	0
5	s	0	1	0	1	0	1

Матрица $\boldsymbol{\Delta}$ (горизонтальные разности)

	$\boldsymbol{j}$	**1**	**2**	**3**	**4**	**5**	**6**
i		s	t	r	e	s	s
1	r	0	0	0	-1	-1	-1
2	e	0	0	0	1	-1	-1
3	s	0	-1	-1	-1	1	-1
4	t	0	1	-1	-1	-1	0
5	s	0	-1	1	0	0	0

Матрица $\boldsymbol{\Delta}'$ (вертикальные разности)

Двухшаговая рекурсия

Теперь получим формулы для вычисления векторов строк $\boldsymbol{\Delta}_i$ и Δ'_i. Сначала рассмотрим $\boldsymbol{\Delta}_i$. Применяя формулу (10.18) и базовую рекурсию (10.4), получаем

следующую формулу, справедливую для всех $i \in 1..n$ и $j \in 0..m$.

$$\begin{aligned}\boldsymbol{\Delta}_i[j] &= \boldsymbol{c}[i,j] - \boldsymbol{c}[i,j-1] = \\ &= \min\{\boldsymbol{c}[i-1,j]+1, \boldsymbol{c}[i,j-1]+1, \boldsymbol{c}[i-1,j-1]+d(\boldsymbol{x}[i],\boldsymbol{p}[j])\} - \\ &\quad - \boldsymbol{c}[i,j-1].\end{aligned}$$

Как и в алгоритмах ДБГ и ПДБГ, будем считать, что значение $d(\boldsymbol{x}[i],\boldsymbol{p}[j])$ можно получить как значение элемента, стоящего в позиции $[\boldsymbol{x}[i],j]$ бинарного массива $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$ (определение этого массива см. в (10.6)). Напомним, что массив $\boldsymbol{t}$ вычисляется на предварительном этапе алгоритма за время порядка $\Theta(\alpha\lceil m/w\rceil + m)$ (теорема 7.4.4). Далее для краткости записи будем использовать обозначение $\boldsymbol{t}_{ij} = \boldsymbol{t}[\boldsymbol{x}[i], j]$. На основании формул (10.18), (10.19), (10.24) и (10.25) нетрудно получить формулу для вычисления горизонтальных разностей

$$\boldsymbol{\Delta}_i[j] = \min\left\{\boldsymbol{\Delta}_{i-1}[j], \boldsymbol{\Delta}'_i[j-1], t_{ij}-1\right\} + \left(1 - \boldsymbol{\Delta}'_i[j-1]\right), \qquad (10.27)$$

справедливую для всех $i \in 1..n$ и $j \in 0..m$. Аналогично получаем формулу для вычисления вертикальных разностей:

$$\boldsymbol{\Delta}'_i[j] = \min\left\{\boldsymbol{\Delta}_{i-1}[j], \boldsymbol{\Delta}'_i[j-1], t_{ij}-1\right\} + \left(1 - \boldsymbol{\Delta}_{i-1}[j]\right), \qquad (10.28)$$

Таким образом, элементы $\boldsymbol{\Delta}[i,j]$ и $\boldsymbol{\Delta}'[i,j]$ вычисляются на основе только элементов $\boldsymbol{\Delta}[i-1,j]$ и $\boldsymbol{\Delta}'[i,j-1]$ и предварительно вычисленных значений $\boldsymbol{t}_{ij}$. Формулы (10.27) и (10.28) составляют двухшаговую рекурсию, которая является основой алгоритма Майерса.

Использование бинарных операций

Чтобы реализовать вычисления, определяемые формулами (10.27) и (10.28), с помощью бинарных операций надо сначала представить переменные, участвующие в этих формулах, в бинарном виде. Для этого введем четыре новых бинарных вектора, которые будут представлять положительные (P) и отрицательные (M) значения векторов $\boldsymbol{\Delta}_i$ и Δ'_i. В частности для векторов $\boldsymbol{\Delta}_i$ определим бинарные векторы $\boldsymbol{P}_i[1..m]$ и $\boldsymbol{M}_i[1..m]$ следующим образом ($j \in 1..m$):

$$\begin{aligned}\boldsymbol{P}_i[j] &= 1, \text{ если } \boldsymbol{\Delta}_i[j] = 1, \text{ и } \boldsymbol{P}_i[j] = 0 \text{ в противном случае;}\\ \boldsymbol{M}_i[j] &= 1, \text{ если } \boldsymbol{\Delta}_i[j] = -1, \text{ и } \boldsymbol{M}_i[j] = 0 \text{ в противном случае.}\end{aligned}$$

Аналогично для Δ'_i определяются бинарные векторы $P'_i[1..m]$ и $M'_i[1..m]$:

$$\begin{aligned}\boldsymbol{P}'_i[j] &= 1, \text{ если } \boldsymbol{\Delta}'_i[j] = 1, \text{ и } \boldsymbol{P}'_i[j] = 0 \text{ в противном случае;}\\ \boldsymbol{M}'_i[j] &= 1, \text{ если } \boldsymbol{\Delta}'_i[j] = -1, \text{ и } \boldsymbol{M}'_i[j] = 0 \text{ в противном случае.}\end{aligned}$$

Из этих определений следует, что $\boldsymbol{\Delta}_i[j] = 0$ только тогда, когда $\boldsymbol{P}_i[j] = \boldsymbol{M}_i[j] = 0$, и $\boldsymbol{\Delta}'_i = 0$ только в том случае, если $\boldsymbol{P}'_i[j] = \boldsymbol{M}'_i[j] = 0$. Поэтому мы имеем возможность во всех ситуациях заменить вычисление векторов $\boldsymbol{\Delta}_i$ и $\boldsymbol{\Delta}'_i$ вычислением векторов $\boldsymbol{P}_i$, $\boldsymbol{M}_i$, $\boldsymbol{P}'_i$ и $\boldsymbol{M}'_i$.

Рассмотрим процесс вычисления векторов горизонтальных разностей в соответствии с формулами (10.27). Покажем, что эти вычисления можно заменить вычислениями с использованием только бинарных операций, выполняемых над отдельными битами. Для этого можно применить следующую последовательность операций.

$$\begin{aligned} \boldsymbol{K}[j] &\leftarrow \overline{t_{ij}} \vee \boldsymbol{M}_{i-1}[j], \\ \boldsymbol{P}_i[j] &\leftarrow \boldsymbol{M}'_i[j-1] \vee \overline{(\boldsymbol{K}[j] \vee \boldsymbol{P}'_i[j-1])}, \\ \boldsymbol{M}_i[j] &\leftarrow \boldsymbol{P}'_i[j-1] \wedge \boldsymbol{K}[j], \end{aligned} \tag{10.29}$$

где

- $\boldsymbol{K} = \boldsymbol{K}[1..m]$ — бинарный вектор, введенный для удобства вычислений,
- знак $\vee$ обозначает бинарную операцию логического сложения OR (результат этой операции равен 0 только в том случае, если *оба* операнда равны 0),
- знак $\wedge$ обозначает бинарную операцию логического умножения AND (результат этой операции равен 0 только в том случае, если *хотя бы один* из операндов равен 0),
- обозначение $\overline{<\text{выражение}>}$ указывает на операцию логического отрицания, примененную к логическому выражению <выражение>.

Чтобы удостовериться в правильности последних формул, сначала заметим, что в формуле (10.27) участвует всего три переменные: две из них ($\boldsymbol{\Delta}_{i-1}[j]$ и $\boldsymbol{\Delta}'_i[j-1]$) могут принимать только значения $\{-1, 0, 1\}$, а третья (t_{ij}) является по определению бинарной. Поэтому всего может быть $3 \times 3 \times 2 = 18$ комбинаций значений переменных, для которых следует выполнить вычисления по формуле (10.27). Рассмотрим одну из возможных комбинаций переменных, например такую: $\boldsymbol{\Delta}_{i-1}[j] = 1$, $\boldsymbol{\Delta}'_i[j-1] = -1$, $t_{ij} = 1$. Для использования в формулах (10.29) вычисляем значения $\boldsymbol{M}_{i-1}[j] = 0$, $\boldsymbol{M}'_i[j-1] = 1$, $\boldsymbol{P}'_i[j-1] = 0$ и $\overline{t_{ij}} = 0$. Теперь, применяя формулу (10.27), находим, что $\boldsymbol{\Delta}_i[j] = 1$, а на основании формул (10.29) имеем $\boldsymbol{K}[j] = 0$, $\boldsymbol{P}_i[j] = 1$ и $\boldsymbol{M}_i[j] = 1$. Очевидно, результаты вычисления по формулам (10.27) и (10.29) совпадают. Для доказательства правильности формул (10.29) осталось проверить еще 17 комбинаций значений переменных, что мы оставим для упражнения 10.3.8.

Вследствие определенной симметрии горизонтальных и вертикальных разностей, нетрудно получить аналог формул (10.29) для векторов вертикальных

разностей (10.28):

$$\begin{aligned} \boldsymbol{K}'[j] &\leftarrow \overline{t_{ij}} \vee \boldsymbol{M}'_i[j-1], \\ \boldsymbol{P}'_i[j] &\leftarrow \boldsymbol{M}_{i-1}[j] \vee \overline{(\boldsymbol{K}'[j] \vee \boldsymbol{P}_{i-1}[j])}, \\ \boldsymbol{M}'_i[j] &\leftarrow \boldsymbol{P}_{i-1}[j] \wedge \boldsymbol{K}'[j], \end{aligned} \tag{10.30}$$

где $\boldsymbol{K}' = \boldsymbol{K}'[1..m]$ — бинарный вектор, введенный для удобства вычислений.

Логика "клеточных" вычислений

Если значения $\boldsymbol{K}[j]$ вычислены, то на формулы (10.29) можно смотреть как на формулы вычисления горизонтальных разностей, записанных в ячейке ("клетке") $[i, j]$ массива $\boldsymbol{c}$, использующие значения вертикальных разностей, записанных в ячейке $[i, j-1]$ того же массива $\boldsymbol{c}$. Аналогично, если известны значения $\boldsymbol{K}'[j]$, тогда формулы (10.30) можно применить для вычисления вертикальных разностей, записанных в ячейке $[i, j]$ массива $\boldsymbol{c}$, используя при этом только значения горизонтальных разностей, записанных в ячейке $[i-1, j]$. Эта зависимость между горизонтальными и вертикальными разностями показана графически на рис. 10.1, где в объединенной таблице представлены элементы как матрицы горизонтальных разностей $\boldsymbol{\Delta}$ (эти элементы обозначены буквой h), так и матрицы вертикальных разностей $\boldsymbol{\Delta}'$ (эти элементы обозначены буквой v).

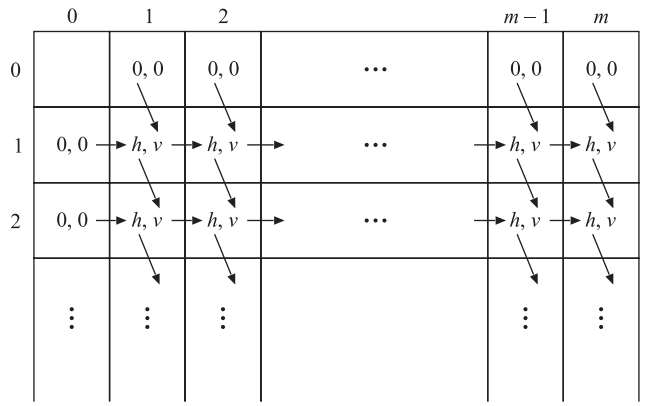


Рис. 10.1. Схема вычисления горизонтальных и вертикальных разностей

Оставив на время в стороне вопрос о вычислении векторов $\boldsymbol{K}$ и $\boldsymbol{K}'$, здесь отметим, что теперь можно вычислить все элементы матриц $\boldsymbol{\Delta}$ и $\boldsymbol{\Delta}'$ строка за строкой, используя в качестве начальных значений нулевые значения из нулевых строки и столбца (см. формулы (10.24) и (10.25)). Так, на i-м шаге вычислений

- на основе значений $\boldsymbol{\Delta}_{i-1}[j]$ вычисляются значения $\boldsymbol{\Delta}'_i[j]$ (т.е. на основе горизонтальных разностей вычисляются вертикальные),
- на основе значений $\boldsymbol{\Delta}'_i[j-1]$ вычисляются значения $\boldsymbol{\Delta}_i[j]$ (здесь на основе вертикальных разностей вычисляются горизонтальные).

Если ввести переменную *cost* (стоимость) с начальным значением $cost \leftarrow m$, то одновременно с описанными выше вычислениями на i-м шаге можно вычислить значение $\boldsymbol{c}[i,m]$:

$$cost \leftarrow cost + \boldsymbol{\Delta}'_{i-1}[m]. \tag{10.31}$$

Точно так же, как и в алгоритмах ДБГ и ПДБГ, в алгоритме Майерса в повышении эффективности вычислений определяющую роль играет тот факт, что бинарные операции над бинарными векторами можно выполнять совместно, а не по отдельности. Стандартные операции над машинными словами могут выполняться за константное время с помощью одной машинной команды сразу применительно ко всем w бит, составляющим машинное слово (обычно w равно 32 или 64). Так, используя стандартную операцию *rightshift* (сдвиг вправо), можно получить доступ сразу ко всем нужным $j-1$ элементам бинарного вектора без непосредственного указания адресов этих элементов и не используя ссылку на j-й элемент. Более того, если $m > w$, то в этом случае бинарный вектор будет храниться в $\lceil m/w \rceil$ смежных машинных словах, и стандартные бинарные операции над ними можно реализовать как операцию над одним словом.

После такого "лирического" отступления, попытаемся записать с помощью стандартных бинарных операций вычисления, задаваемые формулами (10.29)–(10.31). (Далее будем использовать обозначение $\boldsymbol{t}_i = \boldsymbol{t}[\boldsymbol{x}[i], 1..m]$.)

$$\begin{aligned}
&1.\quad \boldsymbol{K}' \leftarrow \overline{t_i} \vee \mathit{rightshift}(\boldsymbol{M}'_i, 1).\\
&2.\quad \boldsymbol{P}'_i \leftarrow \boldsymbol{M}_{i-1} \vee \overline{(\boldsymbol{K}' \vee \boldsymbol{P}_{i-1})}.\\
&3.\quad \boldsymbol{M}'_i \leftarrow \boldsymbol{P}_{i-1} \wedge \boldsymbol{K}'.\\
&4.\quad cost \leftarrow cost + \boldsymbol{P}'_i[m] - \boldsymbol{M}'_i[m].\\
&5.\quad \boldsymbol{K} \leftarrow \overline{t_i} \vee \boldsymbol{M}_{i-1}.\\
&6.\quad \boldsymbol{P}_i \leftarrow \mathit{rightshift}(\boldsymbol{M}'_i, 1) \vee \overline{(\boldsymbol{K} \vee \mathit{rightshift}(\boldsymbol{P}'_i, 1))}.\\
&7.\quad \boldsymbol{M}_i \leftarrow \mathit{rightshift}(\boldsymbol{P}'_i, 1) \wedge \boldsymbol{K}.
\end{aligned} \tag{10.32}$$

Второй и третий операторы из набора (10.32) вычисляют значения $\boldsymbol{P}'$ и $\boldsymbol{M}'$ на основании значений $\boldsymbol{K}'$ (вычисленных с помощью первого оператора) и предыдущих значений $\boldsymbol{P}$ и $\boldsymbol{M}$. Это позволяет четвертому оператору вычислить значение *cost*. Шестой и седьмой операторы вычисляют текущие значения $\boldsymbol{P}$ и $\boldsymbol{M}$ на основании значений $\boldsymbol{K}$ (вычисленных с помощью пятого оператора) и сдвигов текущих значений $\boldsymbol{P}'$ и $\boldsymbol{M}'$. Очевидно, что для реализации этих вычислений необходимо хранить в памяти только переменные $\boldsymbol{K}$, $\boldsymbol{P}$, $\boldsymbol{M}$, $\boldsymbol{K}'$, $\boldsymbol{P}'$ и $\boldsymbol{M}'$, каждая из которых

является бинарным вектором длиной m, при этом операторы 1–3 и 5–7 выполняются за время порядка $\Theta(\lceil m/w \rceil)$, а 4-й оператор — за константное время. Таким образом, для реализации этих вычислений необходима память объемом порядка $\Theta(\lceil m/w \rceil)$ машинных слов, а общее время выполнения всех n шагов вычислений составит величину порядка $\Theta(\lceil m/w \rceil n)$. (Мы предполагали, что оператор *rightshift* после сдвига ставит в позицию $j = 1$ значение 0.)

К сожалению, набор операторов (10.32) работать не будет! Проблема заключается в том, что вычисление значений $\boldsymbol{K}'$ в первом операторе зависит от вектора M'_i, который вычисляется только в третьем операторе. Однако в работе [187] с помощью достаточно сложных выкладок показано, что $\boldsymbol{K}'$ можно вычислить по формуле

$$\boldsymbol{K}' \leftarrow \overline{t_i} \vee \left(\left((\overline{t_i} \wedge \boldsymbol{P}_{i-1}) + P_{i-1} \right) \hat{} \, \boldsymbol{P}_{i-1} \right), \tag{10.33}$$

где символ $\hat{}$ обозначает логическую операцию исключающего ИЛИ (эта операция возвращает значение 0 только в том случае, если операнды имеют одинаковые значения). Мы не будем приводить доказательство формулы (10.33), однако заменим этой формулой 1-й оператор в наборе (10.32).

Мы полностью описали алгоритм Майерса, осталось его записать в явном виде — оставим это для упражнения 10.3.10. Сформулируем основной результат данного подраздела.

Теорема 10.3.5. Алгоритм Майерса, основанный на формулах (10.32) и (10.33), корректно вычисляет все k-приближенные совпадения паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ на основе расстояния преобразования за время порядка $\Theta(\lceil m/w \rceil n)$ с использованием памяти объемом порядка $\Theta(\lceil m/w \rceil)$. ■

Как отмечалось в разделе 7.4, мы можем положить, что $w \in \Omega(\log n)$. Поэтому верхние границы теоремы 10.3.5 можно записать в форме $O(kn/\log n)$ и $O(mn/\log n)$ соответственно.

Усовершенствование алгоритма Майерса

В предыдущем разделе мы рассмотрели алгоритм Укконена; ожидаемое время вычисления достигает величины порядка $O(kn)$ благодаря тому, что диагональные элементы в массиве $\boldsymbol{c}$ составляют неубывающую последовательность (лемма 10.3.1). Здесь существенно, что для каждой строки i массива $\boldsymbol{c}$ известно значение j', такое, что $\boldsymbol{c}[i, j'] \leqslant k$. Это ограничивает вычисляемые элементы в строке $i+1$ столбцом $j'+1$. Поскольку математическое ожидание числа j' имеет порядок $O(k)$, то и математическое ожидание времени обработки каждой строки массива $\boldsymbol{c}$ также будет иметь такой же порядок.

В работе [187] показано, что такую же стратегию можно применить и в алгоритме Майерса, ограничивая величиной порядка $O(k)$ ожидаемое число битов,

которые будут обрабатываться в каждой строке матриц разностей. Далее такой усовершенствованный алгоритм Майерса для краткости будем называть алгоритмом MM (модифицированный Майерса). Для этого алгоритма имеет место следующая теорема.

Теорема 10.3.6. Алгоритм MM вычисляет все k-приближенные совпадения паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ на основе расстояния преобразования в среднем за время порядка $\Theta(\lceil k/w \rceil n)$ и в самом худшем случае за время порядка $\Theta(\lceil m/w \rceil n)$. ■

Эффективность алгоритмов решения задачи k-разностей

В отличие от алгоритмов непосредственного использования массива стоимостей $\boldsymbol{c}$ (такой алгоритм описан в разделе 10.1), самый ранний нетривиальный алгоритм решения задачи k-разностей был предложен Ландау и Вишкиным в 1986 году [150]. Этот алгоритм неоднократно усовершенствовался, в особенности Чангом и Лоулером (Chang, Lawler) [45, 46]. В результате появилось целое семейство алгоритмов, которые получили название "фильтрационные". Время выполнения этих алгоритмов в самом худшем случае составляет $O(kn+m)$ и не меньше в среднем величины порядка $O((kn\log m)/m)$ для значений $k < m/(\log m + O(1))$. Как видно, эти алгоритмы могут конкурировать с алгоритмом MM.

Другой подход, основанный на специальном определении расстояния между строками [81, 224], был разработан Укконеном [224]. Этот подход позволяет разрабатывать эффективные алгоритмы k-приближенного сравнения с паттерном. Здесь сначала вычисляется множество подстрок строки $\boldsymbol{x}$, которое гарантированно содержит все решения задачи k-разностей, а затем используется эффективный алгоритм для удаления из этого множества тех подстрок, которые не содержат решения данной задачи.

В работе [187] приведены результаты экспериментального сравнения алгоритма MM с другими современными алгоритмами решения задачи k-разностей. В частности, было проведено сравнение с алгоритмом [44], основанным на подходе Укконена, и с алгоритмами [232, 26], основанными на использовании бинарных векторов, методика работы с которыми получила дальнейшее развитие в работах [24, 233]. (Интересный алгоритм [26] будет описан в следующей главе (подраздел 11.2.4) как основной алгоритм для приближенного сравнения с множественными паттернами.) На очень широком наборе тестовых данных алгоритм MM показал более быструю работу по сравнению с другими алгоритмами, что подтвердило на практике теоретические утверждения теоремы 10.3.6.

Таким образом, на сегодняшний день алгоритм MM является лучшим выбором для приближенного сравнения с паттерном, если используется функция расстояния преобразования. В следующем разделе рассмотрим еще один алгоритм, который не является таким быстрым на практике и в теории, как алгоритм

ММ для расстояния преобразования, однако имеет значительно более широкую область применения.

Упражнения 10.3

1. Докажите утверждение c) леммы 10.3.1.
2. Докажите корректность алгоритма Укконена.
3. Охарактеризуйте ситуации, в которых алгоритм Укконена требует времени выполнения порядка $\Theta(mn)$.
4. Сформулируйте и докажите аналог леммы 10.3.1 для расстояния Левенштейна.
5. На основе предыдущего упражнения разработайте аналог алгоритма Укконена для расстояния Левенштейна.
6. Справедлива ли лемма 10.3.2 для других типов функций расстояния? Расширьте рамки этой леммы и докажите ее расширенный вариант, явно сформулировав предположения, на которых она будет основана.
7. Докажите неравенство (10.16).
8. Докажите корректность формул (10.27) и (10.28).
9. Проверьте эквивалентность формул (10.27) и (10.29).
10. Запишите в явном виде алгоритм Майерса.
11. Обсудите проблемы, которые могут возникнуть в алгоритме Майерса, если вместо расстояния преобразования использовать расстояние Левенштейна.

10.4 Быстрый и гибкий алгоритм Ву и Менбера

В разделе 10.2 описано обобщение алгоритма ДБГ (см. раздел 7.4), которое эффективно решает задачу k-несовпадений. Здесь мы рассмотрим другой алгоритм (технически более сложный), который расширяет подход алгоритма ДБГ, что обеспечивает эффективное решение задач приближенного сравнения с паттерном для широкого круга различных функций расстояния. Этот алгоритм решает не только задачу k-разностей, описанную в предыдущем разделе, но и задачу сравнения с регулярными выражениями (раздел 2.2). Описываемый в этом разделе алгоритм предложен Ву и Менбером (Wu, Manber) [233]; будем для краткости называть его алгоритмом ВМ. Алгоритм ВМ для большинства типов расстояний в самом худшем случае выполняется за время порядка $O(k\lceil m/w\rceil n)$, где w — длина машинного слова, а k ($k \geqslant 1$) — константа (граница) аппроксимации [233]. Хотя этот алгоритм не такой быстрый, как алгоритм ММ, он обладает большей гибкостью.

Как и большинство алгоритмов приближенного сравнения с паттерном, алгоритм BM использует соотношения рекурсии (10.4) для неотрицательной матрицы стоимостей $\boldsymbol{c} = \boldsymbol{c}[0..n, 1..m]$ (раздел 9.1), где для всех $i \in 1..n$ и $j \in 2..m$

$$\begin{aligned}\boldsymbol{c}[i,j] \leftarrow \min\{&\boldsymbol{c}[i-1,j] + d(\boldsymbol{x}[i], \varepsilon),\\ &\boldsymbol{c}[i,j-1] + d(\varepsilon, \boldsymbol{p}[j]),\\ &\boldsymbol{c}[i-1,j-1] + d(\boldsymbol{x}[i], \boldsymbol{p}[j])\}.\end{aligned}$$

Эта формула показывает, что $\boldsymbol{c}[i,j]$ вычисляется как минимум сумм трех значений соседних элементов массива $\boldsymbol{c}$ и стоимостей операций соответственно удаления (буквы $\boldsymbol{x}[i]$), вставки (буквы $\boldsymbol{p}[j]$) и постановки (буквы $\boldsymbol{p}[j]$ вместо буквы $\boldsymbol{x}[i]$). В разделе 9.1 мы видели, что алгоритм Винера–Фишера использует эту формулу для непосредственного вычисления массива $\boldsymbol{c}$. Однако в алгоритме BM, как и в алгоритме ДБГ, для более эффективного вычисления данного массива используются бинарные векторы.

Будем использовать бинарный вектор $\boldsymbol{s} = \boldsymbol{s}[1..m]$ и бинарный массив $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$, определенные в разделе 7.4. Напомним, что для заданного значения i и для всех $j \in 1..m\boldsymbol{s}[j] = 0$ только тогда, когда $\boldsymbol{x}[i-j+1..i] = \boldsymbol{p}[1..j]$, тогда как $\boldsymbol{t}[h,j] = 0$ только в том случае, если $\boldsymbol{p}[j] = h$, где h — буква алфавита. Конечно, здесь, как и в алгоритме ДБГ, на алфавит накладываются ограничения: алфавит A должен быть конечен, известен заранее и индексирован (см. раздел 4.1). В этом случае без уменьшения общности его можно представить в форме $\{1, 2, \ldots, \alpha\}$.

В алгоритме BM вместо одного бинарного вектора $\boldsymbol{s}$ используется $k+1$ подобных бинарных векторов, которые для $q \in 0..k$ определяются следующим образом:

$$\boldsymbol{s}_q[j] = 0, \text{ если } \boldsymbol{c}[i,j] \leqslant q, \text{ и } \boldsymbol{s}_q[j] = 1 \text{ в противном случае.} \tag{10.34}$$

Отметим, что $\boldsymbol{s}_0 = \boldsymbol{s}$, где $\boldsymbol{s}$ — определенный выше "стандартный" бинарный вектор. Также заметим, что для фиксированного i и любого целого $q \in 0..k-1$ выполняется импликация

$$\boldsymbol{s}_q[j] = 0 \Rightarrow \boldsymbol{s}_{q+1}[j] = 0. \tag{10.35}$$

Для сокращения записи введем обозначения $I = d(\boldsymbol{x}[i], \varepsilon)$, $D = d(\varepsilon, \boldsymbol{p}[j])$ и $S = d(\boldsymbol{x}[i], \boldsymbol{p}[j])$. Таким образом, I, D и S — неотрицательные целые числа, причем $S = 0$ только в том случае, если $\boldsymbol{t}[\boldsymbol{x}[i], j] = 0$. В этих обозначениях формулу (10.4) можно переписать так:

$$\boldsymbol{c}[i,j] \leftarrow \min\{\boldsymbol{c}[i-1,j] + I, \boldsymbol{c}[i,j-1] + D, \boldsymbol{c}[i-1,j-1] + S\}. \tag{10.36}$$

Также обозначим через $\boldsymbol{s}_q^{(i)}$ $(q = 0, 1, \ldots, k)$ $k+1$ бинарные векторы, соответствующие строкам i массива $\boldsymbol{c}$. Фактически, что необходимо сделать в алгоритме BM,

так это для каждого i ($i = 1, 2, \ldots, n$) вычислить все векторы $\boldsymbol{s}_q^{(i)}$ ($q = 0, 1, \ldots, k$) на основе вычисленных на предыдущем шаге векторов $\boldsymbol{s}_q^{(i-1)}$. Как и в алгоритме ДБГ, здесь вычисления производятся *на месте*, поэтому требуется хранить в памяти только $k+1$ бинарных векторов, а не все $(k+1)n$ векторов.

Рассмотрим значение $\boldsymbol{s}_q^{(i)}[j]$ для некоторого $q \in 0..k$. Мы хотим, чтобы $\boldsymbol{s}_q^{(i)}[j]$ равнялось 0, если выполняется неравенство $\boldsymbol{c}[i,j] \leqslant q$, а в противном случае $\boldsymbol{s}_q^{(i)}[j]$ должно равняться 1. Из формулы (10.36) следует, что $\boldsymbol{s}_q^{(i)}[j] = 0$ только в том случае, если выполняется одно из следующих условий.

1. $\boldsymbol{c}[i-1,j] \leqslant q - I \Leftrightarrow +\boldsymbol{s}_{q-I}^{(i-1)}[j] = 0$.

2. $\boldsymbol{c}[i,j-1] \leqslant q - D \Leftrightarrow +\boldsymbol{s}_{q-D}^{(i)}[j-1] = 0$.

3. $\boldsymbol{c}[i-1,j-1] \leqslant q - S \Leftrightarrow +\boldsymbol{s}_{q-S}^{(i-1)}[j-1] = 0$.

Отметим, что третье условие выполняется только тогда, когда $S = 0$ (т.е. когда $\boldsymbol{t}[\boldsymbol{x}[i], j] = 0$). Учитывая данные условия, значение $\boldsymbol{s}_q^{(i)}[j]$ можно вычислить по формуле

$$\begin{aligned}\boldsymbol{s}_q^{(i)}[j] \leftarrow \boldsymbol{s}_{q-I}^{(i-1)}[j] \wedge \boldsymbol{s}_{q-D}^{(i)}[j-1] \wedge \boldsymbol{s}_{q-S}^{(i-1)}[j-1] \\ \wedge \left(\boldsymbol{s}_q^{(i-1)}[j-1] \vee t[\boldsymbol{x}[i], j]\right),\end{aligned} \tag{10.37}$$

где символ $\wedge$ обозначает операцию логического умножения AND, а символ $\vee$ — операцию логического сложения OR.

Напомним (см. раздел 7.4), что бинарный вектор $\boldsymbol{s}^{(i)}$ формируется путем сдвига вправо вектора $\boldsymbol{s}^{(i-1)}$, т.е. для всех $j \in 2..m$ $\boldsymbol{s}^{(i)}[j] = \boldsymbol{s}^{(i-1)}[j-1]$, а в самую левую позицию нового вектора помещается 0 (поэтому всегда $\boldsymbol{s}^{(i)}[1] = 0$). Используя форму записи из раздела 7.4 и учитывая, что для всех значений $q\boldsymbol{s}_q^{(0)}[0..m] = 01^m$, формулу (10.37) можно записать следующим образом ($i = 1, 2, \ldots, n$).

$$\begin{aligned}\boldsymbol{s}_q^{(i)} \leftarrow \boldsymbol{s}_{q-I}^{(i-1)} \wedge rightshift(s_{q-D}^{(i)}, 1) \\ \wedge \mathit{rightshift}(\boldsymbol{s}_{q-S}^{(i-1)}, 1) \wedge \left(\mathit{rightshift}\left(\boldsymbol{s}_q^{(i-1)}, 1\right) \vee \boldsymbol{t}[x[i], 1..m]\right).\end{aligned} \tag{10.38}$$

После каждого вычисления формулы (10.38) необходимо проверить текущее значение $\boldsymbol{s}_k[m] = \boldsymbol{s}_k^{(i)}[m]$, чтобы определить (в соответствии с определением (10.34)), выполняется ли неравенство $\boldsymbol{c}[i,m] \leqslant k$, т.е. достигается ли k-приближенное совпадение паттерна $\boldsymbol{p}$ с подстрокой $\boldsymbol{x}[1..i]$. На основе формулы (10.38) и строится алгоритм BM, являясь полным аналогом алгоритма ДБГ. Явную запись алгоритма BM оставим для (простого) упражнения 10.4.1.

Отметим, что вычисления по формуле (10.38) зависят от фиксированных целочисленных значений I, D и S, которые не превосходят значения q. Если $I = D = S = 1$, то формула (10.38) вычисляется для расстояния преобразования.

Если же $I = D = 1$ и $S = 2$, то формула (10.38) работает для расстояния Левенштейна. В табл. 10.7 приведены массив $\boldsymbol{t}$ и векторы $\boldsymbol{s}_0^{(i)}$ и $\boldsymbol{s}_1^{(i)}$, вычисленные для расстояния Левенштейна для задачи 1-приближенного сравнения паттерна $\boldsymbol{p} = str$ и строки $\boldsymbol{x} = rests$ (массив стоимостей для этого примера приведен в табл. 10.2). Чтобы формулу (10.38) использовать для других функций расстояния, надо просто в соответствии с этой функцией изменить значения констант I, D и S.

Таблица 10.7. Приближенное сравнение с паттерном на основе расстояния Левенштейна: $A = \{e, r, s, t\}$, $\boldsymbol{p} = str$, $\boldsymbol{x} = rests$, $k = 1$

	$\boldsymbol{s}$	$\boldsymbol{t}$	$\boldsymbol{r}$
e	1	1	1
r	1	1	0
s	0	1	1
t	1	0	1

Массив $\boldsymbol{t}$

	$\boldsymbol{s}_0$	$\boldsymbol{s}_1$
r	111	111
e	111	111
s	011	001
t	001	000
s	011	001

Бинарные векторы

Для того чтобы найти все k-приближенные совпадения паттерна и заданной строки, в алгоритме BM вычисления по формуле (10.38) должны выполняться $(k + 1)n$ раз. Каждое вычисление по этой формуле требует выполнения трех сдвигов вправо, трех операций AND и одной операции OR над каждым машинным словом, составляющих каждый вектор $\boldsymbol{s}_q$ ($q = 1, 2, \ldots, k$). Эти операции над одним машинным словом выполняются за константное время. Если, как и ранее, через w обозначить количество битов в машинном слове и принять допущения о предварительном вычислении и хранении массива $\boldsymbol{t}$ (теорема 7.4.4), то мы можем сформулировать аналог теоремы 7.4.3.

Теорема 10.4.1. Алгоритм BM вычисляет все k-приближенные совпадения паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ за время порядка $\Theta(k\lceil m/w\rceil n)$ с использованием памяти объемом порядка $\Theta((\alpha + k)\lceil m/w\rceil)$. ■

Как и для всех алгоритмов, использующих методологию алгоритма ДБГ, в случае малых значений m (и даже в случае только ограниченных значений m) для получения асимптотической оценки сложности алгоритма выражение $\lceil m/w\rceil$ можно включить в константу пропорциональности, что для алгоритма BM дает асимптотическую оценку временно́й сложности в виде $\Theta(kn)$. (Дальнейшие исследования в этом направлении оставим для упражнения 10.4.4.) Напомним также замечание, сделанное в конце раздела 7.4, в соответствии с которым можно положить $w \in \in\Omega(\log n)$. Тогда для алгоритма BM получаем временну́ю границу $O(kmn/\log n)$.

Итак, в этом разделе описан эффективный и очень гибкий алгоритм для решения задачи приближенного сравнения с паттерном. Этот алгоритм легко адаптируется ко всем функциям расстояния, для которых стоимости операций вставки, удаления и подстановки фиксированы и выражаются целыми числами. С этой точки зрения алгоритм BM имеет бóльшую область применимости (более гибкий), чем алгоритмы решения задач k-несовпадений и k-разностей, описанные в разделах 10.2 и 10.3. Однако этот алгоритм имеет свои ограничения: его нельзя использовать, если стоимость операций редактирования переменная или принимает нецелые значения, как это бывает в случае весовых матриц (см. раздел 2.2). Но, как будет показано далее, алгоритм BM относительно просто и без потери эффективности можно модифицировать для решения практически любых задач сравнения с паттерном. Отметим, что такой расширенный алгоритм BM реализован в виде пакета *agrep* на языке Unix; его можно найти в Internet по адресу `http://webglimpse.net`.

Сравнение множеств букв

Во многих приложениях задачи сравнения с паттерном возникает ситуация, когда совпадение с паттерном считается достигнутым в некоторой позиции j, если в этой позиции стоит буква из некоторого заранее определенного множества букв. Например, в строке можно искать вхождение подстроки "ref", которая предшествует натуральному числу из интервала от 1 до 7, либо можно искать вхождение подстроки "ion", которой предшествует одна из букв "c", "s" или "t". В этих ситуациях можно определить паттерны, подобные $\boldsymbol{p} = ref[1-7]$ или $\boldsymbol{p} = [c,s,t]ion$, где множество букв для сравнения идентифицируется метасимволами в виде квадратных скобок []. Алгоритм BM может работать с такими паттернами, для чего необходимо внести только небольшие изменения на предварительном этапе вычисления массива $\boldsymbol{t}$. Все, что необходимо сделать для корректной работы алгоритма BM, так это положить

$$\boldsymbol{t}[1,4] \leftarrow \boldsymbol{t}[2,4] \leftarrow \ldots \leftarrow \boldsymbol{t}[7,4] \leftarrow 0$$

для паттерна $\boldsymbol{p} = ref[1-7]$, что позволит распознать цифры 1–7 в 4-й позиции паттерна, или

$$\boldsymbol{t}[c,1] \leftarrow \boldsymbol{t}[s,1] \leftarrow \boldsymbol{t}[t,1] \leftarrow 0$$

для паттерна $\boldsymbol{p} = [c,s,t]ion$, что позволит распознать одну из букв c, s, t в 1-й позиции паттерна. Никаких других изменений в алгоритме BM делать не требуется.

Сравнение с символами замещения

Напомним (см. раздел 2.2), что существует два общепринятых символа замещения:

символ • замещает одну любую букву из алфавита A;

символ $*$ замещает любую строку из множества A^*.

Использование символа $\bullet$ — это случай использования специального множества букв для сравнения, который рассмотрен в предыдущем подразделе и который требует только небольших изменений в вычислении массива $\boldsymbol{t}$ на предварительном этапе алгоритма BM.

Использование символа замещения $*$ для строк уже требует внесения некоторых изменений в сам алгоритм BM. Рассмотрим паттерн вида

$$\boldsymbol{p} = \boldsymbol{p}[1..j_1] * \boldsymbol{p}[j_1+1..j_2] * \cdots * \boldsymbol{p}[j_{r-1}+1..j_r] * \boldsymbol{p}[j_r+1..m],$$

где метасимвол $*$ вставлен в паттерн после позиций j_1, j_2, ..., j_r, но этот метасимвол не рассматривается как буква паттерна $\boldsymbol{p}$. Поэтому, чтобы указать местоположение метасимволов $*$, вводится новый бинарный вектор $\boldsymbol{t}^* = \boldsymbol{t}^*[1..m]$, в котором $\boldsymbol{t}^*[j] = 0$ только в том случае, если $j \in \{j_1, j_2, \ldots, j_r\}$.

Если для некоторых $h \in 1..r$, $i \in 1..n$ и $q \in 0..k$ выполняется $\boldsymbol{s}_q^{(i-1)}[j_h] = 0$, то в этом случае имеет место q-приближенное совпадение $\boldsymbol{p}[1..j_h]$ (включая все вхождения $*$ слева от позиции j_h) с подстрокой $\boldsymbol{x}[i'..j-1]$ строки $\boldsymbol{x}$. Поскольку после $\boldsymbol{p}[1..j_h]$ следует символ $*$, *каждая* подстрока $\boldsymbol{x}[i'..i'']$, где $i'' > i-1$, также имеет q-приближенное совпадение с $\boldsymbol{p}[1..j_h]$. Другими словами, если для некоторой позиции $i-1$ строки $\boldsymbol{x}$ выполняется равенство $s_q^{(i-1)}[j_h] = 0$, то для всех позиций $i'' > i-1$ должно выполняться аналогичное равенство $\boldsymbol{s}_q^{(i'')}[j_h] = 0$. Чтобы добиться такого выполнения, надо после каждого вычисления по формуле (10.38) сделать присвоение

$$\boldsymbol{s}_q^{(i)} \leftarrow \boldsymbol{s}_q^{(i)} \wedge (\boldsymbol{s}_q^{(i-1)} \vee \boldsymbol{t}^*). \tag{10.39}$$

В этом случае, если в результате вычисления формулы (10.38) получили $\boldsymbol{s}_q^{(i)} \leftarrow 1$, то присвоение $\boldsymbol{s}_q^{(i)}[j] \leftarrow 0$ будет иметь место только тогда, когда $\boldsymbol{t}^*[j] = 0$ и $\boldsymbol{s}_q^{(i-1)}[j] = 0$. Поэтому формула (10.39) будет присваивать нулевые значения только в соответствии с позициями j_1, j_2, ..., j_r паттерна и для позиций, следующих после позиции $i-1$ строки $\boldsymbol{x}$.

Хотя вычисление по формуле (10.39) требует выполнение дополнительных операций AND и OR для каждого значения q и каждого значения i, это не влияет на асимптотическую оценку сложности алгоритма.

Комбинирование точного и приближенного совпадений

Возможны ситуации, когда при сравнении строки с паттерном необходимо найти точное совпадение с одной частью паттерна, а с другой частью паттерна достаточно приближенного совпадения. Например, в последовательностях ДНК надо найти фрагмент вида CGATGAGAGCGC, в котором первые четыре буквы (CGAT) и три последние (CGC) будут точно соответствовать этим буквам, а для

средней части, состоящей из пяти букв, достаточно 2-приближенного совпадения с GAGAG. Осуществить такое сравнение достаточно просто, поскольку точное совпадение означает, что в определенных позициях ($j = 1 - 4$и j=10-12) паттерна не разрешается выполнять операции удаления, вставки и подстановки, тогда как для позиций приближенного совпадения следует положить $\boldsymbol{s}_q[j] \leftarrow 1$ для всех $q \in$ $\in 0..k$. Чтобы сделать такое, введем новый бинарный вектор $\boldsymbol{e} = \boldsymbol{e}[1..m]$, в котором $\boldsymbol{e}[j] = 1$ только в том случае, если в позиции j требуется точное совпадение. В нашем примере $\boldsymbol{e} \leftarrow 111100000111$. Этот вектор используется для присваивания $\boldsymbol{s}_q^{(i)}[j] \leftarrow 1$, если не выполняются условия точного совпадения $\boldsymbol{s}_q^{(i-1)}[j-1] = 0$ и $\boldsymbol{x}[i] = \boldsymbol{p}[j]$. С использованием вектора $\boldsymbol{e}$ формулу (10.38) следует изменить следующим образом:

$$\begin{gathered}\boldsymbol{s}_q^{(i)} \leftarrow ((\boldsymbol{s}_{q-I}^{(i-1)} \wedge rightshift(\boldsymbol{s}_{q-D}^{(i)}, 1) \wedge \textit{rightshift}(\boldsymbol{s}_{q-S}^{(i-1)}, 1)) \vee e) \wedge \\ \wedge(\textit{rightshift}(\boldsymbol{s}_q^{(i-1)}, 1) \vee \boldsymbol{t}[x[i], 1..m]).\end{gathered} \tag{10.40}$$

Хотя формула (10.40) добавляет одну операцию OR к операциям, выполняемым по формуле (10.38), асимптотическая оценка сложности алгоритма не изменяется.

Сравнение с минимальным k

Представляет интерес задача нахождения *минимального* значения k^*, для которого имеет место одно или несколько k^*-приближенных совпадений паттерна и заданной строки. Чтобы вычислить такое k^*, сначала можно выполнить точное сравнение ($k = 0$) с помощью, например, алгоритма ДБГ. Если точного совпадения нет, то затем выполняется алгоритм ВМ для $k = 1, 3, \ldots, 2^B - 1$ до тех пор, пока для некоторого значения $B \geqslant 1$ не будет найдено, по крайней мере, одно k^*-приближенное совпадение с паттерном, при этом $k^* \in 2^{B-1}..2^B - 1$. Каждое выполнение алгоритма ВМ требует времени, пропорционального используемому значению k. Поэтому общее время, требуемое для B выполнений алгоритма ВМ, пропорционально сумме

$$\Sigma = 1 + 3 + .. + (2^B - 1) < 2^{B+1}.$$

Поскольку $k^* \geqslant 2^{B-1}$, то $\Sigma / k^* < 4$; следовательно, время вычисления k^* имеет порядок $O(4k^* \lceil m/w \rceil n)$.

Другие применения алгоритма ВМ

Как мы увидим в следующей главе, алгоритм ВМ можно применить для сравнения с регулярными выражениями, а после небольшой модернизации — для эффективного сравнения с множественными паттернами. И конечно, эти применения алгоритма можно комбинировать с рассмотренными выше применениями. Например, алгоритм ВМ можно использовать для поиска совпадения с несколькими паттернами, каждый из которых может содержать символы замещения, или

для поиска комбинированного точного и приближенного совпадения, а также для нахождения минимального значения k. Таким образом, если паттерн не является экстремально длинным, то эти примеры еще раз свидетельствуют о чрезвычайной гибкости алгоритма BM!

В работе [231] предложены определенные усовершенствования алгоритма BM. В разделе 11.2.4 будет описан другой алгоритм [26], который предлагает эффективный подход к решению задачи приближенного сравнения с множественными паттернами.

Упражнения 10.4

1. Запишите алгоритм BM (используя формулу (10.38)) и докажите его корректность. Удостоверьтесь, что алгоритм работает корректно, если одно из значений I, D или S превышает значение q. Объясните, почему при любом i необходимо проверить только значение $\boldsymbol{s}_k[m]$, а не все значения $\boldsymbol{s}_q[m]$, $q = 1, 2, \ldots, k$.
2. Как изменить формулу (10.38), чтобы алгоритм BM вычислял расстояние Хемминга? Сравните пространственную и временну́ю сложности измененного алгоритма BM с аналогичными показателями алгоритма ДБГ.
3. Для расстояния преобразования рассчитайте аналог табл. 10.7 для $k = 1$ и $k = 2$.
4. Постройте таблицу, показывающую зависимость максимальной длины паттерна m, который может обрабатывать алгоритм BM, от величин $k \in 1..10$ и $B \in 1..10$ для машинных слов длиной $w = 32$ и $w = 64$ (B — количество машинных слов, необходимых для хранения одного бинарного вектора $\boldsymbol{s}_q$).

10.5 Сложность алгоритмов приближенного сравнения с паттерном

Как и в случае точного сравнения с паттерном (раздел 8.8), здесь рассмотрим теоретические границы сложности алгоритмов приближенного сравнения с паттерном. Поскольку для малых значений аппроксимационной константы k приближенное сравнение паттерна и строки близко к их точному сравнению, естественно предположить, что должны существовать алгоритмы с линейным временем выполнения в самом худшем случае и почти линейным временем в среднем — это временны́е ограничения алгоритмов точного сравнения с паттерном. Действительно, существуют алгоритмы решения задачи k-разностей, асимптотические оценки времени выполнения которых близки к этим требованиям. В табл. 10.8 для сравнения приведены оценки времени выполнения двух уже изученных алгоритмов (алгоритм Майерса и алгоритм MM), одного алгоритма, который будет рассмотрен

в следующей главе (алгоритм BYN), и двух недавно разработанных алгоритмов, которые подробно в этой книге не рассматриваются.

Таблица 10.8. Сложность алгоритмов приближенного сравнения с паттерном

Алгоритм	Временна́я граница в самом худшем случае	Временна́я граница в среднем
Майерса (подразд. 10.3.2)	$O(mn/\log n)$	$O(mn/\log n)$
ММ (подразд. 10.3.2)	$O(mn/\log n)$	$O(kn/\log n)$
BYN (подразд. 11.2.4)	$O(kn(m-k)/\log n)$	
Чанга и Лоулера [45]	$\Theta(kn+m)$	$O(kn\log m/m)$
Коула и Харихарана [51]	$O(n(k^4/m+1)+m)$	

На основе этой таблицы можно сделать такие заключения.

- Если значение m мало по сравнению со значением n (например, $m \in \in O(\log n)$), алгоритмы Майерса и ММ в самом худшем случае выполняются за линейное и почти линейное по n время. Если же принять $k \in O(\log n)$, то алгоритм ММ выполняется в среднем за время порядка $O(n)$.
- Для алгоритма BYN временная граница выполнения в самом худшем случае в общем случае превышает аналогичные границы для алгоритмов Майерса и ММ. Разработан вариант алгоритма BYN, среднее время выполнения которого имеет порядок $O(n)$, если значение отношения k/m "среднего размера". Однако достижение устойчивого линейного поведения в среднем для этого алгоритма является пока нерешенной задачей.
- В алгоритме Чанга–Лоулера линейное в среднем время выполнения порядка $O(n)$ гарантировано, если выполняется условие $k < m/(\log m + O(1))$, тогда как в алгоритме ММ соответствующее условие имеет вид $k \in O(\log n)$. Поэтому при выполнении условия

$$m \in \Theta(\log m \log n) \tag{10.41}$$

  верхние границы по k времени выполнения обоих алгоритмов эквивалентны с некоторым преимуществом, по крайней мере теоретическим, алгоритма ММ для случая значения m, меньшего, чем требует условие (10.41), и с преимуществом алгоритма Чанга–Лоулера для больших значений m.
- Алгоритм Коула–Харихарана имеет линейное время выполнения в самом худшем случае при выполнении условия $k \in O(m^{1/4})$. В работе [51] сделано предположение, что это условие можно ослабить, потребовав $k \in O(m^{1/3})$.

В статье [191] проведен более детальный сравнительный анализ эффективности алгоритмов приближенного сравнения с паттерном. Заинтересованных читателей мы направляем к этой работе.

ГЛАВА 11

Регулярные выражения и множественные паттерны

> Я ненавижу неверные слова и тщательно, упрямо и угрюмо ищу те слова, которые соответствуют данному предмету.
>
> — Уолтер Севидж Лэндор (1775–1864), Воображаемые беседы

Я осознаю, что многие мои коллеги будут удивлены и даже шокированы, что только в главе 11 почти на 300-й странице книги я ввожу в рассмотрение конечные автоматы, которые являются центральной идеей как компьютерных наук в целом, так и вычислительных процессов обработки строковых последовательностей в частности.

В свою защиту я могу сказать, что в книге основное внимание уделяется алгоритмам, и при их описании я использую "минималистский" подход: объяснение идей, которые лежат в основе алгоритмов, дано очень кратко с минимумом обращений к теоретическим изысканиям. Правильно это или нет — судить читателю, но мне кажется, что объяснение основ алгоритмов будет более ясным и понятным без привлечения идеи конечного автомата.

Однако теперь ситуация изменилась: алгоритмы сравнения с регулярными выражениями и с множественными паттернами невозможно описать без использования понятия конечного автомата. В этой главе мы встретимся с различными типами конечных автоматов, каждый из которых решает свои задачи. Чтобы сделать дальнейший материал книги более понятным, представим очень краткое и не

формальное введение в теорию конечных автоматов. Желающим ознакомиться с теорией конечных автоматов в более полном объеме предлагаем обратиться к книге [159].

Самое простое определение конечного автомата — это представить его в виде помеченного ориентированного графа, в котором

- вершины графа представляют "состояния" — состояния конечного автомата, состояния процесса вычисления;
- метки вершин идентифицируют состояния;
- дуги обозначают переход из одного состояния в другое;
- метки дуг — отдельные буквы или строки, относящиеся к соответствующему переходу из одного состояния в другое.

Из этого определения нетрудно заметить, что мы уже встречались в этой книге с конечными автоматами — это дерево суффиксов! Чтобы сделать это замечание очевидным, на рис. 11.1 показано дерево суффиксов, которое ранее было представлено на рис. 5.2. (Отличие этих рисунков состоит только в том, что здесь из рис. 5.2 удалены связи, внутренние вершины помечены как состояния s_1, s_2, ..., s_{11} и нисходящие ребра заменены ориентированными дугами.)

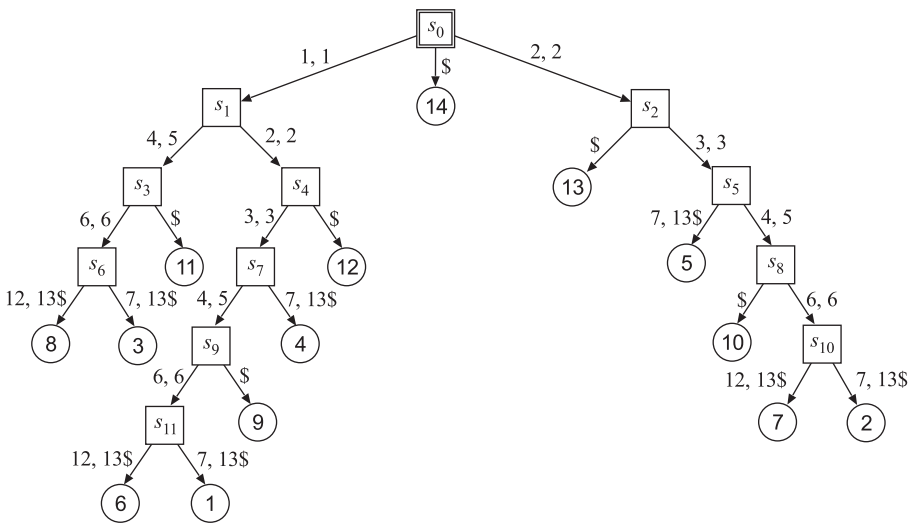


Рис. 11.1. Дерево суффиксов для $\boldsymbol{f}_6 = abaababaabaab\$$ в виде конечного автомата

Если мы подставим входную строку $\boldsymbol{p} = aab\$$ в вершину (состояние) s_0, тогда первая буква $\boldsymbol{p}[1] = a$ будет равна $\boldsymbol{f}_6[1] = a$, что определяет переход в состояние s_1. В этом состоянии $\boldsymbol{p}[2..3] = \boldsymbol{f}_6[4..5] = ab$, поэтому далее осуществляется переход в состояние s_3, где наконец $\boldsymbol{p}[4] = \$$ с переходом в конечное состояние

11, которое показывает, что $\boldsymbol{f}_6[11..14] = \boldsymbol{p}$. В данном случае очевидно, что произвольный паттерн $\boldsymbol{p}$, поступающий на вход "суффиксного" автомата, достигнет конечного состояния t только в том случае, если $\boldsymbol{p}$ является суффиксом $\boldsymbol{f}_7[t..14]$ строки $abaababaabaab\$$. В противном случае паттерн $\boldsymbol{p}$ "застрянет" в одной из внутренних вершин, поскольку в нем нет буквы или подстроки, совпадающих с метками дуг, выходящих из этой вершины.

Дерево суффиксов любой строки $\boldsymbol{x}$ (и даже набора строк) можно интерпретировать и исследовать подобным способом — таким образом, мы видим, что конечные автоматы действительно предлагают новый графический способ представления вычислений, выполняемых над строковыми последовательностями. В этой главе мы изучим несколько подобных типов конечных автоматов, которые будем называть ***суффиксными автоматами***.

Однако следует отметить, что суффиксные автоматы решают задачу, которая в некотором смысле обратная задаче, рассматриваемой в этой главе: суффиксные автоматы легко решают задачу сравнения одного паттерна с некоторым набором текстовых строк путем включения всех этих строк в одно дерево суффиксов. В этой главе решается задача одновременного сравнения одной текстовой строки с набором паттернов (с множественным паттерном). Как мы увидим далее, для решения последней задачи можно построить конечный автомат, который включал бы все паттерны.

В разделе 11.1 сначала будут рассмотрены недетерминированные конечные автоматы (сокращенно, NDFA — от non-deterministic finite automata), с помощью которых решим задачу сравнения с регулярными выражениями. Затем покажем, как на основе NDFA построить детерминированные конечные автоматы (сокращенно, DFA) для более эффективного решения той же задачи. В последнем подразделе раздела 11.1 рассмотрим вариант алгоритма BM (см. раздел 10.4), решающий задачу k-приближенного сравнения с регулярными выражениями. Здесь этот алгоритм будет представлен в виде модели NDFA, использующей бинарные векторы. На этом основании будет сделан вывод, что алгоритм ДБГ (раздел 7.4) и его многочисленные варианты и модификации также можно представить в виде конечных автоматов.

Раздел 11.2 начнем с описания конечных автоматов, решающих задачу точного сравнения с множественными паттернами, причем основой для решения сначала будет служить алгоритм КМП (раздел 7.1), а затем — алгоритм БМ (раздел 7.2). Далее представим два алгоритма для приближенного сравнения с множественным паттерном, один из которых основан на алгоритме BM, а второй предложен Бейза-Ятсом и Наварро (Baeza-Yates, Navarro).

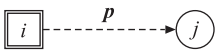
11.1 Алгоритмы для регулярных выражений

В этом разделе рассмотрим три алгоритма, два из которых "классические" (разработаны до 1970 года) и один относительно новый. Эти алгоритмы вычисляют все совпадения всех подстрок строки $\boldsymbol{x}$ с паттерном $\boldsymbol{p}$, который представлен в виде регулярного выражения. Как указывалось в разделе 2.2, сравнение с регулярными выражениями можно считать экстремальным случаем приближенного сравнения.

11.1.1 Недетерминированные конечные автоматы

Прежде чем использовать конечные автоматы (применительно к регулярным выражениям), необходимо сначала научиться их строить. Для построения таких автоматов опишем метод, восходящий к Томпсону (Thompson) [119], основанный на четырех правилах, применяемых рекурсивно. Мы будем строить ***недетерминированный конечный автомат*** (NDFA). "Недетерминированность" здесь проявляется в том, что из некоторых состояний (вершин) возможно несколько переходов (выходных дуг), помеченных пустой строкой ε, и при этом определение выходной дуги неоднозначно. Как мы увидим далее, NDFA, подобно суффиксному автомату, описанному выше, имеет единственное ***начальное состояние*** или ***источник*** и, в отличие от суффиксного автомата, только одну конечную вершину, соответствующую ***поглощающему состоянию*** или ***стоку***. Описываемый здесь NDFA имеет дуги, помеченные или одной буквой или пустой строкой, что напоминает некомпактное дерево суффиксов, описанное в разделе 2.1.

Будем представлять автомат NDFA($\boldsymbol{p}$), соответствующий регулярному выражению $\boldsymbol{p}$, в виде



Здесь i и j — метки начального и поглощающего состояний соответственно, а пунктирная линия обозначает возможные пути от i до j.

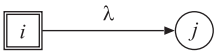
Пусть регулярное выражение $\boldsymbol{p}$ определено на алфавите $A = \{\lambda_1, \lambda_2, \ldots, \lambda_\alpha\}$. Построение автомата NDFA начнем с вычисления

$$\boldsymbol{p}^* \leftarrow (\lambda_1|\lambda_2|\ldots|\lambda_\alpha)^*\boldsymbol{p}.$$

Это позволит сравнивать паттерн $\boldsymbol{p}$ и входную строку $\boldsymbol{x}$ начиная с любой позиции этой строки, поскольку сравнение паттерна $\boldsymbol{p}$ с любой позицией строки $\boldsymbol{x}$ эквивалентно сравнению $\boldsymbol{p}^*$ со строкой $\boldsymbol{x}$, начинаемого с первой позиции этой строки.

Далее предполагаем, что $\boldsymbol{p}^*$ сканируется по одной позиции слева направо. Автомат NDFA($\boldsymbol{p}$) строится по следующим правилам.

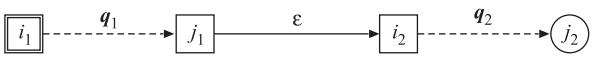
П.1. Для произвольной буквы $\lambda \in A$ (не метасимвола) автомат NDFA(λ) определяется следующим образом:



Очевидно, что определенный таким образом автомат NDFA(λ) допустим только для одиночной буквы, но не для какой-либо строки, состоящей из двух и более букв.

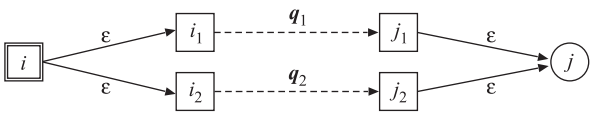
П.2. Предположим, что для некоторых регулярных выражений $\boldsymbol{q}_1$ и $\boldsymbol{q}_2$ уже определены автоматы NDFA($\boldsymbol{q}_1$) и NDFA($\boldsymbol{q}_2$):

Тогда NDFA($\boldsymbol{q}_1\boldsymbol{q}_2$) строится так:



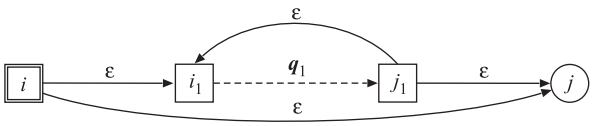
Здесь пустая метка ε обозначает переход из одного состояния в другое без инициации этого перехода каким-либо входным воздействием. В данном случае такой "пустой" переход переопределяет поглощающее состояние j_1 автомата NDFA($\boldsymbol{q}_1$) в начальное состояние j_2 автомата NDFA($\boldsymbol{q}_2$), при этом оба состояния j_1 и j_2 становятся внутренними состояниями автомата NDFA($\boldsymbol{q}_1\boldsymbol{q}_2$). Отметим, что автомат NDFA($\boldsymbol{q}_1\boldsymbol{q}_2$) определен только для строк вида $\boldsymbol{q}_1\varepsilon\boldsymbol{q}_2 = \boldsymbol{q}_1\boldsymbol{q}_2$.

П.3. Пусть для регулярных выражений $\boldsymbol{q}_1$ и $\boldsymbol{q}_2$ уже определены автоматы NDFA($\boldsymbol{q}_1$) и NDFA($\boldsymbol{q}_2$). Тогда автомат NDFA($\boldsymbol{q}_1 \mid \boldsymbol{q}_2$) строится так:



Таким образом, поглощающие состояния автомата NDFA($\boldsymbol{q}_1$) и начальные состояния автомата NDFA($\boldsymbol{q}_2$) становятся обычными внутренними состояниями автомата NDFA($\boldsymbol{q}_1 \mid \boldsymbol{q}_2$). Отметим, что здесь пустой строкой ε помечены как выходные дуги (переходы) нового начального состояния i, так и входные дуги нового поглощающего состояния j. Автомат NDFA($\boldsymbol{q}_1 \mid \boldsymbol{q}_2$) доступен только для строк $\boldsymbol{q}_1$ и $\boldsymbol{q}_2$, но не доступен ни для каких-либо других строк.

П.4. Предположим, что для некоторого регулярного выражения $\boldsymbol{q}_1$ определен автомат NDFA($\boldsymbol{q}_1$). Тогда автомат NDFA(q_1^*) строится так:



В силу пустых переходов из состояния j_1 в состояние i_1 и из состояния j_1 в состояние j автомат NDFA(q_1^*) доступен как для $\boldsymbol{q}_1$, так и для любой кратной строки q_1^k, $k \geqslant 2$. Также в силу пустых переходов из состояния i в состояние j автомат NDFA(q_1^*) доступен и для пустых строк. Но для любых других строк автомат не доступен. Здесь также начальное состояние i_1 и поглощающее состояние j_1 исходного автомата NDFA($\boldsymbol{q}_1$) становятся обычными внутренними вершинами автомата NDFA(q_1^*).

Эти правила позволяют построить NDFA для выражения $\boldsymbol{p}^*$ в тех случаях, когда в этом выражении встречаются отдельные буквы, конкатенации букв и регулярных выражений, а также метасимволы | и *. Однако эти правила не помогут в случае вложенных операторов (определяемых с помощью метасимволов "скобки"), например, для такого выражения:

$$\boldsymbol{p} = (\boldsymbol{q}_1 \mid (q_2^* \mid \boldsymbol{q}_3))^* \mid (\boldsymbol{q}_4 \mid \boldsymbol{q}_5)^*, \tag{11.1}$$

где $\boldsymbol{q}_i$ ($i = 1, 2, \ldots, 5$) — регулярные выражения (возможно, обычные строки).

Предполагая, что буквы алфавита можно распознать за константное время, выражение, подобное (11.1), можно распознать и проанализировать за время, пропорциональное количеству символов, входящих в ***синтаксическое дерево***. На рис. 11.2 показано такое дерево, соответствующее выражению (11.1). Отметим, что в этом дереве каждое выражение $\boldsymbol{q}_i$, входящее в $\boldsymbol{p}$, соответствует листу (конечному узлу). Синтаксическое дерево можно преобразовать в автомат NDFA($\boldsymbol{p}$) за время, пропорциональное $|\boldsymbol{p}|$. Подробности этого процесса можно найти в [118].

Уже на основе сформулированных выше правил построения автомата NDFA($\boldsymbol{p}$) можно сделать некоторые выводы о свойствах такого автомата.

- Как показано в упражнении 11.1.2, если m — количество букв в выражении $\boldsymbol{p}$, тогда $|\boldsymbol{p}| \leqslant 6m$.
- Поскольку после применения любого из правил П.1–П.4 в NDFA остается только одно начальное и одно поглощающее состояние, то и в NDFA($\boldsymbol{p}$) также имеется единственное начальное и единственное поглощающее состояние.
- Каждая вершина, не являющаяся поглощающим состоянием, имеет или одну исходящую дугу, помеченную буквой, или не более двух исходящих дуг,

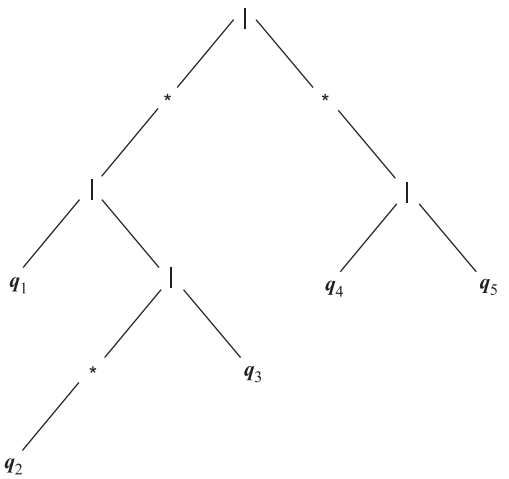


Рис. 11.2. Синтаксическое дерево для выражения (11.1)

помеченных пустой строкой ε. Аналогично каждая вершина, не являющаяся начальным состоянием, имеет или одну входящую дугу, помеченную буквой, или не более двух входящих дуг, помеченных ε. Отсюда вытекают еще два вывода, которые в дальнейшем будут полезны при построении алгоритма сравнения с регулярными выражениями:

- вершина, не являющаяся поглощающим состоянием, имеет единственную исходящую "непустую" дугу (т.е. дугу, помеченную не ε) тогда и только тогда, когда она не имеет исходящих "пустых" дуг;
- каждая "непустая" дуга ведет к вершине, которая либо соответствует поглощающему состоянию, либо имеет "пустые" исходящие дуги.

■ Пусть NDFA($\boldsymbol{p}$) $= (V, A)$, где V — множество вершин, а A — множество дуг. Когда, как показано в упражнении 11.1.3,

$$|V| \leqslant 8m, |A| \leqslant 13m, \tag{11.2}$$

где, как и ранее, m — количество отдельных букв в выражении $\boldsymbol{p}$. Отсюда следует, что NDFA($\boldsymbol{p}$) можно вычислить за время порядка $\Theta(m)$.

На рис. 11.3 показан типичный NDFA, в данном случае построенный для паттерна (регулярного выражения) $\boldsymbol{p} = (a \mid b)^*ab^*a$, где префикс $(a \mid b)^*$, как отмечалось выше, гарантирует совпадение с буквами, стоящими в любой позиции любой строки $\boldsymbol{x}$, определенной на алфавите $\{a, b\}$.

Теперь, когда показано, что автомат NDFA($\boldsymbol{p}$) можно построить за время, пропорциональное количеству букв в регулярном выражении $\boldsymbol{p}$, расскажем, как

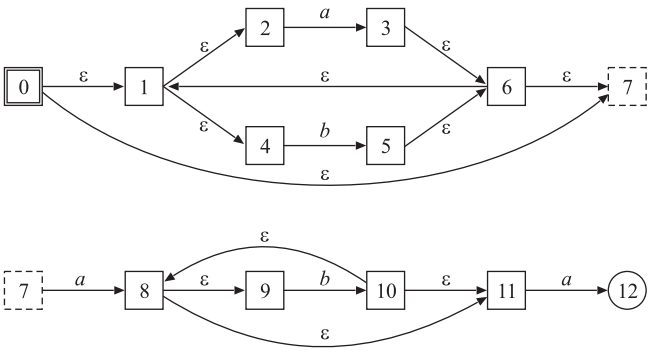


Рис. 11.3. Пример NDFA

применить такой автомат. Допустим, на вход автомата поочередно поступают буквы текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Для каждой вновь поступающей буквы строки $\boldsymbol{x}$ необходимо определить возможные переходы из текущего активного множества состояний (множества вершин) автомата NDFA($\boldsymbol{p}$), при этом совпадение с паттерном будет зарегистрировано только в том случае, если эти переходы приведут к поглощающему состоянию.

Для выполнения таких вычислений введем функцию *trans* (от transition — переход), которая для заданного множества состояний S и заданной буквы $\lambda \in A$ (текущая входная буква) вычислит новое множество состояний, достижимых путем следования по дугам, помеченным буквой λ и выходящим из вершин множества S. Более точно, состояние j автомата NDFA($\boldsymbol{p}$) назовем λ-***достижимым*** из состояния i, если выполняются следующие условия:

- состояние (вершина) i имеет выходную дугу, помеченную буквой λ;
- в NDFA($\boldsymbol{p}$) существует ориентированный путь от вершины i до вершины j, дуги которого (пути) помечены как $\lambda\varepsilon^k$ для некоторого целого $k \geqslant 0$;
- состояние j не имеет выходных дуг, помеченных ε.

Данные условия полностью определяют функцию $trans(S, \lambda)$. Расширим это определение для того, чтобы включить два особых случая.

- В случае, когда $S = \varnothing$ (пустое множество), функция $trans(S, \lambda)$ интерпретируется как $trans(\{0\}, \lambda)$, где 0 — метка начального состояния автомата NDFA($\boldsymbol{p}$).
- Обозначение $trans(S, \varepsilon)$ будем понимать как множество всех состояний, достижимых из множества S, если следовать по выходящим из этого множества дугам, помеченным пустой строкой ε.

Для автомата, показанного на рис. 11.3, приведем для примера несколько значений функции *trans*.

$$\mathit{trans}(\varnothing, a) = \mathit{trans}(\{3\}, a) = \mathit{trans}(\{12\}, \varepsilon) = \varnothing.$$
$$\mathit{trans}(\{2\}, a) = \mathit{trans}(\varnothing, \varepsilon) = \mathit{trans}(\{5\}, \varepsilon) = \{2, 4, 7\}.$$

Приведенный ниже алгоритм 11.1.1 использует функцию *trans* для вычисления в данной строке $\boldsymbol{x}$ всех позиций i, для которых имеет место совпадение подстрок $\boldsymbol{x}[i'..i]$ с непустым регулярным выражением $\boldsymbol{p}$. Доказательство корректности этого алгоритма оставим для упражнения 11.1.8, доказательство базируется на корректности реализации функции *trans* и на свойствах автомата NDFA.

Алгоритм 11.1.1. (Алгоритм NDFA)

▷ *Нахождение всех подстрок строки* $\boldsymbol{x}$,
▷ *совпадающих с заданным непустым регулярным выражением* $\boldsymbol{p}$
построение NDFA($\boldsymbol{p}$) с поглощающим состоянием w
$S \leftarrow \mathit{trans}(\varnothing, \varepsilon)$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow \mathit{trans}(S, \boldsymbol{x}[i])$
    if $w \in S$ **then**
        output i
    $S \leftarrow \mathit{trans}(S, \varepsilon)$
    if $w \in S$ **then**
        output i

Для того чтобы определить сложность алгоритма NDFA, очевидно, сначала надо определить сложность функции *trans*. Эта функция выполняется $2n + 1$ раз, поочередно используя в качестве аргументов ε и $\boldsymbol{x}[i]$. Функцию *trans* можно реализовать с использованием двух очередей, требующих памяти порядка $O(|V|)$ и бинарного отображения, требующего памяти порядка $\Theta(|V|)$, где $|V| \in \Theta(m)$ — количество состояний (вершин) автомата NDFA($\boldsymbol{p}$), m — количество букв в выражении $\boldsymbol{p}$. Записав множество состояний NDFA в одну очередь, нетрудно вычислить переходы (не более двух переходов) из этих состояний и записать состояния, полученные в результате этих переходов, в другую очередь. Бинарное отображение необходимо для того, чтобы все состояния, записанные во вторую очередь, были различны. Поскольку во второй очереди надо хранить не более $|V|$ состояний и поскольку вычисляется не более двух переходов для каждого состояния, то функция *trans* будет выполняться за время порядка $O(|V|) = O(m)$. Напомним, что автомат NDFA($\boldsymbol{p}$) строится за время порядка $\Theta(m)$. Таким образом, доказан следующий основной результат данного подраздела.

Теорема 11.1.1. Для непустого регулярного выражения $\boldsymbol{p}$, содержащего m букв, и для заданной строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ алгоритм 11.1.1 вычисляет все подстроки строки $\boldsymbol{x}$, совпадающие с паттерном $\boldsymbol{p}$, за время порядка $O(mn)$ с использованием памяти объемом $\Theta(m)$. ■

С помощью "четырех русских" методов [20] можно уменьшить время выполнения алгоритма NDFA до величины порядка $O(mn/\log n)$, но за счет увеличения объема используемой памяти до величины порядка $O(mn/\log n)$ [188].

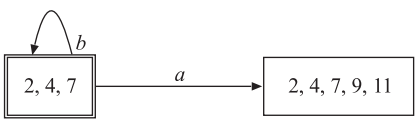
11.1.2 Детерминированные конечные автоматы

В этом подразделе мы рассмотрим способ построения детерминированного конечного автомата (сокращенно, DFA — от deterministic finite automaton) для заданного регулярного выражения $\boldsymbol{p}$, т.е. такого конечного автомата, где не будет "пустых" переходов, а из каждого состояния можно будет перейти только в одно состояние, соответствующее какой-либо букве алфавита A. Такой автомат DFA($\boldsymbol{p}$) можно использовать, как мы увидим ниже, для вычисления всех совпадений с паттерном $\boldsymbol{p}$ заданной строки $\boldsymbol{x}$, причем в большинстве случаев это можно сделать более эффективно, чем посредством автомата NDFA($\boldsymbol{p}$). Далее будем предполагать, что алфавит A индексирован (раздел 4.1).

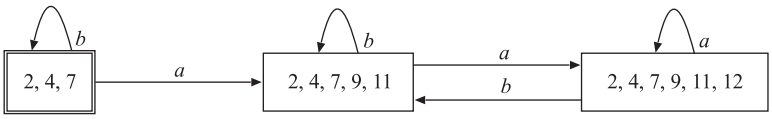
Построение DFA($\boldsymbol{p}$) для заданного регулярного выражения $\boldsymbol{p}$ основано на простой идее: исключить из автомата NDFA($\boldsymbol{p}$) пустые переходы путем объединения вершин, которые можно достичь посредством пустых переходов.

Для примера рассмотрим NDFA, показанный на рис. 11.3. Начиная с начального состояния 0 следуем по всем дугам, помеченным ε, до тех пор, пока не встретится вершина, из которой выходит "непустая" дуга — в данном случае это вершины 2, 4 и 7. Таким образом, начальной вершиной автомата DFA($(a \mid b)^*ab^*a$) будет

Теперь найдем в NDFA все состояния, достижимые из состояний 2, 4, 7 путем a-перехода и последующих пустых переходов. Это состояния 2, 4, 7, 9 и 11, которые объединим в новую вершину:



Здесь также показано, что b-переход из вершин 2, 4, 7 приводит к тому же множеству вершин. Продолжая этот процесс, получим автомат DFA($(a \mid b)^*ab^*a$) следующего вида.



Такие построения можно выполнить на основе любого NDFA, получив в результате DFA, в котором каждой вершине соответствует уникальное множество состояний, достижимых из ранее определенного множества состояний для новой входной буквы. Процесс построения завершается, когда у любой вершины не будет исходящих дуг, требующих определения нового множества состояний, отличного от ранее определенных множеств. Любая вершина DFA, включающая поглощающее состояние автомата NDFA, также называется поглощающим состоянием. Нетрудно показать, что описанный процесс построения DFA должен обязательно завершиться и что этот DFA должен иметь хотя бы одно поглощающее состояние. На основе рассмотренного примера можно сделать предположения, что для любого регулярного выражения $\boldsymbol{p}$

- количество вершин в DFA($\boldsymbol{p}$) имеет порядок $O(m)$, где m — количество букв в выражении $\boldsymbol{p}$;
- в DFA($\boldsymbol{p}$) существует в точности одно поглощающее состояние.

Однако, как показано в упражнении 11.1.10, оба эти предположения неверны! Рассмотрим, например, регулярное выражение

$$\boldsymbol{p} = (a \mid b)^* a (a \mid b)^{(m-3)/2},$$

описывающее все строки, определенные на алфавите $\{a, b\}$, в которых после буквы a стоят в точности $m' = (m-3)/2$ произвольных букв (включая букву a). Для $m' = 2$ паттерн $\boldsymbol{p}$ будет иметь одно совпадение со строкой $\boldsymbol{x} = abbab$, два совпадения со строкой $\boldsymbol{x} = abaab$ и три совпадения с $\boldsymbol{x} = aaabb$. Чтобы подсчитать количество букв, следующих за последней буквой a, в автомате DFA необходимо создать $\Theta(2^{m/2})$ различных вершин, многие из которых будут содержать поглощающее состояние автомата NDFA, и вследствие этого такие вершины DFA сами будут поглощающими состояниями. Как показано в упражнении 11.1.23, в самом худшем случае количество вершин в DFA может иметь порядок $\Theta(2^m)$.

Описанный процесс построения DFA не труден для понимания, однако его непросто реализовать программно. Вот одна из трудностей, возникающая при попытке практической реализации этого процесса. Каждый новый переход в автомате DFA определяет вершину, которая помечается набором целых чисел (эти числа принадлежат некоему множеству из $m+1$ чисел). Необходимо определить, встречался ли текущий набор чисел ранее (как метка уже созданных вершин DFA) или нет, и если не встречался, то его надо сохранить. Нетрудно присвоить каждой вершине, помеченной набором чисел $\{s_1, s_2, \ldots, s_t\}$, $1 \leqslant t \leqslant m+1$, уникальный целочисленный идентификатор вида

$$S = 2^{s_1} + 2^{s_2} + \cdots + 2^{s_t},$$

задающий бинарное отображение меток и требующий для хранения $O(m)$ бит. Таким образом, на каждом шаге построения DFA идентификатор S новой вершины надо сравнивать с $O(2^m)$ идентификаторами ранее созданных вершин.

Будем предполагать, что идентификаторы S принимают значения из интервала от 1 до 2^{m+1}, — всегда можно так перенумеровать метки автомата NDFA (в общем случае, за время, пропорциональное m), чтобы это условие выполнялось. Теперь можно определить массив $\boldsymbol{I}[1..2^{m+1}]$, в котором любой элемент $\boldsymbol{I}[S]$ или принимал бы значение `NULL`, или был бы указателем на вершину с идентификатором S автомата DFA. Итак, для хранения массива $\boldsymbol{I}$ требуется память объемом $\Theta(2^m)$, каждый элемент $\boldsymbol{I}[S]$ этого массива можно изменить за константное время, а поиск нужного идентификатора в массиве $\boldsymbol{I}$ можно выполнить за время порядка $O(2^m)$. Заметим также, что для каждой вершины DFA необходима дополнительная память, пропорциональная α (размеру алфавита), поскольку каждая вершина имеет не более α исходящих дуг. Для заполнения такой памяти необходимо время порядка $O(\alpha)$ в расчете на одну вершину. Таким образом, мы доказали следующую теорему.

Теорема 11.1.2. Для заданного регулярного выражения $\boldsymbol{p}$, содержащего m букв, автомат DFA($\boldsymbol{p}$) можно построить за время порядка $O(2^m\alpha)$ с использованием памяти объемом $O(2^m\alpha)$. ■

Как показано в [175], автомат DFA можно построить непосредственно на основе синтаксического дерева, однако такой способ не уменьшает границы сложности процесса построения DFA, приведенные в теореме 11.1.2.

Когда DFA($\boldsymbol{p}$) построен, нахождение всех подстрок строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$, совпадающих с паттерном $\boldsymbol{p}$, считается уже тривиальной задачей. Назовем алгоритмом DFA алгоритм, решающий эту задачу на основе построенного автомата DFA. В этом алгоритме должна использоваться функция, подобная функции *trans*, которая бы выполняла переходы в DFA и возвращала текущую позицию строки $\boldsymbol{x}$, для которой достигнуто поглощающее состояние. Для индексированного алфавита любой переход можно выполнить за константное время, для неиндексированного алфавита для этого необходимо время порядка $O(\log\alpha)$ (как для дерева суффиксов). Имеет место следующая теорема.

Теорема 11.1.3. На основе автомата DFA($\boldsymbol{p}$), построенного для непустого регулярного выражения $\boldsymbol{p}$, для заданной строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ ($\boldsymbol{p}$ и $\boldsymbol{x}$ определены на алфавите A) можно вычислить все подстроки строки $\boldsymbol{x}$, совпадающие с паттерном $\boldsymbol{p}$, за время порядка $O(n\log\alpha)$ (порядка $O(n)$, если алфавит A индексирован) с использованием памяти фиксированного объема. ■

Приведенную оценку эффективности для алгоритма DFA в случае индексированного алфавита можно доказать с помощью ***переходной матрицы*** $T =$

$= T[1..M, 1..\alpha]$, где M — количество вершин в DFA($\boldsymbol{p}$). Введение этой матрицы требует переопределения вершин числами от 1 до M вместо идентификаторов S, введенных при построении DFA. Каждый элемент $T[j, h]$ переходной матрицы равен метке (номеру) вершины DFA($\boldsymbol{p}$), которую можно достигнуть из вершины j при предъявлении на вход автомата буквы $h \in 1..\alpha$.

Даже для неиндексированного алфавита алгоритм DFA находит отдельные совпадения с регулярными выражениями чрезвычайно быстро, особенно в ситуациях, когда с одним паттерном надо сравнить несколько текстовых строк (поскольку построенный автомат DFA можно использовать повторно).

Однако наши дифирамбы в адрес алгоритма DFA несколько не обоснованы: если в процесс сравнения с паттерном включить процесс построения DFA, то общее время выполнения такого алгоритма составит время порядка $O(2^m\alpha + n \log \alpha)$ с использованием памяти объемом $O(2^m\alpha)$ — весьма сомнительное улучшение аналогичных оценок $O(mn)$ и $O(m)$ алгоритма NDFA. Но есть и хорошие новости: случай, когда количество вершин в DFA имеет порядок $O(2^m)$, является патологическим. В большинстве случаев (как в рассмотренном выше примере) количество вершин в DFA имеет порядок $O(m)$. Поэтому временны́е и пространственные границы для алгоритма DFA можно уменьшить до величин порядка $O(m\alpha + n)$ и $O(n\alpha)$ соответственно. Кроме того, в [6] предложено несколько подходов, применение которых уменьшает временны́е и пространственные показатели вычислений на основе DFA. Один из таких подходов основан на "отложенных вычислениях переходов", когда возможные переходы вычисляются не в процессе построения DFA, а, по мере необходимости, в процессе сравнения с паттерном. Такой подход, согласно [2], уменьшает общее время построения и использования DFA до величины порядка $O(m + n)$ даже в самом худшем случае.

11.1.3 Модификация алгоритма BM

В этом подразделе мы покажем, как для решения задачи сравнения с регулярным выражением можно применить методику использования бинарных векторов алгоритма BM (раздел 10.4), где эта методика позволила эффективно вычислить все k-приближенные совпадения текстовой строки $\boldsymbol{x}$ и паттерна $\boldsymbol{p}$. Для применения этой методики в данном случае положим, что в NDFA($\boldsymbol{p}$) каждое состояние представимо одним битом[1] и если вершины помечены числами от 0 до m', то каждой вершине соответствует бинарный вектор, содержащий $m' + 1$ бит. Напомним (подраздел 11.1.1), что если паттерн $\boldsymbol{p}$ содержит m букв, то $m' \in \Theta(m)$. Здесь, как и в алгоритме BM, должны выполняться некоторые условия: алфавит A должен быть конечным, известным заранее и индексированным. В этом случае можно считать, что $A = \{1, 2, \ldots, \alpha\}$.

[1] Этот бит должен только показать, активно ли данное состояние. — *Примеч. ред.*

В новой модификации алгоритма BM (назовем модифицированный алгоритм алгоритмом BM*) построим бинарные векторы $\boldsymbol{s}_q[1..m']$, $q = 0, 1, \ldots, k$, которые будут почти полным аналогом векторов (10.34). Единственное отличие состоит в том, что теперь j-й элемент в $\boldsymbol{s}_q$ определяет не j-ю позицию в $\boldsymbol{p}$, а j-ю метку в NDFA($\boldsymbol{p}$). Конечно, если ищется точное совпадение (тогда $k = 0$), то в этом случае будет только один бинарный вектор $\boldsymbol{s}_0$.

Аналогично переопределим бинарный массив $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 0..m']$, положив, что $\boldsymbol{t}[h, j] = 0$ только тогда, когда дуга, направленная из вершины j в вершину $j + 1$, имеет меткой букву h. Теперь для любой позиции i строки $\boldsymbol{x}$ выражение

$$\boldsymbol{s}_q^{(i-1)}[j-1] \vee \boldsymbol{t}[x[i], j] \tag{11.3}$$

в формуле (10.37) будет вычислено. Возвращаясь к рис. 11.3, мы видим, что выражение (11.3) корректно вычисляется для вершин 2, 3 и буквы a, для вершин 4, 5 и буквы b и т.д. Более того, из этого рисунка видно, что эту корректность для непустых переходов можно обеспечить путем присваивания последовательных меток концевым вершинам непустых дуг.

Однако для обеспечения корректности вычислений для всех вершин необходимо учесть еще две возможные ситуации.

1. Если в $\boldsymbol{p}$ входит метасимвол |, тогда в NDFA($\boldsymbol{p}$) должна существовать по крайней мере одна пара вершин с последовательными метками $(j, j + 1)$, между которыми *нет* перехода (правило 3 построения NDFA). На рис. 11.3 есть такая пара вершин — это вершины $(3, 4)$. Такую ситуацию легко формализовать, если ввести специальный шаблон, запрещающий для подобного случая переход из j в $j + 1$.
2. В NDFA($\boldsymbol{p}$), как правило, много пар вершин (j_1, j_2), соединенных пустыми дугами (т.е. дугами, помеченными пустой строкой ε), причем среди них обязательно есть пары вершин с последовательными метками, когда $j_2 = j_1 + 1$. Но вхождение в $\boldsymbol{p}$ метасимволов | и * обязательно порождает дуги (j_1, j_2), для которых $j_2 \neq j_1 + 1$, просто как результат применения правил 3 и 4 построения NDFA. На рис. 11.3 это дуги $(1, 4)$, $(3, 6)$, $(8, 11)$ и $(10, 8)$.

Ниже опишем механизм обращения с пустыми дугами независимо от того, соединяют они последовательные вершины или нет.

Существование пустой дуги (j_1, j_2) просто показывает, что если имеет место приближенное совпадение, соответствующее вершине j_1, то это же утверждение справедливо и для вершины j_2. Напомним (см. подраздел 11.1.1), что в NDFA существует два типа вершин:

- вершины, из которых исходит одна или две пустые дуги (первый тип вершин);
- вершины, из которых исходит одна непустая дуга или которые вообще не имеют исходящих дуг (второй тип вершин).

Нам надо все вершины j_1 первого типа заменить вершинами j_2 второго типа. Для этого можно проследить пути от вершин первого типа, проходящие по пустым дугам и заканчивающиеся в вершинах второго типа, — правила построения NDFA предоставляют такую возможность. Этот процесс можно представить в виде функции $f : J_1 \to J_2$, отображающей множество J_1 вершин первого типа на множество J_2 вершин второго типа. Поскольку элементами множеств J_1 и J_2 являются несовпадающие целые числа из интервала от 0 до m', то функцию f рационально представить как отображение бинарных векторов длиной $m'+1$ на множество подобных бинарных векторов.

Чтобы реализовать такое отображение эффективно, в алгоритме BM* на предварительном этапе строятся массивы T_r, которые представляют *все возможные* переходы $J_1 \to J_2$. Каждый такой массив соответствует одному байту (восьми последовательным битам) в некотором векторе $\boldsymbol{s}_q$. Всего $\lceil m'/8 \rceil$ таких массивов T_r. Тогда $r = 0, 1, \ldots, \lceil m'/8 \rceil - 1$, что соответствует позициям

$$8r..8r+7 = 0..7, 8..15, \ldots, 8(\lceil m'/8 \rceil - 1)..m'$$

векторов $\boldsymbol{s}_q$ (вершинам NDFA($\boldsymbol{p}$)). Каждый такой байт представляет целое число b из интервала от 0 до 255. Для каждого значения $b \in 0..255$ в некоторой позиции $j_1 \in 0..7$ этого байта стоит 0, показывая, что вершина $j_1 + 8r$ является текущей активной вершиной NDFA($\boldsymbol{p}$). Другими словами, значение 0 в позиции j_1 указывает на то, что вершине $j_1 + 8r$ соответствует q-приближенное совпадение некоторой подстроки строки $\boldsymbol{x}$ и регулярного выражения $\boldsymbol{p}$. Тогда в бинарном векторе $T_r[b]$ длиной $m'+1$ в каждой позиции $j_2 \in 0..m'+1$ стоит значение 0 только в том случае, если в NDFA($\boldsymbol{p}$) вершина j_2 является вершиной второго типа, достижимой из одной из вершин $j_1 + 8r$ посредством пути, состоящего только из пустых дуг. Каждый массив T_r является двухмерным массивом, содержащим $2^8(m'+1)$ байт; его можно вычислить за время порядка $\Theta(m')$. Тогда общее время вычисления всех $m'/8$ массивов T_r составляет величину порядка $\Theta((m')^2/8)$.

Обозначим через $s_{q,r}$ значение r-го байта вектора $\boldsymbol{s}_q$. С помощью T_r для любого $r \in 0..\lceil m'/8 \rceil$ можно вычислить выражение

$$T_r[s_{q,r}] \wedge \boldsymbol{s}_q, \tag{11.4}$$

где $\wedge$ обозначает операцию логического умножения AND. Результатом вычисления (11.4) всегда будет значение 0, кроме случая, когда оба операнда в (11.4) равны 1. Поэтому активная в настоящий момент вершина $j_1 + 8r$ будет оставаться активной до тех пор, пока все вершины j_2 второго типа, достижимые из вершины $j_1 + 8r$, также будут активными. Поскольку операцию AND можно выполнить сразу над всем машинным словом, то $\lceil m'/8 \rceil$ вычислений выражения (11.4) на каждом шаге алгоритма требуют времени порядка $\Theta((m')^2/8w)$, где, как и ранее, w — длина машинного слова. Всего в процессе вычислений по формуле (10.37)

в алгоритме BM выражение (11.4) будет вычисляться kn раз. Отметим, что совпадение с паттерном будет зарегистрировано, если будет выполняться равенство $\boldsymbol{s}_k[m'] = 0$.

В статье [233] доказано, что время выполнения алгоритма BM* имеет порядок

$$O(k\lceil 2m'/\log_2 n\rceil\lceil m'/w\rceil n).$$

Если значение m' не слишком велико, а значение n достаточно велико, тогда можно считать, что время выполнения алгоритма BM* имеет порядок $O(kn)$ с дополнительным временем порядка $\Theta((m')^2)$ для вычислений, выполняемых на предварительном этапе алгоритма.

Упражнения 11.1

1. Постройте NDFA($\boldsymbol{p}$) для регулярного выражения (11.1).
2. Пусть в регулярное выражение $\boldsymbol{p}$ входит m букв. Покажите, что в этом выражении не может быть
   - более $2m$ скобок,
   - более m метасимволов $|$,
   - более $2m$ метасимволов *.

   Затем докажите, что количество $|\boldsymbol{p}|$ символов в выражении $\boldsymbol{p}$ удовлетворяет неравенству $|p| \leqslant 6m$.
3. Основываясь на предыдущем упражнении, докажите неравенства (11.2). Можно ли изменить правила П.1.–П.4. так, чтобы уменьшить правые части этих неравенств?
4. Для автомата NDFA, показанного на рис. 11.3, докажите равенство $\mathit{trans}(\{5\}, \varepsilon) = \{2, 4, 7\}$. Правильно ли, что в последнее множество не включены состояния 1 и 6?
5. На рис. 11.3 видно, что NDFA может содержать циклы. Несмотря на это докажите, что $s \notin \mathit{trans}(s, \varepsilon)$ для любого состояния s, не являющегося начальным. Справедливо ли такое же утверждение для начального состояния?
6. Запишите алгоритм для функции $\mathit{trans}(S, \lambda)$ и докажите его корректность. Обратите особое внимание на случай, когда вторым аргументом функции будет пустая строка ε!
7. Покажите на примере, что если выражение $(\lambda_1 \mid \lambda_2 \mid \cdots \mid \lambda_\alpha)^*$ *не предшествует* регулярному выражению $\boldsymbol{p}$, то алгоритм 11.1.1, работающий с NDFA, построенным без учета этого выражения, может не обнаружить совпадения некоторых подстрок строки $\boldsymbol{x}$ с паттерном $\boldsymbol{p}$.
8. Докажите корректность алгоритма 11.1.1.

9. Докажите, что в синтаксическом дереве, соответствующем некоторому регулярному выражению, понятия "внутренний узел" и "узел, имеющий метку в виде метасимвола", эквивалентны.

10. Постройте синтаксические деревья и автоматы NDFA и DFA для следующих регулярных выражений.

- $\boldsymbol{p} = (a \mid b)^* aba$;
- $\boldsymbol{p} = ab^{m-1}$;
- $\boldsymbol{p} = a(a \mid b)(a \mid b)$;
- $\boldsymbol{p} = a(a \mid b \mid c)(a \mid b \mid c)$.

Основываясь на последних двух примерах, подсчитайте количество вершин в DFA($\boldsymbol{p}$), если $\boldsymbol{p} = \lambda_1(\lambda_1 \mid \lambda_2 \mid \cdots \mid \lambda_\alpha)^{(m-1)/\alpha}$. Сколько поглощающих состояний в этом DFA?

11. Докажите, что процесс преобразования NDFA в DFA должен обязательно завершиться.

12. Докажите, что в любом автомате DFA имеется хотя бы одно поглощающее состояние.

13. Покажите, что метки вершин в DFA совпадают с метками только тех вершин NDFA, которые имеют непустые исходящие дуги. Это утверждение не относится к вершинам DFA, соответствующим поглощающим состояниям. На основе этого утверждения докажите, что максимально возможное число вершин в DFA равно 2^{m+1}.

14. Запишите алгоритм преобразования NDFA в DFA.

15. Запишите алгоритм DFA нахождения всех совпадений подстрок строки $\boldsymbol{x}$ с регулярным выражением $\boldsymbol{p}$ с временем выполнения порядка $\Theta(n)$ для алфавита, который

- упорядочен, но не индексирован;
- индексирован.

Для этого измените функцию *trans* так, чтобы она для заданного идентификатора S вершины и буквы λ вычисляла следующую вершину $\mathit{trans}(S, \lambda)$.

16. Запишите алгоритм выполнения предварительного этапа алгоритма BM*.

11.2 Алгоритмы сравнения с множественными паттернами

В этом разделе рассмотрим решение задачи сравнения заданной текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ с множественным паттерном $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$. Поскольку

множественный паттерн P можно записать в форме $\boldsymbol{p}_1 \mid \boldsymbol{p}_2 \mid \cdots \mid \boldsymbol{p}_r$, то данную задачу можно рассматривать как задачу сравнения с регулярными выражениями специального типа; алгоритмы для решения этой задачи представлены в предыдущем разделе. В этом разделе мы опишем более эффективные алгоритмы решения данной задачи, которые будут учитывать специфику множественных паттернов.

Алгоритмы для точного сравнения с паттерном описаны в подразделах 11.2.1 и 11.2.2. Это модификации алгоритма КМП (алгоритм Кнута–Морриса–Пратта, раздел 7.1) и алгоритма БМ (алгоритм Бойера–Мура, раздел 7.2). В двух последних подразделах описаны алгоритмы для приближенного сравнения с множественными паттернами: в подразделе 11.2.3 представлена модификация алгоритма BM (алгоритм Ву–Менбера, раздел 10.4), а в подразделе 11.2.4 — новый (в этой книге) алгоритм Бейза-Ятса–Наварро (Baeza-Yates–Navarro, сокращенно — алгоритм BYN). Новая модификация алгоритма BM применима к функциям расстояния любых типов, а алгоритм BYN — только для расстояний преобразования, Левенштейна и Хемминга.

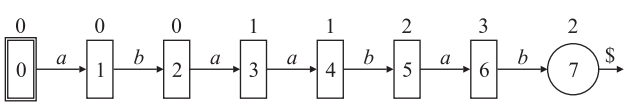
11.2.1 Автомат Ахо–Корасика: модификация алгоритма КМП

Теперь, когда мы познакомились с конечными автоматами, читателю должно быть ясно, что алгоритм КМП можно представить в виде конечного автомата. Действительно, в алгоритме КМП сначала сравнивается очередной входной символ, и если такого совпадения не наблюдается, тогда выполняется сравнение с символами наибольшей грани, соответствующей текущей входной позиции. Другими словами, мы уже имеем конечный автомат, построенный по правилам алгоритма КМП построения массива граней.

Чтобы понять "природу" конечного автомата Ахо–Корасика (сокращенно, ACFA — Aho–Corasick finite automaton) [3], рассмотрим небольшой пример. Предположим сначала, что имеем простой паттерн $\boldsymbol{p} = abaabab$, который сравнивается с текстовой строкой $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Для данного паттерна $\boldsymbol{p}$ получаем следующий массив граней (раздел 1.3):

	1	2	3	4	5	6	7
$\boldsymbol{p} =$	a	b	a	a	b	a	b
$\boldsymbol{\beta} =$	0	0	1	1	2	3	2

Тогда автомат ACFA($\boldsymbol{p}$) будет иметь вид



В ACFA($\boldsymbol{p}$) метки *внутри* вершин обозначают позиции символов в паттерне $\boldsymbol{p}$, метки на дугах — символы паттерна, определяющие переход к следующей вершине (состоянию). Вспоминая определение деревьев суффиксов в разделе 2.1, здесь мы видим, что ACFA($\boldsymbol{p}$) является ничем иным, как некомпактным синтаксическим деревом для паттерна $\boldsymbol{p}$. Добавим в это дерево метки *над* вершинами, которые соответствуют элементам массива граней и указывают вершину (состояние), в которую должен перейти автомат в случае *несовпадения* входного символа с меткой исходящей дуги. (С этой точки зрения массив граней можно назвать "функцией несовпадения"!) Таким образом, эти метки в действительности определяют "обратные" дуги (переходы) от текущей вершины (состояния) в одну из предыдущих вершин, что соответствует "текущему" значению массива граней. В случае несовпадения в начальной вершине (вершине 0) для определенности положим "возврат" в эту же вершину.

Как и в случае деревьев суффиксов, примем соглашение, что в ACFA($\boldsymbol{p}$) исходящие дуги конечных вершин помечены специальным символом \$, который не может совпадать ни с одной буквой входной строки. Это обеспечит корректное поведение автомата после установления факта совпадения входной строки и паттерна.

В общем случае, поскольку массив граней можно вычислить за линейное время (теоремы 1.3.2 и 1.3.3), автомат ACFA($\boldsymbol{p}$) для любого одиночного паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ можно построить за время порядка $\Theta(m)$ с использованием памяти объемом $\Theta(m)$. Когда такой автомат построен, алгоритм ACFA (представленный ниже как алгоритм 11.2.1) без труда найдет все совпадения паттерна $\boldsymbol{p}$ и строки $\boldsymbol{x}$. Алгоритм ACFA можно рассматривать как вариант алгоритма КМП, однако на самом деле он является зеркальным отражением алгоритма 1.3.1 вычисления массива граней.

Алгоритм 11.2.1. (Алгоритм ACFA)

▷ *Вычисление позиций i в строке $\boldsymbol{x}$, таких, что подстрока $\boldsymbol{x}[i'..i]$ в ACFA*
▷ *определяет путь от состояния 0 до поглощающего состояния*
$s \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  while $s > 0$ **and not** $match(s, \boldsymbol{x}[i])$ **do**
    $s \leftarrow \boldsymbol{\beta}[s]$
  if $match(s, \boldsymbol{x}[i])$ **then**
    $s \leftarrow s + inc(s, \boldsymbol{x}[i])$
    if $accept(s)$ **then output** (i, s)

Выполнение алгоритма ACFA зависит от четырех типов вычислений.

- Массив граней $\boldsymbol{\beta}$ вычисляется на предварительном этапе построения автомата NDFA. Поэтому для любого состояния s значение $\boldsymbol{\beta}[s]$ можно получить за константное время. Напомним, что мы положили $\boldsymbol{\beta}[0] = 0$.
- В случае обнаружения совпадения текущее значение s должно возрасти на величину, определяемую буквой $\boldsymbol{x}[i]$, для которой зафиксировано совпадение. В нашем простом примере шаг возрастания значения s всегда равен 1, однако для множественных паттернов шаг уже не будет постоянным. Значение шага будет храниться в метке исходящей из вершины s дуги, которая соответствует букве $\boldsymbol{x}[i]$. В этом случае значение шага можно получить за константное время с помощью функции inc.
- Каждое состояние содержит бит, который указывает, является ли данное состояние поглощающим. Значение этого бита возвращает функция $accept$.
- Функция $match$ возвращает значение истины тогда, когда имеет место совпадение метки на дуге, исходящей из вершины s, и буквы $\boldsymbol{x}[i]$. В нашем примере с одним паттерном эта функция выполняет одно буквенное сравнение и поэтому вычисляется за константное время. Однако в случае множественных паттернов, как мы увидим далее, вычисление функции $match$ усложняется.

Расширим приведенный выше пример и рассмотрим множественный паттерн, состоящий из трех простых паттернов:

$$P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\} = \{abaabab, abab, ababcabab\}. \tag{11.5}$$

Применяя к этому паттерну ту же методологию построения некомпактного синтаксического дерева, получим автомат ACFA(P), показанный на рис. 11.4. Отметим, что общее количество *всех* дуг в ACFA(P) (т.е. включая как явно показанные на рисунке дуги, так и "обратные" дуги, определяемые метками над вершинами) не превышает величины $2m + 1$, где m — сумма длин частных паттернов в множественном паттерне P.

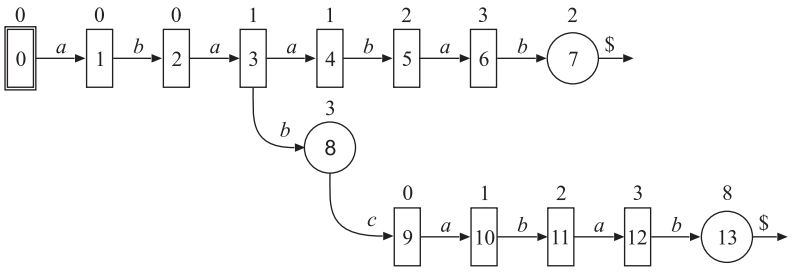


Рис. 11.4. Автомат ACFA(P) для паттерна (11.5)

На этом примере очевидна необходимость внесения изменений в процесс вычисления обычного массива граней для того, чтобы пересчитать перемещение некоторых позиций в паттернах с учетом номеров (меток) состояний автомата ACFA(P). Так позиции 4 в паттернах $abab$ и $abaabab$ соответствует состояние 8, позиции 9 в паттерне $ababcabab$ — состояние 13, а $\boldsymbol{\beta}[13] = 8$. Оставим подробности этого процесса для упражнения 11.2.2. В этом же упражнении рассмотрен случай, когда один паттерн является гранью другого паттерна, как и в паттерне (11.5).

Имея массив $\boldsymbol{\beta}$, корректно вычисленный для автомата ACFA(P), нетрудно показать, что алгоритм 11.2.1 корректно определяет все подстроки строки $\boldsymbol{x}$, совпадающие с паттернами множественного паттерна P. Цикл **while** гарантирует, что в случае обнаружения несовпадения с некоторым префиксом $\boldsymbol{p}[1..j]$ некоторого паттерна $\boldsymbol{p}$ далее сравнения с гранями подстроки $\boldsymbol{p}[1..j-1]$ выполняются в убывающем порядке длин этих граней, точно так же как в алгоритме КМП. Таким образом, корректность алгоритма ACFA для множественных паттернов следует непосредственно из корректности алгоритма КМП. Более того, если функцию $match$ можно вычислить за константное время, тогда время выполнения алгоритма 11.2.1 будет пропорциональным величине n. Это утверждение вытекает из того факта, что общее количество итераций цикла **while**, просуммированное по всем n этапам алгоритма, не может превышать величины $n-1$. Такой же факт имеет место для цикла **while** процесса вычисления массива граней (см. раздел 1.3)! Таким образом, предполагая, что функция $match$ вычисляется за константное время, линейность времени выполнения алгоритма 11.2.1 доказывается на основе тех же аргументов, что и утверждение о линейности времени выполнения алгоритма вычисления массива граней (теорема 1.3.3).

Что же на самом деле можно сказать о времени вычисления функции $match$? В общем случае автомат ACFA(P) является (слегка приукрашенным) некомпактным синтаксическим деревом, построенным на множестве $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ различных строк длиной m_1, m_2, ..., m_r соответственно. (Общую длину этих строк обозначим m, т.е. $m = m_1 + m_2 + \cdots + m_r$.) Если алфавит имеет размер α, то в ACFA(P) может быть несколько вершин, имеющих $O(\alpha)$ исходящих дуг. Поэтому для корректного выбора исходящей дуги в каждой вершине необходимо хранить данные объемом $O(\alpha)$. Тогда выбрать правильную дугу можно за константное время, если алфавит индексирован, либо за время порядка $O(\log \alpha)$, если алфавит упорядочен, но не индексирован. Подобная ситуация для деревьев суффиксов исследована в обсуждении 2.1.1.

Даже когда частные паттерны длинные, а их количество относительно невелико, процесс корректного выбора дуги может быть трудоемким. Например, предположим, что $r = \alpha$ и паттерны $\boldsymbol{p}_h$ имеют префиксы λ_h^2, $h = 1, 2, \ldots, \alpha$. Тогда в ACFA(P) вершина 0 имеет α исходящих дуг. Пусть в строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$ нет квадратов отдельных букв, а любая буква алфавита встречается n/α раз. При сравнении этой строки с паттерном P вершина 0 будет посещаться n раз, и при

каждом ее посещении необходимо выбрать одну дугу среди α дуг. Если алфавит не индексирован, то для определения нужной дуги необходимо использовать или дерево поиска, или массив соответствия. Таким образом, только обработка вершины 0 потребует времени порядка $O(n \log \alpha)$.

Наши рассуждения обобщим в виде следующей теоремы.

Теорема 11.2.1. Пусть $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ — множество паттернов общей длиной m, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ — заданная текстовая строка, строка $\boldsymbol{x}$ и паттерны P определены на алфавите размером α. Тогда для хранения автомата ACFA(P) необходима память объемом $\Theta(\alpha m)$ и

- для индексированного алфавита A автомат ACFA(P) строится за время порядка $\Theta(m)$, а алгоритм 11.2.1 выполняется за время порядка $\Theta(n)$;
- для упорядоченного, но не индексированного алфавита A автомат ACFA(P) строится за время порядка $\Theta(m \log \alpha)$, а алгоритм 11.2.1 выполняется за время порядка $\Theta(n \log \alpha)$. ■

Таким образом, автомат Ахо–Корасика позволяет сравнить текстовую строку $\boldsymbol{x}[1..n]$ с множеством паттернов общей длиной m за время порядка $\Theta(m+n)$ для индексированных алфавитов и за время порядка $\Theta((m+n)\log \alpha)$ для упорядоченных алфавитов. Другими словами, алгоритм КМП можно распространить на случай множественных паттернов практически без потери эффективности вычислений.

Если в алгоритме ACFA использовать ориентированные ациклические графы слов (см. раздел 5.3), тогда этот алгоритм в самом худшем случае потребует выполнения не более $2n$ буквенных сравнений. Такая модификация алгоритма ACFA наиболее эффективна тогда, когда размер алфавита небольшой, а все частные паттерны длинны.

11.2.2 Автомат Комменца–Вальтера: модификация алгоритма БМ

Так же как алгоритм ACFA является модификацией алгоритма КМП для случая множественных паттернов, так и алгоритм CWFA (Commentz–Walter Finite Automata — конечный автомат Комменца–Вальтера) является модифицированным вариантом алгоритма БМ [57]. В алгоритме CWFA существенно использование тех же массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$, что и в алгоритме БМ (раздел 7.2). Различие в использовании этих массивов заключается только в том, что в алгоритме CWFA сдвиги в этих массивах вычисляются как соответствующие минимумы по позициям во *всех* паттернах множественного паттерна P, а не как минимумы по позициям одного паттерна. Чтобы сделать это различие более ясным, рассмотрим автомат CWFA(P) для множественного паттерна (11.5), введенного в предыдущем под-

разделе. Как показано на рис. 11.5, состояния в автомате CWFA соответствуют позициям в строках, записанным в обратном порядке. Такой конечный автомат соответствует алгоритму БМ, в котором сравнение букв текстовой строки с буквами паттерна происходит при перемещении по паттерну справа налево. Напомним, что в алгоритме БМ несовпадение букв в процессе сравнения требует вычисления нового значения i и последующего сравнения буквы $\boldsymbol{x}[i]$ с самым правым символом паттерна — на рис. 11.5 это соответствуют возврату в вершину 0 после каждого несовпадения, установленного в процессе сравнения.

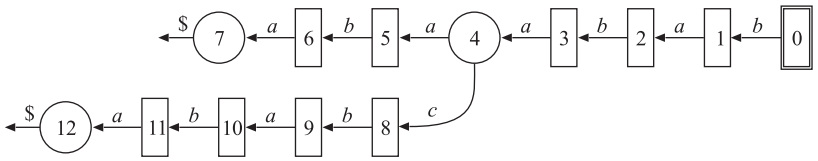


Рис. 11.5. Автомат CWFA(P) для паттерна (11.5)

Предположим, что на вход автомата CWFA(P), показанного на рис. 11.5, подается текстовая строка $\boldsymbol{x} = \cdots b\underline{bab}\cdots$. (Здесь подчеркивание указывает на буквы, совпадающие с паттерном.) Для определенности положим, что $\boldsymbol{x}[i'+1..i'+3] = bab$. Тогда на дуге, исходящей из вершины 3, будет иметь место несовпадение $\boldsymbol{x}[i'] = b \neq a$. Так как имеет место несовпадение, в стандартном алгоритме БМ вычисляются $\boldsymbol{\delta}_2$-сдвиги, отдельно для каждого паттерна

$$abab, abaabab, ababcabab \tag{11.6}$$

из множественного паттерна P. Получим следующие значения для новой позиции i: $i = i' + 5$, $i = i' + 8$, $i = i' + 9$, каждое из которых соответствует "своему" паттерну из (11.6). В алгоритме CWFA выбирается минимальный сдвиг, т.е. новой позицией будет $i' + 5$. Аналогично входная буква c в вершине 0 определяет $\boldsymbol{\delta}_1$-сдвиги $i = i' + 4$, $i = i' + 6$, $i = i' + 4$, соответствующие паттернам (11.6), среди которых выбирается минимальное значение $i = i' + 4$.

Рассмотрим общий случай, когда множественный паттерн $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ задан как множество r частных паттернов длиной соответственно $m_1, m_2, \ldots, m_r$, общая длина этих паттернов равна $m = m_1 + m_2 + \cdots + m_r$. Значения массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$ вычисляются отдельно для каждого паттерна из P, затем определяется минимум по r паттернам. Это означает, что для каждого состояния s находится минимум по не более чем r паттернам, которые имеют общий префикс со строкой, определяемой путем, ведущим к s. Поскольку значения массивов $\boldsymbol{\delta}_1$ и $\boldsymbol{\delta}_2$ для одного паттерна можно вычислить за время, пропорциональное длине паттерна (см. упражнение 7.2.4 и теорему 7.2.2), то сдвиги для множественного паттерна P

можно вычислить за время порядка $O(m)$. Технические детали этого процесса мы опускаем.

Далее, в алгоритме БМ процедуру $(i, j) \leftarrow hctam(i, m)$ необходимо заменить новой процедурой $(i, s) \leftarrow hctam(i, 0)$, которая, увеличивая значение i, осуществляет перемещение по пути от вершины 0 до некоторой вершины s, в которой будет иметь место несовпадение (такое несовпадение обязательно будет хотя бы потому, что исходящие дуги конечных вершин помечены специальным символом \$, которого по условию нет в текстовых строках). Функции $hctam$ мы также делегируем операцию вывода пары значений (i, s) в том случае, когда будет достигнуто поглощающее состояние s (т.е. когда будет достигнуто состояние, для которого $accept(s) = 0$). Таким образом, алгоритм CWFA (представлен ниже как алгоритм 11.2.2) формально почти полностью идентичен алгоритму БМ.

Алгоритм 11.2.2 (Алгоритм CWFA)

▷ *Вычисление всех позиций i в строке $\boldsymbol{x}$, таких, что*
▷ *подстрока $\boldsymbol{x}[i..i']$ в автомате CWFA(P) определяет путь*
▷ *от состояния 0 до некоторого поглощающего состояния,*
▷ *при этом $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$*
$i \leftarrow \min\{m_1, m_2, \ldots, m_r\}$
while $i \leqslant n$ **do**
    $(i, s) \leftarrow hctam(i, 0)$
    $i \leftarrow i + \max\{\boldsymbol{\delta}_1[\boldsymbol{x}[i], s], \boldsymbol{\delta}_2[s]\}$

Отметим, что поскольку $\boldsymbol{\delta}_2$-сдвиг положителен для любого паттерна $\boldsymbol{p}$, то значение i возрастает на каждом шаге алгоритма не менее, чем на 1. Это гарантирует завершение выполнения алгоритма 11.2.2.

Подобно автомату ACFA(P), автомат CWFA(P) требует для хранения памяти объемом $\Theta(\alpha m)$ и может быть построен за время $\Theta(m)$, если алфавит индексирован, или за время $\Theta(m \log \alpha)$, если алфавит только упорядочен. Как и алгоритм БМ, алгоритм CWFA имеет сложности при работе с периодическими паттернами, поэтому в самом худшем случае его время выполнения может иметь порядок $\Theta(mn)$. Как показано в [2], алгоритм CWFA в среднем работает быстрее, чем алгоритм ACFA, в том случае, если количество r частных паттернов невелико, и медленнее в случае большого значения r.

11.2.3 Аппроксимирующие паттерны: модификация алгоритма BM

Рассмотрим задачу k-приближенного сравнения заданной текстовой строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ с множеством паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$, которые имеют длины

соответственно $m_1, m_2, \ldots, m_r$. В этом подразделе опишем модификацию алгоритма Ву–Менбера (алгоритм BM), который эффективно решает эту задачу [233].

Основная идея модификации алгоритма BM — обрабатывать множественный паттерн $\boldsymbol{p} = \boldsymbol{p}_1\boldsymbol{p}_2 \ldots \boldsymbol{p}_r$ как единый паттерн длиной $m = m_1 + m_2 + \cdots + m_r$, при этом, конечно, необходимы средства от возможных ошибок, связанных с границами паттернов.

Напомним, что в алгоритме BM для каждого $q \in 0..k$ и $i \in 1..n$ вычисляется бинарный вектор $\boldsymbol{s}_q^{(i)}$, где для всех $j \in 1..m$

$$\boldsymbol{s}_q^{(i)}[j] = 0 \Leftrightarrow \boldsymbol{c}[i, j] \leqslant q,$$

а $\boldsymbol{c}[i, j]$ — элемент матрицы стоимостей (раздел 10.1), вычисляемый для подстроки $\boldsymbol{x}[i'..i]$ и префикса $\boldsymbol{p}[1..j]$ паттерна. В алгоритме BM также используется бинарный массив $\boldsymbol{t}$, где для каждого $h \in 0..\alpha$ и $j \in 1..m$

$$\boldsymbol{t}[h, j] = 0 \Leftrightarrow \boldsymbol{p}[j] = h.$$

Для того чтобы определить границы между паттернами, в модифицированном алгоритме применим шаблон

$$M = 01^{m_1-1}01^{m_2-1} \cdots 01^{m_r-1}$$

длиной m, который будет использоваться для вычисления

$$\boldsymbol{t}'[h, 1..m] = \boldsymbol{t}[h, 1..m] \vee M,$$

где, как и ранее (раздел 10.4), $\vee$ обозначает операцию логического сложения OR. Тогда $\boldsymbol{t}'[h, j] = 0$ только в том случае, когда j будет первой позицией одного из паттернов $\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r$ и при этом $\boldsymbol{p}[j] = h$.

Первый бинарный вектор $\boldsymbol{s}_0^{(i)}[j]$ вычисляется по стандартным формулам (10.37):

$$\boldsymbol{s}_q^{(i)}[j] \leftarrow \boldsymbol{s}_{q-I}^{(i-1)}[j] \wedge \boldsymbol{s}_{q-D}^{(i)}[j-1] \wedge \boldsymbol{s}_{q-S}^{(i-1)}[j-1] \wedge (\boldsymbol{s}_q^{(i-1)}[j-1] \vee \boldsymbol{t}[x[i], j]).$$

При $q = 0$ эти вычисления корректны для всех позиций j, за исключением тех позиций j_0, с которых начинаются частные паттерны. Чтобы избежать ошибок в этих позициях, необходимо положить $\boldsymbol{s}_0^{(i)}[j_0] \leftarrow 0$, если $\boldsymbol{x}[i] = \boldsymbol{p}[j_0]$, и $\boldsymbol{s}_0^{(i)}[j_0] \leftarrow 1$ в противном случае. Добиться такого результата можно с помощью вычисления $\boldsymbol{s}_0^{(i)}[j] \wedge \boldsymbol{t}'[\boldsymbol{x}[i], j]$ для всех $j \in 1..m$, где $\wedge$ обозначает операцию логического умножения AND. В результате таких вычислений битовые значения в позициях, не совпадающих с начальными позициями паттернов, не изменятся, а значения в начальных позициях паттернов изменятся (с 1 на 0) только в том случае, если $\boldsymbol{t}'[\boldsymbol{x}[i], j_0] = 0$.

При $q = 1$, чтобы избежать подобных ошибок в процессе вычисления $\boldsymbol{s}_1^{(i)}[j]$, необходимо положить $\boldsymbol{s}_1^{(i)}[j_0] \leftarrow 0$ для начальных позиций j_0 всех паттернов из P. (Здесь просматривается аналогия со сдвигом 0 в $\boldsymbol{s}_1^{(i)}[1]$, выполняемым процедурой *rightshift* в формуле (10.38) исходного алгоритма BM.) Итак, после вычислений по формуле (10.37) для всех $j \in 1..m$ следует выполнить дополнительные вычисления

$$\boldsymbol{s}_1^{(i)}[j] \leftarrow \boldsymbol{s}_1^{(i)}[j] \wedge M_1,$$

где $M_1 = M$. Эти дополнительные вычисления гарантируют, что первые биты для каждого частного паттерна будут равны 0, тогда как значения остальных битов останутся вычисленными по формуле (10.37).

В общем случае после вычисления по формуле (10.37) необходимо выполнить дополнительные вычисления

$$\boldsymbol{s}_q^{(i)}[j] \leftarrow \boldsymbol{s}_q^{(i)}[j] \wedge M_q,$$

где $M_q = 0^q 1^{m_1 - q} 0^q 1^{m_2 - q} \cdots 0^q 1^{m_r - q}$.

Дополнительные вычисления, необходимые для коррекции векторов $\boldsymbol{s}_q^{(i)}$, можно выполнить за константное время, и поскольку эти дополнительные вычисления заключаются в выполнении логических операций, то их можно реализовать применительно к целым машинным словам, а не к отдельным битам, как показано здесь. Следовательно, сложность модифицированного алгоритма BM не увеличилась, и мы имеем право сформулировать теорему, которая является аналогом теоремы 10.4.1.

Теорема 11.2.2. Пусть заданы текстовая строка $\boldsymbol{x}[1..n]$ и множество паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$, которые определены на индексированном алфавите размером α. Обозначим $m = |\boldsymbol{p}_1| + |\boldsymbol{p}_2| + \cdots + |\boldsymbol{p}_r|$. Тогда алгоритм BM вычисляет все k-приближенные совпадения множественного паттерна P и строки $\boldsymbol{x}$ за время порядка $\Theta(k\lceil m/w \rceil n)$ с использованием памяти объемом порядка $\Theta((\alpha + k)\lceil m/w \rceil)$, где w — длина машинного слова. ■

В работе [186] описан алгоритм сравнения с множественным аппроксимирующим паттерном, в котором скомбинированы бинарное отображение и методы хеширования. Этот алгоритм, по-видимому, будет очень быстрым, однако он применим только для расстояния преобразования и только для аппроксимационной константы k, равной 1.

11.2.4 Аппроксимирующие паттерны: алгоритм Бейза-Ятса–Навarро

Мы закончим этот раздел и данную главу описанием еще одного алгоритма, использующего бинарные операции, который первоначально предлагался автора-

ми Бейзом-Ятсом и Наварро (Baeza-Yates, Navarro) для решения задачи приближенного сравнения с одним паттерном [26] и только позже был модифицирован для приближенного сравнения с множественными паттернами [27]. Этот алгоритм (назовем его алгоритмом BYN — от Baeza-Yates–Navarro), примененный к множественным паттернам, является примером "фильтрационного" алгоритма, о котором упоминалось в подразделе 10.3.2. В таких алгоритмах сначала очень быстро определяется *сверхмножество* подстрок текстовой строки, которые приближенно совпадают хотя бы с одним паттерном, затем из этого множества с помощью также очень быстрой технологии отфильтровываются все ложные решения. Подобно всем алгоритмам, основанным на бинарных операциях, в алгоритме BYN предполагается, что текстовые строки и паттерны определены на индексированном алгоритме 1, 2, ..., α (раздел 4.1).

Описание алгоритма BYN начнем со случая приближенного сравнения с одним паттерном, где будет использоваться расстояние преобразования (т.е. будет решаться задача k-разностей, описанная в разделе 10.3). Затем покажем, как этот алгоритм можно применить к множественным паттернам. Важно отметить, что методология алгоритма BYN не ограничивается только расстоянием преобразования. Как будет показано, этот алгоритм можно представить в виде модели конечного автомата, в котором будут "видны" операции вставки, удаления и подстановки. Поэтому данный алгоритм после соответствующей "обработки" этих операций будет применим и к расстояниям Левенштейна и Хемминга.

Автомат BYNFA($\boldsymbol{p}, \boldsymbol{k}$)

Алгоритм BYN для k-приближенного сравнения заданной строки $\boldsymbol{x}$ и одного паттерна $\boldsymbol{p}$ можно представить в виде конечного автомата "нестандартного" вида — назовем этот автомат BYNFA($\boldsymbol{p}, k$). Как показано на рис. 11.6, автомат BYNFA для паттерна $\boldsymbol{p} = this$ и аппроксимационной константы $k = 2$ представим в виде прямоугольной решетки с $k+1$ строкой и $m+1$ столбцом (m — длина паттерна). Без потери общности можно считать, что $m \geqslant k$. Строка h ($h \in 0..k$) решетки соответствует h-разностям между входной текстовой строкой и паттерном (подсчитанным на основе расстояния преобразования). Столбец j ($j \in 0..m$) определяет приближенное сравнение с префиксом $\boldsymbol{p}[1..j]$ паттерна. Пересечения (h, j) строк и столбцов решетки интерпретируются как состояния автомата. Если состояние (h, j) ***активно***, тогда имеет место h-приближенное совпадение между $\boldsymbol{p}[1..j]$ и подстрокой $\boldsymbol{x}[1..i]$ (i — текущая позиция в строке $\boldsymbol{x}$). Поэтому первоначально все состояния в столбце $j = 0$ будут активными и, кроме того, все состояния (h, j), для которых $0 \leqslant j \leqslant h \leqslant k$, также будут активны, что соответствует h-несовпадениям между первыми k позициями паттерна и пустыми префиксами строки $\boldsymbol{x}$. Первоначально активные состояния на рис. 11.6 помечены меткой 0.

Как и в других конечных автоматах, изученных нами, здесь дуги определяют переходы из одного состояния в другое, определяемое текущей (i-й) входной

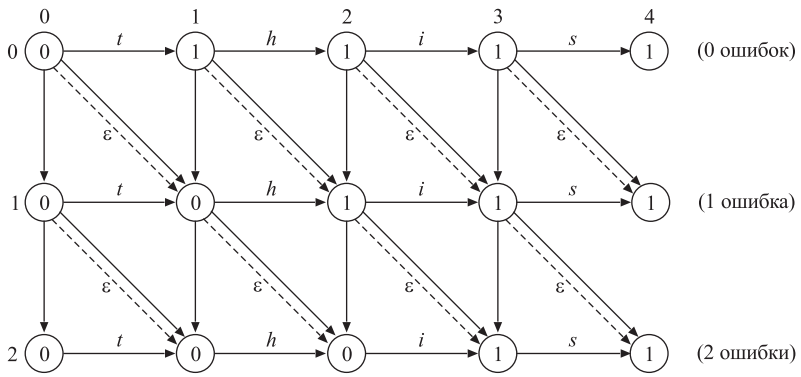


Рис. 11.6. Автомат BYNFA$(this, 2)$

буквой строки $\boldsymbol{x}$. Состояния будут активными или неактивными в соответствии со следующими правилами.

- Первоначально активное состояние остается таковым, если из этого состояния начинается сравнение с любой позицией текстовой строки.
- Любое состояние, достижимое из другого активного состояния в соответствии с текущей буквой строки $\boldsymbol{x}$, становится активным.
- Любое состояние, не являющееся начальным и не достижимое в соответствии с текущей буквой строки $\boldsymbol{x}$, становится неактивным.

В автомате BYNFA все переходы между состояниями можно отнести к одному из четырех типов. Один тип переходов оставляет без изменения уровень ошибки h, а три других типа увеличивают этот уровень на единицу.

1. Горизонтальный переход $(h, j-1) \rightarrow (h, j)$ соответствует совпадению буквы $\boldsymbol{x}[i]$ и буквы $\boldsymbol{p}[j]$, которой помечен этот переход. Значение h остается неизменным, поскольку новая буква $\boldsymbol{x}[i]$ не увеличивает текущую разность между $\boldsymbol{x}$ и $\boldsymbol{p}$, вычисляемую на основе расстояния преобразования.
2. Вертикальный переход $(h-1, j) \rightarrow (h, j)$ становится возможным тогда, когда для достижения совпадения текущую букву $\boldsymbol{x}[i]$ необходимо вставить в паттерн $\boldsymbol{p}$ в позицию j. При этом позиция j паттерна не изменяется, а значение разности h увеличивается на единицу.
3. Диагональный переход $(h-1, j-1) \rightarrow (h, j)$ по сплошной линии соответствует подстановке вместо $\boldsymbol{p}[j]$ буквы $\boldsymbol{x}[i]$, при этом позиции в паттерне $\boldsymbol{p}$ и строке $\boldsymbol{x}$ сдвигаются на одну позицию вправо, а значение разности h увеличивается на единицу.
4. Диагональный переход $(h-1, j-1) \rightarrow (h, j)$ по пунктирной линии (помеченной ε) становится возможным тогда, когда для достижения совпадения

текущую букву $\boldsymbol{p}[j]$ можно удалить из паттерна. Это "пустой" или автоматический переход, поскольку здесь удаляется буква из паттерна без каких-либо изменений в текстовой строке $\boldsymbol{x}$.

Отметим, что пустые диагональные переходы гарантируют активность состояний $(h+q, j+q)$ для всех целых $q \in 0..k-h$, если будет активно состояние (h, j). И наоборот, состояние (h, j) становится активным, если активно какое-либо состояние $(h-q, m-q)$, $0 \leqslant q \leqslant k$. Таким образом, активность состояния (h, m) указывает на то, что имеет место h-приближенное совпадение $\boldsymbol{p}[1..j]$ с некоторой подстрокой строки $\boldsymbol{x}$, заканчивающейся в текущей позиции i.

Чтобы лучше понять операции, выполняемые автоматом BYNFA, рассмотрим два варианта, когда на вход автомата BYNFA$(this, 2)$ поступают строки $\boldsymbol{x} = = hash$ и $\boldsymbol{x} = whist$. В табл. 11.1 и 11.2 представлены результаты обработки этих строк; здесь показаны состояния, которые становятся активными после обработки каждой входной буквы (но сюда не включены начальные активные состояния). Отметим, что пара чисел (i, h) будет выходным результатом тогда, когда существует наименьшее целое число h, такое, что для входной буквы $\boldsymbol{x}[i]$ становится активным состояние (h, m).

Таблица 11.1. Обработка автоматом BYNFA$(this, 2)$ строки $\boldsymbol{x} = hash$

i	$\boldsymbol{x}[i]$	Активные состояния	Совпадения (i, h)
1	h	(1, 2), (2, 3)	
2	a	(2, 3)	
3	s	(2, 4)	(3, 2)
4	h	(1, 2), (2, 3)	

Таблица 11.2. Обработка автоматом BYNFA$(this, 2)$ строки $\boldsymbol{x} = whist$

i	$\boldsymbol{x}[i]$	Активные состояния	Совпадения (i, h)
1	w		
2	h	(1, 2), (2, 3)	
3	i	(1, 3), (2, 3), (2, 4)	(3, 2)
4	s	(1, 4), (2, 4)	(4, 1)
5	t	(0, 1), (1, 2), (2, 3)	

Компактное представление автомата BYNFA

Нетрудно заметить, что для организации работы автомата BYNFA$(\boldsymbol{p}, k)$ необходимо отслеживать изменение его состояний. Для этих целей введем бинарный

массив $A = A[0..k, 0..m]$, в котором для всех $h \in 0..k$ и $j \in 0..mA[h, j] = 0$ только в том случае, если состояние (h, j) активно, и $A[h, j] = 1$ в противном случае. Сначала положим $A[h, j] = 0$ для всех h и j, таких, что $0 \leqslant j \leqslant h \leqslant k$. Обозначим через $A^{(0)}$ начальное состояние автомата BYNFA, а через $A^{(i)}$ $(i = 1, 2, \ldots, n)$ — последующие состояния этого автомата, соответствующие входным буквам $\boldsymbol{x}[1]$, $\boldsymbol{x}[2]$, $\ldots$, $\boldsymbol{x}[n]$.

Если известны состояния BYNFA для некоторого значения $i - 1$, тогда для следующего значения i и для всех h и j, таких, что $0 \leqslant h \leqslant j \leqslant m$, новые состояния пересчитываются по формуле

$$A^{(i)}[h, j] \leftarrow (A^{(i-1)}[h, j-1] \vee \boldsymbol{t}[\boldsymbol{x}[i], j]) \wedge A^{(i-1)}[h-1, j] \wedge \\ \wedge A^{(i-1)}[h-1, j-1] \wedge A^{(i)}[h-1, j-1], \tag{11.7}$$

где

- бинарный массив $\boldsymbol{t} = \boldsymbol{t}[1..\alpha, 1..m]$ вычисляется заранее точно так, как в алгоритмах ДБГ (раздел 7.4) или BM (подраздел 11.2.3), т.е. для каждой буквы $r \in 1..\alpha$ и для всех $j \in 1..m\boldsymbol{t}[r, j] = 0$ только в том случае, если $\boldsymbol{p}[j] = r$;
- символ $\vee$ обозначает операцию логического сложения OR (результатом этой операции будет значение 0 только тогда, когда *оба* операнда равны 0);
- символ $\wedge$ обозначает операцию логического умножения AND (результатом этой операции будет значение 0 тогда, когда *хотя бы один* операнд равен 0).

Отметим, что для корректного пересчета $A^{(i)}[h, j]$ на основе значения $A^{(i)}[h-1, j-1]$, стоящего на той же диагонали решетки, значения h и j должны идти в возрастающем порядке.

Чтобы при пересчете состояний автомата BYNFA можно было использовать логические операции, выполняемые параллельно над всеми битами машинного слова, необходимо выполнить некоторые условия. Так для хранения состояний используем массив A, состоящий из $(k+1)(m+1)$ бит. Но, как показано в упражнении 11.2.9, для каждой позиции i строки $\boldsymbol{x}$ в этом массиве необходимо пересчитывать только $(k+1)(2m-k)/2 \in \Theta(mk)$ бит. Кроме того, как показано выше, наличие пустых переходов позволяет установить активные состояния, просто перемещаясь по диагоналям решетки от некоторого активного состояния. Поэтому для пересчета состояний на диагоналях необходимо хранить в памяти для каждой диагонали только номер строки, где находится активное состояние.

Отметим также, что благодаря эффекту пустых переходов никакое состояние (h, j), $j > k + h$, не может стать активным, если не будет активным состояние (k, m). Следовательно, пересчет состояний на диагоналях можно начинать с состояний $(0, j)$, $j = 1, 2, \ldots, m - k$. Такое упрощение позволяет определить все позиции i строки $\boldsymbol{x}$, где имеет место приближенное совпадение, однако не позволяет определить соответствующий уровень ошибки h, как это сделано в табл. 11.1 и 11.2.

Для работы с диагональными состояниями определим одномерный массив $D = D[0..m-k+1]$, в котором j-й элемент соответствует диагонали, начинающейся в состоянии $(0, j)$. Значением элемента $D[j]$ является целое число из интервала $0..k+1$. Если $D[j] = k+1$, то это означает, что на j-й диагонали нет активных состояний. Во всех других случаях значение $D[j]$ равно наименьшему номеру строки, где есть активное состояние на j-й диагонали. Обозначим через $D^{(0)}$ первоначальное состояние массива D: здесь $D^{(0)}[0] = D^{(0)}[m-k+1] = 0$, а для остальных элементов $j \in 1..m-k D^{(0)}[j] = k+1$. На основании формулы (11.7) получаем следующее рекуррентное соотношение для значений массива $D^{(i)}$, соответствующего позиции i строки $\boldsymbol{x}$ ($j \in 1..m-k$):

$$D^{(i)}[j] \leftarrow \min\{D^{(i-1)}[j+1]+1, D^{(i-1)}[j]+1, g(i-1, j-1)\}, \qquad (11.8)$$

где $g(i-1, j-1)$ — минимальный номер строки $h_{\min}$ на диагонали $j-1$, такой, что

- $D^{(i-1)}[j-1] \leqslant h_{\min} \leqslant k$ (другими словами, $A[h_{\min}, j-1]$ активно для $\boldsymbol{x}[i-1]$),
- $\boldsymbol{t}[\boldsymbol{x}[i], j-1+h_{\min}] = 0$ (это означает, что $\boldsymbol{x}[i] = \boldsymbol{p}[j-1+h_{\min}]$).

Если такого номера $h_{\min}$ не существует, тогда $g(i-1, j-1) = k+1$. В правой части формулы (11.8) первое выражение соответствует вертикальному переходу в BYNFA$(\boldsymbol{p}, k)$ (вставке буквы $\boldsymbol{x}[i]$), второе выражение — диагональному переходу (подстановка $\boldsymbol{x}[i]$ вместо $\boldsymbol{p}[j]$), третье — горизонтальному переходу (имеет место совпадение буквы $\boldsymbol{x}[i]$ и некоторой буквы паттерна, стоящей в позиции, соответствующей активному состоянию на диагонали $j-1$). Таким образом, формула (11.8) определяет все активные состояния каждой диагонали j ($j \in 1..m$) на основе входной буквы $\boldsymbol{x}[i]$ и на знании наименьших (по номеру строки) активных состояний на диагоналях $j+1$, j и $j-1$, соответствующих букве $\boldsymbol{x}[i-1]$.

Отметим, что h-приближенное совпадение паттерна $\boldsymbol{p}$ с некоторой подстрокой $\boldsymbol{x}[i'..i]$ строки $\boldsymbol{x}$ будет иметь место только тогда, когда $D^{(i)}[m-k] \leqslant k$ (здесь, конечно, $h \in 0..k$).

Алгоритм BYN для одного паттерна

Обычное компьютерное представление неотрицательных целых значений элементов $D[j]$, которые по величине не превышают $k+1$, требует $\lceil \log_2(k+2) \rceil$ бит, а всего для массива $D[1..m-k]$ — $\lceil \log_2(k+2) \rceil (m-k)$ бит. Но, к сожалению, такое представление элементов массива D не ведет к эффективному алгоритму вычислений. По этой причине в алгоритме BYN для элементов $D[j]$ используется представление вида $0^{k+1-D[j]}1^{D[j]}$. Тогда значение $D[j] = 0$ представимо как 0^{k+1}, $D[j] = 2$ — как $0^{k-1}1^2$, $D[j] = k+1$ — как 1^{k+1} и т.д. Если ввести дополнительно нулевой разделяющий бит слева от каждого значения $D[j]$, тогда весь

массив $D[1..m-k]$ можно представить в виде битовой строки

$$D = 0\ 0^{k+1-D[1]}1^{D[1]}0\ 0^{k+1-D[2]}1^{D[2]}\dots 0\ 0^{k+1-D[m-k]}1^{D[m-k]}$$

длиной $(k+2)(m-k)$ бит. Примерно такой же объем памяти необходим для хранения массива A.

Имея такое представление элементов $D[j]$, легко реализовать в виде бинарных операций вычисления, выполняемые над этими элементами в формуле (11.8), используя для этого такие замены.

О1. Вычисление $D[j]+1$ заменяется на *leftshift*$(D[j],1) \vee 0^k1$.

О2. Вычисление $\min\{D[j_1], D[j_2]\}$ заменяется на $D[j_1] \wedge D[j_2]$.

О3. Получение доступа к $D[j \pm 1]$ — *leftshift*$(D[j+1], k+2)$ или *rightshift*$(D[j - -1], k+2)$.

Для того чтобы реализовать функцию g, сначала надо записать в обратном порядке для каждой буквы $r \in 1..\alpha$ строку бинарного массива $\boldsymbol{t}$:

$$\boldsymbol{t}'[r] = \boldsymbol{t}[r,m]\boldsymbol{t}[r,m-1]\dots\boldsymbol{t}[r,1].$$

Затем опять для каждой буквы r зададим шаблон для массива $D[1..m-k]$ вида

$$T'[r] = 0\textit{rightshift}(\boldsymbol{t}'[r],0)0\textit{rightshift}(\boldsymbol{t}'[r],1)\dots 0\textit{rightshift}(\boldsymbol{t}'[r],m-k-1).$$

Делая обычное предположение о том, что после реализации любого сдвига, независимо от того, вправо был сделан этот сдвиг или влево, в позиции, "освобожденные" сдвигом заносятся нули, наконец можем представить все $m-k$ вычислений по формуле (11.8) (когда j меняется от 1 до $m-k$) в виде одной формулы бинарных операций, выполняемых над машинным словом, в котором записан массив D:

$$\begin{aligned} D = {} & (\textit{leftshift}(D, k+3) \vee (0^{k+1}1)^{m-k-1}0\ 1^{k+1}) \wedge \\ & \wedge (\textit{leftshift}(D,1) \vee (0^{k+1}1)^{m-k}) \wedge \\ & \wedge \textit{rightshift}((D' + (0^{k+1}1)^{m-k}) \wedge D', 1) \wedge \\ & \wedge D^{(0)}[1..m-k], \end{aligned} \tag{11.9}$$

где $D^{(0)}[1..m-k] = (0\ 1^{k+1})^{m-k}$ и $D' = \textit{rightshift}(D, k+2) \vee T[\boldsymbol{x}[i]]$.

Первые три строки в формуле (11.9) соответствуют трем выражениям в правой части формулы (11.8), однако здесь эти выражения записаны с помощью бинарных операций О1–О3. Массив $D^{(0)}$ в четвертой строке формулы введен для того, чтобы восстановить разделяющие нули между элементами массива D.

Текущий номер i буквы строки $\boldsymbol{x}$ будет выводиться как результат выполнения алгоритма только тогда, когда после вычисления по формуле (11.9) будет выполняться равенство

$$D \wedge 0^{(k+2)(m-k-1)}0\ 10^k = 0,$$

поскольку в этом случае $D^{(i)}[m-k] \leqslant k$. Затем, чтобы не было "ложных срабатываний" на следующих этапах алгоритма, значение $D[m-k]$ очищается с помощью операции

$$D \leftarrow D \vee 0^{(k+2)(m-k-1)}0\ 10^{k+1}.$$

Вычисления по формуле (11.9) слишком сложны для быстрого анализа, поэтому оставим для упражнения 11.2.10 получение точной оценки сложности алгоритма BYN. В работе [26] в качестве меры скорости выполнения алгоритма предложена форма цикла с перескоками (раздел 8.1): текстовая строка $\boldsymbol{x}$ сканируется для обнаружения совпадения с одной из первых $k+1$ букв паттерна и только после обнаружения такого совпадения начинает работу автомат BYNFA.

До сих пор мы описывали автомат BYNFA в предположении, что длина компьютерного слова достаточна для хранения всего массива D, т.е. предполагали, что $w \geqslant (k+2)(m-k)$. Если это предположение выполняется, тогда на каждом из n шагов алгоритма BYN требуется только константное время для выполнения бинарных операций над одним машинным словом и, следовательно, алгоритм завершится за время порядка $\Theta(n)$.

Декомпозиция BYNFA

В общем случае, когда $w < (k+2)(m-k)$, задача распределения по машинным словам всей информации, необходимой для вычисления в автомате BYNFA, вследствие двухмерной структуры этого автомата становится более сложной, чем аналогичная задача для других подобных алгоритмов, основанных на бинарных отображениях (раздел 10.4). Для автомата BYNFA различают три ситуации, в каждой из которых используется свой подход к декомпозиции автомата.

1. Ситуация, когда $w \geqslant k+2$.

В этой ситуации в одно машинное слово можно поместить значения элементов $\lceil w/(k+2) \rceil$ диагоналей из общего числа $m-k$ диагоналей, формируя таким образом "подавтоматы". Всего для автомата BYNFA потребуется $\lceil (m-k)/\lceil w/(k+2) \rceil \rceil$ машинных слов. Поскольку каждый элемент $D[j]$ зависит от элементов $D[j-1]$ и $D[j+1]$, лежащих на соседних диагоналях, в алгоритме необходимо предусмотреть взаимодействие каждого подавтомата со своими соседями. Однако и в этом случае, как описано в работе [26] и показано в упражнении 11.2.11, каждое машинное слово обрабатывается за константное время. Поэтому общее время выполнения алгоритма имеет порядок $O(k(m-k)n/w)$.

2. Ситуация, когда $w < k+2$, но $w \geqslant 2(m-k)$.

В этом случае для хранения значений элементов одной диагонали не хватает одного машинного слова. В такой ситуации можно попробовать сделать декомпозицию BYNFA на основе вертикальных компонентов, состоящих из $k+1$ элементов. Однако, поскольку эти элементы все равно будут содержать

части $m-k$ диагональных значений, необходимо предусмотреть $m-k$ разделяющих битов. Следовательно, для хранения самих значений в машинном слове останется только $w-(m-k)$ бит. (Именно поэтому мы поставили условие, что $w \geqslant 2(m-k)$.) Итак, при вертикальной декомпозиции автомата для хранения одного значения, соответствующего одной диагонали, в машинном слове остается

$$\lfloor (w-(m-k))/(m-k) \rfloor = \lfloor w/(m-k) \rfloor - 1$$

бит. Всего для полного представления автомата BYNFA необходимо $\lceil (k+$ $+1)/(\lfloor w/(m-k) \rfloor - 1) \rceil$ машинных слов. Здесь, как и в предыдущей ситуации, в алгоритме также необходимо предусмотреть взаимодействие между отдельными компонентами, однако и в этом случае каждое машинное слово можно обработать за константное время. Поэтому, как и ранее, общее время выполнения алгоритма имеет порядок $O(k(m-k)n/w)$.

3. Ситуация, когда $w < k+2$ и $w < 2(m-k)$.

Подробно данная ситуация рассмотрена в [26], где показано, что и в этом случае общее время выполнения алгоритма имеет порядок $O(k(m-k)n/w)$, а для хранения автомата BYNFA необходима память объемом $\Theta(k(m-$ $-k)n/w)$.

Напомним (см. обсуждение 1.3.1), что всегда можно положить $w \in \Omega(\log n)$. Поэтому справедлива следующая теорема.

Теорема 11.2.3. Для заданного одиночного паттерна $\boldsymbol{p} = \boldsymbol{p}[1..m]$ и неотрицательного целого числа $k < m$ алгоритм BYN на основе расстояния преобразования вычисляет все k-приближенные совпадения с текстовой строкой $\boldsymbol{x} = \boldsymbol{x}[1..n]$ за время порядка $O(k(m-k)n/\log n)$ с использованием памяти объемом $\Theta(k(m-$ $-k)/w)$, где w — длина машинного слова. ■

В работе [26] предложено несколько стратегий для уменьшения времени выполнения алгоритма BYN и проведен детальный анализ временны́х и пространственных требований алгоритма для того, чтобы показать завышенность оценок в самом худшем случае, данных в теореме 11.2.3. В частности, доказано, что модифицированный алгоритм BYN в среднем выполняется за время порядка $O(n)$, если величина k/n является медианным значением некоторого распределения. Таким образом, мы видим, что и в самом худшем случае и в среднем временные границы алгоритма BYN (в случае приближенного сравнения с одним паттерном) превышают аналогичные границы для модифицированного алгоритма Майерса (подраздел 10.3.2).

Алгоритм BYN для множественного паттерна

Основная идея алгоритма BYN для множественного паттерна заключается в том, что сначала с помощью алгоритма BYN для одного паттерна вычисляется

сверхмножество всех k-приближенных совпадений заданной строки $\boldsymbol{x}$ с набором паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$, затем это сверхмножество эффективно фильтруется для получения искомого множества решений. При описании алгоритма BYN для множественного паттерна сделаем упрощающее предположение, что все паттерны имеют одинаковую длину m/r, где m — общая длина всех паттернов. Это предположение не ограничивает общности, поскольку для того, чтобы длина всех паттернов была одинаковой, всегда в конец любого паттерна можно добавить символы, которые не будут совпадать с буквами алфавита.

Для вычисления сверхмножества эффективным оказался способ совмещения (наложения) паттернов, когда, например, на первой итерации автомат BYNFA(P, k) распознает совпадение не просто с первой буквой одного паттерна, а совпадение с любой из букв $\boldsymbol{p}_1[1]$, $\boldsymbol{p}_2[1]$, …, $\boldsymbol{p}_r[1]$; или на j-итерации автомат распознает совпадение с любой из букв $\boldsymbol{p}_1[j]$, $\boldsymbol{p}_2[j]$, …, $\boldsymbol{p}_r[j]$.

Для примера вместо автомата BYNFA$(this, 2)$, показанного на рис. 11.6, рассмотрим его расширение — автомат BYNFA$(\{this, worm\}, 2)$. Этот автомат показан на рис. 11.7. Если на вход данного автомата подается строка $\boldsymbol{x} = what$, то будут обнаружены 2-совпадения при $i = 2$, хотя $d_E(what, this) = d_E(what, worm) = 3$. С другой стороны, автомат корректно найдет совпадение при $i = 4$ в слове $\boldsymbol{x} = wham$ (расстояние преобразования между словом $wham$ и паттерном $worm$ равно 2), а также совпадения при $i = 3$ и $i = 4$ в слове $\boldsymbol{x} = whim$ (расстояние преобразования между этим словом и обоими паттернами $this$ и $worm$ равно 2).

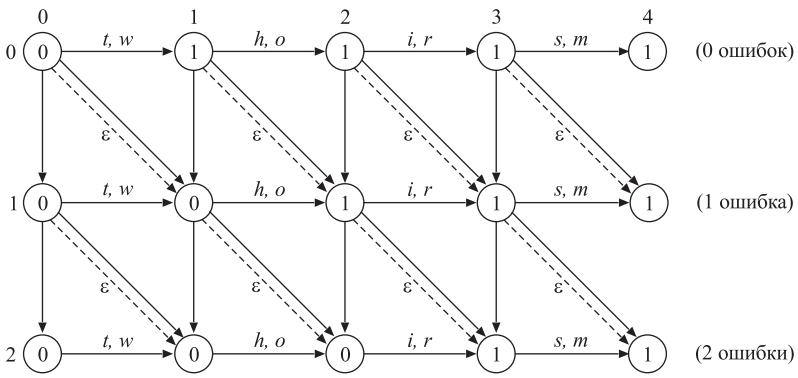


Рис. 11.7. Автомат BYNFA$(\{this, worm\}, 2)$

Такой подход для произвольной строки $\boldsymbol{x}$ позволяет определить в этой строке все подстроки, которые имеют k-совпадения хотя бы с одним частным паттерном из P. Кроме того, такой подход не сложно реализовать алгоритмически: для того чтобы на каждой итерации распознать, имеет ли место совпадение с несколькими

буквами, достаточно переопределить бинарный массив $\boldsymbol{t}[1..\alpha, 1..m]$ таким образом, чтобы равенство $\boldsymbol{t}[h, j] = 0$ для *любой* буквы h выполнялось только тогда, когда эта буква присутствует в позиции j хотя бы в одном частном паттерне. Эти вычисления можно выполнить на предварительном этапе алгоритма так, как показано в подразделе "Сравнение множеств букв" раздела 10.4. Каждый новый пересчет массива $\boldsymbol{t}$ выполняется за константное время. Поэтому, согласно теореме 7.4.4, массив $\boldsymbol{t}$ можно вычислить за время порядка $\Theta(\alpha\lceil m/w\rceil + m)$, где, как и ранее, m — сумма длин всех r паттернов из P. После предварительного вычисления массива $\boldsymbol{t}$ сверхмножество решений задачи приближенного сравнения с множественным паттерном, как следует из теоремы 11.2.3, можно вычислить за время порядка $O(k(m/r - k)n/\log n)$.

Пусть $S = S(P, k, \boldsymbol{x})$ обозначает вычисленное алгоритмом BYN супермножество для множественного паттерна P и заданной строки $\boldsymbol{x}$, k — аппроксимационная константа. Элементами множества S являются позиции в строке $\boldsymbol{x}$, для которых автоматом BYNFA(P, k) обнаружены совпадения.

Для того чтобы отфильтровать элементы множества S, не дающие k-приближенное совпадение хотя бы с одним частным паттерном из P, необходимо подстроки строки $\boldsymbol{x}$, определяемые каждым значением i_s ($s = 1, 2, \ldots, q$), сравнить с каждым паттерном $\boldsymbol{p}_t$ ($t = 1, 2, \ldots, r$). Если нет такого паттерна $\boldsymbol{p}_t$, что вычисленное расстояние преобразования между такой подстрокой и этим паттерном равно k, то элемент i_s удаляется из множества S. Каждое такое сравнение на основе расстояния преобразования можно выполнить с помощью алгоритма Укконена–Майерса (раздел 9.5) за время порядка $O(km/r)$. Тогда на обработку q элементов множества S потребуется время порядка $O(kmq)$, а поскольку $q \leqslant n$, то время, необходимое для фильтрации этого множества, имеет порядок $O(kmn)$.

Эффективность такого подхода сравнима с эффективностью модифицированного алгоритма Ву–Менбера (подраздел 11.2.3), поскольку зависит как от величины q, так и от числа q' элементов множества S, удаляемых в процессе фильтрации этого множества. Если значение q ограничено малым целым числом, тогда время, необходимое для выполнения алгоритма Укконена–Майерса, уменьшается до величины порядка $O(km)$. Если же значение q' мало, то основное время выполнения алгоритма Укконена–Майерса будет потрачено на проверку элементов множества S, а не на их отбраковку. В работе [27] предложено несколько стратегий, уменьшающих величину q'.

Упражнения 11.2

1. В алгоритме ACFA функция $match$ при некоторых значениях i может вызываться дважды. Перепишите этот алгоритм таким образом, чтобы исключить повторные вызовы функции $match$.

2. Как показано на рис. 11.4, в автомате ACFA(P), построенном для множественного паттерна, нумерация многих состояний может не совпадать с их естественной нумерацией, определяемой для одного конкретного паттерна. Объясните, как надо правильно назначать номера состояниям в процессе построения автомата ACFA(P).

3. Дано множество паттернов $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$. Разработайте эффективный алгоритм построения автомата ACFA(P) при выполнении *одного* из следующих условий:

- алфавит индексирован;
- алфавит упорядочен, но не индексирован.

В каждом случае оцените временну́ю сложность вашего алгоритма и исследуйте эффективность построенного автомата ACFA.

4. Для каждого случая, определенного в предыдущем упражнении, разработайте подходящую функцию $match$ и охарактеризуйте ее временны́е параметры.

5. Предположим, что в множестве $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_r\}$ паттерн $\boldsymbol{p}_1$ является гранью паттерна $\boldsymbol{p}_2$. Тогда, как видно на рис. 11.4, поглощающему состоянию, определяемому паттерном $\boldsymbol{p}_1$, не будет соответствовать конечная вершина автомата ACFA(P). В этом случае, если алгоритм ACFA фиксирует в позиции i строки $\boldsymbol{x}$ совпадение с паттерном $\boldsymbol{p}_2$, он не может зафиксировать в этой же позиции совпадение с паттерном $\boldsymbol{p}_1$. Обсудите возможность и необходимость изменить алгоритм ACFA, чтобы он в подобных ситуациях фиксировал совпадение и с паттерном $\boldsymbol{p}_1$.

6. Напомним, что в алгоритме КМП (раздел 7.1) на предварительном этапе вычисляется вспомогательный массив граней $\boldsymbol{\beta}'$. Однако более эффективен массив $\boldsymbol{\beta}''$, где j-й элемент равен наибольшей грани подстроки $\boldsymbol{p}[1..j-1]$, за которой *не следует* буква $\boldsymbol{p}[j]$.

7. Покажите, что с помощью автомата ACFA($\boldsymbol{p}$) можно вычислить массив граней $\boldsymbol{\beta}$, и запишите алгоритм (с линейным временем выполнения) для вычисления массива, подобного массиву $\boldsymbol{\beta}''$.

8. Напишите процедуру *hctam* для алгоритма CWFA.

9. (Исследовательский проект!) Для знакомого вам варианта алгоритма БМ (раздел 7.2) разработайте конечный автомат, подходящий для сравнения с множественным паттерном, и соответствующую процедуру сдвига. На этой основе создайте алгоритм сравнения с множественным паттерном. Оцените асимптотическую сложность этого алгоритма и сравните ее с аналогичными оценками алгоритмов ACFA и CWFA.

10. Покажите, что массив A, определенный в алгоритме BYNFA, содержит в точности $(k+1)(2m-k)/2$ элементов, соответствующих первоначально неактивным состояниям.

11. Предположим, что длина машинного слова достаточна для хранения всего массива $D = D[1..m-k]$. Для этого случая запишите реализацию алгоритма BYN, выполняющего сравнение с одним паттерном.

12. Предположим, что длина машинного слова w удовлетворяет неравенствам $w < (k+2)(m-k)$, $w \geqslant k+2$. Для этого случая измените алгоритм BYN таким образом, чтобы время его выполнения имело порядок $O(k(m-k)n)$.

13. Покажите, как надо изменить алгоритм BYN, чтобы он выводил не только позицию i строки $\boldsymbol{x}$, где имело место h-приближенное совпадение ($h \in 0..k$), но и само значение h. Возрастет ли сложность алгоритма после внесения в него таких изменений?

14. В тексте главы очень кратко говорилось о циклах с перескоками применительно к конечным автоматам, когда автомат начинает работу только с позиции i строки $\boldsymbol{x}$, такой, что буква $\boldsymbol{x}[i]$ совпадает хотя бы с одной буквой подстроки $\boldsymbol{p}[1..k+1]$. Используя идею циклов с перескоками, докажите, что k-приближенное совпадение строки $\boldsymbol{x}$ с паттерном $\boldsymbol{p}$ будет иметь место только тогда, когда по крайней мере одна из букв подстроки $\boldsymbol{p}[1..k+1]$ будет представлена в любой подстроке строки $\boldsymbol{x}$.

15. Запишите алгоритм для вычисления бинарного массива $B[1..\alpha]$, в котором r-й элемент показывал бы, входит ли буква r ($r \in 1..\alpha$) в подстроку $\boldsymbol{p}[1..k+1]$ паттерна $\boldsymbol{p}$. Какова сложность такого алгоритма?

16. Покажите, что при выполнении условия $m \leqslant 2(\sqrt{w}-1)$ автомат BYNFA($\boldsymbol{p}[1..m], k$) можно сохранить в одном машинном слове длиной w независимо от значения k. Из приведенного условия вычислите максимально возможное значение m для $w = 32$ и $w = 64$.

ЧАСТЬ IV

Вычисление характеристических паттернов

> Мой отец до сих пор каждый день читает словарь. Он говорит, что наша жизнь зависит от нашего умения пользоваться словами.
>
> — Артур Скарджилл (1938).
> The Sunday Times, 10 января 1982 г.

Наше путешествие в мир алгоритмов анализа строковых последовательностей завершается изучением методов вычисления характеристических паттернов — таких паттернов, которые описывают некоторые заданные свойства строк, а не указывают, какие символы в них содержатся. Например, мы распознаем строку aa и строку $bcbc$ как кратные строки, поскольку они обладают общим свойством; и нас не интересует, какие именно символы они содержат.

Характеристические паттерны, как правило, связаны со свойствами периодичности строковых последовательностей. Но, конечно, в качестве "характеристических" можно также рассматривать и другие свойства, не связанные с периодичность. В главе 12 сначала (в разделе 12.1) будут показаны "классические" алгоритмы вычисления кратных строк в произвольной строковой последовательности, а затем (в разделе 12.2) будут описаны недавние алгоритмы, которые предлагают более компактное представление кратных строк в виде серий (определение 2.3.3).

В главе 13 рассматриваются более слабые формы периодичности. Первой из них является "квазипериодичность": периодическая строка содержит последовательные вхождения образующей подстроки $\boldsymbol{u}$, но в квазипериодической строке подстроки $\boldsymbol{u}$ могут также накладываться друг на друга. В соответствии с определением 2.3.3, мы говорим, что квазипериодическая строка "покрывается" подстрокой $\boldsymbol{u}$. В разделе 13.1 описан алгоритм, который вычисляет все оболочки каждого префикса строки за линейное время. В разделах 13.2 и 13.3 обобщается идея кратных строк до понятия "раппорта", в котором не только не требуется последовательности вхождений повторяющейся подстроки, но и они могут быть отделены друг от друга. Мы рассмотрим алгоритмы построения раппортов, как точных, так и приближенных, и затем, наконец, используем обобщенное понятие "приближенной периодичности".

ГЛАВА 12

Периодичность

> Благодаря словам мы смогли подняться над животными; и благодаря же словам мы часто опускаемся до уровня демонов.
>
> — Олдос Хаксли (1894–1963).
> Адонис и Алфавит

В этой главе описаны алгоритмы для вычисления кратных строк в произвольной строковой последовательности $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Прежде чем приступить к этому материалу, читателю необходимо повторить начало раздела 2.3, где представлена основная терминология и определены кодировки кратных строк в виде троек чисел (i, p^*, r^*) и серий (i, p^*, r^*, t).

В конце 60-х годов было показано [157], что в любой строке $\boldsymbol{x}$ имеется порядка $O(n \log n)$ квадратов при условии, что образующие квадратов сами не являются кратными строками. В начале 80-х были разработаны три алгоритма [69, 18, 176], которые вычисляют все кратные строки в строковой последовательности $\boldsymbol{x}$ за время $O(n \log n)$. Два из них представлены в двух первых разделах этой главы. Но сначала покажем, что верхняя граница времени выполнения таких алгоритмов наиболее вероятна для строк, определенных на неупорядоченном алфавите, — результат, установленный в [176].

Теорема 12.0.4. Пусть $\boldsymbol{x} = \boldsymbol{x}[1..n]$ — строковая последовательность, определенная на неупорядоченном алфавите. Любой алгоритм, предназначенный для нахождения кратных строк в строке $\boldsymbol{x}$ и основанный на посимвольном сравнении, в самом худшем случае требует времени выполнения порядка $\Omega(n \log n)$.

Доказательство. Напомним (см. раздел 4.1), что символы из обычного алфавита могут сравниваться только для определения их равенства или неравенства. Предположим, что $\boldsymbol{x}$ — это строка, в которой каждый из n символов отличается от другого. Приняв для простоты, что n является четным числом, покажем, что для установления факта, что $\boldsymbol{x}$ не содержит квадратов и, следовательно, ни одной кратной строки, необходимо $\Omega(n \log n)$ посимвольных сравнений.

Пусть для некоторых целых i и j, таких, что $1 \leqslant i < j \leqslant n$, при сравнении букв $\boldsymbol{x}[i]$ и $\boldsymbol{x}[j]$ обнаружено, что $\boldsymbol{x}[i] \neq \boldsymbol{x}[j]$. В строке $\boldsymbol{x}$ единственными квадратами, которые могут исключаться по этому сравнению, будут те, для которых необходимо выполнение равенства $\boldsymbol{x}[i] = \boldsymbol{x}[j]$. Если обозначить $k = j - i$, то не более k таких квадратов, каждый длиной $2k$, исключаются:

$$\boldsymbol{x}[i-k+1..j], \boldsymbol{x}[i-k+2..j+1], \ldots, \boldsymbol{x}[i..j+k-1]. \tag{12.1}$$

Отметим, что в действительности может быть исключено менее k квадратов, поскольку, например, список исключенных квадратов, получаемый на основе неравенства $\boldsymbol{x}[i+1] \neq \boldsymbol{x}[j+1]$,

$$\boldsymbol{x}[i-k+2..j+1], \boldsymbol{x}[i-k+3..j+2], \ldots, \boldsymbol{x}[i+1..j+k]$$

содержит только один квадрат, не включенный ранее в список (12.1).

Для любого целого числа $k \in 1..n/2$ существует (как отмечалось в разделе 1.2) всего $n - 2k + 1$ различных возможных подстрок длиной $2k$, следовательно, возможно $n-2k+1$ различных квадратов длиной $2k$. Таким образом, для исключения всех возможных квадратов длиной $2k$ необходимо сделать не менее $(n-2k+1)/k$ символьных сравнений.

Заметим также, что после исключения всех возможных квадратов длиной $2k$ мы не получим информации о наличии или отсутствии каких-либо квадратов длиной $2k'$, если $k' \neq k$, поскольку для определения того, является ли какая-либо подстрока длиной $2k'$ квадратом, необходимо сравнить пары символов, отделенные друг от друга на k' позиций. Таким образом, исключение квадратов различной длины $2k$, $k = 1, 2, \ldots, n/2$, должно рассматриваться как независимые действия. Следовательно, чтобы быть уверенным в том, что в строке $\boldsymbol{x}$ нет квадратов, необходимо сделать не менее

$$\sum_{k=1}^{n/2} (n - 2k + 1)/k \geqslant (n+1)H_{n/2} - n$$

символьных сравнений, где H_i — i-е гармоническое число[1]. Поскольку $H_i > \ln i + \gamma$ [135], где $\gamma \approx 0{,}577$ — константа Эйлера, то отсюда находим, что необходимое

[1] Гармоническое число H_i — это просто обозначение суммы $1 + 1/2 + \cdots + 1/i$. — *Примеч. ред.*

количество символьных сравнений превышает величину

$$(n+1)(\ln n - \ln 2 + \gamma)) - n > (n+1)(\ln n - 1{,}13) \in \Omega(n \log n). \quad \blacksquare$$

Этот результат является одним из основных при вычислении кратных строк, однако он порождает новые принципиальные вопросы. Например, такие.

- Справедлива ли нижняя граница $n \log n$ в тех случаях, когда алфавит не общего вида, а, например, упорядочен? В работе [70] Крочемор (Crochemore) описывает алгоритм, который на основе деревьев суффиксов (раздел 5.2) за время порядка $O(n \log \alpha)$ определяет, действительно ли данная строка $\boldsymbol{x}[1..n]$, определенная на упорядоченном алфавите размером α, не содержит квадратов. В статье [170] Мейн и Лоренц (Main and Lorentz) предлагают алгоритм с подобным временем выполнения, но в котором не используются деревья суффиксов. Если $\alpha \in \Theta(n)$, тогда оценка временнóй сложности этих алгоритмов не является улучшением результата теоремы 12.0.1. С другой стороны, если α — это известная константа, то и $\log \alpha$ является константой, и, таким образом, эти алгоритмы можно рассматривать как алгоритмы, которые выполняются за время порядка $O(n)$. См. также подраздел 5.2.5.
- Можно ли для строковой последовательности $\boldsymbol{x}[1..n]$, определенной на упорядоченном алфавите, вычислить ее кратные строки за время $O(n)$ или, по крайней мере, за время, меньшее чем $\Theta(n \log n)$? До некоторого времени казалось, что вряд ли можно на этот вопрос ответить утвердительно, поскольку, принимая во внимание результат Крочемора (см. [79] и раздел 3.4), строки Фибоначчи длиной f_n содержат порядка $\Theta(f_n \log f_n)$ кратных строк, закодированных в тройки (i, p^*, r^*), которые по существу являются нормальными формами (раздел 1.2) любых максимальных повторяемых подстрок. Поэтому казалось, что для вычисления всех кратных подстрок строки длиной n необходимо время порядка $\Theta(n \log n)$. Но относительно недавно было введено понятие серии (раздел 2.3), кодирующей строки в виде кортежа из 4 элементов (i, p^*, r^*, t) [168], которое было использовано различными авторами [108, 123, 138]. Затем в [138, 139] было показано, что количество серий в любой строке линейно по отношению к длине строки. Таким образом, возникает возможность, по крайней мере для строк, определенных на известном упорядоченном алфавите, за линейное время вычислить кратные строки, кодированные в виде серий. В [139] для вычисления серий в строке $\boldsymbol{x}$ представлен алгоритм, в котором используются деревья суффиксов (следовательно, предполагается упорядоченность алфавита). Если дополнительно предположить, что алфавит не очень велик либо что он фиксирован или индексирован (раздел 4.1), то потенциальным фактором $\log n$, скрытым в конструкции дерева суффиксов, можно пренебречь. Тогда получим алгоритм со временем выполнения порядка $\Theta(n)$. Этот алгоритм мы рассмотрим

в разделе 12.2. Вопрос о существовании алгоритма вычисления серий для упорядоченных алфавитов со временем выполнения порядка $\Theta(n)$, который не использовал бы деревьев суффиксов, на сегодняшний день остается открытым.

12.1 Все кратные строки

В этом разделе представлены два алгоритма, значительно отличающиеся друг от друга, но которые с сопоставимой сложностью достигают одинаковой цели. Это алгоритм Крочемора [79] со временем выполнения $O(n \log n)$ и алгоритм Мейна и Лоренца [176] со временем выполнения $\Theta(n \log n)$, которые вычисляют все кратные строки в произвольной строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$.

12.1.1 Алгоритм Крочемора

Прежде чем перейти к объяснению алгоритма Крочемора, вначале опишем простой алгоритм, который на каждом этапе достигает такого же результата, что и алгоритм Крочемора, а именно: в строке $\boldsymbol{x}$ он вычисляет все повторяющиеся подстроки длиной ℓ, где $\ell = 1, 2, \ldots, n-1$.

Рассмотрим строку Фибоначчи

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{f}_6 =$	a	b	a	a	b	a	b	a	a	b	a	a	b

Предположим, что на этапе (уровне) $\ell = 1$ в строке $\boldsymbol{f}_6$ вычислены последовательности позиций, в которых встречаются одинаковые символы:

$\ell = 1$	$<1, 3, 4, 6, 8, 9, 11, 12>$	$<2, 5, 7, 10, 13>$
	a	b

Напомним (раздел 4.1), что для строк, определенных на неупорядоченном алфавите, эти вычисления можно выполнить за время порядка $\Theta(n^2)$, а для определенных на упорядоченных и индексированных алфавитах — за время, соответственно, $O(n \log n)$ и $\Theta(n)$.

Далее для $\ell = 2$ мы хотим найти все повторяющиеся подстроки длиной 2. Поскольку повторяющиеся подстроки длиной $\ell \geqslant 2$ будут иметь общий префикс длиной $\ell - 1$, то, очевидно, вычисления уровня ℓ должны привести к последовательностям, которые будут *подпоследовательностями* последовательностей, вычисленных на уровне $\ell - 1$. Другими словами, на уровне $\ell \geqslant 2$ разбиение множества позиций строки на отдельные непересекающиеся множества (последовательности) — это ***декомпозиция*** разбиения на уровне $\ell - 1$. Пример такой декомпозиции показан в табл. 12.1.

Таблица 12.1. Последовательная декомпозиция строки $\boldsymbol{f}_6 = abaababaabaab$

$\ell = 2$	$<1,4,6,9,12>$	$<3,8,11>$	$<2,5,7,10>$	$<13>$	
	ab	aa	ba	$b\$$	
$\ell = 3$	$<1,4,6,9>$	$<12>$	$<3,8,11>$	$<2,7,10>$	$<5>$
	aba	$aa\$$		baa	bab
$\ell = 4$	$<1,6,9>$	$<4>$	$<3,8>$	$<11>$	$<2,7,10>$
	$abaa$	$abab$	$aaba$	$aab\$$	$baab$
$\ell = 5$	$<1,6,9>$	$<3>$	$<8>$	$<2,7>$	$<10>$
	$abaab$	$aabab$	$aabaa$	$baaba$	$baab\$$
$\ell = 6$	$<1,6>$	$<9>$	$<2>$	$<7>$	
	$abaaba$	$abaab\$$	$baabab$	$baabaa$	
$\ell = 7$	$<1>$	$<6>$			
	$abaabab$	$abaabaa$			

Примечание. В этой таблице одноэлементные множества после их первого появления были опущены, поскольку их дальнейшая декомпозиция невозможна.

Нетрудно реализовать процесс декомпозиции так, чтобы в самом худшем случае для его реализации потребовалось время порядка $\Theta(n^2)$. Это процесс *непосредственной* декомпозиции каждой последовательности, полученной на уровне ℓ, на подпоследовательности для уровня $\ell + 1$. Например, строка $\boldsymbol{x} = (ab)^{n/2}$ для каждого уровня ℓ потребует $n - \ell + 1$ сравнений, следовательно, всего необходимо $\Theta(n^2)$ сравнений. Однако в [117] описаны две идеи, которые уменьшают временну́ю сложность этого процесса.

1. Декомпозицию каждой последовательности можно получить косвенным путем, а не путем прямых вычислений. Идея такого подхода состоит в следующем.

На каждом уровне ℓ выполняется непосредственная декомпозиция любой последовательности $c_j^{(\ell)}$, основанной на конкретных символах, находящихся в первой позиции справа от подстроки, соответствующей $c_j^{(\ell)}$. Более точно, если $c_j^{(\ell)} = <p_1, p_2, \ldots, p_r>$, то необходимо проверить совпадение букв $\boldsymbol{x}[p_1 + \ell], \boldsymbol{x}[p_2 + \ell], \ldots, \boldsymbol{x}[p_r + \ell]$, и если какие-либо пары букв равны, например $\boldsymbol{x}[p_{h_1} + \ell] = \boldsymbol{x}[p_{h_2} + \ell]$, тогда p_{h_1} и p_{h_2} помещаются в одну и ту же последовательность на уровне $\ell + 1$.

Таким образом, например, в приведенном выше примере строки $\boldsymbol{f}_6$ последовательность $<1,4,6,9>$ на уровне $\ell = 3$ разбивается на уровне $\ell = 4$ на последовательности $<1,6,9>$ и $<4>$, поскольку

$$\boldsymbol{f}_6[1+3] = \boldsymbol{f}_6[6+3] = \boldsymbol{f}_6[9+3] \neq \boldsymbol{f}_6[4+3].$$

Но декомпозицию можно также выполнить, основываясь не на разбиваемой последовательности, а на последовательностях, *относительно которых* будут разбиваться другие последовательности. Снова рассмотрим уровень $\ell = 3$ и последовательность

$$c_1^{(3)} = <p_1, p_2, p_3, p_4> = <1, 4, 6, 9>,$$

относящуюся к подстроке aba. Для каждой позиции $p_h > 1$ известно, что подстрока $\boldsymbol{f}_6[p_h - 1..p_h + \ell - 1]$ (длиной $\ell + 1$) принадлежит к некоторой последовательности $c_{j'}^{(\ell+1)}$ на уровне $\ell + 1$. Поскольку последовательность $c_j^{(\ell)}$ соответствует уникальной подстроке строки $\boldsymbol{f}_6$, каждая такая последовательность $c_{j'}^{(\ell+1)}$ должна формироваться из тех же позиций $p_{h_1}, p_{h_2}, \ldots, p_{h_k}$ последовательности $c_j^{(\ell)}$, которые определяют класс эквивалентности

$$\boldsymbol{f}_6[p_{h_1} - 1] = \boldsymbol{f}_6[p_{h_2} - 1] = \cdots = \boldsymbol{f}_6[p_{h_k} - 1].$$

В нашем примере $c_j^{(\ell)} = c_1^{(3)} = <1, 4, 6, 9>$, и, поскольку $\boldsymbol{f}_6[4 - 1] = \boldsymbol{f}_6[9 - 1] = a$, последовательность $<3, 8, 11>$ уровня 3 на уровне 4 разбивается на подпоследовательности $<3, 8>$ и $<11>$. Далее, так как $\boldsymbol{f}_6[5 - 1] = b$, последовательность $<5>$ уровня 3 на уровне 4 переходит просто в саму себя. Таким образом, декомпозиция последовательностей $<3, 8, 11>$ и $<5>$ выполняется *по отношению к* последовательности $<1, 4, 6, 9>$.

Подобным образом декомпозиция относительно последовательности $<2, 7, 10>$ на уровне 3 приводит к равенствам

$$\boldsymbol{f}_6[2 - 1] = \boldsymbol{f}_6[7 - 1] = \boldsymbol{f}_6[10 - 1] = a,$$

и, следовательно, на уровне 4 формируется последовательность $<1, 6, 9>$, которая является декомпозицией последовательности $<1, 4, 6, 9>$ уровня 3.

Таким способом декомпозицию на уровне $\ell + 1$ можно выполнить косвенным путем, рассматривая каждую последовательность уровня ℓ с позиции, находящейся на одну позицию левее от начальной позиции этой последовательности.

2. Косвенная декомпозиция не обязательно должна выполняться относительно *каждой* последовательности уровня ℓ: в каждом наборе последовательностей, порожденных одной последовательностью уровня $\ell - 1$, всегда можно исключить использование одной из них для декомпозиции последовательностей на уровне ℓ.

Снова рассмотрим декомпозицию последовательности $<1, 4, 6, 9>$ уровня 3 на подпоследовательности $<1, 6, 9>$ и $<4>$ уровня 4. Последовательность

$<1,4,6,9>$ соответствует строке с собственным суффиксом ba, представленной на уровне 2 последовательностью $<2,5,7,10>$: таким образом, любая декомпозиция последовательности $<1,4,6,9>$ на уровне 4 получается из соответствующей декомпозиции последовательности $<2,5,7,10>$ уровня 3. Фактически последовательность $<1,6,9>$ получена из подпоследовательности $<2,7,10>$ (суффикса baa), в то время как $<4>$ получена из подпоследовательности $<5>$ (суффикс bab). Этот пример показывает, что для декомпозиции последовательности $<1,4,6,9>$ на уровне 4 достаточно использовать *все, за исключением одной*, декомпозиции последовательности $<2,5,7,10>$ уровня 3 — конечным разбиением будет то, что останется от последовательности $<1,4,6,9>$ после того, как будут вычислены другие разбиения. В этом примере есть два выбора: можно получить или $<1,6,9>$ из $<2,7,10>$, оставляя $<4>$ как остаток, или $<4>$ из $<5>$, оставляя $<1,6,9>$ в виде остатка. Конечно, второй подход предпочтительнее, поскольку требует только одного вычисления.

Эти примеры подводят к следующему определению.

Определение 12.1.1. В декомпозиции последовательности $c_j^{(\ell)}$ на подпоследовательности $(c_1^{(\ell+1)}, c_2^{(\ell+1)}, \ldots, c_q^{(\ell+1)})$ $(q \geqslant 1)$ назовем одну подпоследовательность с самым большим количеством элементов **большой**, а остальные $q-1$ подпоследовательностей — **малыми**. Для уровня $\ell = 1$ каждая последовательность **малая**. ■

Отметим, что в случае $q = 1$ не будет ни одной малой подпоследовательности, в частности, всегда, когда $c_j^{(\ell)}$ состоит из одного элемента.

Теперь основную идею алгоритма Крочемора кратко можно изложить в следующем виде: выполнять декомпозицию последовательностей на каждом уровне ℓ *только* относительно последовательностей, которые на этом уровне являются малыми. Как показывает следующий результат, эта стратегия также определяет эффективность процесса декомпозиции.

Лемма 12.1.2. Предположим, что декомпозиция последовательностей, соответствующих произвольной строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$, выполняется для уровней $\ell = 1, 2, \ldots, \ell^*$, где ℓ^* ($\ell^* \in 1..n-1$) — наименьший уровень, на котором каждая последовательность содержит единственную позицию. Тогда каждая позиция i строки $\boldsymbol{x}$ входит в малые последовательности $O(\log n)$ раз.

Доказательство. Заметим, что если последовательность $c_j^{(\ell)}$ разбивается на подпоследовательности $(c_1^{(\ell+1)}, c_2^{(\ell+1)}, \ldots, c_q^{(\ell+1)})$, то каждая малая последовательность $c_{j'}^{(\ell+1)}$ должна удовлетворять условию $\left|c_{j'}^{(\ell+1)}\right| \leqslant \left|c_j^{(\ell)}\right|/2$. Другими словами,

каждая малая подпоследовательность при $\ell \geqslant 2$ не превышает половины размера своей исходной последовательности. Поскольку для $\ell - 1$ начальная малая последовательность может содержать не более n позиций, то из этого следует, что ни одна из позиций не может входить в больше чем $\lceil \log_2(n+1) \rceil$ малых последовательностей. ■

Поскольку строка $\boldsymbol{x}$ содержит n позиций, то из леммы 12.1.2 следует, что всего в малых последовательностях на всех уровнях содержится $O(n \log n)$ позиций. Таким образом, если время обработки последовательностей на каждом уровне ℓ пропорционально количеству элементов в малых последовательностях этого уровня, то полный процесс декомпозиции будет выполнен за время порядка $O(n \log n)$. Представленный ниже алгоритм 12.1.1 показывает схему этого процесса, который выглядит в достаточной степени простым. Это и стало причиной известности алгоритма Крочемора, однако основная сложность этого алгоритма (и других, ему подобных) заключается в структурах данных, необходимых для получения времени выполнения алгоритма порядка $O(n \log n)$.

Алгоритм 12.1.1 (Алгоритм Крочемора)

▷ *Вычисление всех кратных строк в строке* $\boldsymbol{x} = \boldsymbol{x}[1..n]$
$\ell \leftarrow 1$
вычисление последовательностей на уровне 1,
все они помечаются как *малые*
while на уровне ℓ есть малая последовательность **do**
    output кратные строки с периодом ℓ (если они есть)
    декомпозиция последовательности уровня ℓ на подпоследовательности
      уровня $\ell + 1$, используя позиции только
      в малых последовательностях
    $\ell \leftarrow \ell + 1$
    вычисление малых последовательностей на уровне ℓ

Прояснив основную идею алгоритма Крочемора, классифицируем используемые им структуры данных в соответствии с их основными функциями.

1. Запись текущей последовательности для каждой позиции в строке $\boldsymbol{x}$.

Массив SEQ$[\ell..n]$:	SEQ$[i]$ содержит индекс текущей последовательности, которой принадлежит i-я позиция в строке $\boldsymbol{x}$.
Массив SEQLIST$[1..2n]$:	SEQLIST$[j]$ — указатели на двусвязный список позиций, принадлежащих последовательности с индексом j и расположенных в порядке их возрастания.

Массив SEQSIZE$[1..2n]$: SEQSIZE$[j]$ равно количеству позиций в последовательности с индексом j, т.е. количеству позиций в списке, на который указывает элемент SEQLIST$[j]$.

Стек INDEXSTACK: стек неиспользованных индексов последовательностей, инициализированный для размещения индексов $1, 2, \ldots, 2n$.

Всякий раз, когда необходимо сформировать новую последовательность или на уровне 1, или в качестве части декомпозиции существующей последовательности, из стека INDEXSTACK извлекается индекс последовательности. Если последовательность становится пустой в результате ее декомпозиции на подпоследовательности, индекс последовательности возвращается в стек INDEXSTACK. Всякий раз, когда к последовательности с индексом j добавляется позиция i, выполняются операторы присваивания

$$\text{SEQ}[i] \leftarrow j; \text{SEQSIZE}[j] \leftarrow \text{SEQSIZE}[j] + 1,$$

а номер i добавляется в конец списка SEQLIST$[j]$. Если i-я позиция удаляется из последовательности с индексом j в результате его вхождения в подпоследовательность с индексом j, выполняется присваивание SEQSIZE$[i] \leftarrow$ $\leftarrow$ SEQSIZE$[j] - 1$, а i-я позиция удаляется из списка SEQLIST$[i]$.

Как показано в упражнении 12.1.4, в любой момент используется не более $2n$ индексов последовательности. Следовательно, массива размером $[1..2n]$ будет достаточно для выполнения всех операций. Заметим также, что поскольку i-е позиции всегда сохраняются в списке SEQLIST в порядке их возрастания и поэтому всегда удаляются из этого списка в таком же порядке, то операции и сохранения и удаления можно выполнить за константное время. Отсюда следует, что любую операцию, выполняемую в процессе декомпозиции для записи изменений принадлежности позиций к разным последовательностям, можно реализовать за константное время.

2. Управление малыми последовательностями.

Очередь SMALL: очередь индексов j малых последовательностей, организованная таким образом, что (для $\ell > 1$) все имеющиеся в ней малые последовательности являются подпоследовательностями одной и той же последовательности предыдущего уровня.

Очередь QUEUE: очередь пар (i, j), где i — позиция в малой последовательности с индексом j, которая должна использоваться для декомпозиции текущего уровня; для каждого j позиции i располагаются в возрастающем порядке.

Массив LASTSMALL$[1..2n]$: LASTSMALL$[j'] = j > 0$ тогда и только тогда, когда j — это индекс малой последовательности, использованной последней по времени для декомпозиции последовательности с индексом j'.

Для $\ell = 1$ очередь SMALL содержит все индексы, которые первоначально вычисляются в соответствии с каждым символом строки $\boldsymbol{x}$; для $\ell > 1$ очередь SMALL создается повторно путем просмотра подпоследовательностей тех последовательностей, для которых была выполнена декомпозиция на уровне ℓ, и добавлением к очереди SMALL всех подпоследовательностей, кроме единственной "большой" подпоследовательности (см. описание массивов SPLIT и SUBSEQ в следующем пункте). Для каждого элемента j в очереди SPLIT эти действия можно выполнить за один просмотр списка, на который указывает элемент SUBSEQ$[j]$. Таким образом, очередь SMALL модифицируется за время, пропорциональное количеству имеющихся в ней элементов.

Очередь QUEUE создается в начале обработки для каждого уровня $\ell > 1$ путем удаления элементов из очереди SMALL. Для каждой последовательности j, обнаруженной в очереди SMALL, извлекаются найденные в списке SEQLIST$[j]$ позиции $i > 1$, а элементы (i, j) добавляются к очереди QUEUE.

Основная обработка каждого из уровней $\ell > 1$ выполняется после того, как из очереди QUEUE один за другим удаляются элементы (i, j). Для каждой удаленной позиции i последовательность $j' = \text{SEQ}[i-1]$ является той последовательностью, которая будет подвергнута декомпозиции. Следовательно, если LASTSMALL$[j'] \neq j$, то последовательность j' была последней по времени подвергнута декомпозиции относительно некоторого другого класса, поэтому необходим новый индекс последовательности. Соответственно с этим устанавливаем LASTSMALL$[j'] \leftarrow j$, извлекаем из стека INDEXSTACK следующий номер последовательности и добавляем его к списку, на который указывает SUBSEQ$[j']$ (см. следующий пункт).

Таким образом, для выполнения операций, связанных с каждым элементом очереди QUEUE, необходимо одинаковое константное время.

3. Организация подпоследовательностей.

Массив SPLITFLAG$[1..2n]$: SPLITFLAG$[j] = 1$ тогда и только тогда, когда последовательность с индексом j будет подвергнута декомпозиции на текущем уровне (с использованием последовательностей, которые хранятся в очереди SMALL).

Список SPLIT: список последовательностей j, которые будут подвергнуты декомпозиции на текущем уровне (т.е. для которых SPLITFLAG$[j] = 1$).

Массив SUBSEQ$[1..2n]$: при декомпозиции последовательности j на текущем уровне SUBSEQ$[j]$ указывает на список индексов подпоследовательностей последовательности j, первым элементом в этом списке будет сам индекс j.

Все эти структуры данных инициализируются для каждого уровня $\ell > 1$ одновременно с формированием из очереди SMALL очереди QUEUE. В частности, когда (i, j) добавляется к очереди QUEUE, известно, что i будет использоваться для декомпозиции последовательности j', в которой имеется позиция $i - 1$ ($j' = \text{SEQ}[i - 1]$). Поэтому, если SPLITFLAG$[j'] = 0$, то j' в это время хранится в SPLIT — списке последовательностей, которые подлежат декомпозиции на уровне ℓ, в то время как выполняются следующие присваивания:

$$\text{SPLITFLAG}[j'] \leftarrow 1; \text{SUBSEQ}[j'] \leftarrow <j'>; \text{LASTSMALL}[j'] \leftarrow 0.$$

Из этого следует, что для помещения в очередь QUEUE любого элемента требуется одинаковое константное время, и, как было показано во втором пункте, на удаление любого элемента из очереди QUEUE также необходимо одинаковое константное время. Поскольку элементы в очереди QUEUE являются позициями в малых последовательностях, то, очевидно, подпоследовательности и малые последовательности можно организовать заранее пропорционально количеству малых последовательностей, на что, в соответствии с леммой 12.1.2, необходимо время порядка $O(n \log n)$.

По завершении обработки каждого уровня ℓ ($\ell > 1$) последовательности j удаляются один за другим из списка SPLIT: если SEQSIZE$[j] = 0$, то эта последовательность уже пуста и поэтому ее индекс больше не требуется. Поэтому j помещается в стек INDEXSTACK и удаляется из списка, на который указывает SUBSEQ$[j]$.

4. Вычисление кратных строк.

Массив GAP$[\ell..n]$: элемент GAP$[i]$ равен положительной разности между i-й позицией в строке $\boldsymbol{x}$ и следующей большей позицией в той же самой последовательности на текущем уровне; если нет большей позиции, то GAP$[i] = \infty$.

Массив GAPLIST$[\ell..n]$: GAPLIST$[g]$ указывает на двусвязный список i-х позиций, для которых GAP$[i] = g$.

Для $\ell = 1$ массивы GAP и GAPLIST инициализируются непосредственно путем обработки каждого списка, на который указывает SEQLIST$[j]$, где $j = 1, 2, \ldots, \alpha'$, вычисляя GAP$[i]$ для каждой i-й позиции в списке SEQLIST$[j]$, затем добавляя i-ю позицию к списку, на который указывает GAPLIST[GAP$[i]$].

При $\ell > 1$ массивы GAP и GAPLIST обновляются при удалении элемента (i, j) из очереди QUEUE, что приводит к декомпозиции последовательности $j' =$ SEQ$[i-1]$, при которой позиция $i-1$ в SEQLIST$[j']$ будет перемещена в конец SEQLIST$[j'']$. Поскольку списки, на которые указывают и SEQLIST и GAPLIST, двусвязные, повторное вычисление элементов массива GAP, необходимое из-за удаления позиции $i-1$ из одного списка и ее вставки в другой список, можно выполнить непосредственно.

Мы видим, что, после создания последовательностей для уровня ℓ, каждый элемент в GAPLIST$[\ell]$ описывает две идентичные подстроки строки $\boldsymbol{x}$ длиной ℓ, первые буквы которых находятся друг от друга на расстоянии ℓ, другими словами, имеем квадрат этой подстроки. В этом случае массивы GAP и GAPLIST можно использовать так, как показано в следующем фрагменте листинга, для вывода нормальной формы максимальных кратных строк для каждого уровня ℓ за время, пропорциональное количеству квадратов.

```
while GAPLIST[ℓ] not пусто do
    i_0 ← следующая позиция в GAPLIST[ℓ]
    i ← i_0; r ← 1
    repeat
        удаление i из GAPLIST[ℓ]
        i ← i + ℓ; r ← r + 1
    until GAP[i] ≠ ℓ
    output (i_0, ℓ, r)
```

Предположим, что в строке $\boldsymbol{x}$ встречается подстрока $\boldsymbol{u}^2$, где $|\boldsymbol{u}| = \ell$, а $\boldsymbol{u}$ сама является кратной строкой $\boldsymbol{v}^k$ для некоторого целого числа $k \geqslant 2$. Тогда $\boldsymbol{u}^2 = \boldsymbol{v}^{2k}$ содержит $k+1$ вхождений $\boldsymbol{u}$ с пропуском $\ell/k < \ell$, поэтому подстрока $\boldsymbol{u}$ никогда не будет выбрана в качестве образующей кратных строк. Поскольку, как было отмечено ранее, в строке $\boldsymbol{x}$ имеется не более $O(n \log n)$ квадратов, образующие которых сами не являются кратными строками [157], тогда все кратные строки можно найти за время $O(n \log n)$.

Этим завершается наше описание алгоритма Крочемора. Из-за необходимости реализации описанных пп. 1–4, этот алгоритм очень объемен и технически сложен. Мы не представляем здесь его во всех деталях; заинтересованного читателя отправляем к работе [69]. Тем не менее мы попытались показать читателю, каким образом сложные структуры данных могут использоваться для декомпозиции каждого уровня и обеспечивать время вычисления кратных строк порядка $O(n \log n)$. Вспомнив, что декомпозицию уровня 1 на упорядоченном алфавите можно вычислить за время $O(n \log n)$, получаем следующий важный результат.

Теорема 12.1.3. Алгоритм 12.1.3 вычисляет все кратные строки в произвольной строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$, определенной на упорядоченном алфавите, за время порядка $O(n \log n)$ с использованием памяти объемом $\Theta(n)$.

Доказательство. Истинность приведенной оценки верхней границы времени выполнения алгоритма следует из истинности трех отдельных утверждений:

- алфавит является упорядоченным;
- лемма 12.1.2;
- количество обработанных квадратов в любой строке длиной n имеет порядок $O(n \log n)$ [157]. ■

Важно отметить, что для вычисления кратных строк с помощью алгоритма Крочемора необходимо просмотреть не более $\lfloor n/2 \rfloor$ уровней, поскольку в любой строке $\boldsymbol{x}$ максимальный период ℓ ее кратных строк удовлетворяет условию $\ell \leqslant \lfloor n/2 \rfloor$. Но оценка верхней временно́й границы $O(n \log n)$ будет справедлива даже тогда, когда декомпозиция уровней продолжается до тех пор, пока в SEQL1ST не останутся только одноэлементные списки. В этом случае придется рассмотреть $\ell > \lfloor n/2 \rfloor$ уровней, например, как для строки $\boldsymbol{x} = abaababaababa$, содержащей длинные перекрывающиеся подстроки. Из этого видно, что алгоритм Крочемора можно легко расширить для вычисления всех раппортов строки $\boldsymbol{x}$ за время порядка $O(n \log n + Q)$, где Q — размер выводимых данных. Мы возвратимся к этой задаче в разделе 13.2, где покажем, как все повторяемые подстроки в строке $\boldsymbol{x}$ можно представить в виде данных объемом $Q \in O(n)$.

Нетрудно заметить, что алгоритм Крочемора в расширенном виде по существу является алгоритмом для построения дерева суффиксов. Это легко увидеть, если

представить декомпозицию строки $\boldsymbol{f}_6$ из предыдущего примера в виде дерева (рис. 12.1), где каждое ребро помечено буквой, на которой основывается текущая декомпозиция, а конечный узел создается тогда, когда эта буква оказывается единственным элементом декомпозиции.

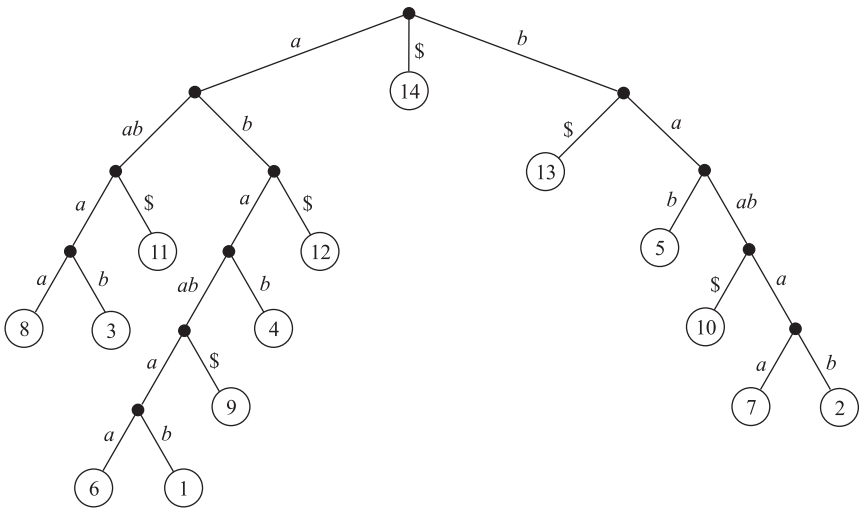


Рис. 12.1. Дерево декомпозиции Крочемора для строки $\boldsymbol{f}_6 = abaababaabaab\$$

Сравнивая дерево на рис. 12.1 с деревом суффиксов, изображенным на рис. 5.2, мы видим, что для преобразования одного в другое достаточно простого переименования меток некоторых ребер. Кроме того, необходимо удостовериться, что в дереве Крочемора все ребра, которые ведут к конечным узлам, представляют суффиксы. Таким образом, расширенная версия алгоритма Крочемора позволяет вычислять все общие префиксы суффиксов $\boldsymbol{x}[\ell..n]$ — эта информация содержится в дереве суффиксов $T_{\boldsymbol{x}}$.

К сожалению, алгоритм Крочемора также обладает и другим свойством, присущим всем алгоритмам построения деревьев суффиксов: он требует для своего выполнения большого объема памяти. Действительно, в силу широкого использования массивов, очередей и двусвязных списков, известные реализации алгоритма Крочемора потребовали не менее $80n$ байт памяти, хотя это количество было недавно урезано наполовину с помощью нового подхода [94]. Напомним, что в подразделе 5.2.5 было сделано предположение о том, что любая реализация дерева суффиксов потребует не менее $10n$ байт памяти, — это нижняя граница объема памяти, необходимой любому алгоритму построения дерева суффиксов.

12.1.2 Алгоритм Мейна и Лоренца

Подход, предпринятый в алгоритме Мейна–Лоренца (Main–Lorentz, сокращенно — алгоритм МЛ), полностью отличается от подхода, используемого в алгоритме Крочемора. Интересно, что алгоритм МЛ работает с неупорядоченным алфавитом, но все равно выполняется за время порядка $\Theta(n \log n)$. Этот алгоритм не требует лексикографического упорядочения букв и подстрок строки $\boldsymbol{x}$, тогда как эти условия являются критическими на первом шаге алгоритма Крочемора. Алгоритм МЛ — это алгоритм типа "разделяй и властвуй" [5], который вычисляет все квадраты подстрок, следовательно, все кратные строки в строковой последовательности $\boldsymbol{x} = \boldsymbol{x}[\ell..n]$, распределяя их на три различных класса.

- Класс C1 — класс квадратов $\boldsymbol{w}^2$, которые начинаются и заканчиваются в подстроке $\boldsymbol{x}[1..\lfloor n/2 \rfloor]$, т.е. $\boldsymbol{w}^2 = \boldsymbol{x}[i..i + 2|\boldsymbol{w}| - 1]$, где $i + 2|\boldsymbol{w}| - 1 \leqslant \lfloor n/2 \rfloor$.
- Класс C2 — класс квадратов $\boldsymbol{w}^2$, которые начинаются и заканчиваются в подстроке $\boldsymbol{x}[\lfloor n/2 \rfloor + 1..n]$, т.е. $\boldsymbol{w}^2 = \boldsymbol{x}[i..i + 2|\boldsymbol{w}| - 1]$, где $i > \lfloor n/2 \rfloor$.
- Класс C3 — класс квадратов $\boldsymbol{w}^2$, которые начинаются в подстроке $\boldsymbol{x}[1..\lfloor n/2 \rfloor]$ и заканчиваются в подстроке $\boldsymbol{x}[\lfloor n/2 \rfloor + 1..n]$, т.е. здесь $\boldsymbol{w}^2 = \boldsymbol{x}[i..i+2|\boldsymbol{w}|-1]$, где $1 \leqslant i \leqslant \lfloor n/2 \rfloor$ и $\lfloor n/2 \rfloor < i + 2|\boldsymbol{w}| - 1 \leqslant n$.

Обозначим $\boldsymbol{u} = [1..\lfloor n/2 \rfloor]$ и $\boldsymbol{v} = \boldsymbol{x}[\lfloor n/2 \rfloor + 1..n]$. Основная идея алгоритма МЛ заключается в вычислении квадратов подстрок в строке $\boldsymbol{x} = \boldsymbol{uv}$ на основе эффективного вычисления новых квадратов класса C3. Действительно, рекурсивное разбиение строки $\boldsymbol{x}$ на половинные подстроки сводит вычисление *всех* квадратов до квадратов класса C3. Эта идея приводит к следующему прямому рекурсивному алгоритму.

Алгоритм 12.1.2 (Алгоритм Мейна–Лоренца)

```
▷ Вычисление всех квадратов подстрок в строке x = x[1..n]
procedure ML(x)
    if |x| > 1 then
        ▷ Вычисление квадратов класса C3
        C3(x[1..⌊n/2⌋], x[⌊n/2⌋ + 1..n])
        ▷ Применение рекурсии
        ML(x[1..⌊n/2⌋])
        ML(x[⌊n/2⌋ + 1..n])
```

Далее мы увидим, что для произвольных строк $\boldsymbol{u}$ и $\boldsymbol{v}$ процедуру C3$(\boldsymbol{u}, \boldsymbol{v})$ можно реализовать за время $\Theta(|\boldsymbol{uv}|)$. Если обозначить через t_n максимальное время, необходимое для выполнения процедуры ML на любой строке длиной n, то очевидно, что для $n > 1$ существует константа k_2, которая удовлетворяет условию

$$t_n \leqslant k_2 n + 2t_{\lceil n/2 \rceil}, \tag{12.2}$$

тогда как для $0 \leqslant n \leqslant 1$ существует константа k_1, такая, что

$$t_1 \leqslant k_1. \tag{12.3}$$

Как показано в упражнении 12.1.5, разрешая неравенства (12.2) и (12.3) относительно t_n, получим, что $t_n \in O(n \log n)$. Но поскольку время, необходимое для выполнения процедуры C3, в точности пропорционально величине $|\boldsymbol{uv}|$, то, следовательно, существуют константы k'_1 и k'_2, которые удовлетворяют условиям

$$t'_1 \geqslant k'_1, t'_n \geqslant k'_2 n + 2t'_{\lceil n/2 \rceil}, n > 1, \tag{12.4}$$

где t'_n — *минимальное* время, необходимое для выполнения алгоритма МЛ на любой строке длиной n. Здесь опять, разрешая рекуррентное соотношение (12.4), можно показать, что $t'_n \in \Omega(n \log n)$. Таким образом, имеем следующую теорему.

Теорема 12.1.4. Если процедура C3 выполняется за линейное время, то алгоритм 12.1.2 вычисляет все квадраты подстрок в произвольной строке $\boldsymbol{x} = \boldsymbol{x}[1..n]$ за время $\Theta(n \log n)$. ■

Для описания процедуры C3$(\boldsymbol{u}, \boldsymbol{v})$ вначале отметим, что новые квадраты $\boldsymbol{w}^2$ в строке $\boldsymbol{uv}$ могут быть двух видов:

- ***правый квадрат***, когда второе вхождение $\boldsymbol{w}$ в строку $\boldsymbol{uv}$ полностью находится в подстроке $\boldsymbol{v}$;
- ***левый квадрат***, когда в подстроке $\boldsymbol{u}$ имеется непустой префикс второго вхождения $\boldsymbol{w}$.

Например, подстроки $\boldsymbol{u} = aba$ и $\boldsymbol{v} = ba$ порождают правый квадрат $\boldsymbol{w}^2 = (ba)^2$ и левый квадрат $\boldsymbol{w}^2 = (ab)^2$. Здесь мы опишем процесс вычисления правых квадратов и оставим для упражнения 12.1.10 симметричный процесс вычисления левых квадратов.

Процедура C3 реализуется как комбинация процессов вычисления правых и левых квадратов. Пусть $\boldsymbol{u} = \boldsymbol{u}[\ell..n_1]$ и $\boldsymbol{v} = \boldsymbol{v}[\ell..n_2]$. Чтобы в $\boldsymbol{uv}$ найти все правые квадраты, необходимо вычислить два массива, которые являются некоторым подобием массива граней (раздел 1.3). Для определения этих массивов напомним (подраздел 5.2.1), что $\mathrm{lcp}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ обозначает длину наибольшего общего префикса строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Используем подобное обозначение $\mathrm{lcs}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ для длины наибольшего общего суффикса строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. (Не путать lcs с LCS — наибольшей общей *подпоследовательностью* из главы 9!) Новые массивы можно кратко определить следующим образом.

- Массив LP$[2..n_2 + 1]$: для каждой позиции $i = 2, 3, \ldots, n_2$ строки $\boldsymbol{v}$ LP$[i] =$ $= \mathrm{lcp}(\boldsymbol{v}[i..n_2], \boldsymbol{v})$, при этом для $i = n_2 + 1$ принимаем, что LP$[i] = 0$.
- Массив LS$[1..n_2]$: для каждой позиции $i = 1, 2, \ldots, n_2$ строки $\boldsymbol{v}$ LS$[i] =$ $= \mathrm{lcs}(\boldsymbol{v}[1..i], \boldsymbol{u})$.

Например, для строки $\boldsymbol{v} = abaababa$ массив $\mathrm{LP}[2..9] = 01303010$. Если $\boldsymbol{u} = abaab$, тогда $\mathrm{LS}[1..8] = 02005020$. Приближенно можно сказать, что массив LP рекуррентно определяет максимальные длины префиксов $\boldsymbol{v}$ внутри самой $\boldsymbol{v}$, а массив LS определяет вхождения суффиксов $\boldsymbol{u}$ максимальной длины в строку $\boldsymbol{v}$.

Эти массивы, которые похожи на массивы граней, можно вычислить за время $\Theta(n_2)$. Позже мы опишем процесс их вычисления, но сначала покажем, каким образом эти массивы можно использовать для вычисления правых квадратов строк $\boldsymbol{u}$ и $\boldsymbol{v}$. Для этого необходима следующая лемма.

Лемма 12.1.5. Пусть $\boldsymbol{u}[1..n_1]$ и $\boldsymbol{v}[1..n_2]$ — произвольные непустые строки и пусть $p \leqslant n_2$ и $i \in p..2p-1$ — положительные целые числа. В $\boldsymbol{uv}$ имеется квадрат длиной $2p$, заканчивающийся на i-й позиции строки $\boldsymbol{v}$, тогда и только тогда, когда

$$2p - \mathrm{LS}[p] \leqslant i \leqslant p + \mathrm{LP}[p+1].$$

Доказательство. В строке $\boldsymbol{uv}$ для любого квадрата $\boldsymbol{w}^2$ длиной $2p$, заканчивающегося на i-й позиции, первое вхождение $\boldsymbol{w}$ должно начинаться с буквы $\boldsymbol{u}[n_1 - 2p + i + 1]$, а второе вхождение — с буквы $\boldsymbol{v}[i - p + 1]$. Поэтому такой квадрат имеет "право на существование" тогда и только тогда, когда

$$\boldsymbol{u}[n_1 - 2p + i + 1..n_1]\boldsymbol{v}[1..i-p] = \boldsymbol{v}[i-p+1..i];$$

т.е. только тогда, когда

- $\boldsymbol{u}[n_1 - 2p + i + 1..n_1] = \boldsymbol{v}[i - p + 1..p]$ и
- $\boldsymbol{v}[1..i-p] = \boldsymbol{v}[p+1..i]$.

Отметим, что если $i = p$, вследствие чего квадрат формируется из суффикса $\boldsymbol{w}$ строки $\boldsymbol{u}$ и префикса $\boldsymbol{w}$ строки $\boldsymbol{v}$, второе условие сводится к утверждению, что $\varepsilon = \varepsilon$, и поэтому становится ненужным.

Первое условие, которое говорит о том, что строка $\boldsymbol{v}[i-p+1..p]$ длиной $2p-i$ является суффиксом строки $\boldsymbol{u}$, выполняется только тогда, когда $\mathrm{LS}[p] \geqslant 2p - i$, т.е. только в случае выполнения неравенства $2p - \mathrm{LS}[p] \leqslant i$.

Аналогично, как показано в упражнении 12.1.7, второе условие эквивалентно неравенству $i \leqslant p + \mathrm{LP}[p+1]$, что и завершает доказательство. ■

Правый квадрат возможен только для тех значений p и i, которые удовлетворяют условиям леммы 12.1.5. Кроме того, для каждого допустимого значения $p \in 1..n_2$ соответствующие допустимые значения величины i могут находиться только в интервале $2p - \mathrm{LS}[p]..p + \mathrm{LP}[p+1]$. Поэтому при условии, что массивы LP и LS вычислены заранее, фактические значения i определяются за константное время путем вывода границ этих интервалов всякий раз, когда $\mathrm{LP}[p+1] + \mathrm{LS}[p] \geqslant p$. Следовательно, суммируя по всем допустимым значениям p, правые квадраты можно вычислить за время $\Theta(n_2)$.

Оставим для упражнения 12.1.8 запись простого алгоритма, который использует массивы LP и LS для выполнения этих вычислений.

Для вычисления левых квадратов нужны аналогичные массивы.

- Массив $\mathrm{LP}'[2..n_1]$: для каждой позиции $i = 2, 3, \ldots, n_1$ строки $\boldsymbol{u}$ $\mathrm{LP}'[i] = \mathrm{lcp}(\boldsymbol{u}[i..n_1], \boldsymbol{v})$.
- Массив $\mathrm{LS}'[1..n_1 - 1]$: для каждой позиции $i = 1, 2, \ldots, n_1 - 1$ строки $\boldsymbol{u}$ $\mathrm{LS}'[i] = \mathrm{lcs}(\boldsymbol{u}[\ell..i], \boldsymbol{u})$.

Как показано в упражнении 12.1.9, для массивов LP' и LS' имеет место аналог леммы 12.1.5. Поэтому время вычисления всех левых квадратов в строке $\boldsymbol{uv}$ имеет порядок $\Theta(n_1)$.

Как было отмечено выше, процедура C3 вычисляет правые и левые квадраты класса C3 за время $\Theta(n_1 + n_2) = \Theta(|\boldsymbol{uv}|)$. Таким образом, если массивы $|LP[2..n_2 + 1]$ и $\mathrm{LS}[1..n_2]$ можно вычислить за время $\Theta(n_2)$, а аналогичные массивы $\mathrm{LP}'[2..n_1]$ и $\mathrm{LS}'[1..n_1 - 1]$ — за время $\Theta(n_1)$, тогда, в соответствии с теоремой 12.0.1, для выполнения алгоритма МЛ необходимо время порядка $\Theta(n \log n)$.

Далее покажем, каким образом можно эффективно вычислить необходимые для алгоритма массивы.

Вычисление массивов LP, LS, LP′, LS′

Вспомним, что для произвольной строки $\boldsymbol{v}[1..n_2]$ и любого целого числа $i \in$ $\in 1..n_2 + 1$ элемент $\mathrm{LP}[i]$ равен длине наибольшего префикса подстроки $\boldsymbol{v}[i..n_2]$, который совпадает с префиксом самой строки $\boldsymbol{v}$. Будем вести вычисление массива LP слева направо для возрастающих значений i, считая, что при вычислении элемента $\mathrm{LP}[i]$ все элементы $\mathrm{LP}[k]$, где $2 \leqslant k < i$, уже вычислены. Предположим, что для некоторого значения $i \geqslant 3$ существует такое целое число $k \in 2..i - 1$, что $k + \mathrm{LP}[k] > i$. Как показано на рис. 12.2, это означает, что существует подстрока $\boldsymbol{w} = \boldsymbol{v}[i..k + \mathrm{LP}[k] - 1]$ длиной $\ell = k + \mathrm{LP}[k] - i$, совпадающая с суффиксом $\boldsymbol{v}[i - k + 1.. \mathrm{LP}[k]]$ префикса $\boldsymbol{v}[1.. \mathrm{LP}[k]]$.

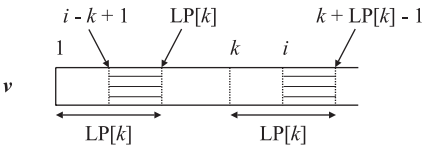


Рис. 12.2. Вычисление $\mathrm{LP}[i]$ при $k + \mathrm{LP}[k] > i$

Теперь рассмотрим элемент $\mathrm{LP}[i-k+1]$, равный длине наибольшей подстроки, совпадающей с префиксом строки $\boldsymbol{v}$. Если $\mathrm{LP}[i - k + 1] < |\boldsymbol{w}| = \ell$, тогда должно выполняться равенство

$$\mathrm{LP}[i] = \mathrm{LP}[i - k + \ell]. \tag{12.5}$$

С другой стороны, если имеет место неравенство $\mathrm{LP}[i-k+1] \geqslant \ell$, тогда

$$\mathrm{LP}[i] \geqslant \ell, \tag{12.6}$$

и поэтому $\boldsymbol{w}$ — префикс наибольшей (при данном i) подстроки, которая совпадает с префиксом строки $\boldsymbol{v}$. Следовательно, если существует k, такое, что $k+\mathrm{LP}[k] > i$, то элемент $\mathrm{LP}[i]$ вычисляется точно согласно выражению (12.5) или, в соответствии с (12.6), для него установлена нижняя граница.

Конечно, если нет такого целого числа $k \in 2..i-1$, что $k + \mathrm{LP}[k] > i$, тогда единственной нижней границей для $\mathrm{LP}[i]$ будет нуль, т.е. в этом случае

$$\mathrm{LP}[i] \geqslant 0. \tag{12.7}$$

Три случая (12.5)–(12.7) в алгоритме вычисления массива LP необходимо рассмотреть отдельно. Как показано в алгоритме 12.1.3, после определения нижней границы для $\mathrm{LP}[i]$ (случаи (12.6) и (12.7) обрабатываются одинаково) путем посимвольного сравнения вычисляется точное значение элемента $\mathrm{LP}[i]$. Заметим, что алгоритм может работать и со значениями k, которые меньше i, лишь бы это k доставляло максимум выражению $k+\mathrm{LP}[k]$. Для удобства вычисления конечных позиций в массиве LP в конец строки $\boldsymbol{v}$ добавлен символ $\$_2$, который не совпадает ни с одной буквой алфавита.

Алгоритм 12.1.3 (Вычисление массива LP)

```
▷ Вычисление массива LP[2..n2 + 1], соответствующего строке v[1..n2]$2
▷ Вначале вычисляется LP[2] (случай (12.7)) и инициализуется k
LP[2] ← 0
while v[LP[2] + ℓ] = v[LP[2] + 2] do LP[2] ← LP[2] + 1
k ← 2
for i ← 3 to n2 + 1 do
    ℓ ← k + LP[k] − i        ▷ возможно отрицательное значение!
    if LP[i − k + 1] < ℓ then
        ▷ Случай (12.5)
        LP[i] ← LP[i − k + 1]
    else
        ▷ Случаи (12.6) и (12.7) — в точности так же, как для i = 2!
        LP[i] ← max{0, ℓ}
        while v[LP[i] + ℓ] = v[LP[i] + i] do LP[i] ← LP[i] + 1
        k ← i
```

Проведенное выше обсуждение доказывает правильность алгоритма 12.1.3. Он реализует все эти три случая (12.5)–(12.7) и делает это даже для $\boldsymbol{v}[n_2+1] = \$_2$.

Для того чтобы оценить временну́ю сложность алгоритма, заметим, что в цикле **for** все операторы, за исключением операторов цикла **while**, выполняются за

константное время. Таким образом, время выполнения алгоритма равно $\Theta(n_2)$ плюс время выполнения циклов **while**, которое пропорционально числу приращений значений $\mathrm{LP}[i]$. Но каждое приращение $\mathrm{LP}[i]$ увеличивает на единицу значение $k + \mathrm{LP}[k]$, которое в последующих итерациях будет в строке $\boldsymbol{v}$ минимальной позицией буквы, подлежащей сравнению. Поскольку начальное значение k равно 2, то, очевидно, оператор $\mathrm{LP}[i] \leftarrow \mathrm{LP}[i] + 1$ выполняется не более $n_2 - 1$ раз в *обоих* циклах **while**. Отсюда следует, что алгоритм вычисления массива LP выполняется за время порядка $\Theta(n_2)$.

Не трудно применить подход, реализованный в алгоритме 12.1.3, для вычисления массива $\mathrm{LP}'[2..n_1]$, т.е. для вычисления длины наибольшего префикса подстроки $\boldsymbol{u}[i..n_1]$ (а не префикса подстроки $\boldsymbol{v}[i..n_2]$), который совпадает с префиксом строки $\boldsymbol{v}$. Здесь также для каждого $i \geqslant 3$ определяется целое число $k \in 2..i-1$, которое обеспечивает максимальное значение выражению $k + \mathrm{LP}'[k]$. Как показано на рис. 12.3, это означает, что для $k + \mathrm{LP}'[k] > i$ существует подстрока $\boldsymbol{w} = \boldsymbol{u}[i..k + \mathrm{LP}'[k] - 1]$ длиной $\ell' = k + \mathrm{LP}'[k] - i$, совпадающая с подстрокой $\boldsymbol{v}[i - k + 1.. \mathrm{LP}'[k]]$.

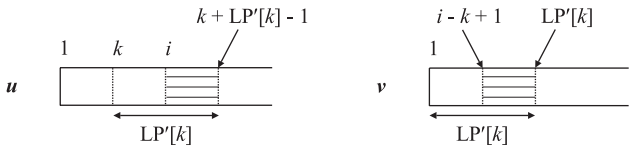


Рис. 12.3. Вычисление $\mathrm{LP}'[i]$ при $k + \mathrm{LP}'[k] > i$

Здесь также имеют место три случая, подобные (12.5)–(12.7). Предположим, что элемент $\mathrm{LP}[2..n_2 + 1]$ уже вычислен и остается неизменным, если $\mathrm{LP}[i - k + + 1] < \ell'$, тогда

$$\mathrm{LP}'[i] = \mathrm{LP}[i - k + 1]. \tag{12.8}$$

В противном случае

$$\mathrm{LP}'[i] \geqslant \ell', \tag{12.9}$$

и если $k + \mathrm{LP}'[k] \leqslant i$, тогда

$$\mathrm{LP}'[i] \geqslant 0. \tag{12.10}$$

Реализация трех случаев (12.8)–(12.10) приводит к алгоритму, почти идентичному алгоритму 12.1.3 и представленному в следующем листинге. С помощью таких же аргументов, что были использованы для обоснования оценки временно́й сложности алгоритма 12.1.3, здесь нетрудно доказать, что алгоритм вычисления массива LP' выполняется за время порядка $\Theta(n_1)$.

Алгоритм 12.1.4 (Вычисление массива LP')

▷ *Вычисление массива* $\mathrm{LP}'[2..n_1+1]$*, соответствующего строкам* $\boldsymbol{u}[1..n_1]\$_1$
▷ *и* $\boldsymbol{v}[\ell..n_2]\$_2$ *при условии, что массив* $\mathrm{LP}[2..n_2+1]$ *уже вычислен*
▷ *Сначала вычисляется массив* $\mathrm{LP}'[2]$
▷ *и инициализируется* k
$\mathrm{LP}'[2] \leftarrow 0$
while $\boldsymbol{v}[\mathrm{LP}'[2]+1] = \boldsymbol{u}[\mathrm{LP}'[2]+2]$ **do** $\mathrm{LP}'[2] \leftarrow \mathrm{LP}'[2]+1$
$k \leftarrow 2$
for $i \leftarrow 3$ **to** n_1 **do**
    $\ell' \leftarrow k + \mathrm{LP}'[k] - i$ ▷ *возможно отрицательное значение!*
    ▷ *Отметим, что использование* LP *предпочтительнее* LP'
    if $\mathrm{LP}[i-k+\ell] < \ell'$ **then**
        $\mathrm{LP}'[i] \leftarrow \mathrm{LP}[i-k+1]$
    else
        $\mathrm{LP}'[i] \leftarrow \max\{0, \ell'\}$
        while $\boldsymbol{v}[\mathrm{LP}'[i]+1] = \boldsymbol{u}[\mathrm{LP}'[i]+i]$ **do** $\mathrm{LP}'[i] \leftarrow \mathrm{LP}'[i]+1$
        $k \leftarrow i$

Разработку алгоритмов для вычисления массивов LS и LS' оставим для упражнения 12.1.14.

Заключение

Если для вычисления массивов LP, LP', LS и LS' используются алгоритмы с линейным временем выполнения, тогда можно утверждать, что процедура $\mathrm{C3}(\boldsymbol{u}, \boldsymbol{v})$ выполняется за время $\Theta(|\boldsymbol{uv}|)$. Следовательно, согласно теореме 12.1.4, алгоритм 12.1.2 выполняется за время $\Theta(n \log n)$. Однако вспомним, что алгоритм 12.1.1 выполняется за время $O(n \log n)$. Но с другой стороны, поскольку алгоритм 12.1.2 оперирует только простыми матричными структурами, он требует очень небольшого объема памяти — определенно меньше $40n$ байт, необходимых для алгоритма 12.1.1 даже в его самом экономичном варианте [94].

Интересно сравнить результаты, выводимые алгоритмами Крочемора и Мейна–Лоренца. В алгоритме Крочемора каждая кратная строка выводится в нормальной форме, используя кодирование (i, p^*, r^*). Однако напомним, что хотя алгоритм Крочемора обрабатывает каждый квадрат в отдельности, он объединяет смежные квадраты до тех пор, пока не будет получен максимальный показатель степени, который может превышать значение 2. С другой стороны, алгоритм МЛ выводит только квадраты, но, принимая во внимание лемму 12.1.5 и ее аналоги для массивов LP' и LS', он выводит их в блоках, которые близки к понятию "серий",

введенному в разделе 2.3. Например, рассмотрим строку

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x} =$	a	b	a	a	b	a	b	a	a	b	a	b	a

Для кратных строк с периодом $p = 5$, алгоритм Крочемора выдаст результаты:

$$(1, 5, 2), (2, 5, 2), (3, 5, 2), (4, 5, 2),$$

соответствующие квадрату $(abaab)^2$ и его трем перестановкам, полученным в результате циклического сдвига, $(baaba)^2$, $(aabab)^2$ и $(ababa)^2$. Но алгоритму МЛ в этом случае необходимо вывести только тройку

$$(p, \text{первая}, \text{последняя}) = (5, 10, 13),$$

указывающую на то, что квадраты с периодом 5 заканчиваются на позициях 10–13.

Из этого видно, что алгоритм МЛ предвосхищает понятие серии, которое было введено в [168] и применено там для уменьшения асимптотической сложности проблемы кратных строк. Мы исследуем эти вопросы в следующем разделе.

Упражнения 12.1

1. Докажите следующую "декомпозицию" леммы 12.1.2: ни одна из позиций строки $\boldsymbol{x}$ не может входить в более чем $\lceil \log_2(n - \alpha' + 2) \rceil$ малые последовательности, где α' — количество различных букв в строке $\boldsymbol{x}$.
2. *Вектор Париха* (Parikh), или *частотный вектор* строки $\boldsymbol{x}[1..n]$, которая содержит α' различных букв, — это целочисленный вектор $\phi_{\boldsymbol{x}} = \phi_{\boldsymbol{x}}[1..\alpha']$, где $\phi_{\boldsymbol{x}}[h]$ равно числу вхождений h-й буквы в строку $\boldsymbol{x}$ [193]. Выразите сложность алгоритма Крочемора через n и вектора Париха. Используйте это выражение как основу для обсуждения поведения алгоритма Крочемора в тех случаях, когда α' велико.
3. В этом разделе часто использовался термин "последовательность", но без его формального определения. Имея в виду, что о последовательности можно думать как своего рода о строке на основе индексированного алфавита (раздел 4.1), предоставьте отсутствующие определения для последовательности, подпоследовательности и, возможно, для других подобных терминов.
4. Покажите, что для выполнения декомпозиции всех последовательностей уровня ℓ на подпоследовательности уровня $\ell+1$ необходимо не более $2n$ индексов последовательностей. Можно ли более точно оценить эту верхнюю границу?

5. Решите неравенства (12.2) и (12.3) и покажите, что $t_n \in O(n \log n)$. Затем на основе неравенств (12.4) докажите, что $t_n \in \Omega(n \log n)$.

Совет. Ели для некоторого неотрицательного числа r выполняются неравенства $2^r < n < 2^{r+1}$, тогда сформулированные выше утверждения будут выполняться для n в том случае, если они выполняются для 2^{r+1}.

6. Предположим, что неравенство (12.2) обобщено в виде неравенства

$$t_n \leqslant k_2 f_n + 2t_{\lceil n/2 \rceil}, f_n \in \Omega(n^2),$$

справедливого для любого $n \geqslant 1$. Одновременно выполняется неравенство (12.3). Покажите, что в этом случае $t_n \in O(f_n)$.

Замечание. Мы используем этот результат в разделе 13.3 для того, чтобы оценить сложность алгоритма Шмидта для аппроксимирующих раппортов.

7. Покажите, что второе условие леммы 12.1.5 эквивалентно неравенству $i \leqslant$ $\leqslant p + \mathrm{LP}[p+1]$.

8. Предположим, что массивы LP и LS вычислены заранее для строк $\boldsymbol{u}[1..n_1]$ и $\boldsymbol{v}[1..n_2]$. Запишите алгоритм вычисления всех правых квадратов в строке $\boldsymbol{uv}$ с временем выполнения $\Theta(n_2)$.

9. Докажите следующий аналог леммы 12.1.5.

> Пусть $\boldsymbol{u}[1..n_1]$ и $\boldsymbol{v}[1..n_2]$ — произвольные строки, а p $(p < n_1)$ и i — положительные целые числа, такие, что $n_1 - 2p < i - 1 < n_1 - p$. Тогда в строке $\boldsymbol{uv}$ имеется квадрат длиной $2p$, начинающийся с i-й позиции строки $\boldsymbol{u}$, только в том случае, если выполняются неравенства
>
> $$(n_1 - p) - \mathrm{LS}'[n_1 - p] \leqslant i - 1 \leqslant (n_1 - 2p) + \mathrm{LP}'[(n_1 - p) + 1].$$

10. Предположим, что для строк $\boldsymbol{u}[1..n_1]$ и $\boldsymbol{v}[1..n_2]$ предварительно вычислены массивы LP' и LS'. Запишите алгоритм вычисления всех левых квадратов в строке $\boldsymbol{uv}$ с временем выполнения $\Theta(n_1)$.

11. Используя алгоритмы вычисления массивов и правых и левых квадратов с линейным временем выполнения, запишите процедуру C3, которая вычисляет все квадраты класса C3 в строке $\boldsymbol{uv}$, где $\boldsymbol{u} = \boldsymbol{u}[1..n_1]$, $\boldsymbol{v} = \boldsymbol{v}[1..n_2]$.

12. Дайте формальное доказательство равенства (12.5).

13. Докажите, что алгоритм 12.1.3 действительно может работать со значениями $k \in 2..i-1$, при которых достигается максимум выражения $k + \mathrm{LP}[k]$.

14. Аналогично алгоритмам вычисления массивов LP и LP', запишите алгоритмы для вычисления массивов LS и LS'.

12.2 Серии

В этом разделе мы рассмотрим два подхода к проблеме кратных строк, которые уменьшают время их обработки и вывода результатов путем вычисления серий (i, p^*, r^*, t), а не кратных строк (i, p^*, r^*). В обоих подходах используется s-факторизация, вычисление которой с использованием деревьев суффиксов было описано в разделе 6.3. Кроме того, оба алгоритма используют массивы LS, LS′, LP, LP′, представленные в подразделе 12.1.2 при описании алгоритма МЛ. Второй алгоритм, алгоритм Колпакова–Кучерова (сокращенно, алгоритм КК), является по существу расширением первого алгоритма, алгоритма Мейна, при этом алгоритм КК вычисляет все серии, а алгоритм Мейна — только ***крайние левые*** серии, т.е. только крайние левые вхождения каждой отдельной серии в строку $\boldsymbol{x}$.

Поскольку оба алгоритма зависят от использования деревьев суффиксов, то для их выполнения в случае произвольного упорядоченного алфавита, неизвестного заранее, потребуется время порядка $O(n \log n)$. Но для небольшого упорядоченного алфавита, известного заранее, обоим алгоритмам, скорее всего, потребуется одинаковое время порядка $\Theta(n)$. Как отмечается в работе [139], основная сложность этих алгоритмов заключается в "памяти, занятой структурой данных, необходимой для вычисления s-факторизации".

Для того чтобы алгоритмы Мейна и КК выполнялись за линейное (относительно длины строки) время, размер вывода должен быть линейным (относительно количества серий). Как мы увидим, для алгоритма Мейна это следует из того факта, что сам процесс вычисления, который определяет крайние левые серии, будет линейным. Это также подтверждается более поздним результатом [88], цитируемым ниже, где доказано, что число различных (следовательно, и крайних левых) квадратов линейно. Гораздо более трудной проблемой является доказательство того, что общее количество серий, крайних левых или других, является линейным (относительно длины строки).

Теорема 12.2.1. Пусть $\rho(n)$ обозначает максимальное количество серий, которые могут содержаться в произвольной строке длиной n, определенной на произвольном алфавите. Существуют положительные константы k_1 и k_2, независимые от n, такие, что для каждого целого $n \geqslant 1$ справедливо неравенство

$$\rho(n) \leqslant k_1 n - k_2 \sqrt{n} \log_2 n. \tag{12.11}$$

Доказательство см. в [139]. ■

Мы не приводим доказательство этого значительного результата, поскольку оно является очень длинным и техническим. Хотя теорема 12.2.1 является существенным продвижением в изучении периодичности в строковых последовательностях, она также является показателем того, сколько еще осталось неизученного.

Доказательство неравенства (12.11) не позволяет сделать какую-либо оценку относительно величины констант k_1 и $k_2 - k_1$ может равняться 10^9, а k_2 может быть равно 10^{-9}. В то же время исчерпывающие компьютерные эксперименты до $n = 60$ [138] показали такие результаты:

1. $\rho(n) < n$;
2. $0 \leqslant \rho(n+1) - \rho(n) \leqslant 2$;
3. среди строк, определенных на алфавите $\{a, b\}$, существует строка без кубов, в которой содержится $\rho(n)$ серий.

Кажется вполне вероятным, что все эти фундаментальные результаты являются истинными, но, похоже, что никто не имеет какого-либо представления, как их доказать!

В недавно опубликованной статье [88] вместо $\rho(n)$ была рассмотрена величина $\sigma(n)$, равная максимальному количеству различных квадратов, которые могут содержаться в любой строке длиной n, и было показано, что $\sigma(n) \leqslant 2n - 8$ для всех $n \geqslant 5$. Как предполагается в [139], между $\rho(n)$ и $\sigma(n)$ может быть некоторая связь, которая может помочь в определении более точных границ величины ρ. В работе [140] доказан один более сильный результат, чем в теореме 12.2.1, а именно: там показано, что сумма показателей всех серий в любой строке длиной n также линейна относительно n.

12.2.1 Крайние левые серии — алгоритм Мейна

Крочемор [70], кажется, был первым, кто осознал, что s-факторизацию [156], использованную в работе [235] для сжатия строк (раздел 6.3), можно также применить для эффективного вычисления кратных строк. В этом подразделе мы описываем алгоритм Мейна [168], который использует s-факторизацию вместе с методами, разработанными в [176] (подраздел 12.1.2) для вычисления крайнего левого вхождения каждой отдельной серии в произвольной строке $\boldsymbol{x}$. Основой этого алгоритма является следующая теорема.

Теорема 12.2.2. Предположим, что строка $\boldsymbol{x}[\ell..n]$ имеет s-факторизацию $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_k$, а $\boldsymbol{r}$ обозначает крайнее левое вхождение серии $\boldsymbol{x}[i..j]$ в строку $\boldsymbol{x}$, конечная позиция j которой находится в факторе $\boldsymbol{w}_h$ для некоторого $h \leqslant k$. Тогда

а) начальная позиция i серии $\boldsymbol{r}$ находится в факторе $\boldsymbol{w}_{h'}$ для некоторого $h' \leqslant h$;

б) если $\boldsymbol{r} = \boldsymbol{r}_L\boldsymbol{u}_R$ для некоторого префикса $\boldsymbol{r}_R$ фактора $\boldsymbol{w}_h$, то

$$|\boldsymbol{r}_L| < 2|\boldsymbol{w}_{h-1}| + |\boldsymbol{w}_h|.$$

Доказательство. Предположим, что утверждение *а*) не выполняется. Тогда серия $\boldsymbol{r}$ будет подстрокой фактора $\boldsymbol{w}_h$, и поэтому $|\boldsymbol{w}_h| \geqslant 2$. В этом случае по

определению s-факторизации должно существовать другое вхождение $\boldsymbol{w}_h$ в строку $\boldsymbol{x}$ слева от фактора $\boldsymbol{w}_h$. Но тогда существует слева другое вхождение серии $\boldsymbol{r}$, что противоречит условию теоремы. Таким образом, утверждение *а*) должно выполняться.

На основании утверждения *а*) можно считать, что $\boldsymbol{r} = \boldsymbol{r}_L\boldsymbol{u}_R$ для некоторой непустой подстроки $\boldsymbol{r}_L$ и некоторого непустого префикса $\boldsymbol{r}_R$ фактора $\boldsymbol{w}_h$. Предположим, что утверждение *б*) не выполняется, поэтому $|\boldsymbol{r}_L| \geqslant 2|\boldsymbol{w}_{h-1}| + |\boldsymbol{w}_h|$. Тогда по крайней мере половина серии $\boldsymbol{r}$ находится слева от фактора $\boldsymbol{w}_{h-1}$. Рассмотрим суффикс $\boldsymbol{v} = \boldsymbol{w}_{h-1}\boldsymbol{u}_R$ серии $\boldsymbol{r}$, как показано на рис. 12.4.

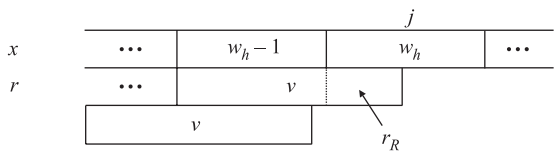


Рис. 12.4. Периодический суффикс $\boldsymbol{v}$ крайней левой серии $\boldsymbol{r}$

Поскольку $\boldsymbol{v}$ является суффиксом серии $\boldsymbol{r}$ с периодом p^* и показателем $r^* \geqslant 2$, а $|\boldsymbol{v}|$ не превышает половины длины серии $\boldsymbol{r}$, то существует другое вхождение $\boldsymbol{v}$ слева на расстоянии p^*, как показано на рис. 12.4. Но поскольку $\boldsymbol{w}_{h-1}$ — собственный префикс $\boldsymbol{v}$, а $\boldsymbol{v}$ — это повторяющаяся подстрока, то $\boldsymbol{w}_{h-1}$ не может быть повторяющейся подстрокой максимальной длины, необходимой для s-факторизации, что является противоречием с условием теоремы. Отсюда заключаем, что утверждение *б*) выполняется. ■

Фактически, эта непритязательная теорема предоставляет намного больше информации, чем кажется на первый взгляд. В частности, заметим, что доказательство утверждения *б*) теоремы требует только того, чтобы серия $\boldsymbol{r}$, не будучи подстрокой фактора $\boldsymbol{w}_h$, заканчивалась в $\boldsymbol{w}_h$, т.е. она не должна быть крайней левой серией в определенном выше смысле. Таким образом, все серии строки $\boldsymbol{x}$ можно разделить на два непересекающихся класса.

- Класс I содержит серии, которые являются подстроками одного фактора $\boldsymbol{w}_h$, $2 \leqslant h \leqslant k$, полученного при s-факторизации строки $\boldsymbol{x}$.
- Класс II содержит серии, которые заканчиваются в некотором факторе $\boldsymbol{w}_h$, $h \geqslant 2$, и начинаются в некотором предыдущем факторе $\boldsymbol{w}_{h'}, h' \leqslant h$.

В этом подразделе покажем, как вычислить все серии класса II, серии класса I рассмотрим в следующем подразделе.

Проявив некоторую интуицию относительно серий класса II, рассмотрим наш универсальный пример строки Фибоначчи f_6, s-факторизация которой имеет вид

$$\boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_6 = a/b/a/aba/baaba/ab.$$

Тогда легко связать серии класса II с соответствующими факторами следующим образом:

$$(aba)^2(\boldsymbol{w}_1 - \boldsymbol{w}_4), a^2(\boldsymbol{w}_3 - \boldsymbol{w}_4), (ab)^2a(\boldsymbol{w}_3 - \boldsymbol{w}_4),$$
$$(abaab)^2a(\boldsymbol{w}_1 - \boldsymbol{w}_5), (aba)^2ab(\boldsymbol{w}_4 - \boldsymbol{w}_6).$$

Отметим, что серия класса I aa из $\boldsymbol{w}_5$ в этот список не включена.

Алгоритм 12.2.1 (Алгоритм Мейна)

▷ *Вычисление серий класса II в строке* $\boldsymbol{x} = \boldsymbol{x}[\ell..n]$
▷ *Выполнение алгоритма 6.3.1*
вычисление s-факторизации $\boldsymbol{x} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_k$
$j^* \leftarrow 0$;
for $h \leftarrow 2$ **to** k **do**
    ▷ *Вычисление префикса* $\boldsymbol{r}_L^*$ *максимальной длины серии,*
    ▷ *заканчивающейся в* $\boldsymbol{w}_h$ *(теорема 12.2.2)*
    $j^* \leftarrow j^* + |\boldsymbol{w}_{h-1}|$
    $\ell \leftarrow \min\{j^*, 2|\boldsymbol{w}_{h-1}| + |\boldsymbol{w}_h|\}$
    $\boldsymbol{r}_L^* \leftarrow \boldsymbol{x}[j^* - \ell + 1..j^*]$
    ▷ *Вычисление серий класса II, заканчивающихся в* $\boldsymbol{w}_h$
    calcruns$(\boldsymbol{r}_L^*, \boldsymbol{w}_h)$

Заметим, что на основании утверждения *б*) теоремы 12.2.2 длина префикса $\boldsymbol{r}_L^*$ каждой серии класса II ограничена значением $2|\boldsymbol{w}_{h-1}| + |\boldsymbol{w}_h|$. Как оказывается, это ограничение позволяет применить метод алгоритма МЛ для вычисления всех серий класса II, заканчивающихся в любом факторе $\boldsymbol{w}_h$, $h \geqslant 2$, за время, пропорциональное $2|\boldsymbol{w}_{h-1}\boldsymbol{w}_h|$. Поскольку

$$2n \leqslant \sum_{h=2}^{k} 2\,|w_{h-1}w_h| < 4n, \tag{12.12}$$

отсюда следует, что все серии класса II в строке $\boldsymbol{x}$ можно вычислить за время $\Theta(n)$ на основе предварительно вычисленной s-факторизации. Поэтому мы вправе формулировать следующую теорему.

Теорема 12.2.3. При наличии ранее вычисленной s-факторизации $\boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_k$ строки $\boldsymbol{x} = \boldsymbol{x}[\ell..n]$ и при условии линейного времени выполнения процедуры *calcruns* алгоритм 12.2.1 вычисляет все серии класса I в строке $\boldsymbol{x}$ за время порядка $\Theta(n)$. ■

Сделаем важное замечание. Как мы видим, каждый запрос к программе *calcruns*$(\boldsymbol{u}, \boldsymbol{v})$ приводит к выводу не более чем $|\boldsymbol{u}| + |\boldsymbol{v}|$ серий класса II. Тогда оценка верхней границы, задаваемая неравенствами (12.12), применима также

к общему количеству серий класса II. Таким образом, общее количество серий класса II не может превышать $4n$. Это более удобная оценка, чем задаваемая теоремой 2.2.1. Но эта оценка, к сожалению, также не полна, поскольку не учитывает серии класса I.

Чтобы завершить описание алгоритма Мейна, осталось объяснить работу процедуры *calcruns*$(\boldsymbol{u}, \boldsymbol{v})$, вычисляющей в строке $\boldsymbol{uv}$ все серии, которые не принадлежат полностью ни $\boldsymbol{u}$, ни $\boldsymbol{v}$. Пусть $\boldsymbol{u} = \boldsymbol{u}[1..n_1]$ и $\boldsymbol{v} = \boldsymbol{v}[1..n_2]$. Напомним, что в подразделе 12.1.2 мы различали (и отдельно вычисляли) правые и новые квадраты в зависимости от того, находится ли второе вхождение образующей квадрата полностью в пределах $\boldsymbol{v}$. Здесь вводится подобное различие, разделяя процедуру *calcruns* на две отдельные процедуры *calcR* и *calcL*, которые вычисляют "правые максимальные" и "левые максимальные" серии соответственно, определяемые следующим образом.

- **Правая максимальная серия** — эта серия класса II из $\boldsymbol{uv}$ имеет суффикс в $\boldsymbol{v}$, по меньшей мере, длиной p, где p — период этой серии.
- **Левая максимальная серия** — эта серия класса II из $\boldsymbol{uv}$ имеет непустой суффикс в $\boldsymbol{v}$ длиной $\ell < p$.

В приведенном выше примере строки $\boldsymbol{f}_6$ все перечисленные серии, за исключением $(aba)^2ab$, являются правыми максимальными сериями; $(aba)^2ab$ — это левая максимальная серия, поскольку она имеет период $p = 3$.

Аналогия с подразделом 12.1.2 продолжается. Определим два массива, соответствующие правым максимальным сериям, следующим образом.

- Массив $\mathrm{LP}[2..n_2 + 1]$: для каждой позиции $i = 2, 3, \ldots, n_2$ в $\boldsymbol{v}$ $\mathrm{LP}[i] = \mathrm{lcp}(\boldsymbol{v}[i..n_2], \boldsymbol{v})$, тогда как для $i = n_2 + 1$ $\mathrm{LP}[i] = 0$.
- Массив $\mathrm{LS}[\ell..n_2]$: для каждой позиции $i = 1, 2, \ldots, n_2$ в $\boldsymbol{v}$ $\mathrm{LS}[i] = \mathrm{lcs}(\boldsymbol{uv}[\ell..i], \boldsymbol{u})$.

Определение массива LP идентично тому, которое было дано в подразделе 12.1.2; определение массива LS немного изменено — вместо суффикса $\boldsymbol{v}[1..i]$ в нем рассматривается суффикс $\boldsymbol{uv}[1..i]$.

Подобным образом определяются массивы LP' и LS', соответствующие левым максимальным сериям.

- Массив $\mathrm{LP}'[2..n_1]$: для каждой позиции $i = 2, 3, \ldots, n_1$ в $\boldsymbol{u}$ $\mathrm{LP}'[i] = \mathrm{lcp}(\boldsymbol{u}[i..n_1]\boldsymbol{v}, \boldsymbol{v})$.
- Массив $\mathrm{LS}'[\ell..n_1 - 1]$: для каждой позиции $i = 1, 2, \ldots, n_1 - 1$ в $\boldsymbol{u}$ $\mathrm{LS}'[i] = \mathrm{lcs}(\boldsymbol{u}[\ell..i], \boldsymbol{u})$.

В оставшейся части этого подраздела мы ограничимся описанием только процедуры *calcR* и ее массивов LP и LS, оставляя процедуру *calcL* вместе с массивами LP' и LS' для упражнений 12.2.3 и 12.2.4. Кроме того, опустим описание алгоритмов вычисления массивов LP и LS, поскольку они отличаются только

незначительными деталями от аналогичных алгоритмов, описанных в подразделе 12.1.2. Далее будем предполагать, что каждый из этих четырех массивов можно вычислить за время, пропорциональное их размерам.

Чтобы понять работу процедуры *calcR*, необходима следующая лемма, подобная лемме 12.1.5.

Лемма 12.2.4. Пусть $\boldsymbol{u}[\ell..n_1]$ и $\boldsymbol{v}[\ell..n_2]$ — произвольные строки, а $\boldsymbol{r} = \boldsymbol{r}_L\boldsymbol{r}_R$ — подстрока строки $\boldsymbol{uv}$, такая, что $\boldsymbol{r}_L$ является непустым суффиксом строки $\boldsymbol{u}$, а $\boldsymbol{r}_R$ — непустым префиксом строки $\boldsymbol{v}$. Предположим, что $|\boldsymbol{r}_R| \geqslant p$ для некоторого целого $p \in 1..\lfloor|\boldsymbol{r}|/2\rfloor$. Тогда $\boldsymbol{r}$ имеет период p тогда и только тогда, когда выполняются следующие условия:

а) $|\boldsymbol{r}_L| \leqslant \mathrm{LS}[p]$;

б) $|\boldsymbol{r}_R| \leqslant p + \mathrm{LP}[p+1]$.

Доказательство. Для того чтобы серия $\boldsymbol{r}$ имела период p, необходимо и достаточно, чтобы серия $\boldsymbol{r}$ имела грань длиной $r - p$, где $r = |\boldsymbol{r}|$. В соответствии с условиями теоремы имеем $r - p \geqslant r - |\boldsymbol{r}_R| = |\boldsymbol{r}_L|$. Поэтому эта грань будет существовать тогда, когда выполняются условия

А) $\boldsymbol{r}[1..|\boldsymbol{r}_L|] = \boldsymbol{r}[p+1..p+|\boldsymbol{r}_L|]$;

Б) $\boldsymbol{r}[|\boldsymbol{r}_L|+1..r-p] = \boldsymbol{r}[p+|\boldsymbol{r}_L|+1..r]$.

Условие А) эквивалентно утверждению, что подстрока $\boldsymbol{r}[p+1..p+|\boldsymbol{r}_L|]$ является суффиксом $\boldsymbol{u}$. Но эта подстрока также является суффиксом подстроки $\boldsymbol{uv}[1..p]$. Поэтому условие А) выполняется только тогда, когда длина $|\boldsymbol{r}_L|$ этого суффикса не превышает $\mathrm{LS}[p]$. Таким образом, условие А) эквивалентно условию *а*).

Условие Б) эквивалентно утверждению, что подстрока $\boldsymbol{r}[p+|\boldsymbol{r}_L|+1..r]$ является префиксом $\boldsymbol{v}$. Но эта подстрока является префиксом (возможно, пустым) подстроки $\boldsymbol{v}[p+1..n_2]$. Поэтому условие Б) выполняется только тогда, когда длина $r - p - |\boldsymbol{r}_L|$ этого префикса не превышает $\mathrm{LP}[p+1]$, что эквивалентно условию *б*). ■

Заметим, что если подстрока $\boldsymbol{r}$, описанная в этой лемме, существует, то она может быть записана в следующем виде:

$$\boldsymbol{r} = (\boldsymbol{r}[1..p])^k\boldsymbol{r}[1..p'] \tag{12.13}$$

для некоторых целых $k \geqslant 2$, $p' \in 0..p-1$. Кроме того, лемма говорит, что существует максимальной длины подстрока $\boldsymbol{r}^* = \boldsymbol{r}^*_L\boldsymbol{r}^*_R$ с левой частью $\boldsymbol{r}^*_L$, удовлетворяющей условию

$$|\boldsymbol{r}^*_L| = \mathrm{LS}[p] > 0, \tag{12.14}$$

и с правой частью $\boldsymbol{r}^*_R$, удовлетворяющей условию

$$|\boldsymbol{r}^*_R| = p + \mathrm{LP}[p+1]. \tag{12.15}$$

Из определения 2.3.3 следует, что серия должна быть представима в форме (12.13), но, кроме того, она должна быть непродолжаемой, т.е. она не может быть собственной подстрокой любой другой подстроки с тем же периодом p. Поскольку, в соответствии с теоремой 12.2.2 и нашим выбором $\boldsymbol{uv}$ можем предполагать, что каждая серия класса II не может быть продолжена влево или вправо за пределы $\boldsymbol{uv}$, то, следовательно, подстрока $\boldsymbol{r}^*$, определяемая условиями (12.14) и (12.15), имеет максимальную длину и непродолжаема в пределах строки $\boldsymbol{x}$, поэтому она является серией строки $\boldsymbol{x}$. Более того, она, по определению, является правой максимальной серией.

Таким образом, имея заранее вычисленные значения $\mathrm{LP}[p+1]$ и $\mathrm{LS}[p]$, соответствующие $\boldsymbol{u}$ и $\boldsymbol{v}$ для каждого $p \in 1..n_2$, можно в $\boldsymbol{uv}$ выделить каждую правую максимальную серию $\boldsymbol{r}^*$, применяя условия (12.14) и (12.15) вместе с условием $|\boldsymbol{r}^*| \geqslant 2p$. Эти три условия эквивалентны неравенствам

$$\mathrm{LS}[p] > 0 \text{ и } \mathrm{LS}[p] + \mathrm{LP}[p+1] \geqslant p.$$

На этой основе строится следующий алгоритм.

Алгоритм 12.2.2 (Процедура calcR)

▷ *Вычисление в* $\boldsymbol{uv} = \boldsymbol{u}[\ell..n_1]\boldsymbol{v}[\ell..n_2]$ *всех правых максимальных серий*
вычисление $\mathrm{LP}[2..n_2+1]$, $\mathrm{LS}[1..n_2]$ для $\boldsymbol{uv}$
for $p \leftarrow 1$ **to** n_2 **do**
  if $\{\mathrm{LS}[p] > 0$ **and** $\mathrm{LS}[p] + \mathrm{LP}[p+1] \geqslant p\}$ **then**
    $i \leftarrow n_1 - \mathrm{LS}[p] + 1$ ▷ *начало серии в* $\boldsymbol{u}$
    $j \leftarrow p + \mathrm{LP}[p+1]$ ▷ *конец серии в* $\boldsymbol{v}$
    output i, j

На выполнение цикла **for** алгоритма процедуры *calcR* требуется время $\Theta(n_2)$, а для вычисления массивов LP и LS, как показано в упражнении 12.2.2, необходимо время $\Theta(n_1+n_2)$. Таким образом, время выполнения процедуры *calcR*$(\boldsymbol{u}, \boldsymbol{v})$ линейно относительно $|\boldsymbol{uv}|$ и требует памяти, объем которой линеен относительно $|\boldsymbol{v}|$. Аналогично время выполнения процедуры *calcL*$(\boldsymbol{u}, \boldsymbol{v})$ линейно относительно $|\boldsymbol{uv}|$ и требует памяти, объем которой линеен относительно $|\boldsymbol{u}|$. Следовательно, процедура *calcruns*$(\boldsymbol{u}, \boldsymbol{v})$ выполняется за время порядка $\Theta(|\boldsymbol{u}| + |\boldsymbol{v}|)$ с использованием памяти такого же объема, при этом теорема 12.2.3 сохраняет силу, а алгоритм Мейна определяет все серии класса II в строке $\boldsymbol{x}[\ell..n]$ за время $\Theta(n)$ (при условии, что s-факторизация вычислена заранее).

Необходимо отметить, что алгоритм Мейна может вывести одну и ту же серию класса II более одного раза, если серия имеет более одного периода. Рассмотрим, например, строку $\boldsymbol{x} = (ab)^3(aba)^4a$ с s-факторизацией

$$\boldsymbol{w}_1\boldsymbol{w}_2\boldsymbol{w}_3\boldsymbol{w}_4 = a/b/ababab/abaabaabaa.$$

Правая максимальная серия $(aba)^4a$, соответствующая фактору $\boldsymbol{w}_4$, будет выведена дважды: один раз для $p = 3$ (образующая aba) и второй раз для $p = 6$ (образующая $(aba)^2$). Из этого видно, что период, использованный для выделения серии, не всегда должен быть минимальным, следовательно, образующая серии может иногда сама быть кратной строкой. Но даже с этим потенциальным дублированием серий, общее количество выводов должно все еще быть меньше чем $4n$.

Отметим также, что в соответствии с необходимым условием алгоритма процедуры *calcR*

$$\mathrm{LS}[p] + \mathrm{LP}[p+1] \geqslant 1$$

в дополнение ко всем сериям мы выводим все минимальной длины непримитивные подстроки с периодом p, удовлетворяющие условиям $|\boldsymbol{r}_L^*| \geqslant 1$, $|\boldsymbol{r}_R^*| \geqslant p$. Однако количество выводов, следовательно, и общее количество таких подстрок должно быть меньше $4n$.

Как отмечалось в [168], "дальнейшая модификация может позволить алгоритму находить все [серии класса I] простым способом". В следующем подразделе мы увидим, что это действительно может быть сделано также за время $\Theta(n)$.

12.2.2 Все серии — алгоритм Колпакова и Кучерова

В этом подразделе кратко описывается модификация [139] алгоритма 12.2.1, которая вычисляет серии класса I в произвольной строке $\boldsymbol{x}[\ell..n]$ за время $\Theta(n)$. Таким образом, первоначальный алгоритм Мейна и эта модификация позволяют вычислить все серии в строке $\boldsymbol{x}$ за линейное время, предполагая, что заранее вычислена s-факторизация строки $\boldsymbol{x}$ и, следовательно, дерево суффиксов $T_{\boldsymbol{x}}$ строки $\boldsymbol{x}$ (раздел 6.3).

Алгоритм Колпакова–Кучерова (сокращенно, алгоритм KK) представляет каждую серию класса II $\boldsymbol{r} = \boldsymbol{x}[i, j]$, найденную с помощью алгоритма Мейна, как пару чисел (i, j), где i — начальная позиция, а j — конечная позиция серии в строке $\boldsymbol{x}$. (В случае, если алгоритм Мейна нашел больше одной серии (i, j), дубликаты удаляются, как мы это вскоре увидим.) Формируется массив OCCURS$[\ell..n]$, в котором OCCURS$[i]$ — это указатель на связный список позиций $j_{i1}, j_{i2}, \ldots, j_{ir_i}$, которые первоначально являются конечными позициями серий класса II, начинающихся с i-й позиции строки $\boldsymbol{x}$. Будем придерживаться соглашения, что

$$j_{i1} < j_{i2} < \cdots < j_{ir_i}. \tag{12.16}$$

Если ни одна из серий не начинается в i-й позиции, то значение OCCURS$[i]$ полагается равным `NULL`.

Чтобы вычислить массив OCCURS и его связанные списки, сначала необходимо выполнить блочную сортировку пар (i, j), в результате которой они будут размещаться в порядке возрастания значений j. Затем массив OCCURS модифи-

цируется путем обеспечения доступа к отсортированным парам в порядке их возрастания и дополнения каждого значения i соответствующим значением j в конце списка, на который указывает OCCURS$[i]$, — это является гарантией того, что условие (12.16) будет выполняться. Поскольку количество серий класса II равно $O(n)$ и поскольку i и j являются целыми числами из интервала $1..n$, то блочную сортировку и формирование OCCURS можно выполнить за время $\Theta(n)$. Разумеется, дубликаты пар (i, j) можно легко устранить в процессе сортировки.

Для того чтобы сделать эффективным использование этой структуры данных, для каждого фактора $\boldsymbol{w}_h$ ($1 \leqslant h \leqslant k$) необходимо сохранять две величины: его начальную позицию i_h и начальную позицию $i'_h < i_h$ предыдущей копии $\boldsymbol{w}_h$ в строке $\boldsymbol{x}$. В случае, когда $\boldsymbol{w}_h$ является крайним левым вхождением некоторого символа, тогда не существует предыдущей копии $\boldsymbol{w}_h$, и поэтому полагаем $i'_h \leftarrow 0$. Итак, создается два массива $I = I[\ell..k+1]$ (с $I[k+1] = n+1$) и $I' = I'[\ell..k]$ для сохранения этой промежуточной информации, определяемой по первоначально вычисленной s-факторизации, значения массива I' легко вычисляются из дерева суффиксов.

Используя эти структуры данных, можно определить, что копия $\boldsymbol{w}_h$ (если она существует) расположена в позиции $I'[h] < I[h]$. Затем можно переместить все серии подходящей длины из копии непосредственно в фактор $\boldsymbol{w}_h$. Поскольку для каждой начальной позиции i доступ к сериям из массива OCCURS$[i]$ осуществляется в порядке возрастания их длины, перемещение, соответствующее каждой i, можно реализовать за время, пропорциональное количеству перемещаемых серий. Так как, согласно теореме 12.2.1, общее количество серий равно $O(n)$, поэтому дополнительное время, необходимое для сохранения серий класса I в структуре данных, также равно $O(n)$. Детали этого процесса представлены в алгоритме 12.2.3. Основной результат данного раздела сформулирован в теореме 12.2.5.

Алгоритм 12.2.3 (Алгоритм KK)

▷ *Заданы массивы $I[\ell..k+1]$ и $I'[\ell..k]$, которые описывают*
▷ *s-факторизацию $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_k$ строки $\boldsymbol{x}[1..n]$, и заданы списки массива*
▷ *OCCURS$[1..n]$, которые определяют (в порядке возрастания*
▷ *их длины) все серии класса II, встречающиеся в каждой позиции i.*
▷ *Алгоритм модифицирует списки для включения всех серий строки $\boldsymbol{x}$*
for $h \leftarrow 2$ **to** k **do**
  if $I'[h] > 0$ **then**
    $\delta \leftarrow I[h] - I'[h]$ ▷ *смещение фактора $\boldsymbol{w}_h$ относительно его копии*
    for $i \leftarrow I[h]$ **to** $I[h+1] - 1$ **do**
      $\forall j \in$ список(OCCURS$[i - \delta]$)
      ▷ *Серия класса I должна начинаться и заканчиваться в $\boldsymbol{w}_h$,*
      if $(j + \delta) - i < I[h+1] - i$ **then**
        вставка списка (OCCURS$[i]$) $\leftarrow j + \delta$

Теорема 12.2.5. Если s-факторизация $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_k$ строки $\boldsymbol{x}[\ell..n]$ вычислена заранее, то массив OCCURS$[\ell..n]$ и связанные с ним списки вместе с массивами $I[\ell..k+\ell]$ и $I'[\ell..k]$ можно вычислить за время порядка $\Theta(n)$. На основе этих структур данных алгоритм 12.2.3, модифицирующий массив OCCURS и связанных с ним списков с целью включения всех серий строки $\boldsymbol{x}$, выполняется за время порядка $O(n)$. ■

Имеется несколько других алгоритмов, которые вычисляют наборы кратных строк в $\boldsymbol{x}[\ell..n]$ за время $\Theta(n)$ на алфавите размером α. Во всех из них используются деревья суффиксов, и поэтому в оценках времени выполнения присутствует множитель $\log\alpha$. В [141] описывается алгоритм, который вычисляет самый короткий квадрат в каждой позиции в $\boldsymbol{x}$. На основе упомянутого ранее в [88] результата, что максимальное количество различных квадратов в любой строке равно $O(n)$, в работе [107] предлагается алгоритм вычисления всех различных квадратов в строке $\boldsymbol{x}$. В [108] описан алгоритм, который вычисляет все отличающиеся друг от друга непродолжаемые (определение 2.3.2) повторяющиеся подстроки в $\boldsymbol{x}$: эти подстроки — различные образующие не только кратных строк, но и раппортов, которые могут быть расщеплением (непоследовательным) или покрытием (определение 2.3.1). Вычисление всех пар непродолжаемых повторяющихся подстрок, которые удовлетворяют различным ограничениям, описано в [41].

В заключение скажем, что в упражнении 12.2.10 мы предлагаем другой подход к вычислению серий класса I — подход, сложность которого пока не совсем ясна!

Упражнения 12.2

1. Докажите неравенства (12.12).

2. Определение массива LP и алгоритм его вычисления даны в подразделе 12.1.2. Массив LS немного отличается от массива LP. Запишите алгоритм вычисления массива LS для произвольных строк $\boldsymbol{u}$ и $\boldsymbol{v}$ и покажите, что для этого требуется время порядка $\Theta(|u|+|v|)$.

3. Запишите алгоритм вычисления массива LP$'$ для произвольных строк $\boldsymbol{u}$ и $\boldsymbol{v}$, выполняемый за время $\Theta(|u|+|v|)$.

4. Сформулируйте и докажите аналог леммы 12.2.4 для левых максимальных серий. На этой основе опишите алгоритм процедуры *calcL*.

5. Предположим, что подстрока $\boldsymbol{x}[i..j]$ — это серия с периодом p в строке $\boldsymbol{x}$. Докажите, что для любого значения $i'\in i+p..j-p+1$ ни одна подстрока в строке $\boldsymbol{x}$, начинающаяся в позиции i', не может быть серией с периодом p. Покажите на примере, что границы для таких значений i' являются неулучшаемыми.

6. Предположим, что подстрока $\boldsymbol{x}[i..j]$ — это серия с периодами p и p' ($p' \neq p$) в строке $\boldsymbol{x}$. Докажите, что наибольший общий делитель чисел p и p' также является периодом подстроки $\boldsymbol{x}[i..j]$.

Совет. Не забудьте о лемме периодичности!

7. Объясните, каким образом можно вычислить массив I' за линейное время на основе дерева суффиксов $T_{\boldsymbol{x}}$ во время выполнения s-факторизации строки $\boldsymbol{x}$.

8. Пусть h — наименьшее целое число, при котором фактор $\boldsymbol{w}_h$ в s-факторизации строки $\boldsymbol{x}$ содержит конечную позицию серии. Является ли необходимым, чтобы каждая серия в факторе $\boldsymbol{w}_h$ была серией класса II?

9. Предположим, что фактор $\boldsymbol{w}_h$ в s-факторизации строки $\boldsymbol{x}$ начинается с позиции i и что он является копией подстроки $\boldsymbol{x}$, начинающейся с позиции $i' < i$. Далее предположим, что $i - i' \leqslant |\boldsymbol{w}_h|/2$. Покажите, что подстрока $[i'..i + |\boldsymbol{w}_h| - 1]$ является серией строки $\boldsymbol{x}$.

10. Заметим, что многие серии класса I, которые располагаются в пределах какого-либо фактора $\boldsymbol{w}_h$ в s-факторизации строки $\boldsymbol{x}$, сами являются сериями класса II в s-факторизации фактора $\boldsymbol{w}_h$. Используйте этот факт при разработке алгоритма, который рекурсивно обращается к алгоритму Мейна и вычисляет все серии в строке $\boldsymbol{x}$. Что можно сказать о сложности этого алгоритма?

11. Пусть $\boldsymbol{z}_0 = (\lambda_0)^2$ и $\boldsymbol{z}_i = \boldsymbol{z}_{i-1}\lambda_i\boldsymbol{z}_{i-1}$ для всех $i = 1, 2, \ldots, n$, где символы $\lambda_1, \lambda_2, \ldots, \lambda_{n-1}$ попарно различаются. Как предполагает Роман Колпаков, строка $\boldsymbol{z}_n$ представляет худший случай для рекурсивного алгоритма, описанного в предыдущем упражнении. На основе s-факторизации $\boldsymbol{z}_n$ определите временну́ю сложность применяемого к $\boldsymbol{z}_n$ алгоритма.

ГЛАВА 13

Обобщение периодичности

Для мудреца достаточно одного слова,
а множество слов не заполнят сосуд.

— Бенджамин Франклин (1706–1790).
Альманах Бедного Ричарда

В этой главе мы обсуждаем темы, являющиеся прямыми обобщениями тех тем, которые возникают из понятия периодичности. Если говорить кратко, то основными проблемами, связанными с периодичностью, являются следующие две:

- вычисление всех периодов произвольной строковой последовательности $\boldsymbol{x} = \boldsymbol{x}[1..n]$;
- вычисление всех кратных строк в строке $\boldsymbol{x}$.

Первая из этих проблем решается с помощью алгоритма вычисления массива граней (раздел 1.3). Это первый алгоритм, который изучается в этой книге: он вычисляет за время порядка $\Theta(n)$ массив граней $\boldsymbol{\beta}[1..n]$ строки $\boldsymbol{x}$, который определяет каждую грань, следовательно, и каждый период не только строки $\boldsymbol{x}$, но и каждого префикса $\boldsymbol{x}$. Массив граней, как обсуждалось в разделе 5.1, соответствует структуре дерева, называемого деревом граней. Корневой узел дерева помечен нулем, а другие его узлы помечены позициями $i \in 1..n$ букв строки $\boldsymbol{x}$. Метки узлов на пути от корня дерева к любому узлу i (за исключением самого узла i) являются точными длинами (в возрастающем порядке) граней подстрок $\boldsymbol{x}[1..i]$.

В разделе 13.1 приведен довольно примечательный факт, что квазипериоды строки $\boldsymbol{x}$ (см. определения раздела 2.3) можно вычислить с использованием структур, формально идентичных тем, которые применяются для вычисления периодов:

- за время порядка $\Theta(n)$ можно вычислить ***массив оболочек*** $\boldsymbol{\gamma}[1..n]$, который определяет каждую оболочку (т.е. каждый квазипериод) каждого префикса строки $\boldsymbol{x}$;
- массив оболочек определяет ***дерево оболочек*** с узлами, помеченными числами из интервала $0..n$ (как и в случае дерева граней, метки, отличные от нуля, на пути от корня до любого узла i дерева оболочек — это точные длины (в возрастающем порядке) оболочек $\boldsymbol{x}[1..i]$).

Как и о массиве граней, о массиве оболочек, наверное, следует думать как о внутреннем паттерне: каждая строка имеет массив оболочек. Однако из-за его тесной связи с периодичностью здесь мы рассмотрим его как характеристический паттерн.

В разделах 13.2 и 13.3 вместо рассмотрения естественных расширений понятия кратной строки обратимся к вычислению точных и приближенных раппортов (раздел 2.3). В разделе 13.2 показано, что, как и кратные строки, непродолжаемые раппорты вычисляемы за время порядка $O(n \log n)$ и, подобно сериям, требуют памяти порядка $O(n)$. Попутно обнаружим, что алгоритм Крочемора для вычисления всех кратных строк (подраздел 12.1.1) является, в конечном счете, алгоритмом построения дерева суффиксов, что дерево суффиксов компактно представимо в виде массива пар целых чисел и что вычисление непродолжаемых раппортов является естественным началом для вычисления всех непродолжаемых покрываемых подстрок (задача 2.17 — раздел 2.3).

В разделе 13.3 рассмотрим дальнейшее обобщение проблемы раппортов до задачи приближенных раппортов, которые используем для вычисления расстояния и для приближенного сравнения с паттерном (вместо понятий и методов, представленных в главах 9 и 10). Вычисление приближенных раппортов имеет особенно важное значение для молекулярной биологии, например для идентификации длинных последовательностей ДНК, которые повторяются в другом месте генома, но с незначительным изменением.

Наконец, в разделе 13.4 рассматриваются некоторые проблемы, которые возникают при определении приближенных образующих строковой последовательности.

13.1 Все оболочки — алгоритм Ли–Смита

Вспомним из раздела 2.3, что раппорт

$$M_{\boldsymbol{x},\boldsymbol{u}} = (p; i_1, i_2, \ldots, i_r) \tag{13.1}$$

является ***оболочкой*** подстроки $\boldsymbol{x}[i_1..i_r + p - 1]$ тогда и только тогда, когда $r \geqslant 2$, $p = |\boldsymbol{u}|$ и

$$\boldsymbol{u} = \boldsymbol{x}[i_1..i_1 + p - 1] = \boldsymbol{x}[i_2..i_2 + p - 1] = \cdots = \boldsymbol{x}[i_r..i_r + p - 1],$$

а каждая ***лакуна*** (gap) $g_j = i_{j+1} - i_j$ $(j = 1, 2, \ldots, r - 1)$ лежит в интервале $1..p$. В этом разделе рассмотрим такую проблему: существует ли для произвольной строки $\boldsymbol{x} = \boldsymbol{x}[1..n]$ такая подстрока $\boldsymbol{u}$, что раппорт (13.1) является оболочкой для случая, когда $i_1 = 1$ и $i_r + p - 1 = n$. Если нет необходимости в определении местонахождения r, в котором появляется подстрока $\boldsymbol{u}$, то можно сказать, что подстрока $\boldsymbol{u}$ является оболочкой строки $\boldsymbol{x}$ или что $\boldsymbol{u}$ ***покрывает*** строку $\boldsymbol{x}$. Например, для строки $\boldsymbol{x} = abaababaaba$ подстроки $\boldsymbol{u} = abaaba$ и $\boldsymbol{u}' = aba$ являются оболочками строки $\boldsymbol{x}$. Заметим, что поскольку в (13.1) значение r должно быть не меньше 2, то сама строка $\boldsymbol{x}$ никогда не будет своей оболочкой.

Перечислим некоторые элементарные факты о строке $\boldsymbol{x}$ и ее оболочке $\boldsymbol{u}$, оставляя их доказательство для упражнения 13.1.1.

Fl. $0 < |\boldsymbol{u}| < |\boldsymbol{x}|$.

F2. Подстрока $\boldsymbol{u}$ является гранью строки $\boldsymbol{x}$.

F3. Пусть $\boldsymbol{u}'$ — такая строка, что $|\boldsymbol{u}'| < |\boldsymbol{u}|$. Тогда $\boldsymbol{u}'$ будет оболочкой $\boldsymbol{u}$ только в том случае, если $\boldsymbol{u}'$ будет оболочкой $\boldsymbol{x}$.

F4. Подстрока $\boldsymbol{u}$ имеет грань длиной $p - g_j$, где g_j $(j = 1, 2, \ldots, r - 1)$ — произвольная лакуна, определенная в (13.1).

В 1990-х годах было опубликовано несколько алгоритмов вычисления оболочек строки $\boldsymbol{x}$ со временем выполнения порядка $\Theta(n)$: в [19] вычислялась только самая короткая оболочка $\boldsymbol{x}$, затем в [40] — самая короткая оболочка каждого префикса, позже в [182, 181] — каждая оболочка строки $\boldsymbol{x}$, но без оболочек ее собственных префиксов, наконец, в [160] вычислялась каждая оболочка каждого префикса $\boldsymbol{x}$. Здесь опишем последний из этих алгоритмов, алгоритм Ли–Смита (Li–Smith, сокращенно — алгоритм ЛС), выполнение которого стало возможным благодаря введению понятия ***массива оболочек*** $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ строки $\boldsymbol{x}$, который определяется следующим образом: для каждого $i \in 1..n$ элемент массива $\boldsymbol{\gamma}[i]$ равен длине наибольшей оболочки подстроки $\boldsymbol{x}[1..i]$; $\boldsymbol{\gamma}[i] = 0$, если такой оболочки не существует. Приведем для сравнения пример массивов граней и оболочек.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$\boldsymbol{x} =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\boldsymbol{\beta} =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\boldsymbol{\gamma} =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

Непосредственным следствием фактов F1 и F3 является то, что длины оболочек подстроки $\boldsymbol{x}[1..i]$ являются элементами убывающей последовательности

$$\langle \gamma[i], \gamma^2[i], \ldots, \boldsymbol{\gamma}^{k-1}[i] \rangle, \tag{13.2}$$

где для каждого $h \in 2..k$ $\boldsymbol{\gamma}^h[i] = \boldsymbol{\gamma}[\boldsymbol{\gamma}^{h-1}[i]]$, k — наименьшее положительное целое число, такое, что $\boldsymbol{\gamma}^k[i] = 0$. Это аналог леммы 1.3.1, которая говорит, что $\boldsymbol{u} = \boldsymbol{x}[1..p]$ является гранью подстроки $\boldsymbol{x}[1..i]$ тогда и только тогда, когда $p = \boldsymbol{\beta}^h[i]$ для некоторого целого числа $h \in 1..k$, где k — наименьшее положительное целое число, такое, что $\boldsymbol{\beta}^k[i] = 0$. Эти результаты являются основанием для того, чтобы представить все грани (соответственно, оболочки) строки $\boldsymbol{x}$, используя только n ячеек памяти. Как показано на рис. 13.1, отношение между элементами i и $\boldsymbol{\beta}[i]$ (соответственно, и $\boldsymbol{\gamma}[i]$) — это отношение наследования (типа "сын–родитель") в дереве $T_{\boldsymbol{\beta}}$ (соответственно, в дереве $T_{\boldsymbol{\gamma}}$), которое обеспечивает доступ к элементам множества граней (соответственно, и оболочек) подстроки $\boldsymbol{x}[1..i]$ через простое обращение к предкам i в дереве $T_{\boldsymbol{\beta}}$.

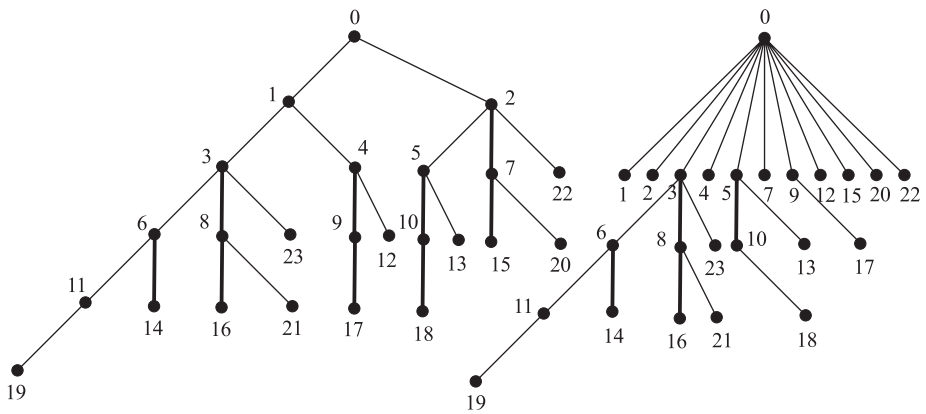


Рис. 13.1. Дерево граней и дерево оболочек строки $\boldsymbol{x} = abaababaabaababaababa$

Возможно, изложенное выше убедило читателя в том, что массив оболочек в какой-то степени является комбинаторным объектом. Для большей убедительности в следующем подразделе мы докажем две теоремы, которые накладывают серьезные ограничения на значения, допустимые в массиве оболочек, и фактически связывают их с ограничениями массива граней. Довольно удивительно, по крайней мере для автора, что доказательство одной из этих теорем основывается на не очень трудных упражнений о расстоянии Хемминга и о циклическом сдвиге строк, которые, на первый взгляд, совсем не связаны с оболочками.

В заключительном подразделе мы продолжим описание того, как можно вычислить массив оболочек за время порядка $\Theta(n)$.

Свойства массива оболочек

Наблюдательный читатель в процессе решения упражнения 2.2.6 уже убедился в том, что для произвольной строки $\boldsymbol{x}[1..n]$ справедливо неравенство

$$d_H(\boldsymbol{x}, R_j(\boldsymbol{x})) \neq 1, \tag{13.3}$$

где d_H — расстояние Хемминга (раздел 2.2), а $R_j(\boldsymbol{x})$ — j-й циклический сдвиг строки $\boldsymbol{x}$, $j \in 0..n-1$ (раздел 1.4). Этот результат мы используем для доказательства леммы, которая, в свою очередь, будет использована для доказательства первых двух интересных свойств массивов граней.

Лемма 13.1.1. Если $\boldsymbol{u}$ является оболочкой строки $\boldsymbol{x}$ и $|\boldsymbol{u}| = p$, тогда каждая подстрока $\boldsymbol{v}$ длиной p в строке $\boldsymbol{x}$ удовлетворяет неравенству

$$d_H(\boldsymbol{u}, \boldsymbol{v}) \neq 1.$$

Доказательство. Рассмотрим два различных вхождения $\boldsymbol{u}$ в $\boldsymbol{x} = \boldsymbol{x}[1..n]$, которые являются смежными или перекрывающимися. Обозначим левое вхождение $\boldsymbol{u}$ как $\boldsymbol{u}_1 = \boldsymbol{x}[i_1..i_1 + p - 1]$, а правое вхождение — как $\boldsymbol{u}_2 = \boldsymbol{x}[i_2..i_2 + p - 1]$, где $1 \leqslant i_1 \leqslant n - p + 1$ и $i_1 < i_2 \leqslant i_1 + p$.

Покажем, что утверждение леммы справедливо для каждой подстроки $\boldsymbol{v}$ строки $\boldsymbol{U} = \boldsymbol{x}[i_1..i_2 + p - 1]$, следовательно, и для самой строки $\boldsymbol{x}$.

Сначала рассмотрим случай $i_2 = i_1 + p$. Здесь $\boldsymbol{U} = \boldsymbol{u}^2$, поэтому каждая подстрока $\boldsymbol{v}$ длиной p в $\boldsymbol{U}$ является циклическим сдвигом $\boldsymbol{u}$. Тогда утверждение леммы является прямым следствием неравенства (13.3).

Теперь предположим, что $i_2 < i_1 + p$. В этом случае $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$ перекрываются некоторой непустой гранью $\boldsymbol{u}'$ строки $\boldsymbol{u}$. Тогда, как показано на рис. 13.2, можно записать $\boldsymbol{u} = \boldsymbol{u}'\boldsymbol{u}''$ для некоторого непустого суффикса $\boldsymbol{u}''$ строки $\boldsymbol{u}$.

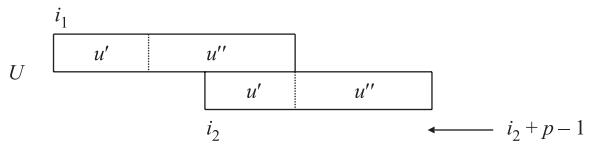


Рис. 13.2. Перекрывающиеся вхождения $\boldsymbol{u}$ в $\boldsymbol{U}$

Пусть $\boldsymbol{u}'' = \boldsymbol{w}^k$ для некоторого наибольшего целого числа $k \geqslant 1$ (если $\boldsymbol{u}''$ не является кратной строкой, то $\boldsymbol{w} = \boldsymbol{u}''$ и $k = 1$). Тогда $\boldsymbol{U}$ имеет период $|\boldsymbol{w}|$, и поэтому каждая подстрока $\boldsymbol{v}''$ длиной $|\boldsymbol{u}''|$ в $\boldsymbol{U}$ также имеет период $|\boldsymbol{w}|$. Таким образом, подстрока $\boldsymbol{v}''$ должна удовлетворять одному из следующих двух условий.

- $\boldsymbol{v}''$ является циклическим сдвигом $\boldsymbol{u}''$, но не равным $\boldsymbol{u}''$, и поэтому $d_H(\boldsymbol{u}'', \boldsymbol{v}'') \geqslant 2$ в соответствии с (13.3). Следовательно, для каждого вхождения $\boldsymbol{v} = \boldsymbol{v}'\boldsymbol{v}''$ длиной p в $\boldsymbol{U}$ выполняется неравенство $d_H(\boldsymbol{u}, \boldsymbol{v}) \geqslant 2$.

- $\boldsymbol{v}'' = \boldsymbol{u}''$. В этом случае, поскольку $\boldsymbol{v}''$ является подстрокой $\boldsymbol{U}$ с периодом $|\boldsymbol{w}|$ и поскольку строка $\boldsymbol{u} = \boldsymbol{u}'\boldsymbol{u}'' = \boldsymbol{u}'\boldsymbol{v}''$ входит в $\boldsymbol{U}$, по меньшей мере, один раз, то каждому вхождению $\boldsymbol{v}''$ в $\boldsymbol{U}$ предшествует вхождение $\boldsymbol{u}'$, такое, что $d_H(\boldsymbol{u}, \boldsymbol{u}'\boldsymbol{v}'') = 0$.

Таким образом, показано, что в $\boldsymbol{U}$ не существует ни одной подстроки $\boldsymbol{v}$, такой, что $d_H(\boldsymbol{u}, \boldsymbol{v}) = 1$, что и требовалось доказать. ■

Этот результат впервые был доказан (очень изящно) в статье [210] путем использования варианта леммы о периодичности [32]; более слабая версия была сформулирована (с тяжеловесным определением) в [160]. С их помощью можно доказать следующее фундаментальное свойство массива оболочек $\boldsymbol{\gamma}[1..n]$.

Теорема 13.1.2. Если $\boldsymbol{\gamma}[i] \neq 0$, тогда или $\boldsymbol{\gamma}[i+1] > \boldsymbol{\gamma}[i]$, или $\boldsymbol{\gamma}[i+1] = 0$ ($\forall i \in$ $\in 1..n-1$).

Доказательство. Пусть $p = \boldsymbol{\gamma}[i]$. Тогда $p > 0$ и $\boldsymbol{x}[1..p]$ является наибольшей оболочкой подстроки $\boldsymbol{x}[1..i]$. Предположим, что теорема неверна, и поэтому существует положительное целое число $p' \leqslant p$, такое, что $\boldsymbol{x}[1..p']$ будет наибольшей оболочкой подстроки $\boldsymbol{x}[1..i+1] = \boldsymbol{x}[1..i]\lambda$. Отсюда следует, что $\boldsymbol{x}[1..p']$ является оболочкой подстроки $\boldsymbol{x}[1..p]\lambda$ (рис. 13.3).

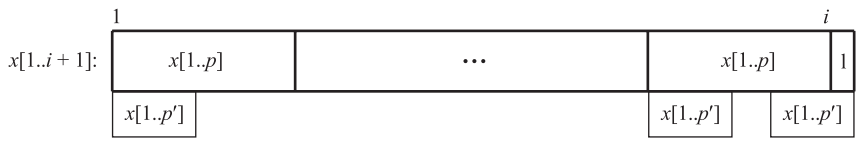


Рис. 13.3. Подстрока $\boldsymbol{x}[1..p']$ является оболочкой строки $\boldsymbol{x}[1..p]\lambda$

Если за каждым вхождением строки $\boldsymbol{x}[1..p]$ в $\boldsymbol{x}[1..i+1]$ будет следовать буква λ, тогда $\boldsymbol{x}[1..p+1]$ будет оболочкой $\boldsymbol{x}[1..i+1]$, что входит в противоречие с предположением о том, что $\boldsymbol{x}[1..p']$ является наибольшей оболочкой. Таким образом, за некоторым вхождением $\boldsymbol{x}[1..p]$ в $\boldsymbol{x}[1..i]$ должна следовать буква $\lambda \neq \lambda'$. Это говорит о том, что

- имеется, по меньшей мере, две различные буквы в строковой последовательности $\boldsymbol{x}[1..i+1]$, следовательно, и в $\boldsymbol{x}[1..p']$, так что $p' \geqslant 2$;
- поскольку $\boldsymbol{x}[1..p'] = \boldsymbol{x}[1..p'-1]\lambda$ является оболочкой, тогда в строке $\boldsymbol{x}[1..i+$ $+1]$ существует подстрока $\boldsymbol{u} = \boldsymbol{x}[1..p'-1]$, такая, что $d_H(\boldsymbol{x}[1..p'], \boldsymbol{u}) = 1$. Это противоречит лемме 13.1.1.

Таким образом, предположение $0 < p' \leqslant p$ является неверным, что доказывает теорему. ■

Следующий результат связывает значения массива оболочек $\boldsymbol{\gamma}[1..n]$ со значениями массива граней $\boldsymbol{\beta}[1..n]$: значения $\boldsymbol{\gamma}[i+1]$ могут быть ненулевыми только тогда, когда $\boldsymbol{\beta}[i+1] = \boldsymbol{\beta}[i] + 1$.

Теорема 13.1.3. Если $\boldsymbol{\beta}[i+1] \leqslant \boldsymbol{\beta}[i]$, тогда $\boldsymbol{\gamma}[i+1] = 0$ $(\forall i \in 1..n-1)$.

Доказательство. Из факта F2 следует, что если $\boldsymbol{\beta}[i+1] = 0$, тогда $\boldsymbol{\gamma}[i+1] = 0$. Следовательно, без потери общности можем предположить, что $\boldsymbol{\beta}[i+1] > 0$. Поэтому, в соответствии с гипотезой, $\boldsymbol{\beta}[i] > 0$.

Теперь предположим, что теорема неверна, поэтому $\boldsymbol{\gamma}[i+1] = p' > 0$. Тогда $\boldsymbol{x}[1..p']$ является оболочкой подстроки $\boldsymbol{x}[1..i+1]$. Таким образом, положив $p = \boldsymbol{\beta}[i]$, имеем

$$0 < p' = \boldsymbol{\gamma}[i+1] \leqslant \boldsymbol{\beta}[i+1] \leqslant p.$$

Обозначив $\boldsymbol{x}[i+1] = \lambda$, видим, что оболочка $\boldsymbol{x}[1..p'] = \boldsymbol{x}[i-p'+2..i]\lambda$ является собственным суффиксом строковой последовательности

$$\boldsymbol{x}[i-p+1..i+1] = \boldsymbol{x}[i-p+1..i]\lambda = \boldsymbol{x}[1..p] = \boldsymbol{x}[1..p+1]. \tag{13.4}$$

Но поскольку $\boldsymbol{\beta}[i+1] \neq \boldsymbol{\beta}[i] + 1$, то невозможно равенство $\boldsymbol{x}[p+1] = \lambda$, что противоречит (13.4). Поэтому утверждение $\boldsymbol{\gamma}[i+1] > 0$ не имеет места. ■

Из этой теоремы естественным образом вытекает понятие ***ступени*** (staircase) — максимальной последовательности $S_{i,h}$, состоящей из h номеров $i, i+1, \ldots, i+h-1$ элементов массива граней $\boldsymbol{\beta}[1..n]$, таких, что $\boldsymbol{\beta}[j+1] = \boldsymbol{\beta}[j]+1$ для каждого $j \in i..i+h-2$. Отметим, что "максимальность" ступени означает, что $\boldsymbol{\beta}[i] \leqslant \boldsymbol{\beta}[i-1]$ для $i-1 \geqslant 1$ и $\boldsymbol{\beta}[i+h-1] \geqslant \boldsymbol{\beta}[i+h]$ для $i+h \leqslant n$. Например, в массиве граней $\boldsymbol{\beta} = 0011232345645$ имеется 5 ступеней: $S_{1,1}$, $S_{2,2}$, $S_{4,3}$, $S_{7,5}$, $S_{12,2}$.

Теорема 13.1.3 говорит, что $\boldsymbol{\gamma}[i] = 0$ в начальной позиции i каждой ступени массива $\boldsymbol{\beta}$. Позднее мы увидим, насколько полезным может быть понятие ступени для вычисления массива оболочек.

Еще одно полезное понятие — это понятие "живой" вершины. Если строка $\boldsymbol{x}$ является префиксом строки $\boldsymbol{y}$, то будем говорим, что $\boldsymbol{y}$ является ***правым расширением*** строки $\boldsymbol{x}$. Если префикс $\boldsymbol{u}$ строки $\boldsymbol{x}$ может быть оболочкой некоторого правого расширения, тогда об $\boldsymbol{u}$ говорят, что он ***живой*** (live) по отношению к $\boldsymbol{x}$; иначе об $\boldsymbol{u}$ говорят, что он ***мертвый*** (dead) по отношению к $\boldsymbol{x}$. Например, если $\boldsymbol{x} = abaab$, то $\boldsymbol{u} = aba$ живой по отношению к $\boldsymbol{x}$, поскольку aba покрывает правое расширение $\boldsymbol{x}a$; с другой стороны, $\boldsymbol{u} = a$ и $\boldsymbol{u} = ab$ не могут покрывать правое расширение $\boldsymbol{x}$, и поэтому они по отношению к $\boldsymbol{x}$ мертвые. Заметим, что $\boldsymbol{x}$ всегда жива по отношению к самой себе.

Поскольку префикс определяется своей длиной и поскольку мы всегда говорим об одной данной строке $\boldsymbol{x}$, расширим понятия "мертвый–живой" до отдельных позиций в строке $\boldsymbol{x}$: будем говорить, что позиция $j \in 1..i$ жива по отношению

к позиции i тогда и только тогда, когда $\boldsymbol{x}[1..j]$ может быть оболочкой некоторого правого расширения строки $\boldsymbol{x}[1..i]$. Итак, поскольку позиции в $\boldsymbol{x}$ соответствуют узлам дерева оболочек, мы далее расширим эту терминологию до живых и мертвых узлов в дереве T_γ. Заметим, что если позиция j мертва по отношению к i, она также мертва и по отношению к $i+1$. Таким образом, при просмотре строки $\boldsymbol{x}$ слева направо, т.е. при возрастании величины i, количество мертвых позиций будет монотонно неубывающей последовательностью. В строках, как и в реальной жизни, если умер, то это навсегда.

Лемма 13.1.4. Если $j \in i - \boldsymbol{\beta}[i]$, то j жива по отношению к i.

Доказательство. В соответствии с условием леммы, подстрока $\boldsymbol{x} = \boldsymbol{x}[1..i]$ имеет минимальный период $p^* = i - \boldsymbol{\beta}[i]$ и поэтому может быть записана в нормальной форме как $\boldsymbol{x} = \boldsymbol{x}[1..p^*]^{r^*}\boldsymbol{x}[1..p']$ для целых $r^* = \lfloor i/p^* \rfloor$ и $p' = i \bmod p^*$. Для каждого $j \in p^*..i$ префикс

$$\boldsymbol{x}[1..j] = \boldsymbol{x}[1..p^*]^{\lfloor i/p^* \rfloor}\boldsymbol{x}[1..j \bmod p^*]$$

является оболочкой правого расширения

$$\boldsymbol{x}\boldsymbol{x}[p'+1..p^*]\boldsymbol{x}[1..j \bmod p^*] = \boldsymbol{x}[1..p^*]^{r+1}\boldsymbol{x}[1..j \bmod p^*].$$

Поэтому позиция j жива по отношению к i. ■

В качестве примера к этой лемме рассмотрим строку $\boldsymbol{x}[1..10] = abcaabcaab$ длиной $i = 10$ и периода $p^* = 4$. Тогда для каждого $j \in 4..10$ подстроки $\boldsymbol{x}[1..j]$ являются оболочками правого расширения

$$\boldsymbol{x}\boldsymbol{x}[3..4]\boldsymbol{x}[1..j \bmod 4] = (abca)^3\boldsymbol{x}[1..j \bmod 4].$$

Однако заметим, что достаточное условие леммы 13.1.4 не является необходимым, например $\boldsymbol{x}[1..3] = aba$ является оболочкой строки $\boldsymbol{x}[1..8] = ababaaba$ длиной $i = 8$ и периода $p^* = 5$, хотя $j = 3$ не принадлежит интервалу $5..8$. Следующая лемма дает полную характеристику живых позиций.

Лемма 13.1.5. По отношению к каждому $i \in 1..n$ позиция j жива тогда и только тогда, когда выполняется одно из следующих двух условий:

а) $j \in i - \boldsymbol{\beta}[i]..i$;

б) $\boldsymbol{x}[1..j]$ является оболочкой подстроки $\boldsymbol{x}[1..j']$ для некоторого $j' \in i - \boldsymbol{\beta}[i]..i$.

Доказательство. Чтобы доказать, что условия *а*) и *б*) необходимы, сначала предположим, что позиция j жива по отношению к позиции i. Принимая во внимание лемму 13.1.4, необходимо показать только то, что условие *б*) выполняется для $j < p^* = i - \boldsymbol{\beta}[i]$. Поэтому предположим, что условие *б*) не выполняется: $\boldsymbol{x}[1..j]$

не является оболочкой $\boldsymbol{x}[1..j']$ ни при каком $j' \in p^*..i$. Но тогда, поскольку $j < p^*$, $\boldsymbol{x}[1..j]$ не может покрывать ни одно из расширений $\boldsymbol{x}$ — приходим к противоречию. Итак, если условие *а*) не выполняется, тогда должно выполняться условие *б*) для того, чтобы позиция j могла быть живой по отношению к позиции i.

Достаточность условия *а*) уже установлена леммой 13.1.4. Чтобы доказать достаточность условия *б*) предположим, что для некоторого целого числа $j \leqslant p^* - 1$ подстрока $\boldsymbol{x}[1..j]$ является оболочкой некоторой подстроки $\boldsymbol{x}[1..j']$, $j' \in \in p^*..i$.

Сначала положим, что $j' < i - p^*$. Тогда, поскольку $\boldsymbol{x}[1..i]$ имеет период p^*, если $\boldsymbol{x}[1..j]$ покрывает $\boldsymbol{x}[1..j']$, то она также должна покрывать $\boldsymbol{x}[p^* + 1..p^* + j']$ и, следовательно, $\boldsymbol{x}[1..p^* + j']$. Фактически, для каждой позиции $j' < i - p^* \boldsymbol{x}[1..j]$ покрывает $\boldsymbol{x}[1..p^* + j']$, и поэтому без потери общности можем допустить, что $j' \geqslant i - p^*$.

Теперь можем показать, что правое расширение $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{x}[i - p^* + 1..j']$ длиной $p^* + j' \geqslant i$ имеет грань $\boldsymbol{x}[1..j']$ длиной $j' \geqslant (p^* + j')/2$ и поэтому покрывается оболочкой $\boldsymbol{x}[1..j']$. Тогда, по определению, j жива по отношению к i, что и требовалось доказать.

Оставим для упражнения 13.1.4 доказательство равенства

$$\boldsymbol{y} = \boldsymbol{x}[1..j']\boldsymbol{x}[j' + 1..i]\boldsymbol{x}[i - p^* + 1..j'] = \boldsymbol{x}[1..p^*]\boldsymbol{x}[1..j']. \tag{13.5}$$

■

Как будет видно из следующего подраздела, для вычисления массива оболочек очень важно охарактеризовать мертвые узлы по отношению к началу ступени.

Лемма 13.1.6. Пусть i — позиция, с которой начинается ступень в массиве граней $\boldsymbol{\beta}$. Тогда позиция $j < i - \boldsymbol{\beta}[i]$ будет мертвой по отношению к i только тогда, когда j не будет иметь узлов–сыновей в $T_{\boldsymbol{\gamma}}$, которые живы по отношению к i.

Доказательство. Предположим, что позиция $j < i - \beta[i]$ является мертвой по отношению к i. Тогда $\boldsymbol{x}[1..j]$ не может быть оболочкой ни одного правого расширения $\boldsymbol{x}[1..i]$. Если j имеет узла–сына j' в $T_{\boldsymbol{\gamma}}$, который жив по отношению к i, тогда $\boldsymbol{x}[1..j']$ будет потенциальной оболочкой $\boldsymbol{x}[1..k]$ для некоторого $k \geqslant i$. Но поскольку j' является сыном j, то тогда $\boldsymbol{x}[1..j]$ покрывает $\boldsymbol{x}[1..j']$ и поэтому, в соответствии с фактом F3, покрывает также $\boldsymbol{x}[1..k]$, что является противоречием. Таким образом, необходимость условий теоремы доказана.

Чтобы доказать их достаточность, предположим, что $j < i - \beta[i]$ не имеет сынов, живых по отношению к i, но сам узел j жив по отношению к узлу i. Тогда, согласно лемме 13.1.5, $\boldsymbol{x}[1..j]$ должна покрывать некоторую подстроку $\boldsymbol{x}[1..k]$, $i - \boldsymbol{\beta}[i] \leqslant k \leqslant i$. Поэтому j имеет сына k в $T_{\boldsymbol{\gamma}}$, который, снова согласно лемме 13.1.5, должен быть живым по отношению к i, что является противоречием. Мы приходим к заключению, что j мертв по отношению к i. ■

Следствия леммы: если узел j жив по отношению к узлу i, тогда жив и его узел-родитель $\boldsymbol{\gamma}[i]$ в $T_{\boldsymbol{\gamma}}$.

В заключение докажем простое свойство функции $i-\boldsymbol{\beta}[i]$, которое необходимо для вычисления массива оболочек.

Лемма 13.1.7. Функция $i - \boldsymbol{\beta}[i]$ инвариантна для каждого i в одной и той же ступени и монотонно неубывающая по i; в частности, для любой позиции $i > 1$, которая начинает новую ступень в $\boldsymbol{\beta}$, выполняется неравенство

$$(i-1) - \boldsymbol{\beta}[i-1] < i - \boldsymbol{\beta}[i].$$

Доказательство является следствием того факта, что неравенство $\boldsymbol{\beta}[i] \leqslant \boldsymbol{\beta}[i-1]$ справедливо тогда и только тогда, когда i начинает новую ступень. ■

Исходя из лемм 13.1.5 и 13.1.7, приходим к заключению, что совокупность мертвых узлов для всех значений i одной и той же ступени остается неизменной; таким образом, ее повторное вычисление (с использованием леммы 13.1.6) необходимо только тогда, когда начинается новая ступень.

Вычисление массива оболочек

Алгоритм ЛС, обрабатывая слева направо для каждого $i = 1, 2, \ldots, n$ как строку $\boldsymbol{x}$, так и массив граней $\boldsymbol{\beta}$, выполняет две основные задачи:

- вычисляет $\boldsymbol{\gamma}[i]$ и добавляет i в качестве нового сына узла $\boldsymbol{\gamma}[i]$ в дерево оболочек $T_{\boldsymbol{\gamma}}$ — вспомним, что по определению $\boldsymbol{\gamma}[i]$ является предком i в $T_{\boldsymbol{\gamma}}$;
- для каждой позиции i, которая является началом ступени в $\boldsymbol{\beta}$, вычисляет узлы в $T_{\boldsymbol{\gamma}}$, которые мертвы по отношению к i.

В дополнение к массивам $\boldsymbol{\beta}[1..n]$ и $\boldsymbol{\gamma}[1..n]$, алгоритм также использует следующие массивы.

- Массив $dead[0..n-1]$: согласно лемме 13.1.6, $dead[i] =$ `TRUE` тогда и только тогда, когда для текущего значения $ij < i - \boldsymbol{\beta}[i]$ и j не имеет сыновей в $T_{\boldsymbol{\gamma}}$, которые живы по отношению к i (как можно видеть, корневой узел 0 в $T_{\boldsymbol{\gamma}}$ всегда живой);
- Массив $livechildren[0..n]$: $livechildren[i] = k$ тогда и только тогда, когда узел i имеет в дереве оболочек $T_{\boldsymbol{\gamma}}$ ровно k живых узлов-сыновей;
- Массив $largestlive[0..n]$: $largestlive[i] = j$, где j — наибольший предок i в дереве оболочек $T_{\boldsymbol{\gamma}}$ (разумеется, узел i всегда живой по отношению к самому себе).

Этот алгоритм начинается с помещения узла 0 в дерево оболочек $T_{\boldsymbol{\gamma}}$ и установки каждого элемента в массиве $dead$ в состояние `FALSE`, каждого элемента в $livechildren$ в состояние 0, а $largestlive[i]$ в состояние i для каждого $i \in 1..n$. Затем для текущей i-й позиции в строке $\boldsymbol{x}$ ($i = 1, 2, \ldots, n$) в алгоритме ЛС выполняются следующие три шага.

- Шаг 1 обновляет массив $largestlive$.
- Шаг 2 присоединяет i к его собственному узлу-родителю в $T_{\boldsymbol{\gamma}}$.
- Шаг 3 гарантирует обновление массива $dead$ в начале каждой новой ступени.

Эти шаги более подробно описываются в алгоритме 13.1.1.

Алгоритм 13.1.1 (Алгоритм ЛС)

```
▷ Вычисление массива оболочек γ для строки x[1..n]
for i ← 1 to n do
    ▷ Шаг 1: если β[i] мертвый в T_γ, то β[i] должен иметь
    ▷ того же наибольшего предка, что и его родитель
    if dead[β[i]] = TRUE then
        largestlive[β[i]] ← largestlive[γ[β[i]]]
    ▷ Шаг 2: Вычисление γ[i]
    ▷ если β[i] живой, положить γ[i] ← β[i]; иначе, поскольку каждая
    ▷ оболочка x[1..i] должна покрывать x[1..β[i]], положить γ[i] равным
    ▷ самому большому живому предку β[i] (возможно, 0)
    γ[i] ← largestlive[β[i]]
    livechildren[γ[i]] ← livechildren[γ[i]] + 1
    ▷ Шаг 3: В T_γ определяются все узлы, которые становятся
    ▷ мертвыми в результате начала новой ступени
    if i > 1 and β[i] ⩽ β[i − 1] then
        c1 ← i − β[i]; c2 ← (i − 1) − β[i − 1]        ▷ Лемма 13.1.7
        for j ← c1 downto c2 do                       ▷ Лемма 13.1.6
            if not dead[j] and livechildren[j] = 0 then
                dead[j] ← TRUE
                j′ ← γ[j]
                livechildren[j′] ← livechildren[j′] − 1
                ▷ Установить мертвыми все предки,
                ▷ у которых нет живых сынов
                while livechildren[j′] = 0 do
                    dead[j′] ← TRUE
                    j′ ← γ[j′]
                    livechildren[j′] ← livechildren[j′] − 1
```

Справедлива следующая теорема.

Теорема 13.1.8. Алгоритм 13.1.1 корректно вычисляет массив оболочек $\boldsymbol{\gamma}[1..n]$ произвольной строки $\boldsymbol{x}[1..n]$.

Доказательство. Рассмотрим каждый шаг алгоритма в отдельности, начиная с шага 3. Пусть $1 = i_1 < i_2 < \cdots < i_k \leqslant n$ обозначают позиции (номера)

элементов массива $\boldsymbol{\beta}$, с которых начинаются ступени. На шаге 3 для каждой позиции $j \in i_{h-1} \ldots i_h - 1$ ($h = 2, 3, \ldots, k-1$) проверяется, будет ли эта позиция мертвой по отношению к i_{h+1}. В соответствии с леммой 13.1.6, узел–родитель $\boldsymbol{\gamma}[j]$ любого элемента j, установленного мертвым, должен быть проверен: если $\boldsymbol{\gamma}[j]$ ранее был живым, то теперь он может быть мертвым тогда и только тогда, когда у него нет живых сыновей. Таким образом, на шаге 3 должна выполняться рекурсивная проверка родителя *любой* позиции, установленной мертвой, до тех пор пока не будет найден предок, имеющий, по меньшей мере, одного живого сына (поэтому этот предок останется живым). Отметим, что поскольку позиция i_{h+1} по отношению к самой себе всегда живая, то, согласно лемме 13.1.6, должен существовать путь, ведущий от i_{h+1} до корня и содержащий только живые узлы. Таким образом, корень всегда живой, а выполнение шага 3 будет всегда заканчиваться на первом живом узле по пути от j до корня.

Отметим, что позиции $j \in i_{h-1} \ldots i_h - 1$ просматриваются в обратном порядке $i_h - 1, i_h - 2, \ldots, i_{h-1}$ из-за того, что, возможно, j или также и $\boldsymbol{\gamma}[j] \in i_{h-1}..i_h - 1$ установлены мертвыми. Именно по этой самой причине, чтобы избежать избыточной обработки того же пути к корню, необходимо проверить в цикле **for** шага 3 условие **not** $dead[j]$: узел j может быть установлен мертвым из-за того, что он является родителем большего узла из интервала $i_{h-1}..i_h - 1$, который уже установлен мертвым.

Поскольку $livechildren[\boldsymbol{\gamma}[j]]$ всегда уменьшается на единицу для каждого узла j, который устанавливается мертвым, приходим к заключению, что шаг 3 корректно справляется с задачей установки мертвыми узлов, находящихся в начале каждой новой ступени.

Шаг 2 заключается в непосредственном обновлении $T_{\boldsymbol{\gamma}}$, базирующемся на текущей позиции i. Его корректность полностью зависит от корректного обновления $largestlive[\boldsymbol{\beta}[i]]$ в шаге 1.

Оператор присваивания в шаге 1 будет выполнен только в том случае, если $\boldsymbol{\beta}[i]$ был установлен мертвым во время выполнения одного из предыдущих циклов **for** в шаге 3. Как было отмечено в доказательстве шага 3, если некоторый узел j должен быть установлен мертвым относительно i_{h+1}, он должен быть живым по отношению к i_h, следовательно, согласно лемме 13.1.5, он живой относительно $i_{h+1} - 1$. Поэтому $\boldsymbol{x}[1..j]$ может быть *потенциальной* оболочкой $\boldsymbol{x}[1..i_{h+1}]$; таким образом, она может покрывать некоторое правое расширение $\boldsymbol{x}[1..\boldsymbol{\beta}[i_{h+1}]]$, которое является суффиксом подстроки $\boldsymbol{x}[1..i_{h+1}]$. Следовательно, если какая-либо позиция j устанавливается мертвой, тогда j будет превышать $\boldsymbol{\beta}[i_{h+1}]$, где i_{h+1} — начало ступени, которое используется на шаге 3.

Поскольку, согласно лемме 1.3.1, значения в массиве $\boldsymbol{\beta}$ могут увеличиваться не более чем на единицу от одной позиции к следующей, тогда каждая такая мертвая позиция j должна позже стать значением в массиве $\boldsymbol{\beta}$, т.е. для того чтобы был выполнен шаг 1, необходимо положить $j = \boldsymbol{\beta}[i']$ для некоторых $i' > i_{h+1}$.

В частности, значения j должны обрабатываться в шаге 1 в порядке возрастания их величин: т.е. в порядке их убывания в дереве оболочек $T_{\boldsymbol{\gamma}}$. Это означает, что шаг 1 передаст правильные значения наибольшего живого предка от родителя к сыну. ■

Теперь рассмотрим время, необходимое для выполнения алгоритма ЛС. Каждая операция на шаге 3, кроме цикла **for**, выполняется за константное время. В таком случае для выполнения алгоритма ЛС необходимо время порядка $\Theta(n)$ плюс общее время, использованное циклом **for**. Для оценки этого общего времени заметим, что время, необходимое для *каждого однократного* выполнения цикла **for**, пропорционально выражению

$$\max\{(c_1 - c_2), \text{количество узлов, установленных мертвыми}\}.$$

Поскольку сумма разностей $c_1 - c_2$ по всем ступеням не превышает $n-1$ и каждый из n узлов хотя бы один раз может быть установлен мертвым, то из этого следует, что общее время для выполнения всех циклов **for** будет иметь порядок $O(2n)$. Следовательно, справедлива теорема.

Теорема 13.1.9. Алгоритм 13.1.1 требует для своего выполнения время порядка $\Theta(n)$ и объем памяти порядка $\Theta(n)$. ■

Алгоритм ЛС — это оптимальный ***онлайновый*** алгоритм в слабом смысле (см. определение в разделе 4.1): он вычисляет $\boldsymbol{\gamma}$ таким образом, что делает доступными все оболочки каждого префикса строки $\boldsymbol{x}$, затрачивая на это асимптотически наименьшие возможное время и объем памяти. Отметим, что поскольку вычисление $\boldsymbol{\beta}$ — также онлайновый процесс, то алгоритм ЛС можно легко модифицировать для одновременного онлайнового вычисления обоих массивов: массива граней и массива оболочек.

Читателю было бы полезно посмотреть, каким образом алгоритм можно применить к примеру, данному на рис. 13.1. В частности, отметим, что в результате того факта, что $\boldsymbol{\beta}[22] = 2$, установлены мертвыми следующие узлы: 5, 6, 9, 10, 11, 13–19.

Упражнения 13.1

1. Докажите факты F1–F4.
2. На основе фактов F1–F3 покажите, что $\boldsymbol{u} = \boldsymbol{x}[1..p]$ является оболочкой $\boldsymbol{x}[1..i]$ тогда и только тогда, когда для некоторого целого $h \in 1..k-1$ $p = \boldsymbol{\gamma}^h[i]$, где k — наименьшее положительное целое, такое, что $\boldsymbol{\gamma}^k[i] = 0$.
3. Строго говоря, массив оболочек наиболее близко соответствует не массиву граней, а, скорее, ***массиву периодов*** $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$, в котором $\boldsymbol{\pi}[i]$ равен

наибольшему периоду строки $\boldsymbol{x}[1..i]$, являющейся кратной строкой, иначе $\boldsymbol{\pi}[i]$ равен нулю. Без вычисления массива граней определите алгоритм, который вычисляет массив $\boldsymbol{\pi}$ за линейное время. Затем покажите, что этот массив периодов можно представить в виде ***дерева периодов*** со свойствами, аналогичными свойствам дерева граней и дерева оболочек.

4. Используя факты, что $\boldsymbol{x}$ имеет период p и что $j' \geqslant \max\{p, i-p\}$, докажите равенство (13.5).
5. Измените алгоритм ЛС таким образом, чтобы обеспечить онлайновое вычисление обоих массивов $\boldsymbol{\beta}$ и $\boldsymbol{\gamma}$.

13.2 Все раппорты — алгоритм Франека–Смита–Танга

В этом разделе мы рассматриваем проблему точных раппортов в ее самой общей форме: говоря языком раздела 2.3, мы вычислим все NE (непродолжаемые) полные раппорты в произвольной строковой последовательности $\boldsymbol{x}[1..n]$ за время порядка $O(n \log n)$ и представим эти раппорты в виде массива или дерева, для хранения которых требуется память порядка $\Theta(n)$. Сначала опишем алгоритм Франека–Смита–Танга (Franek–Smyth–Tang [93], сокращенно — алгоритм ФСТ), использующий деревья суффиксов (раздел 5.2), затем покажем, каким образом он может быть реализован с помощью массивов суффиксов (подраздел 5.3.2) при значительной экономии пространства памяти. Алгоритм вычисления всех раппортов основан на простом факте.

Лемма 13.2.1. Обозначим через $\widehat{\boldsymbol{x}}$ строку $\boldsymbol{x}[n]\boldsymbol{x}[n-1]\ldots\boldsymbol{x}[1]$, обратную к заданной строке $\boldsymbol{x}$. Тогда раппорт $M_{\boldsymbol{x},\boldsymbol{u}}$ будет раппортом типа LE (продолжаемым влево) тогда и только тогда, когда раппорт $M_{\widehat{\boldsymbol{x}},\widehat{\boldsymbol{u}}}$ будет раппортом типа RE (продолжаемым вправо).

Доказательство предложено дать в упражнении 13.2.1. ■

Этот результат предлагает прямой подход к вычислению всех NE-раппортов в строке $\boldsymbol{x}$.

- **Шаг 1.** Вычисление всех NRE-раппортов (не RE-раппортов) в обеих строках $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$.
- **Шаг 2.** Сравнение (каким-либо образом) NRE-раппортов строки $\boldsymbol{x}$ с аналогичными раппортами обратной строки $\widehat{\boldsymbol{x}}$ — согласно лемме 13.2.1, совпадающие раппорты будут NE-раппортами.

Этот, простой с виду подход обладает труднопреодолимыми недостатками, связанными с тем, что раппортов типов NRE и NLE может быть очень много,

и с тем, что не совсем понятно, каким образом их можно эффективно сравнивать друг с другом. В частности, этот подход может быть очень дорогостоящим и по времени вычисления, и по объему памяти. И тем не менее, использование, в сущности, этого подхода представляется привлекательным по двум причинам.

- Как отмечено в подразделе 12.1.1, NRE-раппорты естественно представимы в виде дерева суффиксов (и, как мы увидим позже, также в виде массива суффиксов). Таким образом, шаг 1 может быть реализован путем построения деревьев (массивов) суффиксов для строки $\boldsymbol{x}$ и для обратной строки $\widehat{\boldsymbol{x}}$.

  Для примера рассмотрим дерево суффиксов, построенное для нашего обычного примера строки Фибоначчи $\boldsymbol{f}_6$, показанного на рис. 13.4 в упрощенном виде, немного отличающемся от рис. 12.1. Листья дерева (квадратики) помечены начальными позициями $i \in 1..13$ суффиксов $\boldsymbol{f}_6[i..13]$, в то время как каждый внутренний узел (кружок) помечен числом, равным длине наибольшего общего префикса (lcp) совокупности суффиксов, идентифицированных расположенными ниже листьями. Таким образом, значения lcp определяют длины полных NRE-раппортов в строке. Поэтому такое дерево суффиксов назовем ***NRE-деревом***, внутренние (круглые) и конечные (квадратные) узлы которого будем называть соответственно ***lcp-узлами*** и ***позиционными узлами.***

  В нашем примере подстроки $\boldsymbol{u} = \boldsymbol{x}[9..13] = \boldsymbol{x}[6..10] = \boldsymbol{x}[1..5]$ определяются поддеревом с корнем в lcp-узле 5 и составляют полный NRE-раппорт $M^*_{\boldsymbol{x},\boldsymbol{u}} = (5;\ 9, 6, 1)$.

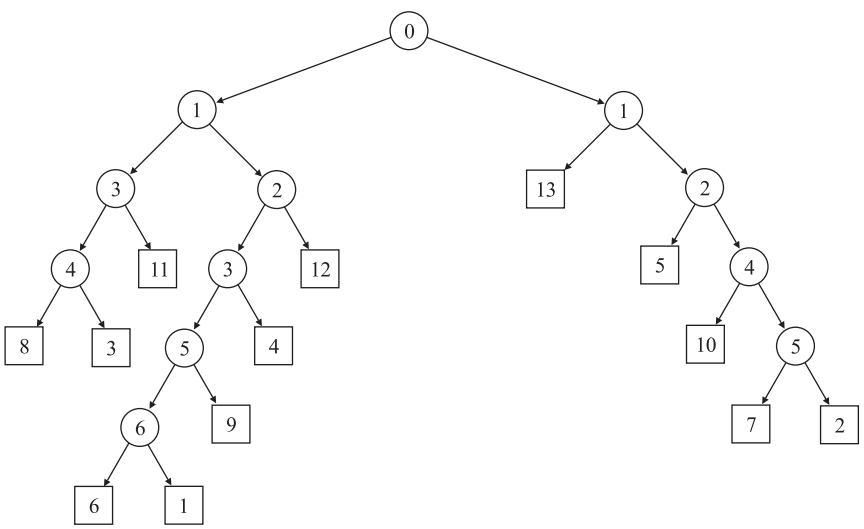


Рис. 13.4. NRE-дерево для строки $\boldsymbol{f}_6$

- Как показано ниже, шаг 2 можно реализовать с помощью линейной по времени процедуры, которая определяет совпадения между NRE-деревьями (массивами) строк $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$ для создания NE-дерева (NE-массива), которое описывает все NE-раппорты в строке $\boldsymbol{x}$.

Основываясь на этих замечаниях, в подразделе 13.2.1 покажем, как, имея NRE-деревья для $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$, можно вычислить NE-дерево, затем в подразделе 13.2.2 покажем, как можно вычислить NE-массив на основе известных массивов суффиксов строк $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$.

13.2.1 Вычисление NE-дерева

Предполагаем, что деревья суффиксов $T_{\boldsymbol{x}}$ и $T_{\widehat{\boldsymbol{x}}}$ для строк $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$ уже вычислены, определив таким образом полные NRE-раппорты этих строк. Из леммы 13.2.1 следует, что NRE-раппорты строки $\widehat{\boldsymbol{x}}$ определяют NLE-раппорты строки $\boldsymbol{x}$. Для краткости образующую $\boldsymbol{u}$ раппорта $M_{\boldsymbol{x},\boldsymbol{u}}$ типа NRE (соответственно, NLE и NE) назовем повторяющейся NRE-подстрокой (соответственно, NLE- и NE-подстрокой), при этом просим читателя иметь в виду, что понятия NRE, NLE, NE связаны именно с раппортом, который является совокупностью повторяющихся подстрок.

Предположим, что некоторый lcp-узел $\widehat{p}$ ($\widehat{p} > 0$) в дереве $T_{\widehat{\boldsymbol{x}}}$ имеет в качестве узла-сына позиционный узел $\widehat{\imath}$. Тогда в $\widehat{\boldsymbol{x}}$ существует повторяющаяся NRE-подстрока $\widehat{\boldsymbol{u}} = \widehat{\boldsymbol{x}}[\widehat{\imath}..\widehat{\imath} + \widehat{p} - 1]$, обратная повторяющейся NLE-подстроке $\boldsymbol{u} = \boldsymbol{x}[n - (\widehat{\imath} + \widehat{p} - 1) + 1..n - \widehat{\imath} + 1]$ строки $\boldsymbol{x}$. Таким образом, присваивание

$$i \leftarrow n - (\widehat{\imath} + \widehat{p} - 2) \tag{13.6}$$

определяет одну начальную позицию i в $\boldsymbol{x}$ повторяющейся NLE-подстроки $\boldsymbol{u}$ длиной $\widehat{p}$. Заметим, что фактически i является также начальной позицией одной повторяющейся NLE-подстроки из набора таких подстрок, имеющих длину $j \in \in 1..\widehat{p}$. Таким образом, если в $T_{\boldsymbol{x}}$ мы находим для некоторых i наибольшее значение $\widehat{p}$, такое, что i будет одной из совокупности подстрок длиной $\widehat{p}$, которые являются и NRE-, и NLE-подстроками, тогда каждый родитель $\widehat{p}$ в $T_{\boldsymbol{x}}$ также определяет совокупности подстрок, которые являются и NRE-, и NLE-подстроками.

Назовем подстроку $\boldsymbol{u}$ строки $\boldsymbol{x}$ ***максимальной повторяющейся NE-подстрокой*** строки $\boldsymbol{x}$, если

- $\boldsymbol{u}$ встречается в строке $\boldsymbol{x}$ по меньшей мере дважды;
- $\boldsymbol{u}$ не является собственной подстрокой любой повторяющейся подстроки строки $\boldsymbol{x}$.

Следующий результат говорит о том, что максимальная повторяющаяся NE-подстрока должна определяться как в $T_{\boldsymbol{x}}$, так и в $T_{\widehat{\boldsymbol{x}}}$.

Лемма 13.2.2. Если $\boldsymbol{u} = \boldsymbol{x}[i..i+p-1]$ является максимальной повторяющейся NE-подстрокой строки $\boldsymbol{x}$, тогда позиционный узел i будет узлом-сыном lcp-узла в $T_{\boldsymbol{x}}$, а узел $n-(i-p-2)$ — сыном p в $T_{\widehat{\boldsymbol{x}}}$.

Доказательство предложено дать в упражнении 13.2.4. ■

Вспомним сделанное выше замечание, что каждый lcp-узел в $T_{\boldsymbol{x}}$, который является предком повторяющейся NE-подстроки, сам должен быть корнем поддерева $T_{\boldsymbol{x}}$, позиционные узлы которого являются повторяющимися подстроками полного NE-раппорта. Поэтому последняя лемма обеспечивает нас простой стратегией для определения всех повторяющихся NE-подстрок: требуется найти только максимальные подстроки (с наибольшим значением p), затем определить местонахождение всех их предков в дереве $T_{\boldsymbol{x}}$. Алгоритм 13.2.1 реализует эту схему.

Алгоритм 13.2.1 (Вычисление NE-дерева)

▷ *Заданы деревья суффиксов $T_{\boldsymbol{x}}$ и $T_{\widehat{\boldsymbol{x}}}$, вычисляется NE-дерево для $\boldsymbol{x}[1..n]$*

1: Обход дерева $T_{\boldsymbol{x}}$ для создания таблицы POINTER, где для каждой позиции i в строке $\boldsymbol{x}$ POINTER$[i]$ является указателем на соответствующий узел в $T_{\boldsymbol{x}}$

2: **for** каждый lcp-узел в $T_{\boldsymbol{x}}$ **do**
    NE$[p] \leftarrow$ `FALSE`

3: **for** каждая пара родитель–сын $(\widehat{\imath},\widehat{p}\,)$ в $T_{\widehat{\boldsymbol{x}}}$ **do**
    ▷ *Здесь используется* POINTER$[i]$*:*
    if $i = n-(\widehat{\imath}+\widehat{p}-2)$ является сыном lcp-узла в $T_{\boldsymbol{x}}$ **then**
        while not NE$[\widehat{p}]$ **do**
            NE$[\widehat{p}\,] \leftarrow$ `TRUE`
            if $\widehat{p} \neq 0$ **then**
                $\widehat{p} \leftarrow$ родитель $\widehat{p}$ в $T_{\boldsymbol{x}}$

4: Обход $T_{\boldsymbol{x}}$ для удаления каждого поддерева с корнем в lcp-узле p, для которого NE$[p]$ = `FALSE`

Алгоритм разбит на четыре шага. Шаг 1 — это обход дерева $T_{\boldsymbol{x}}$, во время которого создается таблица, предоставляющая доступ к каждому позиционному узлу i в дереве $T_{\boldsymbol{x}}$ за константное время. На шаге 2 булева переменная, соответствующая каждому lcp-узлу p в $T_{\boldsymbol{x}}$, устанавливается в исходное состояние `FALSE`, указывая на то, что ни один из конечных узлов в поддереве с корнем в p не был на данный момент определен как NE-узел. На шаге 3 обрабатывается каждая пара родитель–сын $(\widehat{\imath},\widehat{p}\,)$ дерева $T_{\widehat{\boldsymbol{x}}}$ с целью определить, существует ли в $T_{\boldsymbol{x}}$ эквивалентная пара родитель–сын $(\widehat{p}, n-(\widehat{\imath}+\widehat{p}-2))$: если это так, тогда lcp-узел $\widehat{p}$ и все его предки в $T_{\boldsymbol{x}}$ должны быть типа NE. Соответственно, пока не будет найден предок, который уже NE типа, $\widehat{p}$ и его предки в $T_{\boldsymbol{x}}$ идентифицируются как NE типа. На заключительном шаге выполняется обход дерева $T_{\boldsymbol{x}}$ с целью удаления всех

поддеревьев с корнем в любом узле p, для которого $\mathrm{NE}[p] = \texttt{FALSE}$: оставшееся дерево $T_{\boldsymbol{x}}^{\mathrm{NE}}$ является NE-деревом строки $\boldsymbol{x}$.

На шагах 1, 2 и 4 алгоритма 13.2.1 совершаются обходы NRE-дерева, при этом для обработки каждого узла необходимо время порядка $O(1)$, следовательно, для обработки всех узлов необходимо время порядка $O(n)$. На шаге 3 также совершается обход дерева, но с более сложной обработкой узлов: поскольку оператор **if** использует массив POINTER, он при каждом выполнении также требует только $O(1)$ времени, следовательно, полное время выполнения этого оператора будет также порядка $O(n)$. В цикле **while** шага 3 переменная NE для не более n lcp-узлов устанавливается в состояние TRUE, а также проверяется текущее состояние не более n lcp-узлов, для которых переменная NE уже установлена в TRUE. Таким образом, полное время, затрачиваемое на выполнение цикла **while**, также имеет порядок $O(n)$. Следовательно, справедлива теорема.

Теорема 13.2.3. Если известны деревья суффиксов $T_{\boldsymbol{x}}$ и $T_{\widehat{\boldsymbol{x}}}$ для строки $\boldsymbol{x}[1..n]$, алгоритм 13.2.1 правильно вычисляет NE-дерево за время порядка $O(n)$ с использованием памяти порядка $\Theta(n)$. ■

На рис. 13.5 изображено дерево суффиксов $T_{\widehat{\boldsymbol{f}_6}}$, измененное таким образом, что позиционные узлы $\widehat{i}$ заменены на узлы $i = n - (\widehat{i} + \widehat{p} - 2)$. Отметим, что только позиции 1, 6 и 9 приводят к NLE-раппортам. Таким образом, NE-дерево сводится до поддерева дерева $T_{\boldsymbol{f}_6}$, которое соответствует этим позициям — это пути от корня дерева $T_{\boldsymbol{f}_6}$ до lcp-узла 6, который соответствует подстроке $abaaba$. Как видно из рис. 13.4, все полные раппорты строки $\boldsymbol{f}_6$ являются вхождениями подстрок $a, ab, aba, abaab$ и $abaaba$. В общем случае вхождения NE-раппортов, всех из них или выбранных в соответствии с какими-либо критериями, можно локализовать и вывести путем обхода NE-дерева за время, пропорциональное их количеству. Такие задачи обсуждаются в работе [41]. Другой, совершенно отличный подход

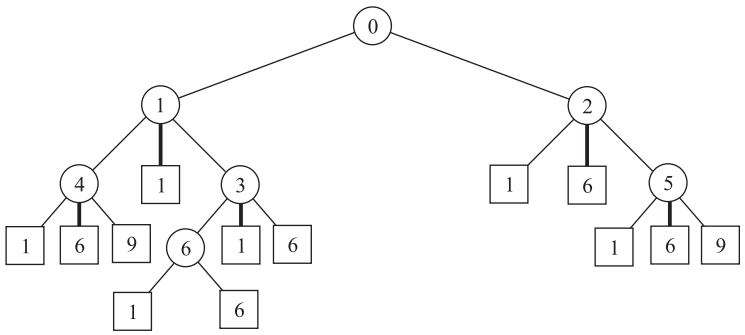


Рис. 13.5. Дерево $T_{\widehat{\boldsymbol{f}_6}}$ для $\widehat{\boldsymbol{f}_6}$ с замененными узлами

к вычислению непродолжаемых раппортов (хотя тоже с использованием деревьев суффиксов) описан в [108].

13.2.2 Вычисление NE-массива

В подразделе 5.3.2 мы определили массив суффиксов $\boldsymbol{\sigma}_{\boldsymbol{x}} = \boldsymbol{\sigma}[1..n]$ для заданной строки $\boldsymbol{x}[1..n]$ как перестановку чисел $1..n$ с таким свойством, что $\boldsymbol{x}[\boldsymbol{\sigma}_{\boldsymbol{x}}[i]..n] < \boldsymbol{x}[\boldsymbol{\sigma}_{\boldsymbol{x}}[j]..n]$ для всех $1 \leqslant i < j \leqslant n$. Таким образом, $\boldsymbol{\sigma}_{\boldsymbol{x}}$ определяет начальные позиции суффиксов строки $\boldsymbol{x}$ в возрастающем лексикографическом порядке. Например, если

$$\begin{array}{rcccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \boldsymbol{f}_6 = & a & b & a & a & b & a & b & a & a & b & a & a & b \end{array} \tag{13.7}$$

тогда

$$\boldsymbol{\sigma}_{\boldsymbol{f}_6} = 11 \quad 8 \quad 3 \quad 12 \quad 9 \quad 6 \quad 1 \quad 4 \quad 13 \quad 10 \quad 7 \quad 2 \quad 5. \tag{13.8}$$

В подразделе 5.3.2 также показно, как путем присваивания lcp-значений диапазонам позиций в $\boldsymbol{\sigma}_{\boldsymbol{x}}$ можно определить вспомогательный массив $\boldsymbol{\pi}_{\boldsymbol{x}}$. Здесь мы определим более простой массив $\boldsymbol{\lambda}_{\boldsymbol{x}} = \boldsymbol{\lambda}[1..n]$, где $\boldsymbol{\lambda}[1] = 0$, а для каждого $i \in 2..n$

$$\boldsymbol{\lambda}[i] = \mathrm{lcp}(\boldsymbol{x}[\boldsymbol{\sigma}[i-1]..n], \boldsymbol{x}[\boldsymbol{\sigma}[i]..n]]).$$

Будем использовать сокращенную форму записи:

$$\boldsymbol{\lambda}[i] = \mathrm{lcp}(\boldsymbol{\sigma}[i-1], \boldsymbol{\sigma}[i]). \tag{13.9}$$

Например, в нашем примере

$$\boldsymbol{\lambda}_{\boldsymbol{f}_6} = 0341256301452, \tag{13.10}$$

а это свидетельствует о том, что $\mathrm{lcp}(11, 8) = 3$, $\mathrm{lcp}(8, 3) = 4$ и т.д. Так же, как и в случае массива $\boldsymbol{\pi}_x$, массив $\boldsymbol{\lambda}_{\boldsymbol{x}}$ можно вычислить в качестве промежуточного результата при вычислении массива суффиксов. Напомним (подраздел 5.3.2), что массивы суффиксов можно вычислить независимо от дерева суффиксов [172].

Мы представим здесь схему одного алгоритма, в котором не используются деревья суффиксов: все вычисление выполняется на основе массивов суффиксов (как $\boldsymbol{\sigma}_{\boldsymbol{x}}$, так и $\boldsymbol{\lambda}_{\boldsymbol{x}}$), и его конечным результатом является NE-массив, а не NE-дерево. Поскольку для хранения массива суффиксов строки $\boldsymbol{x}[1..n]$ требуется только $2n$ компьютерных слов, то этот алгоритм приводит к существенной экономии памяти.

Пусть $p_j = \boldsymbol{\lambda}[j]$ для каждого $j \in 1..n$. Тогда из выражения (13.9) следует, что для любого $j \in 2..n$

if $p_j > p_{j-1}$ **then**
    p_j — потомок p_{j-1} в дереве суффиксов
else if $p_j < p_{j-1}$ **then**
    p_j — предок p_{j-1} в дереве суффиксов
else
    p_j и p_{j-1} определяют один и тот же узел в дереве суффиксов

Поскольку массив суффиксов содержит такую же информацию о длинах наибольших общих префиксов, которая предоставляется деревом суффиксов, то эти отношения подразумевают, что массив суффиксов должен быть разложен на подмассивы, каждый из которых определяет узлы, лежащие на одном пути от корня до конечного узла в дереве суффиксов. Например, значения lcp в массиве (13.10) могут быть разделены на три подмассива

$$0, 3, 4, \underline{1}/\underline{1}, 2, 5, 6, 3, \underline{0}/\underline{0}, 1, 4, 5, 2,$$

соответствующие трем основным путям в дереве суффиксов, показанном на рис. 13.4. Здесь повторяющиеся значения $\underline{0}$ и $\underline{1}$ определяют корни поддеревьев, в которых начинаются новые пути. Поэтому назовем такие узлы ***узлами ветвления***, поскольку они отмечают точки, в которых новый путь отклоняется от предыдущего пути.

Эти понятия можно ввести более точно следующим образом.

Определение 13.2.4. ***I*-серией** $I_{j,h}$ в $\boldsymbol{\lambda_x}$ называется последовательность из $h \geqslant 2$ lcp-значений $p_j, p_{j+1}, \ldots, p_{j+h-1}$ (которые не все равны), таких, что

а) или $j = 1$ или $p_{j-1} > p_j$;

б) $p_j \leqslant p_{j+1} \leqslant \cdots \leqslant p_{j+h-1}$;

в) или $j + h - 1 = n$ или $p_{j+h-1} > p_{j+h}$. ■

Таким образом, I-серия является неубывающей последовательностью lcp-значений в $\boldsymbol{\lambda_x}$. Аналогично ***D*-серией** $D_{j,h}$ назовем невозрастающую последовательность lcp-значений. Теперь, используя эти определения, можно охарактеризовать пути в дереве суффиксов.

Лемма 13.2.5. В $\boldsymbol{\lambda_x}$ последовательности $I_{j,h}D_{j+h-1,h'}$ и $I_{j,h}$, где $j + h - 1 = n$, определяют узлы, которые в дереве суффиксов лежат на одном пути.

Доказательство. Поскольку $p_1 = 0$, то из определения 13.2.4 следует, что или в $\boldsymbol{\lambda_x}$ нет серий ($p_j = 0$ для всех $j \in 1..n$), или последовательность серий начинается с I-серии $I_{j,h}$. При этом определение 13.2.4 гарантирует, что за каждой I-серией следует D-серия, иначе достигается конец массива; аналогично за каждой D-серией следует I-серия либо достигается конец массива.

Теперь рассмотрим I-серию $I_{j,h}$. Поскольку $p_{j+(t-1)} \leqslant p_{j+t}$ для каждого $t \in 1..h-1$, то из этого следует, что $I_{j,h}$ определяет последовательность неубывающих p_j, которые лежат на одном пути в дереве суффиксов. Если $i + h = n$, тогда $I_{j,h}$ — конечный путь, определенный в $\boldsymbol{\lambda_x}$. Если же нет, тогда за $I_{j,h}$ следует D-серия $D_{j+h-1,h'}$, где $p_{(j+h-1)+(t+1)} \leqslant p_{(j+h-1)+t}$ для каждого $t \in 1..h'-1$. Эти lcp-значения определяют последовательность предков p_{j+h-1}, лежащих на одном пути от корня, уже определенного с помощью $I_{j,h}$. Последующая I-серия, если она существует, естественно, определяет потомки узла ветвления $p_{j+h+h'-2}$, который лежит на пути, отличном от пути, определенного с помощью $I_{j,h}$. ■

Очевидно, что в $\boldsymbol{\lambda_x}$ ID-серии, по существу, разбивают внутренние (lcp) узлы массива суффиксов на отдельные пути. При этом смежные пары серий определяют в дереве два пути, которые имеют только один общий узел (узел ветвления). Заметим также, что для каждой ID-серии минимальный узел на соответствующем пути должен быть или первым узлом I-серии, или последним узлом D-серии.

Поскольку узел ветвления является *как* последним узлом D-серии, *так и* первым узлом I-серии, то значения пар смежных узлов в $\boldsymbol{\lambda_x}$ определяют способ, каким связаны два пути в дереве суффиксов. Если ввести в $\boldsymbol{\lambda_x}$ первое lcp-значение ($p_1 = 0$) в качестве узла ветвления и обозначить последовательные узлы ветвления соответственно как b_1 и b_2, то нетрудно заметить, что есть только три возможных отношения между соответствующими путями, которые показаны на рис. 13.6. Отметим, что в каждом из этих трех случаев будущие пути могут быть добавлены на текущем пути только в качестве потомков b_2 или потомков предков b_2. Для пути с корнем в b_2, который был определен предыдущим путем, содержащим b_1, никакого дальнейшего изменения не может быть сделано.

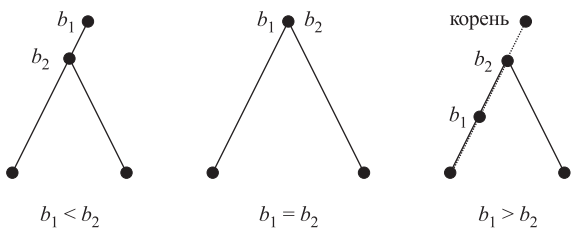


Рис. 13.6. Узлы ветвления b_1 и b_2 смежных путей

Наконец, заметим, что регулярность структуры массива суффиксов дает возможность определить в соответствующем дереве суффиксов $T_{\boldsymbol{x}}$ родителя каждого позиционного узла $\boldsymbol{\sigma}[j]$, $j = 1, 2, \ldots, n$. Сделаем два важных утверждения, доказательство которых оставим для упражнения 13.2.12.

O1. Родитель $\boldsymbol{\sigma}[j]$ в $T_{\boldsymbol{x}}$ равен $\max\{\boldsymbol{\lambda}[j], \boldsymbol{\lambda}[j+1]\}$, где принимаем, что $\boldsymbol{\lambda}[n+1] = 0$.

O2. (По Эндрю Фрэнсису (Andrew Francis)) Если $\boldsymbol{\lambda}[j]$ — узел ветвления и $j < n$, то $\boldsymbol{\sigma}[j]$ имеет родителя $\boldsymbol{\lambda}[j+1]$ в $T_{\boldsymbol{x}}$; иначе родителем $\boldsymbol{\sigma}[j]$ будет $\boldsymbol{\lambda}[j]$.

Эти утверждения предоставляют альтернативные способы определения узлов ветвления в $\boldsymbol{\lambda}_{\boldsymbol{x}}$.

Теперь опишем метод вычисления NE-раппортов, который основан только на массиве суффиксов. Как и ранее, для строки $\boldsymbol{x}$ обозначим массив позиций и lcp-массив соответственно как $\boldsymbol{\sigma}_{\boldsymbol{x}}$ и $\boldsymbol{\lambda}_{\boldsymbol{x}}$; а для обратной строки $\widehat{\boldsymbol{x}}$ обозначим соответствующие массивы как $\widehat{\boldsymbol{\sigma}_{\boldsymbol{x}}}$ и $\widehat{\boldsymbol{\lambda}_{\boldsymbol{x}}}$. Введем также массив LOC, который для каждого позиционного узла $\widehat{\imath}$ в $T_{\widehat{\boldsymbol{x}}}$ покажет его местоположение в массиве $\widehat{\boldsymbol{\sigma}_{\boldsymbol{x}}}$; это даст возможность найти любую позицию в $\widehat{\boldsymbol{\sigma}_{\boldsymbol{x}}}$ за фиксированное время. Бинарный вектор NE$[1..n]$ определяет для каждого $j \in 1..n$ состояние lcp-узла $\boldsymbol{\lambda}[j]$ (`TRUE` или `FALSE`), а также его тип (NLE или нет). Наконец, чтобы определить начало и конец путей ID в $\boldsymbol{\lambda}_{\boldsymbol{x}}$, используем паттерны серий ID (лемма 13.2.5) для вычисления массива BRANCH, который, кроме узлов ветвления, определенных выше, включает также первые и последние позиции в $\boldsymbol{\lambda}_{\boldsymbol{x}}$. В результате получим более сложную схему вычислений, чем аналогичная схема на основе деревьев. Однако алгоритм 13.2.2, построенный по этой схеме, не требует бóльших временны́х затрат, чем алгоритм 13.2.1.

Алгоритм 13.2.2 (Вычисление NE-массива)

▷ *Известны массивы суффиксов (NRE-массивы) для строк* $\boldsymbol{x} = \boldsymbol{x}[1..n]$ *и* $\widehat{\boldsymbol{x}}$,
▷ *вычисляется NE-массив для строки* $\boldsymbol{x}$

1: **for** $j \leftarrow 1$ **to** n **do**
   LOC$[\widehat{\boldsymbol{\sigma}}[j]] \leftarrow j$; NE$[j] \leftarrow$ `FALSE`
2: вычисление BRANCH$[1..b^*]$, где для $b \in 1..b^*$ BRANCH$[b]$
   является позицией b-го узла ветвления в $\boldsymbol{\lambda}_{\boldsymbol{x}}$
   (позиции $j = 1$ и $j = n$ рассматриваются как узлы ветвления)
3: $b \leftarrow 1$; $p_0 \leftarrow 0$
   for $j \leftarrow 1$ **to** n **do**
      определение j', такого, что $(p, i) = (\boldsymbol{\lambda}[j'], \boldsymbol{\sigma}[j])$ является
      парой родитель–сын в дереве суффиксов строки $\boldsymbol{x}$
      if $p > 0$ **then**
         if $p < p_0$ **then**
            NE$[j'] \leftarrow$ `TRUE`; $p_0 \leftarrow p$
         else
            $\widehat{\imath} \leftarrow n - (i + p - 2)$; $\widehat{\jmath} \leftarrow$ LOC$[\widehat{\imath}]$
            определение $\widehat{j'}$, такого, что $(\widehat{p}, \widehat{\imath}) = (\widehat{\boldsymbol{\lambda}}[\widehat{j'}], \widehat{\boldsymbol{\sigma}}[\widehat{\jmath}])$
            является парой родитель–сын в дереве
            суффиксов строки $\widehat{\boldsymbol{x}}$

if $p = \widehat{p}$ **then**
NE$[j'] \leftarrow$ TRUE; $p_0 \leftarrow p$
if $j =$ BRANCH$[b+1]$ **then**
определение $j^* \in$ BRANCH$[b]$..BRANCH$[b+1]$,
такого, что $p^* = \boldsymbol{\lambda}[j^*]$ равен максимальной lcp,
для которой NE$[j^*]$ = TRUE
for каждого $j^* \in$ BRANCH$[b]$..BRANCH$[b+1]$ **do**
if $\boldsymbol{\lambda}[j'] \leqslant p^*$ **then**
NE$[j'] \leftarrow$ TRUE
$b \leftarrow b+1$

4: обработка пути, определяемого с помощью BRANCH, в обратном порядке, при установке типа NE (как в шаге 3) для предков p^*

Шаг 1 этого алгоритма — простой цикл **for**, который инициализирует массив LOC и бинарный вектор NE. Шаг 2 на основе леммы 13.2.5 вычисляет массив BRANCH и выполняет линейный просмотр массива $\boldsymbol{\lambda}[1..n]$. На шаге 3 выполняется другой цикл **for**, который сначала на основе утверждения O1 определяет все совпадения пар (p, i) в массивах NRE строк $\boldsymbol{x}$ и $\widehat{\boldsymbol{x}}$, затем для каждого пути выполняет проверку, имеет ли каждый предок совпавших пар (p, i) в пределах этого пути тип NE. На выполнение каждого из двух этапов шага 3 необходимо время порядка $\Theta(n)$. На шаге 4 повторяется обработка второго этапа шага 3, но в обратном порядке — это гарантирует, что все предки NE-элементов установлены должным образом и тоже имеют тип NE. В конце алгоритма NE-массив определяется всеми теми позициями j в $\boldsymbol{\lambda}_{\boldsymbol{x}}$, для которых $NE[j] =$ TRUE. Таким образом, имеет место следующая теорема.

Теорема 13.2.6. Для заданных массивов суффиксов (NRE-массивов), соответствующих строкам $\boldsymbol{x}[1..n]$ и $\widehat{\boldsymbol{x}}$, алгоритм 13.2.2 корректно вычисляет NE-массив за время порядка $\Theta(n)$ с использованием памяти объемом порядка $\Theta(n)$. ■

Отметим, что, при необходимости, для вычисления NE-дерева за линейное время можно использовать соответствующий NE-массив.

Поскольку NE-дерево (или соответствующий ему NE-массив) определяет *все* NE-раппорты в строке $\boldsymbol{x}$, то его можно использовать для определения последовательных NE-раппортов (повторений) и NE-подстрок, имеющих оболочки (задача 2.17 из раздела 2.3). Напомним, что листы поддерева с корнем в любом lcp-узле p в NE-дереве являются точными NE-раппортами, соответствующими p. Если эти узлы были отсортированы в последовательность $I = <i_1, i_2, \ldots, i_r>$, где $i_1 < i_2 < \cdots < i_r$, тогда за единственный просмотр последовательности I можно легко определить как последовательные раппорты максимальной длины, так и подстроки максимальной длины, имеющие оболочки и соответствующие узлу p.

В первом случае лакуна между смежными входами в I должна быть в точности равна p, во втором случае не должна превышать p. Для каждого узла p этот процесс можно выполнить за время порядка $O(n)$ с использованием памяти такого же порядка.

Однако более сложной является проблема представления в строке $\boldsymbol{x}$ всех NE-подстрок, имеющих оболочки. Существует три алгоритма, имеющих отношение к проблеме вычисления всех таких подстрок типа NRE (с использованием дерева типа NRE, а не NE). Первый алгоритм, представленный в работе [15], выполняет сложные операции на NRE-дереве (дереве суффиксов) и требует времени порядка $O(n \log n)$, в то время как второй алгоритм [42] применяет еще более сложные методы по отношению к NRE-дереву для уменьшения затрат времени до величины порядка $O(n \log n)$. Третий алгоритм [124], непосредственно обрабатывающий последовательности, определенные декомпозицией Крочемора, также выполняется за время порядка $O(n \log n)$.

Научный мир ожидает "простого" алгоритма с временем выполнения порядка $O(n \log n)$, который вычислял бы NE-подстроки, имеющие оболочки, строки $\boldsymbol{x}$ предпочтительно со скромными требованиями по отношению к необходимому объему памяти.

Упражнения 13.2

1. Докажите лемму 13.2.1.
2. Определение дважды связанного списка, использованное в алгоритме ФСТ, неизбежно приводит к определению строки, данному в разделе 1.1. Если дважды связанный список действительно является строкой, тогда каким алфавитом он определяется?
3. Покажите, что ни один узел в NRE-дереве, возможно за исключением корня, не может иметь только единственного узла–сына. Определите условия, при которых корень действительно имеет единственный узел–сын.
4. Докажите лемму 13.2.2.
5. Вычислите NRE-массив строки $\widehat{\boldsymbol{f}_6}$, затем используйте его для проверки корректности NRE-дерева на рис. 13.5. Далее используйте NRE-деревья, показанные на рис. 13.4 и 13.5, для вычисления NE-дерева для строки $\boldsymbol{f}_6$ в соответствии с алгоритмом 13.2.1.
6. Строка $\boldsymbol{g} = abaababaababa$ отличается от строки $\boldsymbol{f}_6$ только своими последними двумя позициями. Вычислите NRE-массивы строк $\boldsymbol{g}$ и $\widehat{\boldsymbol{g}}$, затем NRE-деревья, а потом NE-дерево для строки $\boldsymbol{g}$. Подумайте над поразительными различиями между этими структурами и аналогичными структурами из предыдущего упражнения.

7. Заметим, что для каждой позиции i, вычисленной для узла $n - (\widehat{i} + \widehat{p} - 2)$ дерева $\widehat{T_{\boldsymbol{x}}}$, имеется только один возможный узел–родитель в дереве $T_{\boldsymbol{x}}$. Объясните, каким образом использование поразрядной сортировки может улучшить алгоритм 13.2.1.

8. Запишите алгоритм, который на основе NRE-массива длиной n вычислял бы соответствующую строку $\boldsymbol{x}[1..n]$, определенную на индексированном алфавите $1, 2, \ldots, \alpha$.

9. Дайте точное определение D-серии в стиле определения 13.2.4.

10. Опишите алгоритм, который вычисляет NE-дерево строки $\boldsymbol{x}$ на основе NE-массива.

11. Опишите алгоритм, который вычисляет как последовательные NE-раппорты, так и подстроки типа NE, имеющие оболочки, соответствующие произвольному lcp-узлу p в NE-дереве.

12. Докажите утверждения O1 и O2.

13.3 *k*-приближенные раппорты — алгоритм Шмидта

В этом разделе описана схема алгоритма, который в общем случае эффективно вычисляет аппроксимирующие (приближенные) раппорты. Этот алгоритм особенно интересен специалистам в области молекулярной биологии, где в вычислении расстояния между строками используется общая матрица весов (раздел 2.2). Алгоритм, взятый из работы Шмидта (Schmidt) [205], является сложным алгоритмом с большим количеством довольно беспорядочных специальных терминов. В этой книге мы старались избегать описания таких алгоритмов. Однако здесь мы рассмотрим алгоритм Шмидта в *краткой* форме, поскольку он предоставляет действительно эффективное, возможно, даже асимптотически оптимальное, решение особо важной проблемы обработки строковых последовательностей. Кроме того, алгоритм Шмидта расширяет и объединяет несколько рассмотренных ранее в этой книге алгоритмических идей таким способом, который обеспечивает новое понимание их значения и эффективности.

■ Алгоритм Шмидта основан на обычном подходе динамического программирования к массиву стоимостей $\boldsymbol{c}$ (раздел 9.1), но теперь с иной интерпретацией, применимой к повторяющимся подстрокам. Часть этой новой интерпретации включает вычисление максимальных "баллов", а не минимального расстояния; таким образом, массив стоимостей $\boldsymbol{c}$, в котором расстояние минимизировано, становится ***массивом полезности*** (benefit array) $\boldsymbol{b}$, в котором эти баллы максимизированы.

- Понятие графа зависимостей, введенное в разделе 9.5 в качестве основы алгоритма Укконена–Майерса для вычисления расстояния между строками, вновь появляется здесь в виде ***сеточного графа*** (grid graph), "пути с наивысшим баллом" которого являются основным предметом рассмотрения алгоритма Шмидта.
- Понятие непродолжаемости, введенное в разделе 2.3 и затем примененное в разделе 12.2 к точным кратным строкам, а в разделе 13.2 к точным раппортам, для приближенных раппортов переформулировано в понятие ***локальной оптимальности***. (Получается, что первоначальное определение локальной оптимальности [84] фактически предшествует определению непродолжаемости [168].)
- Рекурсивный метод декомпозиции, используемый в алгоритме Мейна—Лоренца (подраздел 12.1.2) для вычисления квадратов на основе их средних точек, расширен для вычисления приближенных последовательных квадратов в сеточных графах.

В дополнение к расширениям этих знакомых понятий, в алгоритме Шмидта представлены также некоторые новые интересные идеи, и мы просто не можем избежать их обсуждения!

По существу, этот алгоритм не вычисляет раппорты в принятом нами смысле; скорее, он вычисляет набор всех "локально оптимальных" (непродолжаемых) приближенных непокрываемых квадратов, которые удовлетворяют данному значению "точности" k. Как мы вскоре увидим, покрываемые квадраты исключаются, потому что их включение может привести к потере важных последовательных квадратов. Вычисление квадратов, а не раппортов с произвольным показателем степени связано с замечанием, сделанным в конце раздела 2.3, о нетранзитивности приближенного совпадения, следовательно, о присущей приближенным раппортам неопределенности. Например, в подстроке $\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_3$ может быть, что $d(\boldsymbol{u}_1, \boldsymbol{u}_2) \leqslant k$, $d(\boldsymbol{u}_2, \boldsymbol{u}_3) \leqslant k$. Вследствие этого подстроки $\boldsymbol{u}_1\boldsymbol{u}_2$ и $\boldsymbol{u}_2\boldsymbol{u}_3$ являются k-приближенными квадратами. Однако если $d(\boldsymbol{u}_1, \boldsymbol{u}_3) > k$, тогда $\boldsymbol{u}_1\boldsymbol{u}_3$ таковой не будет. Таким образом, понятие k-приближенных кратных строк или раппортов непросто определить удобным способом: в нестрогом смысле строки вида $\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_3$ могут быть k-приближенными кратными строками, хотя $\boldsymbol{u}_1\boldsymbol{u}_3$ не является даже k-приближенным квадратом. В то время в более строгом смысле обозначение $\boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_m$ как k-приближенной кратной строки может потребовать C_m^2 проверок возможных k-приближенных квадратов (как последовательных, так и расщепленных). Как отмечалось в разделе 2.3, образующие $\boldsymbol{u}_1$ и $\boldsymbol{u}_2 k$-приближенного квадрата $\boldsymbol{u}_1\boldsymbol{u}_2$ необязательно должны быть одинаковой длины.

Следовательно, для вычисления приближенных кратных строк, кажется, нет никакой надежды на использование какого-либо аналога тройки Крочемора (i, p, r) для кодирования точных кратных строк (подраздел 12.1.1), который уменьшает ко-

личество выводов до величины порядка $O(n \log n)$, и еще меньше надежды для использования аналога четверок (i, p, r, t) для кодирования серий (раздел 12.2), который уменьшает размер вывода до $O(n)$. Поэтому очевидно, что вычисление приближенных кратных строк или раппортов потребует вывода квадратов, следовательно, в наихудшем случае необходимо $\Omega(n^2)$ выводов (как показано в разделе 2.3 для строки $\boldsymbol{x} = a^n$). Это наблюдение подтверждается теоретическим результатом работы [120], в которой проблема расстояний между строками решена с помощью произвольной матрицы баллов за время порядка $\Omega(n^2)$.

Как мы увидим в следующем разделе, по существу, те же трудности возникают при определении и вычислении приближенных образующих строки $\boldsymbol{x}$.

Восстановленная симметрия

Напомним, что массив стоимостей $\boldsymbol{c}$ (или так называемая матрица динамического программирования) был сначала введен в разделе 9.1, где рассматривалась проблема вычисления расстояния между строками $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$. Эти вычисления выполнялись путем нахождения для всех $i \in 0..n_1$ и $j \in 0..n_2$ расстояний между подстроками $\boldsymbol{x}_1[1..i]$ и $\boldsymbol{x}_2[1..j]$; при этом начальные значения нулевых строки и столбца массива $\boldsymbol{c}$ вычислялись симметрично, определяя расстояния от пустой строки ε до $\boldsymbol{x}_2[j]$ и от $\boldsymbol{x}_1[i]$ до ε. На основе начальных значений другие элементы массива $\boldsymbol{c}$ вычисляются методом динамического программирования, как определено в лемме 9.1.1. Таким образом, на каждом шаге рассматривался *префикс*, и поэтому всегда выполнялось равенство

$$\boldsymbol{c}[i, j] = d(\boldsymbol{x}_1[1..i], \boldsymbol{x}_2[1..j]), \tag{13.11}$$

в котором $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ снова играют симметричные роли. Для этой задачи конечный результат всегда равен $\boldsymbol{c}[n_1, n_2]$.

Однако эта "приятная" симметрия была потеряна в главе 10 при использовании массива стоимостей $\boldsymbol{c}$ для нахождения k-приближенных паттернов $\boldsymbol{p}[1..m]$ в заданной строке $\boldsymbol{x}[1..n]$. Здесь для каждой позиции $i \in 1..n$ в $\boldsymbol{x}$ просматривалась непустая *подстрока* $\boldsymbol{x}[i'..i]$ ($i' \in 1..i$), что обеспечивало наилучшее k-приближенное совпадение с *префиксом* $\boldsymbol{p}[1..j]$ ($j \in 1..m$). Как объяснялось в разделе 10.1, новая задача требует того, чтобы элементы столбцов инициализировались другим способом. Таким образом, массив $\boldsymbol{c}$ становится массивом $\boldsymbol{c}[0..n, 1..m]$, но, как оказывается, для всех $i \in 1..n$ и $j \in 2..m$ все еще можно использовать тот же самый подход динамического программирования, который приводит к асимметричному соотношению

$$\boldsymbol{c}[i, j] = \min_{1 \leqslant i' \leqslant i} d(\boldsymbol{x}[i'..i], p[1..j]) \tag{13.12}$$

В этом случае k-приближенные совпадения паттерна $\boldsymbol{p}$ с $\boldsymbol{x}$ определяются с помощью позиций i в столбце m массива $\boldsymbol{c}$, для которых $\boldsymbol{c}[i, m] \leqslant k$.

Дальнейшее обобщение задачи происходит тогда, когда мы пытаемся определить для каждого $i \in 1..n_1$ и каждого $j \in 0..n_2$ совпадения с заданным уровнем k между непустыми подстроками $\boldsymbol{x}_1[i'..i]$ и $\boldsymbol{x}_2[j'..j]$ заданных строк $\boldsymbol{x}_1[1..n_1]$ и $\boldsymbol{x}_2[1..n_2]$. Тогда восстанавливается симметрия между первыми строкой и столбцом массива $\boldsymbol{c} = \boldsymbol{c}[1..n_1, 1..n_2]$, а каждый элемент $\boldsymbol{c}[i, j]$ удовлетворяет соотношению

$$\boldsymbol{c}[i,j] = \min_{\substack{1 \leqslant i' \leqslant i \\ 1 \leqslant j' \leqslant j}} d(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j]). \tag{13.13}$$

Здесь о совпадениях, представляющих интерес, сообщают значения $\boldsymbol{c}[i, j] \leqslant k$.

Как увидим ниже, массив стоимостей $\boldsymbol{c}$, элементы которого определяются выражением (13.13), предоставляет основу для вычисления k-приближенных квадратов при $\boldsymbol{x} = \boldsymbol{x}_1 = \boldsymbol{x}_2$, но с массивом полезности $\boldsymbol{b}$, заменяющим массив стоимостей $\boldsymbol{c}$. В этом случае элементы $\boldsymbol{b}[i..j] \geqslant k$, где $i \neq j$, будут определять квадрат, образованный двумя различными подстроками, удовлетворяющими соотношению

$$\boldsymbol{x}[i'..i] \overset{(k)}{=} \boldsymbol{x}[j'..j].$$

Однако, чтобы иметь возможность эффективного выполнения вычислений, сначала надо показать, что аналог выражения (13.13) можно вычислить с использованием подхода динамического программирования леммы 9.1.1. Этот вопрос рассмотрен в следующем подразделе.

Массив полезности b

Вначале рассмотрим реализацию вычисления выражения (13.13) на основе динамического программирования (см. лемму 9.1.1):

$$\begin{aligned} \boldsymbol{c}[i,j] \leftarrow \min\{&\boldsymbol{c}[i-1,j] + d(\boldsymbol{x}_1[i], \varepsilon), \boldsymbol{c}[i,j-1] + d(\varepsilon, \boldsymbol{x}_2[j]), \\ &\boldsymbol{c}[i-1,j-1] + d(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j])\}, \end{aligned} \tag{13.14}$$

используемого теперь для всех $i \in 2..n_1$ и $j \in 2..n_2$. Но мы сразу же столкнемся с трудностями, которые станут очевидными, если, как показано в табл. 13.1, применим выражение (13.13) к нашему примеру $\boldsymbol{x}_1 = rests$, $\boldsymbol{x}_2 = stress$, где используется расстояние преобразования d_E.

Сначала заметим, что единственными возможными значениями в этом массиве являются "бесполезные" значения 0 и 1: согласно (13.13), первое значение является минимумом, который имеет место всякий раз, когда есть позиции $i' \leqslant i$ и $j' \leqslant j$, для которых $\boldsymbol{x}_1[i'..i] = \boldsymbol{x}_2[j'..j]$, в то время как значение 1 имеет место тогда, когда $\boldsymbol{x}_1[i] \neq \boldsymbol{x}_2[j]$. Таким образом, выражение (13.13) фактически говорит нам только о совпадении или несовпадении позиций $\boldsymbol{x}_1[i]$ и $\boldsymbol{x}_2[j]$, не давая какой-либо информации о любом предыдущем диапазоне позиций в этих двух строках,

Таблица 13.1. Массив стоимостей для $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$, полученный с помощью d_E и (13.13)

	$\boldsymbol{j}$	**1**	**2**	**3**	**4**	**5**	**6**
i		s	t	r	e	s	s
1	r	1	1	0	1	1	1
2	e	1	1	1	0	1	1
3	s	0	1	1	1	0	0
4	t	1	0	1	1	1	1
5	s	0	1	1	1	0	0

в которых, возможно, имело место частичное совпадение. Но вторая трудность даже еще серьезнее: метод динамического программирования, кажется, больше не работает! Например, используя (13.14), мы не можем вычислить в табл. 13.1 $\boldsymbol{c}[2,2] = 1$ или $\boldsymbol{c}[5,5] = 0$.

Стандартный подход обычно обходит эти трудности. В работе [213] использовался массив, названный нами массивом полезности $\boldsymbol{b}$, в котором представляет интерес максимальная польза, а не минимальные стоимости. При этом расстояния между буквами $d(\lambda, \mu)$ заменены на ***баллы*** $S(\lambda, \mu)$, которые положительны в желательных обстоятельствах (например, если $\lambda = \mu$) и отрицательны (или, по крайней мере, имеют меньшие положительные значения) в нежелательных обстоятельствах (например, если $\lambda \neq \mu$). Представим эти баллы как элементы ***матрицы баллов*** S, аналогичной матрице весов W, представленной в разделе 2.2, за исключением того, что теперь элементы с бóльшим значением, как правило, определяют совпадения букв, а не их несовпадения. Конечно, использование такой матрицы требует индексированного алфавита, как было указано в разделе 4.1.

Предположим, что для нашего примера матрица баллов S представима в следующем виде.

$$S = \begin{array}{c|ccccc} & \varepsilon & e & r & s & t \\ \varepsilon & 0 & -1 & -1 & -1 & -1 \\ e & -1 & 2 & -1 & -1 & -1 \\ r & -1 & -1 & 2 & -1 & -1 \\ s & -1 & -1 & -1 & 2 & -1 \\ t & -1 & -1 & -1 & -1 & 2 \end{array} \qquad (13.15)$$

Таким образом, в этом простом случае штрафом (отрицательной полезностью), равным -1, определяется любое несовпадение, удаление или вставка, в то время как полезность, равная $+2$, предоставляется за любое совпадение, за исключением, как обычно, пустой строки: $S(\varepsilon, \varepsilon) = 0$. Заметим, что в некоторых случаях,

например в биологических приложениях, некоторые баллы $S(\lambda, \mu)$ положительны, даже когда $\lambda \neq \mu$.

Таким же образом, как расстояние между буквами было обобщено до расстояния между строками, здесь мы используем баллы между буквами для определения баллов между строками. По-прежнему предполагаем, что строка $\boldsymbol{x}_1 = \boldsymbol{x}[1..n_1]$ преобразуется в строку $\boldsymbol{x}_2 = \boldsymbol{x}[1..n_2]$ с помощью операций редактирования отдельных букв: вставки, удаления, подстановки. Также предполагаем, что эти операции необратимы (нельзя выполнить подстановку $a \to b$, затем повторно заменить $b \to a$, даже если, выполняя эти операции, можно увеличить общее количество баллов). Таким образом, преобразование строки $\boldsymbol{x}_1$ в $\boldsymbol{x}_2$ можно представить в виде последовательности операций редактирования, при которых не происходит возврата к предыдущему состоянию — такая же модель ранее использовалась для расстояний.

Конечно, как и прежде, может быть больше одной последовательности операций редактирования, преобразующих $\boldsymbol{x}_1$ в $\boldsymbol{x}_2$. Для каждой такой последовательности можно вычислить сумму всех баллов операций редактирования, как это представлено в матрице баллов. Тогда для строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ определим баллы $S(\boldsymbol{x}_1, \boldsymbol{x}_2)$ как максимальную сумму по всем последовательностям необратимых операций редактирования, которые преобразовывают $\boldsymbol{x}_1$ в $\boldsymbol{x}_2$.

Теперь можно представить аналог выражения (13.13) для вычисления элементов ***массива полезности*** $\boldsymbol{b} = \boldsymbol{b}[1..n_1, 1..n_2]$: для всех $i \in 1..n_1$, $j \in 1..n_2$

$$\boldsymbol{b}[i, j] = \min_{\substack{1 \leqslant i' \leqslant i \\ 1 \leqslant j' \leqslant j}} S(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j]) \tag{13.16}$$

Это выражение определяет элементы массива полезности, которые вычисляемы методом динамического программирования, используя для этого аналог выражения (13.14): для всех $i \in 2..n_1$, $j \in 2..n_2$

$$\begin{aligned} \boldsymbol{b}[i, j] \leftarrow \min\{0, \boldsymbol{b}[i-1, j] + S(\boldsymbol{x}_1[i], \varepsilon), \boldsymbol{b}[i, j-1] + S(\varepsilon, \boldsymbol{x}_2[j]), \\ \boldsymbol{b}[i-1, j-1] + S(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j])\}. \end{aligned} \tag{13.17}$$

Исходные значения для $j = 1$ устанавливаются на основе выражений, аналогичных (10.2) и (10.3). Для удобства предположим, что существует единственное значение $\boldsymbol{b}[0, 1] = 0$, такое, что для всех $i = 1, 2, \ldots, n_1$ можно вычислить

$$\boldsymbol{b}[i, 1] \leftarrow \max\{0, \boldsymbol{b}[i-1, 1] + S(\boldsymbol{x}_1[i], \varepsilon), S(\boldsymbol{x}_1[i], \boldsymbol{x}_2[1])\}. \tag{13.18}$$

Подобным образом для $i = 1$ предполагаем, что $\boldsymbol{b}[1,0] = 0$, и выполняем инициализацию с помощью следующего выражения: для $j = 1, 2, \ldots, n_2$

$$\boldsymbol{b}[1, i] \leftarrow \max\{0, \boldsymbol{b}[1, j-1] + S(\varepsilon, \boldsymbol{x}_2[j]), S(\boldsymbol{x}_1[1], \boldsymbol{x}_2[j])\}. \tag{13.19}$$

Теперь нас интересуют элементы $\boldsymbol{b}[i,j] \geqslant k > 0$, другими словами, те элементы, которые обеспечивают полезность, по меньшей мере равную некоторому положительному порогу k.

Отметим, что выражения (13.17)–(13.19) накладывают на значения $\boldsymbol{b}[i,j]$ условие неотрицательности. Это сделано из-за того, что в (13.16) мы стремимся определить такие i' и j', которые увеличивают до максимума положительное количество баллов; таким образом, в вычислении на основе динамического программирования (13.17) предыдущие значения $\boldsymbol{b}[i-1,j]$, $\boldsymbol{b}[i,j-1]$ и $\boldsymbol{b}[i-1,j-1]$ не будут представлять интереса, если они отрицательные, а для удобства вычислений отрицательные значения целесообразно преобразовать в нулевые. Поэтому сохранение в массиве $\boldsymbol{b}$ только неотрицательных значений гарантирует, что присвоение (13.17) эквивалентно выражению (13.16), и, таким образом, мы избегаем описанных выше сложностей, которые возникли при использовании формулы (13.14).

В табл. 13.2 показан массив полезности для строк из нашего примера, в котором выделены локальные максимумы $\boldsymbol{b}[4,2]$, $\boldsymbol{b}[3,5]$ и $\boldsymbol{b}[5,6]$. Эти значения определяют точные совпадения st и res, так же как и приближенные совпадения $rests$ с $ress$. Значение этих максимумов станет ясным позже, когда будем обсуждать понятие локальной оптимальности.

Таблица 13.2. Массив полезности для $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$, полученный с помощью S и (13.16)

	$\boldsymbol{j}$	**1**	**2**	**3**	**4**	**5**	**6**
i		s	t	r	e	s	s
1	r	0	0	2	1	0	0
2	e	0	0	1	4	3	2
3	s	2	1	0	3	**6**	5
4	t	1	**4**	3	2	5	5
5	s	2	3	3	2	4	**7**

Сеточный граф

В разделе 9.5 был введен граф зависимостей — ориентированный граф, вершины которого помечались элементами $[i,j]$ массива стоимостей $\boldsymbol{c}$ ($i \in 1..n_1, j \in 1..n_2$), а дуги исходили из предшествующих элементов $[i-1,j]$, $[i,j-1]$ и $[i-1,j-1]$, значения которых явились вкладом в значение $\boldsymbol{c}[i,j]$. Эта структура уточнялась формулами (9.8) и показана на рис. 9.2 для $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$.

Здесь мы используем, по существу, ту же структуру, называемую теперь сеточным графом и переопределенную для массива полезности $\boldsymbol{b}$. Для заданных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$ и связанной с ними матрицы баллов S ***сеточный граф*** $G = G(\boldsymbol{x}_1, \boldsymbol{x}_2, S)$

будет ориентированным ациклическим графом (V, E), вершины V которого помечены элементами $[i, j]$ массива полезности $\boldsymbol{b}$, где каждая вершина $[i, j]$ имеет входящие дуги E тогда и только тогда, когда $\boldsymbol{b}[i, j] > 0$, и где применяются следующие правила, аналогичные (9.8).

$$\begin{aligned}
([i-1, j], [i, j]) &\in E, \text{ если } \boldsymbol{b}[i, j] - \boldsymbol{b}[i-1, j] = S(\boldsymbol{x}_1[i], \varepsilon), \\
([i, j-1], [i, j]) &\in E, \text{ если } \boldsymbol{b}[i, j] - \boldsymbol{b}[i, j-1] = S(\varepsilon, \boldsymbol{x}_2[j]), \\
([i-1, j-1], [i, j]) &\in E, \text{ если } \boldsymbol{b}[i, j] - \boldsymbol{b}[i-1, j-1] = S(\boldsymbol{x}_1[i], \boldsymbol{x}_2[j]).
\end{aligned} \tag{13.20}$$

Таким образом, согласно этим правилам, каждая дуга в сеточном графе входит в путь, соответствующий набору операций редактирования, приводящему к максимальному количеству баллов $\boldsymbol{b}[i, j]$. Заметим, что, поскольку вершин $[0, j]$, $[i, 0]$ и $[0, 0]$ в этом графе нет, вершина $[1, 1]$ является источником (не имеет входящих дуг). В общем случае справедлива следующая лемма.

Лемма 13.3.1. Вершина $[i, j]$ в сеточном графе является источником тогда и только тогда, когда целые $i' = i$ и $j' = j$ приводят к максимальному значению $\boldsymbol{b}[i, j]$ в выражении (13.16).

Доказательство предложено дать в упражнении 13.3.4. ■

На рис. 13.7 показан сеточный граф для нашего примера строк $\boldsymbol{x}_1 = rests$ и $\boldsymbol{x}_2 = stress$. Всего в графе имеется 10 вершин-источников, из которых восемь фактически изолированы, — ситуация, которая невозможна в графах зависимостей.

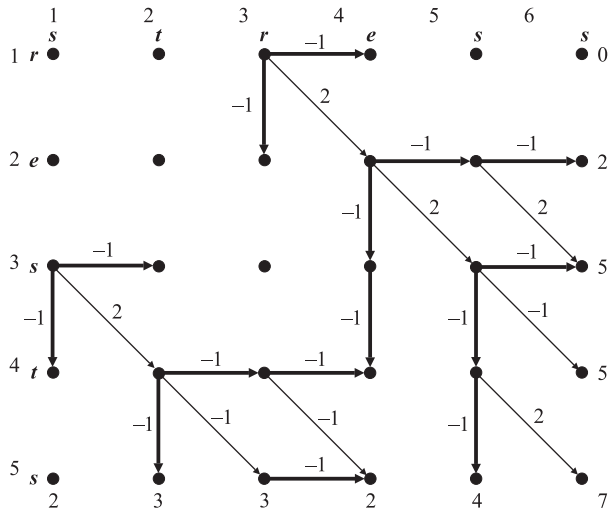


Рис. 13.7. Сеточный граф $G = G(rests, stress, S)$

Как видно на рис. 13.7, сеточный граф G создает структуру, которая показывает, каким образом можно вычислить каждый балл $\boldsymbol{b}[i,j]$: после создания графа G проход по любому пути от вершины $[i',j']$ к вершине $[i,j]$ позволяет легко вычислить величину $\boldsymbol{b}[i,j]$ путем суммирования баллов всех дуг, входящих в путь, и прибавления полученной суммы к величине $\boldsymbol{b}[i,j]$. Назовем эту сумму баллов дуг ***баллом пути*** от $[i',j']$ к $[i,j]$ и обозначаем ее как $S_{ij}^{i'j'}$, т.е. $S_{ij}^{i'j'} = \boldsymbol{b}[i,j] - \boldsymbol{b}[i',j']$.

Таким образом, естественно при каждом выборе $[i,j]$ находить такие вершины-источники $[i',j']$, от которых начинаются пути, ведущие к $[i,j]$. Как показывает следующий результат, для фиксированных i и j эти вершины-источники приводят к максимальным значениям величин $S_{ij}^{i'j'}$.

Лемма 13.3.2. Пусть P обозначает путь в сеточном графе G от вершины-источника $[i',j']$ до некой выбранной вершины $[i,j]$. Тогда величина $S_{ij}^{i'j'}$ является единственным максимальным баллом пути.

Доказательство предложено дать в упражнении 13.3.4. ■

Фактически, на основе формулы (13.16) и этого результата, приходим к вполне удовлетворительному заключению: вершины-источники $[i',j']$ определяют целые числа i' и j', такие, что подстроки $\boldsymbol{x}_1[i'..i]$ и $\boldsymbol{x}_2[j'..j]$ обеспечивают максимальный балл $S(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j])$. Таким образом, вычисление путей с наивысшим баллом в сеточном графе $G = G(\boldsymbol{x}_1, \boldsymbol{x}_2, S)$ является естественным подходом к решению задачи, определяемой формулой (13.16).

Как видно из примера сеточного графа, показанного на рис. 13.7, вершина $[1,3]$ является источником пути с наивысшим баллом, ведущего к каждой из вершин $[i,6]$, $i \in 2..5$. Отсюда, принимая во внимание лемму 13.3.2, заключаем, что суффикс $\boldsymbol{x}_2[3..6] = ress$ строки $\boldsymbol{x}_2$ обеспечивает лучшее совпадение с каждой из подстрок $\boldsymbol{x}_1[1..i]$. С другой стороны, заметим, что вершина $[4,4]$ имеет два источника $[1,3]$ и $[3,1]$. Это говорит о том, что наилучшее совпадение суффиксов подстрок $\boldsymbol{x}_1[1..4] = rest$ и $\boldsymbol{x}_2[1..4] = stre$ можно получить двумя способами: или совпадением $\boldsymbol{x}_1[1..4]$ с $\boldsymbol{x}_2[3..4] = re$, или совпадением $\boldsymbol{x}_1[3..4] = st$ с $\boldsymbol{x}_2[1..4]$.

Принимая во внимание лемму 13.3.2, заманчиво предположить, что максимальный балл пути обязательно достигается как источниками $[i',j']$, так и стоками $[i,j]$. Но это не так — на рис. 13.7 вершина $[3,5]$ лежит на пути от $[1,3]$ до стока $[4,6]$, но тем не менее $\boldsymbol{b}[4,6] = 5 < 6 = \boldsymbol{b}[3,5]$.

В общем случае мы не будем интересоваться всеми путями с высоким баллом, превышающим заданный порог k, а только их подмножеством. Как мы только что видели на рис. 13.7, есть пути с высоким баллом, ведущие и к вершинам $[3,5]$ и $[4,6]$, но путь, ведущий к вершине $[3,5]$, кажется более интересным. Точно так же вершина $[4,2]$, вероятно, будет более интересной, чем $[5,3]$. В следующем под-

разделе рассмотрим важный вопрос, каким образом в сеточных графах определять самые интересные пути с высоким баллом.

Локальная оптимальность

При обсуждении точных раппортов (раздел 13.2) мы увидели, что понятие непродолжаемости играет определяющую роль в отборе выводимых данных, но при этом значительно уменьшает объем вывода. В данном случае для сопоставления с приближенным паттерном понятие локальной оптимальности играет аналогичную роль. И непродолжаемость, и локальная оптимальность зависят от максимального продолжения некоторых свойств как влево (случай *б*) в следующем определении), так и вправо (случай *в*)).

Определение 13.3.3. [84] Пара вершин $[i', j']$, $[i, j]$ в сеточном графе G называется **локально оптимальной**, если выполняются три условия:

а) $S(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j]) > 0$;

б) для всех $I \in 1..i$ и $J \in 1..j$

$$S(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j]) \geqslant S(\boldsymbol{x}_1[I..i], \boldsymbol{x}_2[J..j]) > 0;$$

в) для всех $I \in i'..n_1$ и $J \in j'..n_2$,

$$S(\boldsymbol{x}_1[i'..i], \boldsymbol{x}_2[j'..j]) \geqslant S(\boldsymbol{x}_1[i'..I], \boldsymbol{x}_2[j'..J]).$$

■

Из определения (13.16) массива полезности видно, что условие *б*) будет удовлетворено путем перечисления всех вершин $[i', j']$, которые соответствуют максимуму $\boldsymbol{b}[i, j]$, вычисленному для каждой вершины $[i, j]$. Условие *в*) можно удовлетворить путем вычисления массива полезности $\widehat{\boldsymbol{b}}$ для обратных строк $\widehat{\boldsymbol{x}_1}$ и $\widehat{\boldsymbol{x}_2}$ и перечисления всех вершин $[i', j']$, которые соответствуют каждому максимальному значению $\widehat{\boldsymbol{b}}[i, j]$. Сравнение этих двух списков вершин с учетом преобразований позиций из $\widehat{\boldsymbol{x}_1}$ в $\boldsymbol{x}_1$ и $\widehat{\boldsymbol{x}_2}$ в $\boldsymbol{x}_2$, определяет набор пар вершин, которые удовлетворяют обоим условиям определения 13.3.3.

Для сеточных графов эта процедура означает идентификацию вершин-источников в G, соответствующих каждой вершине $[i, j]$, как было описано выше, и выполнение того же для "обратного" сеточного графа $\widehat{G}$. Требование условия a определения 13.3.3 позволяет избежать рассмотрения изолированных вершин. И чтобы не затруднять ни читателя, ни себя, мы опускаем дальнейшие подробности этой процедуры.

Однако полезно посмотреть, что получается, если понятие локальной оптимальности применить к нашему примеру. В табл. 13.3 показан массив полезности $\widehat{\boldsymbol{b}}$ для обратных строк $stser$ и $sserts$ с выделенными локальными максимумами в $\widehat{\boldsymbol{b}}[3, 6]$ и $\widehat{\boldsymbol{b}}[5, 4]$, которые соответствуют точному совпадению ts и хорошему

совпадению подстрок $stser$ и $sser$. Обращаясь к табл. 13.2, мы видим, что эти совпадения соответствуют точному совпадению st и хорошему совпадению подстрок $rests$ и $ress$, которые были выделены в массиве $\boldsymbol{b}$. Таким образом, эти совпадения могут определяться как локально оптимальные с помощью только что описанной процедуры или с помощью вывода $([1,3],[5,6])$ и $([3,1],[4,2])$, определяющего пары вершин в графе G, либо, в качестве альтернативы, с помощью вывода $([1,1],[5,4])$ и $([2,5],[3,6])$, определяющего пары вершин в графе $\widehat{G}$.

Таблица 13.3. Массив полезности $\widehat{\boldsymbol{b}}$ для $\widehat{\boldsymbol{x}_1} = stser$ и $\widehat{\boldsymbol{x}_2} = sserts$

	$\boldsymbol{j}$	**1**	**2**	**3**	4	5	6
i		s	s	e	r	t	s
1	s	2	2	1	0	0	2
2	t	1	1	0	0	2	1
3	s	2	3	2	1	1	**4**
4	e	1	2	5	4	3	3
5	r	0	1	4	**7**	6	5

Однако отметим отсутствие вывода точного совпадения раппорта ser, выделенного в табл. 13.2, поскольку $\widehat{\boldsymbol{b}}[3,2]$ не соответствует узлу-источнику в обратном сеточном графе $\widehat{G}$ и, следовательно, не удовлетворяет требованию локальной оптимальности. В нашем примере это не является серьезной проблемой: поскольку ser является суффиксом как строки $stser$, так и $sser$, о совпадении которых показывает вывод, то можно сказать, что ser не попал в вывод потому, что он является "продолжаемым влево". Однако могут быть созданы примеры (см. [205]), в которых "хорошие" или "интересные" совпадения пропущены только потому, что, по меньшей мере, один конец совпавших строк не соответствует вершине-источнику в G или $\widehat{G}$.

Таким образом, данное здесь определение локальной оптимальности для приближенного совпадения не совсем обеспечивает все то, что предоставляет понятие непродолжаемости для точного совпадения; тем не менее это самое полезное определение, которым мы располагаем.

Алгоритм со временем выполнения $\Theta(n^3)$

В предыдущих трех подразделах мы представили общую задачу (13.16) вычисления приближенных совпадений, в частности, локально оптимальных совпадений между подстроками двух заданных строк $\boldsymbol{x}_1$ и $\boldsymbol{x}_2$. Теперь обратимся к упомянутому ранее специальному случаю этой задачи, когда $\boldsymbol{x} = \boldsymbol{x}_1 = \boldsymbol{x}_2$. Здесь найденные приближенные совпадения будут подстроками *той же самой* строки $\boldsymbol{x}$ и поэтому будут приближенными квадратами.

"Специальность" рассматриваемого случая позволяет сделать некоторые упрощения. В частности, в результате симметрии, присущей сравнению $\boldsymbol{x}$ с $\boldsymbol{x}$, достаточно вычислить только верхнюю треугольная часть матрицы полезности. Кроме того, поскольку нас не интересуют совпадения $\boldsymbol{x}[i'..i]$ с $\boldsymbol{x}[i'..i]$, все диагональные элементы в $\boldsymbol{b}$ можно положить равными нулю.

Кажется, что алгоритм решения нашей задачи ясен: для заданной строки $\boldsymbol{x} = \boldsymbol{x}[1..n_1]$ вычисляем соответствующую верхнюю треугольную часть матрицы полезности $\boldsymbol{b}$ за время порядка $\Theta(n^2)$, используя для этого упрощенные версии формул (13.17)–(13.19), затем вычисляем $\widehat{\boldsymbol{b}}$, применяя ту же методологию к $\widehat{\boldsymbol{x}}$, и, наконец, идентифицируем локально оптимальные пары, которые определят приближенные квадраты. На вычисления потребуется время порядка $\Theta(n^2)$. Какие здесь могут быть проблемы?

Увы, проблема имеется, и она указана в предыдущем подразделе. Критерий локальной оптимальности является слишком ограничительным: в $\boldsymbol{x}$ могут иметь место приближенные квадраты, которые не обнаруживаются из-за того, что, по меньшей мере, одна из вершин в паре вершин $([i', j'], [i, j])$, которые определяют квадрат, не соответствует источнику в графах G или $\widehat{G}$. В работе [205] приводится пример, в котором обнаруживаются и выводятся различные расщепленные или покрывающиеся квадраты, в то время как более значительный последовательный квадрат пропущен.

В алгоритме Шмидта для преодоления этой трудности просто устраняется вычисление покрывающихся квадратов в целом. Это гарантирует, что будут выводиться все последовательные и расщепленные квадраты. Конечно, в случае точного совпадения, каждый покрывающийся квадрат подстроки $\boldsymbol{u}$ обязательно подразумевает существование последовательного квадрата собственного префикса подстроки $\boldsymbol{u}$, так что, в некотором смысле, теряется немного информации, если не вычисляются покрывающиеся квадраты. Точно так же при приближенном сравнении покрывающиеся квадраты обычно подразумевают более короткие последовательные квадраты.

Поэтому для данной строки $\boldsymbol{x}[1..n]$ в каждой позиции $t \in 1..n-1$ алгоритм Шмидта ведет поиск таких значений $i \in 1..t$ и $i \in t+1..n$, которые дают максимальный балл

$$S(\boldsymbol{x}[i..t], \boldsymbol{x}[t+1..j]). \tag{13.21}$$

В терминах сеточного графа это означает определение всех путей с наивысшим баллом от вершин $[i, t+1]$ в столбце $t+1$ графа G до вершин $[i, t]$ в строке t. Непосредственно величины

$$\min_{\substack{1 \leqslant i \leqslant t \\ t+1 \leqslant j \leqslant n}} S(\boldsymbol{x}[i..t], \boldsymbol{x}[t+1..j]) \tag{13.22}$$

можно вычислить за время порядка $\Theta(n^2)$ для $\Theta(n)$ значений t. Следовательно, для определения всех k-приближенных последовательных квадратов в $\boldsymbol{x}$ потребуется время порядка $\Theta(n^3)$.

Конечно, чтобы определить более интересное подмножество этих квадратов, состоящее только из локально оптимальных квадратов, необходимо снова сформулировать и решить проблему обратного графа $\widehat{G}$ (как было описано выше) так, чтобы можно было определить пути с наивысшим баллом от вершины-источника обратного графа $\widehat{G}$ к вершине-источнику графа G. Таким образом, локально оптимальные k-приближенные последовательные квадраты в $\boldsymbol{x}$ можно вычислить за время порядка $\Theta(n^3)$.

В алгоритме Шмидта предложен способ сокращения времени вычисления до величины порядка $O(n^2 \log n)$, который в общих чертах описан ниже. По существу алгоритм Шмидта разбивает исходную задачу на $n-1$ подзадач, которые являются специальными случаями вычисления формулы (13.16), когда сравниваются строки x_1^t и x_2^t ($t = 1, 2, \ldots, n-1$), где

$$x_1^t = \boldsymbol{x}[1..t], x_2^t = x_2^t[1..n-t] = x[t+1..n] \tag{13.23}$$

В t-й подзадаче P^t вычисляется массив полезности $\boldsymbol{b}^t = \boldsymbol{b}^t[1..t, 1..n-t]$ и соответствующий сеточный граф G^t, в котором определяются максимальные баллы

$$\min_{\substack{1 \leqslant i \leqslant t \\ t+1 \leqslant j \leqslant n}} S(\boldsymbol{x}_1^t[i..t], \boldsymbol{x}_2^t[1..n-t]), \tag{13.24}$$

эквивалентные баллам (13.22). Алгоритм Шмидта решает каждую из этих подзадач на основе массивов, которые обладают специальными свойствами, дающими возможность вначале вычислить их элементы, а затем обеспечить к ним быстрый доступ.

В работе [205] показано, каким образом, не жертвуя ни объемом памяти, ни временем вычисления, этот подход может быть расширен для вычисления не только последовательных, но и расщепленных, локально оптимальных k-приближенных квадратов — однако здесь мы удовлетворимся схемой алгоритма вычисления последовательных квадратов.

Массивы Монжа

Сеточный граф G^t, который соответствует максимальным баллам (13.24), будет содержать пути с наивысшим баллом от вершин-источников $[i, 1]$ в первом столбце до вершин-стоков $[t, j]$ в последней строке, где $i \in 1..t$, $j \in 1..n-t$. В работах [1, 14] предложено ввести в рассмотрение специальные DIST-массивы[1] $\boldsymbol{d}^t = \boldsymbol{d}^t[1..t, 1..n-t]$, где $\boldsymbol{d}^t[i, j]$ — балл пути с наивысшим баллом от вершины $[i, 1]$

[1] Название DIST происходит от DISTance — расстояние, что раскрывает смысл этих массивов. — *Примеч. пер.*

до вершины $[t, j]$ в графе G^t ($t = 1, 2, \ldots, n-1$), и поэтому балл пути совпадает с величиной, определенной формулой (13.24); если не существует ни одного такого пути, тогда предполагается, что $\boldsymbol{d}^t[i, j] = -\infty$. Каждый DIST-массив, будучи вычисленным, содержит всю информацию, необходимую для решения подзадачи P^t (в соответствии с (13.24)), в то время как все $n-1$ DIST-массивов обеспечивают основу для вычисления всех последовательных квадратов в $\boldsymbol{x}$ (в соответствии с (13.21) и (13.22)).

DIST-массивы обладают особым свойством, которое уменьшает суммарное время их вычисления и, кроме того, предоставляет эффективный доступ к вычисленным элементам.

Определение 13.3.4. Матрица $M[1..n_1, 1..n_2]$ называется матрицей **Монжа** (Monge), если выполняется одно из следующих условий:

а) $M[i, j] + M[i+1, j+1] \geqslant M[i, j+1] + M[i+1, j]$ для всех $i \in 1..n_1 - 1$ и $j \in 1..n_2 - 1$;

б) $M[i, j] + M[i+1, j+1] \leqslant M[i, j+1] + M[i+1, j]$ для всех $i \in 1..n_1 - 1$ и $j \in 1..n_2 - 1$. ■

Лемма 13.3.5. Каждый DIST-массив $\boldsymbol{d}^t$ является матрицей Монжа.

Доказательство. Покажем, что массив $\boldsymbol{d}^t$ удовлетворяет условию *б*) определения 13.3.4. Предположим, что для некоторых $i < t$ и $j < n - t \boldsymbol{d}^t[i, j] > 0$ и элемент $\boldsymbol{d}^t[i, j]$ равен баллу пути $\pi_{i,j}$ с наивысшим баллом от вершины $[i, 1]$ до вершины $[t, j]$ в графе G^t. Если, кроме того, $\boldsymbol{d}^t[i+1, j+1] > 0$ и равен баллу пути $\pi_{i+1,j+1}$ с наивысшим баллом от $[i+1, 1]$ до $[t, j+1]$ в G^t, тогда, как показано на рис. 13.8, эти пути должны пересечься в некоторой вершине $[i', j']$ графа G^t.

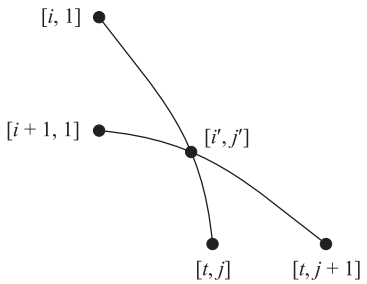


Рис. 13.8. Пересечение путей в сеточном графе G^t

Следовательно, должны существовать пути (не обязательно с наивысшим баллом) $\pi_{i,j+1}$ от вершины $[i, 1]$ до вершины $[t, j+1]$ и $\pi_{i+1,j}$ — от $[i+1, 1]$ до $[t, j]$,

сформированные из подпутей, которые проходят через вершину $[i', j']$. Обозначив как $|\pi|$ балл пути π, мы увидим, что

$$|\pi_{i,j}| + |\pi_{i+1,j+1}| = |\pi_{i,j+1}| + |\pi_{i+1,j}|,$$

где пути в левой части равенства являются по определению путями с наивысшим баллом, в то время как пути в правой части таковыми могут не быть. Этот вытекает из положительности величин $\boldsymbol{d}^t[i,j]$ и $\boldsymbol{d}^t[i+1,j+1]$.

Если хотя бы одна из величин $\boldsymbol{d}^t[i,j]$ и $\boldsymbol{d}^t[i+1,j+1]$ не положительна, то по определению она равна $-\infty$. В этом случае выполняется неравенство

$$\boldsymbol{d}^t[i,j] + \boldsymbol{d}^t[i+1,j+1] = -\infty \leqslant \boldsymbol{d}^t[i,j+1] + \boldsymbol{d}^t[i+1,j].$$

Лемма доказана. ■

Более важным для последующего анализа является аналогичный результат, который связывает значения в массиве $\boldsymbol{d}^t$ со значениями в массиве $\boldsymbol{d}^{t+1}$.

Лемма 13.3.6. Для всех $t \in 1..n-2$, $i \in 1..t$ и $j \in 1..n-t$

$$\boldsymbol{d}^t[i,j] + \boldsymbol{d}^{t+1}[i+1,j] \geqslant \boldsymbol{d}^t[i+1,j] + \boldsymbol{d}^{t+1}[i,j].$$

Доказательство основано на тех же аргументах, которые были использованы при доказательстве леммы 13.3.5. ■

Здесь мы заканчиваем подробное описание алгоритма Шмидта. Лемма 13.3.6 показывает, что мы можем определить массивы $X_j[t,i]$ ($j = 1, 2, \ldots, n-t$), которые являются матрицами Монжа и связывают значения элементов массивов $\boldsymbol{d}^{t+1}$ и $\boldsymbol{d}^t$. Это обеспечивает возможность применения эффективных итерационных процедур, а также эффективный доступ к элементам массивов X_j. Методы, используемые для выполнения этих вычислений, требуют построения и использования совокупности двоичных деревьев, которые "представляют" массивы $\boldsymbol{d}^t$. Объяснение и доказательство корректности этих методов являются длинными, сложными и специализированными, поэтому они не рассматриваются в этой книге. Однако приведем без доказательства два основных результата, касающиеся времени выполнения этих методов (формулировки следующих теорем менее общие по сравнению с оригинальными их версиями в [205]). Первая теорема говорит о вычислении матриц $\boldsymbol{d}^t$.

Теорема 13.3.7. Вычисление DIST-матриц $\boldsymbol{d}^t$ ($t = 1, 2, \ldots, n-1$) может быть выполнено за время порядка $O(n^2 \log n)$ с использованием памяти порядка $O(n^2 \log n)$. Поиск любого отдельного элемента в любом массиве $\boldsymbol{d}^t$ выполняется за время порядка $O(\log t)$, поиск любого элемента в любом столбце любого массива $\boldsymbol{d}^t$ требует времени порядка $O(1)$. ■

Вторая теорема касается последующего вычисления максимумов строк и столбцов в матрицах $\boldsymbol{d}^t$; эти максимумы необходимы алгоритму Шмидта для вычисления путей с наивысшим баллом.

Теорема 13.3.8. Все максимумы строк и столбцов в DIST-матрицах $\boldsymbol{d}^t$ ($t = 1, 2, \ldots, n-1$) могут быть вычислены за время порядка $O(n^2 \log n)$ с использованием памяти порядка $O(n^2)$. ■

Модификация алгоритма Мейна–Лоренца

В этом разделе, избегая излишних подробностей, еще раз возвратимся к первоначальной формулировке задачи приближенных квадратов (13.22) и покажем, каким образом методику алгоритма Мейна–Лоренца определения точных квадратов (подраздел 12.1.2) можно применить в алгоритме Шмидта для определения приближенных квадратов.

Напомним, что алгоритм Мейна–Лоренца находит квадраты в любой подстроке $\boldsymbol{u}$ строки $\boldsymbol{x}$, сначала расщепляя $\boldsymbol{u}$ на левую часть $\boldsymbol{u}_1 = \boldsymbol{u}[1..\lfloor n/2 \rfloor]$ и на правую часть $\boldsymbol{u}_2 = \boldsymbol{u}[\lfloor n/2 \rfloor + 1..n]$ и вычисляя квадраты, покрывающие $\boldsymbol{u}_1$ и $\boldsymbol{u}_2$, затем решая рекурсивные подзадачи для квадратов, распложенных или в $\boldsymbol{u}_1$, или в $\boldsymbol{u}_2$. Алгоритм Шмидта определяет пути с наивысшим баллом от вершин столбца $t+1$ сеточного графа G к вершинам строки t. Далее вспомним, что поскольку наша проблема рефлексивна (мы сравниваем подстроки строки $\boldsymbol{x}$ с ее же подстроками), поэтому необходимо рассмотреть только верхнюю треугольную часть как массива полезности $\boldsymbol{b}$, так и сеточного графа G. Таким образом, как показано на рис. 13.9, можно применить стратегию алгоритма Мейна–Лоренца для расщепления графа G на подграфы на основе положения путей с наивысшим баллом. Пути, которые пересекаются в "средней точке", вычисляются сразу. Опишем другие возможные пути:

а) непосредственно вычисляются и выводятся приближенные квадраты, соответствующие путям с наивысшим баллом не менее k, которые пересекают или столбец $\lfloor n/2 \rfloor + 1$, или строку $\lfloor n/2 \rfloor$;

б) обрабатываются как рекурсивные подзадачи пути с наивысшим баллом, которые ограничиваются или верхним левым треугольником, или нижним правым треугольником;

в) пути с наивысшим баллом, которые ограничиваются ближним квадратом, определяемым неравенствами $0 \leqslant i \leqslant \lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor + 1 \leqslant j \leqslant n$, не могут быть решениями, поскольку нет такого значения t в этой области, которая удовлетворяет требованиям задачи. Таким образом, эта часть графа G может быть отброшена.

Как указано в теоремах 13.3.7 и 13.3.8, пути с наивысшим баллом типа a можно вычислить за время порядка $O(n^2 \log n)$. Таким образом, обозначив как t_n максимальное время, необходимое алгоритму Шмидта для вычисления приближенных

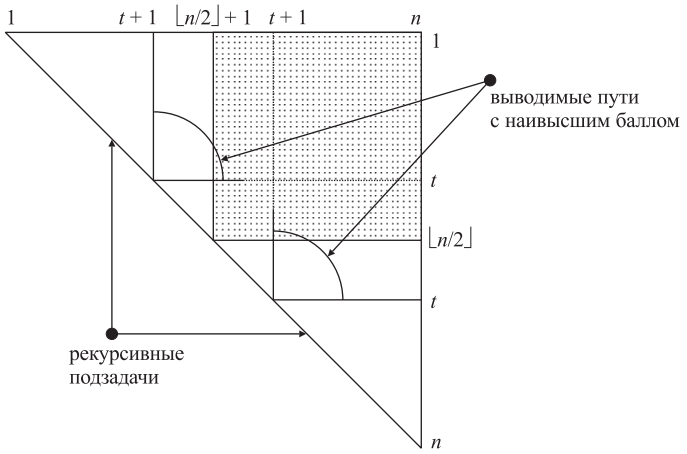


Рис. 13.9. Рекурсивное расщепление сеточного графа в соответствии с алгоритмом МЛ

последовательных квадратов в любой строке $\boldsymbol{x}[1..n_1]$, с учетом вычисления путей типа *б*) и *в*), величину t_n можно определить путем решения рекуррентного соотношения

$$t_n \leqslant k_2 n^2 \log n + 2t_{\lceil n/2 \rceil}, \tag{13.25}$$

где k_2 — положительная константа. Поскольку существует положительная константа k_1, такая, что $t_1 \leqslant k_1$, последняя задача является аналогом рекуррентной задачи (12.2) для алгоритма Мейна–Лоренца. Внимательный читатель может вспомнить решение несколько более общей версии соотношения (13.25) из упражнения 12.1.6, на основании которого можно утверждать, что временна́я сложность алгоритма Шмидта составляет $t_n \in O(n^2 \log n)$.

Детали алгоритма Шмидта, включая незатронутые здесь аспекты, можно найти в работе [205]. Сейчас, на основе теоремы 13.3.7 и предыдущего анализа, сформулируем конечный результат.

Теорема 13.3.9. Алгоритм Шмидта вычисляет локально оптимальные k-приближенные непокрывающиеся квадраты в заданной строке $\boldsymbol{x}[1..n]$ за время порядка $O(n^2 \log n)$ с использованием памяти такого же порядка. ■

Обсуждение

В этом разделе мы постарались представить понятную схему алгоритма Шмидта, не слишком вдаваясь в его технические подробности. Мы выбрали для обсуждения этот алгоритм, потому что он самый быстрый и пригоден для решения проблем, представляющих практический интерес, т.е. алгоритм Шмидта является и полезным, и поучительным. Более раннему алгоритму [111] для вычисления

всех (не только локально оптимальных) непокрывающихся приближенных квадратов требовалось время порядка $O(n^2 \log^2 n)$ и объем памяти порядка $O(n^2 \log n)$; в работе [30] требуемый объем памяти был уменьшен до величины порядка $O(n^2)$. Алгоритм для вычисления всех "хороших" приближенных последовательных раппортов за время порядка $O(kn \log k \log(n/k))$ предложен в [145], но его можно использовать только для расстояния преобразования.

Одной из самых привлекательных особенностей алгоритма Шмидта является критерий локальной оптимальности, применяемый для ограничения всех раппортов до приближенных квадратов, представляющих наибольший интерес. Однако следует отметить, что этот критерий ставит проблемы, которые не присущи критериям непродолжаемости, используемым для точных раппортов. Возможно, переопределение понятия локальной оптимальности может помочь в поиске более удовлетворительного алгоритма для решения задачи приближенных квадратов.

Упражнения 13.3

1. Используя расстояние преобразования, создайте пример такой строки $\boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_3$, где $\boldsymbol{u}_1\boldsymbol{u}_2$ и $\boldsymbol{u}_2\boldsymbol{u}_3$ являются 1-приближенными квадратами, в то время как $\boldsymbol{u}_1\boldsymbol{u}_3$ таковым не является. Затем обобщите этот пример для произвольного $k > 1$. Можно ли привести такой пример, где все эти строки имеют различную длину?
2. Предположим, что разрешаются обратимые операции редактирования. Создайте пример, в котором значение $S(\boldsymbol{x}_1, \boldsymbol{x}_2)$ становится сколь угодно большим.
3. В этом разделе был описан массив полезности из неотрицательных значений, базирующийся на положительных баллах для совпадений и, в основном, на отрицательных баллах для несовпадений. Покажите, что массив стоимостей, состоящий из неположительных значений, будет работать точно так же, и введите необходимые определения.
4. Докажите леммы 13.3.1 и 13.3.2.
5. Примеры, данные в этом разделе, имеют вершины-источники $[i', j']$ для путей только при $i' = 1$ и $j' = 1$. Приведите примеры, где вершины-источники находились бы в произвольном заданном месте в сеточном графе.
6. Нарисуйте сеточный граф $\widehat{G}$, соответствующий строкам $\widehat{\boldsymbol{x}_1} = stser$ и $\widehat{\boldsymbol{x}_2} = sserts$. Затем используйте его для определения всех локально оптимальных пар в графе G, соответствующем строкам $rests$ и $stress$.
7. Покажите, что при точном совпадении каждый покрывающийся квадрат подстроки $\boldsymbol{u}$ определяет последовательный квадрат префикса $\boldsymbol{u}$. Можно ли создать пример для приближенного сравнения, в котором это не выполняется?

8. Сформулируйте задачу, обратную (13.22), и объясните, каким образом ее можно использовать для вычисления локально оптимальных k-приближенных последовательных квадратов за время порядка $\Theta(n^3)$.

9. Докажите лемму 13.3.6.

13.4 k-приближенные периоды — алгоритмы Сима–Илиопулоса–Парка–Смита

Первый алгоритм, описанный в этой книге, вычислял массив граней заданной строки $\boldsymbol{x}$, определяя за линейное время каждый период, включая самый короткий, каждого префикса $\boldsymbol{x}$. В этом заключительном разделе книги мы завершаем почти полный цикл. Теперь попробуем вычислить *приближенные* периоды строки $\boldsymbol{x}$, что является более трудной задачей [209], особенно в тех случаях, когда необходимо, чтобы вычисленные элементы включали нечто, в некотором смысле, подобное самому короткому точному периоду, "наилучшему" или наиболее "содержательному". Мы увидим, что проблемы, рассматриваемые в этом разделе, действительно находятся на грани познания: можно сформулировать несколько интересных проблем, но в настоящее время даже наиболее простые из них могут быть решены только частично.

Однако прежде чем вплотную заняться этими проблемами, необходимо ответить на следующий вопрос.

> Что мы вкладываем в понятие "приближенный период" (или длина "приближенной образующей"), если приближенная образующая не определяется длиной периода?

Рассмотрим в качестве примера простую строку

$$\boldsymbol{x} = abacbc \tag{13.26}$$

Предположим, что необходимо найти для расстояния преобразования d_E 1-приближенную образующую строки $\boldsymbol{x}$, т.е. строку $\boldsymbol{u}$, такую, что для некоторой декомпозиции $\boldsymbol{x} = \boldsymbol{x}_1\boldsymbol{x}_2 \ldots \boldsymbol{x}_r$ ($r \geqslant 1$) строки $\boldsymbol{x}$ выполняются неравенства $d_E(\boldsymbol{u}, \boldsymbol{x}_j) \leqslant 1$ для всех $j \in 1..r$.

Для нашей строки (да и для любой другой), несомненно, "лучшее" решение этой задачи состоит в том, чтобы выбрать $\boldsymbol{u} = \boldsymbol{x}$ и $r = 1$, тогда $d_E(\boldsymbol{u}, \boldsymbol{x}) = d_E(\boldsymbol{u}, \boldsymbol{x}_1) = 0$. Если это решение исключить, то для нашей строки (как и для любой другой) можно выбрать $\boldsymbol{u} = \lambda$ (λ — произвольная буква), тогда $d_E(\boldsymbol{u}, \boldsymbol{x}[i]) = d_E(\lambda, \boldsymbol{x}[i]) = 1$ для всех позиций i в $\boldsymbol{x}$. Продолжая этот процесс, обнаруживаем, что, по меньшей мере для расстояния преобразования, k-приближенные образующие любой строки $\boldsymbol{x}$ всегда будут включать саму строку $\boldsymbol{x}$, так же как все

строки $\boldsymbol{u}$ длиной больше k. Это наблюдение вряд ли можно назвать полезным или содержательным результатом.

Заметим далее, что в нашем примере (13.26) можно также выбрать $\boldsymbol{u} = ac$ и выполнить декомпозицию $\boldsymbol{x} = \boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3 = (ab)(ac)(bc)$, тогда $d_E(\boldsymbol{u}, \boldsymbol{x}_j) = d_E(ac, \boldsymbol{x}_j) = 1$ для каждого $j \in 1..3$. Таким образом, ac — это 1-приближенная образующая, а не префикс строки (13.26). Или, если мы несколько изменим наш пример, взяв строку $\boldsymbol{y} = abbcad$, то здесь (хотя у строки $\boldsymbol{y}$ нет подходящей подстроки $\boldsymbol{u}$ длиной 2, претендующей на роль 1-приближенной образующей строки $\boldsymbol{y}$) снова можно выбрать строку $\boldsymbol{u} = ac$ (не являющейся подстрокой строки $\boldsymbol{y}$), и тогда, согласно декомпозиции $\boldsymbol{y} = \boldsymbol{y}_1\boldsymbol{y}_2\boldsymbol{y}_3 = (ab)(bc)(ad)$, $d_E(\boldsymbol{u}, \boldsymbol{y}_j) = d_E(ac, \boldsymbol{y}_j) = 1$ для каждого $j \in 1..3$. Таким образом, ac также будет 1-приближенной образующей строки $\boldsymbol{y}$.

В обоих этих примерах знание длины возможной образующей дает немного информации: что действительно необходимо знать — так это непосредственно саму образующую. Следовательно, наш первый шаг в формулировании проблемы k-приближенных периодов заключается в устранении любых ссылок на периоды! В этом разделе мы будем иметь дело с *приближенными образующими* заданной строки $\boldsymbol{x}$.

В предыдущем разделе мы видели, что для того, чтобы понятие приближенного раппорта имело смысл, было необходимо тщательно продумать способ определения понятия "расстояние" — здесь мы также видим, что соответствующее определение расстояния является критическим предусловием для того, чтобы оперировать приближенными образующими. Далее в этом разделе будем использовать понятие ***взвешенного расстояния*** d_W (обобщение расстояния преобразования), где используется матрица весов W с неотрицательными значениями, как определено в разделе 2.3. При изменении значений матрицы W, d_W может преобразовываться в другие обычные виды расстояния: d_E, d_L и d_H. Следуя [209], вводим новое полезное понятие.

Определение 13.4.1. Для двух произвольных непустых строк $\boldsymbol{x}$ и $\boldsymbol{y}$, определенных на алфавите A, и данной функции расстояния d_W, определенной на буквах алфавита A, значение

$$\delta_W(\boldsymbol{x}, \boldsymbol{y}) = \frac{2d_W(\boldsymbol{x}, \boldsymbol{y})}{|\boldsymbol{x}| + |\boldsymbol{y}|}$$

называется **относительным расстоянием** между $\boldsymbol{x}$ и $\boldsymbol{y}$. ■

Таким образом, относительное расстояние между двумя строками — это взвешенное расстояние (или ***абсолютное расстояние***), деленное на среднюю длину строк. Расстояние δ_W сохраняет свойства метрики, если таковыми обладает d_W (см. упражнение 13.4.1). Но более важно заметить, что использование относительного расстояния (а не взвешенного) при вычислении расстояния между потенциальными периодами $\boldsymbol{u}$ и подстроками $\boldsymbol{x}_j$ строки $\boldsymbol{x}$ в значительной степени

изменяет полученные результаты. В нашем предыдущем примере (13.26) находим, что для строк длиной 1 значения всех видов расстояния неизменны.

$$\delta_E(\lambda,\mu) = d_E(\lambda,\mu) = 1, \text{ если } \lambda \neq \mu, \text{ иначе } \delta_E(\lambda,\mu) = d_E(\lambda,\mu) = 0. \quad (13.27)$$

Однако расстояния для некоторых строк длиной 2 уменьшаются:

$$\delta_E(ac,ab) = \delta_E(ac,bc) = \delta_E(ac,ad) = 1/2.$$

Таким образом, если бы мы искали образующую на основе относительного расстояния, вероятно, выбрали бы строку, такую, как ac, на которой достигается меньшее значение или абсолютного, или относительного расстояний, чем это может быть достигнуто на одной букве. Этот пример снова поднимает вопрос о том, какие критерии должны использоваться для выбора образующих. Кроме того, отметим, что выбор $\boldsymbol{u} = \boldsymbol{x}$ все еще приводит к результату $\delta_E(\boldsymbol{u},\boldsymbol{x}) = 0$, и поэтому также необходимо "придумать" критерий, который ограничивал бы длину $\boldsymbol{u}$.

Для удовлетворения этих требований дадим следующее определение приближенной образующей, немного изменив определение из [209].

Определение 13.4.2. Для вещественного числа $k \geqslant 0$ и целого числа $\rho \geqslant 1$, строка $\boldsymbol{u}$ будет (ρ,k)-**приближенной образующей** строки $\boldsymbol{x}$, если существует декомпозиция $\boldsymbol{x} = \boldsymbol{x}_1\boldsymbol{x}_2\ldots\boldsymbol{x}_r$ ($r \geqslant \rho$) на непустые факторы $\boldsymbol{x}_j$ ($j \in 1..r$), такие, что $\delta_W(\boldsymbol{u},\boldsymbol{x}_j) \leqslant k$ для всех $j \in 1..r-1$, и при этом $\delta_W(\boldsymbol{u}',\boldsymbol{x}_r) \leqslant k$ для некоторого непустого префикса $\boldsymbol{u}'$ строки $\boldsymbol{u}$. ■

В "настойчивом" требовании этого определения, что $\boldsymbol{u}$ должна быть достаточно близка к *каждому* фактору $\boldsymbol{x}_j$, фиксируется свойство "равномерности", которому удовлетворяет точная образующая. Отметим, что это определение разрешает строке $\boldsymbol{x}$ иметь суффикс, который приближенно совпадает с префиксом строки $\boldsymbol{u}$.

Чтобы узнать, какое влияние оказывает выбор параметра ρ на определение образующей, привлечем свою интуицию и рассмотрим пример строки $\boldsymbol{x} = abcacaabaacaa$. В табл. 13.4 приведены приближенные образующие, удовлетворяющие различным значениям параметров ρ и k. В этом примере, чтобы получить "интересные" приближенные образующие, следует выбрать $\rho \geqslant 3$, но не ясно, будет ли какая-либо польза при $\rho = 4$, поскольку при этом значении ρ исключается возможная интересная образующая $abcaca$. С другой стороны, могут быть такие случаи, где, например, $\boldsymbol{x}$ имеет длинную точную или приближенную образующую, повторенную несколько раз, — тогда желательно большее значение ρ. Отметим, что из этого примера также не ясно, будут ли образующие, соответствующие минимальному относительному расстоянию k (независимо от значения ρ), обязательно наиболее интересными!

Из определения 13.4.2 прямо вытекает определение понятия "минимальной" приближенной образующей.

Таблица 13.4. (ρ, k)-приближенные образующие для $\boldsymbol{x} = abcacaabaacaa$

ρ	k	Найденные образующие
1	0	$\boldsymbol{x}, abcacaabaaca$
2	0	$abcacaabaaca$
3	1/3	$aba, abcaca, abaaca$
3	1/6	$abcaca, abaaca$
4	1/3	aba
4	1/6	—

Определение 13.4.3. (ρ, k)-приближенная образующая $\boldsymbol{u}$ строки $\boldsymbol{x}$ называется ρ-**минимальной**, если не существует (ρ, k')-приближенной образующей строки $\boldsymbol{x}$, где $k' < k$. ■

Отметим, что в табл. 13.4 $abcaca$ и $abaaca$ являются 3-минимальными образующими, в то время как aba — 4-минимальная.

Теперь можно сформулировать три задачи, связанные с (ρ, k)-приближенными образующими [209], которые перечислим в порядке возрастания их сложности.

P1. Заданы непустые строки $\boldsymbol{x}$ и $\boldsymbol{u}$, целое число ρ и функция относительного расстояния δ_W. Требуется найти такое минимальное значение k, при котором $\boldsymbol{u}$ будет (ρ, k)-приближенной образующей строки $\boldsymbol{x}$.

P2. Заданы непустая строка $\boldsymbol{x}$, целое число ρ и функция относительного расстояния δ_W. Требуется найти подстроку $\boldsymbol{u}$ строки $\boldsymbol{x}$ (если она существует), которая является ρ-минимальной образующей строки $\boldsymbol{x}$.

P3. Заданы непустая строка $\boldsymbol{x}$, целое число ρ и функция относительного расстояния δ_W. Требуется найти произвольную строку $\boldsymbol{u}$ (если она существует), которая является ρ-минимальной образующей строки $\boldsymbol{x}$.

В следующих трех подразделах мы вкратце обсудим эти три задачи только для того, чтобы увидеть, что в настоящее время ни для одной из них не существует полного решения! Однако мы увидим, что для задач P1 и P2 имеются частичные решения, полученные путем непосредственного применения алгоритмов, рассмотренных в главе 9.

Задача P1 — вычисление минимального k

Для единообразия мы сформулировали эту задачу для функции относительного расстояния δ_W, но поскольку и $\boldsymbol{x}$ и $\boldsymbol{u}$ определены, ее можно точно также сформулировать для функции абсолютного расстояния d_W.

Описанный в этом подразделе первый алгоритм Сима–Илиопулоса–Парка–Смита (Sim–Iliopoulos–Park–Smyth, сокращенно — алгоритм СИПС1) решает зада-

чу P1 только для *некоторого* значения ρ, не заданного заранее. Другими словами, он вычисляет такое минимальное значение k, что для некоторого ρ $\boldsymbol{u}$ является (ρ, k)-приближенной образующей строки $\boldsymbol{x}$. Выполнение алгоритма СИПС1 делится на два этапа. На первом этапе применяется повторяемое вычисление расстояний между строками, на втором — итерационно вычисляется минимальное пороговое значение k.

На первом этапе вычисляются n стандартных массивов стоимостей (раздел 9.1)

$$\boldsymbol{c}^{(i)}[0..n-i+1, 0..m], i = 1, 2, \ldots, n, \tag{13.28}$$

в соответствии с расстояниями $\delta_W(\boldsymbol{x}[i..n], \boldsymbol{u})$. Эти вычисления определяют расстояния $\delta_W(\boldsymbol{x}[i..h], \boldsymbol{u})$ между $\boldsymbol{x}[i..h]$ и $\boldsymbol{u}$ для каждого $h \in i..n-1$; эти расстояния представлены в m-х столбцах массивов $\boldsymbol{c}^{(i)}$, $i = 1, 2, \ldots, n-1$. Поскольку конечное вхождение $\boldsymbol{u}$ в $\boldsymbol{x}$ должно быть только непустым префиксом $\boldsymbol{u}$, мы можем искать $(n-i+1)$-ю (последнюю) строку каждого массива $\boldsymbol{c}^{(i)}$ для минимального значения, назовем его $\mathrm{minpref}(\boldsymbol{u}, i, n)$ — это будет наименьшим расстоянием между любым префиксом $\boldsymbol{u}$ и $\boldsymbol{x}[i..n]$. Таким образом, используя последний столбец и один элемент из последней строки каждого массива $\boldsymbol{c}^{(i)}$, можно сформировать верхний треугольный массив стоимостей $\boldsymbol{c}[1..n, 1..n]$, где для $i \in 1..n$ $\boldsymbol{c}[i, h] = \delta_W(\boldsymbol{x}[i..h], \boldsymbol{u})$, если $h \in i..n-1$, и $\boldsymbol{c}[i, h] = \mathrm{minpref}(\boldsymbol{u}, i, n)$, если $h = n$.

Вычисление массивов $\boldsymbol{c}^{(i)}$ с использованием алгоритма Вагнера–Фишера (раздел 9.2) требует времени порядка $\Theta(mn^2)$, вычисление массива $\boldsymbol{c}$ требует времени порядка $\Theta(n^2)$ — следовательно, общее время выполнения первого этапа алгоритма СИПС1 должно иметь порядок $\Theta(mn^2)$.

Теперь рассмотрим второй этап алгоритма. Предположим, что $\boldsymbol{u}$ является приближенной образующей строки $\boldsymbol{x}[1..i]$. Разобьем $\boldsymbol{x}[1..i]$ на факторы, у которых наибольшее расстояние (вычисляемое как δ_W) до $\boldsymbol{u}$ равно, скажем, k_i. Тогда для каждого $h \in i+1..n-1$ можно определить соответствующее расстояние k_h путем вычисления

$$\max\{k_i, \delta_W(\boldsymbol{x}[i+1..h], \boldsymbol{u})\} = \max\{k_i, \boldsymbol{c}[i+1, h]\},$$

при этом для $h = n$ вычисляется

$$\max\{k_i, \mathrm{minpref}(\boldsymbol{u}, i, n)\} = \max\{k_i, \boldsymbol{c}[i+1, h]\}.$$

Если, кроме того, для каждого $i \in 1..h-1$ факторы подстроки $\boldsymbol{x}[1..i]$ выбраны таким способом, чтобы минимизировать k_i по всем выборам факторов, то мы видим, что вычисление

$$k_h = \min_{1 \leqslant i \leqslant h} \{\max\{k_i, \boldsymbol{c}[i+1, h]\}\} \tag{13.29}$$

должно приводить к минимальному расстоянию k_h для $\boldsymbol{x}[1..h]$ по всем выборам факторов. Это классический подход динамического программирования, который можно реализовать за время порядка $\Theta(n^2)$. Следовательно, для двух этапов алгоритма СИПС1 справедлива теорема.

Теорема 13.4.4. Для данной функции расстояния δ_W и непустых строк $\boldsymbol{x}$ и $\boldsymbol{u}$ алгоритм СИПС1 вычисляет минимальное значение k, такое, что для некоторого ρ $\boldsymbol{u}$ является (ρ, k)-приближенной образующей строки $\boldsymbol{x}$. Для выполнения этого алгоритма требуется время порядка $\Theta(mn^2)$ и память порядка $\Theta(n^2)$. ■

Ввиду явного подобия между первым этапом алгоритма СИПС1 и вычислением массивов полезности $\boldsymbol{b}^t$ алгоритма Шмидта, кажется вполне вероятным, что здесь также можно использовать DIST-массивы для уменьшения времени порядка $\Theta(mn^2)$, необходимого для выполнения алгоритма на первом этапе, до времени порядка $O(mn \log n)$, следовательно, для уменьшения полного времени до величины порядка $\Theta(n^2)$. Эта увлекательная возможность оставлена в качестве исследовательского проекта упражнения 13.4.4.

Пока мы предполагали, что функция расстояния δ_W является обобщенной, основанной на произвольной матрице баллов. Однако если вместо расстояния преобразования используются расстояния δ_E или d_E, то результатом будет существенная экономия времени и памяти. Принимая во внимание (13.27), не обязательно вычислять расстояние преобразования между $\boldsymbol{u}$ и подстроками $\boldsymbol{x}$ длиной больше $2m$: такие расстояния обязательно превысили бы значение m и поэтому не могли бы способствовать вычислению минимального значения k_i. Таким образом, каждый из n массивов стоимостей (13.28) будет иметь максимальный размер $2m \times m$, в то время как пересмотренный массив стоимостей $\boldsymbol{c}$ будет иметь размер $n \times 2m$. Время, необходимое для выполнения первого этапа, уменьшается, соответственно, до величины порядка $\Theta(m^2 n)$, а на втором этапе нужно рассмотреть не более $2m$ элементов массива $\boldsymbol{c}$ для каждого значения $h = 1, 2, \ldots, n$, что приведет ко времени порядка $O(mn)$. Следовательно, использование расстояния преобразования уменьшает полное время, необходимое для выполнения алгоритма СИПС1, до $\Theta(m^2 n)$, а необходимый объем памяти — до $\Theta(mn)$.

Но другой подход, реализованный в алгоритме СИПС2, уменьшает это время еще в большей степени. Вспомним, что в конце раздела 9.5, была сделана ссылка на "пошаговый" алгоритм вычисления расстояний между строками [144], который для данного расстояния $d_E(\boldsymbol{x}, \boldsymbol{u})$ может вычислить любое расстояние $d_E(\lambda\boldsymbol{x}, \boldsymbol{u})$ и $d_E(\boldsymbol{x}\lambda, \boldsymbol{u})$ для любой буквы λ за время $O(|\boldsymbol{x}| + |\boldsymbol{u}|)$. Теперь предположим, что такой алгоритм применен для вычисления расстояния преобразования между $\boldsymbol{u}$ и следующей последовательностью $n + 1$ строк; каждая длиной не более $2m$ сформирована из предыдущей путем присоединения слева одной буквы:

$$\varepsilon, \boldsymbol{x}[n], \boldsymbol{x}[n-1..n], \ldots, \boldsymbol{x}[n-2m+1..n], \boldsymbol{x}[n-2m..n-1], \ldots, \boldsymbol{x}[1..2m].$$

Эти вычисления делают доступными все расстояния между $\boldsymbol{u}$ и каждой подстрокой $\boldsymbol{x}$ длиной не более $2m$. Поскольку $|\boldsymbol{u}| = m$, время, требуемое этому алгоритму для каждого вычисления, будет равно $O(2m+m) = O(m)$, следовательно, полное время имеет порядок $O(mn)$. Далее матрица $\boldsymbol{c}$ формируется, как и в алгоритме СИПС1, за время порядка $\Theta(mn)$, затем на втором этапе обрабатывается за время порядка $\Theta(mn)$. Таким образом, имеем следующую теорему.

Теорема 13.4.5. Используя расстояние преобразования и данные непустые строки $\boldsymbol{x}$ и $\boldsymbol{u}$, алгоритм СИПС2 вычисляет минимальное значение k, такое, что для некоторого ρ $\boldsymbol{u}$ является (ρ, k)-приближенной образующей строки $\boldsymbol{x}$. На его выполнение требуется время порядка $\Theta(mn)$ и память объемом порядка $\Theta(m^2)$. ■

Задача P2 — вычисление ρ-минимальной подстроки $\boldsymbol{u}$

Один из очевидных способов определить подстроку $\boldsymbol{u}$ строковой последовательности $\boldsymbol{x}$ (для некоторого ρ, не выбранного заранее) как ρ--минимальную образующую строки $\boldsymbol{x}$ состоит в исследовании всех возможных подстрок строки $\boldsymbol{x}$ путем использования алгоритмов СИПС1 или СИПС2 и в последующем выборе подстроки $\boldsymbol{u}$, который приводит к минимальному пороговому значению k в качестве периода. Напомним из упражнения 1.2.10, что в строковой последовательности $\boldsymbol{x}$ имеется $O(n^2)$ различных подстрок средней длины $O(n)$. Таким образом, или алгоритм СИПС1 (для общего расстояния δ_W), или СИПС2 (для расстояния преобразования δ_E) может быть выполнен $O(n^2)$ раз с использованием образующей, средняя длина которой $m \in O(n)$. Следовательно, имеет место такой результат.

Теорема 13.4.6. Для некоторой, заранее не определенной величины ρ задача P2 может быть решена за время порядка $O(n^5)$ с использованием расстояния δ_W и за время порядка $O(n^4)$ с использованием расстояния преобразования δ_E. ■

В работе [209] для случая расстояния преобразования приводится другой метод решения задачи P2, но со временем выполнения все еще порядка $O(n^4)$.

Задача P3 — вычисление произвольной ρ-минимальной строки $\boldsymbol{u}$

Эта задачу мы особенно хотели бы решить с приемлемой эффективностью, но, как оказывается, в такой формулировке эта задача является NP-полной[2]! Для получения более подробной информации см. [209].

Замечание

В этом разделе мы выделили алгоритмы, которые касаются новой области анализа строковых последовательностей, где проблемы еще недостаточно хорошо

[2]NP-полной называется задача, которая эквивалентна задаче полного перебора всех возможных вариантов ее решения. — *Примеч. ред.*

определены, а решений еще меньше. За курс из 13 глав мы достигли этого пункта длинным, но, тем не менее, по структуре довольно естественным и понятным маршрутом. Однако остается тот факт, что проблемы, только что рассмотренные, находятся среди наиболее простых обобщений вычисления периодичности, алгоритм которого первым рассматривался в этой книге. Маршрут был длинным, а покрытое расстояние коротким.

Упражнения 13.4

1. Покажите, что δ_W удовлетворяет условиям (2.2)–(2.5) метрики тогда и только тогда, когда d_W удовлетворяет этим же условиям.
2. Покажите для расстояния преобразования, что в общем случае $0 \leqslant \leqslant \delta_E(\boldsymbol{x}, \boldsymbol{y}) < 2$, а в случае $|\boldsymbol{x}| = |\boldsymbol{y}|$ $\delta_E(\boldsymbol{x}, \boldsymbol{y}) \leqslant 1$.
3. Объясните, почему в алгоритме СИПС1 необходимый объем памяти достигает величины порядка $\Theta(n^2)$, а не порядка $\Theta(mn^2)$.
4. **Исследовательский проект**. Исследуйте возможность использования DIST-массивов (раздел 13.3) для уменьшения времени выполнения алгоритма СИПС1 до времени порядка $\Theta(n^2)$.
5. Разработайте вариант алгоритма СИПС1, который для расстояния Хемминга d_H решает задачу P1 за время порядка $\Theta(n)$.

Литература

[1] *Aggarwal Alok, Park James*. Notes on searching in multidimensional monotone arrays // Proc. 29th Annual IEEE Symp. Foundations of Computer Science. — 1988. — P. 497–512.

[2] *Aho Alfred V.* Algorithms for finding patterns in strings // Handbook of Theoretical Computer Science. J. van Leeuwen (ed.). Elsevler. — 1990. — P. 255–300.

[3] *Aho Alfred V., Corasick Margaret J.* Efficient string matching: an aid to bibliographic search // *CACM, 18–6*, 1975. — P. 333–340.

[4] *Aho Alfred V., Hirschberg Dan S., Ullman Jeffrey D.* Bounds on the complexity of the longest common subsequence problem // J. ACM, 23–1, 1976. — P. 1–12.

[5] *Aho Alfred V., Hopcroft John E., Ullman Jeffey D.* The Design & Analysis of Computer Algorithms // Addison-Wesley. — 1974. (Русский перевод: *Ахо А., Хоркрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. — Издательский дом "Вильямс", 2000.)

[6] *Aho Alfred V., Sethi Ravi, Ullman Jeffrey D.* Compilers: Principles, Techniques & Tools // Addison-Wesley. — 1986. (Русский перевод: *Ахо А.В., Рэви С., Ульман Дж.Д.* Компиляторы: принципы, технологии и инструментарий. — Издательский дом "Вильямс", 2001.)

[7] *Allison Lloyd and Dix Trevor I.* A bit string longest subsequence algorithm // *IPL, 23–6*, 1986. — P. 305–310.

[8] *Allouche Jean-Paul, Astoorian Dan, Randall Jim and Shallit Jeffrey.* Morphisms, square-free strings and the Tower of Hanoi puzzle // Amer. Math. Monthly, 101–7, 1994. — P. 651–658.

[9] *Allouche Jean-Paul.* Sur la complexite des suites infinies // Bull. Belg. Math. Soc. 1. 1994. — P. 133–143.

[10] *Amoux Pierre, Rauzy Gerard.* Representation geometriques de suites de complexite $2n + \ell$ // Bull. Soc. Math. France, 119, 1991. — P. 199–215.

[11] *Andersson Arne, Larsson N. Jesper, Swanson Kurt.* Suffix trees on words // Algorithmica, 23–3, 1999. — P. 246–260.

[12] *Apostolico Albeno.* The myriad virtues of subword trees // Combinatorial Algorithms on Words (*NATO ASI Series F12*). Albeno Apostolico & Zvi Galil (eds.), Springer–Verlag, 1985. — P. 85–96.

[13] *Apostolico Alberto and Crochemore Maxime.* Optimal canonization of all substrings of a string // Information & Computation, 95–1, 1991. — P. 76–95.

[14] *Apostolico Alberto, Atallah Mikhail J., Larmore Lawrence and McFaddin Scott.* Efficient parallel algorithms for string editing problems // SIAM J. Computing, 19–5, 1990. — P. 968–988.

[15] *Apostolico Alberto, Ehrenfeucht Andrzej.* Efficient detection of quasiperiodicities in strings // TCS, 119, 1993. — P. 247–265.

[16] *Apostolico Alberto, Giancarlo Raffaele.* The Boyer-Moore-GaliI string searching strategies revisited // SIAM J. Comput., 15–1, 1986. — P. 98–105.

[17] *Apostolico Alberto, Guerra C.* The longest common subsequence problem revisited // Algorithmica, 2, 1987. — P. 315–336.

[18] *Apostolico Alberto, Preparata Franco P.* Optimal off-line detection of repetitions in a string // TCS, 22, 1983. — P. 297–315.

[19] *Apostolico Atberto, Farach Martin, Iliopoulos Costas S.* Optimal superprimitivity testing for strings // IPL, 39, 1991. — P. 17–20.

[20] *Arlazarov V. L., Dinic E. A., Kronrod M. A., Faradzev I. A.* On economical construction of the transitive closure of a directed graph // Soviet Math. Dokl, 11–5, 1970. — P. 1209–1210.

[21] *Baeza-Yates Ricardo A.* Algorithms for string searching: a survey // ACM SIGIR Forum, 25-3/4, 1989. — P. 34–58.

[22] *Baeza-Yates Ricardo A.* String searching algorithms revisited // Algorithms & Data Structures. Lecture Notes in Computer Science 382, Frank Dehne, Jorg-Riidiger Sack & Nicola Santoro (eds.). Springer–Verlag, 1989. — P. 75–96.

[23] *Baeza-Yates Ricardo A., Chris H. Perleberg.* Fast and practical approximate string matching // Proc. Third Annual Symp. Combinatorial Pattern Matching. Aiberto Apostolico, Maxime Crochemore, Zvi Galil & Udi Manber (eds.) Lecture Notes in Computer Science, 644, Springer–Verlag, 1992. — P. 185–192.

[24] *Baeza-Yates Ricardo A., Gaston Gonnet H.* A new approach to text searching // CACM, 35–70, 1992. — P. 74–82.

[25] *Baeza-Yates Ricardo A., Gonnet Gaston H., Mireille Regnier.* Analysis of Boyer-Moore-type string searching algorithms // Proc. First Annual ACM-SIAMSymp. Discrete Algs., 1990. — P. 328–343.

[26] *Baeza-Yates Ricardo A., Gonzaio Navarro.* A faster algorithm for approximate string matching // Proc. Seventh Annual Symp. Combinatorial Pattern Matching. Dan S. Hirschberg & Gene W. Myers (eds.). Lecture Notes in Computer Science, 1075, Springer–Verlag, 1996. — P. 1–23.

[27] *Baeza-Yates Ricardo A., Gonzaio Navarro.* Multiple approximate string matching // Proc. Fifth Annual Workshop on Algorithms & Data Structures. F. Dehne et al. (eds.), 1997. — P. 174–184.

[28] *Baeza-Yates Ricardo A., Regnier Mireille.* Average running time of the Boyer-Moore-Horspool algorithm // TCS, 92, 1992. — P. 19–31.

[29] *Bean Dwight R., Ehrenfeucht Andrzej, McNulty George F.* Avoidable patterns in strings of symbols // Pacific J. Math., 85–2, 1979. — P. 261–294.

[30] *Benson Gary.* A space efficient algorithm for finding the best non-overlapping alignment score // Proc. Fifth Annual Symp. Combinatorial Pattern Matching. Maxime Crochemore & Dan Gusfield (eds.). Lecture Notes in Computer Science, 807, Springer–Verlag, 1994. — P. 1–14.

[31] *Bentley Jon L., Sedgewick Robert.* Fast algorithms for sorting and searching strings // Proc. Eighth Annual ACM-S1AM Symp. Discrete Algs., 1997. — P. 360–369.

[32] *Berstel Jean, Luc Boasson.* Partial words and a theorem of Fine and Wilf // TCS, 218, 1999. — P. 135–141.

[33] *Berstel Jean, Seebold Patrice.* A characterization of Sturmian morphisms // The Mathematical Foundations of Computer Science. A. Borzyszkowski & S. Sokolowski (eds.), Springer–Verlag, 1993. — P. 281–290.

[34] *Berstel Jean.* Recent results in Sturmian words // Developments in Language Theory. World Scientific Press, 1996. — P. 13–24.

[35] *Blumer Anselm, Blumer Janet, Haussler David, Ehrenfeiicht Andrzej, Chen M. T., Seiferas Joel.* The smallest automaton recognizing the subwords of a text // TCS, 40–1, 1985. — P. 31–55.

[36] *Booth Kellogg S.* Lexicographically least circular substrings // IPL, 10–4/5, 1980. — P. 240–242.

[37] *Boshemitzan M., Praenkel Aviezri S.* A linear algorithm for nonhomogeneous spectra of numbers // J. Algs., 5, 1984. — P. 187–198.

[38] *Boyer RobertS., Moore J. Strother.* A fast string searching algorithm // CACM, 20–10, 1977. — P. 762–772.

[39] *Braunholtz C. H.* Solution to Problem 5030: an infinite sequence of 3 symbols with no adjacent repeats // American Mathematical Monthly, 70, 1963. — P. 675–676.

[40] *Breslauer Daily.* An on-line string superprimitivity test // IPL, 44, 1992. — P. 345–347.

[41] *Brodal Gerth S., Lyngso Rune B., Pedersen Christian N. S., Stoye Jens*. Finding maximal pairs with bounded gap // J. Discrete Algs., 1, 2000. — P. 77–103.

[42] *Brodal Gerth S., Pedersen Christian N. S.* Finding maximal quasiperiodicities in strings // Proc. Eleventh Annual Symp. Combinatorial Pattern Matching. Raffaele Giancarlo & David Sankoff (eds.). Lecture Notes in Computer Science, 1848, Springer-Verlag, 2000. — P. 397–411.

[43] *Castelli M. Gabriella, Mignosi Filippo, Restivo Antonio.* Fine & Wilf's theorem for three periods and a generalization of Sturmian words // TCS, 218. — 1999. — P. 83–94.

[44] *Chang William I., Lampe Jordan.* Theoretical and empirical comparisons of approximate string matching algorithms // Proc. Third Annual Symp. Combinatorial Pattern Matching. Alberto Apostolico, Maxime Crochemore, Zvi Galil & Udi Manber (eds.). Lecture Notes in Computer Science, 644, Springer–Verlag, 1992. — P. 172–181.

[45] *Chang William I., Lawler Eugene L.* Approximate string matching in sublinear expected time // Proc. 31st Annual IEEE Symp. Foundations of Computer Science. — Vol. I. — 1990. — P. 116–124.

[46] *Chang William I., Lawler Eugene L.* Sublinear approximate string matching and biological applications // Algorithmica, 12–4/5, 1994. — P. 327–344.

[47] *Charras Christian, Lecroq Thierry.* Exact String Matching Algorithms. — Laboratoire d'Informatique, Universite de Rouen, 1997: `http://www.igm.univ-mlv.fr/~lecroq/string/index.html`

[48] *Chen K. T., Fox R. H., Lyndon Roger C.* Free differential calculus IV // Ann. Math., 68–1, 1958. — P. 81–95.

[49] *Chen M. T., Seiferas Joel.* Efficient and elegant subword-tree construction // Combinatorial Algorithms on Words (NATO ASI Series F12). Alberto Apostolico & Zvi Galil (eds.). Springer–Verlag, 1985. — P. 97–107.

[50] *Cole Richard, Hariharan Ramesh, Paterson Michael S., Zwick Uri.* Tighter lower bounds on the exact complexity of string matching // *SIAM* J. Comput., 24-1, 1995. — P. 30–45.

[51] *Cole Richard, Hariharan Ramesh.* Faster approximate string matching // Proc. Ninth Annual ACM-SIAM Symp. Discrete Algs., 1998. — P. 463–472.

[52] *Cole Richard, Hariharan Ramesh.* Tighter bounds on the exact complexity of string matching // Proc. 33rd Annual IEEE Symp. Foundations of Computer Science. 1992. — P. 600–609.

[53] *Cole Richard.* Tight bounds on the complexity of the Boyer-Moore string matching algorithm // SIAM J. Comput., 23–5, 1994. — P. 1075–1091.

[54] *Colussi Livio, Galil Zvi, Giancarlo Raffaele.* On the exact complexity of string matching // Proc. 3lst Annual IEEE Symp. on Foundations of Computer Science. — Vol I. — 1990. — P. 135–143.

[55] *Colussi Livio.* Correctness and efficiency of pattern matching algorithms // Information & Computation, 95, 1991. — P. 225–251.

[56] *Colussi Livio.* Fastest pattern matching in strings // J. Algs., 16–2, 1994. — P. 163–189.

[57] *Commentz-Waller Beate.* A string matching algorithm fast on the average // Proc. Sixth Intemat. Colloquium on Automata. Languages & Programming. Hermann A. Maurer (ed.), Lecture Notes in Computer Science, 71, Springer-Verlag, 1979. — P. 118–132.

[58] *Crawford Tim, Iliopoulos Costas S., Rajeev Raman.* String matching techniques for musical similarity and melodic recognition // Computing in Musicology, 11, 1998. — P. 73–100.

[59] *Crochemore Maxime, Czumaj Artur, Gasieniec Leszek, Lecroq Thierry, Plandowski Wojciech, Rytter Wojciech.* Fast practical multi-pattern matching // IPL, 71–3/4, 1999. — P. 107–113.

[60] *Crochemore Maxime, Hancart Christophe, Lecroq Thierry.* A unifying look at the Apostolico-Giancarlo string matching algorithm // J. Discrete Algs., 2–1 (в печати).

[61] *Crochemore Maxime, Hancart Christophe, Thierry Lecroq.* Algorilhmique du Texte, Vuibert. — Paris. — 2001.

[62] *Crochemore Maxime, Holub Jan.* On the implementation of compact DAWGs // Proc. Seventh Internal Conf. Implementation & Application of Automata. — 2002. — P. 293–298.

[63] *Crochemore Maxime, Iliopoulos Costas S., Pinzon Yoan J.* Speeding up Hirschberg and Hunt-Szymanski LCS algorithms // Proc. Eighth IEEE Internal. Symp. String Processing & Information Retrieval. — IEEE Computer Science Press, 2001. — P. 59–67.

[64] *Crochemore Maxime, Iliopoulos Costas S., Pinzon Yoan J., Reid James F.* A fast and practical bit-vector algorithm for the longest common subsequence problem // IPL, 80–6, 2001. — P. 279–285.

[65] *Crochemore Maxime, Lecroq Thierry.* Tight bounds on the complexity of the Apostolico-Giancarlo algorithm // IPL, 63–4, 1997. — P. 195–203.

[66] *Crochemore Maxime, Perrin Dominique.* Two-way string matching // JACM, 38–3, 1991. — P. 651–675.

[67] *Crochemore Maxime, Rytter Wojciech. Text Algorithms.* — Oxford University Press. 1994, 412 p.

[68] *Crochemore Maxime, Verin Renaud.* Direct construction of compact directed acyclic word graphs // Proc. Eighth Annual Symp. Combinatorial Pattern Matching. Alberto Apostolico & Jotun Hein (eds.). Lecture Notes in Computer Science, 1264, Springer–Verlag, 1997. — P. 116–129.

[69] *Crochemore Maxime.* An optimal algorithm for computing all the repetitions in a word // IPL, 12–5, 1981. — P. 244–248.

[70] *Crochemore Maxime.* Linear searching for a square in a word // Bull. EATCS, 24, 1984. — P. 66–72.

[71] *Crochemore Maxime.* Reducing space for index implementation // TCS (в печати).

[72] *Crochemore Maxime.* Transducers and repetitions // TCS, 45–1, 1986. — P. 63–86.

[73] *Crocheroore Maxime, Czumaj Artur, Gasieniec Leszek, Jarominek Stefan, Lecroq Thierry, Plandowski Wojciech, Rytter Wojciech.* Speeding up two string-matching algorithms // Algorilhmica, 12, 1994. — P. 247–267.

[74] *Currie James D.* There are ternary circular square-free words of length n for $n \geqslant 18$ // *Electronic J. Combinatorics* (в печати).

[75] *Currie James D.* Which graphs allow infinite nonrepetitive walks? // Discrete Math., 87, 1991. — P. 249–260.

[76] *Davies G., Bowsher S.* Algorithms for pattern matching // Software — Practice & Experience, 16–6, 1986. — P. 575–601.

[77] *Domolki Balint.* A universal computer system based on production rules // BITS, 1968. — P. 262–275.

[78] *Domolki Balint.* Algorithms for the recognition of properties of symbol strings (Russian) // J. Computing Math. & Math. Physics, 5–1, 1965.

[79] *Duval Jean-Pierre, Lecroq Thierry, Arnaud Lefebvre.* Border array on bounded alphabet // Proc. Prague Stringology Conf. '02, 2002.

[80] *Duval Jean-Pierre.* Factorizing words over an ordered alphabet // J. Algs., 4, 1983. — P. 363–381.

[81] *Ehrenfeucht Andrzej, Haussler David.* A new distance metric on strings computable in linear time // Discrete Appl. Math., 20, 1988. — P. 191–203.

[82] *El-Mabrouk Nadia, Maxime Crochemore.* Boyer-Moore strategy to efficient approximate string matching // Proc. Seventh Annual Symp. Combinatorial Pattern Matching. Dan S. Hirschberg & Gene W. Myers (eds.), Lecture Notes in Computer Science, 1075, Springer-Verlag, 1996. — P. 24–28.

[83] *Erdos Pal.* Some unsolved problems // Magyar Tud. Akad. Mat. Kutato Intez.et Kozl., 6, 1961. — P. 221–254.

[84] *Erickson Bruce W., Sellers Peter H.* Recognition of patterns in genetic sequences // Time Warps, String Edits & Macromolecules: The Theory & Practice of Sequence Comparison. David Sankoff & Joseph B. Kruskal (eds.), Addison–Wesley, 1983. — P. 55–91.

[85] *Farach Martin, Muthukrishnan S.* Optimal logarithmic time randomized suffix tree construction // Proc. 23rd Internal. Colloquium Automata, Languages & Programming, 1996.

[86] *Farach Martin.* Optimal suffix tree construction with large alphabets // Proc. 38th Annual IEEE Symp. Foundations of Computer Science, 1997. — P. 137–143.

[87] *Fine N. J., Wilf Herbert S.* Uniqueness theorems for periodic functions // Proc. Amer. Math. Soc., 16, 1965. — P. 109–114.

[88] *Fraenkel Aviezri S., Simpson R. Jamie.* How many squares can a string contain? // J. Combinatorial Theory (A), 82, 1998. — P. 112–120.

[89] *Fraenkel Aviezri S., Simpson R. Jamie.* The exact number of squares in Fibonacci words // TCS, 218–1, 1999. — P. 83–94.

[90] *Franek Frantisek, Gao Shudi, Lu Weilin, Ryan Patrick J., Smyth W. F., Sun Yu, Yang Lu.* Verifying a Border Array in Linear Time // J. Combinatorial Math. & Combinatorial Comput., 42, 2002.

[91] *Franek Frantisek, Jiang Jiandong, Lu Weilin, Smyth W. F.* Two-pattern strings // Proc. Thirteenth Annual Symposium on Combinatorial Pattern Matching. Alberto Apostolico & Masayuki Takeda (eds.). Lecture Notes in Computer Science, Springer-Verlag, 2002. — P. 76–84.

[92] *Franek Frantisek, Karaman Ayse, Smyth W. F.* Repetitions in Sturmian strings // TCS, 249, 2000. — P. 289–303.

[93] *Franek Frantisek, Smyth W. F., Tang Yudong.* Computing all repeats using suffix arrays // Proc. Thirteenth Australasian Workshop on Combinatorial Algorithms. Elizabeth Billington, Diane Donovan & Abdollah Khodkar (eds.), 2002. — P. 171–184.

[94] *Franek Frantisek, Smyth W. F., Xiao Xiangdong.* A note on Crochemore's repetitions algorithm — a fast space-efficient approach // Proc. Prague Stringology Conf. '02. M. Balik & M. Simanek (eds.), 2002. — P. 36–43.

[95] *Galil Zvi, Giancarlo Raffaele.* Improved string matching with k mismatches // S1GACT News, 17, 1986. — P. 52–54.

[96] *Galil Zvi, Giancarlo Raffaele.* On the exact complexity of string matching: lower bounds // SIAM J. Comput., 20–6, 1991. — P. 1008–1020.

[97] *Galil Zvi, Giancarlo Raffaele.* On the exact complexity of string matching: upper bounds // SIAM J. Comput., 21–3, 1992. — P. 407–437.

[98] *Galil Zvi, Park Kunsoo.* An improved algorithm for approximate string matching // SIAM J. Comput. 19–8, 1990. — P. 989–999.

[99] *Galil Zvi, Seiferas Joel.* Time-space optimal string matching // J. Computer & System Sci. 26–3, 1983. — P. 280–294.

[100] *Galil Zvi.* On improving the worst case running time of the Boyer-Moore string matching algorithm // CACM, 22–9, 1979. — P. 505–508.

[101] *Gao Shudi.* New Properties of Borders & Covers of Strings // M. Sc. thesis, Department of Computing & Software, McMaster University, 2001.

[102] *Giegerich Robert, Kurtz Stefan.* A comparison of imperative and purely functional suffix tree constructions // Science of Computer Programming, 25-2/3, 1995. — P. 187–218.

[103] *Gonnet Gaston H., Baeza-Yates RicardoA.* Handbook of Algorithms & Data Structures in Pascal and C, 2nd edition // Addison–Wesley, 1991.

[104] *Guibas Leo J., Odlyzko Andrew M.* A new proof of the linearity of the Boyer-Moore string searching algorithm // SIAMJ. Comput. 9–4, 1980. — P. 672–682.

[105] *Guibas Leo J., Odlyzko Andrew M.* Periods in strings // J. Combinatorial Theory (A) 30, 1981. — P. 19–42.

[106] *Guibas Leo J., Odlyzko Andrew M.* String overlaps, pattern matching and nontransitive games // J. Combinatorial Theory (A) 30, 1981. — P. 183–208.

[107] *Gusfield Dan, Stoye Jens.* Linear Time Algorithms for Finding & Representing all the Tandem Repeats in a String // Tech. Rep. CSE-98-4, Computer Science Department, University of California, Davis, 1998.

[108] *Gusfield Dan.* Algorithms on Strings, Trees, & Sequences. — Cambridge University Press, 1997, 534 p.

[109] *Hamming Richard.* Coding & Information Theory // Prentice Hall, 1982.

[110] *Hancart Cristophe.* Analyse Exacle el en Moyenne d'Algorithmes de Recherche d'un Motif dans un Texte // Ph. D. thesis, Universite de Paris, 1993, 96 p.

[111] *Hannan Sampath K., Myers Gene W.* An algorithm for locating nonoverlapping regions of maximum alignment score // Proc. Fourth Annual Symp. Combinatorial Pattern Matching. Alberto Apostolico, Maxime Crochemore, Zvi Galil & Udi Manber (eds.). Lecture Notes in Computer Science 684, Springer-Verlag, 1993. — P. 74–86.

[112] *Harel Dov, Tarjan Robert Endre.* Fast algorithms for finding nearest common ancestors // SIAM J. Computing 13-2, 1984. — P. 338–355.

[113] *Haton Jean-Paul.* Contribution a l'Analyse, Parametrisation et la Reconnaissance Automatique de la Parole // Doctoral thesis, Universite de Nancy, 1973.

[114] *Hirschberg Dan S.* A linear space algorithm for computing maximal common subsequences // CACM 18-6, 1975. — P. 341–343.

[115] *Hirschberg Dan S.* Algorithms for the longest common subsequence problem // JACM 24-4, 1977. — P. 664–675.

[116] *Hirschberg Dan S.* An information-theoretic lower bound for the longest common subsequence problem // IPL 7-1, 1978. — P. 40–42.

[117] *Hopcroft John E.* An $n \log n$ algorithm for minimizing states in a finite automaton // Theory of Machines & Computations, Z. Kohavi & A. Paz (eds.). Academic Press, 1971.

[118] *Hopcroft John E., Ullman Jeffrey D.* Introduction to Automata Theory. Languages, & Computation // Addison–Wesley, 1979.

[119] *Horspool R. Nigel.* Practical fast searching in strings // Software — Practice & Experience 70-6, 1980. — P. 501–506.

[120] *Huang Xiaoqiu.* A lower bound for the edit-distance problem under arbitrary cost function // IPL 27-6, 1988. — P. 319–321.

[121] *Hume Andrew, Sunday Daniel.* Fast string searching // Software — Practice & Experience 21-11, 1991. — P. 1221–1248.

[122] *Hunt James W., Szymanski Thomas G.* A fast algorithm for computing longest common subsequences // CACM 20-5, 1977. — P. 350–353.

[123] *Iliopoulos Costas S., Moore Dennis, Smyth W. F.* A characterization of the squares in a Fibonacci string // TCS 172, 1997. — P. 281–291.

[124] *Iliopoulos Costas S., Mouchard Laurent.* Fast local covers // Препринт.

[125] *Iliopoulos Costas S., Pinzon Yoan J.* Recovering an LCS in $O(n^2/w)$ time and space // Proc. Twelfth Australasian Workshop on Combinatorial Algorithms. Edi T, Baskoro (ed.), Institute of Technology Bandung, 2001. — P. 106–117.

[126] *Iliopoulos Costas S., Smyth W. F.* A fast average case algorithm for Lyndon decomposition // Internal. 3. Computer Math. 57-3/4, 1995. — P. 15–31.

[127] *Inenaga S., Hoshino H., Shmohara A., Takeda Masayuki, Arikawa S., Mauri G., Pavesi G.* On-line construction of compact directed acyclic word graphs // Proc. Twelfth Annual Symp. Combinatorial Pattern Matching. Amihood Amir & Gad M. Landau (eds.) Lecture Notes in Computer Science 2087, Springer–Verlag, 2001. — P. 169–180.

[128] *Jacobson Guy, Vo Kiem-Phong.* Heaviest increasing/common subsequence problems // Proc. ThirdAnnual Symp. Combinatorial Pattern Matching. Alberto Apostolico, Maxime Crochemore, Zvi Galil & Udi Manber (eds.) Lecture Notes in Computer Science 644, Springer-Verlag, 1992. — P. 52–66.

[129] *Jennings Christopher G.* A Linear-Time Algorithm for Fast Exact Pattern Matching in Strings // M. Sc. thesis. Department of Computing & Software, McMaster University, 2002.

[130] *Karp Richard M.* Reducibility among combinatorial problems // Complexity of Computer Computations. Raymond E. Miller & James W. Thatcher (eds.), IBM Research Symposia Series, 1972. — P. 85–103.

[131] *Karp Richard M., Rabin Michael O.* Efficient randomized pattern-matching algorithms // IBM J. Res. Develop. 31-2, 1987. — P. 249–260.

[132] *Keranen Veikko.* Abelian squares are avoidable on 4 letters // Automata, Languages & Programming. Lecture Notes in Computer Science 623, Springer-Verlag, 1992. — P. 41–52.

[133] *Kim Sung-Ryul, Park Kunsoo.* A dynamic edit distance table // Proc. Eleventh Annual Symp. Combinatorial Pattern Matching. Raffaele Giancarlo & David Sankoff (eds.), Lecture Notes in Computer Science 1848, Springer-Verlag, 2000. — P. 60–68.

[134] *Knuth Donald E.* Big omichron, big omega, and big theta // ACM SIGACT News. April, 1976. — P. 18–24.

[135] *Knuth Donald E.* The Art of Computer Programming I: Fundamental Algorithms, 2nd edition // Addison–Wesley, 1973, 722 p. (Русский перевод 3-го издания: *Кнут Д.Э.* Искусство программирования. Т.1. Основные алгоритимы. — Издательский дом "Вильямс", 2000.)

[136] *Knuth Donald E.* The Art of Computer Programming III: Sorting & Searching // Addison-Wesley, 1973, 634 p. (Русский перевод 2-го издания: *Кнут Д.Э.* Искусство программирования. Т.3. Сортировка и поиск. — Издательский дом "Вильямс", 2000.)

[137] *Knuth Donald E., Morris James H., Pratt Vaughan R.* Fast pattern matching in strings // SIAM J. Comput. 6-2, 1977. — P. 323–350.

[138] *Kolpakov Roman, Kucherov Gregory. Maximal Repetitions in Words, or How to Find All Squares in Linear Time* // Rapport Inteme LORIA 98-R-227,Laboratoire Lorrain de Recherche en Informatique et ses Applications, France, 1998.

[139] *Kolpakov Roman, Kucherov Gregory.* On maximal repetitions in words // J. Discrete Algs. 7, 2000. — P. 159–186.

[140] *Kolpakov Roman, Kucherov Gregory.* On the Sum of Exponents of Maximal Repetitions in a Word // Rapport Inteme LORIA 99-R-034, Laboratoire Lorrain de Recherche en Informatique et ses Applications, France, 1999.

[141] *Kosaraju S. Rao.* Computation of squares in a string // Proc. Fifth Annual Symp. Combinatorial Pattern Matching. Maxime Crochemore & Dan Gusfield (eds.). Lecture Notes in Computer Science 807, Springer-Verlag, 1994. — P. 146–150.

[142] *Kowalski G., Meltzer A.* New multi-term high speed text search algorithms // Proc. First IEEE Internal. Conf. on Computer Applications, 1984. — P. 514–522.

[143] *Kurtz Stefan.* Reducing the space requirement of suffix trees // Software — Practice & Experience 29-13, 1999. — P. 1149–1171.

[144] *Landau Gad M., Myers Gene W., Schmidt Jeanette P.* Incremental string comparison // SIAMJ. Comput. 27-2, 1998. — P. 557–582.

[145] *Landau Gad M., Schmidt Jeanette P.* An algorithm for approximate tandem repeats // Proc. Fourth Annual Symp. Combinatorial Pattern Matching. Alberto Apostolico, Maxime Crochemore, Zvi Galil & Udi Manber (eds.), Lecture Notes in Computer Science 684, Springer-Verlag, 1993. — P. 120–133.

[146] *Landau Gad M., Vishkin Uzi.* Efficient string matching in the presence of errors // Proc. 26th Annual IEEE Symp. Foundations ofComputer Science, 1985. — P. 126–136.

[147] *Landau Gad M., Vishkin Uzi.* Efficient string matching with k mismatches // TCS 43, 1986. — P. 239–249.

[148] *Landau Gad M., Vishkin Uzi.* Fast parallel and serial approximate string matching // J.Algs. 10, 1989. — P. 157–169.

[149] *Landau Gad M., Vishkin Uzi.* Fast string matching with k differences // J. Computer & System Sci. 37, 1988. — P. 63–78.

[150] *Landau Gad M., Vishkin Uzi.* Introducing efficient parallelism into approximate string matching and a new serial algorithm // Proc. Eighteenth ACM Symp. Theory of Comput., 1986. — P. 220–230.

[151] *Larsson N. Jesper.* Extended application of suffix trees to data compression // Proc. IEEE Data Compression Conference, 1996. — P. 190–199.

[152] *Lecroq Thierry.* A variation on the Boyer–Moore algorithm // TCS 92, 1992. — P. 119–144.

[153] *Lecroq Thierry.* Experimental results on string matching algorithms // Software — Practice & Experience 25-7, 1995. — P. 727–765.

[154] *Lecroq Thierry.* New Experimental Results on Exact String-Matching // Rapport LIFAR 2000.03, Universite de Rouen, 2000.

[155] *Leech John.* A problem on strings of beads // Math. Gazette 4J, 1957. — P. 277–278.

[156] *Lempel Abraham, Ziv Jacob.* On the complexity of finite sequences // IEEE Trans. Information Theory 22, 1976. — P. 75–81.

[157] *Lentin Andre, Schutzenberger Marcel P.* A combinatorial problem in the theory of free monoids // Combinatorial Mathematics & Its Applications. R. C. Bose & T. A. Dowling (eds.), University of North Carolina Press, 1969. — P. 128–144.

[158] *Levenshtein V. I.* Binary codes capable of correcting deletions, insertions and reversals // Cybernetics & Control Theory 10-8, 1966. — P. 707–710.

[159] *Lewis Harry R., Papadimitriou Christos H.* Elements of the Theory of Computation, 2nd edition // Prentice-Hall, 1998.

[160] *Li Yin, Smyth W. F.* Computing the cover array in linear time // Algorithmica 32-1, 2001. — P. 95–106.

[161] *Li Yin.* A Windows-Based String Algorithm Testing System // Undergraduate Computer Science Project, Department of Comp. Sci. & Systems, McMaster University, 1996.

[162] *Lothaire M.* (ed.). Algebraic Combinatorics on Words // Encyclopedia of Mathematics and its Applications. — Vol.90. — Cambridge University Press. — 2002. См. также `http://www-igm.univ-mlv.fr/\~\,berstel/`

[163] *Lothaire M.* (ed.). Combinatorics on Words, 2nd edition // Cambridge University Press, 1997.

[164] *Lothaire M.* (ed.). Combinatorics on Words // Addison–Wesley, 1983.

[165] *Lowrance Roy, Wagner Robert A.* An extension of the string-to-string correction problem // JACM 22-2, 1975. — P. 177–183.

[166] *Luca de Aldo, Filippo Mignosi.* Some combinatorial properties of Sturmian words // TCS 136, 1994. — P. 361–385.

[167] *Lyndon Roger C., Schiitzenberger Marcel P.* The equation $a^M = b^N c^P$ in a free group // Michigan Math. J. 9, 1962. — P. 289–298.

[168] *Main Michael G.* Detecting leftmost maximal periodicities // Discrete Applied Maths. 25, 1989. — P. 145–153.

[169] *Main Michael G., Lorentz Richard J.* An $O(n \log n)$ Algorithm for Recognizing Repetition // Tech. Rep. CS-79-056, Computer Science Department, Washington State University, 1979.

[170] *Main Michael G., Lorentz Richard J.* Linear time recognition of squarefree strings // Combinatorial Algorithms on Words (NATO ASI Series F12). Alberto Apostolico & Zvi Galil (eds.), Springer–Verlag, 1985. — P. 271–278.

[171] *Manber Udi, Myers Gene W.* Suffix arrays: a new method for on-line string searches // Proc. First Annual ACM-SIAM Symp. Discrete Algs., 1990. — P. 319–327.

[172] *Manber Udi, Myers Gene W.* Suffix arrays: a new method for on-line string searches // SIAM J. Comput. 22-5, 1993. — P. 935–948.

[173] *Masek William J., Paterson Michael S.* A faster algorithm for computing string-edit distances // J. Computer & System Sci. 20-1, 1980. — P. 18–31.

[174] *McCreight Edward M.* A space-economical suffix tree construction algorithm // JACM 23-2, 1976. — P. 262–272.

[175] *McNaughton Robert F., Yamada H.* Regular expressions and state graphs for automata // IRE Trans. Electronic Computers EC9-1, 1960. — P. 39–47.

[176] *Michael G. Main & Richard J. Lorentz.* An $O(n \log n)$ algorithm for finding all repetitions in a string // J. Algs., 5, 1984. — P. 422–432.

[177] *Michailidis Panagiotis D., Margaritis Konstantinos G.* On-line string matching algorithms: survey and experimental results // Internal. J. Computer Math. 76-4, 2001. — P. 411–434.

[178] *Mignosi Filippo, Seebold Patrice.* Morphismes sturmiens et regles de Rauzy // J. Theorie de Nombres de Bordeaux 5, 1993. — P. 221–233.

[179] *Mignosi Filippo.* Infinite words with linear subword complexity // TCS 65, 1989. — P. 221–242.

[180] *Mignosi Filippo.* On the number of factors of Sturmian words // TCS 82, 1989. — P. 221–242.

[181] *Moore Dennis, Smyth W. F.* A correction to: An optimal algorithm to compute all the covers of a string // IPL 54, 1995. — P. 101–103.

[182] *Moore Dennis, Smyth W. F.* An optimal algorithm to compute all the covers of a string // IPL 50-5, 1994. — P. 239–246.

[183] *Moore Dennis, Smyth W. F., Miller Dianne.* Counting Distinct Strings // Aigorithmica 13-1, 1999. — P. 1–13.

[184] *Morris James H., Pratt Vaughan R.* A Linear Pattern-Matching Algorithm // Tech. Rep. 40, University of California, Berkeley, 1970.

[185] *Morrison Donald R.* PATRICIA — practical algorithm to retrieve information coded in alphanumeric // JACM 15-4, 1968. — P. 514–534.

[186] *Muth Robert, Manber Udi.* Approximate multiple string search // Proc. Seventh Annual Symp. Combinatorial Pattern Matching. Dan S. Hirschberg & Gene W. Myers (eds.), Lecture Notes in Computer Science 1075, Springer–Verlag, 1996. — P. 75–86.

[187] *Myers Gene W.* A fast bit-vector algorithm for approximate string matching based on dynamic programming // JACM 46-3, 1999. — P. 395–415.

[188] *Myers Gene W.* A Four Russians Algorithm for Regular Expression Pattern Matching // Technical Report 88-34, Dept. of Computer Science, University of Arizona, 1988.

[189] *Myers Gene W.* An $O(ND)$ difference algorithm and its variations // Algorithmica I, 1986. — P. 251–266.

[190] *Nagy Benedek*, частное сообщение.

[191] *Navarro Gonzalo.* A guided tour to approximate string matching // ACM Computing Surveys 33–1, 2001. — P. 31–88.

[192] *Needleman Saul B., Wunsch Christian D.* A general method applicable to the search for similarities in the amino-acid principle of two proteins // J. Molecular Biology 48, 1970. — P. 443–453.

[193] *Parikh Rohit J.* On context-free languages // JACM I3-4, 1966. — P. 570–581.

[194] *Pleasants Peter A. B.* Non-repetitive sequences // Proc. Camb. Phil. Soc. 68, 1970. — P. 267–274.

[195] *Reichert Thomas A., Cohen Donald N., Wong Andrew K. C.* An application of information theory to genetic mutations and the matching of polypeptide sequences // J. Theoretical Biology 42, 1973. — P. 245–261.

[196] *Rivest Ronald L.* On the worst-case behaviour of string-searching algorithms // SIAMJ. Comput. 6-4, 1977. — P. 669–674.

[197] *Rote Gunter.* Sequences with subword complexity $2n$ // J. Number Theory 46-2, 1994. — P. 196–213.

[198] *Ryan Patrick J., Smyth W. F.* An approach to asymptotic complexity // Mathematics & Computer Education 26-2, 1992. — P. 135–146.

[199] *Rytter Wojciech.* A correct preprocessing algorithm for Boyer-Moore string searching // SIAMJ. Comput. 9-3, 1980. — P. 509–512.

[200] *Sadakane Kunihiko.* A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation // Proc. IEEE Data Compression Conference, 1998. — P. 129–138.

[201] *Sakoe Hiroaki, Chiba Seibi.* A dynamic programming approach to continuous speech recognition // Proc. Internal. Congress of Acoustics, Budapest, 1971, *paper 20C13*.

[202] *Sankoff David, Kruskal Joseph B.* (eds.). Time Warps, Siring Edits, & Macromolecules: the Theory & Practice of Sequence Comparison // Addison-Wesley, 1983.

[203] *Sankoff David.* Matching sequences under deletion-insertion constraints // Proc. USA National Acad. Sci., 69, 1972. — P. 4–6.

[204] *Schensted Craige.* Largest increasing and decreasing subsequences // Canadian J. Math., 73, 1961. — P. 179–191.

[205] *Schmidt Jeanette P.* All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings // SIAM J. Comput. 27-4, 1998. — P. 972–992.

[206] *Setubal Joao, Meidanis Joao.* Introduction to Computational Molecular Biology // PWS Publishing, 1997, 296 p.

[207] *Shiloach Yossi.* A fast equivalence-checking algorithm for circular lists // IPL 8-5, 1979. — P. 236–238.

[208] *Shiloach Yossi.* Fast canonization of circular strings // J. Algs., 2, 1981. — P. 107–121.

[209] *Sim Jeong Seop, Iliopouios Costas S., Park Kunsoo, Smyth W. F.* Approximate periods of strings // TCS 262, 2001. — P. 557–568.

[210] *Simpson R. Jamie.* Covers and the periodicity lemma (частное сообщение).

[211] *Sloane N. J. A., Plouffe Simon.* The Encyclopedia of Integer Sequences // Academic Press, 1995. См. также `http://www.research.att.com/~njas/sequences/`

[212] *Smit G. de V.* A comparison of three string matching algorithms // Software — Practice & Experience 12, 1982. — P. 57–66.

[213] *Smith Temple F., Waterman Michael S.* Identification of common molecular subsequences // J. Molecular Biology 747, 1981. — P. 195–197.

[214] *Stephen Graham A.* String Searching Algorithms // World Scientific Publishing, 1994, 243 p.

[215] *Stolarsky Kenneth B.* Beatty sequences, continued fractions, and certain shift operators // Canad. Math. Bull. 19-4, 1976. — P. 473–481.

[216] *Stoye Jens, Gusfield Dan.* Simple and flexible detection of contiguous repeats using a suffix tree // Proc. Ninth Annual Symp. Combinatorial Pattern Matching. Martin Farach–Colton (ed.), Lecture Notes in Computer Science 1448, Springer–Verlag, 1998. — P. 140–152.

[217] *Sunday Daniel M.* A very fast substring search algorithm // CACM 33-8, 1990. — P. 132–142.

[218] *Tarhio Jorma, Ukkonen Esko.* Approximate Boyer–Moore string matching // SIAM J. Comput. 22-2, 1993. — P. 243–260.

[219] *Thompson Ken.* Regular expression search algorithm // CACM 11-6, 1968. — P. 419–422.

[220] *Thue Axel.* Uber unendliche zeichenreihen // Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7, 1906. — P. 1–22.

[221] *Tolkien J. R. R.* The Lord of the Rings, Part II: The Two Towers // Houghton Mifflin, 1955.

[222] *Ukkonen Esko, Wood Derick.* Approximate string matching with suffix automata // Algorithmica 10, 1993. — P. 353–364.

[223] *Ukkonen Esko.* Algorithms for approximate string matching // Information & Control, 64, 1985. — P. 100–118.

[224] *Ukkonen Esko.* Approximate string-matching with q-grams and maximal matches // TCS 92, 1992. — P. 191–211.

[225] *Ukkonen Esko.* Constructing suffix trees on-line in linear time // Proc. IFIP 92. — Vol. I. — 1992. — P. 484–492.

[226] *Ukkonen Esko.* Finding approximate patterns in strings // J.Algs. 6, 1985. — P. 132–137.

[227] *Velichko V. M., Zagoruyko N. G.* Automatic recognition of 200 words // Internal. J. Man-Machine Studies 2, 1970. — P. 223–234.

[228] *Vintsyuk T. K.* Speech discrimination by dynamic programming // Cybernetics 4-1, 1968. — P. 52–57.

[229] *Wagner Robert A., Fischer Michael J.* The string-to-string correction problem // JACM 21-1, 1974. — P. 168–173.

[230] *Weiner Peter.* Linear pattern matching algorithms // Proc. 14th Annual IEEE Symp. Switching &Automata Theory, 1973. — P. 1–11.

[231] *Wright Alden H.* Approximate string matching using within-word parallelism // Software — Practice & Experience 24, 1994. — P. 337–362.

[232] *Wu Sun, Manber Udi, Myers Gene W.* A subquadratic algorithm for approximate limited expression matching // Algorithmica 15, 1996. — P. 50–67.

[233] *Wu Sun, Manber Udi.* Fast text searching allowing errors // CACM 35-10, 1992. — P. 83–91.

[234] *Yao Andrew Chi-Chih.* The complexity of pattern matching for a random string // SIAM J. Comput. 8-3, 1979. — P. 368–387.

[235] *Ziv Jacob, Lempel Abraham.* A universal algorithm for sequential data compression // IEEE Trans. Information Theory 23, 1977. — P. 337–343.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Научно-популярное издание

Билл Смит

Методы и алгоритмы вычислений на строках

Литературный редактор *Т.П. Кайгородова*
Верстка *А.Н. Полинчик*
Художественные редакторы *В.Г. Павлютин*, *Т.А. Тараброва*
Корректоры *Т.А. Корзун*, *В.В. Смоляр*

Издательский дом "Вильямс"
101509, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 21.08.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 40. Уч.-изд. л. 26,5.
Тираж 3000 экз. Заказ №0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.