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 ***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.*

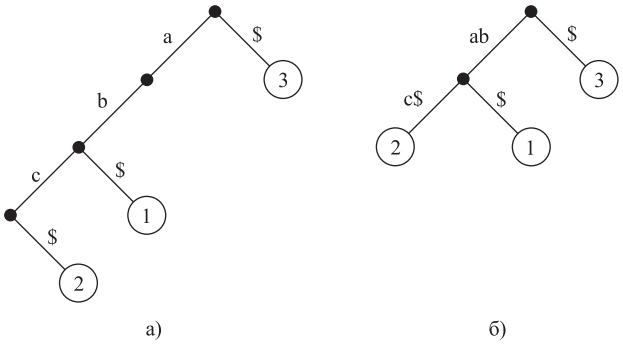

Рис. 2.1. Синтаксическое дерево для множества $X = \{ab, abc\}$

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.

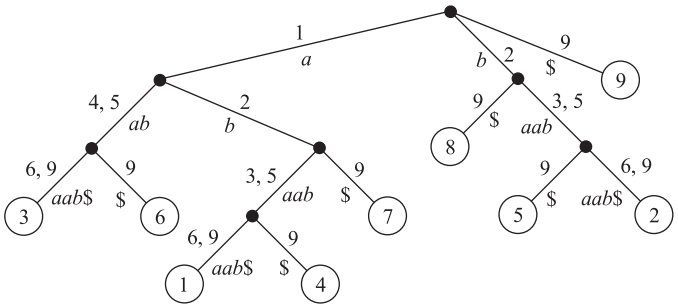

Рис. 2.2. Дерево суффиксов

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* $\boldsymbol{p}$ *в строку* $\boldsymbol{x}$
for $i \leftarrow 1$ **to** $n - m + 1$ **do**
  $j \leftarrow$ *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, $C \rightarrow G$) выполняются чаще или с большей вероятностью, чем другие подстановки (например, $C \rightarrow T$). 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 $a^*bb^*a(a \mid b)^*$;
- 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.