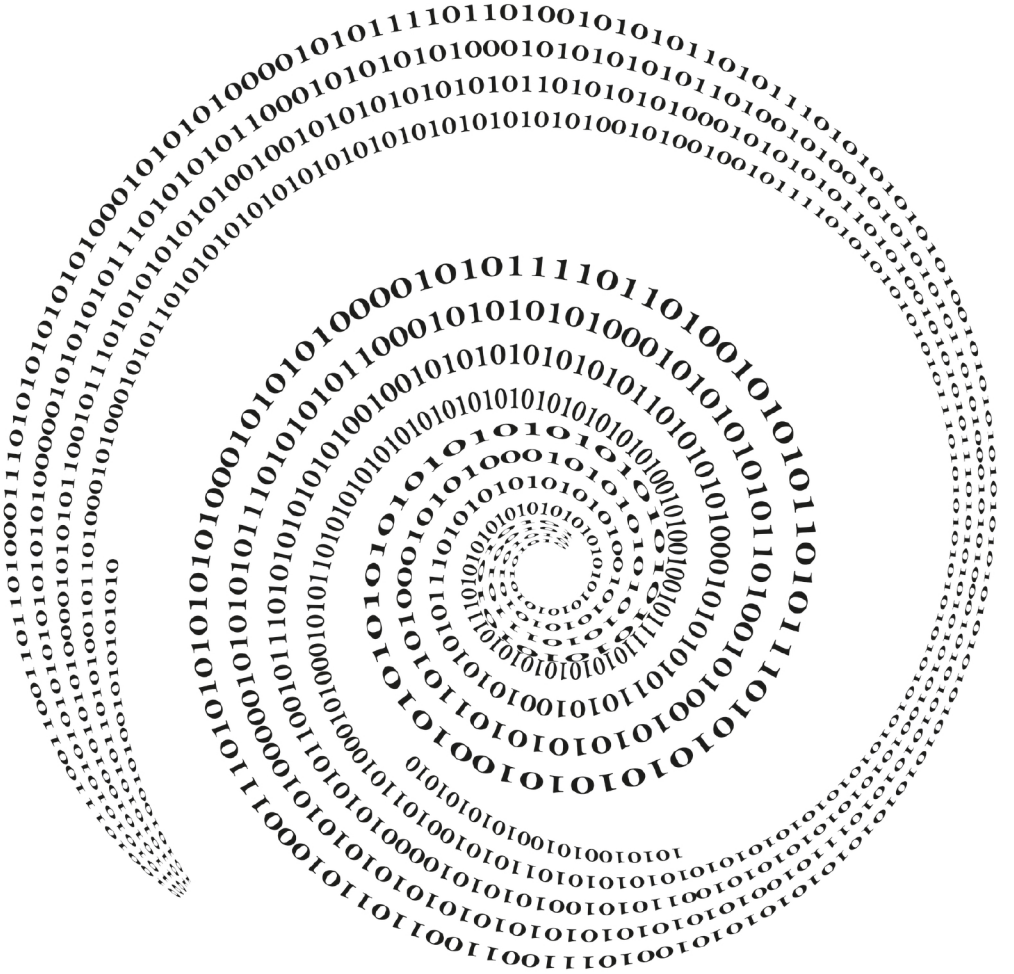
Максим Крошемор, Тьерри Лекрок, Войцех Риттер



Алгоритмы обработки текста

125 задач с решениями

ДМК Пресс
издательство

125 Problems in Text Algorithms

With Solutions

Maxime Crochemore, Thierry Lecroq, Wojciech Rytter

CAMBRIDGE
UNIVERSITY PRESS

Алгоритмы обработки текста

125 задач с решениями

Максим Крошемор, Тьерри Лекрок, Войцех Риттер

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2021

УДК 004.912
ББК 81.112
К83

Крошемор М., Лекрок Т., Риттер В.
К83 Алгоритмы обработки текста: 125 задач с решениями / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 312 с.: ил.

ISBN 978-5-97060-952-1

Сопоставление строк – одна из самых старых тем в теории алгоритмов, но по-прежнему занимает важное место в информатике. За прошедшие 20 лет мы видели технологические прорывы в таких разных приложениях, как информационный поиск и сжатие информации. Эта книга, представляющая собой богатое собрание задач и упражнений по важнейшим вопросам алгоритмов обработки текстов и комбинаторных свойств слов, предлагает студентам и исследователям приятный и прямой путь к изучению и практическому освоению концепций повышенного уровня.

Задачи взяты из многочисленных научных публикаций – как уже ставших классическими, так и сравнительно новых. Начав с основ, авторы рассматривают все более сложные задачи по комбинаторным свойствам слов (включая слова Фибоначчи и Туэ–Морса), поиску строк в тексте (включая алгоритмы Кнута–Морриса–Пратта и Бойера–Мура), эффективным структурам данных для представления текстов (включая суффиксные деревья и суффиксные массивы) и сжатия текста (включая методы Хаффмана, Лемпеля–Зива и Барроуза–Уилера).

Издание будет полезно в качестве пособия для подготовки к олимпиадам по информатике.

УДК 004.912
ББК 81.112

ISBN 978-1-108-83583-1 (англ.)

ISBN 978-5-97060-952-1 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Maker Media очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Эта книга посвящена алгоритмам обработки текста, которые иногда называют алгоритмической стрингологией (stringology). Текст (слово, строка, последовательность строк) – один из основных типов неструктурированных данных, играющий важную роль в информатике.

Предмет нашего рассмотрения многогранный, потому что лежит в основе многих дисциплин, особенно информатики и инженерных наук. Исследование неструктурированных данных – активно развивающаяся область, требующая эффективных методов как вследствие присутствия в разных местах операционных систем, так и для анализа огромного объема данных, порождаемых цифровыми сетями и оборудованием. Последнее относится прежде всего к компаниям в сфере ИТ, которые управляют гигантскими массивами данных в ЦОДах, но также ко многим научным направлениям за пределами информатики.

В этой книге представлен репрезентативный набор самых интересных задач в области обработки текстов. Лаконичное и увлекательное изложение открывает путь к более сложным темам. Материалы были взяты из сотен серьезных научных публикаций – каким-то из них уже сотни лет, а какие-то были написаны совсем недавно. По большей части задачи связаны с конкретными приложениями, но есть и более абстрактные. В основе большинства задач лежит остроумный короткий алгоритм, исключение составляют разве что несколько вводных комбинаторных проблем.

Эта книга – не просто очередная монография, а серия задач (головоломок и упражнений). Ее можно рассматривать как дополнение к книгам на эту тему, в которых предмет излагается более полно, в академическом стиле. Тем не менее большинство относящихся к предмету идей включено, так что книга заполняет доселе существовавший пробел и представляет долгожданный подарок для студентов и преподавателей, поскольку является первым задачником с решениями на тему алгоритмов обработки текста.

Книга состоит из семи глав.

- «Первые понятия стрингологии» – вводная глава, в которой читатель знакомится с терминологией, основными понятиями и инструментарием, который будет использоваться в последующих главах и отражает шесть основных направлений в рассматриваемой области.
- Глава «Комбинаторные задачи» посвящена комбинаторным свойствам слов. Это важная тема, поскольку комбинаторные свойства входных данных лежат в основе многих алгоритмов.
- Глава «Сопоставление с образцом» – классическая тема, охватывающая поиск в тексте и сопоставление строк.
- Глава «Эффективные структуры данных» посвящена структурам данных для индексирования текстов, в частности специализированным для текстов массивам и деревьям. Они применяются во многих алгоритмах.

- В главе «Регулярные структуры в словах» рассматриваются регулярные структуры, встречающиеся в текстах, в частности повторения и симметрии, от которых существенно зависит эффективность алгоритмов.
- Глава «Сжатие текста» посвящена методам, которые особенно важны для сжатия текста без потери информации.
- В главе «Разное» рассматриваются задачи, которым не нашлось места в предыдущих главах, но которые, без сомнения, заслуживают внимания.

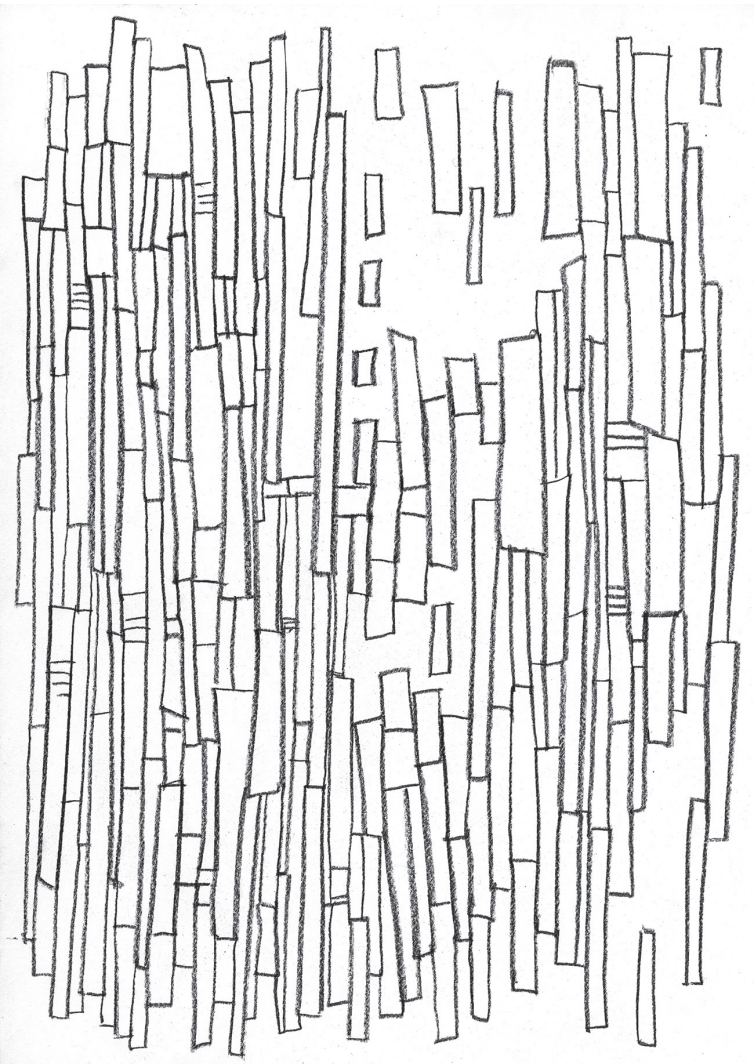
Включенные в книгу задачи отбирались и шлифовались на протяжении нескольких лет преподавания алгоритмов обработки текста в учебных учреждениях Франции, Польши, Великобритании и США, в основном студентам-магистрантам. Они приводятся вместе с решениями и со ссылками для дальнейшего изучения. Учитывался также предыдущий опыт авторов в написании учебников.

Преподаватели курсов по структурам данных и алгоритмам для магистрантов могут взять из книги те темы, которые считают наиболее подходящими для студентов. Однако книга в целом не является элементарной и задумывалась как справочник для исследователей, магистрантов и докторантов, а также для преподавателей вузов, читающих курсы по алгоритмам, в т. ч. не связанным напрямую с обработкой текстов. Ее следует рассматривать как дополнение к стандартным учебникам по данному предмету. Книга содержит всю необходимую для понимания информацию, поэтому позволяет быстро войти в курс дела и разобраться в постановке задач и их решениях без предварительной подготовки.

Издание может быть полезно для специальных курсов по алгоритмам обработки текста и для более общих курсов по алгоритмам и структурам данных. Хотя в нем вводятся все понятия и идеи, необходимые для решения задач, но подготовка в объеме курса по алгоритмам, структурам данных и дискретной математике, рассчитанного на студентов-второкурсников, была бы целесообразна для усвоения материала.

Глава 1

Первые понятия стрингологии



В этой главе мы познакомимся с обозначениями и определениями, а также кратко обрисуем некоторые конструкции, используемые в алгоритмах обработки текста.

Текст – центральный объект в «текстовых процессорах», которые обычно имеют дело с довольно большими объектами. Алгоритмы обработки текстов встречаются в различных областях науки и анализа информации. Во многих текстовых редакторах и языках программирования имеются средства обработки текстов. Например, в молекулярной биологии подобные алгоритмы применяются к анализу молекулярных последовательностей.

Слова

Алфавит – это непустое множество, элементы которого называются ***буквами***, или символами. Обычно используются алфавиты $\mathbf{A} = \{\mathtt{a},\mathtt{b},\mathtt{c},\ldots\}$, $\mathbf{B} = \{\mathtt{0},\mathtt{1}\}$ и натуральные числа. ***Словом***, или ***строкой***, над алфавитом A называется последовательность элементов A.

Последовательность букв нулевой длины называется ***пустым словом*** и обозначается ε. Множество всех конечных слов над алфавитом A обозначается A^*, а $A^+ = A^* \setminus \{\varepsilon\}$.

Длина слова x, т. е. длина последовательности букв, обозначается $|x|$. Буква в позиции $i = 0, 1, \ldots, |x| - 1$ непустого слова x обозначается $x[i]$, а i иногда называют ***индексом*** буквы. Последовательность $x = x[0]x[1]\cdots x[|x| - 1]$ обозначают также $x[0..|x| - 1]$. Множество букв, встречающихся в слове x, обозначается $alph(x)$. Например, для $x = \mathtt{abaaab}$ имеем $|x| = 6$ и $alph(x) = \{\mathtt{a},\mathtt{b}\}$.

Произведением, или ***конкатенацией***, двух слов x и y называется слово, образованное буквами x, за которыми следуют буквы y. Оно обозначается xy или $x \cdot y$, чтобы показать, из каких слов составлен результат. Нейтральным элементом относительно этой операции является пустое слово ε, а zy^{-1} и $x^{-1}z$ соответственно обозначаются такие слова x и y, для которых $z = xy$.

Сопряженным к слову x, а также ***ротацией*** или ***циклическим сдвигом*** x называется любое слово y, которое можно представить в виде vu, где $uv = x$. Это определение имеет смысл, потому что произведение слов, очевидно, некоммутативно. Например, множество слов, сопряженных к $\mathtt{abba}$, – его класс сопряженности (потому что сопряженность – отношение эквивалентности) – равно $\{\mathtt{aabb},\mathtt{abba},\mathtt{baab},\mathtt{bbaa}\}$, а класс сопряженности $\mathtt{abab}$ равен $\{\mathtt{abab},\mathtt{baba}\}$.

Слово x называется ***фактором*** (или ***подстрокой***) слова y, если существуют такие слова u и v, что $y = uxv$. Если $u = \varepsilon$, то говорят, что x является ***префиксом*** y, а если $v = \varepsilon$, то x является ***суффиксом*** y. Множества факторов, префиксов и суффиксов слова x обозначаются соответственно $Fact(x)$, $Pref(x)$ и $Suff(x)$.

Если x – непустой фактор $y = y[0..n-1]$, то он имеет вид $y[i..i + |x| - 1]$ для некоторого i. ***Вхождением*** x в y называется интервал $[i..i + |x| - 1]$, для которого $x = y[i..i + |x| - 1]$. Говорят, что i – ***начальная*** (или левая) ***позиция*** этого вхождения, а $i + |x| - 1$ – ***конечная*** (или правая) ***позиция***. Вхождение x в y можно также определить как тройку (u, x, v) такую, что $y = uxv$. Тогда начальная позиция вхождения равна $|u|$. Например, начальные и конечные позиции слова $x =$ aba в слове $y =$ babaababa равны

i	0	1	2	3	4	5	6	7	8
$y[i]$	b	a	b	a	a	b	a	b	a
Начальные позиции		1			4		6		
Конечные позиции				3			6		8

Для слов x и y $|y|_x$ обозначает количество вхождений x в y. В частности, $|y| = \sum\{|y|_a : a \in alph(y)\}$.

Слово x называется ***подпоследовательностью***, или ***подсловом*** y, если y разлагается в произведение $w_0x[0]w_1x[1]...x[|x| - 1]w_{|x|}$ для некоторых слов w_0, w_1, ..., $w_{|x|}$.

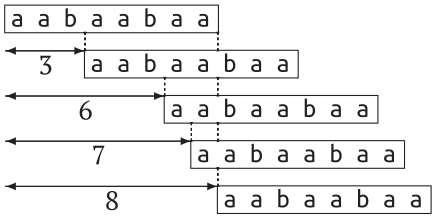
Говорят, что фактор, или подпоследовательность, x слова y является ***собственным***, если $x \neq y$.

Периодичность

Пусть x – непустое слово. Целое число p, $0 < p \leq |x|$, называется ***периодом*** x, если $x[i] = x[i + p]$ для $i = 0, 1, ..., |x| - p - 1$. Заметим, что длина слова является его периодом, так что любое непустое слово имеет хотя бы один период. ***Наименьший период*** x обозначается $per(x)$. Например, 3, 6, 7 и 8 являются периодами слова aabaabaa, а $per($aabaabaa$) = 3$. Отметим, что если p – период x, то все его кратные, не превосходящие $|x|$, также являются периодами x.

Перечислим свойства, эквивалентные определению периода p слова x. Во-первых, x можно единственным способом разложить в произведение $(uv)^k u$, где u и v – слова, причем v не пусто, k – положительное целое число и $p = |uv|$. Во-вторых, x является префиксом ux для некоторого слова u длины p. В-третьих, x является фактором u^k, где u – слово длины p, а k – целое положительное число. В-четвертых, x можно представить в виде произведения $uw = wv$, где u, v и w – слова и $p = |u| = |v|$.

И напоследок введем понятие границы. ***Границей*** слова x называется собственный фактор x, являющийся одновременно префиксом и суффиксом x. Граница x наибольшей длины обозначается $Border(x)$. Таким образом, ε, a, aa и aabaa – границы aabaabaa, а $Border($aabaabaa$) =$ aabaa.

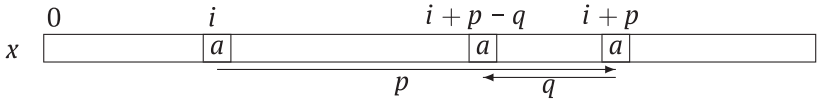


Между границами и периодами x имеется взаимно однозначное соответствие в силу четвертого из приведенных выше утверждений: с периодом p слова x ассоциирована граница $x[p..|x|-1]$.

Заметим, что, в силу определения, граница границы x также является границей x. А раз так, то список $\langle Border(x), Border^2(x), \ldots, Border^k(x) = \varepsilon\rangle$ содержит все границы x. Говорят, что непустое слово x ***свободно от границ***, если его единственной границей является пустое слово, или, что эквивалентно, если его единственным периодом является $|x|$.

Лемма* 1 *(лемма о периодичности). *Если p и q – периоды слова x и $p + q - \gcd(p, q) \le |x|$, то $\gcd(p, q)$ также является периодом x*[1].

Доказательство можно найти в учебниках (см. примечания). Слабой леммой о периодичности называется вариант этой леммы с более сильным условием: $p + q \le |x|$. Ее доказательство не вызывает затруднений и приведено ниже:

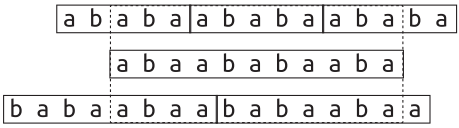


Утверждение леммы, очевидно, справедливо, если $p = q$. В противном случае без ограничения общности предположим, что $p > q$, и покажем сначала, что $p - q$ – период x. Обозначим i позицию в x, для которой $i + p < |x|$. Тогда $x[i] = x[i + p] = x[i + p - q]$, потому что p и q – периоды. И если $i + p \ge |x|$, то из условия следует, что $i - q \ge 0$. Тогда $x[i] = x[i - q] = x[i + p - q]$, как и прежде. Таким образом, $p - q$ – период x. Повторяя это рассуждение или воспользовавшись рекурсией, как в алгоритме Евклида, заключаем, что $\gcd(p, q)$ – период x.

Для иллюстрации леммы о периодичности рассмотрим слово x, допускающее периоды 5 и 8. Если дополнительно предположить, что x включает хотя бы две разные буквы, то $\gcd(5, 8) = 1$ не является периодом x. Таким образом, условие леммы не может удовлетворяться, а значит, $|x| < 5 + 8 - \gcd(5, 8) = 12$.

---

[1] gcd (greatest common divisor) – наибольший общий делитель. – *Прим. перев.*



Пример, изображенный на рисунке, показывает, что условие на периоды в лемме о периодичности не может быть ослаблено.

Регулярные структуры

Степень слова x определяется следующим образом: $x^0 = \varepsilon$, $x^i = x^{i-1}x$ для целого положительного i. k-я ***степень*** x обозначается x^k. Она называется ***квадратом***, если k – положительное четное число, и ***кубом***, если k – положительное, кратное 3.

Следующая лемма является первым следствием из леммы о периодичности.

Лемма 2. *Для слов x и y равенство $xy = yx$ имеет место тогда и только тогда, когда x и y – целые степени одного и того же слова. То же утверждение справедливо, когда существует два целых положительных числа k и l таких, что $x^k = y^l$.*

Доказательства обеих частей леммы по существу одинаковы (на самом деле лемма вытекает из более общего утверждения о кодах). Например, если $xy = yx$, то x и y являются границами слова, следовательно, $|x|$ и $|y|$ – его периоды, а в силу леммы о периодичности периодом является также $\gcd(|x|, |y|)$. Из того, что $\gcd(|x|, |y|)$ делит также $|xy|$, получаем требуемое утверждение. Обратное утверждение очевидно.

Непустое слово x называется ***примитивным***, если оно не является степенью никакого другого слова. Иными словами, x примитивное, если из того, что $x = u^k$ для некоторого слова u и целого положительного k, следует, что $k = 1$, а значит, $u = x$. Например, слово `abaab` примитивное, тогда как ε и `bababa` = $(\mathtt{ba})^3$ таковыми не являются.

Из леммы 2 следует, что всякое непустое слово является степенью ровно одного примитивного слова. Если $x = u^k$ и u примитивное, то u называется ***примитивным корнем*** из x, а k – его ***показателем степени***, обозначаемым $\exp(x)$. Вообще, показателем степени x называется величина $\exp(x) = |x|/per(x)$, необязательно целая, а слово называется ***периодическим***, если его показатель степени не меньше 2.

Отметим, что количество слов, сопряженных данному, т. е. размер его ***класса сопряженности***, равно длине примитивного корня из него.

Следующее утверждение также вытекает из леммы о периодичности.

Лемма 3 (лемма о примитивности, лемма о синхронизации). *Непустое слово x примитивное тогда и только тогда, когда оно входит фактором в свой квадрат только в виде префикса и суффикса, или, эквивалентно, тогда и только тогда, когда $per(x^2) = |x|$.*

Результат этой леммы показан на рисунке. Слово abbaba примитивное, поэтому входит в свой квадрат только двумя способами, тогда как слово ababab примитивным не является и в свой квадрат входит четырьмя способами.

Понятие ***серии***, или ***максимальной периодичности***, охватывает несколько типов регулярных структур в словах. Серией (run) в слове x называется максимальное вхождение периодического фактора. Более формально, это интервал $[i..j]$ позиций внутри x, для которого $\exp(x[i..j]) \geq 2$ и периоды $x[i-1..j]$ и $x[i..j+1]$, если они существуют, больше, чем период $[i..j]$. В этой ситуации, поскольку вхождение идентифицируется индексами i и j, мы также, допуская вольность речи, будем говорить, что $x[i..j]$ – серия.

Еще одна регулярность – наличие в словах взаимно обратных факторов, или палиндромов. ***Обращением***, или ***зеркальным отражением***, слова x называется слово $x^R = x[|x|-1]x[|x|-2]...x[0]$. С этой операцией связано понятие ***палиндрома***: слова, для которого $x^R = x$.

Например, в английском языке слова noon и testset являются палиндромами. Первое – четный палиндром вида uu^R, а второе – нечетный палиндром вида uau^R, где a – буква. Букву a можно заменить коротким словом, что ведет к понятию разрывного палиндрома, полезному при рассмотрении операций фолдинга, встречающихся в последовательностях биологических молекул. Еще пример: целые числа, десятичное представление которых является палиндромом, кратны 11; так, $1661 = 11 \times 151$, $175\,571 = 11 \times 15\,961$.

Упорядочение

Для некоторых алгоритмов важно существование отношения порядка в алфавите, обозначаемого символом $\leq$. Это отношение порядка следующим образом индуцирует ***лексикографический***, или ***алфавитный***, ***порядок*** на множестве слов. Как и порядок в самом алфавите, оно обозначается символом $\leq$. Для $x, y \in A^*$ $x \leq y$ тогда и только тогда, когда либо x является префиксом y, либо x и y можно представить в виде $x = uav$ и $y = ubw$, где u, v и w – слова, а a и b – буквы, причем $a < b$. Так, ababb < abba < abbaab, если считать, что a < b, и вообще на алфавите **A** определен естественный порядок.

Мы говорим, что x ***строго меньше*** y, и обозначаем это $x \ll y$, если $x \leq y$, но x не является префиксом y. Отметим, что из $x \ll y$ следует, что $xu \ll yv$ для любых слов u и v.

На базе лексикографического порядка определяются понятия ***слов Линдона*** и ***ожерелий***.

Словом Линдона x называется примитивное слово, наименьшее среди всех своих сопряженных. Эквивалентно, хотя это и не очевидно, можно сказать, что x меньше любого из своих собственных непустых суффиксов, поэтому его также называют ***самоминимальным словом***. Как следствие x не име-

ет границы. Известно, что любое непустое слово w допускает единственное разложение в произведение $x_0x_1...x_k$, где все x_i – слова Линдона и $x_0 \geq x_1 \geq ... \geq x_k$. Например, слово aababaabaaba раскладывается в произведение aabab · aab · aab · a, где aabab, aab и a – слова Линдона.

Ожерельем, или ***минимальным словом***, называется слово, наименьшее в своем классе сопряженности. Оно является (целой) степенью слова Линдона. Слово Линдона является ожерельем, но обратное неверно; например, $\mathtt{aabaab} = \mathtt{aab}^2$ – ожерелье, но не слово Линдона.

Примечательные слова

Помимо слов Линдона, есть еще три класса слов, которые обладают примечательными свойствами и часто используются в примерах. Это слова Туэ–Морса, слова Фибоначчи и слова де Брёйна. Первые два – префиксы (односторонне) бесконечных слов. Формально ***бесконечным словом*** над алфавитом A называется отображение множества натуральных чисел в A. Множество бесконечных слов обозначается A^∞.

Понятие (моноидного) ***морфизма*** является центральным для определения некоторых бесконечных множеств слов или ассоциированных с ними бесконечных слов. Морфизмом из A^* в себя (или в другой свободный моноид) называется отображение $h : A^* \to A^*$ такое, что $h(uv) = h(u)h(v)$ для любых слов u и v. Следовательно, морфизм полностью определяется образами $h(a)$ букв $a \in A$.

Слово Туэ–Морса порождается повторным применением ***морфизма Туэ–Морса*** μ из $\{\mathtt{a},\mathtt{b}\}^*$ в себя, определенного следующим образом:

$$\begin{cases}\mu(\mathtt{a}) = \mathtt{ab},\\ \mu(\mathtt{b}) = \mathtt{ba}.\end{cases}$$

Итеративное применение этого морфизма, начиная с буквы a, дает список ***слов Туэ–Морса*** $\mu^k(\mathtt{a})$, $k \geq 0$. Первые несколько таких слов показаны ниже:

$\tau_0 = \mu^0(\mathtt{a}) = \mathtt{a}$
$\tau_1 = \mu^1(\mathtt{a}) = \mathtt{ab}$
$\tau_2 = \mu^2(\mathtt{a}) = \mathtt{abba}$
$\tau_3 = \mu^3(\mathtt{a}) = \mathtt{abbabaab}$
$\tau_4 = \mu^4(\mathtt{a}) = \mathtt{abbabaabbaababba}$
$\tau_5 = \mu^5(\mathtt{a}) = \mathtt{abbabaabbaababbabaababbaabbabaab}$

В пределе получается такое ассоциированное бесконечное слово:

$$\mathbf{t} = \lim_{k\to\infty} \mu^k(\mathtt{a}) = \mathtt{abbabaabbaababbabaababbaabbabaab}\ldots.$$

Эквивалентное определение слов Туэ–Морса дает следующее рекуррентное соотношение:

$$\begin{cases}\tau_0 = \mathtt{a},\\ \tau_{k+1} = \tau_k\overline{\tau_k}, \quad k \geq 0,\end{cases}$$

где знак надчеркивания обозначает морфизм $\overline{\mathtt{a}} = \mathtt{b}$, $\overline{\mathtt{b}} = \mathtt{a}$. Отметим, что длина k-го слова Туэ–Морса $|\tau_k| = 2^k$.

Прямое определение **t** выглядит так: буква **t**[n] равна b, если число единиц в двоичном представлении n нечетно, и равна a в противном случае.

Известно, что бесконечное слово Туэ–Морса не содержит перекрытий (факторов вида *аиаиа*, где *а* – буква, а *и* – слово), т. е. факторов с показателем степени больше 2. Говорят, что оно ***свободно от перекрытий***.

Слово Фибоначчи тоже порождается повторным применением морфизма, а именно *морфизма Фибоначчи* φ, отображающего $\{\mathtt{a},\mathtt{b}\}^*$ в себя и определенного следующим образом:

$$\begin{cases}\varphi(\mathtt{a}) = \mathtt{ab},\\ \varphi(\mathtt{b}) = \mathtt{a}.\end{cases}$$

Начав с буквы a, мы получим список слов Фибоначчи $\varphi^k(\mathtt{a})$, $k \geq 0$, первые элементы которого приведены ниже:

$fib_0 = \varphi^0(\mathtt{a}) = \mathtt{a}$
$fib_1 = \varphi^1(\mathtt{a}) = \mathtt{ab}$
$fib_2 = \varphi^2(\mathtt{a}) = \mathtt{aba}$
$fib_3 = \varphi^3(\mathtt{a}) = \mathtt{abaab}$
$fib_4 = \varphi^4(\mathtt{a}) = \mathtt{abaababa}$
$fib_5 = \varphi^5(\mathtt{a}) = \mathtt{abaababaabaab}$
$fib_6 = \varphi^6(\mathtt{a}) = \mathtt{abaababaabaababaababa}$

В пределе получается такое ассоциированное бесконечное слово:

$$\mathbf{f} = \lim_{k\to\infty}\varphi^k(\mathtt{a}) = \mathtt{abaababaabaababaababaabaababaabaab}\cdots.$$

Эквивалентное определение слов Фибоначчи дает рекуррентное соотношение

$$\begin{cases}fib_0 = \mathtt{a},\\ fib_1 = \mathtt{ab},\\ fib_{k+1} = fib_k fib_{k-1}, \quad k \geq 1.\end{cases}$$

Длины этих слов образуют последовательность чисел Фибоначчи, т. е. $|fib_k| = F_{k+2}$. Напомним, что ***числа Фибоначчи*** определяются рекуррентно следующим образом:

$$\begin{cases}F_0 = 0,\\ F_1 = 1,\\ F_{k+1} = F_k + F_{k-1}, \quad k \geq 1.\end{cases}$$

Из их многочисленных свойств отметим следующие:

- $\gcd(F_n, F_{n-1}) = 1$ для $n \geq 2$,
- F_n – ближайшее целое к $\Phi^n/\sqrt{5}$, где $\Phi = ½(1 + \sqrt{5}) = 1.61803\ldots$ – ***золотое сечение***.

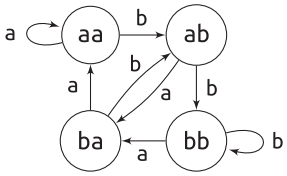
Интерес к словам Фибоначчи проистекает из их комбинаторных свойств и большого числа содержащихся в них повторений. Однако бесконечное число Фибоначчи не содержит ни одного фактора с показателем степени большим, чем $\Phi^2 + 1 = 3.61803\ldots$.

Слова де Брёйна определяются над алфавитом $A = \{a,b\}$ и параметризуются целым положительным числом k. Слово $x \in A^+$ называется словом де Брёйна порядка k, если каждое слово из A^k входит в x ровно один раз. Например, `ab` и `ba` – единственные слова де Брёйна порядка 1. Другой пример: `aaababbbaa` является словом де Брёйна порядка 3, поскольку восемь его факторов длины 3 совпадают с восемью словами A^3: `aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba` и `bbb`.

Существование слов де Брёйна порядка $k \geq 2$ можно проверить с помощью автомата де Брёйна, определенного следующим образом:

- состояниями являются слова из A^{k-1};
- ребра имеют вид (av, b, vb), где $a, b \in A$ и $v \in A^{k-2}$.

На рисунке ниже изображен автомат для слов де Брёйна порядка 3. Заметим, что из каждого состояния выходит ровно два ребра, помеченных буквами `a` и `b`, и что в каждое состояние входит ровно два ребра, помеченных одной и той же буквой. Поэтому ассоциированный с автоматом граф удовлетворяет условию Эйлера: все вершины имеют четную степень. Это означает, что существует эйлеров обход графа. Буквы, помечающие ребра, вошедшие в этот обход, образуют ***круговое слово де Брёйна***. Добавление в его конец его же префикса длины $k - 1$ дает обыкновенное слово де Брёйна.



Можно также доказать, что количество слов де Брёйна порядка k экспоненциально зависит от k.

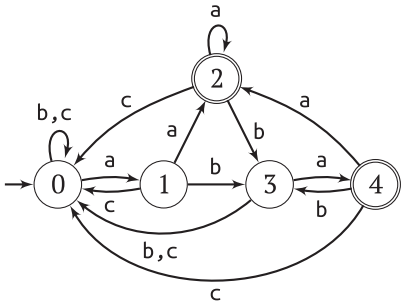
Слова де Брёйна можно определить и над большим алфавитом, они часто используются в качестве примеров предельных случаев, потому что содержат все факторы заданной длины.

Автоматы

Конечный ***автомат*** M над конечным алфавитом A состоит из следующих компонентов: конечное множество ***состояний*** Q, ***начальное*** состояние q_0,

множество ***заключительных*** состояний $T \subseteq Q$ и множество $F \subseteq Q \times A \times Q$ ***помеченных ребер***, или ***дуг***[1], соответствующих ***переходам*** состояний. Будем обозначать автомат M четверкой (Q, q_0, T, F) или иногда просто (Q, F), например если q_0 неявно подразумевается, а $T = Q$. Говорят, что дуга (p, a, q) исходит из состояния p и входит в состояние q, при этом состояние p называется ***исходным***, буква a – ***меткой*** дуги, а состояние q – ***конечным***. На рисунке ниже приведено графическое изображение автомата.

Количество дуг, исходящих из данного состояния, называется ***полустепенью исхода*** этого состояния. ***Полустепень захода*** состояния определяется аналогично. По аналогии с графами состояние q называется ***преемником*** состояния p по букве a, если $(p, a, q) \in F$; будем также говорить, что пара (a, q) является ***помеченным преемником*** состояния p.



Путем длины n в автомате $M = (Q, q_0, T, F)$ называется последовательность n соседних дуг $\langle (p_0, a_0, p'_0), (p_1, a_1, p'_1), \ldots, (p_{n-1}, a_{n-1}, p'_{n-1}) \rangle$ такая, что $p'_k = p_{k+1}$ для $k = 0, 1, \ldots, n-2$. ***Меткой*** пути называется слово $a_0 a \ldots a_{n-1}$, его ***началом*** – состояние p_0, а ***концом*** – состояние p'_{n-1}. Путь в автомате M называется ***успешным***, если его началом является начальное состояние q_0, а конец принадлежит T. Говорят, что автомат ***распознает***, или ***допускает***, слово, если оно является меткой успешного пути. Язык, состоящий из слов, распознаваемых автоматом M, обозначается $Lang(M)$.

Автомат $M = (Q, q_0, T, F)$ называется ***детерминированным***, если для любой пары $(p, a) \in Q \times A$ существует не более одного состояния $q \in Q$ для каждой дуги $(p, a, q) \in F$. В таком случае естественно рассмотреть ***функцию переходов*** автомата $\delta : Q \times A \to Q$, определенную для каждой дуги $(p, a, q) \in F$ так, что $\delta(p, a) = q$, и не определенную больше нигде. Функция δ тривиально распространяется на слова.

Известно, что любой язык, допускаемый автоматом, допускается также детерминированным автоматом и что существует единственный (с точностью до именования состояний) минимальный детерминированный автомат, допускающий язык.

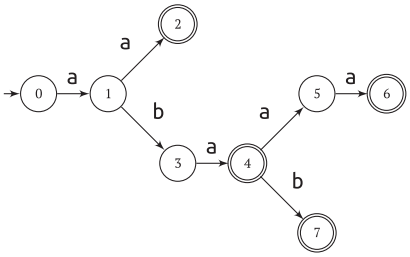
---

[1] Принято называть ребра ориентированного графа дугами, мы будем далее придерживаться этой терминологии. – *Прим. перев.*

Префиксные деревья

Префиксное дерево (trie) $\mathcal{T}$ над алфавитом A – это автомат, для которого пути, исходящие из начального состояния, корня, не сходятся. Чаще всего префиксные деревья используются для представления конечных множеств слов. Если никакое слово не является префиксом никакого другого слова из данного множества, то все слова будут ассоциированы с листьями дерева.

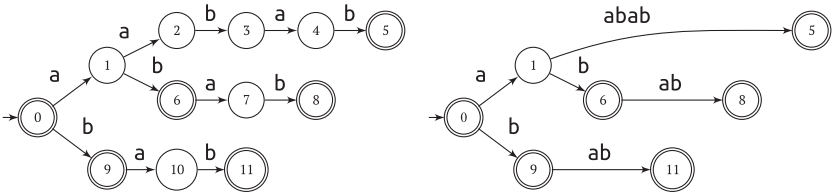
На рисунке ниже изображено префиксное дерево $\mathcal{T}$({aa, aba, abaaa, abab}). Состояния соответствуют префиксам слов, принадлежащих множеству. Например, состояние 3 соответствует префиксу длины 2 слов abaaa и abab. Заключительные состояния (обведенные двойной окружностью) 2, 4, 6 и 7 соответствуют словам из множества.



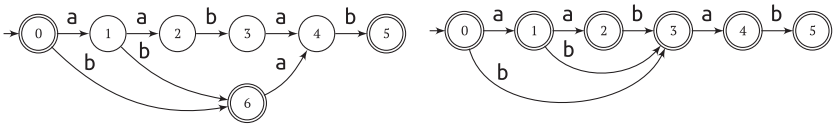
Суффиксные структуры

Суффиксные структуры данных, в которых хранятся суффиксы слова, важны для построения эффективных индексов. В этом качестве можно использовать префиксные деревья, но их размер может расти квадратично. Одно из решений этой проблемы – уплотнить префиксное дерево, получив суффиксное дерево слова. Для этого удаляются нелистовые узлы, из которых исходит только одно ребро, а дуги помечаются соответственными факторами слова. Удаленные узлы иногда называют неявными узлами суффиксного дерева, а оставшиеся узлы – явными.

Ниже показано префиксное дерево $\mathcal{T}$(*Suff*(aabab)) суффиксов aabab (слева) и его ***суффиксное дерево*** $\mathcal{ST}$(aabab) (справа). Чтобы получить полную структуру линейного размера, каждый фактор слова, помечающий дугу, нужно представить парой целых чисел (позиция, длина).



Второе решение, позволяющее уменьшить размер префиксного дерева суффиксов, – минимизировать его, т. е. рассмотреть минимальный детерминированный автомат, допускающий суффиксы слова, его ***суффиксный автомат***. На рисунке ниже слева показан суффиксный автомат $\mathcal{S}$(aabab) слова aabab.



Известно, что в $\mathcal{S}(x)$ менее $2|x|$ состояний и менее $3|x|$ дуг, т. е. его общий размер $O(|x|)$ линейно зависит от $|x|$. Факторный автомат $\mathcal{F}(x)$ слова, т. е. минимальный детерминированный автомат, допускающий его факторы, может быть еще меньше, потому что все его состояния заключительные. На рисунке выше справа показан факторный автомат слова aabab, в котором состояние 6 автомата $\mathcal{S}$(aabab) объединено с состоянием 3.

Суффиксный массив

Суффиксный массив слова также применяется для построения индексов, но работает иначе, чем деревья или автоматы. Его основное назначение – отсортировать суффиксы слова, чтобы сделать возможным двоичный поиск по его факторам. Чтобы повысить эффективность поиска, вводится в рассмотрение еще одно понятие: наибольший общий префикс соседних суффиксов в отсортированном списке.

Эта информация хранится в двух массивах: SA и LCP. Массив SA получен обращением массива Rank, который содержит ранги (т. е. места в отсортированном списке) суффиксов с привязкой к их начальным позициям.

Таблицы ниже построены для слова aababa. Его отсортированный список суффиксов имеет вид a, aababa, aba, ababa, ba, baba, а их начальные позиции равны соответственно 5, 0, 3, 1, 4 и 2. Этот последний список, индексированный рангами суффиксов, и хранится в массиве SA.

i	0	1	2	3	4	5
$x[i]$	a	a	b	a	b	a
Rank[i]	1	3	5	2	4	0

r	0	1	2	3	4	5	6	7	8	9	10	11	12
SA[r]	5	0	3	1	4	2							
LCP[r]	0	1	1	3	0	2	0	0	1	0	0	0	0

Таблица LCP содержит ***наибольшие общие префиксы***, которые хранятся как максимальные длины общих префиксов соседних суффиксов:

$$\text{LCP}[r] = |lcp(x[\text{SA}[r-1]..|x|-1], x[\text{SA}[r]..|x|-1])|,$$

где *lcp* обозначает наибольший общий префикс двух слов. Она, например, дает LCP[0..6]. Следующие значения в диапазоне LCP[7..12] соответствуют той же информации для суффиксов, начинающихся в позициях d и f, когда пара (d, f) встречается при двоичном поиске. Формально для такой пары значение хранится в позиции $|x| + 1 + \lfloor (d + f)/2 \rfloor$. Например, в приведенном выше массиве LCP значение 1, соответствующее паре (0, 2), максимальной длине префиксов между $x[5..5]$ и $x[3..5]$, хранится в позиции 8.

Таблица Rank в основном применяется в приложениях суффиксного массива, не связанных с поиском.

Сжатие

Самые эффективные методы ***сжатия*** текстов общего вида основаны либо на алгоритме факторизации слов Лемпеля–Зива, либо на более простых способах, опирающихся на преобразование слов Барроуза–Уилера.

При обработке слова в онлайновом режиме цель ***схемы сжатия Лемпеля–Зива*** заключается в том, чтобы уловить информацию, которая уже встречалась раньше. В результате имеем факторизацию слова x вида $u_0u_1...u_k$, где u_i – самый длинный префикс $u_i...u_k$, который встречался до появления x. Если этот префикс пуст, то выбирается первая буква $u_i...u_k$, которая не встречается в $u_0...u_{i-1}$. Допуская вольность речи, фактор u_i иногда называют ***наибольшим предыдущим фактором*** в позиции $|u_0...u_{i-1}|$ слова x.

Например, слово `abaababababaaabab` факторизуется следующим образом: `a·b·a·aba·baba·aabab·b`.

Существует несколько способов определить факторы, составляющие разложение, опишем некоторые из них. Фактор u_i может включать букву, которая следует сразу за вхождением наибольшего предыдущего фактора в позиции $|u_0...u_{i-1}|$; это сводится к расширению фактора, встречавшегося ранее. Предыдущие вхождения факторов можно выбирать из факторов u_0, ..., u_{i-1}, или из всех факторов слова $u_0...u_{i-1}$ (чтобы избежать перекрытия вхождений), или из всех факторов, встречавшихся прежде. В результате получается широкое разнообразие программ сжатия текста, основанных на этом методе.

При проектировании алгоритмов обработки слов факторизация используется также с целью уменьшить объем онлайновой обработки, сохранив то, что уже было сделано при рассмотрении предыдущих вхождений факторов.

Преобразование Барроуза–Уилера слова x – это обратимое отображение, которое переводит $x \in A^k$ в $\mathrm{BW}(x) \in A^k$. Смысл его в том, чтобы сгруппировать буквы, имеющие одинаковый контекст в x. Кодирование производится следующим образом. Рассмотрим отсортированный список ротаций x (сопряженных слов). Тогда $\mathrm{BW}(x)$ – это слово, составленное из последних букв отсортированных ротаций, находящихся в последнем столбце соответствующей таблицы.

Например, ротации слова `banana` показаны на рисунке ниже слева, а отсортированный список ротаций – справа. Таким образом, BW(`banana`) = `nnbaaa`.

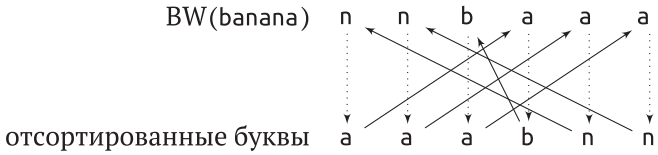
0	b	a	n	a	n	a	5	a	b	a	n	a	n
1	a	n	a	n	a	b	3	a	n	a	b	a	n
2	n	a	n	a	b	a	1	a	n	a	n	a	b
3	a	n	a	b	a	n	0	b	a	n	a	n	a
4	n	a	b	a	n	a	4	n	a	b	a	n	a
5	a	b	a	n	a	n	2	n	a	n	a	b	a

Это отображение переводит сопряженные слова в одно. Если выбирать слово Линдона в качестве представителя класса примитивного слова, то отображение становится биективным. Чтобы восстановить исходное слово x, отличное от слова Линдона, достаточно сохранить позицию первой буквы x в BW(x).

Главное свойство этого преобразования состоит в том, что относительный порядок вхождений данной буквы в BW(x) и в отсортированный список всех букв одинаков. Это позволяет декодировать BW(x).

Чтобы проделать это для слова `nnbaaa` из примера выше, сначала отсортируем буквы – получится `aaabnn`. Зная, что первая буква исходного слова находится в позиции 2 слова `nnbaaa`, мы можем начать декодирование: первой буквой является `b`, за ней следует буква `a` в той же позиции 2 слова `aaabnn`. Это третье вхождение `a` в слово `aaabnn`, соответствующее ее третьему вхождению в слово `nnbaaa`, за которым следует `n`. И так далее.

Процесс декодирования похож на обход цикла в графе ниже, начиная с правильной буквы. Если начать с другой буквы, то получится слово, сопряженное начальному.



Соглашения о псевдокоде алгоритмов

Применяемый нами стиль языка описания алгоритмов близок к реальным языкам программирования, но уровень абстракции выше. Мы придерживаемся следующих соглашений:

- отступами обозначается блочная структура составных предложений;
- строки кода нумеруются, чтобы на них можно было сослаться в тексте;
- символом ▷ начинается комментарий;
- чтобы обозначить доступ к атрибуту объекта, указывается имя атрибута, а за ним в квадратных скобках – идентификатор, ассоциированный с объектом;
- переменная, представляющая объект (таблицу, очередь, дерево, слово, автомат), является указателем на этот объект;
- аргументы передаются процедурам и функциям по значению;

- переменные, употребляемые внутри процедур и функций, считаются локальными, если явно не оговорено противное;
- булевы выражения вычисляются лениво слева направо;
- запись вида $(m_1, m_2, \ldots) \leftarrow (exp_1, exp_2, \ldots)$ – сокращенное обозначение последовательности присваиваний $m_1 \leftarrow exp_1$, $m_2 \leftarrow exp_2$, ...

В качестве примера ниже описан алгоритм Trie. Он порождает префиксное дерево по конечному множеству слов (словарю) X. В цикле **for**, занимающем строки 2–10, рассматривается каждое слово из X, а в цикле **for** в строках 4–9 буквы этого слова вставляются в структуру данных одна за другой. По завершении внутреннего цикла последнее рассмотренное состояние t, завершающее путь из начального состояния и помеченное текущим словом, делается заключительным (строка 10).

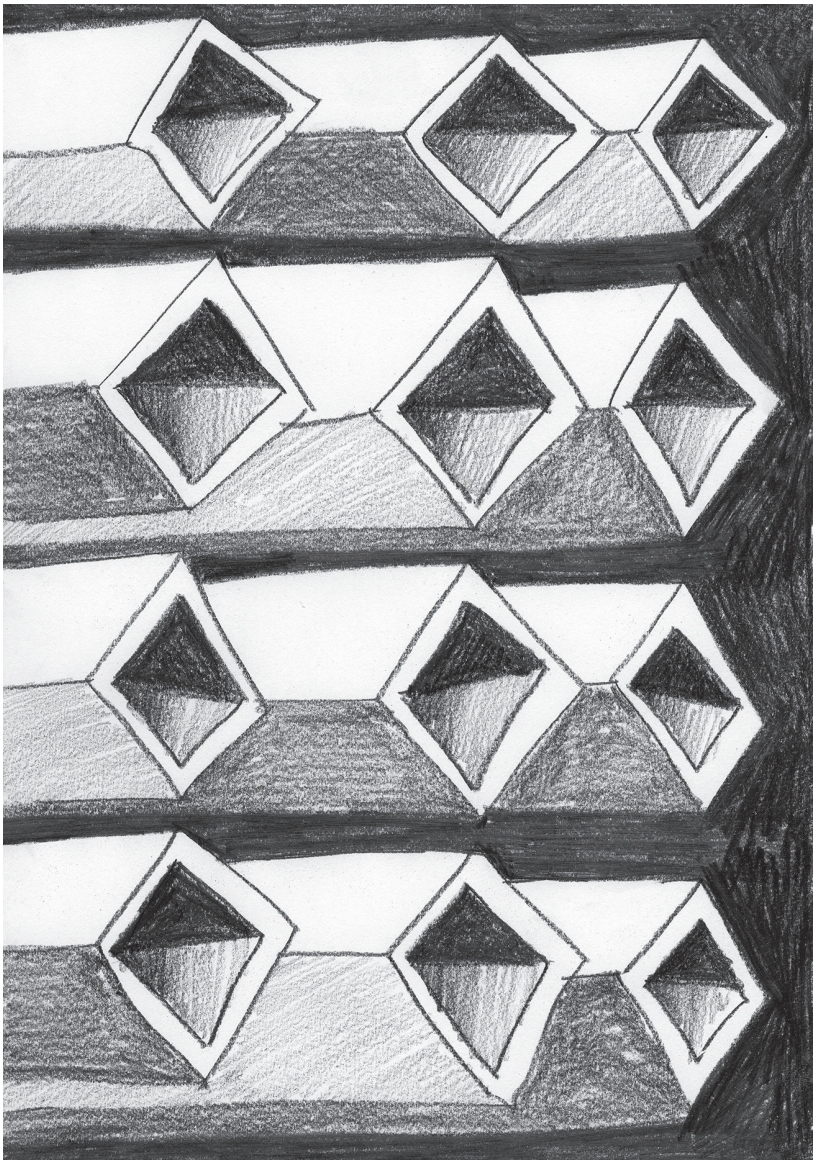
```
Trie(X конечное множество слов)
 1  M ← New-automaton()
 2  for каждой строки x ∈ X do
 3      t ← initial(M)
 4      for каждой буквы a слова x последовательно do
 5          p ← Target(t, a)
 6          if p = nil then
 7              p ← New-state()
 8              Succ[t ] ←Succ[t ] ∪ {(a, p)}
 9          t ← p
10      terminal[t] ← true
11  return M
```

Примечания

При изложении основных понятий, связанных со словами, мы следовали работе [74]. Этот материал можно найти и в других учебниках по алгоритмам обработки текста, например Crochemore and Rytter [96], Gusfield [134], Crochemore and Rytter [98] и Smyth [228], а также в книгах, посвященных более широкой теме комбинаторики слов, например Lothaire [175–177], или в пособии Berstel and Karhumäki [34].

Глава 2

Комбинаторные задачи



1. Стрингологическое доказательство малой теоремы Ферма

В 1640 году великий французский математик Пьер де Ферма доказал следующее утверждение:

Если p – простое число и k – любое натуральное число, то p делит $k^p - k$.

Это утверждение называется *малой теоремой Ферма*. Например:

7 делит $2^7 - 2$, а 101 делит $10^{101} - 10$.

Вопрос. Доказать малую теорему Ферма, пользуясь только стрингологическими рассуждениями.

[**Указание:** подсчитать количество классов сопряженности слов длины p.]

Решение
Для доказательства рассмотрим классы сопряженности слов одинаковой длины. Например, класс сопряженности, содержащий слово `aaaba`, представляет собой множество $C(\texttt{aaaba}) = \{\texttt{aaaab},\texttt{aaaba},\texttt{aabaa},\texttt{abaaa},\texttt{baaaa}\}$. Следующий факт вытекает из леммы о примитивности.

Наблюдение. Класс сопряженности примитивного слова w содержит ровно $|w|$ различных слов.

Рассмотрим множество слов длины p, где p – простое число, над алфавитом $\{1,2,\ldots,k\}$, и пусть $S_k(p)$ – его подмножество, состоящее из примитивных слов. Из k^p слов только k не являются примитивными, а именно слова вида a^p для некоторой буквы a. Поэтому мы приходим к следующему наблюдению.

Наблюдение. Количество $|S_k(p)|$ примитивных слов длины p, где p – простое число, над алфавитом из k букв равно $k^p - k$.

Поскольку слова, входящие в состав $S_k(p)$, примитивные, класс сопряженности каждого из них имеет размер p. Классы сопряженности разбивают $S_k(p)$ на множества размера p, откуда следует, что p делит $k^p - k$ и что всего существует $(k^p - k)/p$ классов. Тем самым теорема доказана.

Примечания
Если для слова $w = u^q$ длины n над алфавитом из k букв существует примитивный корень u длины d, то имеем $n = qd$, и класс сопряженности w содержит d элементов. Если d пробегает множество делителей n, то получаем равенство $k^n = \sum\{d\psi_k(n/d) : d \text{ является делителем } n\}$, где $\psi_k(m)$ – количество классов примитивных слов длины m. Это доказывает теорему в случае, когда n простое. Дополнительные детали см. в книге Lothaire [175, глава 1].

2. Простой случай проверки однозначности декодирования

Множество $\{w_1, w_2, \ldots, w_n\}$ слов, составленных из букв алфавита A, называется (однозначно декодируемым) кодом, если для любых двух последовательностей (обозначаемых как слова) $i_1 i_2 \ldots i_k$ и $j_1 j_2 \ldots j_l$ индексов, выбранных из множества $\{1, 2, \ldots, n\}$, имеем

$$i_1 i_2 \ldots i_k \neq j_1 j_2 \ldots j_l \Rightarrow w_{i_1} w_{i_2} \ldots w_{i_k} \neq w_{j_1} w_{j_2} \ldots w_{j_l}.$$

Иными словами, если определить морфизм h из $\{1, 2, \ldots, n\}^*$ в A^*, положив $h(i) = w_i$ для $i \in \{1, 2, \ldots, n\}$, то приведенное выше условие означает, что этот морфизм инъективен.

Для произвольного целого числа n неизвестен алгоритм проверки свойства однозначности декодирования, работающий за линейное время. Но при $n = 2$ ситуация предельно проста: достаточно проверить, коммутируют ли два кодовых слова, т. е. выполнение равенства $w_1 w_2 = w_2 w_1$.

Вопрос. Доказать, что множество $\{x, y\}$ является кодом тогда и только тогда, когда $xy \neq yx$.

Решение

Идея доказательства приведена выше как следствие из леммы о периодичности. Ниже приведено замкнутое доказательство по индукции.

Если $\{x, y\}$ – код, то утверждение следует из самого определения. Обратно, предположим, что $\{x, y\}$ – не код, и докажем равенство $xy = yx$. Это равенство заведомо имеет место, если одно из слов пустое, поэтому осталось рассмотреть случай, когда оба слова не пусты.

Доказательство проведем индукцией по длине $|xy|$. Базой индукции является простой случай $x = y$, для которого равенство, очевидно, имеет место.

Предположим, что $x \neq y$. Тогда одно из слов является собственным префиксом другого, и без ограничения общности можно предположить, что x – собственный префикс y: $y = xz$ для некоторого непустого слова z. Тогда $\{x, z\}$ – не код, потому что две разные конкатенации x и y, порождающие одно и то же слово, переходят в две разные конкатенации x и z, порождающие одно слово.

Предположение индукции применимо, потому что $|xz| < |xy|$, и из нее вытекает, что $xz = zx$. Следовательно, $xy = xxz = xzx = yx$, и, значит, равенство имеет место для x и y, что и завершает доказательство.

Примечания

Похожее доказательство показывает, что $\{x, y\}$ не является кодом, если $x^k = y^l$ для двух целых положительных k и l.

Мы не знаем, существует ли специальный тест на однозначность декодирования для трех слов, выражаемый в терминах фиксированного набора неравенств. Для конечного числа слов эффективный алгоритм полиномиальной сложности, основанный на теории графов, приведен в задаче 52.

3. Магические квадраты и слово Туэ–Морса

Наша цель – построение магических квадратов с помощью бесконечного слова Туэ–Морса **t** над двоичным алфавитом {0,1} (вместо {a,b}). В роли слова **t** выступает $\mu^{\infty}(0)$, получаемое повторным применением морфизма μ, определенного как $\mu(0) = 01$ и $\mu(1) = 10$:

$\mathbf{t} = 01101001100101101001\ldots.$

Массив S_n размера $n \times n$, где $n = 2^m$ (m – положительное натуральное число), определяется следующим образом:

$$S_n[i, j] = \mathbf{t}[k](k + 1) + (1 - \mathbf{t}[k])(n^2 - k),$$

где $0 \le i, j < n$ и $k = i.n + j$. Массив S_4 выглядит так:

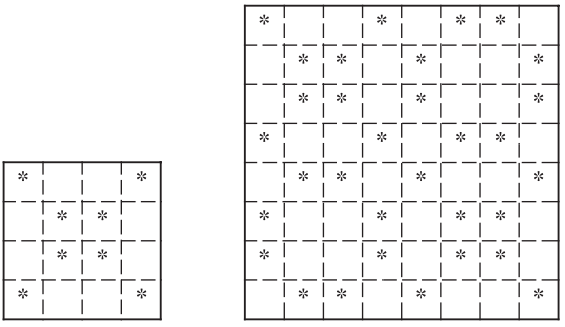
16	2	3	**13**
5	**11**	**10**	8
9	**7**	**6**	12
4	14	15	**1**

Этот массив является магическим квадратом, потому что содержит все целые числа от 1 до 16 и сумма элементов в каждой строке, в каждом столбце и на обеих диагоналях равна 34.

Вопрос. Доказать, что массив S_n размера $n \times n$ является магическим квадратом для любой натуральной степени 2.

Решение

Чтобы понять, как устроен массив S_n, определим T_n – 2-мерное слово Туэ–Морса формы $n \times n$, где $n = 2^m$, следующим образом: $T_n[i, j] = \mathbf{t}[i.n + j]$, $0 \le i, j < n$. На рисунке ниже показаны T_4 и T_8 – $*$ обозначает 0, а пустое место 1.



Заметим, что таблица T_n обладает двумя простыми свойствами:

(i) каждая строка и каждый столбец составлены из блоков `0110` и `1001`;
(ii) обе главные диагонали однородны, т. е. содержат только `0` или только `1` (звездочки и пустые клетки на рисунке).

Из определения ясно, что матрица S_n размера $n \times n$ содержит все целые числа от 1 до n^2. Чтобы доказать, что это магический квадрат, мы должны показать, что сумма элементов в каждой строке, в каждом столбце и на обеих диагоналях одинакова и равна $n/2(n^2 + 1)$.

Доказательство для строк. Согласно свойству (i), каждый блок в строке имеет тип `0110` или `1001`. Рассмотрим блок `0110`, первый элемент которого является k-м элементом массива. Тогда

$$S[k, k+1, k+2, k+3] = [n^2 - k, k+2, k+3, n^2 - k - 3],$$

сумма этих чисел равна $2n^2 + 2$. Если тип блока отличен от `0110`, то он состоит из чисел $[k + 1, n^2 - k - 1, n^2 - k - 2, k + 4]$, сумма которых точно такая же. Поскольку в строке имеется $n/4$ таких блоков, то сумма всех их вкладов равна

$$\frac{n}{4} \cdot (2n^2 + 2) = \frac{n}{2} \cdot (n^2 + 1),$$

что и требовалось доказать.

Для столбцов доказательство проводится аналогично.

Доказательство для диагоналей. Рассмотрим только диагональ из угла $(0, 0)$ в угол $(n - 1, n - 1)$, для другой диагонали всё аналогично. На этой диагонали расположены (в порядке снизу вверх) элементы $1, 1 + (n + 1), 1 + 2(n + 1), \ldots, 1 + (n - 1)(n + 1)$. Их сумма равна

$$n + (n+1)\sum_{i=0}^{n-1} i = n + (n+1)\frac{n}{2}(n-1) = \frac{n}{2}(n^2 + 1),$$

что и требовалось доказать.

Таким образом, мы доказали, что S_n – магический квадрат.

Примечания

Дополнительные сведения о магических квадратах и их долгой истории можно найти в Википедии по адресу https://en.wikipedia.org/wiki/Magic_square.

4. Последовательность Ольденбургера–Колакоски

Последовательность Ольденбургера–Колакоски, которую мы будем обозначать **K**, – самогенерирующаяся бесконечная последовательность символов {1,2}. Формально говоря, она является результатом кодирования самой себя

длинами серий. Это одна из самых странных последовательностей на свете. Несмотря на простоту генерации, ее поведение выглядит случайным.

Под блоком букв в слове мы понимаем серию букв, т. е. максимальный фактор, состоящий из вхождений одной и той же буквы. Операция *blocks*(*S*) заменяет каждый блок слова *S* его длиной. Например,

$$blocks(2111221222) = 13213.$$

K – однозначно определенная последовательность над алфавитом {1,2}, начинающаяся с 2 и удовлетворяющая условию $blocks(\mathbf{K}) = \mathbf{K}$.

Замечание. Обычно последовательность начинается с 1, но в данном случае удобнее начать с 2. На самом деле одна последовательность получается из другой удалением первого вхождения 1.

Вопрос. Доказать, что можно в онлайновом режиме сгенерировать первые n символов последовательности **K** за время $O(n)$ в памяти размером $O(\log n)$.

[**Указание:** порождайте **K**, повторно применяя отображение $h = blocks^{-1}$, начиная с 2.]

Самый интересный аспект этой задачи – очень небольшой объем памяти, необходимый для генерирования **K**.

Решение

Согласно определению h, $h(x) = y$ тогда и только тогда, когда y начинается с 2 и $blocks(y) = x$.

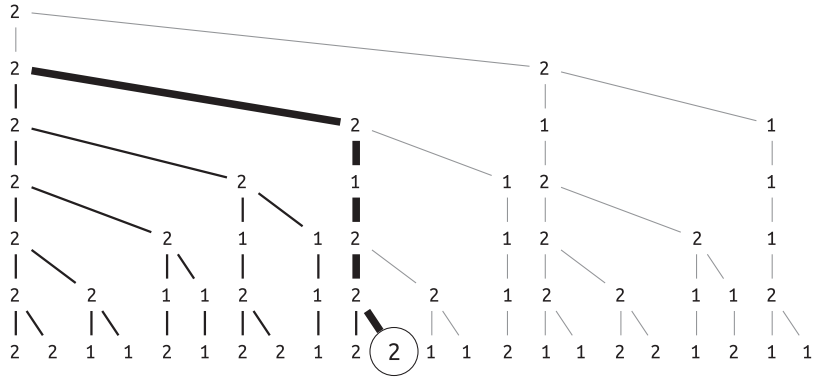
Как построить $h^{k+1}(2)$, **зная** $h^k(2)$**.** Положим $x = h^k(2)$. Тогда $y = h^{k+1}(2) = h(x)$ получается заменой буквы $x[i]$ в x либо $x[i]$ вхождениями буквы 2, если i четно, либо $x[i]$ вхождениями буквы 1, если i нечетно. Слово **K** равно пределу $\mathbf{K}_k = h^k(2)$ при k, стремящемся к бесконечности. Первые итерации h дают

$$\begin{cases} h(2) = 22 \\ h^2(2) = 2211 \\ h^3(2) = 221121 \\ h^4(2) = 221121221 \end{cases}$$

Оставляем читателю формальное доказательство следующего утверждения.

Наблюдение. $n = O(\log|\mathbf{K}_n|)$ и $\sum_{k=0}^{n}|\mathbf{K}_k| = O(|\mathbf{K}_n|)$.

Обозначим T дерево разбора, ассоциированное с $\mathbf{K}_n$. Его листья соответствуют позициям в $\mathbf{K}_n$. Для позиции i, $0 \le i < |\mathbf{K}_n|$, $RightBranch(i)$ определяется как путь из i-го листа вверх до первого узла на самой левой ветви дерева (см. рисунок).



На рисунке изображено дерево разбора $\mathbf{K}_6 = h^6(2)$. Каждый уровень представляет $h^k(2)$ для $k = 0, 1, \ldots, 6$. Правая ветвь RightBranch позиции 10 (обведенной кружком) состоит из жирных ребер и их концов. Она начинается в узле и поднимается вверх до первого узла на самой левой ветви.

С каждым узлом на ветви RightBranch ассоциирован один бит информации: четность числа узлов слева от него на том же уровне.

Если для каждого узла известна его метка и является ли он левым потомком родителя, то, в силу приведенного выше наблюдения, зная *RightBranch*(i), можно вычислить символ в позиции ($i + 1$) и всю ветвь *RightBranch*($i + 1$) в памяти логарифмического объема за амортизированное постоянное время (поскольку длины путей логарифмически зависят от i, а размер всего дерева растет линейно). Процесс следующим образом применяется к суффиксу RightBranch. Поднимаемся вверх по дереву до первого левого потомка, затем спускаемся по правой ветви, исходящей из его родителя, и продолжаем, пока не достигнем следующего листа. По сути дела, мы поднимаемся до самого низкого общего предка листьев i и $i + 1$, и в некотором смысле каждую итерацию можно рассматривать как внутренний обход дерева разбора с использованием памяти небольшого объема.

Ветвь RightBranch может расти вверх, как в случае перехода от *RightBranch*(13) к *RightBranch*(14) в нашем примере. Это верхнеуровневое описание алгоритма, технические детали мы опустили.

Примечания

Последовательность Ольденбургера–Колакоски, которую часто называют просто последовательностью Колакоски, была открыта в работе Ольденбургера [197], а затем популяризирована в работе Kolakoski [166]. Она является примером гладкого слова (см. [46]). Наш набросок алгоритма – вариант алгоритма, предложенного в работе Nilsson [195]; см. также https://en.wikipedia.org/wiki/Kolakoski_sequence.

5. Бесквадратная игра

Нетривиальным квадратом называется слово над алфавитом A вида uu, где $|u| > 1$. Слово называется нечетным квадратом, если дополнительно число $|u|$ нечетно.

Бесквадратная игра длины n над алфавитом A разыгрывается двумя игроками, Анной и Беном. В начале игры имеется пустое слово w, а игроки попеременно продолжают его, добавляя в конец по одной букве. Игра заканчивается, когда длина очередного слова стала равна n или образовался нетривиальный квадрат. Предположим, что первым ходит Бен и что n четно. Анна выигрывает, если в финальном слове нет нетривиальных квадратов. В противном случае выигрывает Бен.

Игра без нечетных квадратов. В этой упрощенной версии игры Анна выигрывает, если не было создано ни одного нечетного квадрата. Опишем выигрышную стратегию Анны для алфавита $A = \{0,1,2\}$. Анна никогда не повторяет последний ход Бена, а если Бен повторил последний ход Анны, то она не повторяет его предыдущий ход.

Для этого Анна запоминает пару (b, a), где a – буква, добавленная на ее предыдущем ходе, а b – буква, добавленная на предыдущем ходе Бена. Иначе говоря, слово w четной длины и после первого хода имеет вид $w = vba$. Затем Бен добавляет c, а Анна отвечает добавлением d, получая слово $w = vbacd$, где

$$d = \begin{cases} a, & \text{если } c \neq a, \\ 3 - b - a & \text{в противном случае} \end{cases}.$$

Анна ведет себя как детерминированный конечный автомат с шестью состояниями. Возможная последовательность ходов, начинающаяся с 12 и приводящая к выигрышу Анны, имеет вид

```
12122010021220.
```

Вопрос. (A) Доказать, что Анна всегда выигрывает у Бена в игре без нечетных квадратов любой длины n.
(B) Описать выигрышную стратегию Анны в бесквадратной игре над алфавитом размера 9.

[**Указание:** для доказательства (A) покажите, что w не содержит нечетных квадратов. Для решения (B) объедините простую стратегию для четных квадратов с описанной выше стратегией.

Решение
(A). Мы докажем утверждение (A), приведя к противоречию предположение о том, что при выигрышной стратегии Анны слово w (история игры) может содержать нечетный квадрат uu ($|u| > 1$).

Случай 1. Первая буква uu является результатом хода Бена.

Квадрат имеет вид

$$uu = b_0a_1b_1a_2b_2\ldots a_kb_k\ a'_0b'_1a'_1b'_2a'_2\ldots b'_ka'_k,$$

где буквы b_i и b'_j соответствуют ходам Бена, а остальные – ходам Анны.

Поскольку uu – квадрат, мы имеем $b_0 = a'_0$, $a_1 = b'_1$, …, $b_k = a'_k$. В силу стратегии Анны, $a_1 \neq b_0$, $a_2 \neq b_1$ и т. д., то есть любые две соседние буквы в uu различны. В частности, отсюда следует, что Бен никогда не повторяет последний ход Анны в uu.

Следовательно, все ходы Анны одинаковы, т. е. все буквы a_i, a'_j одинаковы. Поэтому $a_k = a'_k$, но в то же время $a'_k = bk$, поскольку uu – квадрат. Отсюда следует, что $b_k = a_k$, т. е. Бен повторяет последний ход Анны. Противоречие! Тем самым этот случай доказан.

Случай 2. Первая буква uu является результатом хода Анны.

Квадрат имеет вид

$$uu = a_0b_1a_1b_2a_2\ldots b_ka_k\ b'_0a'_1b'_1a'_2b'_2\ldots a'_kb'_k,$$

где, как и раньше, буквы b_i, b'_j соответствуют ходам Бена, а остальные – ходам Анны.

Как и в предыдущем случае, можно доказать, что Бен всегда делает ход, отличный от последнего хода Анны, и единственное возможное исключение – $a_k = b'_0$. Если так, то $a'_1 \neq b_k$, т. к. $a'_1 = 3 - a_k - b_k$, а затем $a'_1 = a'_2 = \ldots = a'_k$. Следовательно, $a'_k \neq b_k$, но в то же время $a'_k = b_k$, поскольку uu – квадрат. Противоречие!

Если $a_k \neq b'_0$, то все ходы Бена отличаются от ходов Анны, которая, следовательно, всегда делает один и тот же ход в uu. Это приводит к противоречию так же, как в случае 1.

Тем самым мы завершили доказательство и показали, что стратегия Анны выигрышная.

(B). Если в игре учитываются нетривиальные четные квадраты над алфавитом {0,1,2}, то выигрышная стратегия Анны крайне проста: на своем k-м ходе она добавляет k-ю букву произвольного (заранее фиксированного) бесквадратного слова над тем же алфавитом.

Объединяя простые стратегии выигрыша в игре без нетривиальных четных и нечетных квадратов (применяя их одновременно), Анна получает выигрышную стратегию, позволяющую избежать любых нетривиальных квадратов над алфавитом из 9 букв. Теперь алфавит состоит из пар (e, e') букв {0,1,2}. История игры представляется словом вида $w = (e_1, e'_1)(e_2, e'_2)\ldots(e_k, e'_k)$, для которого $e_1e_2\ldots e_k$ не содержит нечетных квадратов, а $e'_1e'_2\ldots e'_k$ не содержит нетривиальных четных квадратов.

Примечания

Представленное выше решение задачи об игре описано в работе [132], в которой применение более сложных рассуждений позволило уменьшить число букв до 7. Однако в работе Kosinski et al. [169] была отмечена ошибка в доказательстве, но удалось уменьшить число букв до 8.

6. Слова Фибоначчи и фибоначчиева система счисления

Обозначим $r(m)$ представление Фибоначчи неотрицательного целого числа m. Это слово x длины l над алфавитом {0,1}, кончающееся единицей всегда, кроме случая $m = 0$, в котором не встречается двух соседних единиц и для которого имеет место соотношение $m = \sum_{i=0}^{l-1} x[i] \cdot F_{i+2}$, где F_{i+2} – $(i + 2)$-е число Фибоначчи (напомним, что $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2$ и т. д.).

Например, $r(0) = \texttt{0}$, $r(1) = \texttt{1}$, $r(2) = \texttt{01}$, $r(3) = \texttt{001}$, $r(4) = \texttt{101}$, $r(5) = \texttt{0001}$, $r(6) = \texttt{1001}$, $r(7) = \texttt{0101}$.

Отметим, что обычное позиционное представление Фибоначчи целого числа m имеет вид $r(m)^{\mathrm{R}}$, т. е. совпадает с $r(m)$, записанным в обратном порядке. Также отметим, что кодирование Фибоначчи, применяемое для преобразования целого числа m в поток данных, имеет вид $r(m)\texttt{1}$ и завершается парой 11, чтобы можно было произвести декодирование.

Вопрос. Доказать, что последовательность первых цифр представлений Фибоначчи натуральных чисел является бесконечным словом Фибоначчи, если сопоставить буквам цифры: a → 0, b → 1.

Обозначим pos(k, c), где $k > 0$, позицию k-го вхождения буквы c в бесконечное слово Фибоначчи **f**.

Вопрос. Покажите, как вычислить позицию k-го вхождения буквы a в бесконечное слово Фибоначчи **f** за время $O(\log k)$. Тот же вопрос для буквы b.

[**Указание:** докажите справедливость следующих формул: $r(pos(k, \texttt{a})) = \texttt{0} \cdot r(k - 1)$ и $r(pos(k, \texttt{b})) = \texttt{10} \cdot r(k - 1)$.]

Решение

Чтобы понять, как устроены представления Фибоначчи, рассмотрим прямоугольник R_n, строками которого являются представления первых $|fib_n| = F_{n+2}$ натуральных чисел. Представления при необходимости дополняются справа символами 0, чтобы всего было n цифр. Рекуррентное построение прямоугольников показано на рисунке ниже.

$$R_1 = \begin{array}{|c|}\hline 0 \\ 1 \\ \hline \end{array} \qquad R_2 = \begin{array}{|cc|}\hline 0&0 \\ 1&0 \\ 0&1 \\ \hline \end{array} \qquad R_3 = \begin{array}{|cc:c|}\hline 0&0&0 \\ 1&0&0 \\ 0&1&0 \\ \hdashline 0&0&1 \\ 1&0&1 \\ \hline \end{array} \qquad R_{n+2} = \begin{array}{|c:c|}\hline \multirow{2}{*}{R_{n+1}} & 0 \\ & \vdots \\ \hdashline \multirow{2}{*}{R_n} & 0\ 1 \\ & \cdots \\ \hline \end{array}$$

Ответ на первый вопрос. Строками прямоугольников R_1 и R_2 являются представления первых $|fib_1|$ и $|fib_2|$ целых чисел, взятых в порядке возрастания. Покажем, что это справедливо и для R_{n+2}, $n > 0$. Действительно, первые $|fib_{n+1}|$ строк прямоугольника R_{n+2} – это дополненные нулями представления первых $|fib_{n+1}|$ целых чисел, в силу рекуррентного построения. Следующие $|fib_n|$ строк – представления вида $x \cdot 01$ (они не могут заканчиваться на 11). Поскольку x – строка R_n, то снова, в силу рекуррентного построения, эти строки представляют следующие $|fib_n|$ целых чисел, так что интересующее нас свойство доказано для R_{n+2}.

Из рекуррентного построения ясно, что последовательность первых цифр (в пределе – первый столбец) соответствует бесконечному слову Фибоначчи. Тем самым ответ на первый вопрос получен.

Ответ на второй вопрос. Пределом таблиц R_n является бесконечная таблица R_∞ представлений Фибоначчи всей последовательности натуральных чисел в порядке возрастания. В каждой строке буквы справа от самого правого вхождения 1 – незначащие цифры, равные нулю.

Нули в первом столбце R_∞ соответствуют буквам a в слове Фибоначчи. Строки, начинающиеся с нуля, имеют вид

$$0 \cdot x_0,\ 0 \cdot x_1,\ 0 \cdot x_2,\ \ldots,$$

где

$$x_0,\ x_1,\ x_2,\ \ldots$$

– представления последовательных натуральных чисел.

Таким образом, k-й нуль соответствует x_{k-1} и встречается в позиции $0 \cdot x_{k-1}$, откуда $r(pos(k, \mathtt{a})) = \mathtt{0} \cdot r(k-1)$.

Аналогично получаем $r(pos(k, \mathtt{b})) = \mathtt{10} \cdot r(k-1)$, поскольку все строки, содержащие 1 в первом столбце R_∞, на самом деле начинаются с 10.

Поэтому вычисление k-го вхождения буквы в слово Фибоначчи сводится к вычислению представления Фибоначчи целого числа и выполнению обратной операции. То и другое занимает время $O(\log k)$, что и требуется.

	0	a	0	0	0	0	0	.
	1	b	1	0	0	0	0	.
	2	a	0	1	0	0	0	.
Позиции	3	a	0	0	1	0	0	.
	4	b	1	0	1	0	0	.
5-го вхождения a:	5	a	0	0	0	1	0	.
$(0 \cdot 101)_F = 7$	6	b	1	0	0	1	0	.
	7	a	0	1	0	1	0	.
	8	a	0	0	0	0	1	.
4-го вхождения b:	9	b	1	0	0	0	1	.
$(10 \cdot 001)_F = 9$	10	a	0	1	0	0	1	.
	11	a	0	0	1	0	1	.
	12	b	1	0	1	0	1	.
	.	.	.	.	.	.	.	

Примечания
Эта задача взята из работы Rytter [216].

7. Игра Витхоффа и слово Фибоначчи

Игра Витхоффа – стратегическая игра двух игроков, вариант игры ним. Имеются две кучки фишек, первоначально не пустых. Игроки по очереди берут положительное число фишек из одной кучки или одинаковое количество фишек из обеих кучек. Выигрывает игрок, забирающий при своем ходе все оставшиеся фишки.

Позиция игры описывается парой натуральных чисел (m, n), $m \leq n$, где m и n – число фишек в обеих кучках. Отметим, что позиции $(0, n)$ и (n, n), $n > 0$, выигрышные. Наименьшая проигрышная позиция – $(1, 2)$, а все позиции вида $(m + 1, m + 2)$, $(1, m)$ и $(2, m)$, $m > 0$, выигрышные.

Известно, что проигрышные позиции описываются регулярным правилом, в котором используется золотое сечение.

Вопрос. Существует ли тесная связь между игрой Витхоффа и бесконечным словом Фибоначчи?

Решение
Проигрышные позиции в игре Витхоффа тесно связаны со словом Фибоначчи. Обозначим *WytLost* множество проигрышных позиций. Оно содержит пары вида (m, n), $0 < m < n$:

$$WytLost = \{(1,2), (3,5), (4,7), (6,10), (8,13),...\}.$$

Пусть (m_k, n_k) обозначает k-ю пару в лексикографическом порядке, тогда:

$$WytLost = \{(m_1, n_1), (m_2, n_2), (m_3, n_3), ...\},$$

где $m_1 < m_2 < m_3 < ...$ и $n_1 < n_2 < n_3 < ....$

Обозначим $pos(k, c)$, $k > 0$, позицию k-го вхождения буквы c в бесконечное слово Фибоначчи **f**. Имеет место следующее свойство, связывающее **f** с игрой Витхоффа.

Факт 1. $m_k = pos(k, \mathtt{a}) + 1$ и $n_k = pos(k, \mathtt{b}) + 1$.

Пусть $M = \{m_1, m_2, m_3, ...\}$ и $N = \{n_1, n_2, n_3, ...\}$. Следующий факт хорошо известен, и здесь мы его доказывать не будем.

Факт 2.
(i) $M \cap N = \emptyset$ и $M \cup N = \{1, 2, 3, ...\}$.
(ii) $n_k = m_k + k$ для любого $k > 0$.

Для доказательства факта 1 мы воспользуемся фактом 2. Достаточно доказать, что оба свойства (i) и (ii) имеют место для множеств $M' = \{pos(k, \mathtt{a}) + 1 : k > 0\}$ и $N' = \{pos(k, \mathtt{b}) + 1 : k > 0\}$.

Свойство (i), очевидно, выполняется, а справедливость свойства (ii) вытекает из указания к задаче 6, которое было там же доказано:

$r(pos(k, \mathtt{a})) = \mathtt{0} \cdot r(k-1)$ и $r(pos(k, \mathtt{b})) = \mathtt{10} \cdot r(k-1)$,

где $r(i)$ – представление Фибоначчи натурального числа i. Чтобы показать, что $pos(k, \mathtt{b}) + 1 - pos(k, \mathtt{a}) + 1 = k$, достаточно показать, что для любого представления Фибоначчи x положительного целого числа имеет место равенство $(\mathtt{10}x)_F - (\mathtt{0}x)_F = (x)_F + 1$, где $(y)_F$ – число i, для которого $r(i) = y$. Но это прямо следует из определения представления Фибоначчи.

Примечания

Эта игра была описана в работе Wythoff [240] как вариант игры ним. Витхофф установил связь между проигрышными позициями и золотым сечением, см. https://en.wikipedia.org/wiki/Wythoff%27s_game. Именно, k-я проигрышная позиция (m_k, n_k), $k > 0$, описывается формулами $m_k = \lfloor k\Phi \rfloor$, $n_k = \lfloor k\Phi^2 \rfloor = m_k + k$. Он также показал, что последовательности m_k и n_k являются дополнительными, т. е. каждое целое положительное число встречается ровно в одной из них.

Еще одно следствие этих свойств – удивительный алгоритм генерирования бесконечного числа Фибоначчи (или его префиксов сколь угодно большой длины). Начнем с бесконечного слова $Fib = \sqcup^\infty$, состоящего из одних пробелов, и выполним следующие действия:

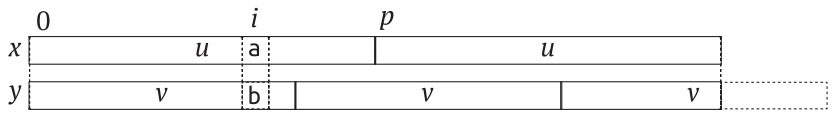
```
1  for k ← 1 to ∞ do
2      i ← наименьшая позиция ⊔ в Fib
3      Fib[i] ← a
4      Fib[i + k] ← b
```

В силу свойств (i) и (ii) *Fib* будет словом Фибоначчи.

8. Различные периодические слова

В этой задаче мы изучим, сколько может быть различных периодических слов заданной длины. Для измерения степени различия применяется расстояние Хэмминга. Расстояние Хэмминга между двумя словами x и y одинаковой длины равно $\text{HAM}(x, y) = |\{j : x[j] \neq y[j]\}|$.

Будем рассматривать слово x с периодом p и слово y длины $|x|$ с периодом $q \leq p$ и предположим, что эти слова различаются хотя бы в одной позиции. Пусть i – позиция, в которой x и y различаются; например $x[i] = \mathtt{a}$, $y[i] = \mathtt{b}$. На рисунке ниже $x = u^2$, $|u| = p$, $|v| = q$.



Пример. Пусть $x = (\texttt{abaababa})^2$ с периодом 8, $y = (\texttt{abaaa})^3$ с периодом 5. Эти слова различны и различаются в нескольких позициях, а именно в позициях 4, 9, 11, 12 и 14.

	0	1	2	3	**4**	5	6	7	8	**9**	10	**11**	**12**	13	**14**	15
x	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a
y	a	b	a	a	a	a	b	a	a	a	a	b	a	a	a	a

Вопрос. Каково минимально расстояние Хэмминга между двумя различными периодическими словами одинаковой длины?

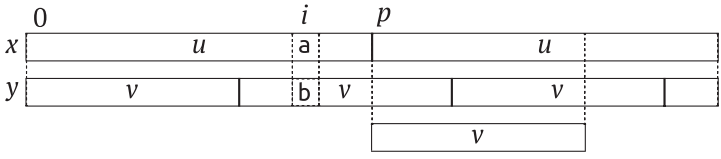
[**Указание:** рассмотрите различные случаи расположения позиции i относительно периодов p и q.]

Решение

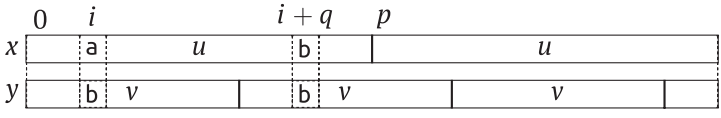
Поскольку x периодическое, его длина не меньше $2p$. Без ограничения общности предположим, что $x = x[0..2p-1] = u^2$. Из соображений симметрии можно также предположить, что позиция несовпадения i удовлетворяет условию $0 \le i < p$. Пусть $v = y[0..q-1]$ – префикс, длина которого равна периоду y. Отметим, что u и v – примитивные слова.

Например, слова aa и bb с периодом 1 различаются ровно в двух позициях, как и слова bbcabcbbcabc и abcabcabcabc, имеющие периоды 6 и 3 соответственно. На самом деле если p кратно q, т. е. $p = hq$ для некоторого целого положительного h, то ясно, что существует еще одно несовпадение в позиции $i + p$. Тогда $\mathrm{HAM}(x, y) \ge 2$.

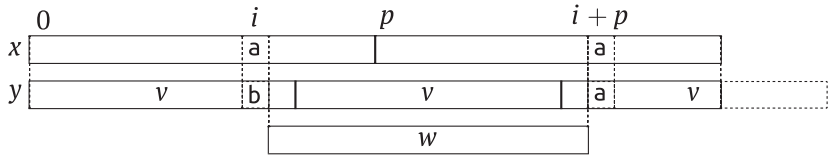
Если p не кратно q, то мы докажем это неравенство от противного. Предположим, что слова x и y совпадают всюду, кроме позиции i. Рассмотрим три случая, показанных на рисунках ниже.



Случай $i \ge q$. Слово v, будучи префиксом u, встречается в x и y в позиции p. Таким образом, оно является внутренним фактором v^2, что противоречит его примитивности, в силу леммы о примитивности.



Случай $i < q$ и $i + q < p$. Поскольку $y[i] = y[i + q] = x[i + q]$, имеем $x[i] \ne x[i + q]$. Тогда q не является периодом u, хотя вхождение u в позиции p имеет период q. Противоречие!



Случай $i < q$ **и** $i + q \geq p$**.** Сначала покажем, что слово $w = y[i + 1..i + p - 1]$ имеет период $p - q$. Действительно, для позиции j если $i < j < p$, то

$$y[j] = x[j] = x[j + p] = y[j + p] = y[j + p - q],$$

а если $p \leq j < i + q$, то

$$y[j] = y[j - q] = x[j - q] = x[j + p - q].$$

Таким образом, слово w длины $p - 1$ имеет период $p - q$ в дополнение к периоду q, который оно имеет, будучи фактором y длины большей, чем v. По лемме о периодичности, $\gcd(q, p - q)$ также является периодом, что противоречит примитивности v, поскольку $p - q < q$.

И наконец, если p не является кратным q, то имеем $\mathrm{HAM}(x, y) \geq 2$, как и раньше, что и завершает доказательство.

Примечания

Другое доказательство этого результата имеется в работе Amir et al. [12], а его развитие в разных направлениях можно найти в работе [9].

9. Вариации на тему слова Туэ–Морса

Пусть $\mathbf{c} = (c_0, c_1, c_2, \ldots)$ – наименьшая возрастающая последовательность целых положительных чисел, начинающаяся с 1 и удовлетворяющая условию

$$n \in \mathbf{C} \Leftrightarrow n/2 \notin \mathbf{C}, \qquad (*)$$

где $\mathbf{C}$ – множество элементов, входящих в последовательность $\mathbf{c}$. Первыми элементами $\mathbf{c}$ являются

1, 3, 4, 5, 7, 9, 11, 12, 13, 15, 16, 17, 19, 20, 21, 23, 25, 27, 28, 29, ...

Заметим, что все нечетные числа входят в последовательность и что длина промежутка между соседними элементами равна 1 или 2.

Вопрос. Какая связь имеется между последовательностью $\mathbf{c}$ и бесконечным словом Туэ–Морса $\mathbf{t}$?

Решение

Напомним, что слово Туэ–Морса $\mathbf{t}$ определено как $\mu^{\infty}(\mathtt{a})$, где морфизм μ, отображающий $\{\mathtt{a},\mathtt{b}\}$ в себя, определен следующим образом: $\mu(\mathtt{a}) = \mathtt{ab}$,

$\mu(\mathtt{b}) = \mathtt{ba}$. Обозначим *end-pos*($x$, y) множество конечных позиций вхождений слова x в слово y.

Ключевое свойство. Для целого положительного n

$$n \notin \mathbf{C} \Leftrightarrow n \in \textit{end-pos}(\mathtt{aa}, \tau) \cup \textit{end-pos}(\mathtt{bb}, \tau). \qquad (**)$$

Для иллюстрации этого свойства в таблице ниже показан префикс **t** и несколько первых элементов **C**, ассоциированных с ним (четные значения выделены полужирным шрифтом).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t	a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a	b	a
c	1			3	**4**	5		7		9	11	**12**	13			15	**16**	17

По определению, слово **t** удовлетворяет следующим соотношениям для $k > 0$:

(i) $\mathbf{t}[n] = \overline{\mathbf{t}[k]}$ и $\mathbf{t}[n-1] = \mathbf{t}[k]$, если $n = 2k+1$;

(ii) $\mathbf{t}[n] = \mathbf{t}[k]$ и $\mathbf{t}[n-1] = \overline{\mathbf{t}[k-1]}$, если $n = 2k$.

Тогда свойство (i) исключает эквиваленцию (**) для нечетных целых чисел, а свойство (ii) по индукции делает то же самое для четных чисел, что доказывает соотношение между **c** и **t**.

Примечания

Возвращаясь к данному выше эквивалентному определению слова Туэ–Морса с использованием четности числа единиц в двоичном представлении целых чисел, отметим, что свойство «$n \in \mathbf{C} \Leftrightarrow \nu(n)$ четно», где $\nu(n)$ – длина конечного блока нулей в двоичном представлении n, также характеризует последовательность **c**. (Заметим, что $\nu(n) = 0$ тогда и только тогда, когда n нечетно.)

10. Слова Туэ–Морса и суммы степеней

Для конечного множества натуральных чисел I обозначим $Sum_k(I) = \sum_{i \in I} i^k$. Для двух конечных множеств I и J натуральных чисел рассмотрим свойство $\mathbf{P}(n, I, J)$:

для любого k, $0 < k < n$, $Sum_k(I) = Sum_k(J)$,

которое исследуем с точки зрения множеств позиций в n-м слове Туэ–Морса τ_n длины 2^n. Точнее, нас интересуют множества

$$T_{\mathtt{a}}(n) = \{i : \tau_n[i] = \mathtt{a}\} \text{ и } T_{\mathtt{b}}(n) = \{j : \tau_n[j] = \mathtt{b}\}.$$

Например, для слова Туэ–Морса $\tau_3 = \mathtt{abbabaab}$ имеем

$$T_{\mathtt{a}}(3) = \{0,3,5,6\},\ T_{\mathtt{b}}(3) = \{1,2,4,7\}.$$

Свойство $\mathbf{P}(3, T_a(3), T_b(3))$ имеет место, поскольку справедливы следующие равенства:

$$0 + 3 + 5 + 6 = 1 + 2 + 4 + 7 = 14,$$
$$0^2 + 3^2 + 5^2 + 6^2 = 1^2 + 2^2 + 4^2 + 7^2 = 70.$$

Вопрос. Доказать, что свойство $\mathbf{P}(n, T_a(n), T_b(n))$ имеет место для любого целого $n > 1$.

Решение
Для натурального числа d обозначим $I + \{d\} = \{a + d : a \in I\}$. Простым вычислением доказывается следующий факт для любого d и любых множеств I, J.

Наблюдение. Предположим, что имеет место свойство $\mathbf{P}(n, I, J)$. Тогда имеют место и два других свойства:

$$\mathbf{P}(n, I + \{d\}, J + \{d\}) \text{ и } \mathbf{P}(n + 1, I \cup (J + \{d\}), J \cup (I + \{d\})).$$

Теперь доказательство сформулированного в вопросе утверждения проводится простой индукцией по n с использованием этого наблюдения и следующих рекуррентных соотношений, верных для всех $n > 1$:

$$T_a(n + 1) = T_a(n) \cup (T_b(n) + 2^n) \text{ и } T_b(n + 1) = T_b(n) \cup (T_a(n) + 2^n).$$

Примечания
Эта задача является частным случаем задачи Тарри-Эскотта; см. [6].

11. Сопряженные слова и ротации слов

Два слова x и y называются сопряженными, если существуют такие два слова u и v, что $x = uv$ и $y = vu$. Говорят также, что одно слово получено ротацией, или циклическим сдвигом другого. Например, слово abaab = aba·ab является сопряженным слову ababa = ab·aba. Ясно, что сопряженность является отношением эквивалентности между словами, но не согласована с произведением слов.

Ниже приведено семь слов, сопряженных aabaaba (слева), и три слова, сопряженных aabaabaab (справа).

```
a a a b a a b          b a a b a a b a a
 b a a a b a a          a b a a b a a b a
  a b a a a b a          a a b a a b a a b
   a a b a a a b
    b a a b a a a
     a b a a b a a
      a a b a a b a
```

Вопрос. Доказать, что два непустых слова одинаковой длины являются сопряженными тогда и только тогда, когда сопряженными являются их (примитивные) корни.

В примере выше слова aabaabaab $=$ (aab)^3 и baabaabaa $=$ (baa)^3 сопряженные, как и их корни aab и baa.

Более удивительное свойство сопряженных слов сформулировано в следующем вопросе.

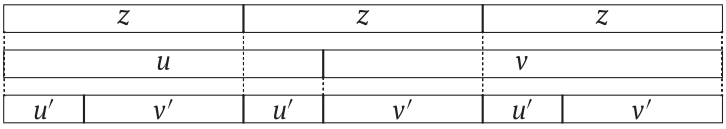
Вопрос. Доказать, что два непустых слова x и y являются сопряженными тогда и только тогда, когда $xz = zy$ для некоторого слова z.

В примере выше (слева) слова aabaaba и baabaaa сопряженные и aabaaba$\cdot$aa $=$ aa$\cdot$baabaaa.

Решение

Предположим, что два слова x и y одинаковой длины имеют сопряженные корни. Пусть uv – корень из x, а vu – корень из y. Тогда $x = (uv)^k$ и $y = (vu)^k$ для некоторого $k > 0$, поскольку их длина одинакова. Таким образом, $x = u \cdot v(uv)^{k-1}$ и $y = v(uv)^{k-1} \cdot u$, а это означает, что x и y сопряженные.

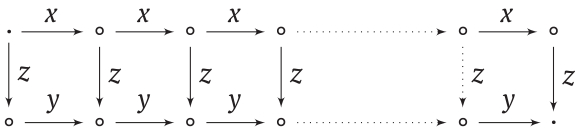
Обратно, предположим, что x и y сопряженные, и пусть u и v таковы, что $x = uv$ и $y = vu$. Обозначим z корень из x и возьмем $k > 0$ такое, что $x = z^k$. Пусть также u' и v' определены так, что u' является префиксом u, v' – суффиксом v и $z = u'v'$.



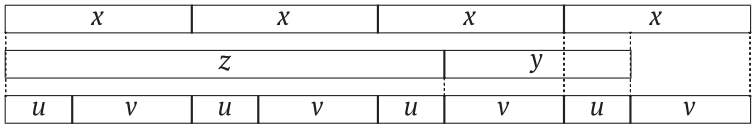
Тогда $y = vu = (v'u')^{k'}v'(u'v')^{k''}u'$, где $k' + k'' = k - 1$. Отсюда $y = (v'u')^k$, и, в силу леммы 2, корень t из y удовлетворяет неравенству $|t| \le |u'v'| = |z|$. Но поскольку роли x и y симметричны, это также доказывает, что $|z| \le |t|$ и, значит, $|z| = |t|$ и $t = v'u'$. Поэтому корни z и t соответственно из x и y являются сопряженными.

Чтобы ответить на второй вопрос, предположим сначала, что x и y сопряженные, т. е. $x = uv$ и $y = vu$. Тогда $xu = (uv)u = u(vu) = uy$, что доказывает утверждение, если положить $z = u$.

Обратно, предположим, что $xz = zy$ для некоторого слова z. Для любого целого положительного числа l получаем $x^l z = x^{l-1}zy = x^{l-2}zy^2 = \ldots = zy^l$. Это иллюстрируется на следующей диаграмме, полученной продолжением начальной диаграммы (левый квадрат) $xz = zy$, где $\circ$ обозначает конкатенацию.



Взяв целое k, удовлетворяющее условиям $(k-1)|x| \le |z| < k|x|$, получим, что z является собственным префиксом x^k длиной по меньшей мере x^{k-1} ($k = 3$ на рисунке ниже).



Таким образом, существуют два слова u и v такие, что $x = uv$ и $z = x^{k-1}u$. Отсюда следует, что $xz = (uv)^k u = zvu$, а значит, в силу условия $xz = zy$, получаем, что $y = vu$. Следовательно, x и y сопряженные.

Примечания

Сопряженность слов тесно связана с их периодичностью, как было показано на стр. 12. Дополнительные сведения о сопряженных словах можно найти в книге Lothaire [175].

12. Сопряженные палиндромы

Эта задача связана с двумя операциями над словами: обращение слова и взятие одного из сопряженных ему. Эти операции не согласованы в том смысле, что лишь изредка сопряженное слово одновременно является обратным.

Для изучения ситуации рассмотрим взаимно сопряженные палиндромы. Например, слова `abba` и `baab` одновременно являются палиндромами и сопряжены друг другу. С другой стороны, слово `aabaa` не имеет сопряженного палиндрома, т. е. его класс сопряженности содержит только один палиндром.

Вопрос. Какое максимальное число палиндромов может быть в классе сопряженности слова?

[**Указание:** рассмотрите примитивный корень из двух сопряженных палиндромов.]

Класс сопряженности слова `abba`, множество {`abba`,`bbaa`,`baab`,`aabb`}, содержит только два палиндрома. Так же обстоит дело со словом $(\texttt{abba})^3$, класс сопряженности которого содержит два палиндрома: `abbaab` `baabba` и `baabba` `abbaab`. Однако в классе сопряженности $(\texttt{abba})^2$ четыре элемента, но только один из них палиндром.

Решение

Рассмотренные выше примеры наводят на мысль, что класс сопряженности может содержать не более двух палиндромов. Но прежде чем убедиться в этом, докажем один промежуточный результат.

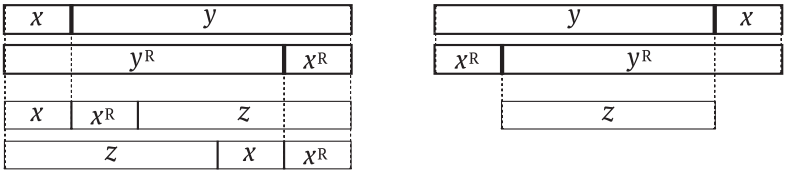
Лемма 4. *Если $x \neq x^R$ и $xx^R = w^k$ для некоторого примитивного слова w и целого положительного k, то k нечетно и $w = uu^R$ для некоторого слова u.*

Доказательство. Если k четно, то $xx^R = (w^{k/2})^2$, и тогда $x = x^R$ – противоречие. Поэтому k нечетно и, значит, $|w|$ четно. Пусть $w = uv$, где $|u| = |v|$. Так как u – префикс x, а v – суффикс x^R, имеем $v = u^R$, что и требовалось доказать. ■

Для двух непустых слов x и y предположим, что сопряженные слова xy и yx – два разных палиндрома. Тогда $xy = (xy)^R = y^R x^R$ и $yx = (yx)^R = x^R y^R$.

Чтобы доказать, что не может существовать больше двух сопряженных палиндромов, сначала покажем, что $xy = (uu^R)^k$ и $yx = (u^R u)^k$, где k – целое положительное число и слово u таково, что слово uu^R примитивное. Рассмотрим два случая: когда длины x и y равны и различны.

Если $|x| = |y|$, то $y = x^R$, откуда следует, что $xy = xx^R$ и $yx = x^R x$. Кроме того, $x \neq x^R$ в силу предположения $xy \neq yx$. В силу леммы 4, примитивный корень из xy имеет вид uu^R и $xy = (uu^R)^k$ для некоторого нечетного k.



Если $|x| \neq |y|$, то без ограничения общности можно предположить, что $|x| < |y|$ (см. рисунок). Тогда x является собственной границей y^R, а x^R – собственной границей y, откуда следует, что xx^R – собственная граница xy. Слово $z = (x^R)^{-1}y$ также является границей xy. Таким образом, слово xy имеет два периода: $|xx^R|$ и $|z|$, которые удовлетворяют условию леммы о периодичности. Следовательно, $q = \gcd(|xx^R|, |z|)$ тоже является периодом xy и делит его длину. Если w – примитивный корень из xy, то xy имеет вид w^k, $k > 1$, и $p = |w|$ – делитель q. Снова воспользовавшись леммой 4, получаем, что примитивный корень имеет вид uu^R, где $u \neq u^R$, потому что корень примитивный. Стало быть, $xy = (uu^R)^k$, где k – нечетное число.

Итак, вне зависимости от того, равны длины x и y или нет, мы приходим к одному и тому же выводу. Чтобы закончить доказательство, рассмотрим класс сопряженности палиндрома $(uu^R)^k$, где uu^R – примитивное число. Такой класс содержит еще один палиндром, а именно $(u^R u)^k$.

Поскольку слова, сопряженные $(u^R u)^k$, имеют вид $(st)^k$, где st – сопряженное к uu^R, то, повторно применив то же рассуждение, что и выше, мы убедимся, что неравенства $u \neq u^R$ и $s \neq u^R$ вступают в противоречие с примитивностью uu^R. Тем самым мы доказали, что класс сопряженности не может содержать больше двух палиндромов.

Примечание
Этот результат получен в работе Guo et al. [133], оттуда же взято приведенное доказательство (с небольшими изменениями).

13. Много слов с большим числом палиндромов

В этой задаче рассматриваются слова с максимально возможным числом палиндромных факторов. Слово w называется *богатым на палиндромы*, если оно содержит $|w|$ различных непустых палиндромов в качестве факторов, включая и однобуквенные палиндромы.

Пример. Слова `poor`, `rich` и `abac` богаты на палиндромы, а слова `maximal` и `abca` нет. Действительно, множество палиндромов, встречающихся в `abac`, – {`a`,`aba`,`b`,`c`}, а в `abca` – {`a`,`b`,`c`}.

Обозначим $Rich_k(n)$ количество богатых на палиндромы слов длины n над алфавитом размера k.

Заметим, что каждая позиция в слове является (начальной) позицией самого правого вхождения не более чем одного палиндрома. Действительно, второй по длине палиндром, начинающийся в той же позиции, был бы собственным суффиксом более длинного палиндрома и, следовательно, встречался бы позже, что противоречит предположению. Отсюда вытекает следующий факт.

Наблюдение. Существует не более $|w|$ палиндромов, являющихся факторами слова w.

Наиболее интересен случай двоичных слов, т. е. $k = 2$, поскольку в этом случае имеем:

$$\begin{cases} Rich_2(n) = 2^n & \text{для } n < 8, \\ Rich_2(n) < 2^n & \text{для } n \geq 8. \end{cases}$$

Вопрос. Доказать, что $Rich_2(2n)$ растет экспоненциально, т. е. что существует такая положительная постоянная c, что $Rich_2(2n) \geq 2^{cn}$.

[**Указание:** воспользуйтесь тем фактом, что число разбиений целых чисел растет экспоненциально.]

Решение
Рассмотрим все разбиения числа n на различные положительные числа:

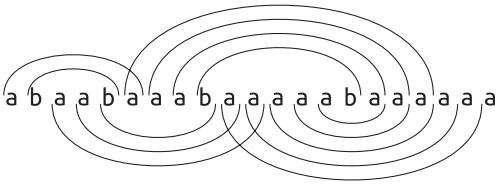
$$n = n_1 + n_2 + \ldots + n_k,\ n_1 < n_2 < \ldots < n_k.$$

Для каждого такого разбиения $\pi = (n_1, n_2, \ldots, n_k)$ рассмотрим слово w_π длины $n + k - 1$, определенное следующим образом:

$$w_\pi = \mathtt{a}^{n_1}\mathtt{ba}^{n_2}\mathtt{b}\ldots\mathtt{ba}^{n_k}.$$

Легко видеть, что слово w_π богато на палиндромы.

На рисунке ниже показаны палиндромы, содержащие разные буквы, встречающиеся в слове $aba^2ba^3ba^5ba^6$ длины 21, ассоциированном с разбиением (1,2,3,5,6) числа 17. Помимо 14 палиндромов, показанных на рисунке, слово содержит еще унарные палиндромы a, aa, aaa, aaaa, aaaaa, aaaaaa и b – всего 21 палиндром.



Дописывание b^{n-k+1} в конец w_π дает слово $v_\pi = w_\pi b^{n-k+1}$ длины $2n$, которое содержит дополнительные палиндромы $ba^{nk}b$, b^2, b^3, ..., b^{n-k+1}. Таким образом, v_π тоже богато на палиндромы. Известно, что число разбиений целого числа n на попарно различные целые положительные числа растет экспоненциально с ростом n. Поэтому $Rich_2(2n)$ также растет экспоненциально.

Примечания

Эта задача основана на обзоре Glen et al. [130], посвященном словам, богатым на палиндромы.

14. Короткое суперслово перестановок

Цель этой задачи – показать, что определенное множество образцов можно упаковать в одно слово, экономно используя память. Можно считать, что это метод сжатия конкретного множества.

Интересующие нас образцы, называемые n-перестановками, выбираются из алфавита натуральных чисел {1,2, ..., n}. Это слова, в которых каждое число встречается ровно один раз. Наша цель – построить так называемые n-суперслова, содержащие все n-перестановки в качестве факторов.

Для $n = 2$ слово 121 является самым коротким 2-суперсловом, потому что содержит обе 2-перестановки 12 и 21. Для $n = 3$ самым коротким 3-суперсловом является 123121321. В нем все шесть 3-перестановок встречаются в следующем порядке:

$\pi_1 = 123$, $\pi_2 = 231$, $\pi_3 = 312$, $\pi_4 = 213$, $\pi_5 = 132$, $\pi_6 = 321$.

Обратите внимание на структуру 123121321: по обе стороны каждого вхождения буквы 3 находятся 2-перестановки.

В приведенных примерах длина суперслов равна соответственно $\alpha_2 = 3$ и $\alpha_3 = 9$, где $\alpha_n = \sum_{i=1}^{n} i!$. Но не ясно, всегда ли самое короткое n-суперслово имеет длину α_n для $n \geq 4$.

Задача состоит в том, чтобы построить короткое n-суперслово, необязательно минимальной длины.

Вопрос. Показать, как можно построить n-суперслово длины α_n для любого натурального числа n.

[**Указание:** воспользуйтесь приведенным выше замечанием о структуре слова 123121321 для построения n-суперслова по $(n-1)$-суперслову.]

Решение
Построение производится итеративно, начиная с базового случая $n = 2$ (или $n = 3$), следующим образом.

Обозначим w_{n-1} $(n-1)$-суперслово длины α_{n-1}. Будем рассматривать $(n-1)$-перестановки в порядке их появления в w_{n-1}. Пусть i_k – конечная позиция первого вхождения k-й $(n-1)$-перестановки в w_{n-1}. Это означает, что существует ровно $k-1$ различных $(n-1)$-перестановок с конечной позицией $i < i_k$ (некоторые из них могут повторяться).

n-суперслово w_n строится путем вставки некоторых n-перестановок в w_{n-1}. В качестве таковых выбираются все слова $n \cdot \pi_k$, $1 \le k \le (n-1)!$, где π_k – k-е вхождение $(n-1)$-перестановки в w_{n-1}. Все эти слова вставляются одновременно – π_k после позиции i_k. Из определения i_k следует, что вставки порождают в w_n факторы вида $\pi_k \cdot n \cdot \pi_k$ для каждой $(n-1)$-перестановки π_k.

Пример. Построение w_4 по $w_3 = 123121321$. Конечные позиции всех шести 3-перестановок π_i в w_3 таковы:

$$i_1 = 2,\ i_2 = 3,\ i_3 = 4,\ i_4 = 6,\ i_5 = 7,\ i_6 = 8.$$

Вставка шести 4-перестановок вида $4 \cdot \pi_i$ порождает следующее 4-суперслово длины $\alpha_4 = 33$:

123**4**123**1**4**2**312**4**3121**3**4**2**132**4**13**2**1**4**321,

в котором выделены вхождения 4.

Длина слова w_n равна α_n. Поскольку мы вставили $(n-1)!$ слов длины n, длина результирующего слова w_n равна $|w_{n-1}| + (n-1)!\ n = \sum_{i=1}^{n} i! = \alpha_n$, что и требовалось доказать.

Все n-перестановки являются факторами слова w_n. n-перестановки, встречающиеся в w_n, имеют вид $u \cdot n \cdot v$, где uv – слово длины $n-1$, не содержащее буквы n. Эта перестановка встречается внутри фактора $vu \cdot n \cdot vu$ слова w_n, где $vu = \pi_k$ для некоторой $(n-1)$-перестановки π_k. Но, по построению, все слова вида $\pi_k \cdot n \cdot \pi_k$ входят в w_n, поэтому все n-перестановки встречаются в w_n. Тем самым мы дали ответ на поставленный вопрос.

Примечания
Была высказана гипотеза, что α_n – минимальная длина n-суперслова. Она подтверждена для $n = 4$ и $n = 5$ в работе Johnston [152], но опровергнута для $n = 6$ в работе Houston [143].

15. Короткая суперпоследовательность перестановок

В этой задаче речь пойдет об эффективном хранении множества образцов в слове. В отличие от суперслова, в данном случае образцы хранятся в виде подпоследовательностей слова, называемого суперпоследовательностью.

Образцы, называемые n-перестановками, выбираются из алфавита {1, 2, …, n}. Это слова, в которых каждое число встречается ровно один раз. Наша цель – построить слова, называемые n-суперпоследовательностями, которые содержат все n-перестановки в качестве подпоследовательностей.

Для $n = 3$ слово 1213212 длины 7 является самой короткой 3-суперпоследовательностью. Для $n = 4$ самой короткой 4-суперпоследовательностью длины 12 является слово 123412314321. Эти две суперпоследовательности имеют длину $n^2 - 2n + 4$ (для $n = 3, 4$). Заметим, что для $n = 4$ наша 4-суперпоследовательность имеет длину 12, тогда как самое короткое 4-суперслово, очевидно, длиннее – его длина равна 33 (см. задачу 14).

Простой способ построить n-суперпоследовательность – рассмотреть слово вида π^n для любой n-перестановки π, иначе говоря, вида $\pi_1\,\pi_2\,\pi_3\ldots\,\pi_n$, где π_i – произвольные n-перестановки. Ясно, что оно содержит все $n!$ n-перестановок в качестве подпоследовательностей, но его длина равна n^2, что далеко не оптимально.

Задача заключается в том, чтобы построить относительно короткую n-суперпоследовательность, пусть даже не минимальной длины.

Вопрос. Показать, как можно построить n-суперпоследовательность длины $n^2 - 2n + 4$ для любого натурального числа n.

[**Указание:** начав с очевидной n-суперпоследовательности длины n^2, покажите, как сократить ее до требуемой длины.]

Решение

Сократим описанную выше n-суперпоследовательность $x = \pi_1\,\pi_2\,\pi_3\ldots\,\pi_n$, выполнив два шага.

До длины $n^2 - n + 1$. Чтобы уменьшить длину x, выберем перестановки вида $n \cdot \rho_i$, $i = 1, \ldots, n-1$, где ρ_i – $(n-1)$-перестановка, и рассмотрим слово

$$y = n \cdot \rho_1 \cdot n \cdot \rho_2 \cdot n \cdots n \cdot \rho_{n-1} \cdot n.$$

Очевидно, что длина n-суперпоследовательности при этом уменьшается на $n - 1$ букв, так что мы получили требуемый результат.

До длины $n^2 - 2n + 4$. Далее техника построения несколько усложняется. Основная идея – более тщательно выбирать $(n-1)$-перестановки ρ_i в слове y.

Имея решения для $n \leq 4$, мы построим решения для $n \geq 5$. Пусть γ_1, γ_2 и γ_3 – три $(n-1)$-перестановки соответственно вида $3 \cdot \gamma_1' \cdot 2$, $1 \cdot \gamma_2' \cdot 3$ и $2 \cdot \gamma_3' \cdot 1$, где γ_1' – перестановка $\{1,2, \ldots, n-1\} \backslash \{2,3\}$ и аналогично для γ_2' и γ_3'.

Сначала конкатенируем по-другому $n-1$ блоков типа γ_i, а затем вставим между ними n; в результате получаем:

$$\gamma_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \cdots,$$
$$w = n \cdot \gamma_1 \cdot n \cdot \gamma_2 \cdot n \cdot \gamma_3 \cdot n \cdots n,$$
$$w = n \cdot 3 \cdot \gamma_1' \cdot 2 \cdot n \cdot 1 \cdot \gamma_2' \cdot 3 \cdot n \cdot 2 \cdot \gamma_3' \cdot 1 \cdot n \ldots n.$$

Из рассмотрения предыдущего случая следует, что это n-суперпоследовательность длины $n^2 - n + 1$.

Главный шаг построения заключается в удалении $n-3$ букв из w, что и дает желаемую длину $n^2 - n + 1 - (n-3) = n^2 - 2n + 4$. Это делается путем удаления буквы i из каждого вхождения γ_i' в w, за исключением первого и последнего, в результате чего получается слово z.

Слово z является n-суперпоследовательностью. Заметим, что удаление буквы i из блока γ_i' для $i = 1,2,3$ компенсируется присутствием i слева и справа от γ_i после буквы n. И далее рассуждение, похожее на то, что мы применили выше к слову y, доказывает, что z – n-суперпоследовательность.

Таким образом, мы построили n-суперпоследовательность длиной $n^2 - 2n + 4$.

Пример. Проиллюстрируем построение для случая $n = 6$. Пусть $\gamma_1 = 31452$, $\gamma_2 = 12453$ и $\gamma_3 = 23451$ – выбранные 5-перестановки. И пусть γ_i^{Rem} – слово γ_i после удаления буквы i.

Рассмотрим последовательность

$$w = 6 \cdot \gamma_1 \cdot 6 \cdot \gamma_2 \cdot 6 \cdot \gamma_3 \cdot 6 \cdot \gamma_1 \cdot 6 \cdot \gamma_2 \cdot 6.$$

Нужная 6-суперпоследовательность получается удалением буквы i из каждого блока γ_i, кроме первого и последнего, что дает

$$z = 6 \cdot \gamma_1 \cdot 6 \cdot \gamma_2^{\mathrm{Rem}} \cdot 6 \cdot \gamma_3^{\mathrm{Rem}} \cdot 6 \cdot \gamma_1^{\mathrm{Rem}} \cdot 6 \cdot \gamma_2 \cdot 6;$$

то есть

$$z = 6\ 31452\ 6\ 1453\ 6\ 2451\ 6\ 3452\ 6\ 12453\ 6.$$

Примечания

Описанный метод – это вариант построения, предложенного в работе Mohanty [191]. Известно, что это построение дает самую короткую суперпоследовательность длины $n^2 - 2n + 4$ для $2 < n \leq 7$. Однако для $n \geq 10$ построение из работы Zalinescu [242] дает суперпоследовательности длины $n^2 - 2n + 3$. Точная общая формула длины самой короткой n-суперпоследовательности до сих пор неизвестна; мы знаем только, что она имеет порядок $n^2 - o(n^2)$.

16. Слова Сколема

Словом Сколема порядка n, где n – целое положительное число, называется слово над алфавитом $A_n = \{1, 2, \ldots, n\}$, обладающее следующими свойствами для каждого $i \in A_n$:

(i) буква i встречается в слове ровно два раза;

(ii) вхождения i находятся друг от друга на расстоянии i.

Определение слов Сколема очень похоже на определение слов Лэнгфорда (задача 17), но небольшое изменение расстояния приводит к весьма значительным последствиям.

Если igi – фактор слова Сколема, то промежуточное слово g не содержит буквы i и $|g| = i - 1$. Например, 11, очевидно, является словом Сколема порядка 1, 23243114 – слово Сколема порядка 4, а 4511435232 – слово Сколема порядка 5. Но простая проверка показывает, что не существует слов Сколема порядка 2 или 3.

Вопрос. Подумайте, для каких целых положительных n существует слово Сколема порядка n, и спроектируйте алгоритм его построения в тех случаях, когда это возможно.

[**Указание:** рассматривайте n по модулю 4.]

Решение

Мы рассмотрим четыре случая, зависящих от остатка при делении n на 4.

Случай $n = 4k$**.** Слово 23243114 – пример слова Сколема порядка 4. Пусть $n = 4k$ для $k > 1$. Ниже описана процедура построения слова Сколема порядка n.

В основе построения лежат два слова w_{even} и w_{odd}. Первое образовано возрастающей последовательностью четных чисел из A_n, а второе – возрастающей последовательностью нечетных чисел из $A_n \setminus \{n - 1\}$ (наибольшее нечетное число отбрасывается).

Алгоритм Skolem порождает искомое слово.

```
Skolem(n кратно 4 и больше 4)
1  (c, d) ← (n/2 − 1, n − 1)
2  w_odd ← 1 3 ... n − 3 ▷ в w_odd нет буквы n − 1
3  α·c·β·1 1·β^R·c·α^R ← декомпозиция w_odd^R w_odd
4  v ← α·1 1·β·c·d·β^R·α^R·c
5  w_even ← 2 4 ... n
6  return v·w_even^R·d·w_even
```

Инструкция в строке 3 факторизует оба конца слова $w_{\text{odd}}^{\text{R}} w_{\text{odd}}$, окружающих букву c.

Пример. Для $n = 12$ алгоритм последовательно вычисляет по словам $w_{\text{odd}} = 1\ 3\ 5\ 7$ и $w_{\text{even}} = 2\ 4\ 6\ 8\ 10\ 12$ декомпозицию $w_{\text{odd}}^{\text{R}} w_{\text{odd}}$ с $c = 5$

$$9\ 7 \cdot 5 \cdot 3 \cdot 1\ 1 \cdot 3 \cdot 5 \cdot 7\ 9,$$

где $\alpha = 9\ 7$ и $\beta = 3$; тогда

$$v = 9\ 7 \cdot 1\ 1 \cdot 3 \cdot 5 \cdot 11 \cdot 3 \cdot 7\ 9 \cdot 5,$$

и окончательно получаем слово Сколема порядка 12:

$$9\ 7\ 1\ 1\ 3\ 5\ \mathbf{11}\ 3\ 7\ 9\ 5\ 12\ 10\ 8\ 6\ 4\ 2\ \mathbf{11}\ 2\ 4\ 6\ 8\ 10\ 12,$$

в котором $d = 11$ выделено полужирным шрифтом.

Почему этот алгоритм работает? Прежде всего отметим, что свойство (i) выполняется. Тогда ясно, что вхождения каждой буквы в $u = w_{\text{odd}}^{\text{R}} w_{\text{odd}}$, в v и в суффикс $w_{\text{even}}^{\text{R}} \cdot d \cdot w_{\text{even}}$ результата отстоят друг от друга на правильные расстояния.

Остается показать, что свойство (ii) имеет место для букв и c, и d. Внутри v расстояние между вхождениями c равно $|\alpha| + |\beta| + 1$ – количеству нечетных чисел, отличных от 1 и c, т. е. $n/2 - 2$, как и положено.

Расстояние между двумя вхождениями буквы d в результат равно $|\alpha| + |\beta| + 1 + |w_{\text{even}}|$, т. е. $|A_n \setminus \{1, d\}| = n - 2$, и это тоже правильно.

Таким образом, результат алгоритма Skolem(n) – слово Сколема порядка n.

Случай $n = 4k + 1$**.** Этот случай рассматривается так же, как предыдущий, но d полагается равным n, а c – равным $\lfloor n/2 \rfloor - 1$. Пусть w_{even}, как и прежде, возрастающая последовательность четных чисел из A_n, а w_{odd} – возрастающая последовательность нечетных чисел из $A_n \setminus \{n\}$ (наибольшее нечетное число отбрасывается).

При такой длине n алгоритм Skolem порождает требуемое слово. Заметим, что в первом случае v и результат содержат фактор $c \cdot d$, а во втором – фактор $d \cdot c$.

```
Skolem(n вида 4k + 1, большее 4)
1  (c, d) ← (⌊n/2⌋ − 1, n)
2  w_odd ← 1 3 ... n − 2 ▷ в w_odd нет буквы n
3  α·c·β·1 1·β^R·c·α^R ← декомпозиция w_odd^R w_odd
4  v ← α·1 1·β·d·c·β^R·α^R·c
5  w_even ← 2 4 ... n − 1
6  return v·w_even^R·d·w_even
```

Пример. Для $n = 13$ алгоритм успешно вычисляет по словам $w_{\text{odd}} = 1\ 3\ 5\ 7\ 9\ 11$ и $w_{\text{even}} = 2\ 4\ 6\ 8\ 10\ 12$ декомпозицию $w_{\text{odd}}^{\text{R}} w_{\text{odd}}$ с $\lfloor n/2 \rfloor - 1 = c = 5$:

$$11\ 9\ 7 \cdot \mathbf{5} \cdot 3 \cdot 1\ 1 \cdot 3 \cdot \mathbf{5} \cdot 7\ 9\ 11,$$

где $\alpha = 11\ 9\ 7$, $\beta = 3$; тогда

$v = 11\ 9\ 7\ \mathbf{1}\ \ \mathbf{1}\ 3\ \mathbf{13}\ \ \mathbf{5}\ 3\ 7\ 9\ 11\ \mathbf{5}$,

и окончательно получаем слово Сколема порядка 13:

$v = 11\ 9\ 7\ 1\ 1\ 3\ \mathbf{13}\ \ \mathbf{5}\ 3\ 7\ 9\ 11\ \mathbf{5}\ 12\ 10\ 8\ 6\ 4\ 2\ \mathbf{13}\ 2\ 4\ 6\ 8\ 10\ 12$,

в котором $c = 5$ и $d = 13$ выделены полужирным шрифтом.

Невозможность в остальных случаях. Обозначим $odd(n)$ количество нечетных натуральных чисел, не превосходящих n.

Наблюдение. Если существует слово Сколема порядка n, то имеет место равенство $odd(n) \bmod 2 = n \bmod 2$.

Для доказательства рассмотрим суммы по модулю 2 (будем называть их 2-суммами) позиций в слове Сколема w порядка n. Во-первых, 2-сумма всех позиций в w равна $n \bmod 2$. Во-вторых, объединим позиции одной и той же буквы i в пары для вычисления суммы. Если i четно, то обе позиции ее вхождений имеют одинаковую четность, поэтому их вклад в 2-сумму нулевой. Но если i нечетно, то позиции имеют разную четность. Поэтому 2-сумма позиций равна $odd(n) \bmod 2$. Следовательно, $odd(n) \bmod 2 = n \bmod 2$, что и требовалось доказать.

Невозможность существования слов Сколема для $n = 4k + 2$ и для $n = 4k + 3$ прямо вытекает из этого наблюдения, потому что в этих случаях $odd(n) \bmod 2 \neq n \bmod 2$.

Итак, слово Сколема порядка n существует, только если n имеет вид $4k$ или $4k + 1$.

Примечания

Слова, рассмотренные в этой задаче, были введены в работе [227].

17. Слова Лэнгфорда

Словом Лэнгфорда порядка n, где n – целое положительное число, называется слово над алфавитом $A_n = \{1, 2, \ldots, n\}$, обладающее следующими свойствами для каждого $i \in A_n$:

(i) буква i встречается в слове ровно два раза;
(ii) вхождения i находятся друг от друга на расстоянии $i + 1$.

Определение слов Лэнгфорда очень похоже на определение слов Сколема (задача 16), но небольшое изменение расстояния приводит к весьма значительным последствиям.

Если igi – фактор слова Лэнгфорда, то промежуточное слово g не содержит буквы i и $|g| = i$. Например, 312132 – слово Лэнгфорда порядка 3, а 41312432 – слово Лэнгфорда порядка 4.

Вопрос. Подумайте, для каких целых положительных n существует слово Лэнгфорда порядка n, и покажите, как его построить в тех случаях, когда это возможно.

[**Указание:** рассматривайте n по модулю 4.]

Решение
Мы рассмотрим четыре случая, зависящих от остатка при делении n на 4.

Случай $n = 4k + 3$**.** Выше был приведен пример слова Лэнгфорда порядка 3. Для $n \geq 7$, т. е. $k > 0$, положим $X_n = \{2k + 1, n - 1, n\}$ и $A'_n = A_n \setminus X_n$. Пусть w_{even} – возрастающая последовательность четных чисел в A'_n, а w_{odd} – возрастающая последовательность нечетных чисел в A'_n.

Заметим, что A'_n насчитывает $4k$ элементов: $2k$ четных и $2k$ нечетных букв. w_{even} и w_{odd} можно разбить на две половины: $w_{\text{even}} = p_1 \cdot p_2$, где $|p_1| = |p_2| = k$, а $w_{\text{odd}} = p_3 \cdot p_4$, где $|p_3| = |p_4| = k$.

Построение начнем со следующего слова, которое является почти словом Лэнгфорда:

$$u = p_2^{\mathrm{R}}\, p_3^{\mathrm{R}} * p_3 * p_2 * p_4^{\mathrm{R}}\, p_1^{\mathrm{R}} * * p_1 * p_4,$$

где $*$ обозначает отсутствующую букву, которую еще предстоит вставить. Ясно, что расстояние между двумя вхождениями каждой буквы $i \in A'_n$ равно $i + 1$.

Теперь достаточно по два раза подставить каждый из оставшихся элементов A_n вместо $*$, что делается в таком порядке:

$$4k + 2,\ 4k + 3,\ 2k + 1,\ 4k + 2,\ 2k + 1,\ 4k + 3.$$

Поскольку длина каждого p_j равна k, нетрудно вычислить расстояния между вставленными элементами из множества $\{2k + 1, n - 1, n\}$ и убедиться, что они удовлетворяют свойству (ii), так что мы действительно получили слово Лэнгфорда порядка n.

Пример. Пусть $n = 11 = 4 \times 2 + 3$, $k = 2$. Имеем $X_{11} = \{5,10,11\}$ и $A'_{11} = \{1,2,3,4,6,7,8,9\}$; тогда $p_1 = 2\ 4$, $p_2 = 6\ 8$, $p_3 = 1\ 3$ и $p_4 = 7\ 9$. Первый шаг дает

$$u = 8\ 6\ 3\ 1 * 1\ 3 * 6\ 8 * 9\ 7\ 4\ 2 * * 2\ 4 * 7\ 9,$$

что приводит к слову Ленгфорда порядка 11:

$$8\ 6\ 3\ 1\ \mathbf{10}\ 1\ 3\ \mathbf{11}\ 6\ 8\ \mathbf{5}\ 9\ 7\ 4\ 2\ \mathbf{10}\ \ \mathbf{5}\ 2\ 4\ \mathbf{11}\ 7\ 9,$$

в котором $2k + 1 = 5$, $n - 1 = 10$ и $n = 11$ выделены полужирным шрифтом.

Случай $n = 4k$**.** Пример слова Ленгфорда порядка 4 был приведен выше. Поэтому будем рассматривать $n = 4k + 4$, где $k > 0$. Этот случай аналогичен предыдущему. Сначала строим решение u для $4k + 3$. А затем производим несколько изменений, чтобы вставить в него наибольший элемент n. Для

этого подставим n вместо первого вхождения $2k+1$, а этот элемент сдвинем в конец слова, где он станет вторым вхождением. Второй экземпляр n вставим после него.

Иначе говоря, вставки в слово u, ассоциированное с $4k+3$, производятся в следующем порядке:

$$4k+2, 4k+3, n, 4k+2, 2k+1, 4k+3, 2k+1, n,$$

где последние два элемента дописаны в конец слова.

Расстояния между элементами, меньшими n, не изменяются, а расстояние между вхождениями наибольшего элемента $n = 4k+4$ именно такое, какое нужно, так что мы получили слово Ленгфорда порядка n.

Невозможность других случаев. Любое слово Ленгфорда w над алфавитом A_{n-1} можно преобразовать в слово Сколема над алфавитом A_n, прибавив 1 ко всем элементам w и вставив в начало две буквы 1. Например, применение этого преобразования к слову Ленгфорда 312132 дает слово Сколема 11423243.

Известно, что слов Сколема не существует для n вида $4k+2$ или $4k+3$ (см. задачу 16). То же наблюдение справедливо для слов Ленгфорда, что доказывает их отсутствие для n вида $4k+1$ или $4k+2$.

Итак, слово Ленгфорда существует, только когда n имеет вид $4k+4$ или $4k+3$ ($k \geq 0$).

Примечания

Существуют различные варианты определения слова Ленгфорда. Например, свойство (i) можно опустить. Как показано в работе Berstel [32], такие слова свободны от квадратов.

18. От слов Линдона к словам де Брёйна

Комбинаторный результат, полученный в этой задаче, позволит нам эффективно строить слова де Брёйна в онлайновом режиме.

Двоичное слово (над алфавитом {0,1}) называется словом де Брёйна порядка (или ранга) k, если, будучи рассматриваемо как цикл (в котором за последним элементом следует первый), содержит каждое двоичное слово длины k ровно один раз. Такое слово имеет длину 2^k. Существует удивительная связь между этими словами и лексикографическим порядком, еще раз показывающая, что упорядочение слов – важный инструмент в алгоритмах обработка текста.

Словом Линдона называется примитивное слово, являющееся лексикографически наименьшим в своем классе сопряженности.

Пусть p – простое число и $\mathcal{L}_p = (L_0, L_1, \ldots, L_m)$ – отсортированная последовательность двоичных чисел Линдона длины p или 1. Обозначим

$$\mathbf{b}_p = L_0 \cdot L_1 \ldots \cdot L_m$$

конкатенацию всех слов, входящих в $\mathcal{L}_p$.

Например, отсортированный список слов Линдона длины 5 или 1 имеет вид

$\mathcal{L}_5 = (0,00001,00011,00101,00111,01011,01111,1)$,

а результат конкатенации этих слов равен

$\mathbf{b}_5 = 0\ 00001\ 00011\ 00101\ 00111\ 01011\ 01111\ 1$.

Это лексикографически наименьшее слово де Брёйна порядка 5, имеющее длину $32 = 2^5$.

Вопрос. Для простого числа p показать, что слово $\mathbf{b}_p$ является словом де Брёйна порядка p.

Решение
Количество двоичных слов Линдона длины p в $\mathcal{L}_p$ равно $(2^p - 2)/p$ (см. задачу 1). Поэтому длина $\mathbf{b}_p$ равна $p(2^p - 2)/p + 2 = 2^p$. Теперь, чтобы показать, что это слово де Брёйна, достаточно убедиться, что каждое слово w длины p встречается в $\mathbf{b}_p$, рассматриваемом как цикл.

Начнем с предварительного наблюдения. Для слова x, принадлежащего $\mathcal{L}_p$, $x \neq 1$, обозначим $next(x)$ следующее за x слово в этой последовательности.

Наблюдение. Если $|x| = |next(x)| = p$, $x = uv$ и в v входит 0, то u является префиксом $next(x)$.

Доказательство. Предположим противное – что $next(x) = u'v'$, где $|u| = |u'| = t$ и $u' \neq u$. Тогда $u < u'$ в силу упорядоченности элементов $\mathcal{L}_p$. Однако слово $u \cdot 1^{n-t}$ является словом Линдона, лексикографически расположенным между uv и $u'v'$, что противоречит предположению $next(x) = u'v'$. Таким образом, u является префиксом $next(x)$. ■

Все слова Линдона длины p, по построению, являются факторами $\mathbf{b}_p$. Слова 0^p и 1^p – соответственно префикс и суффикс $\mathbf{b}_p$. Слова длины p в 1^+0^+ циклически встречаются в последних $p - 1$ позициях $\mathbf{b}_p$. Таким образом, остается доказать, что слова длины p, не являющиеся словами Линдона и не принадлежащие 1^*0^*, встречаются (не циклически) в $\mathbf{b}_p$. Пусть w – такое слово и L_i – сопряженное ему слово Линдона; тогда

$w = vu$ и $L_i = uv$,

где слова v и u непустые, поскольку $w \neq L_i$.

Рассмотрим два случая: когда v содержит вхождение 0 и когда не содержит.

Случай 1: v содержит 0. Тогда из наблюдения выше вытекает, что u – префикс $L_{i+1} = next(L_i)$. Следовательно, $w = vu$ – фактор L_iL_{i+1}.

Случай 2: v не содержит 0. Тогда $v = 1^t$ для некоторого $t > 0$. Пусть L_j – первое слово в $\mathcal{L}_p$, префиксом которого является u, а $L_{j-1} = u'v'$, где $|v'| = t$. Тогда v' не может содержать букву 0, потому что иначе было бы $u' = u$, и L_j было бы не

первым словом в $\mathcal{L}_p$ с префиксом u. Следовательно, $v' = 1^t = v$, и конкатенация $L_{j-1}L_j = u' \cdot v \cdot u \ldots$ содержит vu в качестве фактора.

В обоих случаях w входит в $\mathbf{b}_p$. Тем самым мы доказали, что $\mathbf{b}_p$ является словом де Брёйна.

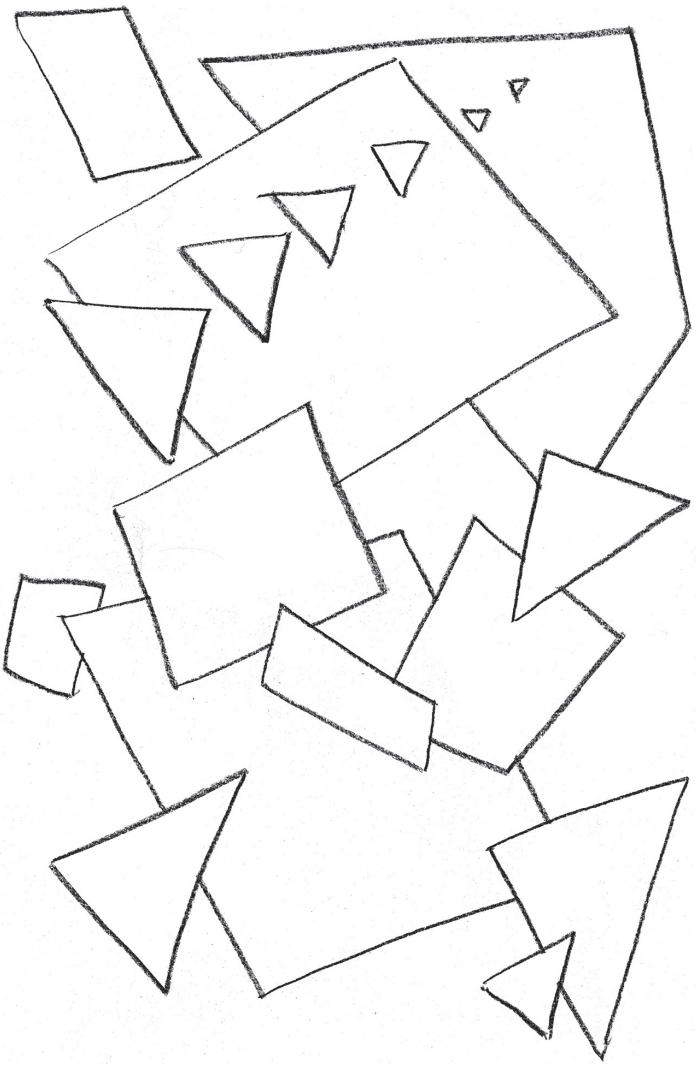
Примечания

Список $\mathcal{L}_p$ можно генерировать динамически, располагая памятью размера $O(p)$. Тогда приведенное выше построение позволяет сгенерировать слово де Брёйна, пользуясь окном размера $O(p)$ для хранения последних вычисленных букв слова.

Если порядок k слов де Брёйна – не простое число, то применимо аналогичное построение. В этом случае отсортированный список $\mathcal{L}_k$ состоит из слов Линдона, длина которых делит k. Конкатенация этих отсортированных слов дает лексикографически наименьшее слово де Брёйна порядка k над заданным алфавитом. Алгоритм был впервые разработан в работе Fredricksen and Maiorana [120]. См. также работу [192], где имеется более простое полное доказательство для общего случая.

Глава 3

Сопоставление с образцом



19. Таблица границ

Таблица границ, так же как и таблица префиксов в задаче 22, – основные инструменты построения эффективных алгоритмов работы со словами. Они используются главным образом для онлайнового поиска различных образцов в тексте.

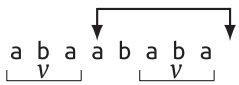
Таблица границ непустого слова x определена на длинах $l = 0, \ldots, |x|$ его префиксов следующим образом: $border[0] = -1$ и $border[l] = |Border(x[0..l-1])|$ для $l > 0$. Ниже показана таблица границ слова `abaababaaba`:

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
l	0	1	2	3	4	5	6	7	8	9	10	11
$border[l]$	-1	0	0	1	1	2	3	2	3	4	5	6

Вопрос. Доказать, что алгоритм Borders правильно вычисляет таблицу границ непустого слова x и выполняет не более $2|x| - 3$ сравнений букв при $|x| > 1$.

```
Borders(непустое слово x)
1  border [0] ← −1
2  for i ← 0 to |x| − 1 do
3      l ←border [i]
4      while l ≥ 0 и x[l] ≠ x[i] do
5          l ←border [l]
6      border [i + 1] ← l + 1
7  return border
```

Пример. Рассмотрим префикс $u =$ `abaababa` взятого ранее слова. Длина его границы $v =$ `aba` равна $3 = border[8]$. Следующая буква `a` расширяет границу, т. е. $Border(u\mathtt{a}) = v\mathtt{a}$.



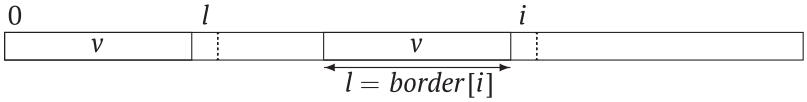
Если следующая буква c не равна `a`, то граница uc совпадает с границей vc; это идея доказательства рекуррентного соотношения для слова u и буквы c:

$$Border(uc) = \begin{cases} Border(u)c, & \text{если } Border(u)c - \text{префикс } u, \\ Border(Border(u)c) & \text{в противном случае.} \end{cases}$$

Вопрос. Показать, как определить непримитивные префиксы слова с помощью его таблицы границ.

Решение

Доказательство правильности опирается на приведенное выше рекуррентное соотношение. Его можно выразить по-другому, сказав, что вторая по длине граница слова u, если таковая существует, является границей его границы.



Правильность алгоритма Borders. Сначала отметим, что $border[0]$ установлено правильно. В строках 3–6 цикла $border[i+1]$ вычисляется по всем предыдущим значениям $border[i']$, $i' \neq 0, \ldots, i$. В цикле while l пробегает длины границ $x[0..i-1]$ от наибольшей к наименьшей.

Если цикл while останавливается и $l = -1$, то никакая граница не может быть расширена буквой $x[i]$; тогда следующая граница пуста, т. е. $border[i+1] = 0$, как и должно быть. В противном случае цикл останавливается при нахождении первого совпадения, что дает самую длинную границу $x[0..i-1]$ (длины l), расширяемую буквой $x[i]$. Тогда $border[i+1] = l+1$, снова как и должно быть. Тем самым доказательство завершено.

Время работы алгоритма Borders. Покажем, что значение $2i - l$ возрастает по крайней мере на 1 после каждого сравнения букв. Действительно, после успешного сравнения i и l увеличиваются на 1, а после неудачного l уменьшается по меньшей мере на 1, а i не изменяется.

Когда $|x| > 1$, значение $2i - l$ изменяется от 1 (при первом сравнении, когда $i = 1$ и $l = 0$) до не более чем $2|x| - 2$ (при последнем сравнении, когда $i = |x| - 1$ и $l \geq 0$). Таким образом, общее количество сравнений ограничено величиной $2|x| - 3$, как и утверждалось. Верхняя граница $2|x| - 3$ достигается на любой строке x вида $a^{|x|-1}b$, где a и b – различные буквы.

Непримитивные префиксы. По определению, непримитивное слово u имеет вид w^k для некоторого непустого слова w и целого $k > 1$. Это означает, что $|u| = k|w|$, т. е. период $|w|$ слова u делит его длину $|u|$. Поэтому, чтобы найти непримитивный префикс длины i слова x, зная таблицу границ x, можно проверить, является ли показатель степени $i/(i - border[i])$ префикса целым числом, большим 1.

Примечания

Применение таблиц границ для сравнения слов – классическая тема, освещаемая в учебниках, например [74, 96, 134, 194, 228]. Идея принадлежит Моррису и Пратту (см. [162]).

Поскольку $|u| - border[|u|]$ – наименьший период u, таблицу границ слова можно преобразовать в таблицу периодов его префиксов. Удивительный способ вычисления этой таблицы для слова Линдона показан в задаче 42.

20. Кратчайшие покрытия

Понятие покрытия пытается уловить регулярность слова. Оно не ограничивается периодичностью слова – рассматривается потенциально более короткий фактор, покрывающий слово целиком. Для периодических слов существуют специальные покрытия соседними отрезками, тогда как в покрытиях общего вида отрезки могут перекрываться. В этом смысле покрытие является обобщением понятия периода. Точнее, покрытие u слова x – это граница x, для которой расстояние между соседними вхождениями не превышает $|u|$.

Пример. Слово $u = \texttt{aba}$ является покрытием слова $x = \texttt{abaababa}$. Действительно отсортированный список начальных позиций u в x $pos(\texttt{aba}, \texttt{abaababa}) = (0,3,5)$, а максимальное расстояние между соседними позициями в списке $MaxGap(\{0,3,5\}) = 3 \leq |u|$. Слово u является кратчайшим покрытием x.

Кратчайшим покрытием слова abaababaaaba является слово целиком, это тривиальное покрытие, существующее всегда. Если оно является единственным покрытием, то слово называется *суперпримитивным*. Очевидно, что суперпримитивное слово является также примитивным.

Таблица покрытий *cover*, ассоциированная со словом x, определяется на длинах l его префиксов следующим образом: $cover[l]$ – наименьшая длина покрытий $x[0..l-1]$. Ниже показана таблица покрытий слова abababaaba.

i		0	1	2	3	4	5	6	7	8	9
$x[i]$		a	b	a	b	a	b	a	a	b	a
l	0	1	2	3	4	5	6	7	8	9	10
$cover[l]$	0	1	2	3	2	3	2	3	8	9	3

Вопрос. Спроектировать алгоритм, который за линейное время вычисляет кратчайшее покрытие каждого префикса слова.

[**Указание**: воспользуйтесь таблицей границ слова.]

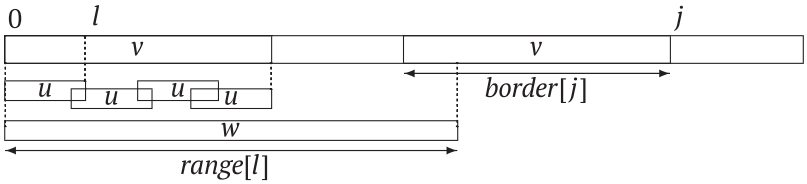
Решение

Представленное ниже решение, онлайновый алгоритм вычисления таблицы покрытий слова x, опирается на следующее важное наблюдение. (Таблица границ слова рассматривается в задаче 19.)

Наблюдение. Единственный кандидат на роль нетривиального кратчайшего покрытия $x[0..j-1]$ – кратчайшее покрытие $u = x[0..l-1]$ слова $v = x[0..border[j]-1]$ – самой длинной границы $x[0..j-1]$ (см. рисунок). В самом деле, любое нетривиальное покрытие $x[0..j-1]$ является покрытием, возможно тривиальным, его границы.

Кроме того, важную роль в алгоритме играет вспомогательная таблица *range*: $range[l]$ – длина самого длинного префикса $x[0..j-1]$, покрытого $x[0..l-1]$ (на рисунке префикс w покрыт u). Следующее наблюдение объясняет роль таблицы *range*.

Наблюдение. Если $u = x[0..l-1]$ – покрытие границы $x[0..j-1]$ и величина $range[l]$ не меньше периода $x[0..j-1]$, то u является покрытием $x[0..j-1]$.



Эти наблюдения реализует алгоритм ShortestCovers.

```
ShortestCovers(непустое слово x)
1   border ← Borders(x)
2   for j ← 0 to |x| do
3       (cover[j], range[j]) ← (j, j)
4   for j ← 2 to |x| do
5       l ← cover[border[j]]
6       if l > 0 и (range[l] ≥ j − border[j]) then
7           (cover[j], range[l]) ← (l, j)
8   return cover
```

После выполнения алгоритма ShortestCovers для слова `abababaaba` получаются такие таблицы:

j	0	1	2	3	4	5	6	7	8	9	10
$border[j]$	-1	0	0	1	2	3	4	5	1	2	3
$range[j]$	0	1	6	10	4	5	6	7	8	9	10
$cover[j]$	0	1	2	3	2	3	2	3	8	9	3

Отметим, что суперпримитивными являются те префиксы, для которых j удовлетворяет равенству $cover[j] = j$.

После вычисления таблицы границ инструкция в строке 3 тривиальным образом инициализирует таблицы *cover* и *range*. Инструкции в строках 4–7 вычисляют $cover[j]$ и обновляют *range*. Условие ($range[l] \geq j - border[j]$) в строке 6 проверяет, в соответствии со вторым наблюдением, действительно ли l равно длине кратчайшего покрытия $x[0..j-1]$. На этом завершается доказательство правильности алгоритма.

Поскольку алгоритм Borders требует линейного времени, то же справедливо и для алгоритма ShortestCovers.

Примечания

Этот алгоритм предложен в работе Breslauer [43] для проверки суперпримитивности слов.

21. Короткие границы

В этой задаче рассматривается таблица границ слова специального вида. Она предназначена для поиска в тексте образцов Зимина, содержащих слова-переменные (см. задачу 43). Показана важная роль таблицы границ для онлайнового поиска образцов различных типов.

Границей непустого слова x называется любое слово, являющееся одновременно его префиксом и суффиксом. Говорят, что граница *короткая*, если его длина меньше $|x|/2$. $Border(x)$ и $ShortBorder(x)$ обозначают соответственно самую длинную границу x и его самую длинную короткую границу. Любая из этих границ может быть пустым словом.

Например, границами слова $x =$ `abababab` являются ε, `ab`, `abab` и `ababab`. Из них только ε и `ab` являются короткими границами, так что $Border(x) =$ `ababab`, а $ShortBorder(x) =$ `ab`.

В некоторых алгоритмах короткую границу желательно знать для всех префиксов слова. Обозначим *shbord* ***таблицу коротких границ*** префиксов непустого слова x. Она определяется для длин префиксов l, $0 < l \le |x|$, следующим образом:

$$shbord[l] = |ShortBorder(x[0..l-1])|$$

(по соглашению $shbord[0] = -1$). Ниже показаны обе таблицы для слова `abaababaaba`. Они различаются в позициях $l = 6, 10, 11$.

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
l	0	1	2	3	4	5	6	7	8	9	10	11
$border[l]$	−1	0	0	1	1	2	3	2	3	4	5	6
$shbord[l]$	−1	0	0	1	1	2	1	2	3	4	2	3

Вопрос. Спроектировать алгоритм, который за линейное время вычисляет таблицу *shbord* непустого слова.

Решение

Прямолинейный подход – вычислить таблицу *shbord* по таблице границ *border* слова, не обращаясь к самому слову. Но тогда для слов вида a^k или $(ab)^k$ время работы, вероятно, будет квадратичным.

Вместо этого поступим по-другому: алгоритм ShortBorders, все же использующий таблицу границ, является модификацией алгоритма Borders, вычисляющего эту таблицу (см. задачу 19). Таким образом, он тоже требует линейного времени. Алгоритм пытается расширить предыдущую короткую границу, а если расширение оказывается слишком длинным, то пользуется таблицей границ, чтобы переключиться на более короткую границу.

```
ShortBorders(непустое слово x)
1   border ← Borders(x)
2   shbord[0] ← −1
3   for i ← 0 to |x| − 1 do
4       l ← shbord[i]
5       while (l ≥ 0 и x[l] ≠ x[i]) или (2l + 1 ≥ i) do
6           l ←border[l]
7       shbord[i + 1] ← l + 1
8   return shbord
```

Правильность алгоритма ShortBorders следует из того факта, что следующая короткая граница является расширением предыдущей короткой границы слова $u = x[0..i-1]$ на один символ $x[i]$ или расширением более короткой границы u.

Поскольку число выполнений строки 6 ограничено числом инкрементов l, полное время работы равно $O(|x|)$, что и требовалось доказать.

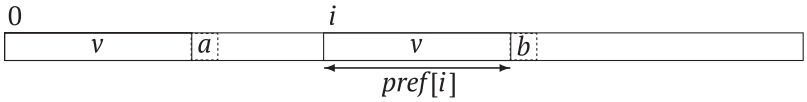
Примечания

Обоснованием введения и вычисления таблицы *shbord* служит ее важная роль для вычисления типов Зимина слов и их принципиальная важность в алгоритмах быстрого поиска образцов Зимина (содержащих переменные) в словах (не содержащих переменных) (задача 43).

22. Таблица префиксов

Таблица префиксов, как и таблица границ из задачи 19, – важное средство построения эффективных алгоритмов работы со словами. Она используется главным образом при поиске в тексте образцов различных видов.

Пусть x – непустая строка. ***Таблица префиксов*** x определена для позиций $i = 0, \ldots, |x| - 1$, следующим образом: $pref[i]$ равно длине самого длинного префикса, начинающегося в позиции i. Очевидно, что $pref[0] = |x|$.



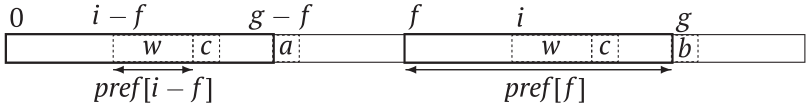
Ниже показана таблица префиксов слова abaababaaba:

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	a	b	a	b	a	a	b	a
$pref[i]$	11	0	1	3	0	6	0	1	3	0	1

Вопрос. Показать, что алгоритм PREFIXES за линейное время вычисляет таблицу префиксов входного слова x и выполняет не более $2|x| - 2$ сравнений букв.

```
PREFIXES(непустое слово x)
 1  pref[0] ← |x|
 2  g ← 0
 3  for i ← 1 to |x| − 1 do
 4      if i < g и pref[i − f] ≠ g − i then
 5          pref[i] ← min{pref[i − f], g − i}
 6      else (f, g) ← (i, max{g, i})
 7          while g < |x| и x[g] = x[g − f] do
 8              g ← g + 1
 9          pref[f] ← g − f
10    return pref
```

Основная идея алгоритма PREFIXES, который вычисляет таблицу последовательно слева направо, – воспользоваться тем, что уже было вычислено до текущей позиции.



Если $v = x[f..g-1]$ – префикс x и позиция i расположена между f и g (см. рисунок), то первый шаг вычисления $pref[i]$ состоит в том, чтобы проверить, можно ли вывести его значение на основе работы, проделанной для вхождения префикса v, т. е. в позиции $i - f$ слова x. Этой экономии достаточно, чтобы время выполнения алгоритма стало линейным.

Решение

Правильность алгоритма PREFIXES. Сначала проясним роль переменных f и g. В какой-то момент работы алгоритма позиция g является самой дальней от начала слова, в которой имело место неудачное сравнение букв. Точнее, для данной позиции i $g = \max\{j + pref[j] : 0 < j < i\}$. А соответствующая позиция f удовлетворяет равенству $f + pref[f] = g$.

На первой итерации цикла for устанавливаются переменные f и g, а также $pref[i] = pref[f]$ согласно их определениям, путем простого сравнения букв; это дает нам инвариант цикла.

Чтобы доказать, что этот инвариант сохраняется на последующих итерациях цикла, рассмотрим внимательно, что делается в строках 4–9.

Если $i < g$ и $pref[i - f] < g - i$, то ясно, что $pref[i] = pref[i - f]$ (см. рисунок выше). Если $i < g$ и $pref[i - f] > g - i$, то самый длинный префикс x, начинающийся в позиции $i - f$, имеет вид $v'x[g - f]v''$, где v' – суффикс $x[i - f..g - f - 1] =$

$x[f..g-1]$. Тогда, поскольку $x[g] \neq x[g-f]$, v' является самым длинным префиксом x, начинающимся в позиции i, т. е. $pref[i] = g - i$. Таким образом, если $i < g$ и $pref[i-f] \neq g - i$, то инструкция в строке 5 правильно устанавливает значение $pref[i]$, не нарушая инварианта.

Если условие в строке 4 не выполнено, как на первой итерации, то значение $pref[i]$ правильно устанавливается путем сравнения букв, и после этой итерации инвариант удовлетворяется. На этом доказательство правильности завершается.

Время работы алгоритма Prefixes. Время работы определяется прежде всего количеством сравнений букв в строке 7. Для каждого значения i производится не более одного неудачного сравнения, потому что после него цикл while прекращается, т. е. всего неудачных сравнений может быть не более $|x| - 1$. Для каждого значения g может быть не более одного успешного сравнения, потому что при этом значение g увеличивается, а уменьшаться оно не может. Таким образом, успешных сравнений также не более $|x| - 1$. Следовательно, общее количество сравнений букв не превосходит $2|x| - 2$, а значит, полное время работы имеет порядок $O(|x|)$.

Примечания

Таблицы префиксов, как и таблицы границ из задачи 19, важны при проектировании алгоритмов обработки текста и иногда неявно упоминаются в учебниках, например [74, 96, 134, 194, 228]. Алгоритм Prefixes назван Z-алгоритмом в книге [134, стр. 9].

23. От таблицы границ к максимальному суффиксу

Максимальным суффиксом слова называется его лексикографически наибольший суффикс. Это понятие применяется при проектировании оптимальных алгоритмов поиска и тестов периодичности. В данной задаче рассматривается его вычисление на основе алгоритма построения таблицы границ.

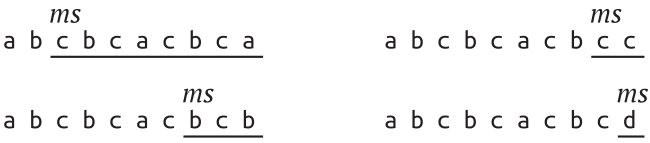
Вопрос. Показать, что следующий вариант вычисления таблицы границ правильно возвращает начальную позицию максимального суффикса входного слова и требует линейного времени.

```
MaxSuffixPos(непустое слово x)
 1  ms ← 0
 2  border[0] ← −1
 3  for i ← 0 to |x| − 1 do
 4      l ←border[i − ms]
 5      while l ≥ 0 и x[ms + l] ≠ x[i] do
```

```
 6          if x[ms + l] < x[i] then
 7              ms ← i − l
 8          l ←border[l]
 9      border[i − ms + 1] ← l + 1
10  return ms
```

Пример. Для слова x = abcbcacbc ***максимальный суффикс*** MaxSuffix(x) = cbcacbc, его самая длинная граница равна v = cbc. Следующую букву нужно сравнивать с буквой, следующей за префиксом cbc максимального суффикса.

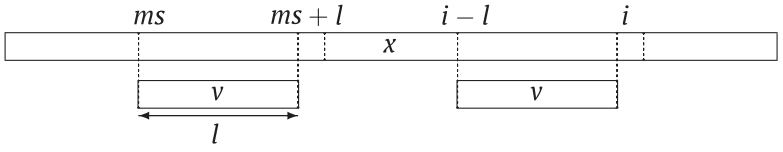
На рисунке ниже показано, как изменяется максимальный суффикс при добавлении в конец буквы a, b, c или d. Заметим, что если периодичность максимального суффикса изменяется, то максимальный суффикс имеет вид va, где v – граница начального максимального суффикса, а a – новая буква.



Вопрос. Показать, как найти лексикографически наибольшее сопряженное слово.

Решение

Онлайновый алгоритм основан на приведенном выше замечании о границе предыдущего максимального суффикса. Инструкции внутри цикла for призваны одновременно обновить начальную позицию максимального суффикса и вычислить длину его границы. Заметим, что не будь инструкций в строках 6–7, переменная *ms* осталась бы нулевой, а алгоритм вычислял бы таблицу границ всего слова x.



Правильность алгоритма MaxSuffixPos. Для доказательства покажем, что после выполнения инструкций внутри цикла for (строки 4–9) сохраняется

следующий инвариант: *ms* является начальной позицией максимального суффикса $x[0..i-1]$ и $border[i-ms]$ равно длине границы этого суффикса длиной $i-ms$.

После инициализации и первой итерации цикла для $i=0$ имеем $ms=0$ и $border[1]=0$ – правильный результат для слова $x[0]$.

Другие итерации начинаются с установки значения l равным длине границы $x[ms..i-1]$. Правильность вычисления длины границы $x[ms..i]$ устанавливается так же, как в рассуждении об изменении *border* в строке 8 при построении таблицы границ в задаче 19. Перед выходом из цикла while переменная *ms* обновляется в строке 7 всякий раз, как граница *v* текущего максимального суффикса удовлетворяет условию $vx[ms+l] < ux[i]$, что по сути дела устанавливает положение максимального суффикса, $x[ms..i]$.

Время работы алгоритма MaxSuffixPos. Время работы такое же, как для вычисления таблицы границ (см. задачу 19), т. е. $O(|x|)$, потому что инструкции в строках 6–7 занимают постоянное время.

Наибольшее сопряженное слово. Положим $ms = \text{MaxSuffixPos}(y)$, где $y = xx$. Тогда наибольшее сопряженное слово *x* равно $y[ms..ms+|x|-1]$. Чтобы не дублировать *x*, можно воспользоваться при операциях с индексами арифметикой по модулю.

Примечания

В основу этого алгоритма положено вычисление наименьшей ротации циклического слова в работе Booth [39].

Два других решения задачи о вычислении максимального суффикса слова приведены в задачах 38 и 40. Время их работы также линейно, но имеется преимущество – постоянный объем дополнительной памяти. Второй алгоритм, кроме того, сообщает период максимального суффикса.

24. Тест периодичности

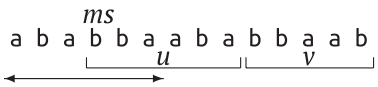
Обнаружение периодичности слова важно при проектировании алгоритмов поиска в тексте и сжатия текста. Цель – найти тест периодичности, не требующий много памяти.

Говорят, что слово *x* ***периодическое***, если его (наименьший) период не превосходит половины длины, т. е. $per(x) \le |x|/2$. Эквивалентно, *x* периодическое, если имеет вид $u^k v$, где целое число $k \ge 2$, слово *u* непустое, а *v* – собственный префикс *u*.

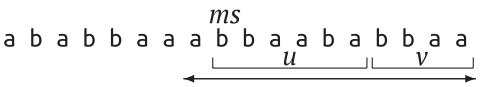
Проверить это свойство нетрудно, воспользовавшись таблицей границ или таблицей префиксов *x*. Задача – сделать это за линейное время и с постоянным объемом дополнительной памяти (помимо необходимой для хранения входного значения). Идея в том, чтобы воспользоваться функцией MaxSuffixPos из задачи 40, которая возвращает позицию и период максимального суффикса *x*.

Примеры ниже иллюстрируют варианты периода слова при разных максимальных суффиксах. Максимальный суффикс начинается в позиции ms и имеет вид $u^k v$ (здесь $k = 1$), где $u = x[ms..ms + p - 1]$, p – его период, а v – собственный префикс u.

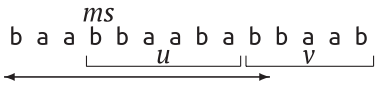
Сначала рассмотрим слово $x =$ `ababbaababbaab`. Префикс `aba` слова x, предшествующий максимальному суффиксу uv, является суффиксом u. В этом случае x периодическое с периодом 6.



Далее пусть $x =$ `ababbaaabbaababbaa`. Префикс x, предшествующий uv, длиннее, чем u. Период x равен $11 > |uv|$, поэтому x непериодическое.



Наконец, пусть $x =$ `baabbaababbaab`. Префикс x, предшествующий uv, короче, чем u, но не является его суффиксом. Период x равен $10 > |x| - |v|$, так что x не является периодическим.



Вопрос. Показать, что периодичность слова x можно проверить, выполнив менее $|x|/2$ сравнений, если заданы начальная позиция и период его максимального суффикса.

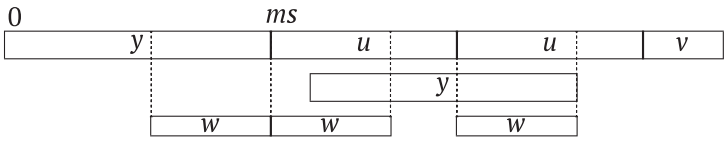
Решение

Пусть ms – начальная позиция, а p – период максимального суффикса x. Решение заключается в том, чтобы проверить условие в строке 2 алгоритма ниже, для чего требуется меньше $|x|/2$ сравнений.

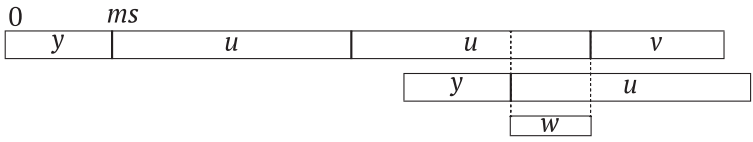
```
PERIODIC(непустое слово x)
1  (ms, p) ← MAXSUFFIXPP(x)
2  if ms < |x|/2 и p ≤ |x|/2 и x[0..ms − 1] – суффикс x[ms..ms + p − 1] then
3      return TRUE
4  else return FALSE
```

Ясно, что если условие выполняется, то x периодическое с периодом p, а если $ms \geq |x|/2$ или $p > |x|/2$, то слово непериодическое.

Пусть $x = yz$, где $y = x[0..ms-1]$, $z = x[ms..|x|-1] = u^k v$ – максимальный суффикс, u – периодический префикс z, v – собственный префикс u и $k > 0$. Предположим, что y не является суффиксом u.



Сначала рассмотрим случай, когда $y| \geq |u|$, и покажем, что $per(x) > |z|$. Если это не так (см. рисунок выше), то второе вхождение y в x перекрывается с некоторым вхождением u. Их перекрытие w является суффиксом y и префиксом u (или v), а значит, и z. Пусть $z = wz'$. И wz, и z' являются суффиксами x, меньшими чем z. Но раз $wz < z = wz'$, то $z < z'$ – противоречие. Следовательно, $per(x) > |z|$ и $per(x) > |y|$ (см. задачу 41), откуда получаем $2per(x) > |y| + |z|$, что и доказывает непериодичность x.



Во втором случае $|y| < |u|$, и мы сначала покажем, что $per(x) > \min\{|z|, |x| - |v|\}$. Если это не так и $per(x) \leq |z|$, то мы приходим к тому же выводу, что и выше. Теперь предположим, что $per(x) \leq |x| - |v|$, и придем к противоречию. На самом деле мы имеем $per(x) < |x| - |v|$, потому что y не является суффиксом u. Таким образом, u строго перекрывает себя же (см. рисунок выше). Противоречие возникает из-за того, что u свободно от границ (см. пример в задаче 40). Раз так, то мы имеем $per(x) > |x| - |v|$, а поскольку также имеет место тривиальное неравенство $per(x) > |v|$, то $2per(x) > |x|$, откуда следует, что x непериодическое. На этом завершается доказательство правильности алгоритма Periodic.

Примечания

Алгоритм Periodic проверяет периодичность слова x, но не вычисляет его период. На самом деле период можно вычислить при той же пространственной и временной сложности, применив оптимальный по времени и пространству алгоритм сопоставления строк, описанный в работе [69]. Оптимальный алгоритм Галила–Сейфераса [124] (см. [97]) тоже можно адаптировать и получить такой же результат.

25. Строгие границы

При использовании в онлайновых алгоритмах поиска таблицу границ образца лучше заменить объектом, рассматриваемым в этой задаче. В результате улучшается поведение поиска, как показано в задаче 26.

Таблица строгих границ непустого слова x определена на длинах $l = 0, \ldots, |x|$ его префиксов следующим образом: $stbord[0] = -1$, $stbord[|x|] = border[|x|]$ и для $0 < l < |x|$, $stbord[l]$ – это наибольшее t, удовлетворяющее условиям:

- $-1 \leq t < l$ и
- ($x[0..t-1]$ – граница $x[0..l-1]$ и $x[t] \neq x[l]$) или $t = -1$.

Слово $x[0..stbord[l]-1]$ является строгой границей префикса $x[0..l-1]$ слова x. Оно существует, только если $stbord[l] \neq -1$.

Ниже показаны таблицы границ и строгих границ слова `abaababaaba`:

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
l	0	1	2	3	4	5	6	7	8	9	10	11
$border[l]$	-1	0	0	1	1	2	3	2	3	4	5	6
$stbord[l]$	-1	0	-1	1	0	-1	3	-1	1	0	-1	6

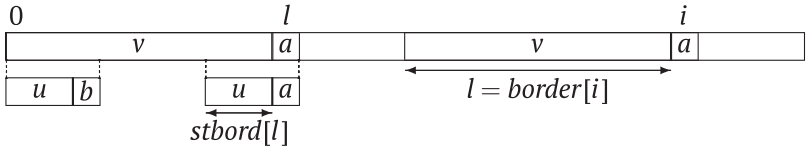
Вопрос. Спроектировать алгоритм, который за линейное время вычисляет таблицу *stbord*, не пользуясь таблицей *border* или еще какой-нибудь дополнительной таблицей.

Решение

Сначала заметим, что таблицу *stbord* можно использовать для вычисления таблицы *border*. Для этого нужно только заменить *border* на *stbord* в строке 5 алгоритма Borders (см. задачу 19), в результате чего получим

```
Borders(непустое слово x)
 1  border[0] ← −1
 2  for i ← 0 to |x| − 1 do
 3      l ← border[i]
 4      while l ≥ 0 и x[i] ≠ x[l] do
 5          l ← stbord[l]
 6      border [i + 1] ← l + 1
 7  return border
```

Это ускоряет вычисление таблицы для примеров типа $a^n b$.



Наблюдение. Вычисление таблицы строгих границ основано на следующем свойстве. Если $l = border[i]$ для некоторого i, $0 \leq i \leq |x|$, то

$$stbord[i] = \begin{cases} l, & \text{если } i = 0, \text{ или } i = |x|, \text{ или } x[i] \neq x[l], \\ stbord[l] & \text{в противном случае } (0 < i < |x| \text{ и } x[i] = x[l]). \end{cases}$$

Действительно, если первое условие выполнено, то определение границы $x[0..i-1]$ совпадает с определением строгой границы, т. е. $stbord[i] = border[i]$. В противном случае (см. рисунок) ситуация такая же, как при вычислении строгой границы $x[0..l]$, и тогда $stbord[i] = stbord[l]$.

```
StrictBorders(непустое слово x)
 1  stbord[0] ← −1
 2  l ← 0
 3  for i ← 1 to |x| − 1 do
 4      ▷ здесь l = border[i]
 5      if x[i] = x[l] then
 6          stbord[i] ← stbord[l]
 7      else stbord[i] ← l
 8          do l ← stbord[l]
 9          while l ≥ 0 и x[i] ≠ x[l]
10      l ← l + 1
11  stbord[|x|] ← l
12  return stbord
```

Алгоритм StrictBorders решает задачу. Инструкции в командах 5–7 реализуют приведенное выше наблюдение. Инструкции в строках 8–9 соответствуют инструкциям в строках 4–5 алгоритма Borders и служат для вычисления длины l границы $x[0..i+1]$. Правильность следует из этих замечаний, а линейность времени выполнения – из линейности алгоритма Borders.

Примечания

Таблица *stbord* – часть алгоритма сопоставления строк Кнута–Морриса–Пратта [162], улучшающего оригинальный алгоритм Морриса–Пратта (см. [74, глава 2]). Для этого онлайнового алгоритма улучшение обусловлено уменьшением задержки между обработкой двух соседних символов текста, в котором производится поиск (см. задачу 26). Дальнейшее улучшение задержки дает автомат сопоставления строк (см. задачу 27).

26. Задержка последовательного сопоставления строк

Алгоритм Кнута–Морриса–Пратта (КМП) включает ключевое средство (использование таблицы границ или чего-то подобного), нужное для проектирования алгоритмов сопоставления строк при последовательной обработке

текста. Эта идея применяется для образцов различных видов после подходящей предобработки.

Алгоритм КМП ищет вхождения образца x в тексте. После предобработки x текст обрабатывается в онлайновом режиме, и выводятся все найденные вхождения. Время работы линейно, поскольку количество сравнений букв не больше удвоенной длины текста. В приведенной ниже версии просто выводится «1» при каждом обнаружении x в тексте и «0» в противном случае.

Алгоритм работает за линейное время, но не в режиме реального времени из-за внутреннего цикла while, в котором обрабатывается один символ текста, что занимает некоторое время. Это называется *задержкой* алгоритма. Цель данной задачи – ограничить задержку, которая определяется как максимальное число сравнений, выполняемых в строке 4 для буквы a входного текста *text*.

```
KMP(непустые слова x, text)
 1  stbord ← StrictBorders(x)
 2  i ← 0
 3  for каждой буквы a текста text последовательно do
 4      while (i = |x|) или (i ≥ 0 и a ≠ x[i]) do
 5          i ← stbord[i]
 6      i ← i + 1
 7      if i = |x| then вывести 1 else вывести 0
```

Если вместо таблицы строгих границ *stbord* слова x (см. задачу 25) используется таблица границ *border* (см. задачу 19), то в худшем случае задержка равна $|x|$. Например, если слово $x = \mathtt{a}^m$ начинается в той же позиции, что фактор $\mathtt{a}^{m-1}\mathtt{b}$ текста, то буква b сравнивается со всеми буквами x. Но при использовании таблицы *stbord* задержка становится логарифмической.

Вопрос. Показать, что задержка алгоритма КМП при поиске слова x равна $\Theta(\log|x|)$.

[**Указание:** рассмотрите перекрывающиеся периоды и примените лемму о периодичности.]

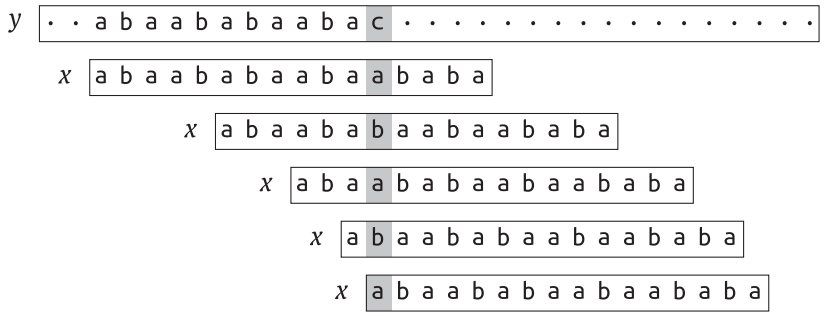
Решение

Нижняя граница задержки. В худшем случае оценка задержки $\Theta(\log|x|)$ достигается, например, когда x является префиксом слова Фибоначчи.

Пусть x ($|x| > 2$) – такое слово и k – такое целое число, что $F_{k+2} \le |x| + 1 < F_{k+3}$. Образец x является префиксом f_{k+1} (длиной F_{k+3}) вида $uabv$, где $uab = f_k$ для некоторых букв $a, b \in \{\mathtt{a}, \mathtt{b}\}$. Если ua совмещено с фактором uc слова *text* и $c \notin \{\mathtt{a}, \mathtt{b}\}$, то буква c неудачно сравнивается k раз попеременно с a и b. По порядку величины k равно $\log F_{k+2}$, а значит, имеет место порядок $\log|x|$, что и дает нижнюю границу.

Пример. Пусть $x = \mathtt{abaababaabaababa}$ – префикс f_6 ($|f_6| = F_8$). Имеем $F_7 = 13$, $|x| + 1 = 17$, $F_8 = 21$. Если префикс abaababaabaa совмещен с фактором abaaba-

baabac текста, то алгоритм КМП производит ровно $k = 5$ сравнений, прежде чем заняться буквой, следующей за c (см. рисунок).



Верхняя граница задержки. Для позиции i в слове x обозначим k наибольшее целое число, для которого $stbord^{k-1}[i]$ определена, а $stbord^k[i]$ нет. Мы покажем, что число k является верхней границей количества сравнений между $x[i]$ и буквой текста.

Сначала покажем, что если $stbord^2[i]$ определена, то префикс $u = x[0..i-1]$ удовлетворяет условию $|u| \geq stbord[i] + stbord^2[i] + 2$. Поскольку $stbord[i]$ и $stbord^2[i]$ – границы $x[0..i-1]$, то $p = |u| - stbord[i]$ и $q = |u| - stbord^2[i]$ являются периодами u. Если предположить, что $|u| < stbord[i] + stbord^2[i] + 2$, то $p + q - 1 \leq |u|$. Тогда, по лемме о периодичности, $q - p$ также является периодом u. Отсюда следует, что $x[stbord^2[i]] = x[stbord[i]]$ – буквы, находящиеся на расстоянии $q - p$ друг от друга в u, что противоречит определению $stbord$.

Рекуррентно применяя это неравенство, можно показать, что $|u| \geq F_{k+2} - 2$. Таким образом, $|x| + 1 \geq |u| + 2 \geq F_{k+2}$. Из классического неравенства $F_{n+2} \geq \Phi^n$ получаем верхнюю границу задержки $O(\log_\Phi |x|)$.

Примечания

Доказательство рассмотренной теоремы можно найти в работе [162] (см. также [74]). Алгоритмы, описанные в работах Simon [225] и Hancart [136], снижают верхнюю границу задержки до $\min\{\log_2 |x|, |alph(x)|\}$ благодаря использованию разреженного автомата сопоставления (см. задачу 27).

27. Разреженный автомат сопоставления

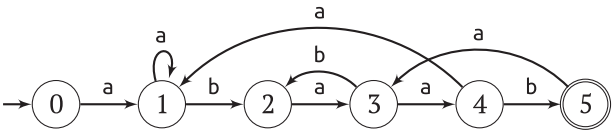
Самый распространенный метод поиска заданного образца в тексте, обрабатываемом последовательно, – использование автомата сопоставления с образцом. Таблицы границ в алгоритме КМП можно рассматривать как реализации такого автомата. Цель этой задачи – продемонстрировать другую технику реализации, позволяющую ускорить поиск. Реализованный подобным образом поиск работает заведомо не хуже КМП, т. е. требует линейного времени и количества сравнений букв, превышающего длину текста не более чем вдвое.

Автомат сопоставления с образом, или ***автомат сопоставления со строкой***, $\mathcal{M}(x)$ слова x над алфавитом A – это минимальный детерминированный автомат, допускающий слова, оканчивающиеся на x. Он допускает язык A^*x и имеет $|x| + 1$ состояний 0, 1, ..., $|x|$, начальное состояние 0 и единственное заключительное состояние $|x|$. Его таблица переходов состояний δ определена следующим образом (здесь i – состояние, a – буква):

$$\delta(i, a) = \max\{s + 1 : -1 \le s \le i \text{ и } x[0..s] \text{ – суффикс } x[0..i-1]\cdot a\}.$$

Заметим, что размер таблицы равен $\Omega(|x|^2)$, когда размер алфавита x такой же, как длина x. Но эта таблица очень разрежена, потому что большинство ее элементов равны нулю.

Ниже показан автомат сопоставления со строкой `abaab` длины 5 над алфавитом {`a`, `b`}. Помеченные дуги представляют ненулевые значения таблицы переходов. Всего существует пять прямых дуг (на главной прямой) и четыре обратные дуги. Для всех остальных дуг (на рисунке не показаны) конечное состояние нулевое. Если бы алфавит содержал третью букву, то все помеченные ей дуги также вели бы в состояние 0.



Вопрос. Показать, что таблица δ, ассоциированная с автоматом сопоставления строки длины n, имеет не более $2n$ ненулевых элементов и что эта граница точная.

Решение

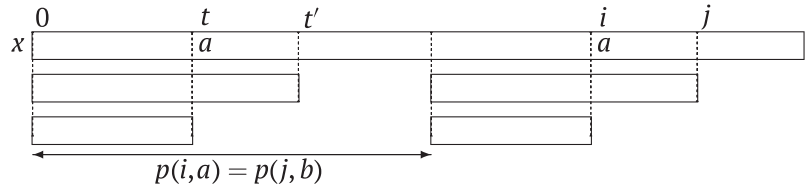
В автомате $\mathcal{M}(x)$ имеется n прямых дуг, соответствующих элементам $\delta(i, a) = i + 1$. Остальные дуги обратные и соответствуют элементам $0 < \delta(i, a) \le i$. Чтобы решить задачу, достаточно показать, что количество обратных дуг не больше n.

Наблюдение. Элемент $\delta(i, a) = t$, ассоциированный с обратной дугой, удовлетворяет условиям $x[t] = a$, $x[i] \ne a$ и $x[0..t-1] = x[i-t..i-1]$. Так как последнее слово является границей длины t слова $x[0..i-1]$, то $i - t$ – период (необязательно наименьший) $x[0..i-1]$, который мы обозначим $p(i, a)$.

В примере выше мы имеем для обратных дуг $p(1, \mathtt{a}) = 1$, $p(3, \mathtt{b}) = 2$, $p(4, \mathtt{a}) = 4$ и $p(5, \mathtt{a}) = 3$. Заметим, что $p(4, \mathtt{a}) = 4$ – не наименьший период $x[0..3] = \mathtt{abaa}$.

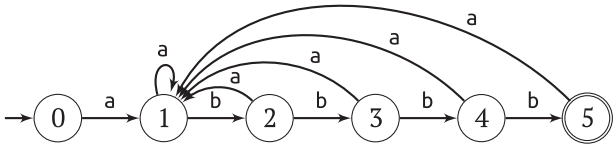
Чтобы найти верхнюю границу количества обратных дуг, заметим предварительно, что p – взаимно однозначная функция, отображающая обратные дуги на периоды префиксов x.

Положим $\delta(i, a) = t + 1$, где $0 \le t < i$, $\delta(j, b) = t'$, где $0 \le t' < j$. Предположим, что $p(i, a) = p(j, b)$, т. е. $i - t = j - t'$, и докажем, что $(i, a) = (j, b)$.



Действительно, если $i = j$, то имеем $t = t'$, и тогда $a = x[t] = x[t'] = b$; таким образом, $(i, a) = (j, b)$. В противном случае если, например, $i < j$, как на рисунке, то, поскольку $i - t = p(i, a) = p(j, b)$ – период $x[0..j - 1]$, мы сразу же получаем $x[t] = x[t + (i - t)] = x[i]$, что противоречит определению t .

Следовательно, p – взаимно однозначная функция, а поскольку ее значения изменяются от 1 до n, количество обратных дуг не может быть больше n, т. е. первая часть утверждения доказана.



Граница в этой оценке точная, потому что автомат сопоставления строки $\mathtt{ab}^{n-1}$ имеет ровно n обратных дуг в дополнение к n прямым.

Примечания

Разреженность таблицы переходов автомата сопоставления со строкой была подмечена в работе Simon [225]. Полный анализ выполнен Ханкартом в работе [136] (см. [74, глава 2]), который показал, как использовать этот факт для улучшения алгоритма сопоставления с образцом Кнута–Морриса–Пратта. [162]. Разреженность не обобщается на аналогичный автомат для конечного множества слов.

Описанный результат применим также к аналогичной таблице, используемой в алгоритме сопоставления с образцом Бойера–Мура (см. [74, 98, 134]).

28. Сопоставление со строкой, эффективное с точки зрения числа сравнений

Если доступ к текстовым данным ограничивается только сравнениями символов, то хорошо бы уменьшить количество сравнений в процессе поиска образца. Например, в алгоритме КМП (задача 26) в худшем случае производится $2|y| - 1$ сравнений букв при поиске предварительно обработанного образца x в тексте y. То же верно для поиска с использованием правильно реализованного автомата сопоставления со строкой.

В этой задаче показано, что для простых образцов количество сравнений с буквами легко можно уменьшить до $|y|$. Аналогичный подход, вкратце описанный в конце решения, позволяет уменьшить число сравнений букв до $\frac{3}{2}|y|$ для образцов общего вида.

Вопрос. Спроектировать алгоритм поиска всех вхождений двухбуквенного образца x в текст y, требующий не более $|y|$ сравнений на равенство.

[**Указание:** рассмотрите два случая: когда буквы совпадают и когда они различны.]

Решение

Рассмотрим два случая: когда образец x имеет вид aa и ab, где a и b – буквы и $a \neq b$.

В первом случае алгоритм КМП производит ровно $|y|$ сравнений при любой таблице границ. Заметим, что при наивном прямолинейном подходе число сравнений, вероятно, было бы близко к $2|y|$.

Во втором случае, когда $x = ab$ и $a \neq b$, алгоритм поиска выглядит следующим образом.

```
SEARCH-FOR-AB-IN(слово y, различные буквы a и b)
1  j ← 1
2  while j < |y| do
3      if y[j ] = b then
4          if y[j − 1] = a then
5              ab встречается в позиции j − 1 слова y
6          j ← j + 2
7      else j ← j + 1
```

Алгоритм можно рассматривать как рекурсивный:

- найти наименьшую позицию $j > 0$, для которой $y[j] = b$,
- проверить, верно ли, что $y[j - 1] = a$,
- затем рекурсивно искать ab в $y[j + 1..|y| - 1]$.

Заметим, что вычисление j на первом шаге рекурсивной версии требует ровно j сравнений, потому что сравнения с $y[0]$ нет.

Мы докажем, что алгоритм SEARCH-FOR-AB-IN производит не более $|y|$ сравнений символов, индукцией по длине y. Предположим, что найдено первое вхождение b в позиции l слова y. Для этого потребовалось l сравнений. Если учесть еще сравнение $y[l - 1] = a$, то получается всего $l + 1$ сравнений при поиске в слове $y[0..l]$ длиной $l + 1$.

Поскольку те же шаги применимы к слову $y[l + 1..|y| - 1]$, то, по предположению индукции, алгоритм выполняет еще не более $|y| - l - 1 = |y[l + 1..|y| - 1]|$ сравнений. Всего получается не более $|y|$ сравнений, что и требовалось доказать.

Образцы общего вида. Чтобы применить похожий метод к непустому образцу x общего вида, представим его в виде $x = a^k bu$, где u – слово, a и b – различные буквы и $k > 0$. Алгоритм заключается в том, что bu ищется в тексте (который просматривается слева направо) и в тех случаях, когда это имеет смысл, проверяется, встречается ли a^k непосредственно перед вхождениями bu. Для поиска bu можно использовать, например, алгоритм Морриса–Пратта (алгоритм КМП, в котором вместо *stbord* используется таблица *border*, см. задачу 26). Существует несколько реализаций этого метода. Но это чисто технический вопрос, так что детали мы опустим.

Примечания

Первый алгоритм сопоставления с образцом, в котором была достигнута верхняя граница $\frac{3}{2}n$ ($n = |y|$) количества сравнений букв, описан в работе Apostolico and Crochemore [14]. Он был улучшен до $\frac{4}{3}n - \frac{1}{3}m$ сравнений в работе Galil and Giancarlo [123]. Точная верхняя граница, $n + \frac{8}{3(m+1)}(n-m)$ сравнений, где $m = |x|$, доказана в работе Cole and Hariharan [60].

При традиционном ограничении, согласно которому поиск в тексте производится в онлайновом режиме, точная верхняя граница количества сравнений букв (очевидно, большая, чем приведенные выше наилучшие оценки) равна $\left(2 - \frac{1}{m}\right)n$; это доказано в работах Hancart [136] и Breslauer et al. [44].

29. Таблица строгих границ слова Фибоначчи

Таблица границ *border* (см. задачу 19) бесконечного слова Фибоначчи **f** имеет простую структуру, но таблица строгих границ *stbord* (см. задачу 25) на первый взгляд выглядит хаотично. В этой задаче исследуется простая связь между обеими таблицами, которая позволит быстро вычислить любой элемент таблицы *stbord*.

Ниже показаны таблицы периодов, границ и строгих границ префиксов слова Фибоначчи. Элементы с индексом l соответствуют префиксу $\mathbf{f}[0..l-1]$ длиной l.

i		0	1	2	3	4	5	6	7	8	9	10
$x[i]$		a	b	a	a	b	a	b	a	a	b	a
l	0	1	2	3	4	5	6	7	8	9	10	11
period$[l]$		1	2	2	3	3	3	5	5	5	5	5
border$[l]$	-1	**0**	0	**1**	1	2	**3**	2	3	4	5	**6**
stbord$[l]$	-1	**0**	-1	**1**	0	-1	**3**	-1	1	0	-1	**6**

Вопрос. Показать, как за логарифмическое время вычислить n-й элемент таблицы строгих границ бесконечного слова Фибоначчи **f**.

[**Указание:** рассмотрите позиции, в которых таблицы *border* и *stbord* совпадают.]

Решение

Позиции l в таблицах *border* и *stbord*, указывающие на один и тот же элемент, имеют особую важность для вычисления *stbord*[l]. Зная их, можно вычислить остальные элементы *stbord*.

Таблица *period* периодов префиксов **f** (определенная для $l > 0$ как *period*[l] = *per* (**f**[0..$l-1$]) = l – border [l]) имеет очень простую структуру, как показывает следующее наблюдение.

Наблюдение. Значения элементов таблицы *period* слова Фибоначчи можно рассматривать как слово

$$122333555558\ldots = 1^1\,2^2\,3^3\,5^5\,8^8\,13^{13}\,21^{21}\ldots,$$

являющееся конкатенацией серий. Каждая серия состоит из повторений числа Фибоначчи в количестве, равном самому этому числу. Первая серия соответствует числу $F_2 = 1$.

Заметим, что для $l > 0$ равенство *stbord*[l] = *border*[l] имеет место точно в тех позициях, где нарушается периодичность, т. е. в конце серии. Обозначим H возрастающую последовательность таких (положительных) позиций. В силу наблюдения над структурой периодов позиции в H равны

$$1, 1+2, 1+2+3, 1+2+3+5, 1+2+3+5+8, \ldots,$$

и нетрудно видеть, что

$H = (1,3,6,11,19,32,\ldots) = (n > 0 : n + 2$ – число Фибоначчи).

Теперь связь между таблицами *border* и *stbord* (продемонстрированную в задаче 25) можно по-другому выразить в виде

$$stbord[l] = \begin{cases} border[l], & \text{если } l \in H, \\ stbord[border[l]] & \text{в противном случае.} \end{cases}$$

Эта формула сразу же транслируется в алгоритм FibStrictBorders, который вычисляет *stbord*[n] для префикса длины n слова Фибоначчи. Вычисление *border*[n] = n − *period*[n] производится быстро вследствие структуры последовательности периодов. Для проверки того, является ли $n + 2$ числом Фибоначчи, можно хранить два самых больших числа Фибоначчи, не превышающих $n + 2$. При переходе к меньшим значениям в процессе рекурсии оба числа Фибоначчи изменяются соответственно.

```
FibStrictBorders(натуральное число n)
 1  if n = 0 then
 2      return −1
 3  elseif n + 2 – число Фибоначчи then
 4      return n − period[n]
 5  else return FibStrictBorders(n − period[n])
```

Следующее наблюдение позволит нам оценить время работы этого алгоритма.

Наблюдение. $period[n]/n \geq \lim_{n\to\infty} \frac{F_{n-2}}{F_n} \geq \frac{1}{3}$.

Отсюда следует, что $n - period[n] \leq \frac{2}{3}n$, а значит, глубина рекурсии растет логарифмически. Поэтому полное время работы алгоритма также логарифмически зависит от n.

30. Слова с подстановочными переменными

В этой задаче иллюстрируется гибкость понятия таблицы границ и быстрого алгоритма вычисления этой таблицы. Эта таблица – ценная вещь при проектировании эффективных алгоритмов сопоставления с образцом, в данном случае поиска образца, содержащего переменную.

Мы рассматриваем слова над алфавитом `A` = {`a`,`b`, ...}, буквы которого интерпретируются как подстановочные переменные (singleton variable), т. е. представляют различные неизвестные буквы алфавита.

Говорят, что два слова u и v эквивалентны, и обозначают это $u \equiv v$, если существует такой биективный морфизм букв $h : alph(u)^* \to alph(v)^*$, что $h(u) = v$. Например, `aacbaba` ≡ `bbdabab` относительно следующего морфизма h, отображающего `A`* в себя: h(`a`) = `b`, h(`b`) = `a`, h(`c`) = `d` и h(`d`) = `c`. Если $alph(u) = alph(v)$ и $u \equiv v$, то одно слово переходит в другое в результате перестановки букв.

Задача сопоставления с образцом естественно переопределяется следующим образом: пусть дано слово-образец x и текст y, требуется проверить, эквивалентен ли какой-то фактор z текста y образцу x: $z \equiv x$. Например, образец x = `aacbaba` встречается в тексте y = `babbdababbacb`, потому что существует фактор z = `bbdabab`, эквивалентный x.

Вопрос. Предположим, что алфавит допускает сортировку за линейное время. Спроектировать алгоритм с линейным временем работы, который ищет в тексте образец с подстановочными переменными.

[**Указание:** придумайте вариант таблицы границ, подходящий для этой задачи.]

Решение

Решение основано на понятии ***таблицы переменных границ***. Она обозначается *vbord* и для параметра m и непустого слова $w = w[0..n-1]$ определена на длинах $l = 0, \ldots, n$ его префиксов следующим образом: $vbord[0] = -1$, а для $0 < j \leq n$ $vbord[j]$ равна

$$\max\{t : 0 \leq t < \min\{j, m+1\} \text{ и } w[0..t-1] \equiv w[j-t+1..j]\}.$$

Иными словами, $vbord[j]$ – длина l самого длинного собственного суффикса длины, не большей m, слова $w[1\,..\,j]$, эквивалентного его префиксу длины l.

Причина ограничения на длину суффикса (не больше m) становится понятна при сравнении с образцом длины m.

Ниже показана таблица *vbord* для $w =$ abaababbabba и $m = 4$:

i		0	1	2	3	4	5	6	7	8	9	10	11
$w[i]$		a	b	a	a	b	a	b	b	a	b	b	a
l	0	1	2	3	4	5	6	7	8	9	10	11	12
$vbord[l]$	-1	0	1	2	1	2	3	3	4	2	3	4	2

В алгоритме, который строит *vbord*, используется еще одна таблица *pred*, определенная для $0 \le i < n$ следующим образом:

$$pred[i] = \max\{t : t < i \text{ и } w[i] = w[t]\} \cup \{-1\}.$$

Например, если $w =$ abcaabac, то $pred = [-1,-1,-1,0,3,1,4,2]$.

Наблюдение 1. Таблицу *pred* можно вычислить за линейное время.

Обозначим ∇_m следующий предикат: $\nabla_m(i, l) =$ TRUE тогда и только тогда, когда

$$(pred[l] \ge 0 \ \&\ i = pred[i] + k) \text{ или } (pred[l] = -1 \ \&\ pred[i] < i - l),$$

где $k = l - pred[l]$.

Формальное доказательство следующего простого утверждения оставляем читателю.

Наблюдение 2. Предположим, что $w[0..l-1] \equiv w[i-l..i-1]$ и $l < m$. Тогда $w[0..l] \equiv w[i-l..i] \Leftrightarrow \nabla_m(i, l)$.

В классическом алгоритме вычисления таблицы границ (см. задачу 19) мы можем заменить проверку на неравенство символов предикатом ∇_m и получим следующий алгоритм вычисления таблицы переменных границ, требующий линейного времени. Его правильность вытекает из наблюдения 2.

```
VarBorders(непустое слово x)
 1  vbord[0] ← −1
 2  for i ← 0 to |x| − 1 do
 3      l ← vbord[i]
 4      while l ≥ 0 и not ∇m(i, l) do
 5          l ← vbord[l]
 6      if l < m then
 7          vbord[i + 1] ← l + 1
 8      else vbord[i + 1] ← vbord[l + 1]
 9  return vbord
```

Как это связано с сопоставлением с образцом. Задача поиска образца x длины m в тексте y решается с помощью таблицы *vbord*. Таблица переменных границ строится для слова $w = xy$ и параметра m. Тогда

$$x \equiv y[i-m+1..i] \Leftrightarrow vbord[m+i-1] = m.$$

Поэтому поиск в y образца x с подстановочными переменными сводится к вычислению таблицы *vbord*.

Примечания
Описанная здесь задача – упрощенный вариант так называемого параметрического сопоставления с образцом, см. [24]. В более общей постановке некоторые символы считаются подстановочными переменными, а остальные – постоянными буквами. Это предмет задачи 2.

31. Образцы, сохраняющие порядок

Для поиска во временном ряде или в списке значений образцов, представляющих некоторые отклонения значений, требуется переопределить понятие образца. Нас интересует распознавание пиков, аномалий, второй волны рецессии или других особенностей затрат, ставок и т. п.

В рассматриваемой задаче слова выбираются из сортируемого за линейное время алфавита Σ целых чисел. Говорят, что два слова u и v одинаковой длины над алфавитом Σ *порядково эквивалентны* ($u \approx v$), если

$$u[i] < u[j] \Leftrightarrow v[i] < v[j]$$

для всех пар позиций i, j в словах. Например,

$$5\,2\,9\,4\,3 \approx 6\,1\,7\,5\,2,$$

откуда, в частности, следует, что центральное значение является наибольшим в обоих словах.

Задача о сопоставлении с образцом с сохранением порядка естественно ставится следующим образом: пусть дан образец $x \in \Sigma^*$ и текст $y \in \Sigma^*$, проверить, верно ли, что x порядково эквивалентен некоторому фактору y. Например, слово 5 2 9 4 3 встречается в этом смысле в позиции 1 текста

4 6 1 7 5 2 9 8 3

и больше нигде. Так, оно не встречается в позиции 4, потому что $5 < 8$, тогда как в соответствующих позициях образца соотношение обратное: $5 > 4$.

Для простоты предположим, что буквы в рассматриваемых словах попарно различны (т. е. слова являются перестановками множеств составляющих их букв).

Вопрос. Спроектировать алгоритм, который за линейное время выполняет сопоставление с образцом с сохранением порядка.

[**Указание:** придумайте вариант таблицы границ, подходящий для этой задачи.]

Решение

Эта задача основана на понятии ***таблицы OP-границ***. Для непустого слова $w = w[0..n-1] \in \Sigma^*$ таблица *opbord* определяется следующим образом: $opbord[0] = -1$, а для $0 < l \leq n$ $opbord[l] = t$, где $t < l$ – наибольшее целое число, для которого $w[0..t-1] \approx w[l-t+1..l]$.

Ниже показана таблица *opbord*, ассоциированная со словом $w =$ `1 3 2 7 11 8 12 9`.

i		0	1	2	3	4	5	6	7
$w[i]$		1	3	2	7	11	8	12	9
l	0	1	2	3	4	5	6	7	8
$opbord[l]$	-1	0	1	1	2	2	3	4	3

Для решения задачи нам понадобятся две вспомогательные таблицы, ассоциированные с w:

$LMax[i] = j$, где $w[j] = \max\{w[k] : k < i$ и $w[k] \leq w[i]\}$

и $LMax[i] = -1$, если такого $w[j]$ не существует, и

$LMin[i] = j$, где $w[j] = \min\{w[k] : k < i$ и $w[k] \geq w[i]\}$

и $LMin[i] = -1$, если такого $w[j]$ не существует.

Наблюдение 1. Обе таблицы *LMax* и *LMin* можно вычислить за линейное время.

Переопределим предикат ∇ (введенный в задаче 30) следующим образом:

$$\nabla n(i, l) = 1 \Leftrightarrow w[p] \leq w[i] \leq w[q],$$

где $p = LMax[l]$ и $q = LMin[l]$ (если p или q равны -1, то считается, что соответствующее неравенство удовлетворяется).

Оставляем читателю доказательство следующего простого факта (см. примечания).

Наблюдение 2. Предположим, что $w[0..l-1] \approx w[i-l..i-1]$ и $l < n$. Тогда

$$w[0..l] \approx w[i-l..\text{i}] \Leftrightarrow \nabla_n(i, l).$$

Сопоставление с образцом. В заключение отметим, что проверка порядковой эквивалентности w некоторому фактору текста совпадает с алгоритмом сопоставления с образцом с подстановочными переменными из задачи 30 с точностью до определения предиката ∇.

Примечания

Описанный алгоритм является вариантом сопоставления с образцом с сохранением порядка из работы Kubica et al. [170], в которой доказано и наблюдение 2 (см. также [54, 139, 160]). Сама задача вкупе с возможностью несовпадения рассматривается в работе Gawrychowski and Uznanski [129]. Суффиксные деревья для индексирования с сохранением порядка введены в работе [81].

32. Параметрическое сопоставление

Эта задача является более общим и более сложным вариантом задачи 30, когда некоторые символы неизвестны, а другие фиксированы. В некоторых контекстах поиска фиксированного образца в тексте недостаточно, поэтому параметрическое сопоставление дает эффективное решение, позволяя включать переменные в образцы. Первоначально задача была поставлена для обнаружения дубликатов в коде, когда, например, вместо оригинальных имен подставляются идентификаторы.

Пусть A и V – два непересекающихся алфавита: A содержит постоянные буквы, а V – переменные буквы. Будем предполагать, что ни один алфавит не содержит целых чисел. Слово над алфавитом $A \cup V$ называется ***параметрическим словом***, или p-словом. Говорят, что два p-слова x и y совпадают, или p-совпадают, если x можно преобразовать в y, применив взаимно однозначное отображение к символам V, встречающимся в x.

Например, если $A = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$ и $V = \{t,u,v,w\}$, то $\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ и $\mathtt{a}w\mathtt{b}u\mathtt{a}w\mathtt{b}$ p-совпадают относительно отображения, переводящего u в w и v в u. Но слова $\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ и $\mathtt{a}v\mathtt{b}w\mathtt{a}z\mathtt{b}$ не p-совпадают, потому что u нельзя отобразить одновременно в v и z.

Задача о параметрическом сопоставлении с образцом формулируется следующим образом: дан образец $x \in (A \cup V)^*$ и текст $y \in (A \cup V)^*$, найти все p-вхождения x в y, т. е. все позиции j в y, $0 \le j \le |y| - |x|$, для которых x и $y[j..j + |x| - 1]$ p-совпадают.

Например, для $y = \mathtt{a}z\mathtt{b}u\mathtt{a}z\mathtt{b}z\mathtt{a}v\mathtt{b}w\mathtt{a}v\mathtt{b}$ образец $x = \mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$ встречается в позиции 0, если отобразить u в z и v в u, и в позиции 8, если отобразить u в v и v в w.

Вопрос. Спроектировать алгоритм, который решает задачу о параметрическом сопоставлении с образцом и требует линейного времени для фиксированного алфавита.

Решение

Задачу можно решить с помощью алгоритма КМП (см. задачу 26), если аккуратно выбрать кодирование переменных.

Для слова $x \in (A \cup V)^*$ обозначим $prev(x)$ слово $z \in (A \cup \mathbf{N})^*$, определенное следующим образом (здесь i – позиция в x):

$$z[i] = \begin{cases} x[i], & \text{если } x[i] \in A, \\ 0, & \text{если } x[i] \in V \text{ не встречается в } x[0..i-1], \\ i - \max\{j < i : x[j] = x[i]\}, & \text{если } x[i] \in V. \end{cases}$$

Например, $prev(\mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}) = \mathtt{a0b0a4b}$. Слово $prev(x)$ можно вычислить за время $O(|x| \times \min\{\log|x|, \log|V|\})$ с использованием памяти объемом $O(|x|)$. Если алфавит V фиксирован, то это сводится к времени $O(|x|)$ и памяти $O(|V|)$.

Обозначим $z_i = z[i..|z| - 1]$ суффикс $z \in (A \cup \mathbf{N})^*$. Тогда $shorten(z_i)$ – слово s, определенное для $0 \le j \le |z_i| - 1$ следующим образом: $s[j] = z_i[j]$, если $z_i[j] \le j$, и $s[j] = 0$ в противном случае. Например, если $z = \mathtt{a0b0a4b}$, то $z_3 = \mathtt{0a4b}$ и $shorten(z_3) = \mathtt{0a0b}$.

Наблюдение. Пусть $z = prev(x)$. Буква $x[i] \in V$ p-совпадает с буквой $y[j] \in V$ слова y, если выполняется одно из двух условий:

- $z[i] = 0$;
- $z[i] \neq 0$ и $y[j - z[i]] = y[j]$.

В силу этого наблюдения, чтобы установить p-совпадение x в позиции j слова y, достаточно времени $O(|x|)$. Повторяя алгоритм КМП с ***таблицей параметрических границ*** *pbord*, получаем решение, требующее линейного времени. Для $x \in (A \cup V)^*$ эта таблица определяется следующим образом: $pbord[0] = -1$, а для $1 \leq i \leq |x|$ $pbord[i] = j$, где $j < i$ – наибольшее целое число, для которого $prev(x[0..j-1])$ – суффикс $shorten(prev(x[i-j..i-1]))$. Ниже показаны таблицы *prev* и *pbord* для $x = \mathtt{a}u\mathtt{b}v\mathtt{a}u\mathtt{b}$:

i	0	1	2	3	4	5	6	7
$x[i]$	a	u	b	v	a	u	b	
$prev(x)[i]$	a	0	b	0	a	4	b	
$pbord[i]$	-1	0	0	0	0	1	2	3

Зная *pbord* для p-слова x, следующий алгоритм сообщает все позиции, в которых слово x встречается в слове y.

```
ParameterisedMatching(x, y ∈ (A ∪ V)*)
 1  z ← prev(x)
 2  i ← 0
 3  for j ← 0 to |y| − 1 do
 4      while i ≥ 0 и not ((x[i], y[j] ∈ A и x[i] = y[j])
                или (x[i], y[j] ∈ V и
                (z[i] = 0 или y[j − z[i]] = y[j]))) do
 5          i ← pbord[i]
 6      i ← i + 1
 7      if i = |x| then
 8          сообщить о вхождении x в позиции j − |x| + 1
 9          i ← pbord[i]
```

Доказательство правильности и анализ сложности алгоритма ParameterisedMatching проводятся так же, как для алгоритма КМП. А таблицу параметрических границ *pbord* можно вычислить, адаптировав алгоритм Borders, вычисляющий таблицу обычных границ (см. задачу 19).

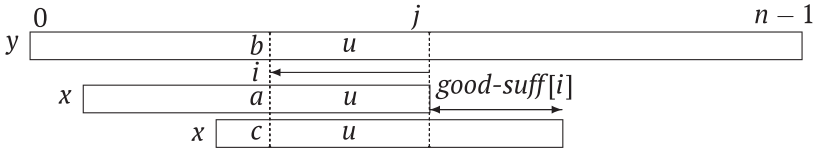
Примечания

Параметрическое сопоставление с образцом впервые было формализовано в работах B. Baker [23, 24]. Там предложено решение на основе суффиксных деревьев для офлайновой постановки задачи. Первое решение для онлайновой постановки было дано в работе [11]. Описанное здесь решение впервые было опубликовано в работе [145] вместе с решением для онлайнового многократно параметрического сопоставления с образцом. Читатель может обратиться к обзору [188].

33. Таблица хороших суффиксов

Алгоритм Бойера–Мура (алгоритм BM в задаче 34) применяет к тексту стратегию скользящего окна с целью найти вхождения образца. Для ускорения поиска образец необходимо подвергнуть предобработке.

На каждом шаге алгоритм сравнивает образец и окно текста, вычисляя их наибольший общий суффикс u. Если $u = x$, то имеет место совпадение. В противном случае образец $x[0..m-1]$ совмещается с окном, т. е. фактором $y[j-m+1..j]$ текста; при этом au является суффиксом x, bu – суффиксом окна, где буквы a и b различны.



Для продолжения поиска алгоритм BM сдвигает окно на период x в случае совпадения, а иначе к фактору bu текста, чтобы пропустить позиции окна, в которых вхождения x заведомо не может быть. Для этого используется **таблица хороших суффиксов** *good-suff*, определенная для позиции i слова x и суффикса $u = x[i+1..m-1]$ следующим образом:

$$good\text{-}suff[i] = \min\{|v| : x \text{ – суффикс } uv \text{ или } cuv \text{ – суффикс } x,\ c \neq x[i]\}.$$

Условие «cuv – суффикс x» при $c \neq a = x[i]$ гарантирует, что когда буква c будет совмещена с буквой $b = y[j-m+1+i]$ после сдвига окна, точно такое же несовпадение не повторится сразу же (см. рисунок). Заметим, что из определения следует, что $good\text{-}suff[0] = per(x)$.

Вопрос. Спроектировать алгоритм, который вычисляет таблицу хороших суффиксов для слова за линейное время.

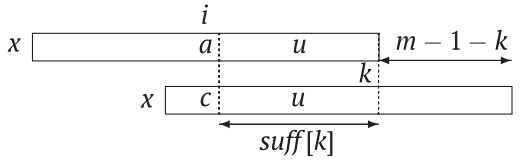
[**Указание:** воспользуйтесь обращенной таблицей префиксов x^R.]

Решение

В решении используется таблица суффиксов x, *suff*, симметричная таблице префиксов (см. задачу 22), которая определена для позиции i следующим образом: $suff[i] = |lcs(x[0..i], x)|$, где *lcs* – ***наибольший общий суффикс*** x и $x[0..i]$.

Пример. Ниже показаны таблицы *suff* и *good-suff* для слова `baacababa`

i	0	1	2	3	4	5	6	7	8
$x[i]$	b	a	a	c	a	b	a	b	a
$suff[i]$	0	2	1	0	1	0	3	0	9
$good\text{-}suff[i]$	7	7	7	7	7	2	7	4	1



Наблюдение. Таблицы *good-suff* и *suff* тесно связаны (см. рисунок, на котором $i = m - 1 - suff[k]$): $good\text{-}suff[m - 1 - suff[k]] \leq m - 1 - k$.

Тогда вычисление таблицы *good-suff* сводится к применению этого неравенства и выполняется алгоритмом GoodSuffixes. Чтобы получить наименьшее значение $m - 1 - k$, таблица *suff* просматривается в направлении возрастания позиций k (строки 8–9), после того как *good-suff* заполнена периодами x (строки 3–7).

```
GoodSuffixes(непустое слово x, его таблица суффиксов suff)
 1  m ← |x|
 2  p ← 0
 3  for k ← m − 2 to −1 do
 4      if k = −1 или suff[k] = k + 1 then
 5          while p < m − 1 − k do
 6              good-suff[p] ← m − 1 − k
 7              p ← p + 1
 8  for k ← 0 to m − 2 do
 9      good-suff[m − 1 −suff[k]] ← m − 1 − k
10  return good-suff
```

Вычисление занимает время $O(|x|)$, потому что таблицу *suff* можно вычислить за линейное время (как и таблицу *pref* в задаче 22), а этот алгоритм также требует линейного времени, коль скоро *suff* известна.

Примечания

Таблица *good-suff* часто ассоциируется с эвристикой, предложенной Бойером и Муром [41], чтобы учесть несовпадение с буквой b (см. также [162]). На самом деле это можно сделать для большинства методов сопоставления с образцом.

Первый точный алгоритм вычисления таблицы хороших суффиксов был предложен в работе Rytter [212].

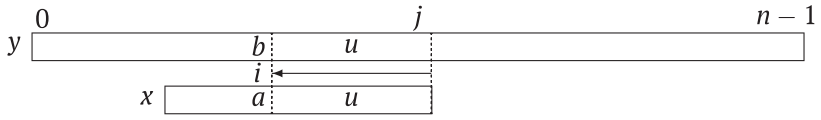
Таблица *good-suff* – важный элемент алгоритма БМ. Однако в приведенном выше определении несовпадения с буквой b не учитываются. Это можно сделать, применив методы, связанные с разреженным автоматом сопоставления (см. задачу 27), для чего понадобится дополнительная память объема $O(|x|)$, не зависящего от размера алфавита.

Таблица *suff* используется в более эффективном алгоритме БМ, описанном в работе Apostolico and Giancarlo [16], для которого максимальное количество сравнений букв равно $1.5|y|$ (см. [89]).

34. Худший случай в алгоритме Бойера–Мура

Алгоритм сопоставления с образцом Бойера–Мура основан на технике, которая позволяет строить самые быстрые алгоритмы поиска для фиксированных образцов. Его основное свойство – просмотр образца в обратном направлении после совмещения с фактором текста, в котором производится поиск. Типичная предобработка образца показана в задаче 33.

При поиске фиксированного образца x длины m в тексте y длины n в общем случае x совмещается с фактором (окном) y, заканчивающимся в позиции j (см. рисунок). Алгоритм вычисляет наибольший общий суффикс (lcs) x и y, возможно, сообщает о вхождении образца и сдвигает окно в направлении конца y на основе информации, вычисленной во время предобработки и собранной в процессе просмотра текста, не теряя ни одного вхождения x. Алгоритм BM реализует эту идею с помощью таблицы *good-suff* из задачи 33.




```
BM(непустые слова x, y и их длины m, n)
 1  j ← m − 1
 2  while j < n do
 3      i ← m − 1 − |lcs(x, y[j − m + 1..j])|
 4      if i < 0 then
 5          сообщить о вхождении x в позиции j − m + 1 текста y
 6          j ← j + per(x) ▷ per(x) = good-suff[0]
 7      else j ← j + good-suff[i]
```

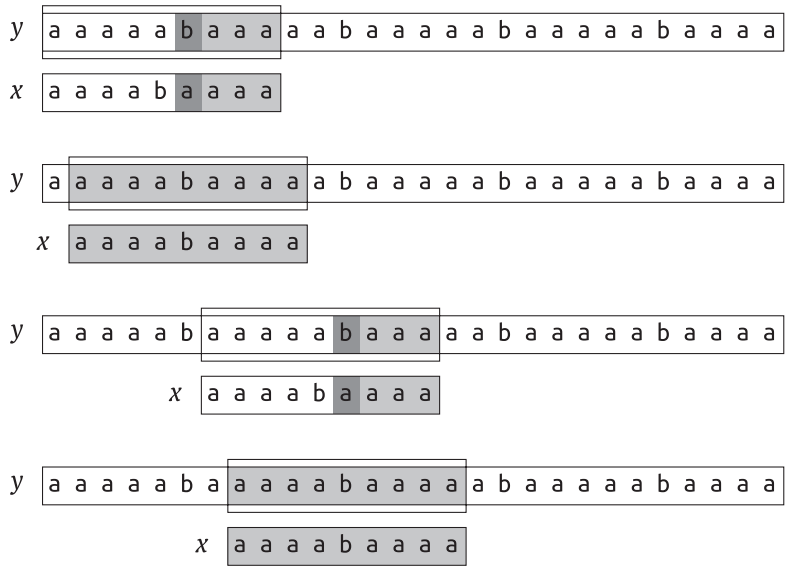
После того как позиция j в тексте y обработана, если было найдено вхождение x, алгоритм естественно сдвигает окно на расстояние $per(x)$. Если же вхождения не было, то расстояние $good\text{-}suff[i]$ зависит от фактора bu текста y. Значение $per(x)$ и массив *good-suff* вычисляются во время предобработки до начала поиска.

Вопрос. Приведите примеры непериодических образцов и текста y, для которого алгоритм BM выполняет в строке 3 примерно $3|y|$ сравнений букв, чтобы вычислить наибольший общий суффикс.

Решение

Положим $x = \mathtt{a}^{k-1}\mathtt{b}\mathtt{a}^{k-1}$ и $y = \mathtt{a}^{k-1}(\mathtt{a}\mathtt{b}\mathtt{a}^{k-1})^l$, где $k \geq 2$. Тогда $m = 2k - 1$ и $n = l(k + 1) + (k - 1)$.

Пример. Положим $k = 5$ и $l = 4$ при рассмотрении образца $\mathtt{a}^4\mathtt{ba}^4$ длины 9 и текста $\mathtt{a}^4(\mathtt{aba}^4)^4$ длины 28. На рисунке показано начало поиска, в ходе которого произведено в общей сложности $4 \times 13 = 52$ сравнения букв.



Рассмотрим позицию $j = (k - 3) + p(k + 1)$ в тексте y, где $p \geq 1$. Имеем $y[j - m + 1..j] = \mathtt{a}^k\mathtt{ba}^{k-2}$ и $|lcs(x, y[j - m + 1..j])| = k - 2$, вычислен с помощью $k - 1$ сравнений букв. Величина сдвига окна равна $good\text{-}suff[m - k - 1] = 1$, при этом j обновляется до $(k - 2) + p(k + 1)$, а $y[j - m + 1..j]$ становится равным $\mathtt{a}^{k-1}\mathtt{ba}^{k-1}$. На этот раз для вычисления суффикса $|lcs(x, y[j - m + 1..j])| = m$ понадобилось m сравнений букв, а величина следующего сдвига равна $per(x) = k$, так что $j = (k - 3) + (p + 1)(k + 1)$.

Оба шага вместе требуют $k - 1 + m = 3k - 2$ сравнений и приводят к аналогичной ситуации, в которой тот же самый процесс повторяется для каждого из $l - 1$ первых вхождений фактора $\mathtt{aba}^{k-1}$ (длины $k + 1$) текста y. Для последнего вхождения производится $(k - 1) + (k + 1) = 2k$ сравнений, а для префикса y длины $k - 1$ производится $k - 2$ сравнений. В общем и целом алгоритм BM с такими входными данными x и y выполняет $\dfrac{3k-2}{k+1}(n - k + 1) = \left(n - \dfrac{m-1}{2}\right)\left(3 - \dfrac{10}{m+3}\right)$ сравнений, что и требовалось доказать.

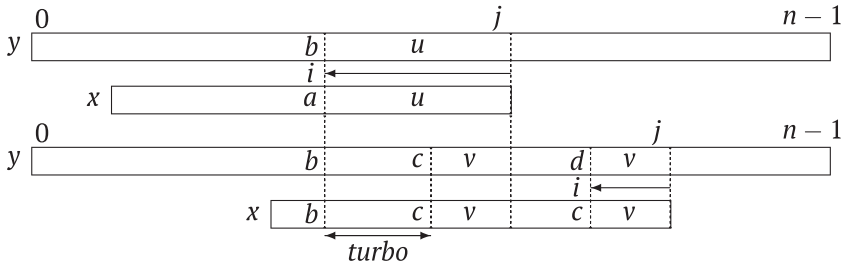
Примечания

Алгоритм Бойера–Мура описан в работе [41] (см. также [162]). Доказательство верхней границы числа сравнений $3n$ при поиске непериодического образца в тексте длины n дано в работе Cole [59]. Подробные описания алгоритма Бойера–Мура и его вариантов можно найти в классических учебниках по алгоритмам работы со строками [74, 96, 98, 134, 228].

35. Алгоритм Turbo-BM

В этой задаче показано, как ценой очень небольшой модификации алгоритма Бойера–Мура можно получить гораздо более быстрый алгоритм, в котором используется метод нахождения всех вхождений образца в текст, подобный алгоритму BM из задачи 34. Первоначальная цель алгоритма Бойера–Мура заключалась в нахождении первого вхождения образца. Известно, что эта цель достигается за время, линейно зависящее от длины текста. Но для нахождения всех вхождений в худшем случае может потребоваться квадратичное время, особенно для периодических образцов. Все дело в том, что алгоритм забывает проделанное раньше, когда сдвигает окно в следующую позицию.

Цель алгоритма Turbo-BM – добиться линейного времени поиска, адаптировав алгоритм BM, не изменяя процедуру предобработки образца с вычислением таблицы *good-suff*. Нужна только дополнительная память постоянного объема, которая используется во время поиска.



В алгоритме Turbo-BM используются две длины – $mem = |u|$ предыдущего совпадения суффикса u и $l = |v|$ текущего совпадения суффикса v – для вычисления $\max\{good\text{-}suff[i], |u| - |v|\}$.

В примере ниже первое совпадение $u =$ `bababa` длины 6 приводит к сдвигу на величину $4 = good\text{-}suff[4]$. После сдвига окна одно новое совпадение $v =$ `a` дало бы сдвиг длины $1 = good\text{-}suff[9]$. Однако применяется ***турбосдвиг*** на величину $turbo = 6 - 1 = 5$.

```
y   b a b a b b a b a b a a b a a . . . . .
x   b b a b a b a b a b a
              u
y   b a b a b b a b a b a a b a a . . . . .
x           b b a b a b a b a b a
                  u           v
```

Вопрос. Показать, что алгоритм Turbo-BM правильно сообщает обо всех вхождениях образца x в текст y.

```
TURBO-BM(непустые слова x, y и их длины m, n)
 1  (j, mem) ← (m − 1,0)
 2  while j < n do
 3      ▷ пропускаем участок памяти, если mem > 0
 4      i ← m − 1 − |lcs(x, y[j − m + 1..j])|
 5      if i < 0 then
 6          сообщить о вхождении x в позиции j − m + 1 текста y
 7          (shift, l) ← (per(x), m − per(x))
 8      else (shift, l) ← (good-suff[i], m − i − 1)
 9      turbo ← mem − (m − i − 1)
10      if turbo > shift then
11          (j, mem) ← (j + turbo, 0)
12      else (j, mem) ← (j + shift, l)
```

Решение

Доказательство правильности алгоритма TURBO-BM основано на аналогичном доказательстве для оригинального алгоритма и опирается на тот факт, что ни одно вхождение образца x в текст y не будет пропущено при сдвиге окна на *turbo* позиций (строка 11).

Действительно, при выполнении строки 11 $turbo > 1$, потому что $turbo > shift \geq 1$. Тогда cv – собственный суффикс u в x и y, поскольку вхождения u совмещены (см. рисунок). Суффикс $uzcv$ образца x, где z – слово, а c – буква, имеет период $|zcv|$, т. к. u – суффикс x. Он совмещен с фактором $uz'dv$ текста y, где z' – слово, а d – буква. Но поскольку $c \neq d$, $|zcv|$ не является периодом последнего фактора.

Таким образом, суффикс $uzcv$ образца x не может покрывать обе буквы c и d, встречающиеся в y, и это показывает, что конечная позиция следующего возможного вхождения x в y находится не раньше позиции $j + turbo$, что и требовалось доказать.

Примечания

Было предложено несколько решений, чтобы справиться с квадратичным временем алгоритма Бойера–Мура в худшем случае. Первое решение описано в работе Galil [122], где приведен вариант с линейным временем. Еще одно решение (Apostolico and Giancarlo [16]) требует дополнительной памяти, объем которой линейно зависит от длины образца. Память нужна как во время предобработки, так и во время поиска, а общее количество сравнений букв не превышает $1.5n$ [89].

Алгоритм TURBO-BM [70], безусловно, требует меньше всего модификаций оригинального алгоритма. Он не только работает за линейное время, но и выполняет не более $2n$ сравнений букв на этапе поиска (см. [74, 96]), расплачиваясь за это дополнительной памятью.

36. Сопоставление с образцом при наличии универсального символа

В этой задаче слова выбираются из алфавита целых положительных чисел с дополнительной буквой $*$. Буква $*$, называемая универсальным символом (или джокером), сопоставляется с любой буквой алфавита, включая себя.

Сопоставление с образцом при наличии универсального символа подразумевает поиск в тексте y всех вхождений образца x в предположении, что оба слова могут содержать универсальные символы. Обозначим $m = |x|$ и $n = |y|$.

Пример. `ab*b` входит в `abaaba*cbcb` только в позициях 3 и 5.

В отличие от других алгоритмов сопоставления со строками, решения этой задачи часто включают использование арифметических операций свертки: если даны две последовательности B и C длиной не более n, то сверткой называется последовательность A длиной $2n$, i-й элемент которой, $0 \le i \le 2n-1$, определяется следующим образом:

$$A[i] = \sum_{j=0}^{n-1} B[j] \cdot C[i+j].$$

Предполагается, что все элементарные арифметические операции выполняются за постоянное время и что вычислить свертку последовательностей длины n можно за время $O(n \log n)$.

Вопрос. Показать, как задачу о сопоставлении с образцом при наличии универсальных символов можно за линейное время свести к задаче о вычислении свертки.

Решение

После замены универсального символа на нуль определим последовательность $A[0..n-m]$ следующим образом: $A[i] = \sum_{j=0}^{m-1} x[j] \cdot y[i+j] \cdot (x[j] - y[i+j])^2$. Это выражение равно

$$\sum_{j=0}^{m-1} x[j]^3 y[i+j] - 2\sum_{j=0}^{m-1} x[j]^2 y[i+j]^2 + \sum_{j=0}^{m-1} x[j] \cdot y[i+j]^3.$$

Вычисление можно выполнить, вычислив три свертки, на что потребуется время $O(n \log n)$. Связь между последовательностью A и нашей задачей вытекает из следующего наблюдения.

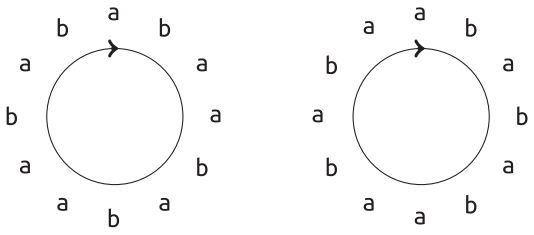
Наблюдение. $A[i] = 0$ тогда и только тогда, когда образец x встречается в позиции i текста y.

Действительно, $A[i]$ равно нулю тогда и только тогда, когда каждый член $x[j] \cdot y[i+j] \cdot (x[j] - y[i+j])^2$ равен нулю, а это означает, что либо хотя бы одно из чисел $x[j]$ и $y[i+1]$ (первоначально равных универсальному символу) равно нулю, либо они равны между собой. Это соответствует совпадению букв и доказывает правильность алгоритма сведения.

Примечания
Применение свертки к задаче о сопоставлении с образцом впервые было предложено в работе Masek and Paterson [186]. Описанное здесь упрощение взято из работы [58].

37. Циклическая эквивалентность

Два слова называются циклически эквивалентными, если одно является сопряженным к другому (получается из него ротацией). Проверка на циклическую эквивалентность встречается в некоторых задачах сопоставления с образцом, а также в алгоритмах на графах, например для проверки изоморфизма ориентированных помеченных графов (в этом случае проверка применяется к циклам в графе).



На рисунке показаны два циклически эквивалентных слова. Следующий алгоритм проверяет циклическую эквивалентность, пользуясь упорядоченностью алфавита.

```
CyclicEquivalence(непустые слова u, v длины n)
 1  (x, i, y, j) ← (uu, 0, vv, 0)
 2  while i < n и j < n do
 3      k ← 0
 4      while k < n и x[i + k] = y[j + k] do
 5          k ← k + 1
 6      if k = n then
 7        return TRUE
 8      if x[i + k] > y[j + k] then
 9          i ← i + k + 1
10      else j ← j + k + 1
11  return FALSE
```

Вопрос. Показать, что алгоритм CyclicEquivalence проверяет, что два слова циклически эквивалентны за линейное время, требуя дополнительной памяти постоянного объема.

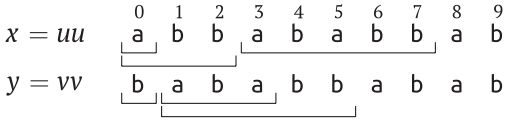
[**Указание:** рассмотрите сопряженные слова Линдона.]

Если в алфавите нет очевидного отношения порядка, то полезно иметь решение, основанное на более простом сравнении символов.

Вопрос. Как можно проверить циклическую эквивалентность двух строк, не прибегая к упорядоченности, а ограничиваясь только сравнениями $=$ и $\neq$?

Решение

Применим алгоритм CyclicEquivalence к словам $u =$ abbab и $v =$ babab и посмотрим, что происходит с парами индексов (i, j) в словах $x = uu$ и $y = vv$ соответственно.



Первоначально $(i, j) = (0, 0)$. После первого выполнения инструкций в главном цикле while эта пара переходит в (0, 1), потому что $x[0] < y[0]$. Затем алгоритм сравнивает фактор $x[0..2] =$ abb слова x с фактором $y[1..3] =$ aba слова y, в результате чего порождается следующая пара (3, 1), потому что $x[2] > y[3]$. И на следующей итерации цикла алгоритм обнаруживает, что u и v сопряженные.

На примере этого выполнения алгоритма мы отмечаем, что индексы i и j резко увеличиваются в начальных позициях вхождений в x и y слова Линдона ababb, сопряженного циклически эквивалентным словам u и v. (Если u или v не примитивное, то это рассуждение применимо к корню из него.) Резкое увеличение i в строке 9 или j в строке 10 интуитивно вытекает из свойства слов Линдона: если wa – префикс слова Линдона и буква a меньше буквы b, то wb – слово Линдона. Отсюда следует, что wb свободно от границ. Поэтому сопоставление с wa можно продолжить сразу после вхождения wb. Это видно, в частности, на примере сравнения aba и abb в нашем примере.

Анализ алгоритма CyclicEquivalence. Обозначим $u^{(k)}$ k-е сопряженное (результат k-й ротации) u, $k = 0,1, \ldots, n-1$. Для $x = uu$ $u^{(k)} = x[k\,..\,k+n-1]$. Аналогично для $y = vv$ $v^{(k)} = y[k..k+n-1]$.

Пусть $D(u)$ и $D(v)$ – множества позиций в x и y соответственно:

$D(u) = \{k : 0 \le k < n$ и $u^{(k)} > v^{(j)}$ для некоторого $j\}$,
$D(v) = \{k : 0 \le k < n$ и $v^{(k)} > u^{(j)}$ для некоторого $i\}$.

Правильность алгоритма вытекает из следующего инварианта главного цикла while: $[0..i-1] \subseteq D(u)$ и $[0..j-1] \subseteq D(v)$, который легко проверить. Действительно, если, например, $x[i+k] > y[j+k]$ в строке 8, то имеем $x[i..i+k] > y[j..j+k]$, $x[i+1..i+k] > y[j+1..j+k]$ и т. д.

Если алгоритм возвращает TRUE, то $u^{(i)} = v^{(j)}$ и оба слова сопряженные. Если же он возвращает FALSE, то имеем либо $i = n$, либо $j = n$. Предполагая без ограничения общности, что $i = n$, получаем $D(u) = [1..n]$. Это означает, что для каждого циклического сопряженного u имеется меньшее сопряженное v. Поэтому слова не могут иметь одного и того же наименьшего сопряженного, а значит, они не являются сопряженными.

Количество сравнений символов, очевидно, линейно. Наибольшее число сравнений производится для сопряженных слов вида $u = \mathtt{b}^k\mathtt{ab}^l$ и $v = \mathtt{b}^l\mathtt{ab}^k$. Отсюда вытекает, что время работы линейно. Чтобы избежать дублирования u и v, можно применить арифметику по модулю, тогда объем дополнительной памяти окажется постоянным.

Отсутствие упорядочения. Чтобы решить задачу, не прибегая к отношению порядка на алфавите, нужно воспользоваться техникой сопоставления строк, обладающей этим свойством, или таблицей границ. Например, таблица границ слова $u\#vv$, где # – буква, не встречающаяся в uv, позволяет найти u и vv. Вхождение существует тогда и только тогда, когда слова u и v одинаковой длины являются сопряженными.

Применив оптимальный по времени и памяти алгоритм сопоставления строк, можно построить алгоритм, обладающий всеми свойствами CyclicEquivalence, но он гораздо сложнее и не так элегантен, как описанный выше.

Примечание

Алгоритм CyclicEquivalence основан на идеях алгоритма установления эквивалентности циклических списков, описанного в работе Shiloach [223].

Не столь прямолинейный подход к установлению циклической эквивалентности дает функция MaxSuffixPos (см. задачи 38 и 40). После вычисления индексов $i = \text{MaxSuffixPos}(uu)$ и $j = \text{MaxSuffixPos}(vv)$, определяющих максимальные суффиксы $\text{MaxSuffix}(uu)$ и $\text{MaxSuffix}(vv)$, для решения задачи нужно лишь проверить равенство их префиксов длины $|u| = |v|$.

Можно также использовать таблицу *Lyn* (см. задачу 87), но такой метод менее эффективен.

Оптимальный по времени и памяти алгоритм сопоставления строк можно найти в работах [94, 97, 124].

38. Простое вычисление максимального суффикса

Максимальным суффиксом слова называется лексикографически самый большой суффикс. Это понятие играет важную роль в некоторых комбинаторных свойствах слов (например, связанных с сериями или критиче-

скими позициями), а также в разработке алгоритмов сопоставления строк (например, в двустороннем алгоритме, который используется в некоторых библиотеках на C, в частности glibc и FreeBSD lib). Алгоритм, представленный в этой задаче, нетривиален, но все же проще, чем алгоритм из задачи 40. Оба работают «на месте», т. е. требуют дополнительной (кроме необходимой для хранения входных данных) памяти постоянного объема (в отличие от решения из задачи 23), что упрощает реализацию.

Для непустого слова x алгоритм MaxSuffixPos вычисляет начальную позицию MaxSuffix(x) – *максимального суффикса* x. Например, MaxSuffixPos(bbabbbba) = 3, позиции суффикса bbbba входного слова.

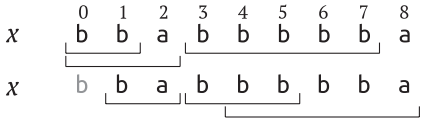
Отметим сходство между MaxSuffixPos и алгоритмом CyclicEquivalence из задачи 37, а также сходство псевдокода с псевдокодом другой версии в задаче 40.

```
MaxSuffixPos(непустое слово x)
 1  (i, j) ← (0,1)
 2  while j < |x| do
 3      ▷ отметим инвариант i < j
 4      k ← 0
 5      while j + k < |x| и x[i + k] = x[j + k] do
 6          k ← k + 1
 7      if j + k = |x| или x[i + k] > x[j + k] then
 8          j ← j + k + 1
 9      else i ← i + k + 1
10      if i ≥ j then
11          j ← i + 1
12  return i
```

Вопрос. Показать, что алгоритм MaxSuffixPos вычисляет позицию максимального суффикса в слове и что он требует линейного времени и дополнительной памяти постоянного объема.

Решение

Применим алгоритм MaxSuffixPos к слову x = bbabbbba, которое продублировано на рисунке ниже, чтобы были лучше видны значения индексов: i в верхней и j в нижней строках.



Первая пара (i, j) индексов в x равна (0,1), поэтому x[0..1] = bb сравнивается с x[1.. 2] = ba. В результате получается следующая пара (0,3). Затем сравнение x[0..2] = bba с x[3..5] = bbb приводит к увеличению i до 3 и дополнительно

увеличению j до 4, чтобы избежать равенства. В конечном итоге j становится равен 9, что вызывает завершение главного цикла и всей процедуры. Алгоритм возвращает позицию 3 максимального суффикса bbbbba слова bbabbbbba.

Правильность алгоритма MaxSuffixPos. Мы только наметим доказательство, поскольку недостающие элементы можно найти в задаче 40. Доказательство опирается на следующий инвариант итерации главного цикла while:

> в интервале $[0..j-1]$ позиция i – единственный кандидат на роль начальной позиции максимального суффикса $x[0..j-1]$. Иными словами, если $t \neq i$ и $t < j$, то t не является начальной позицией максимального суффикса.

Отсюда следует, что в конце j не меньше $|x|$ и i – единственно возможная позиция в интервале $[0..|x|-1]$ (все позиции в x), с которого может начинаться максимальный суффикс. Поэтому i – правильное решение.

Примечания

Представленный алгоритм MaxSuffixPos – вариант алгоритма, описанного в работе Adamczyk and Rytter [1].

После косметической модификации этот алгоритм вычисляет также наименьший период максимального суффикса, как и псевдокод в задаче 40. Действительно, период равен $i - j'$, где j' – предпоследнее значение j (последним является $|x|$, но оно вне диапазона).

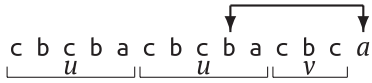
39. Самомаксимальные слова

Понятие ***самомаксимального слова***, т. е. слова, лексикографически наибольшего среди своих суффиксов, в некотором смысле двойственно понятию слова Линдона, которое меньше любого из своих собственных непустых суффиксов. Самомаксимальные слова естественно возникают при нахождении критических позиций в слове (см. задачу 41) и в основанных на них алгоритмах сопоставления строк.

Алгоритм SelfMaximal проверяет, является ли поданное на вход слово x самомаксимальным, т. е. верно ли, что $x = \text{MaxSuffix}(x)$. Он обрабатывает слово в режиме реального времени, потому что инструкция в цикле for выполняется за постоянное время.

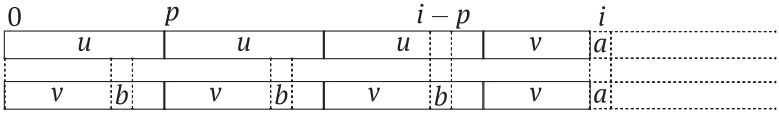
```
SelfMaximal(непустое слово x)
 1  p ← 1
 2  for i ← 1 to |x| − 1 do
 3      if x[i] > x[i − p] then
 4          return FALSE
 5      elseif x[i] < x[i − p] then
 6          p ← i + 1
 7  return TRUE
```

Пример. Слово cbcbacbcbacbc = $(\texttt{cbcba})^2\texttt{cbc}$ является самомаксимальным, равно как и его префиксный период cbcba, который к тому же свободен от границ. Его суффикс cbcbacbc по существу единственный, конкурирующий с самим словом за звание наибольшего суффикса. Стало быть, добавленную в его конец букву следует сравнить с буквой, следующей за вхождением префикса cbcbacbc.



Вопрос. Доказать, что описанный выше очень простой алгоритм правильно определяет, что входное слово больше всех его собственных суффиксов.

На рисунке ниже показана роль переменных в алгоритме SelfMaximal.



Вопрос. Изменить описанный выше алгоритм, так чтобы его новая версия SMPrefix вычисляла самый короткий самомаксимальный префикс слова.

Решение

Перед тем как приступать к доказательствам, сделаем одно замечание. Пусть y – непустое самомаксимальное слово, свободное от границ, т. е. $per(y) = |y|$. Тогда любой собственный непустой суффикс z слова y удовлетворяет условию $z \ll y$ (т. е. $z = ras$ и $y = rbt$, где буква a меньше буквы b). Поэтому $zs' \ll yt'$ для любых слов s' и t'.

Правильность алгоритма SelfMaximal. Рассмотрим инвариант цикла for: $u = x[0..p-1]$ – непустое самомаксимальное, не имеющее границ слово $x[0..i-1] = u^e v$, где $e > 0$ и v – собственный суффикс u.

Инвариант имеет место в начале цикла, потому что $u = x[0]$, $e = 1$ и v – пустое слово. Если инвариант имеет место в строке 7, то $x = u^e v$ и, значит, слово x самомаксимальное с периодом $p = |u|$.

Остается показать, что инструкции в цикле не нарушают инвариант. Необходимо рассмотреть три случая в зависимости от результата сравнения букв.

Если $x[i] > x[i-p]$, то x не самомаксимальное, потому что его фактор $x[p..i-1]x[i] = x[0..i-p-1]x[i]$ больше его префикса $x[0..i-p-1]x[i-p]$. Если $x[i] = x[i-p]$, то инвариант по-прежнему выполняется, быть может, с $e+1$ и $v = \varepsilon$.

Случай $x[i] < x[i-p]$ выражает основное свойство алгоритма: $x[0..i]$ становится свободным от границ, помимо того что является самомаксимальным.

Чтобы убедиться в этом, положим $a = x[i]$ и $b = x[i - p]$, где $a < b$. Сначала рассмотрим суффиксы вида $v'a$ слова va. Поскольку vb – префикс самомаксимального слова u, то $v'b \le u < x[p..i]$, а поскольку $v'a < v'b$, получаем $v'a < x[p..i]$. Далее, суффиксы $x'a$, начинающиеся в позициях $j < e|u|$, не являющихся кратными p, в качестве префикса имеют собственный суффикс u' слова u. В соответствии с высказанным выше замечанием $u' \ll u$, откуда следует, что $x'a \ll u^e va = x[p..i]$. Наконец, осталось рассмотреть суффиксы $x'a$, начинающиеся в позициях $p, 2p, \ldots, ep$. Все x' являются также префиксами u^e, за которыми следует буква b. Поэтому $x'a < x'b < x[p..i]$. На этом доказательство завершается.

Алгоритм SMPrefix. Структура алгоритма очень похожа на SelfMaximal, и его правильность легко следует из приведенного выше доказательства.

```
SMPrefix(непустое слово x)
 1  p ← 1
 2  for i ← 1 to |x| − 1 do
 3      if x[i] > x[i − p] then
 4          return x[0..p − 1]
 5      elseif x[i] < x[i − p] then
 6          p ← i + 1
 7  return x[0..|x| − 1]
```

Примечания

- Алгоритм SelfMaximal основан на версии алгоритма факторизации Линдона, модифицированной в работе Duval [105], и упрощен, чтобы более выпукло показать его основное свойство.

40. Максимальный суффикс и его период

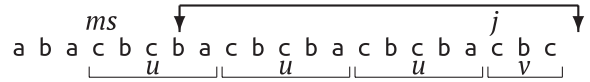
Максимальным суффиксом слова называется его лексикографически наибольший суффикс. Эта задача является продолжением задачи 38, в ней представлен другой псевдокод вычисления максимального суффикса. Как и предыдущий алгоритм, он требует линейного времени и дополнительной памяти постоянного объема.

```
MaxSuffixPP(непустое слово x)
 1  (ms, j, p, k) ← (0,1,1,0)
 2  while j + k < |x| do
 3      if x[j + k] > x[ms + k] then
 4          (ms, j, p, k) ← (j, j + 1, 1, 0)
 5      elseif x[j + k] < x[ms + k] then
 6          (j, p, k) ← (j + k + 1, j − ms, 0)
 7      elseif k = p − 1 then
```

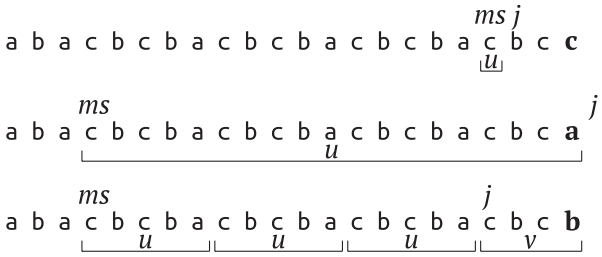
```
8            (j, k) ← (j + k + 1, 0)
9        else k ← k + 1
10   return (ms, p)
```

Вопрос. Доказать, что алгоритм MaxSuffixPP вычисляет начальную позицию и период максимального суффикса входного слова.

Пример. $(\mathtt{cbcba})^3\mathtt{cbc} = \textsc{MaxSuffix}(\mathtt{aba}(\mathtt{cbcba})^3\mathtt{cbc})$. Следующую букву следует сравнить с буквой, следующей за префиксом cbc максимального суффикса.



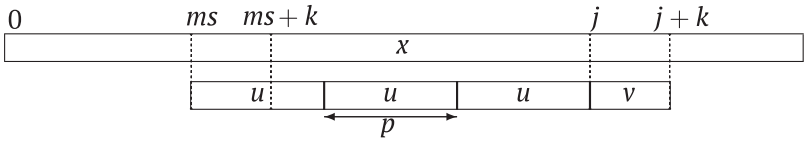
На рисунках показаны три возможности, соответствующие выполнению инструкций в строках 4, 6 и 8.



Вопрос. Показать, как проверить примитивность слова за линейное время и с дополнительной памятью постоянного объема.

Решение

Правильность алгоритма MaxSuffixPP. С точностью до замены переменных структура алгоритма MaxSuffixPP и его доказательство такие же, как для алгоритмов из задачи 39. Здесь ms обозначает начальную позицию $\textsc{MaxSuffix}(x[0..j+k-1]) = u^e v$, где слово u самомаксимальное и не имеет границ, $e > 0$, а v – собственный префикс u. На рисунке показана роль переменных i, j и k. Буква $x[j+k]$ сравнивается с $x[ms+k] = x[j+k-p]$.



Строки 3–4 соответствуют случаю, когда необходимо обновить потенциальную начальную позицию ms максимального суффикса. Позиция j становится следующим кандидатом, потому что суффикс $x[j..j + k]$ больше, чем суффиксы $x[ms..j + k]$, начинающиеся раньше j. И процесс начинается заново с этой позиции, забывая, что было сделано после j.

Строки 5–6 непосредственно связаны с основным свойством алгоритма. Слово $x[ms..j + k]$ самомаксимальное и свободно от границ, его период равен $j + k - ms + 1$, т. е. совпадает с длиной.

Наконец, в строках 7–9 производится увеличение k и, возможно, обновление j, чтобы k оставалось меньше p.

Сложность алгоритма MaxSuffixPP. Требования к объему памяти понятны. Время работы не столь очевидно из-за забывания в строке 4. Но достаточно рассмотреть значение выражения $ms + j + k$.

В строке 4 ms увеличивается по меньшей мере на p, j на 1, а k уменьшается не более чем на $p - 1$, при этом p не изменяется; таким образом, интересующее нас значение строго возрастает. В строке 6 j увеличивается на $k + 1$, перед тем как k обращается в нуль, поэтому значение и на этот раз строго возрастает. В строках 8–9 $j + k$ увеличивается на 1, а ms не изменяется, так что мы приходим к тому же выводу.

Поскольку $ms + j + k$ изменяется от 1 до максимум $2|x| + 1$, это рассуждение показывает, что алгоритм останавливается и выполняет не более $2|x|$ сравнений букв.

Проверка на примитивность. Пусть $ms = \text{MaxSuffixPP}(x)$, а p – период $\text{MaxSuffix}(x)$. Известно, что $ms < p$ (см. задачу 41). Обозначим k наибольшее целое число, для которого $j = ms + kp \leq |x|$. Тогда проверка на примитивность сводится к проверке выполнения равенства $x[j\,..n - 1]x[0..ms - 1] = x[ms..ms + p - 1]$.

Примечания

Алгоритм MaxSuffix описан в работе Crochemore and Perrin [94], где используется на этапе предобработки в двустороннем алгоритме сопоставления строк, оптимальном по времени и памяти.

Если ослабить требование постоянства дополнительной памяти, введя таблицу границ, то время работы увеличится, но асимптотическое поведение в худшем случае не изменится.

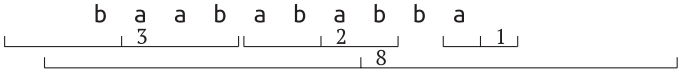
41. Критическая позиция в слове

Существование критических позиций в слове – прекрасный инструмент комбинаторного анализа и проектирования алгоритмов обработки текста. Одно из самых поразительных его применений – двусторонний алгоритм сопоставления строк, реализованный в некоторых стандартных библиотеках C.

Пусть x – непустое слово, и обозначим $lper(i)$ ***локальный период*** в позиции i слова x, $i = 0, \ldots, |x|$, т. е. длину самого короткого непустого слова w, удовлетворяющего условию

$$w \cup A^*x[0..i-1] \neq \emptyset \text{ и } w \cup x[i..|x|-1]A^* \neq \emptyset.$$

Проще говоря, это означает, что самый короткий непустой квадратный участок ww с центром в позиции i имеет период $lper(i)$. Для слова `baababababba` с периодом 8 имеем $lper(1) = |\texttt{aab}| = 3$, $lper(6) = |\texttt{ab}| = 2$, $lper(10) = |\texttt{a}| = 1$ и $lper(7) = |\texttt{bbaababa}| = 8$. Некоторые квадратные участки выходят за границы слова влево, вправо или в обе стороны.



Заметим, что $lper(i) \leq per(x)$ для любого i. Если $lper(i) = per(x)$, то i называется ***критической позицией***. Говорят, что факторизация $x = u \cdot v$ критическая, если $lper(|u|) = per(x)$.

Пусть MaxSuffix($\leq$, x) и ms = MaxSuffixPos($\leq$, x) – соответственно наибольший суффикс x и его позиция относительно отношения порядка $\leq$ на алфавите.

Вопрос. Пусть $x = yz$, где z = MaxSuffix($\leq$, x). Показать, что $|y| < per(x)$, т. е. что MaxSuffixPos($\leq$, x) $< per(x)$.

Алгоритм CriticalPos вычисляет критическую позицию за линейное время, требуя дополнительной памяти постоянного объема. Он основан на задачах 38 и 40.

```
CriticalPos(непустое слово x)
1  i ← MaxSuffixPos(≤, x)
2  j ← MaxSuffixPos(≤^-1, x)
3  return max{i, j}
```

Для слова x = `baababababba` MaxSuffixPos($\leq$, x) = 7 – критическая позиция, а MaxSuffixPos($\leq^{-1}$, x) = 1 нет.

Вопрос. Показать, что алгоритм CriticalPos вычисляет критическую позицию в непустом входном слове.

[**Указание:** обратите внимание, что пересечение двух упорядочений слова является упорядочением префикса, и воспользуйтесь двойственностью между границами и периодами.]

Решение

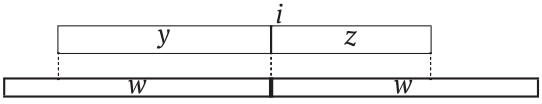
Ответ на первый вопрос. Предположим противное, т. е. что $y \geq per(x)$, и пусть w – суффикс y длиной $per(x)$. В силу периодичности, либо w является префиксом z, либо z является суффиксом w.

Случай 1. Если $z = ww'$, то, по определению, $www' < ww'$, и тогда $ww' < w'$, что противоречит определению w.

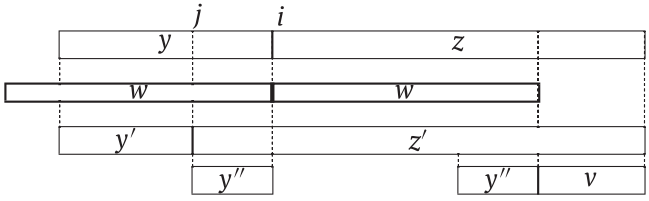
Случай 2. Если $w = zz'$, то $zz'z$ является суффиксом x, и тогда $z < zz'z$, что противоречит определению z.

Ответ на второй вопрос. Случай, когда алфавит x сводится к одной букве, не вызывает трудностей, поскольку тогда любая позиция критическая. Пусть x содержит по крайней мере две разные буквы, и без ограничения общности предположим, что $i > j$. Покажем, что в этом случае i – критическая позиция.

Пусть w – самый короткий непустой квадратный участок с центром в i, т. е. $|w| = lper(i)$. Из первого вопроса мы знаем, что y – собственный суффикс w.



Случай 1. Предположим, что z – префикс w. Тогда $|w| = per(x)$, потому что x, будучи фактором ww, имеет период $|w| == lper(i)$, который не может быть больше $per(x)$. Таким образом, i – критическая позиция.



Случай 2. Предположим, что $z = wv$, где v – непустое слово. Тогда $v < z$, по определению z. Пусть $x = y'z'$, где $z' = \text{MaxSuffix}(\leq^{-1}, x)$ и y'' – непустое слово, для которого $y = y'y''$ и $z' = y''z$. Так как $y''v$ – суффикс x, то он меньше $z' = y''z$ относительно упорядочения, индуцированного отношением $\leq^{-1}$, и, таким образом, v меньше z относительно того же упорядочения. Теперь вспомним указание: v, будучи меньше z относительно обоих упорядочений, является префиксом z, а значит, и его границей. Поэтому $|w|$ – период z, а следовательно, также $x = yz$. Стало быть, как и в предыдущем случае, i является критической позицией.

На этом доказательство завершается.

Примечания

Теорема о критической факторизации, утверждающая существование критической позиции в любом непустом слове, сформулирована в работах Cesari, Duval and Vincent [49, 105] (см. Lothaire [175, глава 8]).

Приведенное здесь доказательство опубликовано в книгах [96, 98] и впервые дано в работе Crochemore and Perrin [94], где является частью двустороннего алгоритма, оптимального по времени и пространству. Оно обобщено на алгоритм, работающий в режиме реального времени, в работе Breslauer et al. [45].

42. Периоды префиксов слов Линдона

Словом Линдона называется непустое самоминимальное слово, т. е. лексикографически меньшее любого из своих непустых собственных префиксов. Двойственные самомаксимальные слова обладают некоторыми общими чертами со словами Линдона. Слова Линдона имеют полезные свойства для проектирования алгоритмов сопоставления и для анализа таких методов, как проверка циклической эквивалентности двух слов (задача 37). В этом разделе рассматривается замечательно простое решение задачи о вычислении периодов и префиксов.

Обозначим *period* таблицу периодов префиксов слова x. Для длин непустых префиксов $l = 1, \ldots, |x|$ она определяется следующим образом:

$period[l]$ = наименьший период $x[0..l-1]$.

Для слова $x =$ `aababababba` имеем:

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	a	b	a	b	a	b	b	a
l	1	2	3	4	5	6	7	8	9
$period[l]$	1	1	3	3	5	5	7	8	8

Вопрос. Показать, что алгоритм PrefixPeriods правильно вычисляет таблицу периодов префиксов слова Линдона.

```
PrefixPeriods(слово Линдона x)
 1  period[1] ← 1
 2  p ← 1
 3  for l ← 2 to |x| do
 4      if x[l − 1] ≠ x[l − 1 − p] then
 5          p ← l
 6      period[l] ← p
 7  return period
```

Вопрос. Какие изменения следует внести в алгоритм PrefixPeriods, чтобы он вычислял периоды префиксов самомаксимального слова?

Вопрос. Показать, что проверку того, является ли заданное слово словом Линдона, можно выполнить за линейное время, воспользовавшись дополнительной памятью постоянного объема.

[**Указание:** модифицируйте алгоритм PrefixPeriods.]

Решение

Решения обоих вопросов очень похожи на решения в задаче 39, хотя понятия самомаксимальности и самоминимальности не являются строго симметричными.

Немного изменив доказательства в задаче 39, несложно показать, что непустые суффиксы слова Линдона имеют вид $u^e v$, где u – слово Линдона, а v – собственный префикс u.

Правильность алгоритма PrefixPeriods. В алгоритме PrefixPeriods в переменной p хранится период префикса $x[0..l-2]$. Напомним, что префикс имеет вид $u^e v$, где $p = |u|$. Переменная p изменяется в цикле for. Цель сравнения в строке 4 – проверить, продолжается ли периодичность p, в этом случае p является также периодом $x[0..l-1]$, что и утверждает присваивание в строке 6. Если результат сравнения отрицательный, то не может быть $x[l-1] < x[l-1-p]$, потому что тогда суффикс $vx[l-1-p]$ был бы меньше x, что противоречит тому факту, что x – слово Линдона. Таким образом, в этом случае $x[l-1] > x[l-1-p]$; это и есть ключевое свойство, упомянутое в решениях задачи 39, и при этом слово $x[0..l-1]$ не имеет границ, а значит, имеет период l, что и утверждает присваивание в строке 5.

Периоды самомаксимальных префиксов. Если на вход алгоритма PrefixPeriods подано самомаксимальное слово, то алгоритм без всяких изменений вычисляет для него таблицу периодов префиксов. Приведенное выше рассуждение остается справедливым после замены $<$ на $>$ и наоборот, главным образом потому, что ключевое свойство по-прежнему имеет место.

Проверка на линдоновость. Алгоритм Lyndon является модификацией приведенного выше алгоритма и напоминает алгоритм SelfMaximal (задача 39) после замены $<$ на $>$ и наоборот. Дополнительная проверка нужна, чтобы убедиться, что слово в целом не имеет границ.

```
Lyndon(непустое слово x)
 1  p ← 1
 2  for l ← 2 to |x| do
 3      if x[l − 1] < x[l − 1 − p] then
 4          return FALSE
 5      elseif x[l − 1] > x[l − 1 − p] then
 6          p ← l
 7  if p = |x| then
 8      return TRUE
 9  else return FALSE
```

Префиксы длиной 1, 3, 5, 7 и 8 слова aababababba являются словами Линдона. Само слово таковым не является, потому что имеет границу a.

В строке 7 если $|x|$ кратно p, то x является ожерельем; в противном случае это префикс ожерелья.

43. Поиск слов Зимина

В этой задаче рассматриваются образцы, являющиеся словами с переменными. Помимо алфавита $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \ldots\}$, содержащего постоянные буквы, есть еще алфавит $\mathtt{V} = \{\alpha_1, \alpha_2, \ldots\}$, из которого выбираются переменные (оба алфавита не пересекаются).

Говорят, что образец $P \in \mathtt{V}^*$ сопоставляется со словом $w \in \mathtt{A}^*$, если $w = \psi(P)$, где $\psi : alph(P)^+ \to \mathtt{A}^+$ – морфизм. ***Слова Зимина*** Z_n, $n \geq 0$, играют важнейшую роль в вопросах устранимости образцов (см. задачу 93). Они определяются следующим образом:

$$Z_0 = \varepsilon,\ Z_n = Z_{n-1} \cdot \alpha_n \cdot Z_{n-1}.$$

Например, $Z_1 = \alpha_1$, $Z_2 = \alpha_1\alpha_2\alpha_1$ и $Z_3 = \alpha_1\alpha_2\alpha_1\alpha_3\alpha_1\alpha_2\alpha_1$.

Типом Зимина слова w называется наибольшее натуральное число k, для которого $w = \psi(Z_k)$, где ψ – некоторый морфизм. Тип всегда определен, потому что пустое слово имеет тип 0, а тип непустого слова равен по меньшей мере 1. Например, тип Зимина слова $w = \mathtt{adbadccccadbad}$ равен 3, потому что оно является образом Z_3 относительно морфизма ψ, определенного следующим образом:

$$\begin{cases} \psi(\alpha_1) = \mathtt{ad}, \\ \psi(\alpha_2) = \mathtt{b}, \\ \psi(\alpha_3) = \mathtt{cccc}. \end{cases}$$

Вопрос. Показать, как за линейное время вычислить типы Зимина всех префиксов данного слова.

[**Указание:** рассмотрите короткие границы префиксов.]

Вопрос. Показать, как за квадратичное время проверить, встречается ли заданный образец Зимина в слове.

[**Указание:** рассмотрите типы Зимина слов.]

Решение

Вычисление типов Зимина. Вычисление типов Зимина префиксов слова $w \in \mathtt{A}^+$ производится в онлайновом режиме следующим образом. Обозначим $Ztype[i]$ тип префикса w длины i. Имеем $Ztype[0] = 0$. Достаточно доказать, что типы Зимина других значений вычисляются итеративно по формуле

$$Ztype[i] = Ztype[j] + 1,$$

где $j = |ShortBorder(w[0..i-1])|$.

Полагая $z = w[0..i-1]$ и $u = ShortBorder(z)$, имеем $z = uvu$, где u и v – два слова и $v \neq \varepsilon$. По определению $Ztype[j]$, слово u является образом $Z_{Ztype[j]}$ относительно морфизма $\psi : \{\alpha_1, \alpha_2, \ldots, \alpha_{Ztype[j]}\}_+ \to A^+$. Продолжим морфизм, положив $\psi(\alpha_{Ztype[j]+1}) = v$, тогда тип Зимина z не меньше $Ztype[j] + 1$. Остается показать, что никакая граница z длины, меньшей u, не может дать большего значения.

Доказательство проведем от противного. Пусть $u'v'$ – факторизация z, для которой $Ztype[|u'|] = Ztype[|z|] - 1$, и предположим, что u' короче, чем u, но $Ztype[|u'|] > Ztype[|u|]$. Поскольку тогда $Ztype[|u|] > 0$, то $Ztype[|u'|] > 1$, откуда следует, что $u' = u''v''u''$, где $v'' \neq \varepsilon$ и $Ztype[|u''|] = Ztype[|u'|] - 1$. Тогда $Ztype[|u''|] = Ztype[|z|] - 2$. Но поскольку u'' также является границей z, то $Ztype[|z|] = Ztype[|u''|] + 1$, т. е. мы пришли к противоречию.

Теперь решение дает алгоритм, вычисляющий короткие границы префиксов (задача 21).

Сопоставление с образцом Зимина. Следующий факт сводит задачу к вычислению типов Зимина.

Факт. Префикс $w[0..i-1]$ слов w сопоставляется с образцом Зимина Z_k тогда и только тогда, когда $Ztype[i] \geq k$.

Действительно, если слово является морфическим образом Z_j для некоторого $j \geq k$, то оно также является морфическим образом Z_k.

Теперь решение очевидно. Мы вычисляем таблицу $Ztype$ для каждого суффикса z слова w, что позволяет найти префиксы z, сопоставляемые с Z_k. И если Z_k входит в w, то оно будет при этом найдено.

```
MatchingZiminPattern(непустое слово w, целое положительное число k)
1  for s ← 0 to |w| − 1 do
2      Ztype[0] ← 0
3      for i ← s to |w| − 1 do
4          вычислить Ztype[i − s + 1] для w[s..|w| − 1]
5          if Ztype[i − s + 1] ≥ k then
6              return TRUE
7  return FALSE
```

При вычислении в строке 4 применяется алгоритм с линейным временем работы из предыдущего вопроса. Поэтому вся проверка занимает время $O(|w|^2)$, что и требовалось доказать.

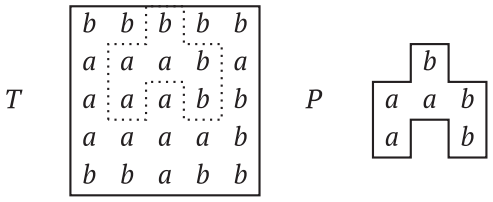
Отметим, что после простой модификации алгоритм может вычислять наибольшее целое k, для которого Z_k встречается во входном слове.

Примечания

Более интересен вопрос об обратном сопоставлении с переменными, когда требуется проверить, входит ли заданное слово с переменными в заданный образец Зимина. Известно, что эта задача имеет класс сложности NP, но неизвестно, является ли она NP-трудной.

44. Поиск нерегулярных двумерных образцов

Пусть P – потенциально нерегулярный двумерный образец. Говоря *нерегулярный*, мы подразумеваем, что P может быть любой формы, необязательно прямоугольной. Наша цель – найти все вхождения P в двумерный текст T в виде таблицы $n \times n'$ размером $N = nn'$.



На рисунке показан нерегулярный образец P. Он целиком помещается в квадрат 3×3 и встречается в T два раза (на рисунке показано одно вхождение).

Вопрос. Показать, что поиск вхождений нерегулярного двумерного образца можно выполнить за время $O(N \log N)$.

Решение

Решение заключается в том, чтобы линеаризировать задачу. Пусть P – непрямоугольный образец, который целиком помещается в прямоугольник $m \times m'$. Без ограничения общности можно предположить, что первый и последний столбцы, а также первая и последняя строки этого прямоугольника содержат элементы P. В противном случае строки или столбцы можно было бы удалить.

Результатом линеаризации текста T является текст T', получающийся конкатенацией строк. Преобразование P устроено более хитро. Сначала P погружается в прямоугольник размером $m \times m'$, элементы которого, не принадлежащие P (пустые клетки), заменяются на $*$. Строки этого прямоугольника конкатенируются, причем между строками вставляется слово, состоящее из $n' - m$ символов $*$. Получившаяся строка P' является результатом линеаризации P.

Пример. Для образца P, изображенного на рисунке, строками прямоугольника 3×3 являются $* \; b \; *$, $a \; a \; b$ и $a * b$, а $P' = * \; b * * * a \; a \; b * * a * b$.

Главным свойством этого преобразования является тот факт, что вхождениям P в T соответствуют вхождения слова P' в слово T', причем P' содержит универсальные символы.

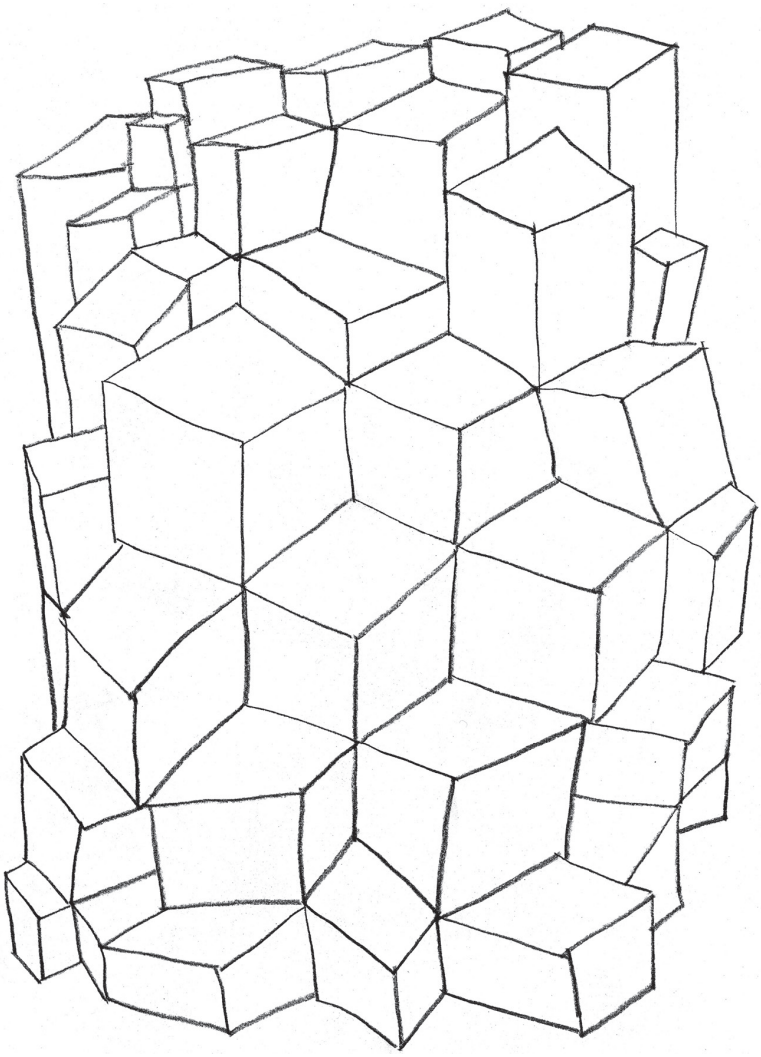
Следовательно, все вхождения P можно найти, применив метод из задачи 36. Поэтому полное время работы алгоритма имеет порядок $O(N \log N)$.

Примечания

Время работы можно уменьшить до $O(N \log(\max(m, m')))$. Описанный здесь метод линеаризации используется в работе [10].

Глава 4

Эффективные структуры данных



45. Списковый алгоритм для кратчайшего покрытия

Покрытие непустого слова x образует один из его факторов, вхождения которого покрывают все позиции x. В каком-то смысле оно сродни повторению. В этой задаче показано, как вычислить кратчайшее покрытие слова, воспользовавшись его таблицей префиксов *pref*, а не таблицей границ, как в задаче 20. Алгоритм проще, но нуждается в дополнительной памяти линейного объема.

Для каждой длины l префикса x положим $L(l) = (i : pref[i] = l)$. Алгоритм ShortestCover вычисляет длину кратчайшего покрытия входного слова.

```
ShortestCover(непустое слово x)
 1  L ← (0,1, ... , |x| − 1)
 2  for l ← 0 to |x| − 1 do
 3      удалить элементы L(l − 1) из L
 4      if maxgap(L) ≤ l then
 5          return l
```

В последних строках таблицы ниже показаны позиции в списке L для $l = 1$, 2, 3 при применении этого алгоритма к слову $x =$ `abababaaba`. Соответствующие значения $maxgap(L)$ равны 2, 3 и 3. Условие в строке 4 впервые удовлетворяется, когда $l = 3$, поэтому кратчайшее покрытие равно `aba` $= x[0..2]$.

i	0	1	2	3	4	5	6	7	8	9
$x[i]$	a	b	a	b	a	b	a	a	b	a
$pref[i]$	10	0	5	0	3	0	1	3	0	1
$L - L[0]$	0	2	4	6	7	9	10			
$L - L[\leq 1]$	0	2	4	7	10					
$L - L[\leq 2]$	0	2	4	7	10					

Вопрос. Показать, что алгоритм ShortestCover вычисляет длину кратчайшего покрытия входного слова и при правильной реализации работает за линейное время.

Решение

Правильность алгоритма ShortestCover очевидна: он удаляет позиции с малыми значениями *pref*, поскольку их префиксы слишком короткие, так что их можно игнорировать. В конечном итоге условие удовлетворяется, когда $l = |x|$.

Если L и $L[l]$ реализованы, например, как двунаправленные отсортированные списки, то для удаления одного элемента и обновления *maxgap* требуется постоянное время. Поэтому полное время работы линейно, т. к. каждый эле-

мент удаляется из L не более одного раза, а общий размер непересекающихся списков $L[l]$ равен $|x| + 1$.

46. Вычисление наибольших общих префиксов

Суффиксный массив непустого слова y – простое и эффективное решение задачи индексирования текста. Идея в том, чтобы можно было применить процедуру двоичного поиска к нахождению образцов в y. Для этого суффиксы y сначала сортируются в лексикографическом порядке, и строится таблица SA начальных позиций отсортированных суффиксов.

Но этой стандартной процедуры недостаточно для реализации мощного метода поиска. Поэтому в дополнение к таблице SA используется вторая таблица LCP, содержащая длины ***наибольших общих префиксов*** соседних суффиксов в отсортированном списке (нужны также еще кое-какие значения, которые легко вывести). Имея обе таблицы, можно найти слово x в тексте y за время $O(|x| + \log|y|)$, а не за время $O(|x| \log|y|)$, как при поиске без таблицы LCP. Ниже показан суффиксный массив слова `abaababababbabbb`:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$y[j]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b	
Ранг r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[r]	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP[r]	0	1	3	4	2	3	0	1	2	3	4	1	2	2	0

где $\text{LCP}[r] = |\text{lcp}(y[\text{SA}[r-1]..|y|-1], y[\text{SA}[r]..|y|-1])|$.

Вопрос. Пусть дана таблица SA для слова y. Показать, что алгоритм LCP вычисляет ассоциированную с ней таблицу LCP за линейное время.

```
LCP(непустое слово y)
 1  for r ← 0 to |y| − 1 do
 2      Rank[SA[r]] ← r
 3  l ← 0
 4  for j ← 0 to |y| − 1 do
 5      l ← max{0, l − 1}
 6      if Rank[j] > 0 then
 7          while max{j + l, SA[Rank[j] − 1] + l} < |y| и
                  y[j + l] = y[SA[Rank[j] − 1] + l] do
 8              l ← l + 1
 9      else l ← 0
10      LCP[Rank[j]] ← l
11  LCP[|y|] ← 0
12  return LCP
```

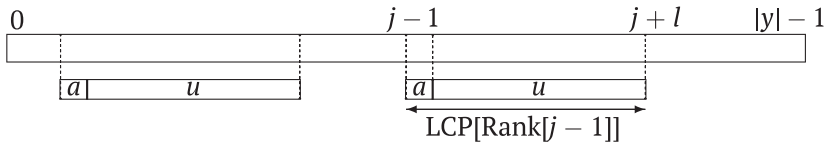
Заметим, что это решение противоречит интуиции, поскольку кажется естественным вычислять значения LCP[r] последовательно, т. е. обрабатывая суффиксы в порядке возрастания их рангов. Но тогда не получится алгоритм с линейным временем работы. Вместо этого алгоритм LCP обрабатывает суффиксы в порядке от самого длинного к самому короткому, что более эффективно и является главной «фишкой».

Решение

Правильность алгоритма следует из неравенства

$$\text{LCP}[\text{Rank}[j-1]] - 1 \leq \text{LCP}[\text{Rank}[j]],$$

иллюстрируемого на следующем рисунке.



Предположим, что мы только что вычислили $l = \text{LCP}[\text{Rank}[j-1]]$, и наибольший общий префикс, ассоциированный с позицией $j-1$, равен au, где a – буква, u – слово, т. е. $\text{LCP}[\text{Rank}[j-1]] = |au|$. Тогда наибольший общий префикс, ассоциированный с позицией j, не может быть короче u. Поэтому сравнение для вычисления LCP[Rank[j]] путем расширения u можно начинать с позиции $j + l$. Именно так алгоритм и поступает в строках 7–8. Строка 5 исключает случай, когда наибольший общий префикс пуст.

При вычислении используется таблица Rank – обращение таблицы SA, вычисляемое в строках 1–2. Она нужна, чтобы найти суффикс, непосредственно предшествующий суффиксу $y[j..|y|-1]$ в отсортированном списке всех суффиксов.

Время работы этой процедуры зависит главным образом от количества проверок в строке 7. Если буквы совпадают, то значение $j + l$ увеличивается и больше никогда не уменьшается. Поэтому таких случаев не может быть больше $|y|$. Для каждого значения переменной j может существовать не более одного несовпадения, значит, и таких случаев не больше $|y|$. Тем самым доказано, что алгоритм работает за линейное время и выполняет не более $2|y|$ сравнений букв.

Примечания

Описанное здесь решение взято из работы Kasai et al. [155]. См. также книгу [74], в которой показано, как вычислить таблицу SA за линейное время, если алфавит допускает линейную сортировку.

47. От суффиксного массива к суффиксному дереву

В этой задаче наша цель – преобразовать суффиксный массив слова x в его суффиксное дерево. Хотя обе структуры данных допускают, по существу, одни и те же операции индексирования, некоторые выполняются проще при наличии суффиксного дерева.

Особенный интерес представляет проектирование такого алгоритма с линейным временем работы в случае, когда алфавит допускает линейную сортировку. Действительно, в предположении, что эта гипотеза справедлива, существует много алгоритмов построения суффиксного массива слова за линейное время, но по существу всего один метод построить его суффиксное дерево за такое же время. Кроме того, разработать методы построения суффиксного массива проще.

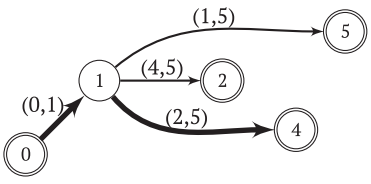
Ниже показаны таблицы SA и LCP для суффиксного массива слова `aacab`:

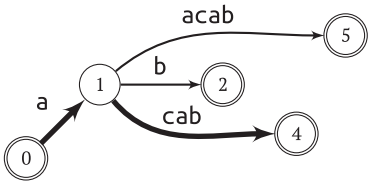
r	SA	LCP	
0	0	0	`aacab`
1	3	1	`ab`
2	1	1	`acab`
3	4	0	`b`
4	2	1	`cab`

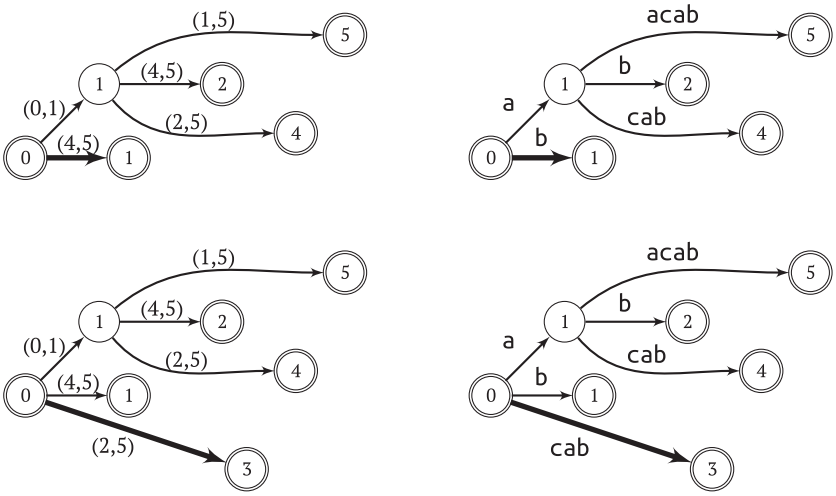
В таблице SA хранятся начальные позиции непустых суффиксов в соответствии с их рангом r относительно лексикографического порядка. Сами суффиксы не являются частью структуры. В элементе LCP[r] хранится наибольший общий префикс суффиксов ранга r и ранга $r - 1$.

Вопрос. Показать, как построить суффиксное дерево слова за линейное время, если известен его суффиксный массив.

На рисунках ниже показаны первые три шага построения возможного суффиксного дерева для слова `aacab`. На первом рисунке обрабатываются суффиксы `aacab`, `ab` и `acab`. Меткой узла является его глубина в слове, а метка дуги имеет вид (i, j) (слева) и представляет факторы $x[i..j - 1]$ (справа) слова x. Двойной окружностью обведены узлы, соответствующие заключительным состояниям, а жирным путем показаны последние вставленные суффиксы.





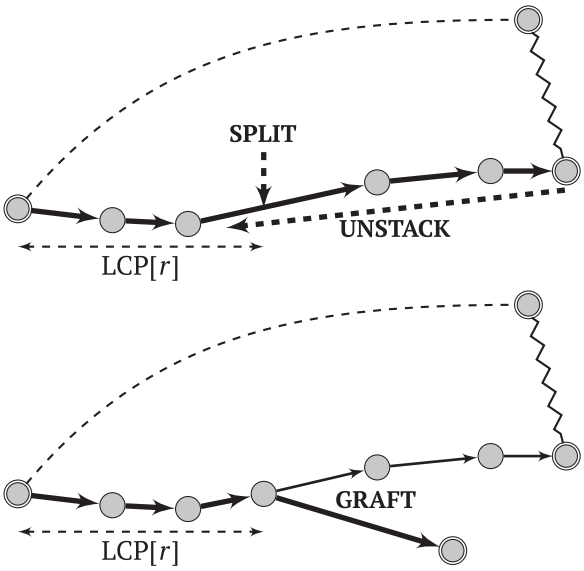


Решение

```
SArray2STree(суффиксный массив непустого слова x)
 1   ▷ (SA, LCP) суффиксный массив x
 2   (q, d[q]) ← (New-Terminal-State(), 0)
 3   Initial ← q
 4   S ← ∅
 5   Push(S, (q, 0, 0, q))
 6   for r ← 0 to |x| − 1 do
 7       do (p, i, j, q) ← Pop(S)
 8       while LCP[r] < d[p]
 9       if LCP[r] = d[q] then
10           Push(S,(p, i, j, q))
11           s ← q
12       elseif LCP[r] = d[p] then
13           s ← q
14       else (s, d[s]) ← (New-State(), LCP[r])
15           Split(p, i, i + LCP[r] − d[p], s, i + LCP[r] − d[p], j, q)
16           Push(S, (p, i, i + LCP[r] − d[p], s))
17       (t, d[t]) ← (New-Terminal-State(), |x| − SA[r])
18       (s, SA[r] + LCP[r], |x|, t) ← New-Arc()
19       Push(S, (s, SA[r] + LCP[r], |x|, t))
20   return (Initial, узлы и дуги)
```

Алгоритм SArray2STree обрабатывает суффиксы поданного на вход слова в лексикографическом порядке, т. е. согласно таблице SA, и вставляет их в дерево. Напомним, что дуги помечены описанным выше образом, чтобы вся структура занимала память линейного объема. Таблица LCP используется в сочетании с глубиной узлов $d[\,]$ (проставленной внутри узлов на рисунках

выше). На каждом шаге в стеке S хранятся дуги на пути, ассоциированном с последним вставленным суффиксом (жирные пути на рисунках). Операция Split (строка 15) вставляет узел s в середину дуги и соответственно проставляет метки на двух образовавшихся в результате дугах.



Инструкции в цикле for, иллюстрируемые на рисунках выше, состоят из трех основных шагов: **UNSTACK** (выбрать из стека), необязательный **SPLIT** (расщепить) и **GRAFT** (привить). Шаг **UNSTACK** реализован циклом while в строках 7–8. Затем найденная дуга расщепляется в строках 14–15, если это необходимо, т. е. если операция расщепления должна быть произведена в середине дуги, а не на одном из ее концов. Наконец, новая дуга прививается в строках 17–18. Новые дуги вдоль пути, помеченные текущим суффиксом, помещаются в стек.

Правильность алгоритма можно вывести из этих замечаний. Что касается времени работы и прежде всего анализа цикла while, то оно зависит от времени обхода дерева, реализованного с помощью стека. Поскольку размер дерева линейно зависит от длины слова, алгоритм работает за линейное время. Заметим, что никаких условий на алфавит слова не накладывается.

Примечания

Первый алгоритм построения за линейное время суффиксного дерева над алфавитом, допускающим линейную сортировку, был разработан в книге Farach [110]. Описанный здесь алгоритм предлагает другое решение – построение на основе суффиксного массива, обладающее такими же характеристиками. Исторически первое такое построение было выполнено в работах Kärkkäinen and Sanders [153, 154] (см. [74]), затем в работах Ko and Aluru [163] и Kim et al. [159], за которыми последовало еще несколько.

48. Линейное суффиксное trie-дерево

Размер суффиксного trie-дерева[1] слова может квадратично зависеть от длины слова. С другой стороны, для хранения его суффиксного дерева нужна память линейного объема, но нужно хранить и само слово.

Наша цель – спроектировать суффиксное trie-дерево, ребра которого помечены одиночными буквами и которое можно сохранить в памяти линейного объема, не храня само слово. Для этого мы добавим дополнительные узлы и несколько элементов в суффиксное дерево.

Узел суффиксного trie-дерева слова y идентифицируется фактором y, который помечает путь от корня к узлу. Узлы линейного суффиксного trie-дерева $\mathcal{LST}(y)$, не являющиеся частью суффиксного дерева $\mathcal{ST}(y)$, имеют вид au, где a – буква, а u – узел $\mathcal{ST}(y)$. То есть, обозначив s суффиксную ссылку дерева, будем иметь $s(au) = u$. Когда в $\mathcal{ST}(y)$ добавляются узлы для создания $\mathcal{LST}(y)$, метки ребер соответственно изменяются.

Вопрос. Показать, что количество дополнительных узлов, добавленных в суффиксное дерево слова y для создания его линейного суффиксного trie-дерева, меньше $|y|$.

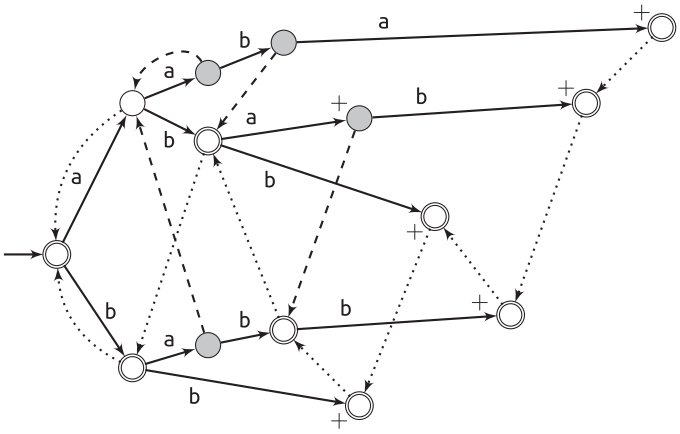
Метки ребер $\mathcal{LST}(y)$ следующим образом редуцируются до первой буквы соответствующего фактора. Если слово v, $|v| > 1$, помечает ребро, идущее из u в uv в дереве $\mathcal{ST}(y)$, то меткой ассоциированного ребра в $\mathcal{LST}(y)$ будет первая буква v, а узел uv помечается знаком $+$, чтобы показать, что фактическая метка длиннее.

Вопрос. Спроектировать алгоритм, который проверяет, входит ли x в y, пользуясь линейным суффиксным trie-деревом $\mathcal{LST}(y)$, и работает ли за время $O(|x|)$ при фиксированном алфавите.

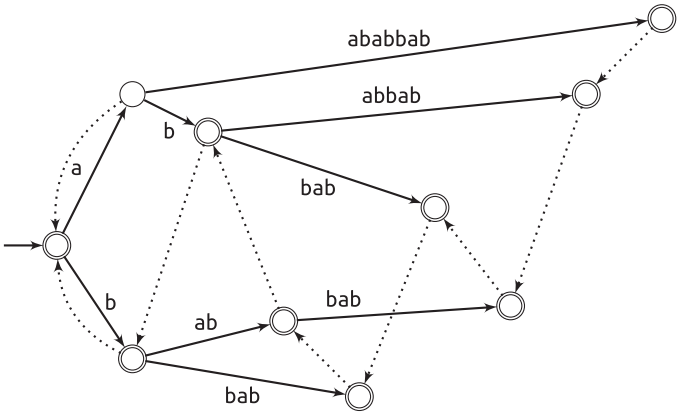
[**Указание:** метки ребер можно восстановить, воспользовавшись суффиксными ссылками.]

---

[1] Слово «trie» в русскоязычной литературе вообще и в этой книге в частности обычно переводится как «префиксное дерево» (реже употребляется термин «бор»). Но поскольку словосочетание «суффиксное префиксное дерево» звучит нелепо, в этом разделе употребляется калька «trie-дерево». Кстати говоря, «trie» – не слово английского языка, а просто средний слог слова «retrieval» (поиск). Этот термин ввел в 1960 году Эдвард Фредкин, хотя сама идея префиксных деревьев значительно старше. – *Прим. перев.*



На рисунке выше показано линейное суффиксное trie-дерево слова `aababbab`. Белые узлы принадлежат его суффиксному дереву (показано ниже с явными метками ребер), а двойными окружностями обведены те из них, которые являются суффиксами. Пунктирные ребра образуют суффиксные ссылки суффиксного дерева. Серым цветом показаны дополнительные узлы, а штриховые ребра – это исходящие из них суффиксные ссылки.



Решение

Дополнительных узлов немного. Для ответа на первый вопрос обозначим u узел $\mathcal{LST}(y)$, не принадлежащий $\mathcal{ST}(y)$. По определению, $s(u)$ является узлом $\mathcal{ST}(y)$. Любой собственный суффикс u имеет вид $s^k(u)$, а это означает, что он также является узлом $\mathcal{ST}(y)$. Поэтому два разных узла типа u не могут разделять одну и ту же правую позицию, и, стало быть, существует не более $|y|$ таких узлов.

Заметим, что для слова с попарно различными буквами имеется ровно $|y| - 1$ дополнительных узлов. Если буква встречается хотя бы дважды (в двух разных правых позициях), то она является узлом $\mathcal{ST}(y)$, так что существует

не более $|y| - 2$ дополнительных узлов. Общее число дополнительных узлов меньше $|y|$.

В нашем примере $\mathcal{LST}$ (для слова aababbab) правые позиции добавленных узлов (показаны серым цветом на рисунке) равны 1 для aa, 2 для aab, 4 для abab, 3 и 6 для ba.

Поиск в $\mathcal{LST}(y)$. Проверка, является ли x фактором y, производится путем обращения к функции Search($root$, x), где $root$ – корень $\mathcal{LST}(y)$. Главный момент, касающийся неполных ребер, т. е. ребер, ведущих в узел, помеченный знаком $+$, опирается на следующее наблюдение.

Наблюдение. Пусть au (здесь a – буква) – узел $\mathcal{LST}(y)$. Если auv является узлом, то таковым является и uv. Это означает, что если v можно прочитать из узла au в дереве, то его можно прочитать и из $s(au) = u$ (обратное неверно).

Это позволяет набросать алгоритм SEARCH, который возвращает TRUE, если слово ux является фактором y.

```
SEARCH(узел u LST(y), слово x)
 1  if x = ε then
 2      return TRUE
 3  elseif ни одно ребро, исходящее из u, не помечено x[0] then
 4      return FALSE
 5  else пусть (u, uv) – ребро, метка v которого равна x[0]
 6      if узел uv не помечен знаком + then
 7          return SEARCH(uv, x[1..|x| − 1])
 8      elseif SEARCH(s(u), v) then
 9          return SEARCH(uv, v^{-1}x)
10      else return FALSE
```

При прямолинейной реализации этой схемы время работы может быть нелинейно из-за неявных меток на некоторых ребрах. Чтобы решить эту проблему, используется еще одна суффиксная ссылка, обозначаемая $\bar{s}$.

Сначала заметим, что для любого ребра (u, uv) $\mathcal{LST}(y)$ пара $(s^k(u), s^k(uv))$ определена для $0 \le k \le |u|$, но узлы $s^k(u)$ и $s^k(uv)$ могут быть не соединены ребром непосредственно. Суффиксная ссылка $\bar{s}$ определена для ребер $\mathcal{LST}(y)$, соответствующих ребрам суффиксного дерева, метки которых содержат больше одной буквы. Если (u, uv) – такое ребро $\mathcal{LST}(y)$, т. е. $|v| > 1$, то $\bar{s}(u, uv) = (s^k(u), s^k(uv))$, где k – наименьшее целое число, для которого узлы $s^k(u)$ и $s^k(uv)$ не соединены ребром. Это определение корректно, потому что все слова длины 1 являются узлами $\mathcal{LST}(y)$ (но необязательно узлами $\mathcal{ST}(y)$). Заметим, что $\bar{s}$ можно вычислить за время, пропорциональное числу ребер $\mathcal{LST}(y)$.

Благодаря использованию $\bar{s}$ реализация работает за линейное время. Действительно, всякий раз, как $\bar{s}(u, uv)$ используется для нахождения явной метки v ребра, буква v восстанавливается. Но оно не может быть использовано более чем $|v|$ раз, поэтому получаем линейное амортизированное время работы. Над алфавитом A общего вида эта реализация работает за время $O(|x| \log |A|)$.

Примечания

Линейное суффиксное trie-дерево слова и ассоциированные с ним методы поиска описаны в [71]. Линейное суффиксное trie-дерево можно построить путем несложной постобработки суффиксного дерева слова.

В работе Hendrian et al. [140] спроектирована процедура онлайнового построения $\mathcal{LST}(y)$ справа налево, работающая за время $O(|y| \log |A|)$. Там же описана процедура онлайнового построения слева направо, работающая за время $O(|y|(\log |A| + \log |y|/\log\log |y|))$.

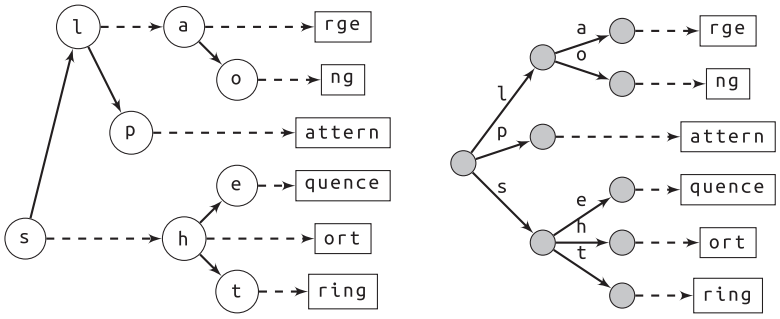
49. Троичное префиксное дерево поиска

Троичные префиксные деревья поиска – эффективная структура для хранения множества слов и поиска в нем. Это изобретательная реализация префиксного дерева множества, аналогичная использованию суффиксного массива для представления суффиксов слова.

Поиск образца в префиксном дереве начинается из начального состояния (корня) и продвигается вниз по ребрам, пока не будет достигнут конец образца или пока не окажется, что ребра, соответствующего текущей букве, не существует. Если алфавит велик, то представление ребер, исходящих из некоторого состояния, может стать причиной непроизводительного расходования памяти, потому что многие потенциальные ребра никуда не ведут. Если же для представления используются линейные списки, то на поиск в них будет тратиться много времени. Цель троичных префиксных деревьев поиска – представить ребра двоичными деревьями поиска над исходящими буквами.

Для этого из каждого узла префиксного дерева исходит три ребра: левое и правое (направленные на рисунке вверх и вниз), принадлежащие двоичному дереву поиска в текущем узле префиксного дерева, и среднее, ведущее в следующий узел префиксного дерева. На рисунке ниже изображены троичное префиксное дерево поиска (слева) и префиксное дерево (справа) следующего множества слов:

{large,long,pattern,sequence,short,string}.



Вопрос. Описать структуру данных для реализации троичного префиксного дерева поиска, позволяющего хранить множество n слов, и показать, как искать в нем слово длины m. Проанализировать время работы.

[**Указание:** обратите внимание на аналогию с поиском в суффиксном массиве.]

Заметим, что в примере выше корень двоичного дерева поиска, соответствующего ребрам, исходящим из начального узла префиксного дерева, помечен буквой `s`, а не средней буквой `p`. Дело в том, что для повышения эффективности поиска двоичные деревья поиска делают сбалансированными по весу. Вес узла соответствует количеству элементов в его поддереве. Именно поэтому в корень двоичного дерева поиска помещена буква `s`, с которой начинается большинство слов в множестве.

Решение

Структура данных троичного дерева поиска T состоит из узлов, связанных в дерево. В каждом узле q хранится три указателя на другие узлы: $q.left$, $q.right$ и $q.mid$, их назначение описано выше. Некоторые узлы листовые (из них не выходит ни одного ребра). В поле $q.val$ каждого узла хранится либо суффикс слова в T, если q – листовой узел, либо буква.

```
TST-SEARCH(троичное дерево поиска T и непустое слово x)
 1  q ← начальный узел T
 2  for i ← 0 to |x| − 1 do
 3      if q – листовой узел then
 4          if x[i..|x| − 1] = q.val then
 5              return TRUE
 6          else return FALSE
 7      q ← BST-SEARCH((q, x[i]))
 8      if q не определен then
 9          return FALSE
10      q ← q.mid
11  return FALSE    ▷ x – префикс слова в T
```

Двоичный поиск в строке 7 выполняется по поддереву с корнем в узле q с использованием только указателей *left* и *right*, при этом поле *val* сравнивается с $x[i]$.

Пусть $n > 0$ – количество слов, хранящихся в T. Грубый анализ худшего случая показывает, что время работы имеет порядок $O(|x| \log n)$. Но роль TST-поиска аналогична двоичному поиску в суффиксном массиве с целью нахождения текущей буквы $x[i]$, что приводит к улучшенной оценке $O(|x| + \log n)$. Точнее, каждое безрезультатное сравнение букв в процессе TST-поиска уменьшает интервал подлежащих поиску слов, поэтому число таких сравнений равно $O(\log n)$. А каждое результативное сравнение приводит к выходу из цикла for, поэтому общее число таких сравнений равно $O(|x|)$. Таким образом,

всего будет произведено порядка $O(|x| + \log n)$ сравнений, включая сравнения в строке 4, что и определяет время работы.

Примечание

Понятие троичного префиксного дерева поиска введено в работе Bentley and Sedgewick [31]. В работе Clément et al. [57] дан скрупулезный анализ этой структуры данных при различных вероятностных условиях.

В применении к суффиксам слова троичное префиксное дерево поиска является структурой данных, которая соответствует алгоритмам, ассоциируемым с суффиксным массивом слова.

50. Наибольший общий фактор двух слов

В этой задаче рассматриваются общие факторы двух слов. Это служит основой для сравнения текстов и обобщается на такие приложения, как совмещение биологических последовательностей или распознавание плагиата.

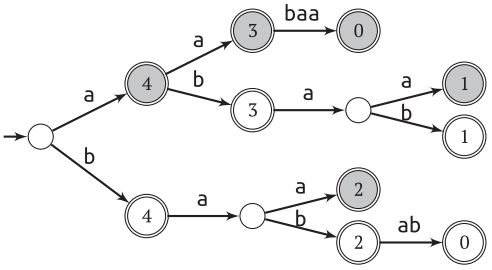
Обозначим $LCF(x, y)$ максимальную длину факторов, встречающихся в двух заданных словах x и y над алфавитом A. Прямолинейный способ ее вычисления состоит в том, чтобы построить общее суффиксное дерево x и y. Его узлами являются префиксы x или y. Самый глубокий узел, поддерево которого содержит одновременно суффиксы x и суффиксы y, дает ответ на вопрос, и этот ответ – глубина узла. Сделать это можно, построив суффиксное дерево слова $x\#y$, где # – буква, не встречающаяся ни в x, ни в y.

Время вычисления такого дерева составляет $O(|xy| \log |A|)$ или $O(|xy|)$ для алфавитов, допускающих линейную сортировку (см. задачу 47), а объем требуемой памяти равен $O(|xy|)$.

Ниже показано общее суффиксное дерево слов $x =$ `aabaa` и $y =$ `babab`. Узлы серого (соответственно белого) цвета, обведенные двойной окружностью, – это непустые суффиксы x (соответственно y). Узел `aba` дает $LCF($`aabaa`, `babab`$) = |$`aba`$| = 3$.

i	0	1	2	3	4
$x[i]$	a	a	b	a	a

j	0	1	2	3	4
$y[j]$	b	a	b	a	b



Наша цель в этой задаче – уменьшить размер структуры данных, ограничившись одним словом, а не двумя, как в приведенном выше решении.

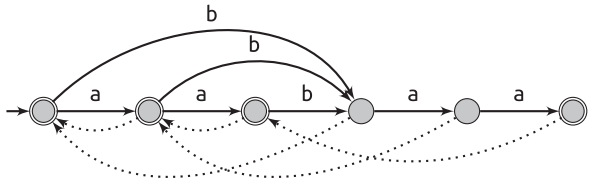
Вопрос. Спроектировать алгоритм вычисления $LCF(x, y)$ с использованием суффиксного автомата (или суффиксного дерева) только одного слова. Проанализировать его временную и пространственную сложность.

[**Указание:** воспользуйтесь индексной структурой, как в поисковой системе.]

Решение

Предположим, что $|x| \leq |y|$, и рассмотрим суффиксный автомат $\mathcal{S}(x)$ слова x. Известно, что его размер равен $O(|x|)$ независимо от алфавита. В дополнение к состояниям и помеченным дугам этот автомат снабжен двумя функциями, определенными на множестве состояний: *fail* (неуспешная ссылка) и L (максимальная глубина). Для состояния q, ассоциированного с непустым словом v (т. е. $q = goto(\text{initial}, v)$), $fail[v]$ – это состояние $p \neq q$, ассоциированное с самым длинным возможным суффиксом u слова v. А $L[q]$ – максимальная глубина слов, ассоциированных с q.

Ниже показан суффиксный автомат слова aabaa и неуспешные ссылки (failure link) (пунктирные дуги) его состояний.



Алгоритм LCF решает задачу, используя $\mathcal{S}(x)$ как поисковую систему.

```
LCF(суффиксный автомат S(x) слова x, непустое слово y)
 1   (m, l, q) ← (0, 0, начальное состояние S(x))
 2   for j ← 0 to |y| − 1 do
 3       if goto(q, y[j]) определено then
 4           (l, q) ← (l + 1, goto(q, y[j]))
 5       else do q ← fail[q]
 6           while q определено и goto(q, y[j]) не определено
 7           if q определено then
 8               (l, q) ← (L[q] + 1, goto(q, y[j]))
 9           else (l, q) ← (0, начальное состояние S(x))
10       m ← max{m, l}
11   return m
```

На каждом шаге алгоритм вычисляет длину l самого длинного совпадения фактора x с суффиксом $y[0..j]$. Для этого он действует как алгоритмы сопоставления строк, основанные на использовании неуспешных ссылок. Единственная специфическая особенность данного алгоритма – механизм

восстановления длины l в $L[q] + 1$ после прохода по последовательности ссылок (см. примечания).

Что касается времени выполнения, оно линейно для алфавитов, допускающих линейную сортировку. Действительно, построить суффиксный автомат x можно за линейное время, а представленный выше алгоритм также требует линейного времени, потому что всякое вычисление $goto(q, y[j])$ ведет к увеличению либо переменной j, либо выражения $j - l$, а обе эти величины изменяются от 0 до $|y|$.

Отметим, что на самом деле алгоритм находит самый длинный фактор x, который заканчивается в любой позиции y.

Примечания

Метод, разработанный в этой задаче, взят из работы Crochemore [68] (см. также [74, глава 6]). Аналогичный метод с использованием суффиксного дерева описан в работе Hartman and Rodeh [138]. С небольшими изменениями эта техника позволяет найти слово, сопряженное x, в тексте y с помощью суффиксного автомата xx.

51. Автомат подпоследовательностей

Подпоследовательности (или подслова), встречающиеся в слове, полезны для фильтрации серии текстов или для их сравнения. Основная структура данных для разработки приложений, связанных с подпоследовательностями, – допускающий их автомат разумного размера.

Для непустого слова y обозначим $\mathcal{SM}(y)$ минимальный детерминированный автомат, допускающий подпоследовательности y. Он также называется детерминированным ациклическим графом подпоследовательностей (Deterministic Acyclic Subsequence Graph – DASG). Ниже показан автомат подпоследовательностей слова `abcabba`. Все его состояния заключительные, и он допускает множество

{a,b,c,aa,ab,ac,ba,bb,bc,ca,cb,aaa,aab,aba,abb,abc, . . .}.

Вопрос. Показать, как построить автомат подпоследовательностей заданного слова, и проанализировать пространственную и временную сложность построения в зависимости от размера автомата.

Автоматы подпоследовательностей – важная структура для нахождения слов, которая различает два слова, поскольку предоставляет прямой доступ ко всем их подпоследовательностям. Слово u различает два слова y и z, если оно является подпоследовательностью только одного из них, т. е. если оно принадлежит симметрической разности множеств их подпоследовательностей.

Вопрос. Показать, как вычислить кратчайшую подпоследовательность, различающую два разных слова y и z с помощью их автоматов $\mathcal{SM}(y)$ и $\mathcal{SM}(z)$.

Для проектирования алгоритма представляет интерес следующее свойство автомата подпоследовательностей y: если существует путь из состояния 0 в состояние i, помеченный словом u, то $y[0..i-1]$ – кратчайший префикс y, содержащий u в качестве подпоследовательности.

Решение

Построение автомата подпоследовательностей. Состояниями автомата $\mathcal{SM}(y)$ являются 0, 1, ... , $|y|$, а его таблицу переходов обозначим *goto*. Предположим, что алфавит слова y фиксирован, имеет размер σ и его элементы являются индексами таблицы t, в которой хранятся состояния. Показанный ниже алгоритм DASG обрабатывает y в онлайновом режиме. Сразу после обработки непустого префикса w значением $t[a]-1$ является самая правая позиция буквы a в w. По-другому можно сказать, что это самое правое из конечных состояний дуг, помеченных буквой a.

```
DASG(y)
 1  for каждой буквы a ∈ alph(y) do
 2      t[a] ← 0
 3  for i ← 0 to |y| − 1 do
 4      for j ← t[y[i]] to i do
 5          goto(j, y[i]) ← i + 1
 6      t[y[i]] ← i + 1
```

Поскольку автомат детерминированный, количество дуг в нем меньше $\sigma|y|$. На самом деле оно не больше $\sigma|y| - \sigma(\sigma-1)/2$. Поэтому инструкция в строке 5 выполняется меньше $\sigma|y|$ раз, так что время работы составляет $O(\sigma|y|)$. Объем дополнительной памяти, занятой таблицей t, равен $O(\sigma)$.

Если алфавит не фиксирован, то буквы, встречающиеся в y, можно сначала отсортировать за время $O(alph(y)\log alph(y))$, чтобы свести задачу к уже рассмотренной. Соответственно, увеличивается полное время работы.

Различение слов. Чтобы найти кратчайшую подпоследовательность, различающую два разных слова, можно воспользоваться общим алгоритмом проверки эквивалентности двух детерминированных конечных автоматов. Этот алгоритм – стандартное применение структуры данных UNION-FIND, он требует времени $O(n\log^* n)$, где n – длина меньшего слова.

Примечания

Понятие автомата подпоследовательностей впервые было введено в работе Baeza–Yates [21] и впоследствии названо DASG в работе Troníček and Melichar [231]. В построении Баеза–Йейтса слово обрабатывается справа налево, а не так, как в приведенном выше алгоритме. Обобщение автомата на конечное множество слов можно найти в работах [21, 100]. Размер DASG проанализирован в работе [232].

Проверка эквивалентности детерминированных автоматов описана Хопкрофтом и Карпом в 1971 году (см. [4]) как применение структуры данных UNION-FIND. Другое описание и анализ структуры см. в работе [63].

52. Проверка однозначности декодирования

Множества слов, особенно двоичных, используются для кодирования информации, например в протоколах передачи, при сжатии данных или в простых текстах. Чтобы извлечь исходную информацию, потоки данных подвергаются разбору в соответствии с множеством слов. Разбор является простой операцией, если все кодовые слова одинаковой длины, как, например, в кодировках символов ASCII или UTF-32, поскольку в этом случае существует единственная факторизация закодированных данных.

Кодом называется множество слов, обладающее свойством единственности декодирования. Вопрос о единственности разбора возникает в основном для кодов переменной длины. В этой задаче наша цель – проверить, является ли множество слов кодом.

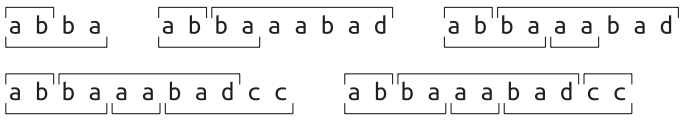
Точнее, множество $C = \{w_1, w_2, \ldots, w_n\}$ слов над алфавитом A является кодом, если для любых двух последовательностей (обозначаемых как слова) $i_1 i_2 \ldots i_k$ и $j_1 j_2 \ldots j_l$ индексов, принадлежащих множеству $\{1, 2, \ldots, n\}$, имеем

$$i_1 i_2 \ldots i_k \neq j_1 j_2 \ldots j_l \Rightarrow w_{i_1} w_{i_2} \ldots w_{i_k} \neq w_{j_1} w_{j_2} \ldots w_{j_l}.$$

Иными словами, если определить морфизм h из $\{1, 2, \ldots, n\}^*$ в A^* как $h(i) = w_i$ для $i \in \{1, 2, \ldots, n\}$, то это условие означает, что h инъективен.

Множество $C_0 = \{\texttt{ab},\texttt{abba},\texttt{baccab},\texttt{cc}\}$ не является кодом, поскольку слово $\texttt{abbaccab} \in C^*$ можно разложить на факторы, принадлежащие C_0, двумя способами: $\texttt{ab} \cdot \texttt{baccab}$ и $\texttt{abba} \cdot \texttt{cc} \cdot \texttt{ab}$. С другой стороны, множество $C_1 = \{\texttt{ab},\texttt{bacc},\texttt{cc}\}$ является кодом, потому что любое слово, принадлежащее C_1^*, может начинаться только одним словом из C_1. Говорят, что этот код является префиксным (никакое слово $u \in C$ не является собственным префиксом другого слова $v \in C$).

Чтобы проверить, является ли множество $C_2 = \{\texttt{ab},\texttt{abba},\texttt{baaabad},\texttt{aa},\texttt{badcc},\texttt{cc},\texttt{dc cbad},\texttt{badba}\}$ кодом, мы можем попытаться построить принадлежащее C_2^* слово, допускающее две разные факторизации. Вот серия таких попыток:



На каждом шаге мы получаем остаток, именно ba, aabad, bad и cc, который пытаемся устранить. В конечном итоге мы получили две факторизации, потому что последний остаток является пустым словом. Таким образом, C_2 кодом не является.

Размер N задачи о проверке однозначности декодирования для конечного множества слов равен суммарной длине $||C||$ всех слов, принадлежащих C.

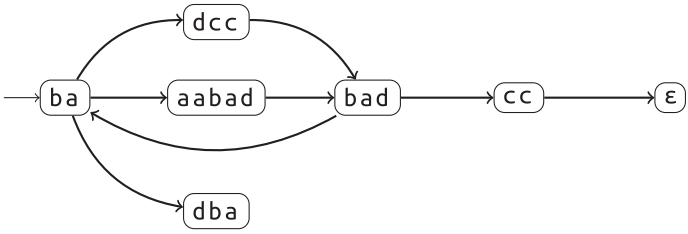
Вопрос. Спроектировать алгоритм, который проверяет, является ли конечное множество C слов кодом, и требует времени $O(N^2)$.

Решение

Чтобы ответить на этот вопрос, мы сведем проверку того, является ли C кодом, к задаче на графе $G(C)$. Вершинами $G(C)$ являются остатки, образовавшиеся при попытках найти две факторизации, которые, следовательно, являются также суффиксами слов из C (включая и пустое слово).

Вершины $G(C)$ определяются путем обхода «в ширину». Начальные вершины на уровне 0 имеют вид $u^{-1}v$, где $u, v \in C$ – разные слова. Это множество может оказаться пустым, если C – префиксный код. На уровне $k+1$ вершинами являются слова из $C^{-1}D_k \cup D_k^{-1}C$, где D_k – вершины на уровне k. Множество вершин включает пустое слово, называемое стоком. В $G(C)$ существует ребро, ведущее из u в v, в том случае, когда $v = z^{-1}u$ или $v = u^{-1}z$, где $z \in C$.

На рисунке ниже показан граф $G(C_2)$, в котором имеется только одна начальная вершина, а столбцы соответствуют уровням вершин. Множество C_2 не является кодом, потому что существует путь из начальной вершины в сток. Средний из таких путей соответствует двум найденным выше факторизациям. На самом деле из-за наличия цикла в графе существует бесконечно много слов, допускающих более одной факторизации.



Наблюдение. Множество C является кодом тогда и только тогда, когда в графе $G(C)$ не существует пути из начальной вершины в сток.

Размер графа $G(C)$ равен $O(N^2)$, поскольку его вершинами являются суффиксы слов из C. Поэтому высказанное наблюдение ведет к эффективной проверке свойства однозначности декодирования. А так как построение и исследование графа можно выполнить за время, пропорциональное размеру графа, то для этой проверки требуется время порядка $O(N^2)$.

Примечания

Алгоритм проверки однозначности декодирования для конечного множества слов предложен в работе Sardinas and Paterson [217]. Формальное доказательство наблюдения приведено в книгах [175, глава 1] и [36, глава 1].

Алгоритм можно реализовать с помощью префиксного дерева множества, дополненного подходящими ссылками. Время его работы составит $O(nN)$, где n – максимальная длина слова; см. [15].

53. Таблица LPF

В этой задаче рассматривается еще одна таблица, определенная для слов и не совсем удачно названная таблицей ***наибольших предыдущих факторов*** (longest previous factor – LPF). Это полезное средство факторизации слов для сжатия данных (см. задачу 97) и вообще для проектирования эффективных алгоритмов нахождения повторов в текстах.

Для непустого слова y в таблице LPF хранятся длины повторяющихся факторов. Точнее, если j – позиция в y, то LPF[j] содержит максимальную длину факторов, начинающихся в позиции j и в какой-то предыдущей (т. е. меньшей) позиции. Ниже показана такая таблица для слова `abaababababbabbb`.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b
LPF[j]	0	0	1	3	2	4	3	2	1	4	3	2	2	1

Приведенный ниже алгоритм вычисляет таблицу LPF для поданного на вход слова y. В нем используется суффиксный массив y и таблица Rank, содержащая ранги суффиксов в лексикографическом порядке. Таблицы *prev* и *next* содержат ссылки для спискового представления рангов суффиксов.

```
LPF(непустое слово y)
 1  for r ← 0 to |y| − 1 do
 2      (prev[r], next[r]) ← (r − 1, r + 1)
 3  for j ← |y| − 1 downto 0 do
 4      r ← Rank[j]
 5      LPF[j] ← max{LCP[r], LCP[next[r]]}
 6      LCP[next[r]] ← min{LCP[r], LCP[next[r]]}
 7      if prev[r] ≥ 0 then
 8          next[prev[r]] ← next[r]
 9      if next[r] < |y| then
10          prev[next[r]] ← prev[r]
11  return LPF
```

Вопрос. Показать, что алгоритм LPF правильно вычисляет таблицу LPF и требует линейного времени.

Внимательное изучение алгоритма показывает, что он делает больше, чем было заложено при проектировании: длины в таблице LPF являются перестановками длин в таблице наибольших общих префиксов LCP.

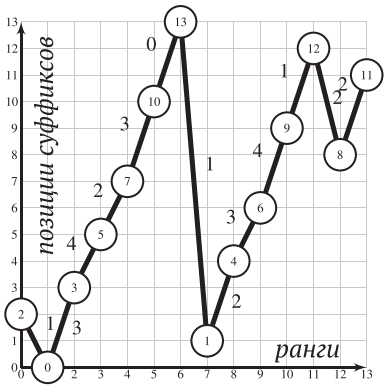
Вопрос. Показать, что элементы таблицы LPF образуют перестановку элементов таблицы LCP и что таблицу LCP можно преобразовать в таблицу LPF.

Решение

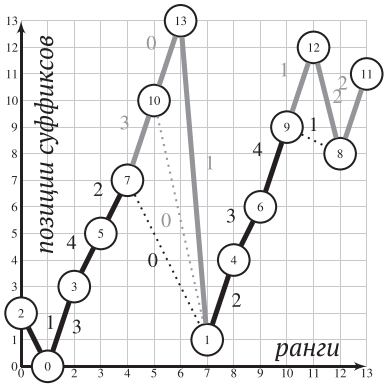
Анализ алгоритма LPF становится очевидным, если изобразить суффиксный массив входного слова графически. Ниже показаны суффиксный массив слова `abaababababbabbb` и ранги его суффиксов.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
y[*j*]	a	b	a	a	b	a	b	a	b	b	a	b	b	b	
Rank[*j*]	1	7	0	2	8	3	9	4	12	10	5	13	11	6	
r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[*r*]	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP[*r*]	0	1	3	4	2	3	0	1	2	3	4	1	2	2	0

На верхнем из двух следующих рисунков показано графическое представление суффиксного массива рассматриваемого слова. Позиции суффиксов изображены в порядке возрастания рангов (по оси *x*), и для каждой позиции по оси *y* отложен ее номер. Ссылка между позициями с рангом $r - 1$ и r снабжена меткой LCP[*r*].



Наблюдение. Длина LCP между позицией с рангом $r - 1$ (соответственно r) и любой позицией с большим (соответственно меньшим) рангом не превышает LCP[*r*].



Идея решения вытекает непосредственно из этого наблюдения. Если позиция j с рангом r находится в верхней точке графика, то ассоциированная с ней длина LPF равна максимуму из LCP[r] и LCP[$r+1$]. А длина LCP между предыдущей и следующей позициями равна минимуму из этих двух значений. Именно в этом и состоит смысл сравнений в строках 5–6.

Это также объясняет, почему позиции обрабатываются в порядке от наибольшей к наименьшей – потому что тогда каждая позиция по очереди оказывается в верхней точке графика.

Следующие инструкции в алгоритме LPF служат для управления списком позиций как двунаправленным; для этой цели введены таблицы *prev* и *next*. Смысл инструкций в строках 8 и 10 – удалить из списка позицию j ранга r.

На нижнем из двух предыдущих рисунков изображена ситуация, складывающаяся после обработки позиций с 13 по 10 (окрашены серым цветом). Пунктирные ссылки по-прежнему помечены значениями LCP.

Это показывает, что алгоритм LPF правильно вычисляет искомую таблицу LPF.

Решение второго вопроса. Приведенное выше рассуждение показывает также, что элементы таблицы LPF являются перестановкой элементов таблицы LCP суффиксного массива y.

Для преобразования таблицы LCP в таблицу LPF входного слова нужно заменить строки 5–6 алгоритма LPF такими:

```
5   if LCP[r] < LCP[next[r]] then
6       (l, LCP[r], LCP[next[r]]) ← (LCP[r], LCP[next[r]], l)
```

где в строке 6 два элемента таблицы обмениваются местами. Алгоритм порождает таблицу LCP′, соответствующую LPF, т. к. LPF[SA[r]] = LCP′[r], или, эквивалентно, LPF[j] = LCP′[Rank[j]]. Если отсортировать пары (SA[r], LCP′[r]) по первому элементу, то во втором элементе будут находиться значения в таблице LPF.

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[r]	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP′[r]	1	0	3	4	2	3	1	0	2	3	4	2	1	2	0

В нашем примере алгоритм порождает показанную выше таблицу LCP′, из которой мы, например, заключаем, что LPF[2] = 1 (соответствует второму вхождению a), потому что 2 и 1 имеют одинаковый ранг $r = 0$.

Примечания

Первый алгоритм, за линейное время вычисляющий таблицу LPF слова по его суффиксному массиву, описан в работе [76]. Более эффективные алгоритмы разработаны в [78], где показано, что вычисление можно провести оптимально по памяти и времени с помощью алгоритма с линейным временем работы, который потребляет под стек дополнительную память объемом всего $O\left(\sqrt{|y|}\right)$.

В работе [87] представлено три варианта таблицы LPF и соответствующие алгоритмы ее построения; см. также [50, 52, 99].

54. Сортировка суффиксов слов Туэ–Морса

Слова Туэ–Морса с их специальной структурой дают примеры ситуации, когда некоторые алгоритмы, требующие линейного или большего времени, можно оптимизировать, так чтобы они работали за логарифмическое время. В этой задаче приведен такой пример, относящийся к суффиксным массивам слов.

Бесконечное слово Туэ–Морса получается путем многократного применения морфизма Туэ–Морса μ, отображающего множество $\{0,1\}^*$ в себя:

$$\begin{cases}\mu(0) = 01\\ \mu(1) = 10\end{cases}.$$

Слово Туэ–Морса τ_n равно $\mu^n(0)$ для натурального n. Такой способ определения слов Туэ–Морса подходит для рекурсивного описания массива SA_n, содержащего начальные позиции непустых суффиксов τ_n, отсортированных в лексикографическом порядке. Например, $\tau_3 = 01101001$ и $SA_3 = [5,6,3,0,7,4,2,1]$.

Вопрос. Пусть даны целые числа n и k, $0 \le k < n$. Показать, как можно вычислить $SA_n[k]$ за время $O(n)$ для слова τ_n длины 2^n.

Решение

Начнем с двух наблюдений, относящихся к слову τ_n.

Наблюдение 1. Пусть i – четная позиция в τ_n. Тогда $\tau_n[i] \ne \tau_n[i+1]$.

Для $c \in \{0,1\}$ определим I^c_{odd} (соответственно I^c_{even}) как множество нечетных (соответственно четных) позиций i, для которых $\tau_n[i] = c$. Наблюдение 2, в котором suf_i определено как $\tau_n[i..2^n - 1]$, следует из наблюдения 1.

Наблюдение 2.

(a) Если $i \in I^0_{\text{odd}}$, $j \in I^0_{\text{even}}$ и $suf_i \ne 01$, то $suf_i < suf_j$.

(b) Если $i \in I^1_{\text{even}}$, $j \in I^1_{\text{odd}}$ и $suf_j \ne 1$, то $suf_i < suf_j$.

По-другому наблюдение 2 можно сформулировать так:

$$I^0_{\text{odd}} < I^0_{\text{even}} < I^1_{\text{even}} < I^1_{\text{odd}}.$$

Для последовательности S целых чисел и двух целых чисел p и q обозначим $p \cdot S$ и $S + q$ соответственно последовательности элементов S, умноженных на p и увеличенных на q. Например, $2 \cdot [1,2,3] = [2,4,6]$ и $[1,2,3] + 3 = [4,5,6]$.

Рассмотрим два случая: для четных и нечетных n.

Случай четного n**.** Если n четно, то таблица SA_n связана с SA_{n-1} следующим образом. Обозначим α и β две половины SA_{n-1} ($SA_{n-1} = [\alpha, \beta]$); тогда

$$SA_n = [2 \cdot \beta + 1,\ 2 \cdot \alpha,\ 2 \cdot \beta,\ 2 \cdot \alpha + 1]. \qquad (*)$$

Доказательство. Пусть X – множество начальных позиций суффиксов слова. Обозначим $sorted(X)$ – отсортированный список этих позиций, когда суффиксы упорядочены лексикографически. Также положим

$$\gamma_1 = sorted(I^0_{odd}),\ \gamma_2 = sorted(I^0_{even}),$$
$$\gamma_3 = \mathrm{sorted}(I^1_{even}),\ \gamma_3 = sorted(I^1_{odd}).$$

Тогда, в силу наблюдения 2, $SA_n = [\gamma_1, \gamma_2, \gamma_3, \gamma_4]$.

По счастью, для четных n в τ_n нет плохих суффиксов `01` и `1`. Мы можем воспользоваться определением слов Туэ–Морса с помощью морфизма. Сначала заметим, что морфизм μ сохраняет лексикографический порядок ($u < v \Leftrightarrow \mu(u) < \mu(v)$). Каждый суффикс τ_{n-1} в позиции i морфизм μ отображает в суффикс τ_n в позиции $2i$. Значит, $2 \cdot SA_{n-1} = [2 \cdot \alpha, 2 \cdot \beta]$ – последовательность отсортированных суффиксов в четных позициях τ_n.

Тогда, в силу предыдущего наблюдения, $SA_n = [\gamma_1, \gamma_2, \gamma_3, \gamma_4]$, где γ_1 соответствует отсортированным суффиксам, начинающимся во вторых позициях суффиксов, ассоциированных с $2 \cdot \beta$. И аналогично для γ_4 и $2 \cdot \alpha$. Поэтому имеем $SA_n = [2 \cdot \beta + 1, 2 \cdot \alpha, 2 \cdot \beta, 2 \cdot \alpha + 1]$, что и требовалось доказать.

Вычисление SA_4 по SA_3. $SA_3 = [5,6,3,0,7,4,2,1]$ состоит из $\alpha = [5,6,3,0]$ и $\beta = [7,4,2,1]$. Имеем $2 \cdot \beta = [14,8,4,2]$, $2 \cdot \beta + 1 = [15,9,5,3]$, $2 \cdot \alpha = [10,12,6,0]$ и $2 \cdot \alpha + 1 = [11,13,7,1]$, что дает

$$SA_4 = [15,9,5,3,\ 10,12,6,0,\ 14,8,4,2,\ 11,13,7,1].$$

Случай нечетного n. Если n нечетно, то мы также можем применить формулу ($*$) с тем отличием, что *плохие* суффиксы `01` и `1` следует *специально* поместить на правильные места: суффикс `1` должен предшествовать всем остальным суффиксам, начинающимся с `1`, а суффикс `01` должен быть помещен сразу после всей последовательности суффиксов, начинающихся с `00`. Поэтому изменение сводится к вычислению количества $p(n)$ вхождений `00` в τ_n.

Начало последовательности чисел $p(n)$ для $n = 2, 3, \ldots, 10$ имеет вид 0, 1, 2, 5, 10, 21, 42, 85, 170. Эти числа удовлетворяют рекуррентным соотношениям

$$p(1) = 0,\ p(2k+1) = 4 \cdot p(2k-1) + 1,\ p(2k+2) = 2 \cdot p(2k+1). \qquad (**)$$

Следовательно, $p(2k+1) = (4^k - 1)/3$.

Вычисление SA_5 по SA_4. Для этого сначала применим преобразование ($*$) и получим четыре блока:

29,17,9,5,23,27,15,3	**30**,18,10,6,20,24,12,0,
28,16,8,4,22,26,14,2	**31**,19,11,7,21,25,13,1.

Плохие суффиксы `01` и `1` начинаются в позициях 30 и 31. Число 30 следует переместить, поставив после 5-го элемента 23, т. к. $p(5) = 5$. Число 31, соответствующее однобуквенному суффиксу, следует переместить в начало третьей четверти (это наименьший суффикс, начинающийся буквой `1`). Окончательную таблицу суффиксов SA_5 получаем путем конкатенации:

29,17,9,5,23,**30**,27,15 3,18,10,6,20,24,12,0,
31,28,16,8,4,22,26,14 2,19,11,7,21,25,13,1.

Заключение. Для ответа на вопрос, как быстро вычислить $SA_n[k]$, перечислим необходимые для этого действия:

- установить, в какой четверти SA_n находится число;
- свести задачу к вычислению $SA_{n-1}[j]$, где позиция j (равная приблизительно половине k) вычисляется по формуле (∗), применяемой в обратном направлении;
- если n нечетно, не забыть переместить два *плохих* суффикса на новые места в SA_n. Для вычисления новых позиций используется значение $p(n)$, вычисляемое по формуле (∗∗);
- повторять эти шаги сведéния, пока не получится таблица постоянного размера.

В целом для вычисления $SA_n[k]$ достаточно $O(n)$ шагов, т. е. время логарифмически зависит от размера таблицы SA_n.

Примечания

Может создаться впечатление, что возможен другой подход с использованием компактного автомата факторов для слов Туэ–Морса, как описано в работе [204]. Однако он ведет к еще более сложному решению.

55. Простое построение суффиксного дерева

Суффиксные деревья – подходящая структура данных для индексирования текстов. Процедура их построения, оптимальная по времени, требует большого объема памяти – большего, чем суффиксные массивы, применяемые для той же цели. В этой задаче рассматривается умеренно эффективный, очень простой, но не совсем наивный метод построения суффиксных деревьев.

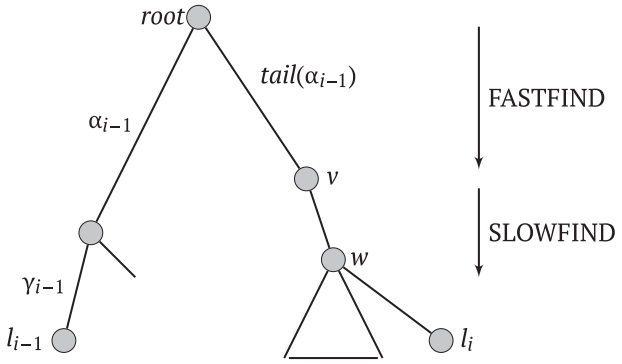
Суффиксное дерево T слова x, оканчивающегося уникальным маркером, – это уплотненное префиксное дерево суффиксов x. Листовые узлы соответствуют суффиксам, а внутренние – факторам, которые встречаются хотя бы два раза с разными последующими буквами. Каждое ребро помечается фактором $x[i..j]$ слова x, представляющим пару (i, j). Длина этого фактора, или его значность, $|x[i..j]| = j - i + 1$. Значностью пути в T называется сумма значностей его ребер, тогда как длиной пути называется количество составляющих его ребер. Обозначим $depth(T)$ максимальную длину пути в T от корня до листа. Пусть l_i – конечный лист ветви, помеченной фактором $x[i..n-1]$.

Вопрос. Спроектировать алгоритм построения суффиксного дерева T слова x, в котором не используется дополнительный массив и который требует времени $O(|x|depth(T))$ для алфавита фиксированного размера.

Решение

Основная идея решения – итеративно вставлять суффиксы в дерево, начиная с самого длинного суффикса x и заканчивая самым коротким.

Обозначим T_{i-1} уплотненное префиксное дерево суффиксов, начинающихся в позициях 0, 1, ..., $i-1$ слова x. Мы покажем, как получить T_i путем модификации T_{i-1}.



i-й суффикс можно разделить на части $\alpha_i\gamma_i$, где $\alpha_i = x[i..i + d_i - 1]$, $\gamma_i = x[i + d_i..n - 1]$. Слово α_i – это метка пути от корня *root* к $w = parent(l_i)$ (см. рисунок). В частности, $\alpha_i = \varepsilon$, если $parent(l_i) = root$.

Если a – первая буква слова au, то $tail(au) = u$. Заметим, что слово $\alpha_k \neq \varepsilon$ встречается в позиции k и в какой-то меньшей позиции. Следовательно, $tail(\alpha_k)$ встречается в позиции $k + 1$ и в какой-то меньшей. Отсюда вытекает важнейший факт.

Наблюдение. Предположим, что $\alpha_{i-1} \neq \varepsilon$. Тогда существует путь в T_{i-1}, на котором написано слово $tail(\alpha_{i-1})\gamma_{i-1}$ (см. рисунок). Иначе говоря, значительная часть вставляемого суффикса $x[i..n]$ уже присутствует в дереве.

В алгоритме используется два типа обхода дерева.

- *FastFind*(α) предполагает, что α присутствует в текущем дереве (в виде метки пути). Узел v ищется путем выписывания слова α по буквам. Если выписывание заканчивается в середине метки ребра, то узел v создается.
  Обход управляется длиной d слова α. Ребра дерева используются как *сокращения*, т. е. читается только первый символ и длина ребра. Стоимость такого обхода составляет $O(depth(T))$.
- *SlowFind*(v, γ) ищет самого низкого потомка w узла v в текущем дереве, следуя по пути, помеченному самым длинным возможным префиксом β слова γ. Как и выше, иногда узел w приходится создавать.
  Обход производится посимвольно, и на каждом шаге обновляется значение d. Его стоимость составляет $O(|\beta|)$.

Алгоритм в целом начинается с дерева, содержащего одно ребро, помеченное всем словом $x[0..n-1]$, и выполняет следующие действия для суффиксов в позициях $i = 1,2, ..., n-1$.

Одна итерация, переход от T_{i-1} к T_i:
if $\alpha_{i-1} = \varepsilon$ then $v \leftarrow root$ else v $\leftarrow$ *FastFind*($tail(\alpha_{i-1})$),
$w \leftarrow$ *SlowFind*(v, γ_{i-1}),
создаются новый лист l_i и новое ребро $w \to l_i$.

Время работы алгоритма. Всего выполняется $n - 1$ итераций. На каждой итерации стоимость *FastFind* равна $O(depth(T))$. Суммарная стоимость всех *SlowFind* равна $O(n)$, т. к. при каждом перемещении длина слова γ_i уменьшается. Полная временная сложность равна $O(n \cdot depth(T))$.

Заметим, что алгоритм не требует дополнительного массива, чего мы и хотели добиться.

Примечания
Описанный здесь алгоритм – упрощенный вариант построения суффиксного дерева, предложенного Маккрейтом в работе [187], который работает линейное время, но требует дополнительных массивов. Наш вариант несколько медленнее, но значительно проще алгоритма Маккрейта. Его можно также рассматривать как первый шаг на пути к пониманию полного алгоритма. Кстати говоря, для многих типов слов коэффициент при $depth(T)$ растет логарифмически.

Алгоритм Укконена [234] можно модифицировать аналогичным образом, что дает еще один простой, но не наивный способ построения суффиксных деревьев.

56. Сравнение суффиксов слова Фибоначчи

Структура слов Фибоначчи, похожая на структуру слов Туэ–Морса, – пример ситуации, когда некоторые алгоритмы удается оптимизировать, так что время их работы логарифмически зависит от длины слова. В этой задаче в качестве наглядной демонстрации данного явления рассматривается быстрое лексикографическое сравнение двух суффиксов конечного слова Фибоначчи.

Чтобы упростить рассуждения, мы будем работать с несколько сокращенными словами Фибоначчи. Обозначим g_n n-е слово Фибоначчи fib_n, из которого удалены две последние буквы, т. е. $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$, где $n \geq 2$. Пусть $suf(k,n)$ – k-й суффикс $g_n[k..|g_n| - 1]$ слова g_n. Например, $g_2 = \mathtt{a}$, $g_3 = \mathtt{aba}$, $g_4 = \mathtt{abaaba}$, $g_5 = \mathtt{abaababaaba}$ и $suf(3,5) = \mathtt{abababaaba}$.

Сравнение суффиксов g_n можно свести к сравнению их компактного представления логарифмической длины, вытекающего из разложения логарифмического размера (см. свойство ниже). Факторами в этом разложении выступают обратные слова Фибоначчи $R_n = fib_n^{\mathrm{R}}$. Перечислим первые факторы: $R_0 = \mathtt{a}$, $R_1 = \mathtt{ba}$, $R_2 = \mathtt{aba}$, $R_3 = \mathtt{baaba}$. Заметим, что $R_{n+2} = R_{n+1}R_n$ и что R_n начинается буквой a тогда и только тогда, когда n четно.

Свойство. Для $n > 2$ $suf(k, n)$ однозначно разлагается в произведение $R_{i_0}R_{i_1} \ldots R_{i_m}$, где $i_0 \in \{0,1\}$ и $i_t \in \{i_{t-1} + 1, i_{t-1} + 2\}$ для $t = 1, \ldots, m$.

Имея в виду эту факторизацию, обозначим $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$. Например, $suf(3,5) = \mathtt{abababaaba} = R_0 \cdot R_1 \cdot R_3 = \mathtt{a{\cdot}ba{\cdot}baaba}$ и $\mathcal{R}_5(3) = (0,1,3)$.

Вопрос. (A) Показать, как сравнить любые два суффикса $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$ за время $O((\log|fib_n|)^2)$.
(B) Улучшить время работы до $O(\log|fib_n|)$.

[**Указание:** в случае (A) воспользоваться алгоритмом из задачи 7.]

Решение
Ассоциируем с $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$ следующую величину

$$\Psi_n(k) = first(R_{i_0})first(R_{i_1}) \ldots first(R_{i_m}),$$

где $first(w)$ обозначает первую букву w. Можно проверить, что

Наблюдение. $suf(p, n) < suf(q, n) \Leftrightarrow \Psi_n(p) < \Psi_n(q)$.

Пример. Для $g_5 = \mathtt{abaababaaba}$ имеем $suf(3,5) = \mathtt{a}\cdot\mathtt{ba}\cdot\mathtt{baaba} = R_0R_1R_3$ и $suf(5,5) = \mathtt{a}\cdot\mathtt{ba}\cdot\mathtt{aba} = R_0R_1R_2$. Тогда $\Psi_5(3) = \mathtt{abb}$ и $\Psi_5(5) = \mathtt{aba}$. Поэтому $suf(5,5) < suf(3,5)$, т. к. $\mathtt{aba} < \mathtt{abb}$.

Благодаря этому наблюдению задача сводится к быстрому вычислению разложения в сформулированном выше свойстве, а также функции $\mathcal{R}$.

Часть (A). $\mathcal{R}_n(k)$ можно вычислить следующим образом, просматривая суффикс $suf(n, k)$ слева направо. Если $g_n[k] = \mathtt{a}$, то мы знаем, что $i_0 = 0$; в противном случае $i_0 = 1$. Далее, на каждой итерации $t > 0$ текущая позиция в g_n увеличивается на длину $F_{i_{t-1}+2} = |R_{i_{t-1}}|$, так чтобы указывать на следующую букву g_n. В зависимости от этой буквы мы знаем, какой случай имеет место: $i_t = i_{t-1} + 1$ или $i_t = i_{t-1} + 2$. Таким образом, $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$ вычислено, и процесс потребовал логарифмического числа итераций.

Для доступа к каждой букве g_n требуется время $O(\log|fib_n|)$, если воспользоваться алгоритмом из задачи 7.

В итоге мы получили алгоритм, который решает часть задачи (A) за время $O((\log|fib_n|)^2)$.

Часть (B). Не должно вызывать удивления, что слова Фибоначчи тесно связаны с фибоначчиевой системой счисления (см. задачу 7). Ниже мы покажем, что они связаны с двойственной версией этой системы в контексте лексикографической сортировки.

Ленивая фибоначчиева система счисления. Обозначим $LazyFib(k)$ ленивое представление Фибоначчи натурального числа k, начинающееся с младших цифр. В этой системе число N единственным образом представляется в виде последовательности битов $(b_0, b_1, b_2, \ldots)$, такой что $N = \sum b_i F_{i+2}$, где F_j – последовательные числа Фибоначчи, причем последовательность не содержит двух соседних нулей. Это соответствует условию $i_{t+1} \in \{i_t + 1, i_t + 2\}$, являющемуся частью свойства факторизации.

Например, $LazyFib(9) = (1011)$ и $LazyFib(23) = (011101)$, т. к. $9 = F_2 + F_4 + F_5$ и $23 = F_3 + F_4 + F_5 + F_7$.

Быстрое вычисление разложения. Пусть $(b_0, b_1, b_2, \ldots)$ – разложение длины $|suf(k, n)|$ в ленивой фибоначчиевой системе. Тогда $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$, где i_j – позиции бита 1 в $(b_0, b_1, b_2, \ldots)$.

Поскольку ленивое представление Фибоначчи можно вычислить за логарифмическое время относительно длины fib_n, часть (B) решена.

Примечания

Доказательство свойства факторизации можно найти в работах [213, 238].

Если мы хотим сравнить два суффикса длины больше 2 стандартных (неукороченных) слов Фибоначчи fib_n, то при нечетном n можно использовать ту же самую функцию Ψ. Но если n четно, то нужно заменить $\Psi(k)$ на $\Psi(k) \cdot \mathtt{b}$. Также известно, что для четных n таблица SA суффиксного массива слов Фибоначчи содержит арифметическую прогрессию (по модулю длины массива), и это дает альтернативный способ сравнения для четных n.

Ленивое представление Фибоначчи позволяет вычислить ранг k-го суффикса слова Фибоначчи (его позицию в SA) за логарифмическое время.

57. Устранимость двоичных слов

Некоторые закономерности встречаются во всех достаточно длинных словах. Говорят, что они неустранимы. Это понятие, очевидно, зависит от размера алфавита, и в данной задаче мы будем рассматривать двоичные образцы.

Говорят, что слово w устраняет множество слов X, если ни один фактор w не принадлежит X. Множество X называется устранимым, если существует бесконечное слово, устраняющее его, или, что эквивалентно в случае конечного алфавита, если существует бесконечно много слов, устраняющих его. Наша цель – проверить, является ли устранимым множество слов над двоичным алфавитом $\mathtt{B} = \{\mathtt{a},\mathtt{b}\}$.

Например, множество {aa,abab,bb} невозможно устранить словом длины меньше 5. С другой стороны, множество {aa,bb} устраняется бесконечным словом $(\mathtt{ab})^\infty$.

Для проектирования теста мы определим две редукции на множестве $X \subseteq \mathtt{B}^+$.

reduce1 (удалить суперслово): если $x, y \in X$ и x – фактор y, то удалить y из X.

reduce2 (удалить последнюю букву): если x – суффикс $y \neq \varepsilon$ и $x\bar{a}$, $ya \in X$, то подставить y вместо ya в X (морфизм $\bar{}$ обменивает a и b).

```
AVOIDABLE(непустое множество X двоичных слов)
1  while reduce1 или reduce2 применимы к X do
2      X ← reduce1(X) или reduce2(X)
3  if X ≠ {a,b} return TRUE else return FALSE
```

Пример. Множество $X = \{\texttt{aaa},\texttt{aba},\texttt{bb}\}$ неустранимо, потому что последовательность редукций дает B (измененные слова подчеркнуты): $\{\texttt{aaa},\underline{\texttt{aba}},\texttt{bb}\} \to \{\underline{\texttt{aaa}},\texttt{ab},\texttt{bb}\} \to \{\texttt{aa},\underline{\texttt{ab}},\texttt{bb}\} \to \{\underline{\texttt{aa}},\texttt{a},\texttt{bb}\} \to \{\texttt{a},\underline{\texttt{bb}}\} \to \texttt{B}$.

Вопрос. Показать, что множество двоичных слов X устранимо тогда и только тогда, когда AVOIDABLE(X) = TRUE.

Вопрос. Показать, что множество $X \subseteq \texttt{B}^{\le n}$ устранимо тогда и только тогда, когда оно устраняется словом длины большей, чем $2^{n-1} + n - 2$, и что эта граница точна.

Решение

Правильность алгоритма AVOIDABLE. Этот факт вытекает из следующих двух свойств.

Свойство 1. Если $Y = \text{reduce1}(X)$ или $Y = \text{reduce2}(X)$, то X устранимо тогда и только тогда, когда Y устранимо.

Доказательство. Ясно, что слово, устраняющее Y, устраняет также и X. Чтобы доказать обратное, предположим, что w – бесконечное слово, устраняющее X. Мы покажем, что w устраняет также Y. Это очевидно, если $Y = \text{reduce1}(X)$. Если же $Y = \text{reduce2}(X)$, то существуют два слова $x\bar{a}, ya \in X$ таких, что x – суффикс y и $Y = X \setminus \{ya\} \cup \{y\}$.

Тогда достаточно показать, что w устраняет y. Если это не так, то yb – фактор w. Буква b не может совпадать с a, потому что w устраняет ya. Но она не может совпадать и с $\bar{a}$, потому что w устраняет $x\bar{a}$, являющееся суффиксом $y\bar{a}$. Таким образом, y не является фактором w. ■

Свойство 2. Если к X не применима ни одна редукция и $X \neq \texttt{B}$, то X устранимо.

Доказательство. Чтобы доказать это утверждение, мы инкрементно построим бесконечное слово w, устраняющее X. Пусть v – конечное слово, устраняющее X (v может быть просто буквой, потому что $X \neq \texttt{B}$). Мы утверждаем, что v можно продолжить на одну букву до слова va, которое тоже устраняет X. Действительно, если это не так, то существуют два суффикса x и y слова v, для которых $x\bar{a} \in X$ и $ya \in X$. Так как одно из слов является суффиксом другого, то к X применима операция reduce2, что противоречит предположению. Поэтому v можно продолжить до бесконечного слова, устраняющего X.

Оценка сверху длины устраняющих слов. Следующее свойство поможет нам ответить на второй вопрос.

Свойство 3. $X \subseteq \texttt{B}^{\le n}$ устранимо тогда и только тогда, когда оно устраняется словом, граница которого принадлежит $\texttt{B}^{n-1}$.

В самом деле, если X устранимо, то устраняющее его бесконечное слово имеет фактор, удовлетворяющий этому условию. Обратно, пусть $w = uv = v'u$ – слово, устраняющее X, такое, что $u \in \texttt{B}^{n-1}$. Поскольку $uv^i = v'uv^{i-1} = \ldots =$

$v'^{i}u$, $i > 0$, ясно, что любой фактор uv^i длины n является фактором w. Поэтому uv^∞ также устраняет X.

Чтобы ответить на второй вопрос, осталось доказать часть «только тогда». Если слово длины большей, чем $2^{n-1} + n - 2$, устраняет X, то оно содержит по крайней мере два вхождения некоторого слова из $\mathtt{B}^{n-1}$ и, следовательно, фактор, упоминаемый в свойстве 3, который устраняет X. Таким образом, X устранимо.

Для доказательства точности этой оценки воспользуемся (двоичными) словами де Брёйна порядка $n - 1$. Такое слово w содержит ровно одно вхождение каждого слова из $\mathtt{B}^{n-1}$ и имеет длину $2^{n-1} + n - 2$. Это слово устраняет множество X слов длины n, которые не являются его факторами. Это завершает доказательство, т. к. X неустранимо.

Примечания

Алгоритм AVOIDABLE, безусловно, не самый эффективный алгоритм проверки устранимости множества в двоичном случае, но, пожалуй, самый простой из существующих. Ссылки по теме можно найти в работе [175]. Решение второго вопроса взято из работы [90].

58. Устранимость множества слов

Пусть F – конечное множество слов над конечным алфавитом A, $F \subseteq A^*$. Обозначим $L(F) \subseteq A^*$ язык, составленный из слов, устраняющих F; это означает, что ни одно слово, принадлежащее F, не встречается среди слов, составляющих $L(F)$. Наша цель – построить автомат, допускающий $L(F)$.

Заметим, что если w устраняет u, то оно также устраняет любое слово, фактором которого является u. Таким образом, можно считать, что F – антифакторный язык (факторный код): никакое слово F не является собственным фактором никакого другого слова F. Напротив, $F(L)$ является факторным языком: любой фактор слова $F(L)$ также принадлежит $F(L)$.

Примеры. Для $F_0 = \{\mathtt{aa},\mathtt{bb}\} \subseteq \{\mathtt{a},\mathtt{b}\}^*$ имеем $L(F_0) = (\mathtt{ab})^* \cup (\mathtt{ba})^*$. Для $F_1 = \{\mathtt{aaa},\mathtt{bbab},\mathtt{bbb}\} \subseteq \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ имеем $(\mathtt{ab})^* \subseteq L(F_1)$, а также $(\mathtt{bbaa})^* \subseteq L(F_1)$ и $\mathtt{c}^* \subseteq L(F_1)$.

Вопрос. Показать, что $L(F)$ распознается конечным автоматом, и спроектировать алгоритм, который по префиксному дереву F строит детерминированный автомат, допускающий его.

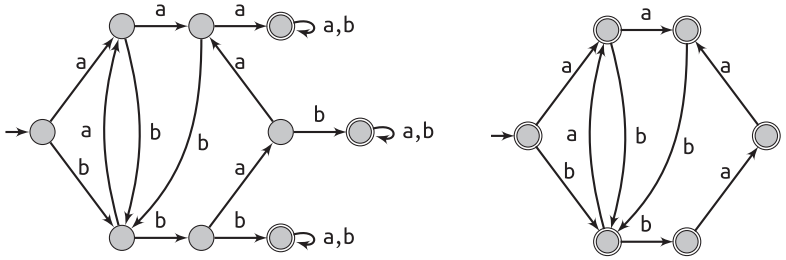
[**Указание:** использовать метод Ахо–Корасик.]

Говорят, что множество F неустранимо, если не существует бесконечных слов, устраняющих его (см. задачу 57). Например, множество F_1 неустранимо над алфавитом $\{\mathtt{a},\mathtt{b}\}$, потому что оно устраняется бесконечным словом $(\mathtt{ab})^\infty$.

Вопрос. Показать, как проверить, является ли множество F неустранимым.

Решение

Предположим, что F – непустое антифакторное множество (в частности, оно не содержит пустого слова), и рассмотрим автомат, допускающий A^*FA^*, т. е. слова, имеющие фактор, принадлежащий F. Состояниями автомата являются префиксы слов из F (или могут быть идентифицированы ими). Действительно, любой такой префикс можно продолжить, породив слово из F, и эти слова образуют состояния-стоки. На левом рисунке ниже показан автомат, допускающий язык $\{a,b\}^*F_1\{a,b\}^*$, в котором вершины, обведенные двойными окружностями, соответствуют словам, имеющим факторы, принадлежащие F_1.



Язык $L(F) = A^* \setminus A^*FA^*$, дополнительный к A^*FA^*, допускается автоматом, допускающим A^*FA^*, после изменения ролей заключительных и незаключительных состояний на противоположные. Таким образом, $L(F)$ допускается конечным автоматом. На правом рисунке изображен автомат, допускающий $L(F_1)$.

Алгоритм AVOIDING повторяет построение автомата F для сопоставления со словарем и в конце изменяет статус состояний и удаляет бесполезные состояния.

```
AVOIDING(префиксное дерево языка F, алфавит A)
 1  q0 ← начальное состояние (корень) префиксного дерева
 2  Q ← ∅        ▷ пустая очередь
 3  for каждой буквы a ∈ A do
 4      if goto(q0, a) не определено then
 5          добавить дугу (q0, a, q0)
 6      else поместить (goto(q0, a), q0) в конец Q
 7  while Q не пусто do
 8      извлечь (p, r) из Q
 9      if r заключительное then
10          сделать p заключительным состоянием
11      for каждой буквы a ∈ A do
12          s ← goto(r, a)
13          if goto(p, a) не определено then
14              добавить дугу (p, a, s)
15          else поместить (goto(p, a), s) в конец Q
```

```
16  удалить все заключительные состояния и ассоциированные с ними дуги
17  сделать все оставшиеся состояния заключительными
18  return преобразованный автомат
```

Алгоритм требует времени $O(|A|\sum\{|w| : w \in F\})$ при подходящей реализации функции *goto*. Но вместо задания всех возможных дуг, исходящих из каждого состояния, можно создавать неуспешную ссылку из состояния p на состояние r, когда (p, r) находится в очереди. Это обычный прием при реализации автоматов такого типа, который позволяет уменьшить размер автомата до $O(\sum\{|w| : w \in F\})$.

Чтобы узнать, является ли F устранимым, нужно лишь проверить, имеется ли цикл в графе, образованном вершинами выходного автомата. Это можно сделать за линейное время, воспользовавшись алгоритмом топологической сортировки. Автомат на правом рисунке выше содержит цикл, и это еще раз показывает, что множество F_1 неустранимо.

Примечания

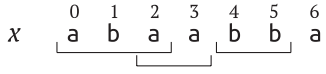
Построение автомата для сопоставления со словарем, называемого также автоматом Ахо–Корасик, описано в работе Aho and Corasick [3] и в большинстве учебников по алгоритмам обработки текста. Обычно автомат реализуется с использованием неуспешных ссылок для экономии памяти.

59. Минимальные уникальные факторы

Предмет этой задачи напоминает понятие минимальных отсутствующих слов. Идею также можно использовать для идентификации, фильтрации или различения файлов. Но соответствующие алгоритмы и комбинаторные свойства, лежащие в их основе, проще.

Минимальным уникальным фактором слова x называется фактор, который встречается в x только один раз и такой, что его собственные факторы повторяются, т. е. встречаются в x по меньшей мере дважды. Минимальный уникальный фактор $x[i..j]$ сохраняется в таблице *MinUniq* с помощью присваивания $MinUniq[j] = i$.

Пример. Минимальными уникальными факторами $x =$ `abaabba` являются `aba` $= x[0\,..\,2]$, `aa` $= x[2..3]$ и `bb` $= x[4..5]$, так что $MinUniq[2] = 0$, $MinUniq[3] = 2$ и $MinUniq[5] = 4$ (остальные значения равны -1).



Алгоритм MinUnique применяется к слову без одиночных букв (каждая буква встречается хотя бы два раза).

```
MinUnique(непустое слово x без одиночных букв)
 1  (SA, LCP) ← суффиксный массив x
 2  for i ← 0 to |x| do
 3      MinUniq[i] ← −1
 4  for r ← 0 to |x| − 1 do
 5      l ← max{LCP[r], LCP[r + 1]}
 6      MinUniq[SA[r] + l] ← max{MinUniq[SA[r] + l], SA[r]}
 7  return MinUniq[0..|x| − 1]
```

Вопрос. Показать, что алгоритм MinUnique вычисляет таблицу *MinUniq*, ассоциированную с входным словом x без одиночных букв.

Существует двойственность между минимальными уникальными факторами и максимальными вхождениями повторений в слово x.

Вопрос. Показать, что минимальный уникальный фактор индуцирует два максимальных вхождения повторений слова без одиночных букв.

Вопрос. Сколько минимальных уникальных факторов имеется в слове де Брёйна порядка k?

Решение

Набросок доказательства правильности. Понятие минимального уникального фактора слова близко к понятию идентификатора позиции в слове. Идентификатором позиции i в слове $x\#$ (# – маркер конца) называется кратчайший префикс слова $x\#[i..|x|]$, встречающийся в $x\#$ ровно один раз. Тогда, если фактор ua с буквой a является идентификатором позиции i, то u встречается в x по меньшей мере дважды, что соответствует длине l, вычисляемой в строке 5 алгоритма MinUnique.

В алгоритме неявно используются идентификаторы, потому что минимальный уникальный фактор является самым коротким из всех идентификаторов, заканчивающихся в данной позиции, скажем j. Это делается в строке 6, где $j = \mathrm{SA}[r] + l$, и $MinUniq[j]$ обновляется соответственно.

Ниже иллюстрируется вычисление минимальных уникальных факторов слова `abaabba`. Значение $MinUniq[7] = 6$ отбрасывается, когда нет маркера конца. Когда $r = 3$, $MinUniq[5]$ устанавливается равным 3 и в конечном итоге заменяется на 4, когда $r = 6$. Все три неотрицательных значения, 0, 2 и 4, соответствуют указанным ранее факторам.

r	0	1	2	3	4	5	6	7
SA[r]	6	2	0	3	5	1	4	
LCP[r]	0	1	1	2	0	2	1	0
j	0	1	2	3	4	5	6	7
$MinUniq[j]$	−1	−1	−1	−1	−1	−1	−1	−1
			0	2		3		6
						4		

Максимальные повторения. Минимальный уникальный фактор в слове x без одиночных букв имеет вид aub, где u – слово, а a, b – буквы, потому что его нельзя свести к одиночной букве. Тогда и au, и ub встречаются в x по меньшей мере дважды, т. е. являются повторениями. Вхождение au (определенное вхождением aub) можно продолжить влево до максимального вхождения повторения. Аналогично вхождение ub можно продолжить вправо до максимального вхождения повторения. Тем самым мы ответили на второй вопрос.

Слова де Брёйна. В слове де Брёйна порядка k над алфавитом A каждое слово длины k встречается ровно один раз. Более короткие слова повторяются. В слове де Брёйна существует ровно $|A|^k$ минимальных уникальных факторов длиной $|A|^k + k - 1$.

Примечания

Части этой задачи взяты из работы Ilie and Smyth [148]. Вычисление кратчайших уникальных факторов описано в работе [233]. Вычисление минимальных уникальных факторов в скользящем окне обсуждается в работе [189].

Вычисление идентификаторов – прямое применение суффиксных деревьев (см. [98, глава 5]). Минимальные уникальные факторы можно продолжить влево и получить все идентификаторы позиций в слове.

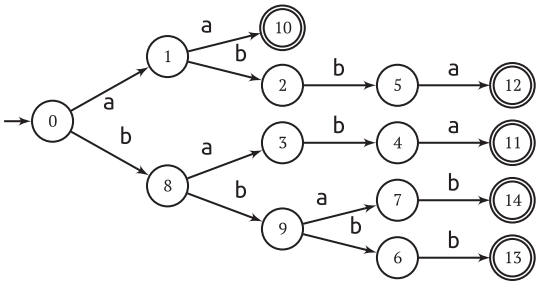
В геномике минимальный уникальный фактор, называемый маркером, расположен в известном месте хромосомы и используется для идентификации особей или видов.

60. Минимальные отсутствующие слова

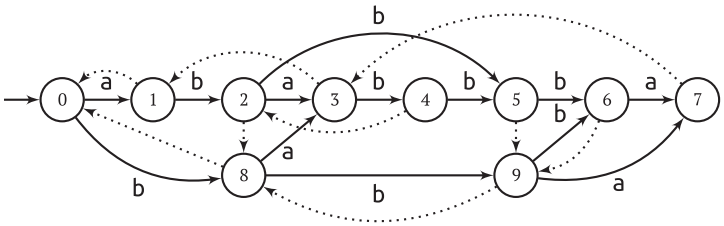
Слова, не встречающиеся в файлах, можно применить для отбрасывания или различения файлов. Они играют роль набора факторов файла, но допускают более компактное представление с помощью префиксного дерева в виде факторных кодов.

Говорят, что слово w, $|w| > 1$, отсутствует или запрещено в слове x, если оно не встречается в x. Отсутствующее слово называется минимальным, если оба слова $w[0..|w| - 2]$ и $w[1..|w| - 1]$ встречаются в x.

Минимальными словами, отсутствующими в `ababbba`, являются `aa`, `abba`, `baba`, `bbab` и `bbbb`. В показанном ниже префиксном дереве с ними ассоциированы листовые узлы (обведенные двойной окружностью), потому что ни одно минимальное отсутствующее слово не может являться фактором другого.



Вычисление минимальных слов, отсутствующих в слове x, естественно начать с его факторного автомата $\mathcal{F}(x)$ – наименьшего детерминированного автомата, допускающего все его факторы. Каждый фактор записывается на пути, начинающемся в начальном состоянии, и все состояния автомата заключительные. Ниже показан факторный автомат слова ababbba вместе с неуспешными ссылками (пунктирные ребра).



Вопрос. Спроектировать алгоритм, который за линейное время вычисляет префиксное дерево минимальных отсутствующих слов слова x, зная его факторный автомат $\mathcal{F}(x)$.

[**Указание:** воспользуйтесь неуспешными ссылками автомата.]

Вопрос. Спроектировать алгоритм, который за линейное время восстанавливает факторный автомат слова x по его префиксному дереву минимальных отсутствующих слов.

[**Указание:** воспользуйтесь методом Ахо–Корасик.]

Решение

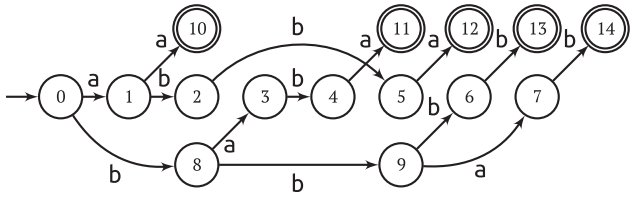
Вычисление минимальных отсутствующих слов. Приведенный ниже алгоритм работает с факторным автоматом $\mathcal{F}(x)$ своего входного слова. Автомат над алфавитом A представляет собой совокупность множества состояний Q, в котором выделено начальное состояние i, и функции переходов *goto*, представленной дугами на рисунке выше. Алгоритм находит отсутствующие слова, рассматривая неопределенные дуги.

Алгоритм обходит автомат в ширину, чтобы гарантировать минимальность отсутствующих слов. В строке 3 он проверяет, не является ли уже некоторый собственный префикс кандидата отсутствующим словом, для чего использует неуспешную ссылку автомата (побочный продукт эффективных алгоритмов построения этого автомата). Ссылка ведет на самый длинный суффикс слова, ассоциированный с другим состоянием. Алгоритм преобразует автомат, добавляя новые состояния и вычисляя новую функцию переходов $goto'$.

```
MinimalAbsentWords(непустое слово x)
 1   (Q, A, i, Q, goto) ← F(x) ▷ факторный автомат x и его функция
                                 неуспехов fail
```

```
2   for каждого p ∈ Q в порядке обхода в ширину из i и каждого a ∈ A do
3       if goto(p, a) не определено и goto(fail(p), a) определено then
4           goto′(p, a) ← новое заключительное состояние
5       elseif goto(p, a) = q и q еще не посещалось then
6           goto′(p, a) ← q
7   return (Q ∪ {новые заключительные состояния}, A, i, {новые
            заключительные состояния}, goto′)
```

Применительно к нашему примеру `ababbba` алгоритм порождает префиксное дерево минимальных отсутствующих слов, нарисованное ниже подругому, чтобы показать сходство со структурой автомата.

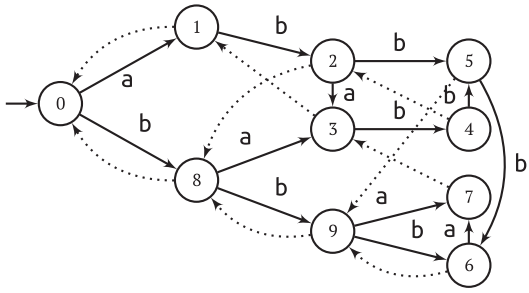


Время работы. Известно, что построить факторный автомат слова можно за линейное время, главным образом потому, что размер структуры линейно зависит от длины слова независимо от размера алфавита. Остальные строки алгоритма, очевидно, занимают время $O(|A| \times |x|)$, если функция переходов и неуспешные ссылки реализованы в виде массивов.

Вычисление факторного автомата. Приведенный ниже алгоритм строит факторный автомат слова x, зная префиксное дерево $(Q, A, i, T', goto')$ его минимальных отсутствующих слов. Построение опирается на функцию неуспехов *fail*, определенную на состояниях префиксного дерева. Весь процесс напоминает построение автомата сопоставления со словарем, допускающего все слова, в которых встречается слово из конечного множества.

```
FactorAutomaton(префиксное дерево минимальных отсутствующих слов
(Q, A, i, T′, goto′))
 1  for каждого a ∈ A do
 2      if goto′(i, a) определено и не принадлежит T′ then
 3          goto(i, a) ← goto′(i, a)
 4          fail(goto(i, a)) ← i
 5  for each p ∈ Q \ {i} в порядке обхода в ширину и каждого a ∈ A do
 6      if goto′(p, a) определено then
 7          goto(p, a) ← goto′(p, a)
 8          fail(goto(p, a)) ← goto(fail(p), a)
 9      elseif p не принадлежит T′ then
10          goto(p, a) ← goto(fail(p), a)
11  return (Q \ T′, A, i, Q \ T′, goto)
```

На рисунке ниже показан факторный автомат слова ababbba, нарисованный по-другому, чтобы показать связь с изображением префиксного дерева выше.



Примечания

Понятие минимального отсутствующего или запрещенного слова было введено в работе Béal et al. in [28]. О проектировании и анализе описанных алгоритмов см. [92]. Обобщение на регулярные языки появилось в работе [30].

Построение факторного автомата за линейное время описано в работе [67]. Его можно получить путем минимизации DAWG, как описано в работе Blumer et al. [38], или суффиксного автомата (см. [74, 96, 98]).

Второй алгоритм похож на построение автомата для сопоставления с образцом, описанное в работе Aho and Corasick [3]. Будучи применен к префиксному дереву минимальных отсутствующих слов, построенному для нескольких слов, этот метод не всегда дает минимальный факторный автомат.

Антисловари – множества отсутствующих слов – используются в методе сжатия данных в работе [93]; дополнительные сведения см. в работах [198–200] и приведенных в них ссылках. Вычисление в скользящем окне обсуждается в работах [75, 189].

Отсутствующие слова применяются также в биоинформатике для обнаружения устранимых образцов в геномных последовательностях и для построения филогенезов; см., например, работы [51, 224].

61. Жадная суперстрока

Суперстроку множества слов можно использовать для компактного хранения этого множества. Формально общей суперстрокой множества слов X называется слово z, в которое все элементы X входят в качестве факторов, т. е. $X \subseteq Fact(z)$. Кратчайшая общая суперстрока X обозначается $SCS(X)$.

Существует простой жадный алгоритм, который дает довольно хорошее приближение к $SCS(X)$. В этой задаче нашей целью будет эффективная реализация такого метода.

Для двух слов u и v обозначим $Overlap(u, v)$ самый длинный суффикс u, являющийся также префиксом v. Если $w = Overlap(u, v)$, $u = u'w$ и $v = wv'$, то $u \otimes v$ определяется как слово $u'wv'$. Заметим, что $SCS(u, v)$ равна либо $u \otimes v$, либо $v \otimes u$. Также заметим, что принадлежащее X слово, которое является фактором другого слова из X, можно отбросить, и $SCS(X)$ при этом не изменится. Таким образом, можно предполагать, что X не содержит факторов.

```
GreedySCS(непустое, не содержащее факторов множество слов X)
1  if |X| = 1 then
2      return x ∈ X
3  else пусть x, y ∈ X, x ≠ y, для которых слово |Overlap(x, y)| максимально
4      return GreedySCS(X \ {x, y} ∪ {x ⊗ y})
```

Вопрос. Для множества X слов над алфавитом {1, 2, ..., n} показать, как реализовать алгоритм GreedySCS(X), требующий времени $O(\sum\{|x| : x \in X\})$.

Пример. Суперстрока fbdiachbgegeakhiacbd порождается алгоритмом GreedySCS по множеству {egeakh,fbdiac,hbgege,iacbd,bdiach}.

```
                          i a c b d
                  e g e a k h
          h b g e g e
  b d i a c h
f b d i a c
f b d i a c h b g e g e a k h i a c b d
```

Решение

Перекрытие двух слов u и v является границей слова $v\#u$, где символ # не встречается ни в одном из слов. Поэтому методы вычисления границ за линейное время (например, описанный в задаче 19) приводят к реализации, требующей времени $O(n \cdot |X|)$, где $n = \{|x| : x \in X\}$. Мы покажем, как построить реализацию, работающую за время $O(n)$.

Если в примере выше обозначить слова x_1, x_2, x_3, x_4 и x_5, то порождаемая алгоритмом суперстрока равна $x_2 \otimes x_5 \otimes x_3 \otimes x_1 \otimes x_4$. Она идентифицируется перестановкой индексов слов (2,5,3,1,4).

Сначала спроектируем итеративную версию алгоритма, который порождает перестановку индексов слов, ассоциированную с искомой суперстрокой. Эта перестановка реализуется в виде двунаправленного списка, для которого задан начальный индекс, а элементы связаны с помощью таблиц *prev* и *next*. В процессе вычисления для (частичного) списка, начинающегося с индекса p и заканчивающегося индексом q, имеем $head[q] = p$ и $tail[p] = q$.

Ниже приведена схема итеративной версии алгоритма.

```
IterGreedy(непустое, не содержащее факторов множество слов {x_1, x_2, ..., x_m})
 1  for i ← 1 to m do
 2      (prev[i], next[i]) ← (i, i)
 3      (head[i], tail[i]) ← (i, i)
 4  for m − 1 раз do
 5      пусть i, j, next[i] = i, prev[j] = j, head[i] ≠ j
            таковы, что |Overlap(x_i, x_j)| максимально
 6      (next[i], prev[j]) ← (j, i)
 7      head[tail[j]] ← head[i]
 8      tail[head[i]] ← tail[j]
 9  пусть индекс i такой, что prev[i] = i
10  return (i, next)
```

Условие $next[i] = i$ в строке 5 гарантирует, что i – конец своего списка, а условие $prev[j] = j$ – что j – начало своего списка. Условие $head[i] \neq j$ означает, что i и j находятся в разных списках, которые конкатенируются в строке 6. Следующие инструкции обновляют начала и концы списков.

Получив на выходе пару $(i, next)$, мы можем построить перестановку индексов, ассоциированную с суперстрокой слов $\{x_1, x_2, \ldots, x_m\}$ в виде i, $next[i]$, $next[next[i]]$ и т. д. Эффективность алгоритма IterGreedy можно повысить, введя полезные структуры данных *Last* и *First*: для каждого $u \in Pref(\{x_1, \ldots, x_m\})$:

- $Last(u)$ – список индексов слов в $\{x_1, \ldots, x_m\}$, для которых u является суффиксом;
- $First(u)$ – список индексов слов в $\{x_1, \ldots, x_m\}$, для которых u является префиксом.

Кроме того, для каждого индекса слова будем хранить все его вхождения в списки, чтобы иметь возможность удалить его из списков. Положим $n = \sum_{i=1}^{m} |x_i|$.

Наблюдение. Суммарная длина всех списков равна $O(n)$.

Действительно, индекс i встречается в списке $First(u)$ для каждого собственного префикса u слова w_i. Поэтому он встречается в $|w_i|$ списках, что в сумме составляет $O(n)$. То же самое справедливо для *Last*.

Алгоритм IterGreedy переписывается в виде алгоритма EffiGreedy, который обрабатывает все потенциальные перекрытия u в порядке убывания длины и объединяет слова, имеющие такие перекрытия, если они пригодны для объединения. Проверка пригодности выполняется так же, как в алгоритме IterGreedy.

```
EffiGreedy(непустое, не содержащее факторов множество слов {x_1, x_2, ..., x_m})
 1  for i ← 1 to m do
 2      (head[i], tail[i]) ← (i, i)
 3  for каждого u ∈ Pref({x_1, x_2, ..., x_m}) в порядке убывания длины do
 4      for каждого i ∈ Last(u) do
 5          пусть j – первый элемент First(u), для которого j ≠ head[i]
```

```
 6          ▷ это первый или второй элемент либо nil
 7          if j ≠ nil then ▷ u – перекрытие x_i и x_j
 8              удалить j из всех списков First
 9              удалить i из всех списков Last
10              next[i] ← j
11              head[tail[j]] ← head[i]
12              tail[head[i]] ← tail[j]
13      пусть i – индекс слова, для которого i ≠ next[j] для всех j = 1, ..., m
14      return (i, next)
```

Алгоритм EffiGreedy работает за время $O(n)$, если все списки предварительно обработаны, поскольку их суммарный размер равен $O(n)$.

Предобработка *Pref* и других списков производится с помощью префиксного дерева множества $\{x_1, x_2, \ldots, x_m\}$ и суффиксного дерева. Единственная нетривиальная часть – вычисление списков $Last(u)$. Для этого обозначим $\mathcal{T}'$ суффиксное дерево слова $x_1\#_1x_2\#_2\ldots x_m\#_m$, где все $\#_i$ – новые попарно различные символы. Затем для каждого x_i обходим дерево $\mathcal{T}'$ посимвольно вдоль пути, помеченного x_i, и для каждого префикса u слова x_i, если из соответствующего узла $\mathcal{T}'$ исходит k ребер, метки которых начинаются символами $\#_i1, \ldots, \#_ik$ соответственно, вставляем индексы $i_1, \ldots, i_k$ в $Last(u)$. Такая предобработка требует времени $O(n)$, если алфавит допускает линейную сортировку.

Примечания

Известно, что задача вычисления кратчайшей общей суперстроки NP-полная. Наш вариант алгоритма GreedySCS построен на основе алгоритма из работы Tarhio and Ukkonen [230].

Одна из самых интересных гипотез, относящихся к этой теме, – порождает ли GreedySCS 2-аппроксимацию кратчайшей общей суперстроки входного множества. Это так для слов длины 3 и, скорее всего, верно в общем случае.

62. Кратчайшая общая суперстрока коротких слов

Общей суперстрокой множества слов X называется слово, в котором все элементы X входят в качестве факторов. Вычисление кратчайшей общей суперстроки (SCS) – NP-полная проблема, но в некоторых простых случаях ее можно решить эффективно, например в частном случае, обсуждаемом в этой задаче.

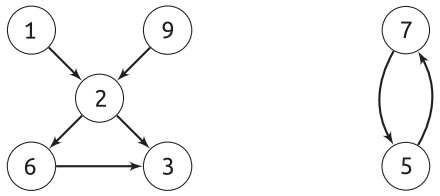
Так, `1263923757` – кратчайшая общая суперстрока множества из семи слов {`12`,`23`,`26`,`57`,`63`,`75`,`92`}.

Вопрос. Показать, как за линейное время вычислить длину кратчайшей общей суперстроки множества X слов длины 2 над алфавитом целых чисел.

[**Указание:** преобразуйте вопрос в задачу на графах.]

Решение

Задача следующим образом преобразуется в задачу на ориентированных графах. Для множества X рассмотрим граф G, вершинами которого являются буквы (целые числа), а ребра соответствуют словам из X. На рисунке ниже приведен граф, соответствующий приведенному ранее примеру.



i	1	2	3	5	6	7	9
$outdegree(i)$	1	2	0	1	1	1	1
$indegree(i)$	0	2	2	1	1	1	0

Ориентированный граф называется слабо связным, если любые две его вершины соединены неориентированным путем без учета ориентации ребер. Следующее наблюдение пролагает дорогу к построению кратчайшей суперстроки.

Наблюдение. Пусть G – слабо связный граф, ассоциированный с множеством слов длины 2, а Z – наименьшее множество ориентированных ребер, которые нужно добавить в G, чтобы в нем появился эйлеров цикл. Тогда длина кратчайшей суперстроки для X равна $|X| + 1$, если Z пусто, и $|X| + |Z|$ в противном случае.

Задача сводится к вопросу о построении множества Z, необходимого для получения эйлерова графа. Напомним, что ориентированный граф является эйлеровым, если он слабо связный и каждая вершина сбалансирована, т. е. $indegree(v) = outdegree(v)$. Каждая компонента слабой связности графа G обрабатывается отдельно. Поэтому для начала мы можем предположить, что сам G слабо связный.

Для добавления минимального числа ориентированных ребер, необходимого, чтобы сделать каждую вершину v сбалансированной, поступим следующим образом. Если $D(v) = outdegree(v) - indegree(v) > 0$, то добавляем $D(v)$ ребер, входящих в вершину v; если $D(v) < 0$, то добавляем $D(v)$ ребер, исходящих из v. Идея в том, чтобы добавляемое ребро всегда исходило из вершины, которая нуждается в исходящем ребре, и входило в вершину, нуждающуюся во входящем ребре. Поскольку каждое ребро одновременно является входящим и исходящим, не может остаться только одна вершина v, для которой $D(v) \neq 0$, и процесс продолжается, пока все вершины не окажутся сбалансированными. Отсюда следует также, что общее число добавленных ребер равно $|Z| = \frac{1}{2}\sum_v |D(v)|$.

Вычислить Z можно за линейное время, воспользовавшись таблицами *outdegree* и *indegree*. Худшим является случай, когда никакие два слова в X не перекрываются. Когда в преобразованном графе появится эйлеров цикл,

удаление одного из добавленных ребер $v \to w$ порождает эйлеров путь из w в v. Если исходный граф уже эйлеров, то путь, начинающийся из любой вершины и заканчивающийся в ней же, дает решение. Пути соответствуют кратчайшим суперстрокам.

Наконец, если граф G не слабо связный, то каждая компонента слабой связности обрабатывается, как описано выше, и получившиеся слова конкатенируются – результатом является кратчайшая суперстрока.

В примере выше в левую компоненту добавляются только два ребра, $3 \to 1$ и $3 \to 9$, что дает новые таблицы:

i	1	2	3	5	6	7	9
$outdegree(i)$	1	2	2	1	1	1	1
$indegree(i)$	1	2	2	1	1	1	1

Удаление первого добавленного ребра дает слово 1263923 для первой компоненты и слово 757 для второй. В результате получается суперстрока 1263923757.

Примечания

Описанный выше метод взят из работы Gallant et al. [126]. Если входное множество состоит из слов длины 3, то задача становится NP-полной.

63. Подсчет факторов по длине

Обозначим $fact_x[l]$ количество различных факторов длины l, встречающихся в слове x.

Вопрос. Показать, как за линейное время вычислить все числа $fact_x[l]$, $l = 1, \ldots, |x|$, связанные со словом x, в предположении, что размер алфавита постоянный.

[**Указание:** воспользуйтесь суффиксным деревом слова.]

Решение

Пусть $T = \mathcal{ST}(x)$ – суффиксное дерево x. Напомним, что его внутренние узлы являются факторами x, которые входят в x как минимум два раза. За ними следуют либо две разные буквы, либо одна буква, если вхождение является суффиксом.

С некорневым узлом v дерева T, родителем которого является узел u, ассоциирован интервал длин

$$I_v = [|u| + 1..|v|].$$

Наблюдение. $fact_x[l]$ равно количеству интервалов I_v, содержащих число l.

Действительно, каждый некорневой узел v суффиксного дерева соответствует набору факторов, разделяющих общее множество вхождений. Их длины различны и образуют интервал I_v. Поэтому общее число различных факторов длины l равно числу всех интервалов I_v, содержащих l.

Это наблюдение сводит нашу задачу к *задаче о покрытии интервалами*: для заданного семейства $I(x)$ подынтервалов $[1..|x|]$ вычислить количество интервалов, содержащих l, для всех $1 \le l \le |x|$.

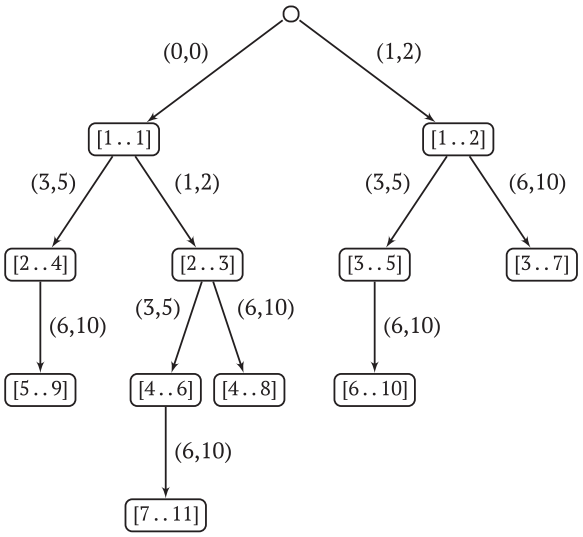
```
NumbersOfFactors(I(x) для непустого слова x)
 1  Count[1..|x| + 1] ← [0, 0, ..., 0]
 2  for каждого [i..j] ∈ I(x) do
 3      Count[i] ← Count[i] + 1
 4      Count[j + 1] ← Count[j + 1] − 1
 5  prefix_sum ← 0
 6  for l ← 1 to n do
 7      prefix_sum ← prefix_sum + Count[l]
 8      fact_x[l] ← prefix_sum
 9  return fact_x
```

Алгоритм NumbersOfFactors вычисляет $fact_x$ по семейству $I(x)$ интервалов, определенных по суффиксному дереву x. Для этого используется вспомогательный массив $Count[1..n + 1]$, инициализированный нулями.

Пример. Пусть $x =$ abaababaaba. Интервалы в $I(x)$ – это метки узлов суффиксного дерева x, показанного на рисунке ниже. Алгоритм вычисляет таблицу $Count = [2, 1, 1, 1, 0, 0, 0, -1, -1, -1, -1]$ (отбрасывая значение в позиции $|x| + 1$) и возвращает последовательность сумм префиксов этой таблицы:

$fact_x = [2,3,4,5,5,5,5,4,3,2,1]$.

Например, значение $fact_x[3] = 4$ соответствует всем четырем факторам длины 3, встречающимся в x: aab, aba, baa и bab.



Понятно, что алгоритм NumbersOfFactors требует линейного времени, в основном потому, что количество узлов в суффиксном дереве x имеет порядок $O(|x|)$.

Примечания
Альтернативный алгоритм можно получить, воспользовавшись факторным автоматом x. В этом автомате каждое состояние v, кроме начального, помечено интервалом $[s(v)..l(v)]$, где $s(v)$ и $l(v)$ – длины соответственно самого короткого и самого длинного путей из корня в v.

64. Подсчет факторов, покрывающих позицию

Говорят, что фактор слова x покрывает позицию k этого слова, если существует его вхождение $x[i..j]$ такое, что $i \le k \le j$.

Обозначим $\mathcal{C}(x, k)$ количество различных факторов x, покрывающих позицию k, а $\mathcal{N}(x, k)$ – количество факторов, вхождение которых не покрывает k.

Вопрос. Показать, как за линейное время вычислить $\mathcal{C}(x, k)$ и $\mathcal{N}(x, \mathrm{k})$ для заданной позиции k в слове x в предположении, что размер алфавита постоянный.

Решение
Узлами суффиксного дерева $\mathcal{ST}(w)$ слова w являются факторы w. Для ребра (u, v) этого дерева положим $weight(v) = |v| - |u|$, т. е. длине его метки.

Вычисление $\mathcal{N}(x, k)$. Пусть # – буква, не встречающаяся в x, а x' – слово, полученное из x заменой буквы $x[k]$ на #. Тогда, воспользовавшись суффиксным деревом $\mathcal{ST}(x')$, мы сможем вычислить количество N различных факторов x' как сумму весов всех некорневых узлов.

Наблюдение. $\mathcal{N}(x, k) = N - M$, где M – количество различных факторов x', содержащих букву #.

Это прямой путь к вычислению $\mathcal{N}(x, k)$, т. к. $M = (k + 1) \times (n - k)$.

Вычисление $\mathcal{C}(x, k)$. Предположим, что x заканчивается специальным маркером конца и каждый лист $\mathcal{ST}(x)$ помечен начальной позицией соответствующего суффикса x. Для каждого узла v обозначим $LeftLeaves(v, k)$ множество листьев i в поддереве с корнем v таких, что $i \le k$ и $k - i < |v|$.

Пусть V – множество узлов v, для которых множество $LeftLeaves(v, k)$ непусто. Иными словами, V является множеством узлов, соответствующих факторам, покрывающим позицию k. Для $v \in V$ обозначим $Dist(v, k) = \min\{k - i : i \in LeftLeaves(v, k)\}$.

Наблюдение. $\mathcal{C}(x, k) = \sum_{v \in V} \min\{|v| - Dist(v, k), weight(v)\}$.

Вычисление $\mathcal{C}(x, k)$ сводится к вычислению всех $Dist(v, k)$, для чего нужно обойти суффиксное дерево снизу вверх.

Для алфавита постоянного размера все вычисления требуют линейного времени.

Примечания
Интересные варианты этой задачи возникают, когда нужно покрыть факторами все позиции из заданного множества. Аттрактор, понятие, введенное в работе Prezza [202] (см. также [157, 183]), – это множество K позиций в слове x, факторы которого имеют хотя бы одно вхождение, покрывающее элемент K. Аттракторы представляют удобный инструмент для анализа словарных алгоритмов сжатия текста и используются в работе [193] для разработки сжатых автоиндексов.

65. Наибольшие факторы с одинаковой четностью

Для слова $v \in \{0,1\}^+$ обозначим $parity(v)$ сумму по модулю 2 единиц, встречающихся в v. Для двух слов $x, y \in \{0,1\}^+$ рассмотрим их факторы u и v одинаковой длины такие, что $parity(u) = parity(v)$. Назовем такой фактор максимальной длины ***наибольшим фактором с одинаковой четностью*** (longest common-parity factor) и обозначим ее $lcpf(x, y)$. Удивительно, но задача нахождения его длины сводится к вычислению всех периодов слов.

Вопрос. Показать, как за линейное время вычислить длины $lcpf(x, y)$ для двух двоичных слов x и y.

Решение
В решении используется структура данных, называемая *таблицей четностей*. Для слова x элемент $Parity[l, x]$ – это множество различных четностей факторов x длины l. Если два фактора длины l имеют разные четности, то $Parity[l, x] = \{0,1\}$.

Наблюдение. Длину $lcpf(x, y)$ можно получить из таблиц четностей x и y: $lcpf(x, y) = \max\{l : Parity[l, x] \cap Parity[l, y] \neq \emptyset\}$.

Быстрое вычисление таблицы *Parity*. Задача сводится к вычислению таблицы *Parity*, а оно, в свою очередь, опирается на следующий простой факт.

Наблюдение. $Parity[l, x] \neq \{0,1\}$ тогда и только тогда, когда l – период x.

Действительно, если l – период x, то, очевидно, четность всех факторов длины l одинакова. Обратно, предположим, что все факторы длины l имеют одинаковую четность. Тогда имеет место равенство $\sum_{j=i}^{i+l-1} x[j] \pmod 2 = \sum_{j=i+1}^{i+1} x[j] \pmod 2$, откуда следует, что $x[i + l] = x[i]$, что и требовалось доказать.

Все периоды x вычисляются, например, как побочный продукт вычисления таблицы границ (см. задачу 19). Далее, если l – период x, то $Parity[l, x]$ – четность префикса x длины l, что вытекает из вычисления суммы префиксов для всех значений l. Если l не является периодом x, то, в силу наблюдения, $Parity[l, x] = \{0,1\}$. Таким образом, таблицы четностей для x и y вычисляются за линейное время, а стало быть, мы можем вычислить $lcpf(x, y)$ за линейное время.

Примечания

Эта задача обобщается на слова над большим алфавитом $\{0, 1, \ldots, k-1\}$, нужно только рассматривать суммы по модулю k. Для решения применяется аналогичный алгоритм.

66. Установление свободы слова от квадратов с помощью словаря базовых факторов

Словарь базовых факторов (dictionary of Basic Factors – DBF) – элементарная структура данных, полезная для создания эффективных алгоритмов решения многих задач, связанных со словами. В данном случае мы воспользуемся ей, чтобы проверить, что слово свободно от квадратов.

DBF слова w состоит из логарифмического количества таблиц $Name_k$, $0 \le k \le \log|w|$. Таблицы индексируются позициями слова w, предназначение $Name_k[j]$ – идентифицировать $w[j..j + 2^k - 1]$, фактор длины 2^k, начинающийся в позиции j слова w. Идентификаторы обладают следующим свойством: для $i \ne j$ $Name_k[i] = Name_k[j]$ тогда и только тогда, когда

$$i + 2^k - 1, j + 2^k - 1 < |w| \text{ и } w[i..i + 2^k - 1] = w[j..j + 2^k - 1].$$

Известно, что DBF слова w можно вычислить за время $O(|w| \log |w|)$.

Чтобы проверить, свободно ли w от квадратов, обозначим $Pred_k$ ($0 \le k < \log|w|$), таблицу, индексируемую позициями w и определенную следующим образом:

$$Pred_k[j] = \max\{i < j : Name_k[i] = Name_k[j]\} \cup \{-1\},$$

а $Cand_w$ – множество пар позиций $(i, 2j - i)$ слова w, являющихся кандидатами на вхождение квадрата $w[i..2j - i - 1]$ в w:

$$Cand_w = \{(i, 2j - i) : 2j - i \le |w| \text{ и } i = Pred_k[j] \ne -1 \text{ для некоторого } k\}.$$

Вопрос. Показать, что слово w содержит квадрат, если $w[p..q - 1]$ является квадратом для некоторой пары $(p, q) \in Cand_w$. Разработать алгоритм, который проверяет, свободно ли слово w от квадратов за время $O(|w| \log |w|)$.

Пример. Ниже показаны таблицы *Pred* для слова $w =$ `abacbaca`:

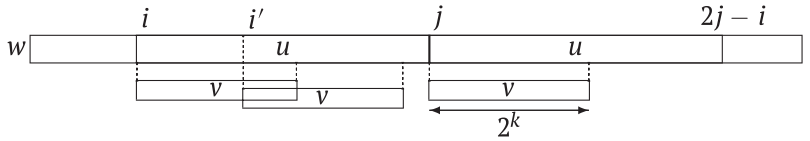
j	0	1	2	3	4	5	6	7
$x[j]$	a	b	a	c	b	a	c	a
$Pred_0[j]$	-1	-1	0	-1	1	2	3	5
$Pred_1[j]$	-1	-1	-1	-1	1	2	-1	
$Pred_2[j]$	-1	-1	-1	-1	-1			

С ними ассоциировано множество $Cand_w$ {(0,4),(1,7),(2,8)}. Только пара (1,7) соответствует квадрату, а именно $w[1..6] =$ bacbac.

Решение

Для ответа на первую часть вопроса обозначим i начальную позицию вхождения самого короткого квадрата, встречающегося в w. Пусть его длина равна $2l$ и $j = i + l$. Квадратный фактор равен $w[i..i + 2l - 1]$ и $u = w[i..i + l - 1] = w[j..j + l - 1]$. Покажем, что пара $(i, i + 2l)$ принадлежит $Cand_w$, т. е. $i = Pred_k[j]$ для некоторого целого k.

Пусть k – наибольшее целое число такое, что $2^k \le l$. Имеем $v = w[i..i + 2^k - 1] = w[j..j + 2^k - 1]$ (поскольку v – префикс u), т. е. $Name_k[i] = Name_k[j]$.



Предположим противное: что $i < Pred_k[j]$, т. е. существует вхождение v, начинающееся в позиции $Pred_k[j]$ (i' на рисунке). Это вхождение отличается от вхождений v префиксов u (см. рисунок) и перекрывается по крайней мере с одним из них в силу своей длины. Но тогда мы получаем более короткий квадрат, что противоречит предположению. Таким образом, $Pred_k[j] = i$, а это означает, что $(i, i + 2l) \in Cand^w$, что и требовалось доказать.

Алгоритм SquareFree применяет доказанное выше свойство, осуществляя поиск $Cand_w$ для пары позиций, соответствующих квадрату.

```
SquareFree(непустое слово w длины n, DBF слова w)
 1  for k ← 0 to ⌊log n⌋ do
 2      вычислить Pred_k, зная DBF слова w
 3  вычислить Cand_w по таблицам Pred_k
 4  for каждой пары (p, q) ∈ Cand_w do
 5      k ← ⌊log(q − p)/2⌋
 6      if Name_k[p] = Name_k[(p + q)/2] и
            Name_k[(p + q)/2 − 2^k] = Name^k[q − 2^k] then
 7          return FALSE
 8  return TRUE
```

Правильность алгоритма SquareFree. Правильность алгоритма вытекает из приведенного выше доказательства, обосновывающего выбор k в строке 5.

Таким образом, проверка того, что пара (p, q) соответствует квадрату, т. е. что факторы $w[p..(p+q)/2-1]$ и $w[(p+q)/2..q-1]$ равны, сводится к проверке того, что их префиксы длины 2^k совпадают и что их суффиксы длины 2^k тоже совпадают. Именно это мы и делаем в строке 6.

Время работы SquareFree. Для данного k таблицу $Pred_k$ можно вычислить за линейное время, просмотрев таблицу $Name_k$ слева направо. Вычисление множества $Cand_w$ путем обхода $\lfloor \log |x| \rfloor$ таблиц *Pred* занимает время $O(|x| \log |x|)$.

Такая же оценка сверху имеет место для строк 5–7 благодаря таблицам *Name* из структуры DBF. Следовательно, SquareFree требует времени $O(|x| \log |x|)$.

Примечания

Существует много алгоритмов, проверяющих свободу слова от квадратов, с такой же оценкой времени работы. Но этот особенно прост в случае, когда доступна структура DBF. Это вариант алгоритма, опубликованного в работе [84].

67. Общие решения факторных уравнений

В данной задаче рассматривается алгоритм построения слов по факторным уравнениям. Факторное уравнение имеет вид $w[p..q] = w[p'..q']$ и длину $q - p + 1$. Коротко уравнение записывается в виде тройки $(p, p', q - p + 1)$.

Говорят, что слово w длины n является решением системы факторных уравнений E, если оно удовлетворяет каждому уравнению. Нас будут интересовать общие решения, содержащие наибольшее число различных букв. Такое решение длины n можно использовать для описания всех остальных решений системы. Оно обозначается $\Psi(E, n)$ и единственно с точностью до переименования букв.

Пример. Для системы уравнений

$$E = \{(2,3,1), (0,3,3), (3,5,3)\}$$

общим решением $\Psi(E, 8)$ является слово $w = \texttt{abaababa}$. Действительно, $w[2] = w[3] = \texttt{a}$, $w[0..2] = w[3..5] = \texttt{aba}$ и $w[3..5] = w[5..7] = \texttt{aba}$. Иными словами, имеет место отношение эквивалентности на множестве позиций w, состоящее из двух классов эквивалентности $\{0,2,3,5,7\}$ и $\{1,4,6\}$. Это самое мелкое отношение эквивалентности, удовлетворяющее уравнениям системы E. Заметим, что $\Psi(E,11) = \texttt{abaababacde}$.

Вопрос. Описать, как построить общее решение $\Psi(E, n)$ для данной системы факторных уравнений E за время $O((n+m)\log n)$, где $m = |E|$.

[**Указание:** воспользуйтесь покрывающими лесами для представления множеств эквивалентных позиций.]

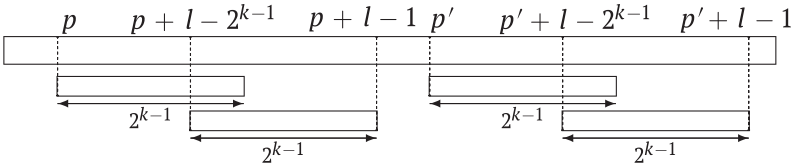
Решение

Для $k \geq 0$ обозначим E_k подмножество уравнений системы E длины l такой, что $2^{k-1} < l \leq 2^k$. В частности, E_0 – подмножество уравнений длины 1.

Операция REDUCE**.** Пусть $k > 0$. Для множества X уравнений длины l, $2^{k-1} < l \leq 2^k$, операция REDUCE(X) порождает эквивалентное множество уравнений более короткой длины следующим образом.

- **Расщепление**. Каждое уравнение (p, p', l) в X заменяется двумя уравнениями

  $(p, p', 2^{k-1})$ и $(p + l - 2^{k-1}, p' + l - 2^{k-1}, 2^{k-1})$.



  После этой операции X преобразуется в эквивалентную систему уравнений одинаковой длины размера $O(n + m)$.
- Затем мы создаем граф G, вершинами которого являются начальные позиции уравнений одинаковой длины 2^{k-1}, а ребра соответствуют уравнениям. Если в G существует цикл, то мы можем удалить одно из его ребер вместе с соответствующим уравнением, не изменяя классов эквивалентности.
- **Сжатие**. Для каждой компоненты связности G строится покрывающее дерево. Деревья образуют лес, покрывающий весь граф. В конечном итоге REDUCE(X) является системой уравнений, соответствующих ребрам покрывающего леса.

Ключевое наблюдение. Поскольку существует $O(n)$ ребер в покрывающем лесе, то размер этой системы уравнений |REDUCE(X)| имеет порядок $O(n)$.

Главный алгоритм. Алгоритм в целом состоит из логарифмического количества итераций, на которых выполняется операция REDUCE. После каждой итерации получающаяся эквивалентная система содержит уравнения гораздо меньшей длины.

В итоге мы получаем систему E_0 уравнений длины 1, из которой $\Psi(E_0, n) = \Psi(E, n)$ легко вычисляется за линейное время.

```
PSI(система уравнений E, положительная длина n)
1  ▷ E = ⋃_{i=0}^{⌈log n⌉} E_i
2  for k ← ⌈log n⌉ downto 1 do
3      E_{k−1} ← E_{k−1} ∪ REDUCE(E_k)
4      ▷ инвариант: система ⋃_{i=0}^{k−1} E_i эквивалентна E
5  return Ψ(E_0, n)
```

Последняя система E_0 включает только одиночные позиции и дает классы эквивалентности позиций. Всем позициям в одном классе эквивалентности сопоставляется одна и та же буква, уникальная для каждого класса. Получившееся слово и есть искомое решение $\Psi(E, n)$. Так как операция REDUCE требует времени $O(n + m)$, то весь алгоритм работает за время $O((n + m)\log n)$, что и требовалось доказать.

Примечания

Описанный алгоритм – вариант алгоритма из работы Gawrychowski et al. [127]. На самом деле в работе [127] алгоритм преобразован в алгоритм с линейным временем работы, но с применением хитроумных структур данных. Его можно использовать для построения слова, имеющего заданное множество серий, если таковое существует.

68. Поиск в бесконечном слове

Наша цель – спроектировать алгоритм для сопоставления с образцом в бесконечном слове. Поскольку для произвольных бесконечных слов задача неразрешима, мы ограничимся чистым морфическим словом. Так называется бесконечное слово, полученное повторным применением морфизма θ из A^+ в себя, где $A = \{\mathtt{a},\mathtt{b}, \ldots\}$ – конечный алфавит. Для этого предположим, что θ продолжаем над буквой $\mathtt{a}$, т. е. $\theta(\mathtt{a}) = \mathtt{a}u$, где $u \in A^+$. Тогда $\Theta = \theta^\infty(\mathtt{a})$ существует и равно $\mathtt{a}u\theta(u)\theta^2(u)\ldots$. Бесконечное слово Θ является неподвижной точкой θ, т. е. $\theta(\Theta) = \Theta$.

Чтобы не рассматривать тривиальные случаи, например когда морфизм η определен как $\eta(\mathtt{a}) = \mathtt{ab}$, $\eta(\mathtt{b}) = \mathtt{c}$, $\eta(\mathtt{c}) = \mathtt{b}$ и $\eta(\mathtt{d}) = \mathtt{d}$, так что буква $\mathtt{d}$ бесполезна, а буква $\mathtt{a}$ встречается в Θ только один раз, будем предполагать, что θ неприводим. Это означает, что любая буква достижима из любой буквы: для любых c, $d \in A$ буква d встречается в $\theta^k(c)$ для некоторого целого k.

Морфизм Туэ–Морса μ и морфизм Фибоначчи φ (см. главу 1) – примеры неприводимых морфизмов.

Вопрос. Показать, как проверить, является ли морфизм неприводимым.

Вопрос. Спроектировать алгоритм, который вычисляет множество факторов длины m, встречающихся в бесконечном слове $\Theta = \theta^\infty(\mathtt{a})$, где θ – неприводимый морфизм.

Если множество факторов длины m слова Θ представлено (детерминированным) префиксным деревом, то проверка, встречается ли образец длины m в Θ, сводится к простому обходу этого дерева сверху вниз.

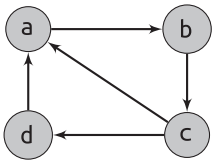
Решение

Чтобы проверить неприводимость морфизма θ, мы построим его граф достижимости букв. Вершинами графа являются буквы, а для двух разных букв

c и d существует дуга, ведущая из c в d, если d встречается в $\theta(c)$. Неприводимость имеет место, если граф содержит цикл, проходящий через все буквы алфавита; это можно проверить за полиномиальное время.

Например, граф морфизма ζ удовлетворяет этому свойству:

$$\begin{cases} \zeta(\mathtt{a}) = \mathtt{ab} \\ \zeta(\mathtt{b}) = \mathtt{c} \\ \zeta(\mathtt{c}) = \mathtt{cad} \\ \zeta(\mathtt{d}) = \mathtt{a} \end{cases}$$



Для ответа на второй вопрос можно выделить факторы длины m из слов $\theta^k(\mathtt{a})$ путем итеративного применения морфизма, начиная с a. Действительно, нетрудно понять, что после конечного числа итераций будут выделены все факторы длины m.

Вместо этого описанный ниже алгоритм обрабатывает только слова, являющиеся образами факторов Θ длины не больше m относительно морфизма θ. Его правильность – следствие неприводимости морфизма, потому из нее вытекает, что любой фактор $\theta^k(\mathtt{a})$ является фактором $\theta^l(b)$ для любой буквы и b и некоторого целого l.

Искомое множество факторов длины m слова Θ является множеством слов длины m, хранящихся в префиксном дереве T, порождаемом алгоритмом.

```
FACTORS(неприводимый морфизм Θ, A ∈ A, положительное целое m)
 1  инициализировать T пустым префиксным деревом
 2  Queue ← A
 3  while Queue не пуста do
 4      v ← выбрать первое слово из Queue
 5      w ← θ(v)
 6      for каждого фактора z слова w длины m do
 7          if z отсутствует в T then
 8              вставить z и T добавить z в Queue
 9      if |w| < m и w отсутствует в T then
10          вставить w в T добавить w to Queue
11  return T
```

В зависимости от свойств морфизма алгоритм можно настроить так, чтобы он работал быстрее. Например, так обстоит дело, если морфизм k-равномерный: $|\theta(c)| = k$ для любой буквы c. Тогда помещать в очередь нужно только факторы длины $\lfloor m/k \rfloor + 1$, что значительно уменьшает количество слов в очереди.

Вставку в префиксное дерево в строке 8 можно реализовать аккуратно, избежав бесполезных операций. На самом деле после вставки фактора $z = cy$ (для некоторой буквы c) естественно продолжить со следующего фактора вида yd (для некоторой буквы d). Если дерево оснащено суффиксными ссыл-

ками (такими же, как в суффиксном дереве), то эта операция занимает постоянное время (или не более $\log|A|$). Тогда вставка всех факторов z слова w занимает время $O(|w|)$ (или $O(|w|\log|A|)$).

Примечания

Более сильное предположение о морфизме – примитивность; это означает, что существует целое число k, для которого буква d встречается в $\theta^k(c)$ для любых $c, d \in A$ (k не зависит от этой пары букв). Для примитивных морфизмов задачу можно решить по-другому, а именно рассмотреть возвратные слова в бесконечном слове x: возвратным словом к фактору w слова x называется кратчайшее (непустое) слово r, для которого rw имеет границу w и является фактором x. В работе Durand and Leroy [104] доказано, что для примитивного морфизма θ существует такая постоянная K, что $|r| \le K|w|$ и все факторы длины m слова Θ встречаются в факторах длины $(K+1)m$. Более того, авторам удалось ограничить постоянную K величиной $\max\{\theta(c) : c \in A\}^{4|A|^2}$. Это позволяет построить другой алгоритм нахождения множества факторов Θ длины m.

69. Совершенные слова

Слово некоторой длины называется *плотным*, если оно имеет наибольшее число (различных) факторов среди всех слов той же длины над тем же алфавитом. Говорят, что слово *совершенное*, если все его префиксы плотные. Заметим, что любой префикс совершенного слова сам является совершенным словом.

Пример. Слово `0110` плотное, а слово `0101` – нет. Самые длинные двоичные совершенные слова – `011001010` и его дополнение `100110101`, их длина равна 9. Но над троичным алфавитом слово `0120022110` длины 10 является совершенным.

Множество двоичных совершенных слов конечно, но при переходе к большим алфавитам ситуация кардинально меняется.

Вопрос. Показать, как за линейное время построить троичное совершенное слово любой заданной длины. Доказать, что существует бесконечное совершенное троичное слово.

[**Указание:** рассмотрите гамильтонов и эйлеров циклы в автомате де Брёйна.]

Решение

Пусть $A = \{0,1,2\}$ – алфавит, и рассмотрим длины $\Delta_n = 3^n + n - 1$ слова де Брёйна порядка n над A. Достаточно показать, как построить совершенные слова с этими длинами, потому что их префиксы совершенны.

Любое совершенное троичное слово длины Δ_n является словом де Брёйна. Поэтому задача сводится к построению совершенных слов де Брёйна.

Нашей базовой структурой данных является граф де Брёйна G_n порядка n (графовая структура автомата де Брёйна) над алфавитом A. Вершинами G_n

являются троичные слова длины $n - 1$. Меткой эйлерова цикла является круговое слово де Брёйна порядка n, которое порождает (линейное) слово де Брёйна того же порядка после добавления в конец своего префикса длины $n - 1$.

Наша первая цель – расширить слово де Брёйна w порядка n до слова де Брёйна порядка $n + 1$. Пусть u – граница w длины $n - 1$, а ua – его префикс длины n. Обозначим $\widehat{w} = wa$.

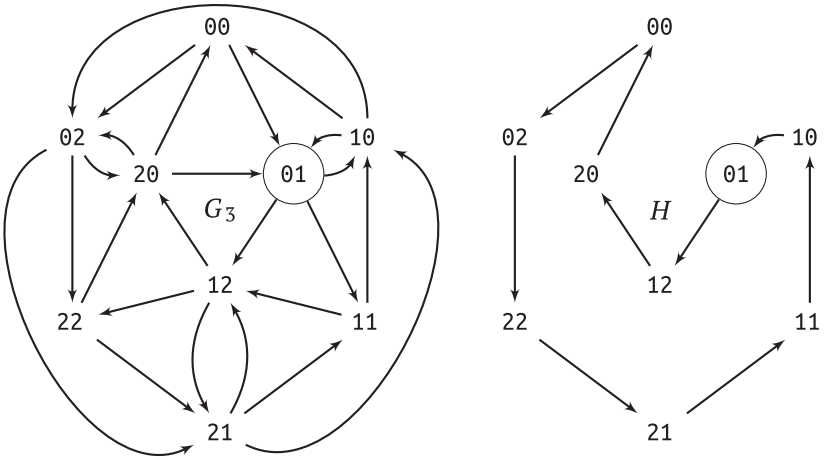
Наблюдение. В графе G_{n+1}, вершинами которого являются слова A_n, $\widehat{w}$ – метка гамильтонова цикла, обозначаемого $Cycle_n(\widehat{w})$, начинающегося и заканчивающегося в вершине ua – префиксе w длины n.

Пример. Слово $w =$ `0122002110` является словом де Брёйна порядка 2 над A. Оно ассоциировано с эйлеровым циклом в G_2:

$$Cycle_1(w) = 0 \to 1 \to 2 \to 2 \to 0 \to 0 \to 2 \to 1 \to 1 \to 0.$$

Поэтому слово $\widehat{w} =$ `01220021101` соответствует гамильтонову циклу H в графе $G3$ (см. рисунок, на котором для ясности опущены петли в вершинах `00`, `11` и `22`):

$$Cycle_2(\widehat{w}) = 01 \to 12 \to 22 \to 20 \to 00 \to 02 \to 21 \to 11 \to 10 \to 01.$$



Согласно приведенному выше наблюдению, цикл конкатенируется с дизъюнктным циклом для создания эйлерова цикла в графе G_{n+1}, что дает слово де Брёйна порядка $n + 1$ с префиксом w.

Следующая цель – расширить совершенное слово де Брёйна порядка n до совершенного слова де Брёйна порядка $n + 1$. Таким образом, построим последовательность совершенных слов де Брёйна w_1, w_2, ... такую, что w_i является префиксом w_{i+1}. Ее предел – бесконечное совершенное слово, что нам и нужно.

Обозначим $EulerExt_n(h)$ эйлеров цикл в графе G_n, расширяющий гамильтонов цикл h в G_n, если это возможно. Пусть $Word_n(e)$ – слово, ассоциированное с эйлеровым циклом e в G_n.

```
PerfectWord(положительная длина N, алфавит {0,1,2})
 1  (w, n) ← (012,1)
 2  while |w| < N do
 3      n ← n + 1
 4      h ← Cycle_n(ŵ)
 5      e ← EulerExt_n(h)
 6      w ← Word_n(e)
 7  return префикс w длины N
```

Неформальное объяснение построения. Слово w_n после расширения его на одну букву до $\widehat{w_n}$ соответствует гамильтонову циклу $h = Cycle_n(\widehat{w_n})$ в графе G_{n+1}. Мы расширяем его до эйлерова цикла e в G_{n+1} и, наконец, определяем w_{n+1} как представление e в виде слова. В этом построении интересно, что мы обращаемся с циклами как со словами, а со словами как с циклами и что на главном шаге для вычисления эйлерова расширения используем теоретико-графовые средства, а не стрингологические рассуждения.

Пример. Для совершенного слова $w_2 = 0120022110$ длины 10 имеем $\widehat{w}_2 = 01200221101$, что соответствует циклу H в G_3 (см. рисунок выше):

$$01 \to 12 \to 20 \to 00 \to 02 \to 22 \to 21 \to 11 \to 10 \to 01.$$

H расширяется до эйлерова цикла E путем конкатенации со следующим эйлеровым циклом в $G_3 - H$:

$$01 \to 11 \to 11 \to 12 \to 21 \to 12 \to 22 \to 22 \to 20 \to 02$$
$$\to 21 \to 10 \to 02 \to 20 \to 01 \to 10 \to 00 \to 00 \to 01.$$

Наконец, из E получаем:

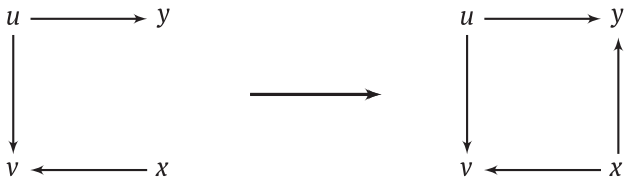
$$w_3 = Word(E) = 01200221101\ 112122202102010001.$$

Прежде чем показать, что слово, порождаемое этим алгоритмом, совершенное, нужно убедиться, что можно получить эйлеров цикл в графе $G_n - H$.

Лемма 5. *Если H – гамильтонов цикл в G_n, то после удаления ребер H граф G_n остается эйлеровым.*

Доказательство. Мы воспользуемся следующим очевидным, но полезным свойством графов де Брёйна, схематически показанным на рисунке ниже: специальная конфигурация трех ребер влечет существование четвертого ребра. Формально:

если $u \to v$, $u \to y$, $x \to v$ – ребра G_n, то $x \to y$ – тоже ребро. (∗)



Мы должны показать, что $G_n - H$ эйлеров. Очевидно, что для каждой вершины $G_n - H$ полустепень захода и полустепень исхода одинаковы. Поэтому достаточно показать, что граф сильно связный. Однако довольно и слабой связности (без учета ориентации ребер), в силу хорошо известного свойства (граф называется регулярным, если степени всех его вершин равны):

Свойство. Регулярный слабо связный ориентированный граф является сильно связным.

Поэтому достаточно показать, что для любого ребра $u \to v \in H$ вершины u и v слабо связаны в $G_n - H$ (т. е. между ними существует неориентированный путь, не проходящий по ребрам H). Действительно, поскольку полустепени захода и полустепени исхода всех вершин равны 3, существуют вершины x, x', y, для которых ребра

$$u \to y,\ x \to v,\ x' \to v$$

принадлежат $G_n - H$. Теперь из свойства (∗) вытекает существование в G_n двух дополнительных ребер $x \to y$ и $x' \to y$ (см. рисунок), и по крайней мере одно из них не принадлежит H. Игнорируя ориентацию этих ребер, мы приходим к выводу о существовании неориентированного пути из u в v, не проходящего по ребрам H.

Следовательно, граф $G_n - H$ слабо связный и эйлеров (если рассматривать его как ориентированный граф), что и завершает доказательство.

Правильность алгоритма PerfectWord. Обозначим w_n значение w непосредственно перед выполнением строки 3. По предположению индукции, все префиксы длины не более $|w_{n-1}|$ слова w_n плотные, поскольку w_{n-1} совершенное. Более длинный префикс w_n содержит все слова длины $n - 1$ и не содержит повторений слов длины n, потому что это префикс слова де Брёйна. Следовательно, он тоже плотный. Итак, каждый префикс w_n плотный, поэтому w_n совершенное.

Сложность. Алгоритм работает за линейное время, потому что эйлеровы циклы можно найти за линейное время, а в графах де Брёйна нахождение гамильтонова цикла сводится к вычислению эйлерова цикла.

Примечания

Совершенные слова называют также *суперсложными*, а их построение описано в работе [237]. В случае двоичных слов понятие совершенного слова ослабляется до полусовершенного, существование таких слов доказано в работе [206].

70. Плотные двоичные слова

Слово называется *плотным*, если оно имеет наибольшее число (различных) факторов среди всех слов той же длины над тем же алфавитом.

Над алфавитом, содержащим по крайней мере три буквы, задача генерирования плотных слов любой заданной длины решается путем генерирования совершенных слов (см. задачу 69). Но это решение неприменимо к двоичным словам, и в настоящей задаче показано, как эффективно решить задачу в подобном случае.

Вопрос. Показать, как за время $O(N)$ построить плотное двоичное слово любой заданной длины N.

[**Указание:** рассмотрите гамильтонов и эйлеров циклы в автомате де Брёйна.]

Решение

Пусть $A = \{0,1\}$ – алфавит. Зафиксируем N, и пусть n такое, что $\Delta_{n-1} < N \leq \Delta_n$, где $\Delta_n = 2^n + n - 1$. Нашей базовой структурой является граф де Брёйна G_n порядка n (графовая структура автомата де Брёйна) над алфавитом A. Вершинами G_n являются двоичные слова длины $n - 1$.

Говорят, что путь π в графе G_n является *эйлеровой цепью*, если он содержит все вершины G_n, возможно не по одному разу, и не содержит повторяющихся ребер. Обозначим $Word_n(\pi)$ слово, ассоциированное с эйлеровой цепью π в G_n.

Свойство 1. Если π является эйлеровой цепью длины $N - (n - 1)$ в графе G_n, то $Word_n(\pi)$ – плотное слово длины N.

Доказательство. Любое двоичное слово длины N, где $\Delta_{n-1} < N \leq \Delta_n$, содержит не больше 2^{n-1} (различных) факторов длины $n - 1$ и не больше $N - n + 1$ факторов длины n. Поэтому слово, для которого эти границы достигаются, плотное. В частности, если π – эйлерова цепь, то слово $Word_n(\pi)$ содержит все слова длины $n - 1$ и все его факторы длины n различны, потому что они соответствуют различным ребрам эйлеровой цепи в G_n. Следовательно, слово $Word_n(\pi)$ плотное. ■

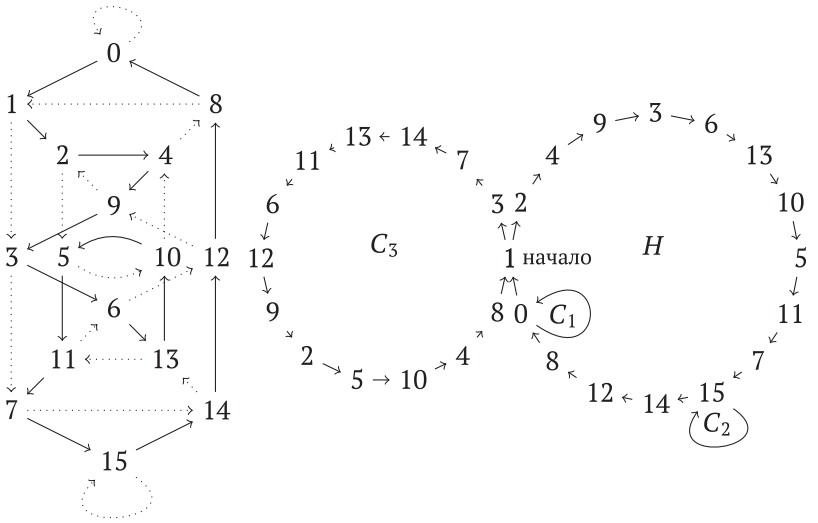
Теперь ответ на наш вопрос заключается в следующем свойстве.

Свойство 2. Эйлерову цепь длины $N - (n - 1)$ в графе G_n можно вычислить за линейное время.

Доказательство. Сначала вычисляем гамильтонов цикл H размера 2^{n-1} в G_n, определяемый эйлеровым циклом в G_{n-1}. Граф $G_n - H$ состоит из дизъюнктных простых циклов $C_1, C_2, \ldots, C_r$, называемых *ушными циклами*. Затем выбираем подмножество $C'_1, C'_2, \ldots, C'_t$ ушных циклов, для которого $\sum_{i=1}^{t-1}|C'_i| < M \leq \sum_{i=1}^{t}|C'_i|$. Далее добавляем префиксный подпуть c'_t цикла C'_t, чтобы получить

$\sum_{i=1}^{t-1}|C'_i| + |c'_t| = M.$

Ясно, что $H \cup C'_1 \cup C'_2 \cup \ldots \cup C'_{t-1} \cup c'_t$ можно расположить в виде последовательности, образующей эйлерову цепь длины M. Она начинается в любой вершине c'_t, затем проходит по ребрам H по каждому встретившемуся ушному циклу C'_i. После возврата в исходную точку она проходит по пути c'_t. ■



Пример. На рисунке выше показан граф G_5 (слева), вершинами которого являются двоичные слова длины 4, записанные для краткости в десятичном виде. На правом рисунке показана декомпозиция G_5 на не имеющие общих ребер ушные циклы H, C_1, C_2 и C_3. Цикл H – это гамильтонов цикл длины 16, C_1 и C_2 – петли, а C_3 – большой ушной цикл длины 14. Последние три ушных цикла покрывают пунктирные ребра (слева) графа G_5, которые не входят в H.

Чтобы вычислить плотное двоичное слово длины $N = 33$, мы сначала построим эйлерову цепь π длиной $21 = 25 - 4$. Мы можем начать с вершины 1, обойти гамильтонов цикл и дополнительно две петли, вернуться в вершину 1, а затем пройти по пути из 4 ребер, принадлежащему большому ушному циклу C_3. В этом случае $t = 3$, C'_1, C'_2 – петли, а $c'_3 = 1 \to 3 \to 7 \to 14$. Получается путь

$\pi = (1,2,4,9,3,6,13,10,5,11,7,15,15,14,12,8,0,0,1,3,7,14),$

помеченный двоичным словом

0 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 0 1 1 1 0.

Окончательное плотное слово длины 25 получается дописыванием в начало двоичного представления 0001 первой вершины 1:

$Word_5(\pi) = 0\,0\,0\,1\,0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1\,1\,0\,0\,0\,0\,0\,1\,1\,1\,0.$

Примечания

Впервые эффективный – и совершенно другой – алгоритм построения плотных слов был предложен в работе Shallit [222]. Заметим, что в нашем примере для $n = 5$ граф G_5 разлагается на четыре простых реберно-непересекающихся (не имеющих общих ребер) графа: гамильтонов цикл H, две петли и один большой ушной цикл длины $2^{n-1} - 2$. Если отбросить петли, то G_5 разлагается на два реберно-непересекающихся простых графа. На самом деле такую специальную декомпозицию любого двоичного графа G_n для $n > 3$ можно найти с помощью так называемых *дополнительных* гамильтоновых циклов (см. [206]). Но для вычисления плотных слов достаточно любой другой декомпозиции.

71. Факторный оракул

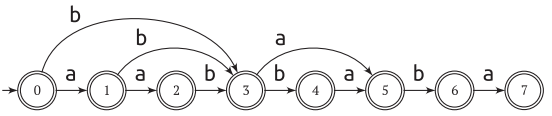
Факторный оракул – индексная структура, похожая на факторный или суффиксный автомат (или DAWG) слова x. Это детерминированный автомат, имеющий $|x| + 1$ состояний – минимальное число состояний, возможное в суффиксном автомате слова x. По этой причине такая структура данных очень удобна во многих приложениях, нуждающихся в простой индексной структуре, поскольку экономно расходует память и допускает эффективное построение в онлайновом режиме. Недостаток же в том, что оракул x допускает не только факторы x, но и некоторые другие слова.

Если v – фактор y, то обозначим $pocc(v, y)$ позицию y, следующую за первым вхождением v в y, т. е. $pocc(v, y) = \min\{|z| : z = wv$ – префикс $y\}$. Следующий алгоритм можно рассматривать как определение факторного оракула $\mathcal{O}(x)$ слова x. Он вычисляет автомат, для которого Q является множеством состояний, а E – множеством помеченных ребер.

```
Oracle(непустое слово x)
 1  (Q, E) ← ({0,1, … , |x|},∅)
 2  for i ← 0 to |x| − 1 do
 3      u ← кратчайшее слово, распознаваемое в состоянии i
 4      for a ∈ A do
 5          if ua ∈ Fact(x[i − |u|..|x| − 1]) then
 6              E ← E ∪ {(i, a, pocc(ua, x[i − |u|..|x| − 1]))}
 7  return (Q, E)
```

У этой структуры есть несколько интересных свойств. Все $|x| + 1$ ее состояний заключительные. Любое ребро, входящее в состояние $i + 1$, помечено буквой $x[i]$. Существует $|x|$ ребер вида $(i, x[i], i + 1)$, называемых внутренними. Остальные ребра, вида $(j, x[i], i + 1)$, где $j < i$, называются внешними. Таким образом, оракул можно представить словом x и его множеством внешних ребер без меток.

Пример. Оракул $\mathcal{O}(\texttt{aabbbaba})$ допускает все факторы aabbbaba, а также слово abab, которое фактором не является. Он определяется своими внешними непомеченными ребрами (0,3), (1,3) и (3,5).



Вопрос. Показать, что факторный оракул слова x имеет от $|x|$ до $2|x| - 1$ ребер.

Решение

Сначала отметим, что границы точны. Действительно, $\mathcal{O}(a^n)$ имеет n состояний для любой буквы a, а $\mathcal{O}(x)$ имеет $2|x| - 1$ состояний, когда буквы x попарно различны, т. е. $|alph(x)| = |x|$.

Факт. Пусть u – самое короткое среди слов, распознаваемых в состоянии i автомата $\mathcal{O}(x)$. Тогда $i = pocc(u, x)$, причем слово u определено однозначно. Обозначим его $sh(i)$.

Поскольку существует $|x|$ внутренних ребер, для ответа на вопрос мы должны показать, что количество внешних ребер меньше $|x|$. Для этого сопоставим каждому внешнему ребру вида (i, a, j), где $i < j - 1$, собственный непустой суффикс $sh(i)ax[j + 1..|x| - 1]$ слова x. Мы покажем, что это отображение инъективно.

Предположим, что существуют такие ребра (i_1, a_1, j_1) и (i_2, a_2, j_2), что

$$sh(i_1)a_1x[j_1 + 1..|x| - 1] = sh(i_2)a_2x[j_2 + 1..|x| - 1],$$

и без ограничения общности будем считать, что $i_1 \le i_2$.

- Если $j_1 < j_2$, то $sh(i_1)a_1$ – собственный префикс $sh(i_2)$. Положив $d = |sh(i_2)| - |sh(i_1)a_1|$, получим $j_1 = j_2 - d - 1$. Вхождение $sh(i_2)$ заканчивается в состоянии i_2, следовательно, вхождение $sh(i_1)a_1$ заканчивается в состоянии $i_2 - d < j_2 - d - 1 = j_1$. Но это противоречит построению факторного оракула x.
- Если $j_1 > j_2$, то слово $sh(i_2)$ является префиксом $sh(i_1)$. Следовательно, существует вхождение $sh(i_2)$, заканчивающееся до $i_1 \le i_2$, что также противоречит построению факторного оракула.

Таким образом, $j_1 = j_2$, откуда следует, что $a_1 = a_2$, $sh(i_1) = sh(i_2)$, $i_1 = i_2$ и, наконец, $(i_1, a_1, j_1) = (i_2, a_2, j_2)$.

Поскольку отображение инъективно и поскольку существует $|x| - 1$ собственных непустых суффиксов x, сложение внутренних и внешних ребер дает максимум $2|x| - 1$ ребер в факторном оракуле, что и требовалось доказать.

Вопрос. Спроектировать процедуру онлайнового построения факторного оракула слова x, работающую за линейное время для фиксированного алфавита и требующую линейного объема памяти.

[**Указание:** воспользуйтесь суффиксными ссылками.]

Решение
Поскольку оракул детерминирован, обозначим δ его функцию переходов, т. е. $\delta(i, a) = j \Leftrightarrow (i, a, j) \in E$. Пусть S – суффиксная ссылка, определенная на множестве состояний следующим образом: $S[0] = -1$ и для $1 \leq i \leq |x|$ $S[i] = \delta(0, u)$, где u – самый длинный (собственный) суффикс $x[0..i]$, для которого $\delta(0, u) < i$. Для примера выше имеем:

i	0	1	2	3	4	5	6	7
$x[i]$	a	a	b	b	a	b	a	
$S[i]$	-1	0	1	0	3	1	3	5

Факт. Пусть $k < i$ – состояние на суффиксном пути состояния i факторного оракула $x[0..i]$. Если $\delta(k, x[i+1])$ определено, то это верно для всех состояний на суффиксном пути k.

В силу этого факта на шаге i, $0 \leq i \leq |x| - 1$, процедуры онлайнового построения факторного оракула стандартным образом используются суффиксные ссылки:

- добавить состояние $i + 1$ и положить $\delta(i, x[i]) = i + 1$;
- пройти по суффиксному пути i, чтобы положить $\delta(S_k[i], x[i]) = i + 1$, когда это необходимо; и
- установить $S[i + 1]$.

Эта стратегия реализована в следующем алгоритме.

```
OracleOnline(непустое слово x)
 1  (Q, δ, S[0]) ← ({0}, не определено, −1)
 2  for i ← 0 to |x| − 1 do
 3      Q ← Q ∪ {i + 1}
 4      δ(i, x[i]) ← i + 1
 5      j ← S[i]
 6      while j > −1 и δ(j, x[i]) не определено do
 7          δ(j, x[i]) ← i + 1
 8          j ← S[j]
 9      if j = −1 then
10          S[i + 1] ← 0
11      else S[i + 1] ← δ(j, x[i])
12  return (Q,δ)
```

Правильность алгоритма OracleOnline вытекает в основном из равенства

$$(S[i], x[i], i + 1) = (S[i], x[i], S[i] + pocc(sh(S[i]), x[i - S[i]..|x| - 1])).$$

Линейность времени выполнения следует из того, что на каждой итерации цикла while в строках 6–8 создается внешний переход, а в $\mathcal{O}(x)$ может быть только $|x| - 1$ таких переходов. Цикл в строках 2–11 выполняется ровно $|x| - 1$ раз, а все остальные инструкции занимают постоянное время.

Линейность памяти следует из того, что факторному оракулу, равно как и массиву S, нужен только линейный объем памяти.

Вопрос. Показать, что факторный оракул $\mathcal{O}(x)$ можно использовать для нахождения всех вхождений слова x в текст, несмотря на то что оракул может допускать слова, не являющиеся факторами x.

[**Указание:** единственное слово длины $|x|$, распознаваемое $\mathcal{O}(x)$, – само x.]

Решение

Возможно решение, имитирующее алгоритм КМП, но по времени более эффективно использовать стратегию Бойера–Мура. С этой целью воспользуемся факторным оракулом слова x^{R}, обратного x. Окно длины $|x|$ скользит вдоль текста, и когда оракул допустит все окно целиком, фиксируется совпадение, поскольку окно содержит x, как сказано в указании.

Если имеет место несовпадение, т. е. фактор au текста не допущен оракулом, то au также не является фактором x. Раз так, то можно безопасно выполнить сдвиг на длину $|x - u|$. Эта стратегия реализована в следующем алгоритме. Он выводит начальные позиции всех вхождений x и y.

```
BackwardOracleMatching(непустые слова x, y)
 1   (Q, δ) ← OracleOnline(x^R)
 2   j ← 0
 3   while j ≤ |y| − |x| do
 4       (q, i) ← (0, |x| − 1)
 5       while δ(q, y[i + j]) определено do
 6           (q, i) ← (δ(q, y[i + j]), i − 1)
 7       if i < 0 then
 8           сообщить о вхождении слова x в позиции j текста y
 9           j ← j + 1
10       else j ← j + i + 1
```

Примечания

Понятие факторного оракула и его использование для поиска в тексте изучены в работе Allauzen et al. [5] (см. также [79]). Усовершенствования, внесенные в работах [109, 111], привели к разработке самых быстрых алгоритмов сопоставления со строками, используемых в большинстве распространенных приложений.

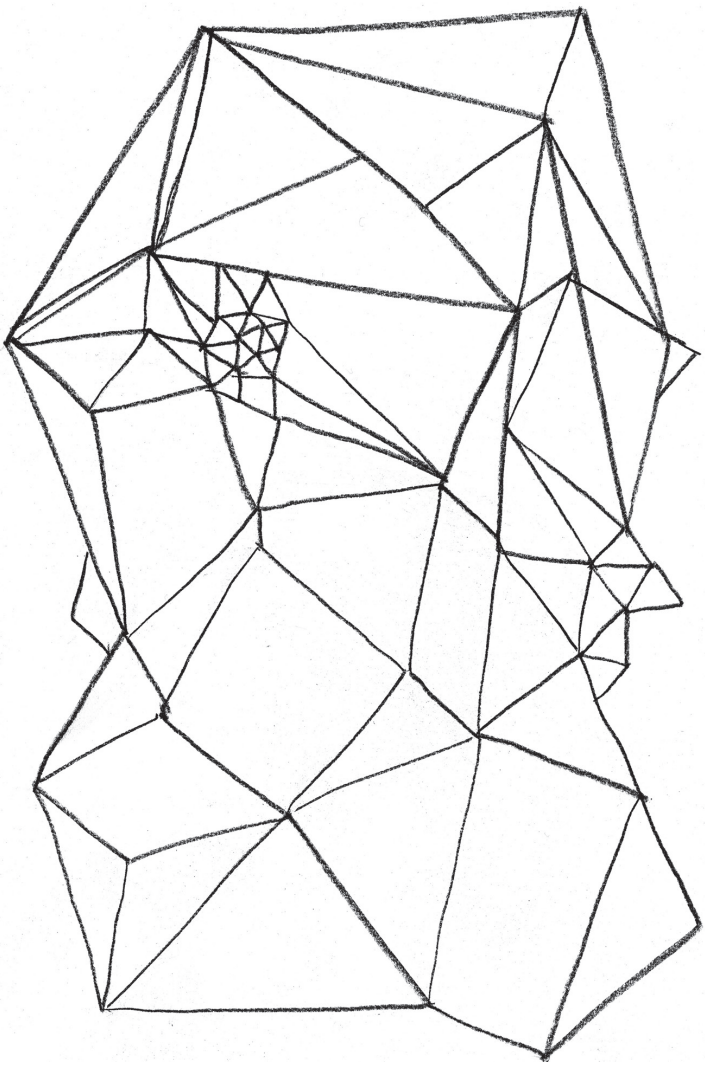
Точная характеристика языка слов, допускаемых факторным оракулом, изучалась в работе [182], а его статистические свойства представлены в [40].

В работе [173] оракул применяется для эффективного поиска повторений в словах при проектировании методов сжатия данных.

Эта структура данных хорошо подходит для компьютерных джазовых импровизаций, в которых состояниями являются ноты; для этой цели она была адаптирована в работе Assayag and Dubnov [17]. Другие работы на эту тему см. в проекте OMax по адресу recherche.ircam.fr/equipes/repmus/OMax/.

Глава 5

Регулярные структуры в словах



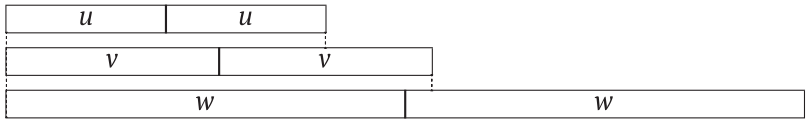
72. Три квадрата префиксов

Комбинаторный анализ квадратных префиксов слова приводит к нескольким следствиям, полезным при проектировании алгоритмов, связанных с периодичностью.

Три непустых слова u, v и w удовлетворяют условию квадрата префикса, если u^2 – собственный префикс v^2, а v^2 – собственный префикс w^2. Например, для слов $u = \mathtt{abaab}$, $v = \mathtt{abaababa}$, $w = \mathtt{abaababaabaabab}$ это условие удовлетворяется:

```
abaababaab
abaababaabaababa
abaababaabaababababaabababaabaabab
```

но ни u^2 не является префиксом v, ни v^2 не является префиксом w, иначе пример был бы тривиальным.



Вопрос. Показать, что если u^2, v^2 и w^2 удовлетворяют условию квадрата префикса и $|w| \leq 2|u|$, то $u, v, w \in z^2z^*$ для некоторого слова z.

Отсюда, в частности, следует, что слово u не примитивное. На самом деле этот вывод имеет место, если выполнены условие квадрата префикса и неравенство $|w| < |u| + |v|$ (лемма о трех квадратах префиксов). Но из условия в вопросе следует более сильный вывод, означающий по существу, что имеет место тривиальная ситуация, когда $w^2 = \mathtt{a}^k$ или что-то в этом роде.

Вопрос. Привести бесконечно много примеров троек слов, удовлетворяющих условию квадрата префикса, и таких, что одновременно $|u|+|v| = |w|$ и u примитивно.

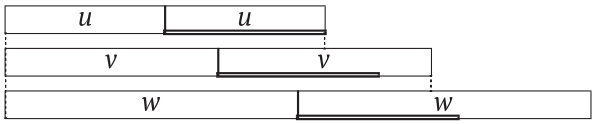
Следующий вопрос является следствием леммы о трех квадратах префиксов или первого утверждения. Точная или хотя бы достаточно жесткая верхняя граница упоминаемой в нем величины до сих пор неизвестна.

Вопрос. Показать, что факторами слова x могут быть менее $2|x|$ (различных) квадратов с примитивными корнями.

Еще одним прямым следствием леммы о трех квадратах префиксов является тот факт, что слово длины n имеет не более $\log_\Phi n$ префиксов, являющихся квадратами с примитивными корнями. Золотое сечение Ф возникает

из рекуррентного соотношения для чисел Фибоначчи, имеющих отношение ко второму вопросу.

Решение
Предположим, что $|w| \leq 2|u|$, как показано на рисунке.

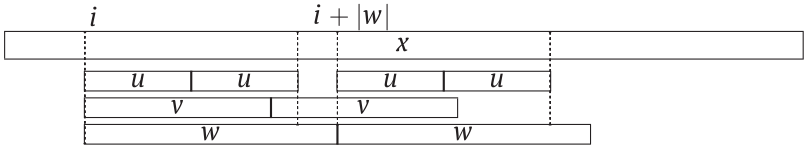


Условие в первом вопросе означает, что все три вхождения u в позициях $|u|$, $|v|$ и $|w|$ попарно перекрываются. Поэтому слово u имеет периоды $|v| - |u|$ и $|w| - |v|$, сумма которых не превосходит $|u|$, а значит, $q = \gcd(|v| - |u|, |w| - |v|)$ также является периодом u, в силу леммы о периодичности. Слово $z = u[0..p]$, где p – наименьший период u, является примитивным и потому встречается в u только в позициях kp, $k = 0, \ldots, \lfloor |u|/p \rfloor$. Кроме того, период p является делителем периода q, потому что $q < |u|/2$.

Слово z встречается в w^2 в позиции $|u|$ и в u в позиции $|u| + |v| - |w|$. Так как $|u| + |v| - |w|$ и $|w| - |v|$ кратны p, их сумма $|u|$ тоже кратна p, и, значит, u является целой степенью z; таким образом, $u \in z^2z^*$. То же самое справедливо для v и w, потому что $|v| - |u|$ и $|w| - |v|$ кратны $p = |z|$.

Бесконечное слово **s**, предел последовательности, определенной рекуррентно: $s_1 = \texttt{aab}$, $s_2 = \texttt{aabaaba}$, $s_i = s_{i-1}s_{i-2}$ для $i \geq 3$, содержит бесконечное количество префиксных троек, что отвечает на второй вопрос. Длины первых троек равны (3,7,10), (7,10,17), (10,17,27). Похожее поведение демонстрирует бесконечное слово Фибоначчи.

Чтобы подсчитать количество квадратов с примитивными корнями, которые являются факторами слова x, сопоставим каждому из них его самую правую начальную позицию в x. Если какая-нибудь позиция i будет сопоставлена трем квадратам u^2, v^2 и w^2, как на рисунке ниже, то, поскольку u примитивно, в силу утверждения из первого вопроса, самый короткий квадрат u^2 является собственным префиксом w. Тогда u^2 еще раз встречается в позиции $i + |w|$, что противоречит тому, что i – самая правая начальная позиция u^2. Таким образом, никакой позиции не может быть сопоставлено более двух квадратов. А поскольку последняя позиция x не рассматривается, общее число квадратных факторов меньше $2|x|$.



Примечания
Лемма о трех квадратах префиксов и следствия из нее изучались в работе Crochemore and Rytter [97] (см. также [74, глава 9] и [175, глава 8]). Первое утверждение и вариации на тему леммы взяты из работы Bai et al. [22].

Задача о подсчете квадратных факторов и представленный результат взяты из работы Fraenkel and Simpson [118]. Прямые простые доказательства даны в работах Hickerson [141] и Ilie [146]. Немного улучшенные верхние границы приведены в работах Ilie [147] и Deza et al. [103].

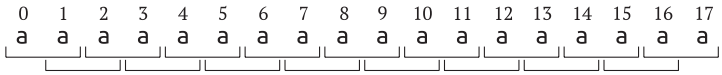
73. Точные границы количества вхождений степеней

В этой задаче рассматриваются нижние границы количества вхождений целых степеней в слово.

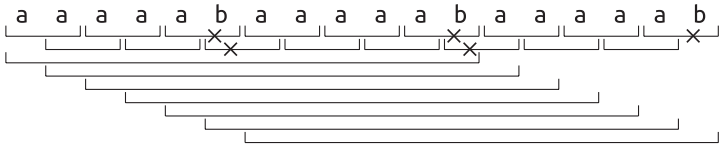
Целой степенью называется слово вида u^k для некоторого непустого слова u и некоторого целого $k > 1$. Известно, что размер множества квадратных факторов (задача 72) и количество серий (задача 86) в слове линейно зависят от длины слова. Но количество вхождений целых степеней не обладает этим свойством.

Чтобы не отвлекаться на тривиальные случаи, будем рассматривать целые степени с примитивными корнями, т. е. степени вида u^k, где u – примитивное слово (т. е. само не являющееся степенью).

Для начала рассмотрим слово $\mathtt{a}^n$. Оно содержит квадратичное число вхождений квадратов, но ровно $n - 1$ вхождений квадратов с примитивными корнями (на рисунке ниже они подчеркнуты).



Однако если несколько вхождений a заменить на b (см. ниже), то количество квадратов с примитивными корнями возрастает, хотя некоторые вхождения коротких квадратов пропадают (когда n достаточно велико).



Рассмотрим следующую рекуррентно определенную последовательность слов:

$$\begin{cases} x_0 = \mathtt{a}^5\mathtt{b}, \\ x_{i+1} = (x_i)^3\mathtt{b}, \quad i \geq 0. \end{cases}$$

Вопрос. Показать, что x_i содержит асимптотически $\Omega(|x_i| \log |x_i|)$ вхождений квадратов с примитивными корнями.

На самом деле это свойство имеет место не только для квадратов, но и для любых степеней $k \geq 2$.

Вопрос. Для данного целого $k \geq 2$ определить последовательность слов y_i, $i \geq 0$, содержащую асимптотически $\Omega(|y_i| \log |y_i|)$ вхождений k-х степеней с примитивными корнями.

Заметим, что граница точная в силу верхней границы на квадратные префиксы, доказанной в задаче 72.

Решение

Рассмотрим последовательность слов x_i длиной ℓ_i, и пусть c_i – число вхождений в x_i квадратов с примитивными корнями.

Имеем (глядя на $(x_0)^3$ на рисунке выше и учитывая вхождение суффикса bb в x_1):

$$\begin{cases} l_0 = 6, & c_0 = 4, \\ l_1 = 19, & c_1 = 20. \end{cases}$$

Заметим, что все короткие квадраты встречаются в каждом вхождении $\mathtt{a}^5\mathtt{b}$ в x_1 и что $\mathtt{a}^5\mathtt{b}$ само является примитивным словом. По индукции, это свойство имеет место для всех квадратов, содержащихся в x_i. Это дает такие рекуррентные соотношения для $i > 0$:

$$\begin{cases} l_1 = 19, & c_1 = l_1 + 1, \\ l_{i+1} = 3l_1 + 1, & c_{i+1} = 3c_i + l_1 + 2. \end{cases}$$

Тогда асимптотически получаем $l_{i+1} \approx 3^i l_1$, $c_{i+1} > 3^i c_1 + i3^{i-1} l_1$ и $i \approx \log |x_{i+1}|$, что доказывает утверждение из первого вопроса.

Если k – показатель степени в рассматриваемых степенях, то для некоторого целого положительного m можно определить такую последовательность слов:

$$\begin{cases} y_0 = \mathtt{a}^m\mathtt{b}, \\ y_{i+1} = (y_i)^{k+1}\mathtt{b}, & i > 0, \end{cases}$$

что индуцирует следующие соотношения:

$$l_{i+1} = (k+1)l_i + 1,\ c_{i+1} = (k+1)c_i + l_i + 2,$$

– и также дает нижнюю границу $\Omega(|y_i| \log |y_i|)$ количества вхождений k-х степеней с примитивными корнями.

Примечания

Нижняя граница числа квадратов с примитивными корнями реализуется для слов Фибоначчи [64]. В доказательстве используется тот факт, что в словах Фибоначчи нет факторов, являющихся четвертыми степенями. Наличие границы было также продемонстрировано в работе Gusfield and Stoye [135].

Асимптотическая нижняя граница числа вхождений k-х степеней доказана в работе [72], откуда заимствована идея представленного выше доказательства.

74. Вычисление серий для алфавитов общего вида

В этой задаче наша цель – спроектировать алгоритм вычисления серий в слове без дополнительных предположений об алфавите. Иначе говоря, алгоритм должен использовать модель равенства букв, т. е. разрешены лишь сравнения на $=$ и $\neq$.

В задаче 87 рассматривается вычисление серий для алфавитов, допускающих линейную сортировку, это необходимое условие для получения алгоритма с линейным временем работы.

Серией в слове x называется максимальное вхождение периодического фактора x. Формально это интервал позиций $[i..j]$, для которых наименьший период p фактора $x[i..j]$ удовлетворяет условию $2p \leq j - i + 1$ и выполняются оба условия $x[i - 1] \neq x[i + p - 1]$ и $x[j + 1] \neq x[j - p + 1]$, когда эти неравенства имеют смысл. Центром серии $[i..j]$ называется позиция $i + p$.

Чтобы не сообщать об одной и той же серии дважды, мы можем отсортировать серии по их центру. Будем говорить, что серия правоцентрирована в произведении слов uv, если она начинается в позиции, принадлежащей u, а ее центр принадлежит v. Серия называется левоцентрированной в uv, если ее центр принадлежит u, а заканчивается она в v.

Вопрос. Спроектировать алгоритм, который за линейное время находит правоцентрированные серии в произведении двух слов uv.

[**Указание:** воспользуйтесь таблицами префиксов, см. задачу 22.]

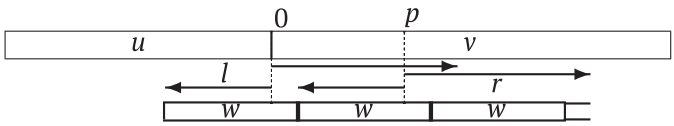
Вопрос. Спроектировать алгоритм, который за время $O(n \log n)$ вычисляет все серии в слове длины n, пользуясь моделью равенства.

[**Указание:** воспользуйтесь методом «разделяй и властвуй».]

Решение

Для ответа на первый вопрос будем искать серии в порядке возрастания их периодов. Как показано на рисунке, если задан потенциальный период серии p, то нужно лишь проверить, как далеко фактор $v[0..p - 1]$ распростра-

няется влево и вправо. Это наибольшие общие продолжения (longest common extension – LCE) из двух позиций, например $r = lcp(v, v[p..|v| - 1)$. Если сумма длин продолжений не меньше периода, то мы нашли серию.



Длина r правого продолжения находится просто из таблицы префиксов v. Длина l левого продолжения вычисляется аналогично с помощью таблицы префиксов слова $z = u^R\#v^Ru^R$, где # – символ, не встречающийся в uv.

Если выполняется условие $l \le p$ в строке 6 алгоритма ниже, то потенциальная серия центрирована в v, что и требуется. Величина *offset*, позиция в x одного из ее факторов, uv, прибавляется, чтобы сообщать о сериях как об интервалах позиций в x (а не uv).

```
RIGHT-CENTRED-RUNS(непустые слова u,v, смещение offset)
1  pref_v ← PREFIXES(v)
2  pref_z ← PREFIXES(u^R#v^Ru^R)
3  for p ← 1 to |v| − 1 do
4      r ← pref_v[p]
5      l ← pref_z[|u| + |v| − p + 1]
6      if l ≤ p и l + r ≥ p then
7          Сообщить о серии [|u| − l..|u| + p + r − 1] + offset
```

Алгоритм LEFT-CENTRED-RUNS, вычисляющий серии uv, центрированные в u, устроен точно так же и работает симметрично.

Время работы обоих алгоритмов зависит от сложности вычисления таблицы префиксов. Оно линейно зависит от длины входного слова, как показано в задаче 22. Кроме того, при вычислении используются только сравнения на $=$ и $\neq$.

Наконец, чтобы вычислить все серии в слове x, процедура разбивает x на два слова приблизительно одинаковой длины, как в алгоритме ниже. Для нахождения серий вызывается алгоритм RUNS(x, n, 0). Вспоминая о времени работы двух предыдущих алгоритмов, получаем, что все вычисление занимает время $O(n \log n)$ при условии использования указанной модели сравнения.

```
RUNS(непустое слово x длины n, смещение offset)
1  if n > 1 then
2      (u, v) ← (x[0..⌊n/2⌋], x[⌊n/2⌋ + 1..n − 1])
3      RUNS(u, ⌊n/2⌋ + 1, offset)
4      RUNS(v, n − ⌊n/2⌋ − 1, offset + ⌊n/2⌋ + 1)
5      RIGHT-CENTRED-RUNS(u, v, offset)
6      LEFT-CENTRED-RUNS(u, v, offset)
```

Заметим, что этот алгоритм может сообщать о некоторых сериях несколько раз. Это бывает, когда длинная серия в первой половине слова заходит во вторую его половину. Чтобы получить список серий без повторений, необходима фильтрация.

Примечания

Описанный метод вычисления серий представлен в работе [84] наряду с другими решениями, работающими за то же время в соответствии с вышеупомянутой моделью вычислений. В этой модели алгоритм оптимален, в силу результата из работы Main and Lorentz [179], где доказана нижняя граница $\Omega(n \log n)$ времени нахождения квадрата в слове.

75. Проверка перекрытий в двоичном слове

В этой задаче наша цель – спроектировать эффективный алгоритм проверки наличия перекрытия в двоичном слове. Перекрытием называется фактор с показателем степени больше 2. Иначе говоря, слово содержит перекрытие, если оно имеет фактор вида *auaua*, где *a* – буква, а *u* – слово.

Слово Туэ–Морса $\mu^\infty(\mathtt{a})$ – пример бесконечного слова, не содержащего перекрытий. Оно порождается с помощью морфизма μ ($\mu(\mathtt{a}) = \mathtt{ab}$ и $\mu(\mathtt{b}) = \mathtt{ba}$), который сохраняет свойство отсутствия перекрытий.

Для двоичного слова x определим его декомпозицию $uyv = x$ (формально тройку (u, y, v)): $|u|$ – наименьшая позиция в x самого длинного фактора y, принадлежащего множеству $\{\mathtt{ab}, \mathtt{ba}\}^+$. Эта декомпозиция называется факторизацией Рестиво–Салеми (RS), если $u, v \in \{\varepsilon,\mathtt{a},\mathtt{b},\mathtt{aa},\mathtt{bb}\}$. RS-факторизации следующим образом преобразуются в слова, принадлежащие $\{\mathtt{ab},\mathtt{ba}\}^*$, с помощью частичных функций f и g (c и d – буквы, а функция «надчерк» меняет $\mathtt{a}$ на $\mathtt{b}$ и наоборот):

$$f(uyv) = \begin{cases} y, & \text{если } u = v = \varepsilon, \\ \overline{c}cy, & \text{если } u = c \text{ или } cc \text{ и } v = \varepsilon, \\ yd\overline{d}, & \text{если } u = \varepsilon \text{ и } v = d \text{ или } dd, \\ \overline{c}cyd\overline{d}, & \text{если } u = c \text{ или } cc \text{ и } v = d \text{ или } dd; \end{cases}$$

$$g(uyv) = \begin{cases} y, & \text{если } u = v = \varepsilon, \\ \overline{c}cy, & \text{если } u = c \text{ или } cc \text{ и } v = \varepsilon, \\ yd\overline{d}, & \text{если } u = \varepsilon, \text{или } c, \text{или } cc \text{ и } v = d \text{ или } dd. \end{cases}$$

```
OverlapFree(непустое двоичное слово x)
1  while |x| > 6 do ▷ ниже c и d – буквенные переменные
2      uyv ← decomp(x)
3      if uyv не является RS-факторизацией then
4          return FALSE
```

```
5       if [u = cc и (ccc или cc̄cc̄c – префиксы uy)] или
          [v = dd и (ddd или dd̄dd̄dd – суффиксы uy)] then
6         return FALSE
7       if (u = c или u = cc) и (v = d или v = dd) и uyv – квадрат then
8         x ← μ^{-1}(g(uyv))
9       else x ← μ^{-1}(f(uyv))
10    return TRUE
```

Вопрос. Показать, что алгоритм OverlapFree за линейное время проверяет, что двоичное слово не содержит перекрытий.

Решение

Доказательство правильности алгоритма OverlapFree выходит за рамки нашего обсуждения, но мы приведем несколько полезных для этой цели свойств. Доказательство опирается на свойство декомпозиции свободных от перекрытий слов, используемое в алгоритме. Чтобы сформулировать его, введем обозначения:

$O = \{\mathtt{aabb},\mathtt{bbaa},\mathtt{abaa},\mathtt{babb},\mathtt{aabab},\mathtt{bbaba}\}$,
$E = \{\mathtt{abba},\mathtt{baab},\mathtt{baba},\mathtt{abab},\mathtt{aabaa},\mathtt{bbabb}\}$.

Пусть x – двоичное слово, не содержащее перекрытий. Тогда если x имеет префикс, принадлежащий O, то $x[j] \neq x[j-1]$ для некоторой нечетной позиции j такой, что $3 \leq j \leq |x| - 2$. А если x имеет префикс, принадлежащий E, то $x[j] \neq x[j-1]$ для некоторой четной позиции j такой, что $4 \leq j \leq |x| - 2$. Следовательно, если слово достаточно длинное, то оно имеет длинный фактор, принадлежащий $\{\mathtt{ab},\mathtt{ba}\}^+$. Именно, если $|x| > 6$, то x единственным образом разлагается в произведение uyv, где $u, v \in \{\varepsilon,\mathtt{a},\mathtt{b},\mathtt{aa},\mathtt{bb}\}$ и $y \in \{\mathtt{ab},\mathtt{ba}\}^+$.

Выполняя такое разложение итеративно, мы получаем единственное разложение x в произведение

$$u_1 u_2 \ldots u_r \cdot \mu^{r-1}(y) \cdot v_r \ldots v_2 v_1,$$

где $|y| < 7$ и $u_s, v_s \in \{\varepsilon, \mu^{s-1}(\mathtt{a}), \mu^{s-1}(\mathtt{b}), \mu^{s-1}(\mathtt{aa}), \mu^{s-1}(\mathtt{bb})\}$.

Что касается времени работы OverlapFree, заметим, что инструкции в цикле while выполняются за время $O(|x|)$. Поскольку длина x уменьшается вдвое на каждом шаге, в силу действия морфизма Туэ–Морса, то полное время работы цикла линейно. Последняя проверка производится для слова длиной не больше 6, так что занимает постоянное время. Таким образом, алгоритм требует времени $O(|x|)$.

Примечания

Большинство свойств свободных от перекрытий слов, рассматриваемых в этой задаче, было доказано в работе Restivo and Salemi [207], где был установлен полиномиальный рост их числа с увеличением длины слова. Представленный здесь алгоритм взят из работы [158], в которой доказаны более

точные свойства свободных от перекрытий слов, что позволило немного уменьшить верхнюю оценку их числа в слове заданной длины.

Описанный алгоритм дает прямое решение вопроса. Более общее решение, требующее дополнительных инструментов, представлено в задаче 87, где описан алгоритм, вычисляющий все серии в слове. Чтобы с его помощью установить, что слово не содержит перекрытий, достаточно проверить, что показатель степени любой серии равен в точности 2 (он не может быть меньше, по определению серии). Этот алгоритм также работает за линейное время на двоичных словах.

76. Игра, свободная от перекрытий

Эта игра опирается на понятие перекрытий, встречающихся в словах. Слово содержит перекрытие (фактор с показателем степени больше 2), если один из его факторов имеет вид *avava*, где *a* – буква, а *v* – слово.

В свободную от перекрытий игру длины n над алфавитом {0,1,2,3} играют двое, Анна и Бен. Игроки продолжают первоначально пустое слово, дописывая в конец по одной букве. Игра заканчивается, когда слово достигает длины n.

Мы предполагаем, что Бен делает первый ход и что n четно. Анна выигрывает, если в конечном слове нет перекрытий, в противном случае выигрывает Бен.

Выигрышная стратегия Анны. Пусть $d \in A$ буква, которую Анна добавляет на k-м ходе. Если Бен только что добавил букву c, то d определяется как

$$d = c \oplus \mathbf{f}[k],$$

где $x \oplus y = (x + y) \bmod 4$ а $\mathbf{f} = f^{\infty}(1)$ – бесконечное свободное от квадратов слово, получающееся повторным применением морфизма f, определенного на множестве $\{1,2,3\}^*$ следующим образом: $f(1) = 123$, $f(2) = 13$ и $f(3) = 2$ (см. задачу 79). Слово **f** и последовательность ходов выглядят так:

f		1		2		3		1		3		2		1		2	...
ходы	0	1	2	0	0	3	2	3	3	2	3	1	1	2	1	3	...

Вопрос. Показать, что, применяя выигрышную стратегию, Анна всегда выиграет у Бена в свободную от перекрытий игру любой четной длины n.

[**Указание:** сумма букв любого фактора **f** нечетной длины не делится на 4.]

Решение

Для ответа на вопрос мы, безусловно, воспользуемся тем, что слово **f** свободно от квадратов, а также важнейшим свойством, упомянутым в указании.

Доказательство свойства в указании. Пусть $\alpha = |v|_1$, $\beta = |v|_2$ и $\gamma = |v|_3$ – количество вхождений в v букв 1, 2 и 3 соответственно. По определению, слово **f** состоит из блоков 123, 13 и 2. Поэтому между любыми двумя последовательными вхождениями 3 ровно один раз встречается 1. Отсюда следует, что $|\alpha - \gamma| \leq 1$.

Если $|\alpha - \gamma| = 1$, то $\alpha + 2\beta + 3\gamma$ не делится на 2 и, следовательно, не может делиться на 4.

В противном случае $\alpha = \gamma$, и тогда β нечетно, потому что длина $\alpha + \beta + \gamma = |v|$ нечетна. Отсюда следует, что $2\beta \bmod 4 = 2$. Следовательно, $\alpha \oplus 2\beta \oplus 3\gamma = 2\beta \bmod 4 = 2$, и сумма букв v не делится на 4, что и требовалось доказать.

Правильность стратегии Анны. Доказательство проведем от противного. Предположим, что в какой-то момент в игре в слове w образуется перекрытие, имеющее вид $cvcvc$ для $c \in A$. Рассмотрим два случая.

Длина $|cv|$ четна. Если положить $u = cv$ или $u = vc$, то слово w будет содержать квадрат uu, для которого $|u|$ четна, а его первая буква – ход Бена в игре. Квадрат имеет вид

$$uu = b_1a_1b_2a_2 \ldots b_ka_k\, b_1a_1b_2a_2 \ldots b_ka_k,$$

где буквы b_i соответствуют ходам Бена, а буквы a_i – ходам Анны. Обозначим $x \ominus y = (x - y) \bmod 4$, тогда слово $e_1e_2\ldots e_k\ e_1e_2\ldots e_k$, где $e_i = (b_i \ominus a_i)$ является квадратом в **f**. Но это невозможно, потому что слово **f** свободно от квадратов.

Длина $|cv|$ нечетна. Как и выше, слово w содержит квадрат uu, для которого $|u|$ нечетна, а его первая буква соответствует ходу Бена. Заметим, что $|u| > 1$, потому что вторая буква соответствует ходу Анны и отлична от хода Бена.

Мы проведем доказательство для $|u| = 7$, и на этом примере будет понятен ход рассуждений в общем случае. Пусть $u = b_1a_1\, b_2a_2\, b_3a_3\, b_4$, где b_i – ходы Бена, а a_i – ходы Анны. Квадрат имеет вид

$$uu = b_1a_1\, b_2a_2\, b_3a_3\, b_4b_1\, a_1b_2\, a_2b_3\, a_3b_4.$$

Следовательно, **f** содержит фактор $e_1e_2e_3e_4e_5e_6e_7$, где

$$e_1 = a_1 \ominus b_1,\ e_2 = a_2 \ominus b_2,\ e_3 = a_3 \ominus b_3,\ e_4 = b_1 \ominus b_4,$$
$$e_5 = b_2 \ominus a_1,\ e_6 = b_3 \ominus a_2,\ e_7 = b_4 \ominus a_3.$$

Имеем:

$$e_1 \oplus e_2 \oplus e_3 \oplus e_4 \oplus e_5 \oplus e_6 \oplus e_7 = 0,$$

что легко видеть, записав эту сумму в виде

$$(a_1 \ominus b_1) \oplus (b_1 \ominus b_4) \oplus (b_4 \ominus a_3) \oplus (a_3 \ominus b_3) \oplus (b_3 \ominus a_2) \oplus (a_2 \ominus b_2) \oplus (b_2 \ominus a_1).$$

Но это невозможно, потому что, согласно утверждению в указании, сумма букв любого фактора **f** нечетной длины не делится на 4.

Поскольку ни тот, ни другой случаи невозможны, w не содержит перекрытий, и стратегия Анны приводит ее к выигрышу.

Примечания

Решение этой задачи – вариант стратегии в игре Туэ, описанной в работе [132]. Отметим, что у Бена имеется простая выигрышная стратегия в случае, когда в игре участвует только три буквы и Анна придерживается похожей стратегии.

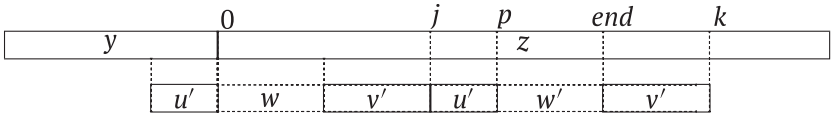
77. Заякоренные квадраты

При поиске квадратных факторов в слове с применением техники «разделяй и властвуй» естественно искать квадраты в произведении двух слов, свободных от квадратов. Мы рассмотрим этот вопрос и затем реализуем алгоритм проверки отсутствия квадратов в слове длины n, работающий за время $O(n \log n)$.

Метод, основанный на таблицах префиксов (см. задачу 74), преследует ту же цель, но требует таблиц размера $O(n)$, тогда как описанное ниже решение нуждается лишь в нескольких переменных, помимо входных данных.

Пусть y и z – свободные от квадратов слова. Алгоритм Right проверяет только, содержит ли yz квадрат, центрированный в z. Другие квадраты в произведении можно найти симметрично.

Алгоритм исследует все возможные периоды квадрата. Для данного периода p (см. рисунок) алгоритм вычисляет самый длинный суффикс $u' = z[j..p-1]$, общий для y и $z[0..p-1]$. Затем он проверяет, встречается ли $z[0..j-1]$ в позиции p слова z, для чего просматривает z, начиная с правой позиции $k-1$ потенциального квадрата. В случае успеха квадрат найден.



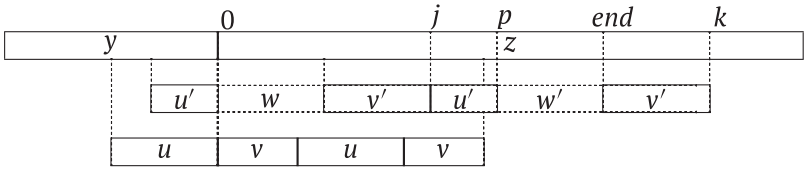

```
Right(непустые свободные от квадратов строки y, z)
 1  (p, end) ← (|z|, |z|)
 2  while p > 0 do
 3      j ← min{q : z[q..p − 1] суффикс y}
 4      if j = 0 then
 5          return TRUE
 6      k ← p + j
 7      if k < end then
 8          end ← min{q : z[q..k − 1] – суффикс z[0..j − 1]}
 9          if end = p then
10              return TRUE
11      p ← max{j − 1, ⌊p/2⌋}
12  return FALSE
```

Вопрос. Показать, что алгоритм Right возвращает TRUE тогда и только тогда, когда слово yz содержит квадрат с центром в z, и что его время работы составляет $O(|z|)$ при постоянном объеме дополнительной памяти.

В алгоритме Right переменная *end* и инструкции в строках 7 и 11 играют важнейшую роль для достижения анонсированного в вопросе времени работы. Заметим, что время не зависит от длины y.

Решение

Правильность алгоритма Right. Доказательство опирается на следующее утверждение, комбинаторное доказательство которого мы оставляем читателю. Оно иллюстрируется на рисунке ниже, где $u' = z[j\,..\,p-1]$ – самый длинный общий суффикс y и $z[0..p-1]$, вычисляемый в строке 3, а v' – самый длинный общий суффикс $z[0..j-1]$ и $z[p..k-1]$, который может вычисляться в строке 8. В алгоритм можно добавить проверку, чтобы отбрасывать пустые u', потому что это не может привести к квадрату, т. к. z свободно от квадратов.



***Лемма 6**. Пусть y и z – слова, свободные от квадратов, и vuv – самый короткий префикс z такой, что и является суффиксом y. Пусть u' и v' определены, как описано выше, а w и w', $|w| = |w'|$ – как показано на рисунке.*

Предположим, что vu – собственный префикс $wv'u'$. Тогда vu – собственный префикс wv' или $|vu| \le |wv'u'|/2$. Слово vuv также является префиксом $wv'u'w'$.

Правильность алгоритма вытекает из леммы после проверки того, что u' и v' правильно вычислены с помощью индексов j и end соответственно. При присваивании p следующего значения в строке 11 применяется первое утверждение леммы. Второе утверждение используется в строке 7 после присваивания переменной k, чтобы пропустить бесполезное вычисление v' в случае, когда условие не выполнено.

Время работы алгоритма Right в худшем случае определяется максимальным числом сравнений букв, которое мы сейчас оценим. Пусть p' и p'' ($p' > p''$) – два последовательных значения переменной p в процессе выполнения алгоритма, т. е. p' – значение p в начале цикла while, а p'' – его значение в конце выполнения цикла.

Если добавить проверку для отбрасывания пустого u', то будем иметь $p'' = p' - 1$ после одного сравнения. В противном случае имеем $p'' = \max\{j' - 1, \lfloor p'/2 \rfloor\}$, где j' – значение j после выполнения строки 3. Если $j' - 1$ – максимум, то число сравнений в этой строке равно $p' - p''$. В противном случае число сравнений не больше $2(p' - p'')$. Суммируя по всем итерациям цикла, получаем, что в строке 3 выполняется не более $2|z|$ сравнений букв.

Благодаря переменной end все успешные сравнения букв $z[p..end-1]$ в строке 8 производятся в разных позициях z, так что максимальное число сравнений равно $|z|$. Кроме того, существует не более одного неудачного сравнения для каждого значения p. Следовательно, в строке 8 производится не более $2|z|$ сравнений букв. Таким образом, общее число сравнений букв не превосходит $4|z|$, так что время работы алгоритма составляет $O(|z|)$.

Примечания

Число сравнений букв, выполняемых алгоритмом Right для слов y и z, равно $2|z| - 1$, когда y – слово Зимина (см. задачу 43), а $z = \#y$, где # – буква, не встречающаяся в y, например когда $y =$ abacabadabacaba.

Впервые алгоритм Right с постоянным объемом дополнительной памяти был описан в работе Main and Lorentz [179]. Небольшое усовершенствование, включенное в представленный вариант, взято из работы [66].

Решение вопроса с применением таблиц префиксов или их аналогов, как, например, в задаче 74, описано в работах [74, 98]. Значения j и end, вычисляемые в алгоритме Right, часто называют наибольшими общими продолжениями (longest common extension – LCE). Их можно найти за постоянное время после некоторой предобработки, если алфавит допускает линейную сортировку; см., например, метод, предложенный в работе Fischer and Heun [115]. Такого рода решение используется в задаче 87.

Решение вопроса с помощью двойственного алгоритма Left приводит к алгоритму, который проверяет, свободно ли от квадратов слово длины n, и требует времени $O(n \log n)$ при постоянном объеме дополнительной памяти. Его оптимальность доказана в работе [179]. Для алфавита фиксированного размера этот алгоритм дает тест на свободу от квадратов, требующий линейного времени (см. [67]), в котором применяется факторизация слова по типу факторизации Лемпеля–Зива, описанной в главе 6.

Обобщение на вычисление за время $O(n \log n)$ серий, встречающихся в слове длины n, рассматривается в работе [84].

78. Слова, почти свободные от квадратов

Проверить, что слово, не содержащее коротких квадратов, вообще свободно от квадратов, можно проще и эффективнее, чем с помощью методов, рассчитанных на обыкновенные слова. Это и есть предмет настоящей задачи.

Говорят, что слово w *почти свободно от квадратов*, если оно не содержит квадратных факторов длины, меньшей $|w|/2$. У таких слов есть полезное свойство, сформулированное в наблюдении ниже, где $Occ(z, w)$ обозначает множество начальных позиций вхождений z в слово w.

Наблюдение 1. Если z – фактор длины $|w|/8$ слова w, почти свободного от квадратов, то z непериодический (его наименьший период больше, чем $|z|/2$), $|Occ(z, w)| < 8$ и $Occ(z, w)$ можно вычислить за линейное время, воспользовавшись дополнительной памятью постоянного объема.

Если выполнены предположения, сформулированные в наблюдении, то вычисление $Occ(z, w)$ реализует, например, алгоритм NaiveSearch, наивный вариант алгоритма КМП.

```
NaiveSearch(непустые слова z, w)
 1  (i, j) ← (0,0)
 2  Occ(z, w) ← ∅
```

```
3  while j ≤ |w| − |z| do
4      while i < |z| и z[i] = w[j + i] do
5          i ← i + 1
6      if i = |z| then
7          Occ(z, w) ← Occ(z, w) ∪ {j}
8      (j, i) ← (j + max{1, ⌊i/2⌋}, 0)
9  return Occ(z, w)
```

Вопрос. Спроектировать алгоритм, который за линейное время и с дополнительной памятью постоянного объема проверяет, является ли почти свободное от квадратов слово w свободным от квадратов, предполагая для простоты, что $|w| = 2^k$, $k \geq 3$.

[**Указание:** воспользуйтесь разложением w на короткие факторы и примените алгоритм NaiveSearch и наблюдение 1.]

Решение

Идея решения заключается в том, чтобы разложить w на короткие блоки, мимо которых не может проскочить большой квадрат.

Для этого положим $l = 2^{k-3}$ и $z_r = w[r \cdot l..r \cdot l + l - 1]$, $r = 0, 1, \ldots, 7$. Положим также $Z = \{z_0, z_1, \ldots, z_7\}$.

Рассмотрим операцию $TestSquare(p, q)$, которая проверяет, существует ли в w квадрат длины $2(q - p)$, содержащий позиции p, q, $p \leq q$. Эта операция легко выполняется за время $O(n)$ с постоянной памятью, нужно лишь воспользоваться продолжением влево и вправо, как в задаче 74. При наличии данной операции нетрудно доказать следующий факт.

Наблюдение 2. Если w почти свободно от квадратов, то оно содержит квадрат тогда и только тогда, когда

$$\exists z \in Z\ \exists p, q \in Occ(z, w)\ TestSquare(p, q) = true.$$

Мы знаем, что множества Z и $Occ(z, w)$ постоянного размера. Теперь искомый алгоритм является прямой реализацией наблюдений 1 и 2 с применением постоянного числа выполнений алгоритмов NaiveSearch и TestSquare (последний реализует операцию $TestSquare(p, q)$). Так как каждый из них работает линейное время и использует память постоянного объема, то мы ответили на вопрос.

Примечания

Описанный выше метод можно без труда обобщить и проверять, является ли свободным от квадратов слово длины $n = 2^k$, не имеющее квадратных факторов длины, меньшей 2^3. При этом получится алгоритм, работающий за время $O(n \log n)$ в памяти постоянного объема. Приведем набросок решения. Для каждого $m = 3, 4, \ldots, k$ именно в таком порядке алгоритм проверяет, свободны ли от квадратов перекрывающиеся отрезки длиной 2^m в предположении, что они почти свободны от квадратов. Отрезки, которые перекрываются,

выбираются интервалами длины 2^{m-1}. Как только квадрат найден, алгоритм останавливается и сообщает о вхождении. Поскольку для данного m суммарная длина отрезков равна $O(n)$, полное время работы алгоритма составляет $O(n \log n)$.

Представленный алгоритм основан на методе из работы Main and Lorentz [180].

79. Двоичные слова с небольшим числом квадратов

В этой задаче наша цель – найти двоичные слова, содержащие наименьшее число различных квадратных факторов.

Квадратом называется слово с четным показателем степени; оно имеет вид $u^2 = uu$, где u – непустое слово. Самыми длинными словами над двоичным алфавитом {0,1}, не содержащими квадратов в качестве факторов, являются 010 и 101. Но над трехбуквенными алфавитами существуют бесконечные слова, свободные от квадратов. Одно из них, над алфавитом {a,b,c}, получается повторным применением морфизма f, определенного следующим образом:

$$\begin{cases} f(\mathtt{a}) = \mathtt{abc} \\ f(\mathtt{b}) = \mathtt{ac} \\ f(\mathtt{c}) = \mathtt{b} \end{cases}.$$

При этом получается свободное от квадратов бесконечное слово:

$$\mathbf{f} = f^{\infty}(\mathtt{a}) = \mathtt{abcacbabcbacabcacbacabcb}\ldots,$$

несмотря на то что f не сохраняет свойство «бесквадратности», поскольку $f(\mathtt{aba}) = \mathtt{abcacabc}$ содержит квадрат $(\mathtt{ca})^2$.

Кубом называется слово с показателем степени 3.

Вопрос. Показать, что никакое бесконечное двоичное слово не может содержать менее трех квадратов. Показать, что никакое бесконечное двоичное слово, содержащее ровно три квадрата, не может быть свободным от кубов.

Определим морфизм g: $\{\mathtt{a},\mathtt{b},\mathtt{c}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$ следующим образом:

$$\begin{cases} g(\mathtt{a}) = \mathtt{01001110001101} \\ g(\mathtt{b}) = \mathtt{0011} \\ g(\mathtt{c}) = \mathtt{000111} \end{cases}.$$

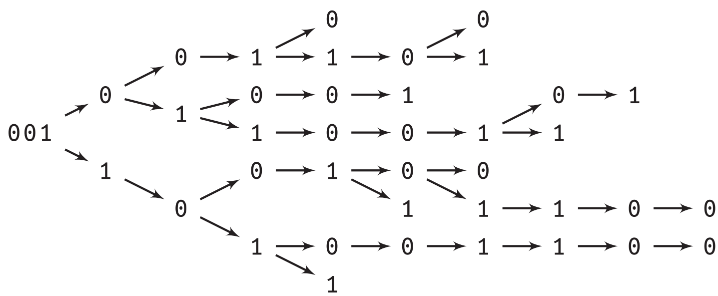
Заметим, что $g(\mathtt{ab})$ содержит три квадрата $\mathtt{0}^2$, $\mathtt{1}^2$ и $\mathtt{10}^2$, а также два куба $\mathtt{0}^3$ и $\mathtt{1}^3$.

Вопрос. Показать, что в слове $\mathbf{g} = g(f^\infty(\mathtt{a}))$ встречается только три квадрата и два куба.

[**Указание:** рассмотрите расстояния между последовательными вхождениями 000.]

Решение

Для проверки первого утверждения достаточно рассмотреть префиксное дерево двоичных слов. Аналогично слово, содержащее ровно три квадрата и ни одного куба, имеет максимальную длину 12, в чем позволяет убедиться следующее префиксное дерево.



Чтобы доказать свойство **g**, рассмотрим вхождения в него слова 000. Расстояния между его последовательными вхождениями принадлежат множеству {7,11,13,17}:

$g(\mathtt{ac})$	= 01001110001101 000111	7
$g(\mathtt{abc})$	= 01001110001101 0011 000111	11
$g(\mathtt{ca})$	= 000111 01001110001101	13
$g(\mathtt{cba})$	= 000111 0011 01001110001101	17.

Факторы **g**, содержащие несколько вхождений 000, имеют ограниченную длину, и можно непосредственно проверить, что они включают не больше квадратов, чем ожидается. Справедливость утверждения для других факторов мы докажем от противного.

Предположим, что **g** содержит достаточно большой квадрат w^2 с четным числом вхождений 000. Рассмотрим два последовательных вхождения по обе стороны от центра этого квадрата с расстоянием между ними 7. Это означает, что центр квадрата принадлежит вхождению 1101 внутри $g(\mathtt{ac})$. Так как множество $\{g(\mathtt{a}), g(\mathtt{b}), g(\mathtt{c})\}$ является префиксным кодом, то, взяв при необходимости слово, сопряженное квадрату, мы обнаружим, что оно имеет вид $g(\mathtt{c}v\mathtt{ac}v\mathtt{a})$ для некоторого слова $v \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$. Это противоречит тому, что $f^\infty(\mathtt{a})$ свободно от квадратов.

Случаи, когда расстояние между последовательными вхождениями 000 равно 11, 13 или 17, рассматриваются аналогично.

Теперь предположим, что w^2 содержит нечетное число вхождений 000. Тог-

да w имеет вид $0y00$ или симметричный $00y0$ для некоторого двоичного слова y. Взятие сопряженного, как и выше, дает квадрат в $f^{\infty}(\mathtt{a})$, что противоречит его определению.

Примечания

Свободное от квадратов слово **f** строится по-другому в задаче 80 после преобразования в другой алфавит с помощью морфизма α, определенного следующим образом: $\alpha(1) = \mathtt{c}$, $\alpha(2) = \mathtt{b}$, $\alpha(3) = \mathtt{a}$.

Существование бесконечного двоичного слова, содержащего только три квадрата и два куба, первоначально было доказано в работе Fraenkel and Simpson [117]. Более простые доказательства даны в работах Rampersad et al. [205] и Badkobeh [18] (см. обсуждение смежных вопросов в [19]). Представленное здесь доказательство с применением морфизма g взято из работы [18].

80. Построение длинных свободных от квадратов слов

Слово называется свободным от квадратов, если оно не содержит факторов вида uu, где u – непустое слово. Генерирование длинных свободных от квадратов слов имеет смысл только для алфавитов размером не меньше 3, потому что самые длинные свободные от квадратов слова над двухбуквенным алфавитом {a,b} – это aba и bab.

В настоящей задаче наша цель – спроектировать алгоритм генерирования длинных свободных от квадратов слов в режиме, близком к реальному времени. Алгоритм SquareFreeWord делает это помощью функции bin-parity(n), возвращающей четность (0 – четное, 1 – нечетное) числа единиц в двоичном представлении натурального числа n. Задержка между вычислением двух последовательных результатов пропорциональна времени вычисления этой функции.

```
SquareFreeWord()
 1  prev ← 0
 2  for n ← 1 to ∞ do
 3      ▷ prev = max{i : i < n and bin-parity(i) = 0}
 4      if bin-parity(n) = 0 then
 5          вывести (n − prev)
 6          prev ← n
```

Сгенерированное слово α начинается так: 3 2 1 3 1 2 3 2 1 2 3 1 . . . .

Вопрос. Показать, что алгоритм SquareFreeWord строит сколь угодно длинные свободные от квадратов слова над троичным алфавитом {1,2,3}.

[**Указание:** условие в строке 4 выполняется, только когда n – позиция вхождения a слова Туэ–Морса **t**.]

Решение

Вопрос тесно связан со свободой слова Туэ–Морса **t** от перекрытий (оно не содержит факторов вида *cucuc*, где *c* – буква, а *u* – слово). Выполнение алгоритма SquareFreeWord для n вплоть до 18 порождает слово 321312321. Сопоставляя букву a позиции n, если условие в строке 4 выполняется, и букву b в противном случае, мы получаем показанную ниже таблицу, в которой каждое выходное значение в нижней строке сопоставлено позиции *prev* – предыдущей позиции, в которой условие выполняется (для текущего значения n).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a	b	a	a
3			2		1	3			1	2		3			2		1	

В алгоритме используется следующее определение **t**: **t**$[n]$ = a тогда и только тогда, когда bin-parity(n) = 0. Легко видеть, что оно эквивалентно другим определениям **t**, данным в главе 1.

Слово α. Доказательство свободы от квадратов слова α, вычисляемого алгоритмом SquareFreeWord, опирается на тот факт, что **t** свободно от перекрытий.

Пусть τ: {1,2,3}* → {a,b}* – морфизм, определенный следующим образом: τ(1) = a, τ(2) = ab и τ(3) = abb. Заметим, что **t** единственным образом разлагается в произведение, в котором суффикс принадлежит множеству {a,ab,abb}. Алгоритм SquareFreeWord выводит i, когда в **t** виртуально обнаруживается фактор τ(i).

Предположим, что доказываемое утверждение неверно, т. е. выходное слово содержит непустой квадратный фактор uu. Тогда в **t** встретится τ(uu). Но поскольку u = av для некоторого слова v и за вхождением τ(uu) сразу следует буква a, **t** содержит перекрытие, чего быть не может. Следовательно, выходное слово α свободно от квадратов.

Заметим, что $a = h^\infty(3)$, где морфизм h, аналогичный f из задачи 79, определен следующим образом: $h(3) = 321$, $h(2) = 31$ и $h(1) = 2$.

Примечания

Доказательство того, что слово Туэ–Морса свободно от перекрытий, можно найти в книге [175, глава 2]. Там же приведены доказательства комбинаторных свойств, из которых вытекает правильность алгоритма SquareFreeWord.

Мы покажем три других построения свободных от квадратов бесконечных слов β, γ, δ, опуская доказательства.

- β$[i]$ = c, если **t**$[i]$ = **t**$[i+1]$, и β$[i]$ = **t**$[i]$ в противном случае.
- γ$[i]$ = c, если **t**$[i-1]$ = **t**$[i]$, и γ$[i]$ = **t**$[i]$ в противном случае.
- δ[0] = 0, и для $n > 0$ δ$[n]$ равно

  $\min\{k \geq 0 : k \neq \delta[\lfloor n/2 \rfloor]$ и $\delta[0..n-1]\cdot k$ свободно от квадратов$\}$.

Слово δ можно вычислить следующим образом:

if $h(n) = 1$ **then** $\delta[n] = 0$
else if bin-parity$(n) = 1$ **then** $\delta[n] = 1$
else $\delta[n] = 2$,

где для $n > 0$ $h(n)$ – четность длины блока нулей в конце двоичного представления n.

Несмотря на различные построения, все четыре слова α, β, γ и δ по существу одинаковы (с точностью до переименования букв и в некоторых случаях удаления первой буквы). Известно, что количество свободных от квадратов слов длины n над троичным алфавитом, $sqf(n)$, растет экспоненциально вместе с n; доказательство этого факта впервые приведено в работе [42], а затем улучшено несколькими авторами. Первые значения $sqf(n)$ приведены в таблице ниже.

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$sqf(n)$	3	6	12	18	30	42	60	78	108	144	204	264	342

n	14	15	16	17	18	19	20	21	22	23	24
$sqf(n)$	456	618	798	1044	1392	1830	2388	3180	4146	5418	7032

С другой стороны, количество свободных от перекрытий двоичных слов длины n над двоичным алфавитом растет всего лишь полиномиально, как показано в работе Restivo and Salemi [207] (см. задачу 75).

81. Проверка свободы морфизма от квадратов

Свободными от квадратов морфизмами называются морфизмы, сохраняющие свойство свободы от квадратов. Такие морфизмы дают полезный метод итеративного генерирования слов, свободных от квадратов. В этой задаче наша цель – дать эффективную характеристику морфизмов, свободных от квадратов, а именно тест, требующий времени, линейно зависящего от длины морфизма над алфавитом фиксированной длины.

Свободный от квадратов морфизм h обладает свойством: слово $h(x)$ свободно от квадратов, если таковым является x. Говорят также, что h является k-свободным от квадратов, если это условие выполняется для $|x| \le k$. В общем случае из k-свободы от квадратов не следует свобода от квадратов. Например, h_1 – кратчайший свободный от квадратов морфизм из $\{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ в себя, но h_2: $\{\mathtt{a},\mathtt{b},\mathtt{c}\}^* \to \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\}^*$ не свободен от квадратов, хотя является 4-свободным от квадратов.

$$\begin{cases} h_1(\mathtt{a}) = \mathtt{abcab} \\ h_1(\mathtt{b}) = \mathtt{acabcb} \\ h_1(\mathtt{c}) = \mathtt{acbcacb} \end{cases}; \qquad \begin{cases} h_2(\mathtt{a}) = \mathtt{deabcbda} \\ h_2(\mathtt{b}) = \mathtt{b} \\ h_2(\mathtt{c}) = \mathtt{c} \end{cases}.$$

Следующая характеристика основана на понятии предквадрата. Пусть z – фактор $h(a)$, $a \in A$. Его вхождение в позиции i называется предквадратом, если существует слово y, для которого ay (соответственно ya) свободно от квадратов и z^2 входит в $h(ay)$ в позиции i (соответственно в $h(ya)$ с центром i). Ясно, что если какое-то $h(a)$ имеет предквадрат, то h не является морфизмом, свободным от квадратов. Обратное верно при дополнительном условии.

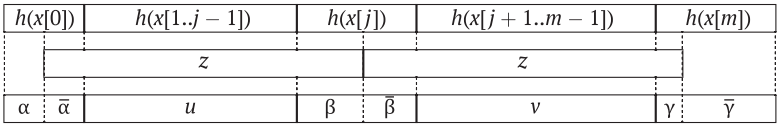
Вопрос. Показать, что морфизм h свободен от квадратов тогда и только тогда, когда он 3-свободен от квадратов и ни одно слово $h(a)$, $a \in A$, не содержит фактора-предквадрата.

[**Указание:** рассмотрите различные случаи, пользуясь приведенным ниже рисунком, на котором показано расположение квадрата z^2 в $h(x)$, где $x = x[0..m]$.]

Вопрос. Показать, что для равномерных морфизмов из 3-свободы от квадратов вытекает свобода от квадратов и что для морфизмов, определенных на 3-буквенном алфавите, из 5-свободы от квадратов вытекает свобода от квадратов.

Решение

Условие на предквадраты. Чтобы доказать утверждение из первого вопроса, нужно только показать, что несвободный от квадратов морфизм нарушает одно из двух условий, потому что обратное очевидно.



Пусть $x = x[0..m]$ – слово, для которого $h(x)$ содержит квадрат z^2. Отбрасывая при необходимости буквы в обоих концах x, мы можем предполагать, что вхождение z^2 начинается в $h(x[0])$ и заканчивается в $h(x[m])$ (см. рисунок).

Заметим, что если $h(a)$ – префикс или суффикс $h(b)$, $a \neq b$, то морфизм не является даже 2-свободным от квадратов. Поэтому можно предположить, что $\{h(a) : a \in A\}$ является (однозначно декодируемым) префиксным и суффиксным кодом.

Пусть α, $\bar{\alpha}$, β, $\bar{\beta}$, γ и $\bar{\gamma}$ – слова, определенные так, как показано на рисунке.

Во-первых, если $\bar{\alpha} = \bar{\beta}$, то, в силу префиксности кода, $x[1..j - 1] = x[j + 1..m - 1]$ и, значит, $\beta = \gamma$. Так как x свободно от квадратов, $x[0] \neq x[j]$ и $x[j] \neq x[m]$. Таким образом, $x[0]x[j]x[m]$ свободно от квадратов, но $h(x[0]x[j]x[m])$ содержит $(\bar{\alpha}\beta)^2$, так что h не является 3-свободным от квадратов.

В последующих случаях можно без ограничения общности предположить, что $\bar{\alpha}\delta = \bar{\beta}$ для $\delta \neq \varepsilon$.

Во-вторых, если $x[1] = x[j]$, то пусть i – наименьший индекс, для которого δ является префиксом $h(x[1..i])$. Тогда $x[j]x[1..i]$ свободно от квадратов, но $h(x[j]x[1..i])$ содержит δ^2, так что в $h(x[j])$ существует предквадрат.

В-третьих, если $x[1] = x[j]$, то $h(x[j])$ следует за $\bar{\alpha}$ в z, и тогда $h(x[j\,..m])$ начинается с $(\beta\bar{\alpha})^2$, так что в $h(x[j])$ существует предквадрат.

Для завершения доказательства нужно точно так же рассмотреть симметричные случаи.

Равномерный морфизм. Если морфизм равномерный, то достаточно заметить, что условие на предквадраты из первого утверждения эквивалентно свойству 2-свободы от квадратов, которое следует из условия 3-свободы от квадратов.

Морфизмы из 3-буквенного алфавита. Пусть A – 3-буквенный алфавит. Предположим, что в $h(a)$, $a \in A$, существует предквадрат и что y продолжает этот предквадрат до квадрата в $h(ay)$. Мы можем при необходимости отбросить некоторый суффикс y, и тогда буква a сможет повторно встретиться только как последняя буква y. Поскольку слово ay свободно от квадратов в 3-буквенном алфавите, ya^{-1} свободно от квадратов в 2-буквенном алфавите, откуда следует, что его длина не больше 3. Поэтому из условия 5-свободы h от квадратов следует, что $|ay| \leq 5$.

Пример h_2 показывает, что 5 – оптимальная граница.

Примечания

Свободный от квадратов морфизм h представляет собой интересный инструмент для генерирования бесконечных свободных от квадратов слов: если $h(a)$ имеет вид ay для некоторой буквы a и непустого слова y, то итеративное применение h к a порождает свободное от квадратов слово $h^\infty(a)$. Заметим, однако, что морфизм f из задачи 79 не является свободным от квадратов, но тем не менее слово $f^\infty(\mathtt{a})$ свободно от квадратов. Дополнительные результаты, полученные Берстелем и Ройтенауэром, приведены в книге Lothaire [175, глава 2]; см. также [35].

Полное доказательство первого утверждения имеется в работе [65], где приведены также некоторые следствия из этого результата.

82. Число квадратных факторов в помеченных деревьях

Известно, что количество различных квадратных факторов в заданном слове линейно зависит от длины слова (см. задачу 72). К сожалению, этим свойством не обладают реберно-помеченные деревья.

В этой задаче продемонстрирована удивительная нижняя оценка, основанная на сравнительно простых примерах деревьев.

Вопрос. Доказать, что реберно-помеченное двоичное дерево размера n может содержать $\Omega(n^{4/3})$ различных квадратных факторов.

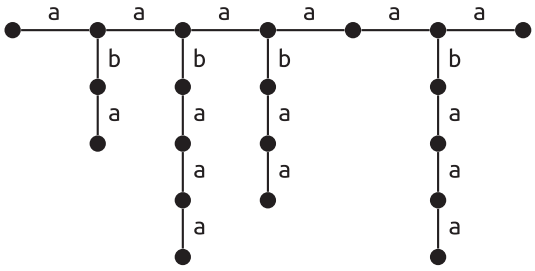
[**Указание:** рассмотрите гребенчатые деревья.]

Решение

Обозначим sq(T) число квадратных факторов вдоль ветвей реберно-помеченного дерева T. Чтобы доказать сформулированный результат, рассмотрим специальное семейство очень простых деревьев, называемых гребнями (comb), для которых достигается асимптотически максимально возможное число квадратов.

Гребнем называется помеченное дерево, в котором имеется путь, называемый *стволом* (spine), такой, что из каждого расположенного на нем узла исходит не более одной *ветви*. Все принадлежащие стволу ребра помечены буквой a. Каждая ветвь представляет собой путь, метка которого начинается буквой b, за которой следует какое-то число букв a. На рисунке ниже гребень содержит 14 квадратных факторов:

- a^2, $(aa)^2$, $(aaa)^2$,
- все циклические сдвиги слов: $(ab)^2$, $(aab)^2$ и $(aaab)^2$,
- квадраты $(abaaa)^2$ и $(aaaba)^2$.



Мы покажем, что существует семейство T_m гребней специального вида таких, что $\text{sq}(T_m) = \Omega(|T_m|^{4/3})$. Отсюда легко следует, что $\text{sq}(T) = \Omega(n^{4/3})$ для дерева T размера n.

Рассмотрим целое число $m = k^2$ и определим множество

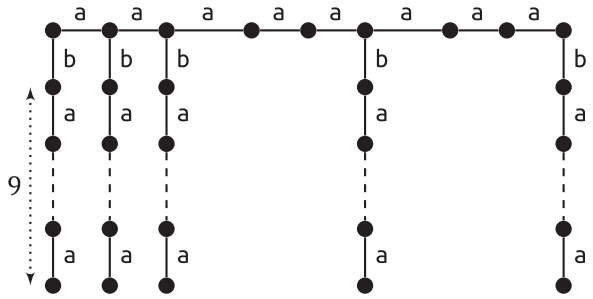
$$Z_m = \{1, \ldots, k\} \cup \{i.k : 1 \le i \le k\}.$$

Например, $Z_9 = \{1,2,3,6,9\}$.

Наблюдение. Для любого целого d, $0 < d < m$, существуют $i, j \in Z_m$ такие, что $i - j = d$.

Доказательство. Любое число d, $0 < d < m$, можно единственным способом представить в виде $p\sqrt{m} - q$, где $0 < p, q \le \sqrt{m}$. Выбирая $i = p\sqrt{m}$ и $j = q$, приходим к требуемому заключению. ■

Затем определим специальный гребень T_m: он состоит из ствола длиной $m-1$ с вершинами, пронумерованными от 1 до m, который помечен словом $\mathtt{a}^{m-1}$, и ветвей с метками $\mathtt{ba}^m$, исходящих из каждой вершины $j \in Z_m$ ствола. На рисунке ниже показан гребень T_9, ассоциированный с множеством $Z_9 = \{1,2,3,6,9\}$, со стволом и пятью ветвями.



Факт. Для каждого дерева T_m имеет место равенство $\mathrm{sq}(T_m) = \Omega(|T_m|^{4/3})$.

Доказательство. Из приведенного выше наблюдения следует, что для любого d, $0 < d < m$, на стволе существуют два узла i, j степени 3 такие, что $i - j = d$. Таким образом, T_m содержит все квадраты вида $(\mathtt{a}^i\mathtt{b}\mathtt{a}^{d-i})^2$ для $0 \le i \le d$.

Всего получается $\Omega(m^2)$ различных квадратов. Так как $m = k^2$, размер T_m, т. е. количество узлов в нем, равен $k(m+2) + (k-1)(k+m+1) = O(m\sqrt{m})$. Следовательно, число узлов в T_m равно $\Omega(|T_m|^{4/3})$. ■

Примечания

Приведенный выше результат оптимален, потому что верхняя граница числа квадратов в помеченных деревьях размера n равна $O(n^{4/3})$. Комбинаторное доказательство этой оценки гораздо труднее, его можно найти в работе [82].

83. Подсчет квадратов в гребнях за линейное время

Гребнем называется помеченное дерево, состоящее из пути, называемого *стволом*, из каждого узла которого исходит не более одной *ветви*. Все ребра, принадлежащие стволу, помечаются буквой a. Каждая ветвь является путем, метка которого начинается буквой b, за которой следует какое-то число букв a.

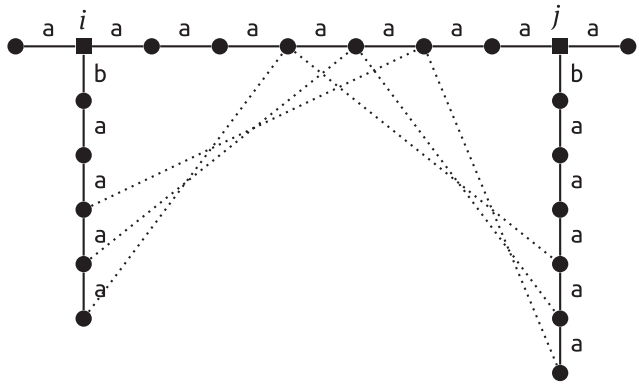
Количество различных квадратов, встречающихся на ветвях гребня T, может быть суперлинейным (см. задачу 82), но, несмотря на оценку снизу, их можно подсчитать за время, линейно зависящее от размера дерева. В этом и состоит цель данной задачи. Мы решим ее, аккуратно закодировав квадраты в соответствии с их глобальной структурой.

Вопрос. Показать, как за линейное время вычислить количество различных квадратных факторов на ветвях двоичного гребня.

Решение

Мы сосредоточимся только на неунарных квадратах, потому что вычислить количество унарных квадратов (с периодом 1) в любом помеченном дереве, очевидно, можно за линейное время.

Чтобы добиться желаемого времени работы, нам нужно будет специальным образом закодировать все квадраты. Кодировка основана на допустимых парах узлов на стволе. Пара (i, j) называется *допустимой*, если $d \leq p + q$, где d – расстояние между i и j ($|(j - i)|$), а p, q – количества вхождений a на ветвях, исходящих из i и j соответственно.



Существенная часть гребня соответствует допустимой паре узлов (i, j) на стволе, ребрам между ними и двум исходящим ветвям, метки которых начинаются буквой b. Все квадраты в каждой такой существенной части можно рассматривать как *пакет* квадратов (множество слов, сопряженных одному фактору), представленный интервалом.

На рисунке выше показана допустимая пара (i, j), для которой $d = 7$, $p = 4$ и $q = 5$. Неунарные квадраты, генерируемые существенной частью дерева, соответствующей этой паре, – это $(\mathtt{a}^2\mathtt{ba}^5)^2$, $(\mathtt{a}^3\mathtt{ba}^4)^2$ и $(\mathtt{a}^4\mathtt{ba}^3)^2$, они показаны пунктирными линиями.

Вообще, множество квадратов, соответствующих паре (i, j), имеет вид $\{\mathtt{a}^k\mathtt{ba}^k\mathtt{a}^{d-k}\mathtt{ba}^{d-k} : k \in [i', j']\}$, где $[i', j'] \subseteq [i, j]$. Это множество можно представить парой $(d, [i', j'])$. В примере выше пара (7,[2,4]) представляет множество квадратов $\{(\mathtt{a}^2\mathtt{ba}^5)^2, (\mathtt{a}^3\mathtt{ba}^4)^2$ и $(\mathtt{a}^4\mathtt{ba}^3)^2\}$.

Факт. Количество допустимых пар в гребне T линейно зависит от размера гребня.

Доказательство. Заметим, что если (i, j) – допустимая пара с расстоянием d, для которой количество букв a на ветвях, исходящих из узлов i и j, равно соответственно p и q, то $d \leq 2\max\{p, q\}$. Следовательно, для данного узла на

стволе достаточно рассмотреть те узлы на стволе, которые отстоят от него влево и вправо на расстояние, не большее k, где k – количество букв a на ветви, исходящей из этого узла.

Таким образом, общее число рассматриваемых узлов ограничено сверху суммарной длиной исходящих ветвей, которая равна $O(|T|)$. ■

Факт. Количество различных квадратных факторов в гребне T можно вычислить за линейное время.

Доказательство. Чтобы добиться линейного времени, соберем допустимые пары в группы, для которых расстояние между узлами в паре одинаково и равно d. Для каждой пары (i, j) множеству квадратов, порождаемых этой парой, соответствует интервал. Эти интервалы (для различных пар) могут пересекаться, однако вычислить объединение всех интервалов и его общий размер можно за линейное время. Результирующее множество снова является объединением интервалов, и его размер легко вычисляется. Множества квадратов, соответствующих различным группам, не пересекаются. Просуммировав размеры всех групп, мы получим окончательный результат. На этом доказательство завершается. ■

Несмотря на то что количество различных квадратов может быть суперлинейным, и это еще не считая унарных квадратов, все их можно представить в виде объединения линейного количества непересекающихся множеств вида $\{\mathtt{a}^k\mathtt{b}\mathtt{a}^k\mathtt{a}^{d-k}\mathtt{b}\mathtt{a}^{d-k} : k \in [i', j']\}$.

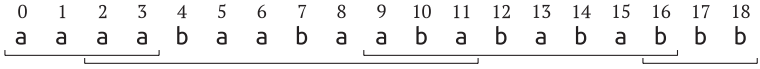
Примечания

Представленный алгоритм основан на алгоритме из работы Kociumaka et al. [164]. До сих пор неизвестно, можно ли за линейное время подсчитать квадраты в деревьях общего вида.

84. Кубические серии

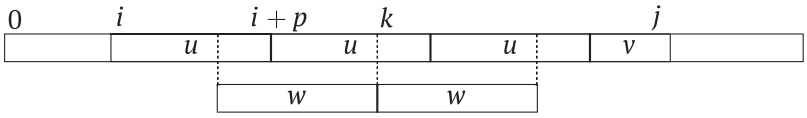
Кубические серии – частный случай серий, для которых получить оценки проще. Они включают другие типы периодических факторов в слове, но в меньшей степени.

Кубической серией в слове x называется максимальная периодическая структура в x, длина которой как минимум в три раза больше ее периода. Точнее, это интервал $[i\,..\,j]$ позиций в x, для которого фактор $u = x[i..j]$ удовлетворяет условию $|u| \geq 3per(u)$ и не расширяется ни влево, ни вправо с таким же периодом. На рисунке подчеркнуты кубические серии в слове aaaabaabaababababbb.



Мы рассматриваем отношение порядка $<$ на алфавите и определяем специальные позиции серии $[i..j]$ в x следующим образом. Пусть p – период $x[i..j]$,

а w – лексикографически наименьшее слово, сопряженное $x[i..i+p-1]$. Тогда k – специальная позиция серии, если $x[i..j]$ содержит квадрат ww с центром в k. Специальные позиции выделены на рисунке полужирным шрифтом.



Вопрос. Показать, что в кубической серии есть хотя бы одна специальная позиция и что у двух разных кубических серий не может быть общих специальных позиций.

[**Указание:** воспользуйтесь тем фактом, что наименьшее сопряженное примитивного слова – слово Линдона – свободно от границ.]

Вопрос. Показать, что количество кубических серий в слове длины n меньше $n/2$ и что для бесконечно многих n оно не меньше $0.4n$.

[**Указание:** рассмотрите обратное отношение порядка на алфавите $<^{-1}$ и подсчитайте кубические серии в словах $x^m = (u^2\mathsf{a}^3v^2\mathsf{b}^3w^2\mathsf{c}^3)^m$, где $u = \mathsf{a}^3\mathsf{b}^3$, $v = \mathsf{b}^3\mathsf{c}^3$ и $w = \mathsf{c}^3\mathsf{a}^3$.]

Решение

Существование хотя бы одной специальной позиции в любой кубической серии. Пусть $[i..j]$ – кубическая серия, $p = per(x[i..j])$ и w – наименьшее сопряженное $x[i..i+p-1]$.

Если $p = 1$, то понятно, что все позиции в серии, кроме первой, специальные, поэтому в такой кубической серии есть по меньшей мере две специальные позиции.

Если $p > 1$, то квадрат $x[i..i+2p-1]$ содержит по меньшей мере одно вхождение w, за которым сразу следует другое вхождение w в серии. Поэтому в такой кубической серии существует по меньшей мере одна специальная позиция.

У различных кубических серий нет общих специальных позиций. Предположим, что некоторая позиция k является центром вхождений ww и $w'w'$, ассоциированных с двумя разными кубическими сериями. В силу условия максимальности, серии, будучи различны, имеют разные периоды. Если, например, слово w' короче, то оно является границей w. Но поскольку w примитивно (по определению p) и является наименьшим сопряженным, то это слово Линдона, которое, как известно, свободно от границ, т. е. мы пришли к противоречию. Таким образом, две разные кубические серии не могут иметь общей специальной позиции.

Количество кубических серий меньше $n/2$**.** Мы уже видели, что в любой кубической серии с периодом 1 есть по меньшей мере две специальные по-

зиции. Для других кубических серий сначала заметим, что ассоциированный префиксный период содержит хотя бы две разные буквы. Следовательно, вторую специальную позицию можно найти, воспользовавшись обратным отношением порядка на алфавите (или наибольшим сопряженным префиксного периода), и, как показано выше, эта позиция не является специальной ни в какой другой серии.

Поскольку позиция 0 в x не может быть специальной, общее число специальных позиций в слове длины n меньше n, откуда следует, что всего существует меньше $n/2$ кубических серий.

Нижняя граница. Заметим, что для любого $m > 0$ слово x_m содержит по меньшей мере $18m - 1$ кубических серий:

$$x^1 = \mathtt{a}^3\mathtt{b}^3\mathtt{a}^3\mathtt{b}^3\ \mathtt{a}^3\ \mathtt{b}^3\mathtt{c}^3\mathtt{b}^3\mathtt{c}^3\ \mathtt{b}^3\ \mathtt{c}^3\mathtt{a}^3\mathtt{c}^3\mathtt{a}^3\ \mathtt{c}^3.$$

Действительно, имеется $15m$ кубических серий периода 1, с которыми ассоциированы факторы $\mathtt{a}^3$, $\mathtt{b}^3$ или $\mathtt{c}^3$, $2m$ кубических серий периода 6 с факторами $(\mathtt{a}^3\mathtt{b}^3)^3$ и $(\mathtt{b}^3\mathtt{c}^3)^3$ и $m - 1$ кубических серий вида $(\mathtt{c}^3\mathtt{a}^3)^3$.

Заметим, что при $m > 2$ все слово x_m образует дополнительную кубическую серию. Следовательно, в этом случае слово x_m имеет длину $45m$ и содержит по меньшей мере $18m$ кубических серий. Таким образом, при $m > 2$ количество кубических серий в слове x_m не меньше $0.4|x_m| = 0.4n$.

Примечания

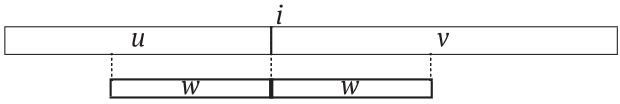
В работах [85, 86] доказаны немного улучшенные нижняя и верхняя границы числа кубических серий.

Применяя рассуждения, похожие на приведенные выше, Harju and Kärki в работе [137] ввели понятия фрейма – квадрата, корень из которого свободен от границ, – и оценили сверху и снизу количество фреймов в двоичных словах. Их оценки близки к выведенным в этой задаче.

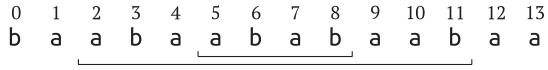
85. Короткий квадрат и локальный период

Понятие локальных периодов в словах дает более точное представление о структуре повторяемости, чем глобальный период. Это понятие лежит в основе критических позиций (см. задачу 41) и их применений.

В данной задаче перед нами стоит цель найти локальный период в заданной позиции i слова x. Локальным периодом $lper(i)$ называется период самого короткого непустого квадрата ww с центром в позиции i и, возможно, выходящего за пределы x слева, справа или с обеих сторон.



Вопрос. Показать, как вычислить все непустые квадраты с центром в позиции i слова x за время $O(|x|)$.



Для $x = \texttt{baabaababaabaa}$ квадратами с центром в позиции 7 являются $(\texttt{abaab})^2$, $(\texttt{ab})^2$ и пустой квадрат. Не существует непустого квадрата с центром в позиции 6 или 9.

Вопрос. Если существует кратчайший непустой квадрат периода p с центром в позиции i слова x, то показать, как его найти за время $O(p)$.

[**Указание:** удвойте длину области поиска.]

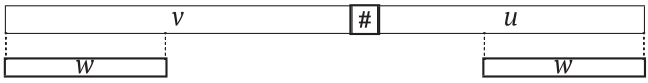
Перечислим несколько локальных периодов для приведенного выше примера: $lper(7) = 2$ – период $(\texttt{ab})^2$, $lper(1) = 3$ – период $(\texttt{aab})^2$, $lper(6) = 8$ – период $(\texttt{babaabaa})^2$ и $lper(9) = 5$ – период $(\texttt{aabab})^2$.

Вопрос. Спроектировать алгоритм вычисления локального периода p в позиции i слова x за время $O(p)$.

[**Указание:** не забудьте о ситуации, когда не существует квадрата с центром в i.]

Решение

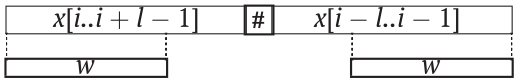
Квадраты. Положим $u = x[0..i-1]$, $v = x[i..|x|-1]$, и пусть # – буква, не встречающаяся в x.



Границы w слова $v\#u$ порождают квадраты с центром в i слова x. Если вместо $v\#u$ взять простую конкатенацию vu, то только ее границы не длиннее $\min\{|u|, |v|\}$ будут порождать искомые квадраты.

Таким образом, квадраты можно найти с помощью таблицы границ слова $v\#u$, для вычисления которой требуется линейное время (см. задачу 19), как и для всего процесса.

Кратчайший квадрат. Если задана длина l, $0 < l \le \min\{|u|, |v|\}$, то квадрат периода p, не большего l, можно найти, как описано выше, рассматривая границы слова $x[i..i+l-1]\#x[i-l..i-1]$ вместо $v\#u$. Изменяя l от 1 до не более чем $4p$, мы сможем найти квадрат периода p.

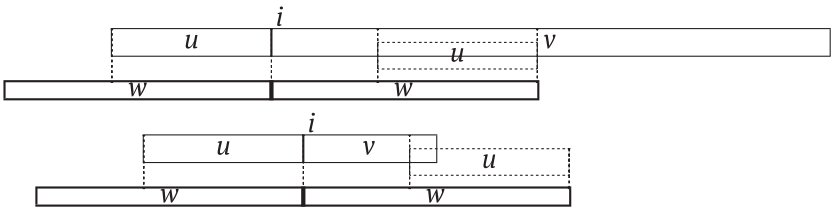


Весь процесс занимает время

$O(\sum\{l : l = 1,2,4,\ldots,2^e$, где $p \le 2^e < 2p\}) = O(p)$.

Локальный период. Если существует непустой квадрат с центром в i, то локальный период в позиции i равен периоду самого короткого такого квадрата.

Если не существует непустого квадрата с центром в i, то квадрат ww, период которого равен локальному периоду в позиции i, может выходить за пределы слова слева, справа или с обеих сторон. На рисунке ниже изображена ситуация, когда квадрат выходит за левый край.



Чтобы обнаружить выход за левый край, в v производится онлайновый поиск u, например, с помощью алгоритма КМП, результат которого следует модифицировать с учетом ситуации, изображенной на нижнем рисунке. В результате будет найден потенциальный локальный период p_1. Симметричная проверка выхода за правый край путем поиска v^R в u^R дает еще один потенциальный локальный период p_2. Окончательный локальный период равен минимуму из обеих величин.

Все вычисление занимает время $O(p)$, что нам и требуется.

Примечания

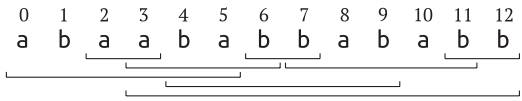
Вычисление таблицы границ обсуждается в задаче 19 и применимо к алфавитам общего вида, как и алгоритм КМП, описанный в задаче 26. См. также книги по стрингологии [74, 96, 98, 134, 228] и другие учебники по алгоритмам.

Для алфавитов фиксированного размера вычислить все локальные периоды слова можно за линейное время [106] с применением факторизации слова, напоминающей факторизацию Лемпеля–Зива (см. главу 6).

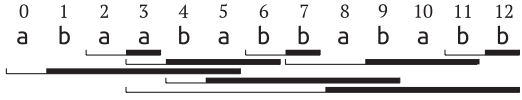
86. Число серий

Серия – это максимальная периодическая структура в слове. Формально серией в слове x называется интервал $[i..j]$ позиций x, для которых фактор $x[i..j]$ является периодическим (т. е. его наименьший период удовлетворяет усло-

вию $2p \le |x[i..j]| = (j - i + 1)$), и эта периодичность не расширяется ни влево, ни вправо (т. е. факторы $x[i-1..j]$ и $x[i..j+1]$, если они определены, имеют большие периоды). На рисунке ниже все восемь серий подчеркнуты.



Мы будем рассматривать отношение порядка на алфавите слов и соответствующий лексикографический порядок, который тоже обозначается $<$. Также будем рассматривать лексикографический порядок $\lesssim$, называемый обратным, индуцированный обратным отношением порядка на алфавите $<^{-1}$. С каждой серией $[i..j]$ ассоциируется ее наибольший суффикс в смысле одного из этих двух упорядочений, который определяется следующим образом. Пусть $p = per(x[i..j])$. Если $j + 1 < n$ и $x[j+1] > x[j-p+1]$, то сопоставим серии позицию k, для которой $x[k..j]$ – наибольший собственный суффикс $x[i..j]$ в смысле порядка $<$. В противном случае k – начальная позиция наибольшего собственного суффикса $x[i..j]$ в смысле порядка $\lesssim$. Позиция k, определенная описанным образом, называется **специальной позицией** в серии. Эти позиции, тесно связанные со словами Линдона (определенными в главе 1), являются предметом первого вопроса. Жирными линиями ниже показаны наибольшие суффиксы, ассоциированные с сериями в слове abaababbababb.



Вопрос. Показать, что если специальная позиция k в серии с периодом p определена в смысле порядка $\lesssim$ (соответственно $<$), то $x[k..k+p-1]$ – самый длинный фактор Линдона слова x, начинающийся в позиции k в смысле порядка $<$ (соответственно $\lesssim$).

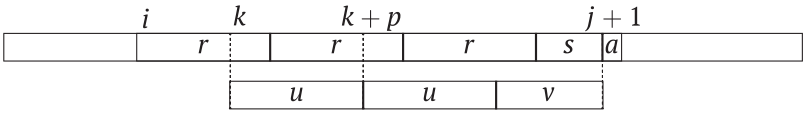
[**Указание:** специальная позиция k серии $[i..j]$ с периодом p удовлетворяет условию $k \le i + p$; см. задачу 40.]

Вопрос. Показать, что две различные серии не имеют общей специальной позиции, и вывести отсюда, что число серий в слове меньше его длины.

Решение

Специальная позиция. Пусть $[i..j]$ – серия с периодом p со специальной позицией k. Чтобы ответить на первый вопрос, заметим, что $x[k..k+p-1]$ – слово Линдона, потому что оно меньше всех своих собственных суффиксов в смысле порядка $<$. Рассмотрим более длинный фактор $x[k..j']$, где $k + p \le j' \le j$. Он имеет период p, меньший его длины; иначе говоря, он не

является свободным от границ, а значит, не может быть словом Линдона ни при каком из двух порядков.



Больше ничего не нужно доказывать, если $j + 1 = |x|$. Поэтому предположим, что $j' > j$, и пусть $a = x[j + 1]$. На рисунке показан наибольший суффикс $x[i..j]$ в смысле порядка $\lesssim$, т. е. $x[k..j] = u^e v$ с периодом $|u|$, а v – собственный префикс u. Так как $x[j + 1] < x[j - p + 1]$, имеем $x[k + p..j + 1] < x[k..j - p + 1]$, откуда следует, что $x[k + p..j'] < x[k..j']$, и, значит, $x[k..j']$ не является словом Линдона в смысле порядка $<$.

Таким образом, $x[k..k + p - 1]$ – самый длинный фактор Линдона слова x, начинающийся в позиции k. Заметим, что роли обоих порядков абсолютно симметричны.

Число серий. Предположим противное – что две серии имеют общую специальную позицию k. В силу доказанного выше результата позицию нельзя определить с одним и тем же упорядочением для обеих серий. А при определении с помощью двух разных упорядочений есть единственная возможность – только одна серия имеет период 1. Но тогда $x[k - 1] = x[k]$, что невозможно для специальной позиции k неунарной серии.

Поскольку позиция 0 не может быть специальной, в слове длины n может существовать не более $n - 1$ специальных позиций серий. Тогда из предыдущего результата следует, что число серий меньше n, что и требовалось доказать.

Примечания

Понятие серии, или, как ее еще называют, максимальной периодической структуры или максимального вхождения повторения, сформулировано в работе Iliopoulos et al. [149], посвященной анализу повторений в словах Фибоначчи. Оно было введено, чтобы лаконично представить все вхождения повторений в слово. Известно, что в слове длины n существует только $O(n)$ серий, это было доказано в работе Kolpakov and Kucherov [167] неконструктивным способом.

Первая явная граница была найдена позже в работе Rytter [214]. Несколько улучшений верхней границы можно найти в работах [77, 80, 102, 203]. Колпаков и Кучеров предположили, что это число на самом деле меньше n, и впоследствии это было доказано в работе Bannai et al. [26]. Приведенное выше, очень похожее доказательство взято из работы [91]. В работе Fischer et al. [116] доказана более точная верхняя граница числа серий, $22n/23$.

Заметим, что в наших обозначениях если $k + 2p \leq j$, то $k + p$ тоже можно рассматривать как специальную позицию с таким же свойством. В частности, серия, для которой ассоциированное слово начинается кубом, имеет по меньшей мере две специальные позиции. Это дает верхнюю границу $n/2$ для

максимального числа кубических серий в слове длины n (см. задачу 84 и дополнительную информацию в работах [25] и [86]).

87. Вычислений серий над отсортированным алфавитом

В этой задаче наша цель – показать, что все серии (максимальные периодические структуры) в слове над алфавитом, допускающим линейную сортировку, можно вычислить за линейное время.

Решение основано на результатах из задачи 86, где показано, что в серии имеется специальная позиция, с которой начинается самый длинный фактор Линдона всего слова. Искать наибольшие слова Линдона можно в смысле отношения порядка $<$ на алфавите, а также в смысле обратного к нему отношения $<^{-1}$. Если найден самый длинный фактор Линдона, то простое расширение вправо и влево (как в задаче 74) может подтвердить, что начальная позиция фактора Линдона является специальной позицией серии, период которой равен длине фактора Линдона.

Для этого мы сначала определим ***таблицу Линдона*** Lyn непустого слова x. Для позиции i в слове x, $i = 0, \ldots, |x| - 1$, $Lyn[i]$ равно длине самого длинного фактора Линдона, начинающегося в i:

$Lyn[i] = \max\{l : x[i..i + l - 1]$ является словом Линдона$\}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$	a	b	b	a	b	a	b	a	a	b	a	b	b	a	b	a
$Lyn[i]$	3	1	1	2	1	2	1	8	5	1	3	1	1	2	1	1

Вопрос. Показать, что алгоритм LongestLyndon правильно вычисляет таблицу Lyn.

```
LongestLyndon(непустое слово x длины n)
 1  for i ← n − 1 downto 0 do
 2      (Lyn[i], j) ← (1, i + 1)
 3      while j < n и x[i..j − 1] < x[j..j + Lyn[j] − 1] do
 4          (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
 5  return Lyn
```

Вопрос. Обобщить алгоритм LongestLyndon так, чтобы он вычислял все серии, встречающиеся в слове.

[**Указание:** используйте наибольшие общие продолжения, как в задаче 74.]

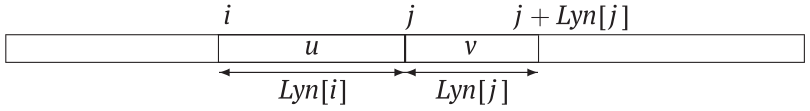
Вопрос. Каково полное время работы алгоритма, если сравнение двух факторов производится с помощью рангов ассоциированных с ними суффиксов в лексикографическом порядке и применяется метод наибольшего общего продолжения?

Решение

Доказательства опираются на следующие хорошо известные свойства слов Линдона, которые можно найти в книге [175]. Во-первых, если u и v – слова Линдона и $u < v$, то uv – также слово Линдона (и $u < uv < v$). Во-вторых, любое непустое слово единственным способом разлагается в произведение $u_0u_1u_2\ldots$, где каждое u_i – слово Линдона и $u_0 \geq u_1 \geq u_2 \geq \ldots$. Кроме того, u_0 – самый длинный префикс Линдона слова u. Эту факторизацию можно вычислить с помощью таблицы Lyn слова, позволяющей переходить от одного фактора к следующему. Но таблица содержит больше информации, чем факторизация.

Слово abbababaabbbaba разлагается в произведение abb·ab·ab·aabbbab·a, соответствующее подпоследовательности 3, 2, 2, 8, 1 значений в его таблице Линдона.

Правильность алгоритма LongestLyndon. Инвариант цикла for обработки позиций i имеет вид: $Lyn[k]$ вычислено для $k = i + 1, \ldots, n - 1$ и $u[i + 1..j - 1] \cdot u[j..j + Lyn[j] - 1]\ldots$, где $j = i + 1 + Lyn[i + 1]$ – факторизация Линдона фактора $x[i + 1..n - 1]$.



Текущий фактор u, начинающийся в позиции i, первоначально равный $x[i]$, сравнивается со следующим фактором v. Если $u < v$, то u заменяется на uv, и сравнение продолжается с фактором, следующим за uv. Цикл while останавливается, когда текущий фактор u оказывается не меньше следующего за ним. Ясно, что в момент остановки цикла u является самым длинным фактором Линдона, начинающимся в позиции i, и, значит, $Lyn[i] = |u|$. Также ясно, что мы получили факторизацию Линдона слова $x[i..n - 1]$, чем доказывается правильность инварианта и алгоритма LongestLyndon.

Вычисление серий. Чтобы вычислить все серии в слове x, мы просто проверяем для каждой позиции i, является ли она специальной позицией серии с периодом $[i..i + Lyn[i] - 1]$. Для этого нужно вычислить длины l и r наибольших общих расширений (LCE) периода влево и вправо и проверить, выполняется ли неравенство $l + r \geq Lyn[i]$. Если да, то алгоритм сообщает о найденной серии.

```
RUNS(непустое слово x длины n)
1  for i ← n − 1 downto 0 do
2      (Lyn[i], j) ← (1, i + 1)
3      while j < n и x[i..j − 1] < x[j..j + Lyn[j] − 1] do
4          (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
5      l ← |lcs(x[0..i − 1], x[0..i + Lyn[i] − 1])|
6      r ← |lcp(x[i..|x| − 1], x[i + Lyn[i]..|x| − 1])|
7      if l + r ≥ Lyn[i] then
8          вывести серию [i − l..i + Lyn[i] + r − 1]
```

Точнее, l – длина самого длинного общего суффикса $x[0..i-1]$ и $x[0..i+Lyn[i]-1]$, а r – длина самого длинного общего префикса $x[i..|x|-1]$ и $x[i+Lyn[i]..|x|-1]$. Они устанавливаются равными нулю, если $i=0$ и $i+Lyn[i]=n$ соответственно.

Описанный выше процесс следует повторить для порядка $\lesssim$, индуцированного обратным отношением порядка на алфавите.

Время работы алгоритма RUNS. Сначала заметим, что количество сравнений слов в строке 3 алгоритма RUNS меньше $2|x|$. Действительно, на каждой итерации может быть не более одного неуспешного сравнения. И не более $|x|$ успешных сравнений, потому что после каждого число факторов в факторизации Линдона слова x уменьшается. Поэтому, чтобы получить алгоритм с линейным временем работы, мы должны обсудить, как сравниваются слова и как вычисляется LCE.

Сравнение слов в строке 3 можно реализовать с помощью рангов суффиксов благодаря следующему свойству.

Свойство. Пусть u – слово Линдона и $v \cdot v_1 \cdot v_2 \ldots v_m$ – факторизация Линдона слова w. Тогда $u < v$ тогда и только тогда, когда $uw < w$.

Доказательство. Предположим, что $u < v$. Если $u \ll v$, то $uw \ll vv_1v_2 \ldots v_m = w$. В противном случае u – собственный префикс v. Обозначим $e > 0$ наибольшее целое число такое, что $v = u^e z$. Так как v – слово Линдона, z не пусто, и мы имеем $u^e < z$. Так как ни u не является префиксом z (по определению e), ни z не является префиксом u (поскольку v свободно от границ), имеем $u \ll z$. Отсюда следует, что $u^{e+1} \ll u^e z = v$ и, значит, $uw < w$.

Обратно, предположим, что $v \leq u$. Если $v \ll u$, то, очевидно, имеем $w < uw$. Остается рассмотреть случай, когда v – префикс u. Если это собственный префикс, то u имеет вид vz, где слово z не пусто. Имеем $v < z$, потому что u – слово Линдона. Слово z не может быть префиксом $t = v_1v_2 \ldots v_m$, потому что тогда v не было бы самым длинным префиксом Линдона слова w, что противоречит свойству факторизации. Таким образом, либо $t \leq z$, либо $z \ll t$. В первом случае если t – префикс z, то $w = vt$ – префикс u и, следовательно, uw, т. е. $w < uw$. Во втором случае для некоторого суффикса z' слова z и некоторого фактора v_k слова t имеем $z' \ll v_k$. Из факторизации следует, что $v_k \leq v$. Поэтому суффикс z' слова u меньше его префикса v, что противоречит тому, что u является словом Линдона. ■

Для каждой начальной позиции i суффикса слова x, $i = 0, \ldots, |x| - 1$, обозначим Rank[i] ранг суффикса $x[i..|x| - 1]$ в лексикографически упорядоченном списке всех непустых суффиксов x (ранги принимают значения от 0 до $|x| - 1$).

В силу сформулированного выше свойства неравенство $x[i..j - 1] < x[j..j + Lyn[j] - 1]$ в строке 3 обоих предыдущих алгоритмов можно переписать в виде Rank[i] < Rank[j].

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$	a	b	b	a	b	a	b	a	a	b	a	b	b	a	b	a
$Lyn[i]$	3	1	1	2	1	2	1	8	5	1	3	1	1	2	1	1
$Rank[i]$	7	15	12	4	11	3	9	1	5	13	6	14	10	2	8	0

Заметим, что факторизацию Линдона можно восстановить, пробежавшись по самой длинной убывающей последовательности рангов, начиная с первого ранга. В примере выше это последовательность (7,4,3,1,0), которая соответствует позициям (0,3,5,7,15) слова x и его факторизации abb · ab · ab · aababbab · a.

Что касается времени работы, после того как таблица рангов суффиксов Rank уже вычислена, то сравнение слов в строке 3 можно реализовать за постоянное время. Известно, что таблицу рангов, обращение отсортированного списка суффиксов (суффиксного массива), можно вычислить за линейное время в предположении, что алфавит допускает линейную сортировку.

Инструкции в строках 5–6 можно выполнить как LCE-запросы, поэтому для их вычисления требуется постоянное время после занимающей линейное время предобработки при том же предположении (см., например, [115]). Таким образом, весь алгоритм Runs работает за линейное время, если алфавит допускает линейную сортировку.

Примечания

Алгоритм LongestLyndon можно немного модифицировать, так чтобы он вычислял лес Линдона заданного слова. Лес состоит из списка деревьев Линдона, соответствующих факторам, входящим в факторизацию Линдона данного слова.

Лес Линдона слова Линдона рекурсивно ассоциируется с (правой) стандартной факторизацией слова Линдона w, не сводимого к одной букве: w можно записать в виде uv, где v определяется либо как наименьший собственный непустой суффикс w, либо как наибольший собственный суффикс Линдона w, что на самом деле одно и то же. Тогда слово u также является словом Линдона и $u < v$ (см. [175]).

Показано, что структура дерева Линдона такая же, как структура декартова дерева рангов суффиксов (см. Hohlweg and Reutenauer [142]). Алгоритм LongestLyndon работает, как построение декартова дерева справа налево (см. https://en.wikipedia.org/wiki/Cartesian_tree).

Связь между суффиксными массивами и факторизациями Линдона изучалась в работе Mantaci et al. [184].

В работе Franek et al. [119] представлено несколько алгоритмов вычисления таблицы Линдона.

Читатель может обратиться к обзору Fischer and Heun [115], касающемуся LCE-запросов. Более сложные методы их реализации над алфавитом общего

вида и вычисления серий можно найти в работах [83, 128] и содержащихся в них ссылках.

88. Периодичность и факторная сложность

Свойство, сформулированное в этой задаче, дает полезное условие для обнаружения периодичности в бесконечных словах.

Говорят, что бесконечное слово x (индексы пробегают множество натуральных чисел) является периодическим в конечном счете, или просто u-периодическим, если его можно записать в виде yz^∞ для некоторых (конечных) слов y и z, $z \neq \varepsilon$.

Обозначим $F_x(n)$ количество различных факторов длины n, встречающихся в бесконечном слове x. Функция F_x называется факторной сложностью x.

Вопрос. Показать, что бесконечное слово x является u-периодическим тогда и только тогда, когда F_x ограничена сверху постоянной.

Решение

Если слово x u-периодическое, то его можно записать в виде yz^∞, где z примитивное и либо y пустое, либо y и z заканчиваются разными буквами. При таком нормализованном представлении x мы имеем $F_x(n) = |yz|$ для любой длины $n \geq |yz|$, откуда следует, что F_x ограничено сверху постоянной.

Обратно, предположим, что F_x ограничено сверху целочисленной постоянной $m > 0$. Так как $F_x(l) \leq F_x(l+1)$ для любой длины l, то отсюда следует, что $F_x(n) = F_x(n+1)$ для некоторой длины n. Это означает, что за каждым вхождением любого фактора v длины n в x следует одна и та же буква b_v. Следовательно, мы можем рассмотреть факторную функцию *next*, определенную для непустых факторов u длины $n+1$ следующим образом: $next(u) = vb_v$, где $u = av$ для некоторой буквы a.

Пуст w – префикс длины n бесконечного слова x. Существуют p и s такие, что $next^s(w) = next^{s+p}(w)$, т. к. количество факторов длины n конечно. Таким образом, x является u-периодической с периодом p, начинающимся в позиции s. Что и требовалось доказать.

Примечания

u-периодичность слова x эквивалентна условию $F_x(n) \leq n$ для некоторой длины n.

Множество граничных бесконечных слов x, для которых $F_x(n) = n+1$ при любом n, называется множеством бесконечных слов Штурма. Это бесконечные слова, не являющиеся u-периодическими, с минимальной факторной сложностью. В частности, таким свойством обладает бесконечное слово Фибоначчи.

Дополнительную информацию по этой теме можно почерпнуть из книги Allouche and Shallit [7] и из пособия Berstel and Karhumäki [34].

89. Периодичность морфических слов

В этой задаче показано, что есть возможность проверить, является ли периодическим бесконечное слово, порожденное (конечным) морфизмом.

Бесконечное морфическое слово получается путем итеративного применения морфизма θ из A^+ в себя, где $A = \{\mathtt{a},\mathtt{b},\dots\}$ – конечный алфавит. Предположим, что θ продолжаем над буквой $\mathtt{a}$, т. е. $\theta(\mathtt{a}) = \mathtt{a}u$ для $u \in A^+$. Тогда $\Theta = \theta^\infty(\mathtt{a})$ существует и равно $\mathtt{a}u\theta(u)\theta^2(u)\dots$. Бесконечное слово является неподвижной точкой θ, т. е. $\theta(\Theta) = \Theta$.

Бесконечное слово Θ периодическое, если его можно записать как z^∞ для некоторого (конечного) слова $z \neq \varepsilon$.

Чтобы избежать ненужных осложнений, будем предполагать, что морфизм θ неприводимый, т. е. любая буква достижима из любой буквы (для любых c, $d \in A$ буква d встречается в $\theta^k(c)$ для некоторого k), и элементарный, т. е. не является произведением $\eta \circ \zeta$ двух морфизмов $\zeta : A^+ \to B^+$ и $\eta : B^+ \to A^+$, где B – алфавит, меньший, чем A. Из второго условия следует, что θ инъективен на A^* и на A^∞.

Вопрос. Для неприводимого и элементарного морфизма θ, продолжаемого над буквой $\mathtt{a}$, спроектировать алгоритм, который проверяет, верно ли, что $\Theta = \theta^\infty(\mathtt{a})$ периодическое, и работает за время $O(\sum\{|\theta(b)| : b \in A\})$.

[**Указание:** Θ периодическое тогда и только тогда, когда в нем нет биспециальных букв, т. е. за вхождением каждой буквы в Θ всегда следует одна и та же буква.]

Морфизм ρ, определенный как $\rho(\mathtt{a}) = \mathtt{ab}$, $\rho(\mathtt{b}) = \mathtt{ca}$ и $\rho(\mathtt{c}) = \mathtt{bc}$, отвечает этим условиям и порождает периодическое слово $\rho^\infty(\mathtt{a}) = \mathtt{abcabcabc}\ \dots = (\mathtt{abc})^\infty$. Ни одна буква не является биспециальной.

С другой стороны, морфизм Фибоначчи φ, определенный как $\varphi(\mathtt{a}) = \mathtt{ab}$ и $\varphi(\mathtt{b}) = \mathtt{a}$, также удовлетворяет условиям, но порождает слово Фибоначчи $\varphi^\infty(\mathtt{a}) = \mathtt{abaababa}\dots$, не являющееся периодическим в конечном счете. В нем буква $\mathtt{a}$ биспециальная, потому что за ее вхождениями следует $\mathtt{a}$ или $\mathtt{b}$, тогда как за любым вхождением буквы $\mathtt{b}$ всегда следует $\mathtt{a}$.

Решение

Алгоритм решения основан на комбинаторном свойстве: Θ периодическое тогда и только тогда, когда в нем нет биспециальных букв. Интуитивно это означает, что если Θ имеет бесконечно много биспециальных факторов, то его факторная сложность не ограничена и оно не является периодическим в конечном счете (см. задачу 88).

Если это условие выполняется, т. е. если Θ не содержит биспециальных букв, то каждая буква полностью определена буквой, находящейся перед ней. А так как в Θ встречаются все буквы алфавита, периодическое слово соответствует перестановке алфавита. Таким образом, период равен $|A|$.

Обратно, предположим, что Θ содержит биспециальную букву, и докажем, что оно непериодическое.

Пусть b – биспециальная буква, т. е. в Θ встречаются bc и bd, где буквы c и d различны. В силу неприводимости θ, буква $\mathtt{a}$ встречается в $\theta^k(b)$ для некоторого k. Поскольку θ инъективен, $\theta^k(bc) \neq \theta^k(bd)$. Пусть i и j – начальные позиции $\theta^k(bc)$ и $\theta^k(bd)$ в Θ. Так как θ инъективен на A^∞, $[i..\infty) \neq [j..\infty)$. Следовательно, их наибольший общий префикс v является биспециальным и содержит букву $\mathtt{a}$.

Покажем, что вообще для любого биспециального фактора v слова Θ, содержащего букву $\mathtt{a}$, существует более длинный фактор, обладающий таким же свойством.

Пусть i и j – две позиции в Θ такие, что $\Theta[i..i+m] = vc$ и $\Theta[j..j+m] = vd$, где c и d – различные буквы. Обозначим $y = [i..\infty)$ и $z = [j..\infty)$. Тогда, опять же в силу инъективности θ на A^∞, получаем $\theta(y) \neq \theta(z)$. Пусть $\theta(v)u$ – самый длинный общий префикс $\theta(y)$ и $\theta(z)$. Тогда существуют две буквы e и f, $e \neq f$ такие, что $\theta(v)ue$ и $\theta(v)uf$ – факторы Θ. Так как v содержит $\mathtt{a}$, $|\theta(v)u| > |v|$.

Повторяя это рассуждение, мы получаем бесконечную последовательность биспециальных факторов Θ. Для каждого v длины n имеем $F_\Theta(n+1) > F_\Theta(n)$ ($F_\Theta(n)$ – число факторов длины n, встречающихся в Θ), потому что любое слово длины n имеет продолжение в Θ, а v имеет два. Отсюда следует, что $\lim_{i\to\infty} F_\Theta(i) = \infty$, а также то (см. задачу 88), что Θ непериодическое и даже непериодическое в конечном счете.

Алгоритм, следующий из этого комбинаторного свойства, заключается в проверке того, содержит ли Θ беспециальную букву, а это можно сделать за время $O(\sum\{|\theta(b)| : b \in A\})$.

Примечания

Представленное доказательство комбинаторного свойства основано на оригинальном доказательстве из работы Pansiot [201], его можно найти в книге Kůrka [171, глава 4]. Понятие элементарного морфизма введено в работе Rozenberg and Salomaa [209]. Возможность установить периодичность в конечном счете для неэлементарных морфизмов также доказана в работе [201].

Общим свойством морфизмов является примитивность, аналог примитивности целочисленных матриц; это свойство сильнее неприводимости (показатель k один и тот же для всех пар букв). Но и более слабое условие может приводить к тому же выводу, например когда все буквы встречаются в $\theta^k(\mathtt{a})$ для некоторого $k > 0$. При таком условии приведенное выше доказательство применимо к следующему морфизму ξ, который не является неприводимым и порождает слово $\Xi = \xi^\infty(\mathtt{a}) = \mathtt{abcdabcd}\ldots = (\mathtt{abcd})^\infty$. То же самое слово порождается неприводимым морфизмом ψ.

$$\begin{cases} \xi(\mathtt{a}) = \mathtt{abcda} \\ \xi(\mathtt{b}) = \mathtt{b} \\ \xi(\mathtt{c}) = \mathtt{c} \\ \xi(\mathtt{d}) = \mathtt{d} \end{cases} \qquad \begin{cases} \psi(\mathtt{a}) = \mathtt{abcd} \\ \psi(\mathtt{b}) = \mathtt{b} \\ \psi(\mathtt{c}) = \mathtt{c} \\ \psi(\mathtt{d}) = \mathtt{dabcd} \end{cases}$$

Дополнительные сведения по теме можно найти в разделе «Сдвиговые пространства» (Shift Spaces) работы [8].

90. Простые антистепени

Двойственным к понятию периодичности или локальной периодичности является понятие антистепени, которое вводится в этой задаче.

Слово $u \in \{1, 2, \ldots, k\}^+$ называется антистепенью, если каждая из его букв встречается в нем ровно один раз. Это перестановка некоторого подмножества алфавита, т. е. $alph(u) = |u|$.

Вопрос. Показать, как за время $O(n)$ найти антистепени длины k, встречающиеся в слове $x \in \{1, 2, \ldots, k\}^n$.

Например, 13542 и 54231 встречаются в слове $341354231332 \in \{1,2,\ldots,5\}^+$ в позициях 2 и 4, и это единственные антистепени в нем длины 5.

Решение

Задачу можно обобщить: найти самую длинную антистепень, заканчивающуюся в любой заданной позиции j слова x. Для этого обозначим $antip[j]$

$\max\{|u| : u$ – антистепенной суффикс фактора $x[0..j]\}$.

В таблице, соответствующей слову 341354231332, показаны две его антистепени длины 5: 13542 и 54231, заканчивающиеся соответственно в позициях 6 и 8, т. к. $antip[6] = antip[8] = 5$.

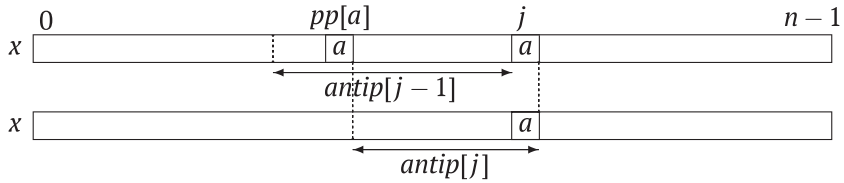
j	0	1	2	3	4	5	6	7	8	9	10	11
$x[j]$	3	4	1	3	5	4	2	3	1	3	3	2
$antip[j]$	1	2	3	3	4	4	**5**	4	**5**	2	1	2

Вычисление таблицы $antip$, ассоциированной с x, решает вопрос, потому что антистепень длины k заканчивается в позиции j слова x, если $antip[j] = k$. Алгоритм AntiPowers вычисляет $antip$ для слова над алфавитом $\{1,2,\ldots,k\}$.

```
AntiPowers(x ∈ {1,2,...,k}^+)
 1  for любой a ∈ {1,2,...,k} do
 2      pp[a] ← −1
 3  pp[x[0]] ← 0
 4  antip[0] ← 1
 5  for j ← 1 to |x| − 1 do
 6      a ← x[j]
 7      if j − pp[a] > antip[j − 1] then
 8          antip[j] ← antip[j − 1] + 1
 9      else antip[j] ← j − pp[a]
10      pp[a] ← j
11  return antip
```

Алгоритм AntiPowers вычисляет таблицу последовательно и использует для этого вспомогательный массив pp. В массиве, индексируемом буквами,

на каждом шаге хранятся предыдущие позиции $pp[a]$ вхождений всех встретившихся к этому моменту букв a.



Правильность алгоритма AntiPowers довольно очевидна. Действительно, если текущая буква a в позиции j не встречается в самой длинной антистепени, заканчивающейся в позиции $j-1$, то длина антистепени, заканчивающейся в позиции j, на единицу больше (строка 8). В противном случае, как показано на рисунке, $x[pp[a]+1..j-1]$ является антистепенью, не содержащей a, что дает длину $j - pp[a]$ самой длинной антистепени, заканчивающейся в позиции j (строка 9).

Ясно, что время работы AntiPowers составляет $O(n)$.

Примечания

Антистепень была определена в работе Fici et al. [113] как слово, являющееся конкатенацией блоков одинаковой длины, но попарно различных. Авторы показали, что любое бесконечное слово содержит антистепени с любым антипоказателем (количество блоков). В работе Badkobeh et al. [20] спроектирован оптимальный алгоритм, который находит такие антистепени с заданным антипоказателем. Описанный здесь алгоритм является первым шагом их решения. См. также [165].

91. Палиндромическая конкатенация палиндромов

Палиндромы – еще один вид регулярности, отличный от периодичности. Они естественно возникают при фолдинге данных, когда требуется сопоставлять участки данных, как в некоторых геномных последовательностях. В этой задаче мы будем заниматься палиндромами, встречающимися в произведении палиндромов.

Если задано конечное множество слов X, то вычислить количество всех палиндромов в X^2 легко за время $n \cdot |X|$, где n – полная длина слов в X. Однако существует гораздо более простой и эффективный метод, если X само является множеством палиндромов.

Вопрос. Пусть дано конечное множество X двоичных палиндромов суммарной длиной n. Спроектировать алгоритм, вычисляющий количество (различных) палиндромов в X^2 за время $O(n + |X|^2)$.

[**Указание:** если x и y – палиндромы, то xy является палиндромом тогда и только тогда, когда $xy = yx$.]

Решение

Приведенный ниже алгоритм основан на важном комбинаторном свойстве, сформулированном в указании. Начнем с его доказательства.

Пусть x и y – палиндромы. Если xy – палиндром, то имеем $x \cdot y = (x \cdot y)^R = y^R \cdot x^R = y \cdot x$.

Обратно, если $xy = yx$, то x и y имеют один и тот же примитивный корень (следствие леммы 2), который также является палиндромом. Следовательно, $xy = (xy)^R$.

Благодаря этому свойству алгоритм сводится к рассмотрению слов в X, которые имеют один и тот же примитивный корень. Мы выполняем следующий алгоритм:

- вычислить корень из каждого слова;
- после лексикографической сортировки корней разбить их на группы с одинаковым корнем;
- в каждой группе Y вычислить количество палиндромов в Y^2. Поскольку корни одинаковы, нам нужно только вычислить размер множества $\{|u| + |v| : u,v \in Y\}$, что можно сделать за время $O(|Y|^2)$.

Последний шаг можно выполнить за время $|Y|^2$ для каждой группы, а всего требуется время $O(|X|^2)$, поскольку сумма размеров групп Y равна $|X|$. Сортировка и вычисление корней занимают время $O(n)$ для алфавита фиксированного размера. Следовательно, алгоритм работает за время $O(n + |X|^2)$, что и требуется.

Примечание

Эта задача предлагалась на 13-й Польской олимпиаде по информатике в 2006 году.

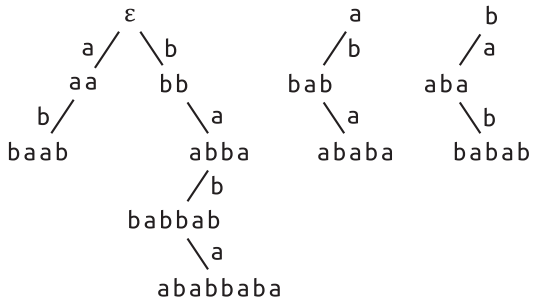
92. Деревья палиндромов

Понятие *леса палиндромов* $\mathcal{P}(x)$ предназначено для структурного представления всех палиндромов, встречающихся в слове x. Эта структура данных используется для выполнения различных операций над множеством палиндромных факторов, например для эффективного доступа к самому длинному палиндрому, заканчивающемуся в заданной позиции x, или для подсчета количества вхождений каждого палиндрома в x.

Лес состоит из набора деревьев, каждое из которых представляет все палиндромные факторы так же, как суффиксное дерево представляет все факторы. Чтобы повысить эффективность построения, в структуру также включены суффиксные ссылки. Однако леса палиндромов проще, чем суффиксные деревья, потому что каждое ребро помечено единственной буквой.

Каждый узел $\mathcal{P}(x)$ представляет палиндром, встречающийся в x. Из узла z исходит ребро $z \xrightarrow{a} aza$, помеченное буквой a, если aza – палиндром в x. Пустое слово ε является корнем дерева для представления четных палиндромов. А каждая буква слова является корнем дерева для представления нечетных палиндромов с центром в этой букве. На рисунке ниже показан лес $\mathcal{P}$(ababbababaab), состоящий из трех деревьев палиндромов.

Каждый узел деревьев можно представить интервалом $[i..j]$ позиций, соответствующих вхождению палиндрома $x[i..j]$. Кроме того, палиндром полностью определяется путем из корня в узел.



Вопрос. Предположим, что длина алфавита постоянна. Покажите, как построить лес палиндромов в слове за время, линейно зависящее от размера слова.

[**Указание:** используйте суффиксные ссылки.]

Решение

Алгоритм PalindromeForest строит лес палиндромов для входного слова x. Главная «фишка» построения – дополнить структуру суффиксными ссылками, определенными следующим образом. Для непустого палиндрома u его суффиксная ссылка указывает на самый длинный палиндром, являющийся собственным суффиксом u. Он обозначается $palsuf(u)$ и может быть пустым словом. Суффиксным предком u является любой суффикс, достижимый из u по суффиксным ссылкам (включая сам u).

Предположим, что u – палиндромный суффикс $x[0..i-1]$. Обозначим $upward(u, x[i])$ либо самый нижний суффиксный предок v узла u, для которого $x[i]vx[i]$ является суффиксом $w[0..i]$, либо пустое слово ε.

Чтобы построить лес для слова x, алгоритм обрабатывает слово в онлайновом режиме. Первоначально лес состоит только из корней деревьев, т. е. узлов ε и a, для каждой буквы a, встречающейся в x. В процессе построения поддерживаются суффиксные ссылки из узлов, а в переменной u хранится самый длинный палиндром, являющийся суффиксом прочитанного к этому моменту префикса x.

В главном цикле for для вычисления следующего значения u, которое включает текущую букву $x[i]$, используется важная функция *upward* (строки 4–7). А в строках 9–15 производится обновление леса в случае, когда нужно добавить новый узел.

```
PalindromeForest(непустое слово x)
 1  инициализировать лес P
 2  u ← x[0]
 3  for i ← 1 to |x| − 1 do
 4      v ← upward(u, x[i])
 5      if v = ε и x[i − 1] ≠ x[i] then
 6          u ← x[i]
 7      else u ← x[i]vx[i]
 8      if u ∉ P then
 9          добавить в P узел u и ребро v --x[i]--> u
10          v ← upward(palsuf(v), x[i])
11          if v = ε then
12              if x[i − 1] ≠ x[i] then
13                  palsuf(u) ← x[i]
14          else palsuf(u) ← ε
15      else palsuf(u) ← x[i]vx[i]
16  return P
```

Алгоритм требует линейного времени, поскольку число шагов при вычислении *upward* уменьшается пропорционально глубине u и *palsuf*(v) в лесе. Кроме того, каждая из этих двух глубин увеличивается не больше чем на единицу на каждой итерации.

Примечания

Древовидная структура палиндромов изучалась в работе Rubinchik and Shur [210], где была названа *овердревом* (англ. eertree). Впоследствии она была использована для проектирования нескольких алгоритмов, связанных с палиндромами.

93. Неустранимые образцы

В этой задаче изучаются образцы, определенные с помощью специального алфавита переменных в дополнение к конечному алфавиту $\mathtt{A} = \{\mathtt{a},\mathtt{b},\ldots\}$. Переменные берутся из бесконечного алфавита $\mathtt{V} = \{\alpha_1, \alpha_2, \ldots\}$. Образцом считается слово, буквы которого являются переменными. Типичный образец – $\alpha_1\alpha_1$: он встречается в слове, содержащем квадрат. Наша цель – породить неустранимые образцы.

Говорят, что слово $w \in \mathtt{A}^+$ содержит образец $P \in \mathtt{V}^*$, если $\psi(P)$ является фактором w для некоторого морфизма $\psi : alph(P)^+ \to \mathtt{A}^+$. В противном случае

говорят, что w устраняет P. Образец называется *устранимым*, если в $\mathtt{A}^+$ существует бесконечно много устраняющих его слов, что (в силу конечности $\mathtt{A}$) эквивалентно существованию бесконечного слова в $\mathtt{A}^\infty$, конечные факторы которого устраняют образец. Например, образец $\alpha_1\alpha_1$ устраним, если алфавит содержит по меньшей мере три буквы, но неустраним для двоичного алфавита (см. задачу 79).

Образцы Зимина Z_n – стандартные примеры неустранимых образцов. Они определены для $n > 0$ следующим образом:

$$Z_0 = \varepsilon \text{ и } Z_n = Z_{n-1} \cdot \alpha_n \cdot Z_{n-1}.$$

В частности, слово содержит образец Зимина Z_n, если оно содержит фактор, тип Зимина которого равен по меньшей мере n (см. задачу 42). Например, слово aaaaabaabbaaaabaabb содержит Z_3, потому что его фактор aaabaabbaaaabaa является образом $Z_3 = \alpha_1\alpha_2\alpha_1\alpha_3\alpha_1\alpha_2\alpha_1$ относительно морфизма ψ, определенного следующим образом:

$$\begin{cases} \psi(\alpha_1) = \mathtt{aa} \\ \psi(\alpha_2) = \mathtt{ab} \\ \psi(\alpha_3) = \mathtt{bba} \end{cases}.$$

Вопрос. Показать, что образцы Зимина Z_n, $n > 0$, неустранимы.

Решение

Пусть k – размер алфавита $\mathtt{A}$. Определим последовательность длин для $n > 1$:

$$l_1 = 1 \text{ и } l_n = (l_{n-1} + 1) \cdot k^{l_n - 1} + l_{n-1} - 1,$$

– и, прежде чем переходить к ответу на вопрос, рассмотрим следующее наблюдение.

Наблюдение. Любое слово длины l_n, $n > 1$, принадлежащее $\mathtt{A}^*$, имеет фактор вида uvu, где $|u| = l_{n-1}$ и $|v| > 0$.

Доказательство. Любое слово w длины l_n содержит $(l_{n-1} + 2) \cdot k^{l_n - 1}$ факторов длины l_{n-1}. Поскольку количество различных факторов длины l_{n-1} не превосходит $k^{l_n - 1}$, существует слово u длины l_{n-1}, которое по меньшей мере $l_{n-1} + 2$ раз входит в w. Следовательно, существуют два вхождения на расстоянии, не меньшем $l_{n-1} + 1$, и между этими вхождениями должно быть непустое слово v. Слово uvu – фактор требуемого вида. ■

Для ответа на вопрос достаточно доказать, что любое слово длины l_n, $n > 0$, содержит образец Зимина Z_n.

Доказательство проведем индукцией по n. Очевидно, что любое непустое слово содержит образец Z_1. В предположении, что любое слово длины l_{n-1} содержит образец Z_{n-1}, покажем, что любое слово w длины l_n содержит Z_n. В силу приведенного выше наблюдения, w содержит фактор вида uvu, где $|u| = l_{n-1}$

и $|v| > 0$. По предположению индукции, u содержит Z_{n-1}, поэтому $u = u_1u_2u_3$, где $u_2 = \psi(Z_{n-1})$ для некоторого морфизма $\psi : \{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}^+ \to A^+$. Тогда w содержит фактор $u_2 \cdot z \cdot u_2$, где $z = u_3vu_1$. Мы можем продолжить ψ, положив $\psi(\alpha_n) = z$, и тогда w будет содержать образ Z_n относительно этого морфизма. Что и требовалось доказать.

Примечания

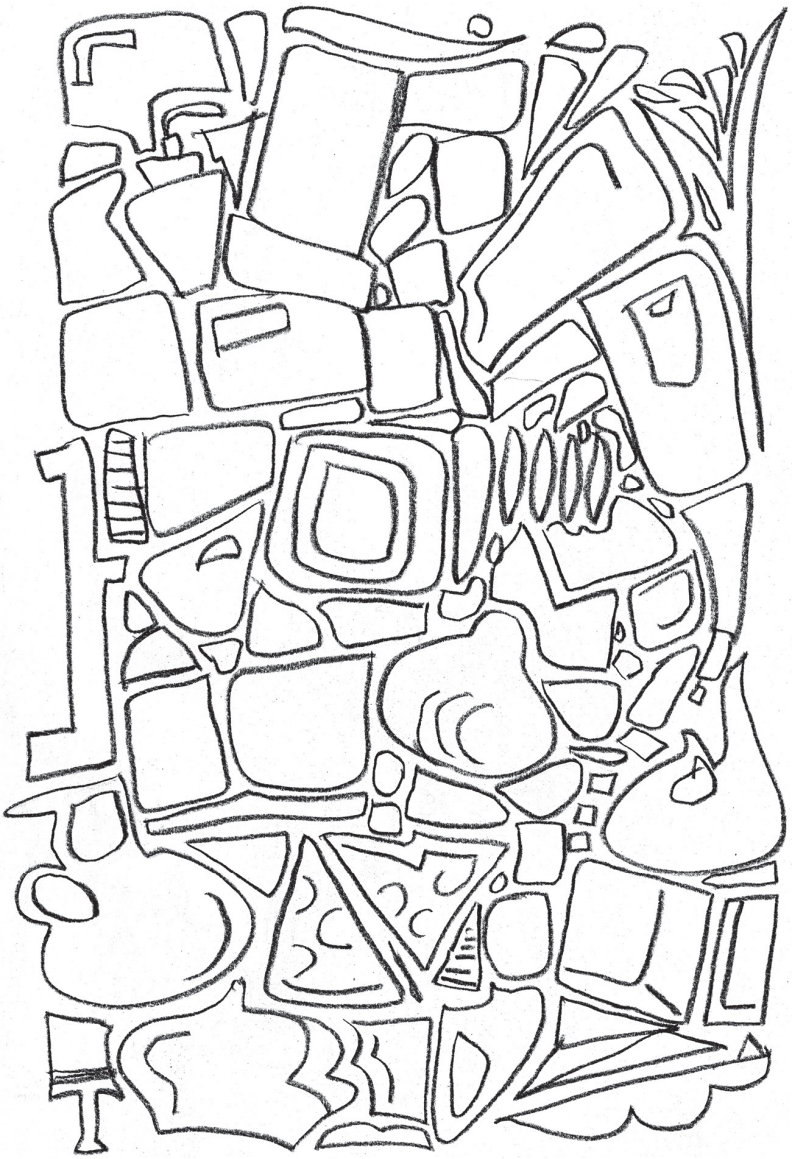
Обозначим $f(n)$ длину самого длинного двоичного слова, не содержащего Z_n. В силу результата о неустранимости, $f(n)$ конечно. Однако здесь конечность встречается с бесконечностью, потому что, например, $f(8) \geq 2^{(2^{16})} = 2^{65536}$ (см. [48]). Даже для коротких образцов значения $f(n)$ могут быть велики; например, существуют двоичные слова длины 10 482, устраняющие Z_4.

Задача о неустранимости образцов разрешима, как доказал Зимин (см. [176, глава 3]). Алгоритм может быть основан на словах Зимина, поскольку известно, что образец P, содержащий n переменных, неустраним тогда и только тогда, когда он содержится в Z_n. Иначе говоря, слова Зимина являются неустранимыми образцами, и они содержат все неустранимые образцы.

Однако вопрос о существовании детерминированного алгоритма решения задачи об устранимости образца за полиномиальное время до сих пор открыт. Известно только, что эта задача принадлежит классу сложности NP.

Глава 6

Сжатие текста



94. Преобразование Барроуза–Уилера слов Туэ–Морса

В этой задаче наша цель – показать индуктивную природу преобазования Барроуза–Уилера слов Туэ–Морса. Эти слова порождаются морфизмом Туэ–Морса μ из $\{\mathtt{a},\mathtt{b}\}^*$ в себя, определенным следующим образом: $\mu(\mathtt{a}) = \mathtt{ab}$, $\mu(\mathtt{b}) = \mathtt{ba}$. Итеративное применение μ, начиная с буквы $\mathtt{a}$, дает последовательные слова Туэ–Морса $\tau^n = \mu^n(\mathtt{a})$ длины 2^n.

Преобразование Барроуза–Уилера BW(w) слова w – это слово, составленное из последних букв отсортированных слов, сопряженных w (его ротаций). Приведем начальные слова Туэ–Морса: $\tau_0 = \mathtt{a}$, $\tau_1 = \mathtt{ab}$, $\tau_2 = \mathtt{abba}$, $\tau_3 = \mathtt{abbabaab}$. Преобразования двух последних равны $\mathrm{BW}(\tau_2) = \mathtt{baba}$, $\mathrm{BW}(\tau_3) = \mathtt{bbababaa}$.

Морфизм «надчеркивание» переводит $\{\mathtt{a},\mathtt{b}\}*$ и определяется так: $\bar{\mathtt{a}} = \mathtt{b}$, $\bar{\mathtt{b}} = \mathtt{a}$.

Вопрос. Показать, что преобразование Барроуза–Уилера $\mathrm{BW}(\tau_{n+1})$, $n > 0$, равно $\mathtt{b}^k \cdot \overline{\mathrm{BW}(\tau_n)} \cdot \mathtt{a}^k$, где $k = 2^{n-1}$.

Решение

Внимательно рассмотрим массив отсортированных сопряженных слов, участвующий в определении преобразования.

Обозначим S_{n+1} массив $2^{n+1} \times 2^{n+1}$, строками которого являются отсортированные ротации τ_{n+1}. По определению, $\mathrm{BW}(\tau_{n+1})$ – самый правый столбец S_{n+1}. Разобьем массив на три массива: T_{n+1} содержит верхние 2^{n-1} строк, M_{n+1} – средние 2^n строк и B_{n+1} – нижние 2^{n-1} строк.

Пример. Ниже показаны ротации $\tau_2 = \mathtt{abba}$ (массив R_2 слева) и отсортированные ротации (массив S_2 справа). Как видим, $\mathrm{BW}(\tau_2) = \mathtt{baba}$.

$$R_2 = \begin{matrix} \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \end{matrix} \qquad S_2 = \begin{matrix} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{a} \end{matrix}.$$

Массив S_3 дает $\mathrm{BW}(\tau_3) = \mathrm{BW}(\mathtt{abbabaab}) = \mathtt{bbababaa}$.

$$R_2 = \begin{matrix} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} \end{matrix}.$$

Разложение S_3 на T_3, M_3 и B_3 показывает, что $\mathrm{BW}(\tau_3)$ равно $\mathtt{b}^2\cdot\mathtt{abab}\cdot\mathtt{a}^2 = \mathtt{b}^2\cdot\overline{\mathrm{BW}(\tau_2)}\cdot\mathtt{a}^2$.

$$
\begin{array}{rl}
T_3 = & \begin{array}{cccccccc} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} \end{array} \\
\hline
 & \begin{array}{cccccccc} \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} \end{array} \\
M_3 = & \begin{array}{cccccccc} \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} \end{array} \\
\hline
B_3 = & \begin{array}{cccccccc} \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} \end{array}
\end{array}
$$

Поскольку строки S_n отсортированы, легко проверить, что они остаются отсортированными и после применения к ним μ. Следовательно, последний столбец $\mu(S_n)$ равен $\overline{\mathrm{BW}(\tau_n)}$, по определению μ.

Остается найти ротации τ_{n+1}, которые находятся в T_{n+1} и в B_{n+1}, и в итоге мы докажем, что $M_{n+1} = \mu(S_n)$.

Наблюдение. Число вхождений a и b в τ_n одинаково и равно 2^{n-1}.

В слове $\tau_{n+1} = \mu(\tau_n)$ рассмотрим вхождения ba, являющиеся образами вхождения b в τ_n. Согласно наблюдению, существует 2^{n-1} таких вхождений ba. Эквивалентное утверждение – все они начинаются в четных позициях τ_{n+1} (существуют и другие вхождения ba, если n достаточно велико).

Строки T_{n+1} состоят из ротаций, полученных при расщеплении τ_{n+1} в середине этих факторов ba. Все строки T_{n+1} начинаются буквой a и заканчиваются буквой b.

Поскольку в τ_n нет вхождений bbb, лексикографически наибольшая строка T_{n+1} не может начинаться с ababa и фактически начинается с abaa. Таким образом, эта строка меньше верхней строки $\mu(S_n)$, которая начинается с abab, т. к. это образ ротации τ_n с префиксом aa.

Симметрично B_{n+1} состоит из ротаций, полученных расщеплением вхождений ab, начинающихся в четных позициях τ_{n+1}. Тот факт, что все они больше последней строки $\mu(S_n)$, доказывается так же, как и выше.

Поскольку количество строк в T_{n+1} и B_{n+1} одинаково и равно $k = 2^{n-1}$, $M_{n+1} = \mu(S_n)$. Строки T_{n+1} заканчиваются буквой b и образуют префикс $\mathtt{b}^k$ слова $\mathrm{BW}(\tau_{n+1})$. Строки B_{n+1} заканчиваются буквой a и образуют суффикс $\mathtt{a}^k$ слова $\mathrm{BW}(\tau_{n+1})$.

95. Преобразование Барроуза–Уилера сбалансированных слов

Преобразование Барроуза–Уилера отображает слово w в слово $\mathrm{BW}(w)$, составленное из последних букв отсортированных сопряженных w. В этой задаче наша цель – охарактеризовать примитивные слова $w \in \{\mathtt{a},\mathtt{b}\}^+$, для которых

BW(w) ∈ b⁺a⁺. Такое слово можно затем сжать в слово длины $\log|w|$, представив показатели степени a и b.

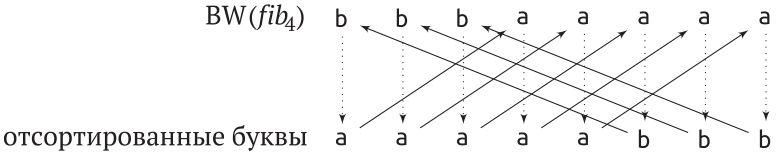
Характеристика основана на понятии сбалансированных слов. Плотностью (или весом) слова $u \in \{a,b\}^+$ называется число вхождений в него буквы a, т. е. $|u|_a$. Говорят, что слово w сбалансировано, если любые его два фактора, u и v, одинаковой длины имеют почти одинаковую плотность. Формально факторы удовлетворяют условию

$$|u| = |v| \Rightarrow -1 \le |u|_a - |v|_a \le 1.$$

Говорят, что слово w циклически сбалансировано, если w^2 сбалансировано.

Вопрос. Для примитивного слова $w \in \{a,b\}^+$ показать, что w циклически сбалансировано тогда и только тогда, когда BW(w) ∈ b⁺a⁺.

Слова Фибоначчи – типичные примеры циклически сбалансированных слов. На графе ниже показан цикл восстановления сопряженного к fib_4 слова (длина F_6 равна 8, а плотность F_5 равна 5) по b^3a^5. Если следовать вдоль цикла, начав с левой верхней буквы, то буквы в нижней строке будут встречаться в порядке aabaabab. Начав с другой буквы, мы получим другое сопряженное к fib_4 = abaababa, а само fib_4 получается, если начать с первого вхождения a в верхней строке. На самом деле слово длиной $|fib_n|$ и плотностью $|fib_{n-1}|$ циклически сбалансировано тогда и только тогда, когда оно является сопряженным к fib_n.



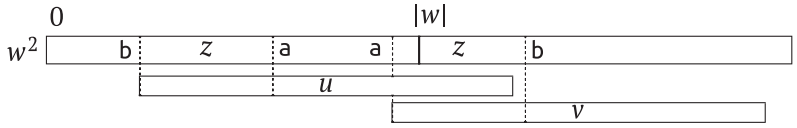
Вопрос. Показать, что BW(fib_n) ∈ b⁺a⁺ для $n > 0$.

Например, BW(fib_1) = BW(ab) = ba, BW(fib_2) = BW(aba) = baa и BW(fib_3) = BW(abaab) = bbaaa.

Решение

Преобразование циклически сбалансированного слова. Начнем с доказательства необходимости условия в первом вопросе. Сначала отметим, что BW(w), составленное из последних букв лексикографически отсортированных факторов длины $|w|$ слова w^2, можно было с тем же успехом составить из букв, предшествующих вхождениям этих отсортированных факторов в w^2, начиная с позиций от 1 до $|w|$. Решение сразу вытекает из следующей леммы.

***Лемма* 7.** *Для циклически сбалансированного примитивного слова w пусть* bu *и* av *– два фактора* w^2 *такие, что* $|u| = |v| = |w|$. *Тогда* $u < v$.

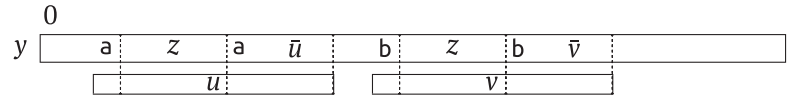


Доказательство. Пусть z – самый длинный общий префикс u и v. Поскольку u и v – сопряженные w, а слово w примитивное, $u \neq v$. Таким образом, либо za является префиксом u и zb является префиксом v (как на рисунке выше), либо zb является префиксом u и za является префиксом v. Но второй случай невозможен, потому что $|\mathtt{b}z\mathtt{b}| = |\mathtt{a}z\mathtt{a}|$ и $|\mathtt{b}z\mathtt{b}|_\mathtt{a} - |\mathtt{a}z\mathtt{a}|_\mathtt{a} = -2$, что противоречит условию сбалансированности. А в первом случае $u < v$. ■

Прямым следствием этой леммы является тот факт, что любое сопряженное к w, вхождению которого в w^2 предшествует буква b, меньше любого сопряженного, которому предшествует буква a. Таким образом, $\mathrm{BW}(w) \in \mathtt{b}^+\mathtt{a}^+$.

Доказательство достаточности. Для доказательства обратного утверждения мы покажем, что $\mathrm{BW}(w) \notin \mathtt{b}^+\mathtt{a}^+$, если w не является циклически сбалансированным. Это является прямым следствием следующей леммы.

Лемма 8. *Если примитивное слово* $y \in \{\mathtt{a},\mathtt{b}\}^+$ *не сбалансировано, то оно содержит два фактора вида* aza *и* bzb*, где* z *– некоторое слово.*



Доказательство. Пусть u и v – факторы y минимальной длины $m = |u| = |v|$ такие, что $||u|_\mathtt{a} - |v|_\mathtt{a}| > 1$. В силу минимальности m, u и v начинаются разными буквами, скажем a и b. Пусть z – наибольший общий префикс $\mathtt{a}^{-1}u$ и $\mathtt{b}^{-1}v$. Из неравенства $||u|_\mathtt{a} - |v|_\mathtt{a}| > 1$ следует, что $|z| < m - 2$. Тогда $u = \mathtt{a}zc\bar{u}$ и $v = \mathtt{b}zd\bar{v}$, где $\bar{u}$ и $\bar{v}$ – слова, а c и d – буквы, причем $c \neq d$. Так как длина m минимальна, не могут одновременно иметь место равенства $c = \mathtt{b}$ и $d = \mathtt{a}$. Стало быть, $c = \mathtt{a}$ и $d = \mathtt{b}$ (см. рисунок выше), и это значит, что aza и bzb – факторы y, что и требовалось доказать. ■

Чтобы завершить доказательство, заметим, что если w не является циклически сбалансированным, то w^2 не сбалансировано и, в силу только что доказанной леммы, содержит два фактора вида aza и bzb. Следовательно, слово, сопряженное w, с префиксом za, которому предшествует буква a, меньше, чем сопряженное с префиксом zb, которому предшествует буква b. Таким образом, ab – подпоследовательность $\mathrm{BW}(w)$, откуда следует, что $\mathrm{BW}(w) \notin \mathtt{b}^+\mathtt{a}^+$.

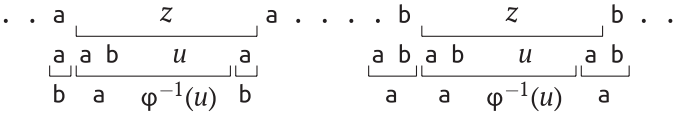
Случай слов Фибоначчи. Чтобы доказать утверждение второго вопроса, покажем, что слова Фибоначчи циклически сбалансированы. Поскольку их квадраты – префиксы бесконечного слова Фибоначчи **f**, достаточно показать,

что последнее не содержит двух факторов вида aza и bzb ни для какого слова z. Это приводит к требуемому результату в силу утверждения, доказанного при рассмотрении первого вопроса.

Напомним, что **f** генерируется итеративным применением морфизма φ из $\{a,b\}^*$ в себя, определенного как $\varphi(a) = ab$ и $\varphi(b) = a$: $\mathbf{f} = \varphi^\infty(a)$. Это слово является также неподвижной точкой φ: $\varphi(\mathbf{f}) = \mathbf{f}$.

В связи с вопросом отметим, что, например, aa является фактором **f**, но bb – нет, и аналогично bab – фактор **f**, а aaa – нет. Это значит, что **f** устраняет bb и aaa наряду со многими другими двоичными словами.

***Лемма* 9.** *Бесконечное слово Фибоначчи* **f** *не содержит двух факторов* aza *и* bzb *ни для какого слова* z.



Доказательство. Предположим противное – что **f** содержит два фактора указанного вида. Пусть z – самое короткое слово, для которого aza и bzb встречаются в **f**.

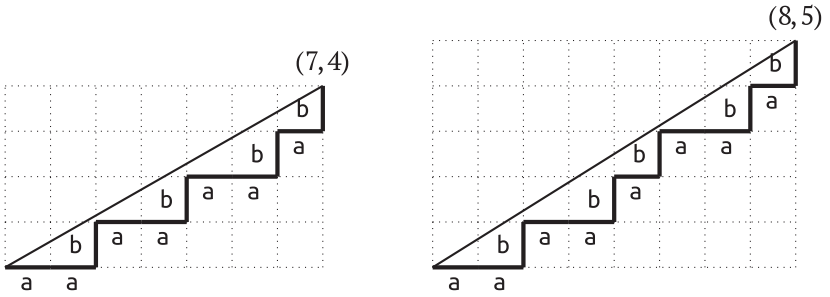
Из рассмотрения слов, устраняемых **f**, таких как a^3 и b^2, следует, что z должно начинаться с ab и заканчиваться буквой a. Простая проверка показывает, что длина z должна быть не меньше 4, т. е. $z = abua$, где $u \neq \varepsilon$ (см. рисунок). Действительно, оба вхождения a не могут совпадать, потому что **f** устраняет a^3. Далее, u не может быть пустым, потому что ababab не входит в **f**, т. к. оно было бы образом aaa относительно φ, а **f** устраняет aaa.

Слова aab$u$a – префикс aza – и abab$u$ab единственным способом разлагаются на факторы из множества {a,ab}, представляющего собой суффиксный код. Таким образом, $\varphi^{-1}(aabua) = ba\varphi^{-1}(u)b$ и $\varphi^{-1}(ababuab) = aa\varphi^{-1}(u)a$ встречаются в **f**. Но это противоречит минимальности длины z, потому что $a\varphi^{-1}(u)$ короче z. Таким образом, **f** не содержит двух факторов вида aza и bzb, что и требовалось доказать. ■

Примечания

Доказанный в этой задаче результат впервые был представлен в другой форме в работе Mantaci et al. [185]. В нашем доказательстве используется предложение 2.1.3 из книги [176, глава 2], которое дополнительно утверждает, что слово z в лемме 8 является палиндромом.

Как показано в работе Berstel and de Luca [33], рассмотренная задача имеет отношение к словам Кристоффеля, которые являются сбалансированными словами Линдона (см. также [35, 176]). Этот результат сформулирован в работе Reutenauer [208] следующим образом: пусть w – слово Линдона такое, что $p = |w|_a$ и $q = |w|_b$ – взаимно простые числа. Тогда w является словом Кристоффеля тогда и только тогда, когда $BW(w) = b^q a^p$.



Нижние слова Кристоффеля аппроксимируют снизу отрезки на плоскости, исходящие из начала координат. На рисунке показано слово aabaabaabab (слева), представляющее путь по линиям сетки, близко примыкающий снизу к отрезку, соединяющему точки (0,0) и (7,4). Слово Линдона, сопряженное слову Фибоначчи fib_5 = abaababaabaab (справа), длиной $F_7 = 13$ и плотностью $F_6 = 8$ аппроксимирует отрезок, соединяющий точки (0,0) и $(F_6, F_5) = (8,5)$.

96. Преобразование Барроуза–Уилера на месте

Преобразование Барроуза–Уилера (BW) слова можно вычислить над алфавитом, допускающим линейную сортировку, за линейное время в памяти линейного объема. Для этого применяется сортировка суффиксов или сопряженных слов, что требует линейной дополнительной памяти (помимо входных данных).

В этой задаче показано, как путем изменения входного слова на месте получить его преобразование, используя дополнительную память постоянного объема, правда, ценой замедления вычислений.

Пусть x – фиксированное слово длины n, последней буквой которого является маркер конца #, меньший любой другой буквы. Тогда сортировка слов, сопряженных x, для получения BW(x) сводится к сортировке его суффиксов. Преобразование состоит из букв, предшествующих суффиксам (целому слову предшествует маркер конца – циклически). Например, если x = banana#, то получаем BW(x) = annb#aa:

BW(x)							
a	#						
n	a	#					
n	a	n	a	#			
b	a	n	a	n	a	#	
#	b	a	n	a	n	a	#
a	n	a	#				
a	n	a	n	a	#		

Вопрос. Спроектировать алгоритм, который вычисляет преобразование Барроуза–Уилера слова длины n с маркером конца за время $O(n^2)$ и потребляет при этом дополнительную память постоянного объема.

Решение

В начальный момент положим $z = x$. Наша цель – преобразовать (массив) z на месте в BW(x). Будем просматривать z справа налево.

Обозначим x_i суффикс $x[i..n-1]$ слова x, $0 \le i < n$. На i-й итерации слово $z = x[0..i] \cdot \mathrm{BW}(x[i+1..n-1])$ преобразуется в слово $x[0..i-1] \cdot \mathrm{BW}(x[i..n-1])$. Для этого буква $c = x[i]$ обрабатывается с целью найти ранг x_i среди суффиксов $x_i, x_{i+1}, \ldots, x_{n-1}$.

Если p – позиция # в слове z, то $p - i$ – ранг x_{i+1} среди суффиксов $x_{i+1}, x_{i+2}, \ldots, x_{n-1}$. Тогда $z[p]$ должно быть равно c в конце i-й итерации, потому что она предшествует суффиксу x_{i+1}.

Для завершения итерации остается найти новую позицию маркера #. Поскольку он предшествует самому x_i, нам нужно найти ранг x_i среди суффиксов $x_i, x_{i+1}, \ldots, x_{n-1}$. Это легко сделать, подсчитав количество q букв, меньших c, в $z[i+1..n-1]$ и количество t букв, равных c, в $z[i+1..p-1]$. Тогда $r = q + t$ – искомый ранг x_i. В итоге вычисление состоит из сдвига $z[i+1..i+r]$ на одну позицию влево в z и записи # в $z[i+r]$.

Пример. Для $x =$ `banana#` на рисунке ниже представлено все вычисление. В начале итерации $i = 2$ (средняя строка) мы имеем $z =$ `ban·an#a` и обрабатываем подчеркнутую букву $c =$ `n`. В суффиксе `an#a` есть три буквы, меньших c, и до # – одна буква, равная c. Следовательно, $r = 4$. После подстановки c вместо # фактор $z[3..3+4-1]$ сдвигается влево и после него вставляется маркер конца. В результате получается $z =$ `ba·anna#`.

i					x			r
	b	a	n	a	n	a	#	
4	b	a	n	a	n	a	#	$2 = 2 + 0$
3	b	a	n	a	a	n	#	$2 = 1 + 1$
2	b	a	n	a	n	#	a	$4 = 3 + 1$
1	b	a	a	n	n	a	#	$3 = 1 + 2$
0	b	a	n	n	#	a	a	$4 = 4 + 0$
	a	n	n	b	#	a	a	

BW(x)

Алгоритм InPlaceBW реализует описанную выше стратегию. Он начинается итерацией $i = n - 3$, т. к. $x[n-2..n-1]$ – свое собственное преобразование.

```
InPlaceBW(x – слово длины n с маркером конца)
 1  for i ← n − 3 downto 0 do
 2      p ← позиция # в x[i + 1..n − 1]
 3      c ← x[i]
 4      r ← 0
 5      for j ← i + 1 to n − 1 do
```

```
 6          if x[j] < c then
 7              r ← r + 1
 8      for j ← i + 1 to p − 1 do
 9          if x[j] = c then
10              r ← r + 1
11      x[p] ← c
12      x[i..i + r − 1] ← x[i + 1..i + r]
13      x[i + r] ← #
14  return x
```

Что касается времени выполнения, то инструкции в строках 2, 5–7, 8–10 и 12 в совокупности требуют времени $O(n - i)$. Следовательно, общее время работы составляет $O(n^2)$ при используемой модели сравнения.

Примечания

Материал для этой задачи взят из работы [73]. Авторы показывают также, как инвертировать преобразование BW на месте, чтобы восстановить начальное слово за такое же время и в таком же объеме памяти для алфавита постоянного размера. Дополнительные сведения о преобразовании Барроуза–Уилера см. в книге Adjeroh et al. [2], посвященной этой теме.

97. Факторизация Лемпеля–Зива

В этой задаче рассматривается факторизация Лемпеля–Зива, применяемая к словам. Имеется в виду разложение слова w в произведение $w_0w_1...w_k$, где каждое w_i – самый длинный префикс of $w_iw_{i+1}...w_k$, встречающийся в w раньше текущей позиции $|w_0w_1...w_{i-1}|$. Если предыдущего вхождения не существует, то w_i – первая буква слова $w_iw_{i+1}...w_k$.

Факторизация хранится в массиве LZ, определенном следующим образом: LZ[0] = 0, а для $1 \leq i \leq k$, LZ[i] = $|w_0w_1...w_{i-1}|$. Например, факторизация слова `abaababababbabbb` имеет вид `a·b·a·aba·bab·babb·b`, что дает массив LZ = [0,1,2,3,6,9,13,14].

Вопрос. Показать, как вычислить массив LZ для заданного слова за линейное время в предположении, что размер алфавита фиксирован.

Такого же времени работы можно достичь, когда алфавит допускает линейную сортировку, что является более слабым условием. Для этого используется массив наибольших предыдущих факторов (longest previous factor – LPF) слова, который при указанном условии вычисляется за линейное время (см. задачу 53). Массив LPF определен для каждой позиции i слова w следующим образом: LPF[i] – длина самого длинного фактора w, который начинается как в позиции i, так и в некоторой меньшей позиции. Ниже показан массив LPF для слова `abaababababbabbb`.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$w[i]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b
LPF[i]	**0**	**0**	**1**	**3**	2	4	**3**	2	1	**4**	3	2	2	**1**

Вопрос. Спроектировать алгоритм, который за линейное время вычисляет факторизацию Лемпеля–Зива слова, заданного своим массивом LPF.

Решение

Прямое вычисление LZ. Для решения первого вопроса используется суффиксное дерево $T = \mathcal{ST}(w)$ слова w. Его конечные узлы (или листья, если w имеет маркер конца) идентифицируются суффиксами w, и можно предполагать, что они помечены своими начальными позициями. Кроме того, для каждого узла v дерева T $first(v)$ обозначает наименьшую метку листа в поддереве с корнем v; ее можно вычислить простым обходом дерева снизу вверх.

Предположим, что LZ[0..$i-1$] уже вычислено и LZ[$i-1$] $= j (1 \le i \le k)$. Чтобы найти LZ[i], мы спускаемся по пути, начинающемуся с $root(T)$, выбирая ветви, соответствующие буквам префикса $w[j\,..\,n-1]$. Спуск прекращается, если продолжение невозможно или просматриваемое слово не встречается раньше позиции j. Последнее условие проверяется следующим образом: на данном шаге текущий узел дерева – это либо явный узел v, либо неявный внутренний узел; во втором случае мы спускаемся вниз в поисках первого явного узла. Проверка существования предыдущего вхождения сводится к проверке того, что $first(v) < j$.

Построение суффиксного дерева занимает линейное время, если алфавит допускает линейную сортировку (см. задачу 47), а обход требует линейного времени для алфавита фиксированного размера. Для алфавита общего вида требуется время $O(|w| \log alph(w))$.

Вычисление LZ по массиву LPF. Следующий алгоритм отвечает на второй вопрос.

```
LZ-FACTORISATION(таблица LPF для слова длины n)
 1   (LZ[0], i) ← (0,0)
 2   while LZ[i] < n do
 3       LZ[i + 1] ← LZ[i] + max{1, LPF[LZ[i]]}
 4       i ← i + 1
 5   return LZ
```

Ясно, что LZ[0] установлено правильно. Предположим, что на i-й итерации цикла while значения LZ[j] правильны для $0 \le j \le i$. В частности, LZ[i] $= |w_0 w_1 \ldots w_{i-1}|$.

Пусть w_i – следующий фактор в факторизации. Если w_i не пуст, то его длина (большая, чем 1) равна LPF[$|w_0 w_1 \ldots w_{i-1}|$]; таким образом, LZ[$i+1$] = LZ[i] + LPF[LZ[i]]. Если же w_i пуст, то LZ[$i+1$] = LZ[i] + 1. В обоих случаях инструкция в строке 3 правильно вычисляет LZ[$i+1$].

Алгоритм останавливается, когда $LZ[i] \geq n$; таким образом, он вычисляет все значения $LZ[i]$ для $0 \leq i \leq k$.

Все инструкции алгоритма требуют постоянного времени, за исключением цикла while, который выполняется $k + 1$ раз; следовательно, алгоритм работает за время $O(k)$.

Примечания

Альтернативный алгоритм можно спроектировать с помощью суффиксного автомата (или DAWG) слова. Об алгоритме для решения второго вопроса и о применениях массива LPF см. работу [76].

Существует много вариантов определения факторизации. Принятый здесь навеян методом сжатия LZ77, описанным в работе Ziv and Lempel [243] (см. [37]). Стимулом для его изучения стала высокая производительность в реальных приложениях.

Факторизация полезна также для порождения эффективных алгоритмов нахождения повторений в словах (см. [67, 167]), но алгоритм вычисления серий, представленный в работе [26], работает быстрее (см. задачу 87). Факторизацию можно применять и при работе с повторениями в других приложениях, например для нахождения приближенных повторений в словах [168] или совмещения геномных последовательностей [88].

98. Декодирование Лемпеля–Зива–Уэлча

Метод сжатия Лемпеля–Зива–Уэлча основан на факторизации Лемпеля–Зива. Он заключается в кодировании повторяющихся факторов входного текста их кодами в словаре D. Словарь, инициализированный всеми буквами алфавита A, замкнут относительно префиксов: любой префикс слова в словаре сам принадлежит словарю.

Ниже показан алгоритм, в котором $code_D(w)$ – индекс фактора w в словаре D.

```
LZW-ENCODER(непустое слово input)
 1  D ← A
 2  w ← первая буква input
 3  while не конец input do
 4      a ← следующая буква input
 5      if wa ∈ D then
 6          w ← wa
 7      else WRITE(code_D(w))
 8          D ← D ∪ {wa}
 9          w ← a
10  write(code_D(w))
```

Алгоритм декомпрессии читает последовательность кодов, порожденную кодировщиком, и обновляет словарь так же, как это делает кодировщик.

```
LZW-DECODER(непустое слово input)
1  D ← A
2  while не конец input do
3      i ← следующий код в input
4      w ← фактор кода i в D
5      WRITE(w)
6      a ← первая буква следующего декодированного фактора
7      D ← D ∪ {wa}
```

Вопрос. Показать, что на шаге декодирования алгоритм LZW-DECODER может прочитать код i, еще не принадлежащий словарю D, тогда и только тогда, когда индекс i соответствует коду слова aua, где au – ранее декодированный фактор, $a \in A$ и $u \in A^*$.

Этот вопрос проясняет единственную критическую ситуацию, встречающуюся в работе декодировщика. Описанное свойство дает тот элемент, который необходим для доказательства того, что входное слово декодируется правильно.

Решение

Сначала докажем, что если сразу после вывода кода функцией write кодировщик читает слово $v = auaua$, где $a \in A$, $u \in A^*$, $au \in D$ и $aua \notin D$, то декодировщик прочитает код, который не входит в словарь.

Кодировщик начинает читать $au \in D$. При чтении следующей буквы a в v кодировщик запишет код au и добавит aua в словарь. Двигаясь дальше, он прочитает второе вхождение ua и запишет код aua (т. к. словарь замкнут относительно префиксов, aua нельзя продолжить).

Когда впоследствии декодировщик прочитает код au, то далее он прочитает код aua еще до того, как он оказался в словаре.

Теперь докажем, что если декодировщик читает код i, который еще не находится в словаре, то он соответствует фактору aua, где au – фактор, соответствующий коду, прочитанному непосредственно перед i.

Пусть w – фактор, соответствующий коду, прочитанному непосредственно перед i. Единственный код, который не был вставлен в словарь перед чтением i, соответствует фактору wc, где c – первая буква фактора, имеющего код i. Таким образом, $c = w[0]$. Если $w = au$, то код i соответствует фактору aua.

Пример. Пусть на вход подано слово ACAGAATAGAGA над 8-битовым алфавитом ASCII.

Первоначально словарь содержит ASCII-символы, и их индексами являются кодовые позиции в ASCII. Также в словаре присутствует искусственный символ конца слова с индексом 256.

Кодирование

```
A C  A  G  A  A  T  A  G  A  G  A      w    Записано  Добавлено в D
  ↑                                    A      65       AC, 257
     ↑                                 C      67       CA, 258
        ↑                              A      65       AG, 259
           ↑                           G      71       GA, 260
              ↑                        A      65       AA, 261
                 ↑                     A      65       AT, 262
                    ↑                  T      84       TA, 263
                       ↑               A
                          ↑            AG     259       AGA, 264
                             ↑         A
                                ↑      AG
                                   ↑   AGA    264
                                          256
```

Декодирование

Входная последовательность имеет вид 65, 67, 65, 71, 65, 65, 84, 259, 264, 256.

Прочитано	Записано	Добавлено
65	A	
67	C	AC, 257
65	A	CA, 258
71	G	AG, 259
65	A	GA, 260
65	A	AA, 261
84	T	AT, 262
259	AG	TA, 263
264	AGA	AGA, 264
256		

Критическая ситуация возникает при чтении индекса 264, потому что в этот момент ни одно слово в словаре не имеет такого индекса. Но поскольку предыдущий декодированный фактор – AG, индекс 264 может соответствовать только фактору AGA.

Примечания

Метод Лемпеля–Зива–Уэлча спроектирован Уэлчем в работе [239]. Это усовершенствование метода, первоначально разработанного Зивом и Лемпелем в работе [243].

99. Стоимость кода Хаффмана

Метод сжатия Хаффмана в применении к тексту $x \in A^*$ сопоставляет двоичное кодовое слово каждой букве x, стремясь породить кратчайший закодированный текст. Принцип заключается в том, чтобы наиболее часто встречающимся буквам сопоставлять самые короткие кодовые слова, а редким буквам – более длинные.

Кодовые слова образуют префиксный код (ни одно кодовое слово не является префиксом другого), с ним естественно ассоциируется двоичное дерево, в котором ребра, ведущие в левый и правый дочерние узлы, помечены соответственно цифрами 0 и 1. Листья дерева соответствуют исходным буквам, а метки ветвей содержат их кодовые слова. В описанном методе код полон: любой внутренний узел дерева содержит ровно два дочерних узла.

Стоимость кода Хаффмана равна сумме $\sum_{a \in A} freq(a) \times |code(a)|$, где $code(a)$ – двоичное кодовое слово буквы a. Это наименьшая длина двоичного текста, являющегося результатом сжатия слова x описанным методом, в котором $freq(a) = |x|_a$ для каждой буквы $a \in alph(x)$. Рассмотрим следующий алгоритм, применяемый к частотам букв (весам).

```
HuffmanCost(список положительных весов S)
 1  result ← 0
 2  while |S| > 1 do
 3      p ← MinDelete(S)
 4      q ← MinDelete(S)
 5      добавить p + q в S
 6      result ← result + p + q
 7  return result
```

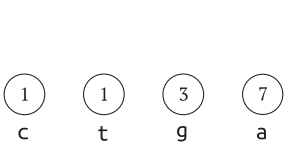
Вопрос. Доказать, что алгоритм HuffmanCost(S) вычисляет наименьшую стоимость кода Хаффмана по списку S весов элементов.

[**Указание:** рассмотрите ассоциированное с кодом дерево Хаффмана.]

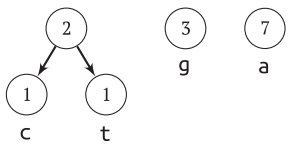
Пример. Пусть $S = \{7,1,3,1\}$. В начальный момент $result = 0$.

Шаг 1: $p = 1$, $q = 1$, $p + q = 2$, $S = \{7,3,2\}$, $result = 2$.
Шаг 2: $p = 2$, $q = 3$, $p + q = 5$, $S = \{7,5\}$, $result = 7$.
Шаг 3: $p = 5$, $q = 7$, $p + q = 12$, $S = \{12\}$, $result = 19$.

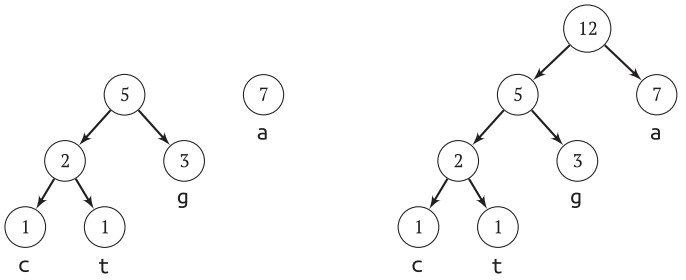
На рисунке ниже показан лес, который в конечном итоге становится деревом Хаффмана. Узлы помечены весами.



Начальное состояние



Шаг 1



Шаг 2 Шаг 3

Конечное дерево дает кодовые слова, ассоциированные с буквами; они показаны в таблице ниже.

	a	c	g	t
freq	7	1	3	1
code	1	000	01	001
|*code*|	1	3	2	3

Стоимость дерева равна $7 \times 1 + 1 \times 3 + 3 \times 2 + 1 \times 3 = 19$. Это длина сжатого слова `000 1 01 1 001 1 1 01 1 01 1 1`, соответствующего слову `cagataagagaa`, для которого частоты букв такие, как в нашем примере. Если каждая буква этого слова закодирована 8-битовым кодовым словом, то его длина равна 96.

Вопрос. Показать, как реализовать алгоритм HuffmanCost(S), чтобы время его работы было линейным, если входной список S отсортирован в порядке возрастания.

[**Указание:** используйте очередь для вставки новых значений (соответствующих внутренним узлам дерева).]

Решение

Правильность алгоритма HuffmanCost. Обозначим S_i значение S на i-м шаге цикла while, $0 \le i \le |S| - 1$.

Цикл имеет следующий инвариант: *result* равно суммарной стоимости кодовых слов Хаффмана, представляющих веса, которые хранятся в S_i.

До начала первой итерации S_0 – лес, состоящий из деревьев нулевой глубины; ему соответствует начальное значение $result = 0$.

На i-й итерации алгоритм выбирает и удаляет два наименьших веса p и q и S_{i-1} и добавляет $p + q$ в S_{i-1}, порождая S_i. Это соответствует созданию нового дерева, корень которого имеет вес $p + q$, а выбранные узлы являются для него дочерними. Следовательно, нужен еще один бит, чтобы можно было записать кодовые слова букв, ассоциированных с листьями нового дерева. Всего это происходит $p + q$ раз, поэтому *result* нужно увеличить на $p + q$, что и делается в строке 6. В итоге в конце i-й итерации *result* равно суммарной стоимости кодовых слов Хаффмана, представляющих веса, которые хранятся в S_i.

В конце ($|S| - 1$)-й итерации в S останется только один узел, и *result* будет равно полной стоимости кода Хаффмана.

Ясно, что если на любой итерации цикла while выбрать какие-либо другие значения – не с минимальными весами в S, то итоговая стоимость будет больше, чем *result*.

Реализация за линейное время. Чтобы алгоритм HuffmanCost(S) работал за линейное время, достаточно вставлять новые веса в очередь Q. Так как новые веса поступают в порядке возрастания, Q также в любой момент отсортирована, поэтому каждый шаг требует постоянного времени. Это решение показано ниже.

```
HuffmanCostLinear(список положительных весов в порядке возрастания S)
1  result ← 0
2  Q ← ∅
3  while |S| + |Q| > 1 do
4  (p, q) ← извлечь 2 наименьших значения среди первых двух значений S
     и первых двух значений Q
5  Enqueue(Q, p + q)
6  result ← result + p + q
7  return result
```

Пример. Пусть $S = (1,1,3,7)$. В начальный момент $result = 0$ и $Q = \emptyset$.

Шаг 1: $p = 1$, $q = 1$, $p + q = 2$, $S = (3,7)$, $Q = (2)$, $result = 2$.
Шаг 2: $p = 2$, $q = 3$, $p + q = 5$, $S = (7)$, $Q = (5)$, $result = 7$.
Шаг 3: $p = 5$, $q = 7$, $p + q = 12$, $S = \emptyset$, $Q = (12)$, $result = 19$.

Примечания

Деревья Хаффмана впервые описаны в работе Huffman [144]. Метод построения за линейное время, при условии что список частот уже отсортирован, предложен в работе Van Leeuwen [235].

100. Кодирование Хаффмана с ограничением на длину

Зная частоты букв алфавита, алгоритм Хаффмана строит оптимальный префиксный код, при котором длина закодированного текста минимальна. В общем случае на длину кодовых слов не налагается никаких ограничений. Но иногда длину кодового слова необходимо ограничить. Построение кода, удовлетворяющего такому ограничению, и является предметом данной задачи.

Задача о нумизмате – пример, когда возникает ограничение. Для нумизмата монета имеет два независимых свойства: номинал (денежная цен-

ность) и нумизматическая ценность (важная для коллекционера). Цель – набрать денежную сумму N, минимизировав общую нумизматическую ценность.

Предположим, что номиналы являются степенями 2: 2^{-i}, где $1 \le i \le L$. Множество монет организовано следующим образом: для каждого номинала имеется список, в котором монеты отсортированы в порядке возрастания нумизматической ценности.

Метод заключается в группировке соседних монет по две, начиная со списка наименьшего номинала; если количество монет в списке нечетно, то последняя отбрасывается. Нумизматическая ценность пакета из двух монет равна сумме нумизматических ценностей этих монет. Сформированные пакеты объединяются с монетами следующего по величине номинала (отсортированного в порядке возрастания нумизматической ценности). Процесс повторяется, пока не будет обработан список монет номинала 2^{-1}.

Вопрос. Спроектировать алгоритм, который для списка n частот вычисляет оптимальный код Хаффмана с ограничением на длину, в котором нет кодовых слов длины меньше L и который работает за время $O(nL)$.

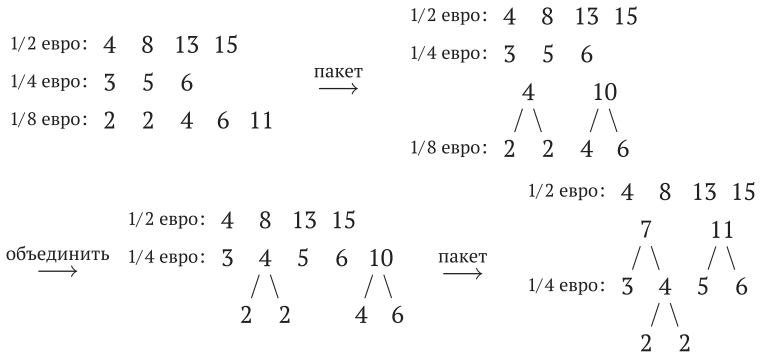
[**Указание:** сведите задачу к задаче о нумизмате с монетами двоичных номиналов.]

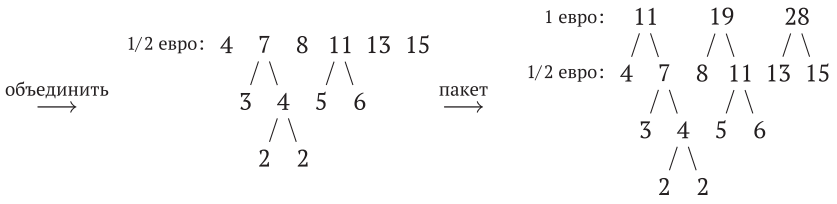
Пример. У нумизмата имеется:

- 4 монеты по 1/2 евро нумизматической ценности 4, 8, 13 и 15;
- 3 монеты по 1/4 евро нумизматической ценности 3, 5 и 6;
- 5 монет по 1/8 евро нумизматической ценности 2, 2, 4, 6 и 11.

Он хочет набрать сумму 2 евро.

Сначала монеты номиналом 1/8 евро группируются по две, так что образуется два пакета стоимостью 1/4 евро с нумизматической ценностью 4 и 10 соответственно, а монета с нумизматической ценностью 11 отбрасывается. Отдельные монеты и пакеты по 1/4 евро группируются, так что получается 2 пакета стоимостью 1/2 евро с нумизматической ценностью 7 и 11; при этом пакет с нумизматической ценностью 10 отбрасывается.





Первые два пакета дают решение: 2 евро набираются из двух монет по 1/8 евро, имеющих нумизматическую ценность 2 каждая; трех монет по 1/4 евро с нумизматической ценностью 3, 5 и 6 и двух монет по 1/2 евро с нумизматической ценностью 4 и 8. Общая нумизматическая ценность такого набора равна 30.

Алгоритм PackageMerge(S, L) реализует стратегию для множества S монет с номиналами от 2^{-L} до 2^{-1}. Алгоритм Package(S) группирует соседние элементы S по два, а алгоритм Merge(S, P) объединяет два отсортированных списка.

В конечном итоге первые N элементов списка PackageMerge(S, L), имеющих наименьшую нумизматическую ценность, возвращаются в качестве решения.

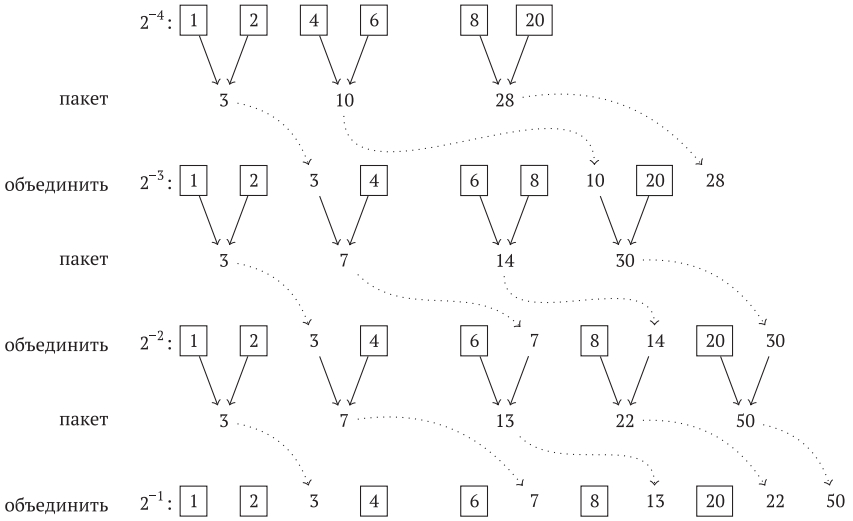
```
PackageMerge(множество монет S, L)
 1  for d ← 1 to L do
 2  S_d ← список монет из S номиналом 2^-d, отсортированный в порядке
      возрастания нумизматической ценности
 3  for d ← L downto 1 do
 4      P ← Package(S_d)
 5      S_{d-1} ← Merge(S_{d-1}, P)
 6  return S_0
```

Время работы обоих алгоритмов Package(S') и Merge(S', P') линейно зависит от $n = |S|$. Таким образом, при условии что списки монет предварительно отсортированы, алгоритм PackageMerge(S, L) требует линейного времени и памяти $O(nL)$.

Решение

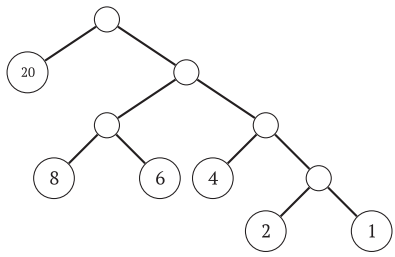
Если задано n частот букв w_i, $1 \le i \le n$, то описанный выше алгоритм можно применить для набора суммы $n - 1$, создав для каждого $1 \le i \le n$ L монет с нумизматической ценностью w_i и номиналом 2^{-j}, $1 \le j \le L$, а затем найти оптимальный код Хаффмана, в котором длины всех кодовых слов не больше L.

Пример. Пусть даны следующие шесть частот, перечисленных в порядке возрастания: $w = (1,2,4,6,8,20)$, и пусть $L = 4$. Тогда алгоритм PackageMerge работает, как показано на рисунке ниже.



Длины кодовых слов, соответствующих каждой частоте, вычисляются путем просмотра в порядке возрастания первых $2n - 2 = 10$ элементов на последнем уровне. Результаты сведены в таблицу; например, 6-й элемент имеет вес 7 и соответствует частотам 1, 2 и 4. Ниже показано также дерево, соответствующее этим длинам кодовых слов.

Элемент	Вес	1	2	4	6	8	20
1	1	1	0	0	0	0	0
2	2	1	1	0	0	0	0
3	3	2	2	0	0	0	0
4	4	2	2	1	0	0	0
5	6	2	2	1	1	0	0
6	7	3	3	2	1	0	0
7	8	3	3	2	1	1	0
8	13	4	4	3	2	1	0
9	20	4	4	3	2	1	1
10	22	4	4	3	3	3	1



Точнее, рассматривается L списков монет, по одному для каждого номинала. Эти списки сортируются в порядке возрастания нумизматической ценности. На самом деле, поскольку в данном случае $L = O(\log n)$, сортировку можно выполнить, не увеличивая вычислительную сложность, так что для нахождения решения нужно время и память порядка $O(nL)$.

На последнем этапе обрабатываются первые $2n - 2$ элементов списка, соответствующего номиналу 2^{-1}. В этих элементах каждое вхождение исходной частоты отвечает за одну единицу длины ассоциированного кодового слова.

Пусть $(i, l) \in [1, n]\times[1, L]$ – узел веса w_i и ширины 2^{-l}. Вес (соответственно ширина) множества узлов равен сумме их весов (соответственно ширин). Определим для двоичного дерева T с n листьями множество узлов $nodeset(T)$ следующим образом: $nodeset(T) = \{(i, l) : 1 \le i \le n, 1 \le l \le l_i\}$, где l_i – глубина i-го листа T.

Таким образом, вес $nodeset(T)$ равен $weight(T) = \sum_n^{i=1} w_i l_i$, а его ширина равна $width(T) = n - 1$ (доказывается по индукции).

***Лемма* 10.** *Первые* $2n - 2$ *элементов последнего списка, вычисленного алгоритмом* PackageMerge, *примененным к* L *спискам* n *монет, отсортированным в порядке возрастания их нумизматической ценности* w_i, $1 \le i \le n$, *соответствуют множеству узлов ширины* $n - 1$, *имеющему минимальный вес.*

Доказательство. Пусть $C = (k_1, k_2, \ldots, k_n)$ – длины кодовых слов, порожденных алгоритмом PackageMerge. Обозначим $K = \sum_n^{i=1} 2^{-k_i}$. В начальный момент $C = (0,0,\ldots,0)$ и $K = n$. Легко проверить, что каждый элемент из числа первых $2n - 2$ элементов, порожденных алгоритмом, уменьшает K на 2^{-1}. Таким образом, порожденное множество узлов имеет ширину $n - 1$. Также легко проверяется, что алгоритм на каждом шаге жадно выбирает наименьший вес, поэтому полный вес порожденного множества узлов минимален. ■

Множество узлов Z называется монотонным, если выполняются следующие два условия:

- $(i, l) \in Z \Rightarrow (i + 1, l) \in Z$ для $1 \le i < n$,
- $(i, l) \in Z \Rightarrow (i, l - 1) \in Z$ для $l > 1$.

Следующие леммы легко доказываются.

***Лемма* 11.** *Для целого* $X < n$ *множество узлов ширины* X *и минимального веса монотонно.*

***Лемма* 12.** *Если* $(l_1, l_2, \ldots, l_n)$ *– список целых чисел таких, что* $1 \le l_i \le L$, *и* Z *– множество узлов* $\{(i, l) : 1 \le i \le n, 1 \le l \le l_i\}$, то $width(Z) = n - \sum_n^{i=1} 2^{-l_i}$.

***Лемма* 13.** *Если* $y = (l_1, l_2, \ldots, l_n)$ *– монотонный возрастающий список неотрицательных целых чисел ширины 1, то* y *является списком глубин листьев некоторого двоичного дерева.*

Теперь можно сформулировать основную теорему.

***Теорема* 14.** *Если множество узлов* Z *имеет минимальный вес среди всех множеств узлов ширины* $n - 1$, *то* Z *– множество узлов дерева* T, *являющегося оптимальным решением задачи о кодировании Хаффмана с ограничением длины.*

Доказательство. Пусть Z – множество узлов минимального веса ширины $n - 1$. Обозначим l_i наибольший уровень такой, что $(i, l_i) \in A$ для любого $1 \le i \le n$. По лемме 11, Z монотонно. Таким образом, $l_i \le l_{i+1}$ для $1 \le i < n$. Так как Z монотонно и имеет ширину $n - 1$, то, по лемме 12, $\sum_n^{i=1} 2^{-l_i} = 1$. Тогда, по лемме 13, $(l_1, l_2, \ldots, l_n)$ – список глубин листьев некоторого двоичного дерева T, и, следовательно, $Z = nodeset(T)$.

Так как Z имеет минимальный вес среди всех множеств узлов ширины $n - 1$, то T оптимально. ■

Примечания

Задача о нумизмате и алгоритм PackageMerge впервые описаны в работе Larmore and Hirschberg [172]. Там же показано, что нахождение оптимального

кода Хаффмана с ограничением длины можно свести к задаче о нумизмате и решить ее с временной и пространственной сложностью $O(nL)$. Далее авторы показали, что пространственную сложность можно уменьшить до $O(n)$. Другие усовершенствования можно найти в работах [156] и [220].

101. Динамическое кодирование Хаффмана

У статического метода сжатия Хаффмана есть два основных недостатка. Во-первых, если частоты букв в исходном тексте заранее неизвестны, то этот текст придется просматривать дважды. Во-вторых, дерево Хаффмана необходимо включать в сжатый файл. В этой задаче продемонстрировано решение, свободное от обоих недостатков.

Это решение основано на динамическом методе, при котором дерево кодирования обновляется после чтения каждого символа из исходного текста. Текущее дерево Хаффмана относится к уже обработанной части текста и изменяется точно так же в процессе декодирования.

Вопрос. Спроектировать метод сжатия Хаффмана, который читает текст только один раз и не требует хранения дерева кодирования в сжатом тексте.

[**Указание:** деревья Хаффмана обладают *свойством братства*.]

Свойство братства. Пусть T – дерево Хаффмана с n листьями (полное двоичное взвешенное дерево, в котором веса всех листьев положительны). Узлы T можно организовать в виде списка $(t_0, t_1, \ldots, t_{2n-2})$, удовлетворяющего следующим условиям:

- узлы расположены в порядке убывания весов:

  $$weight(t_0) \geq weight(t_1) \geq \ldots \geq weight(t_{2n-2});$$

- для любого i, $0 \leq i \leq n-2$, соседние узлы t_{2i} и t_{2i+1} являются братьями (т. е. имеют общего родителя).

Решение

При кодировании и декодировании инициализируется динамическое дерево Хаффмана, состоящее из одного узла с весом 1, с которым ассоциирован искусственный символ `ART`.

Этап кодирования. В процессе кодирования при каждом чтении символа a из исходного текста его кодовое слово из дерева дописывается в конец выхода. Однако это случается, только если a уже встречался ранее. В противном случае выводится код `ART`, а за ним исходное кодовое слово a. После этого дерево модифицируется следующим образом: во-первых, если a не является листом дерева, то вставляется новый узел, который становится родителем листа `ART`, а его вторым дочерним узлом будет новый узел, помеченный символом a; во-вторых, дерево обновляется (см. ниже), так что оказывается деревом Хаффмана для нового префикса текста.

Этап декодирования. В процессе декодирования сжатый текст разбирается с помощью дерева кодирования. Корень инициализируется символом `ART`, как в алгоритме кодирования, и становится текущим узлом, после чего дерево эволюционирует симметрично. Если из сжатого файла прочитан `0`, то производится спуск по левой ветви, а если `1`, то по правой. Если текущий узел является листовым, то ассоциированный с ним символ дописывается в конец выхода и дерево обновляется точно так же, как на этапе кодирования.

Обновление. На этапе кодирования (соответственно декодирования), когда прочитан символ (соответственно код) a, текущее дерево необходимо обновить, чтобы оно правильно отражало частоту символов. При рассмотрении очередного входного символа вес ассоциированного с ним листа увеличивается на 1, и веса предков должны быть модифицированы соответственно.

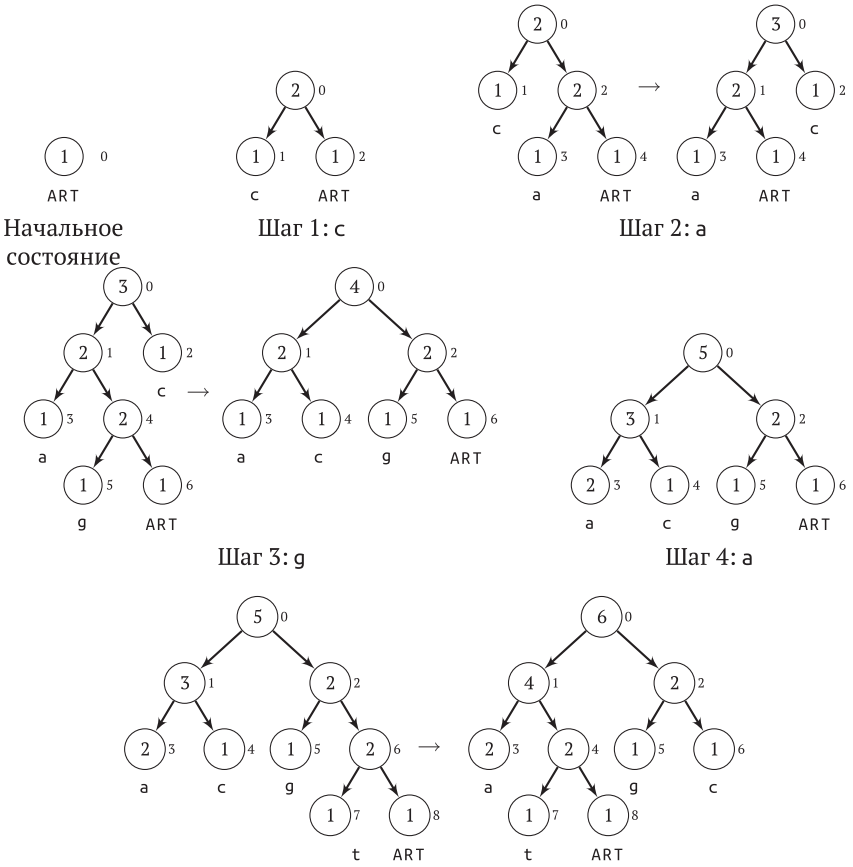
Сначала вес листа t_q, соответствующего a, увеличивается на 1. Затем, если первое условие свойства братства больше не выполняется, узел t_q меняется местами с ближайшим в списке узлом t_p ($p < q$), для которого $weight(t_p) < weight(t_q)$. Это действие заключается в перестановке поддеревьев с корнями в узлах t_p и t_q. При этом узлы по-прежнему расположены в порядке убывания весов. Далее операция повторяется для родителя t_p, пока не будет достигнут корень дерева.

Эта стратегия реализована в следующем алгоритме.

```
UPDATE(a)
 1  t_q ← leaf(a)
 2  while t_q ≠ root do
 3      weight(t_q) ← weight(t_q) + 1
 4      p ← q
 5      while weight(t_{p-1}) < weight(t_q) do
 6          p ← p − 1
 7      обменять местами узлы t_p и t_q
 8      t_q ← parent(t_p)
 9  weight(root) ← weight(root) + 1
```

Набросок доказательства. Предположим, что свойство братства выполняется для дерева Хаффмана со списком узлов $(t_0, t_1, ..., t_q, ..., t_{2n-2})$, расположенных в порядке убывания весов, и пусть вес листа t_q увеличивается на 1. Тогда из неравенств $weight(t_p) \geq weight(t_q)$ и $weight(t_p) < weight(t_q) + 1$ следует, что $weight(t_p) = weight(t_q)$. Веса узлов t_p и t_q одинаковы, поэтому t_p не может быть ни родителем, ни предком t_q, т. к. вес родителя равен сумме весов его дочерних узлов, а эти веса положительны. Обмен t_q с ближайшим узлом t_p таким, что $weight(t_p) = weight(t_q)$, увеличение $weight(t_q)$ на 1 и применение той же процедуры к родителю t_p и далее до корня восстанавливает свойство братства для всего дерева, превращая его в дерево Хаффмана.

На рисунке показано обновление дерева на первых пяти шагах обработки входного текста `cagataagagaa`.



Начальное состояние | Шаг 1: c | Шаг 2: a | Шаг 3: g | Шаг 4: a | Шаг 5: t

Примечания

Представленный здесь динамический вариант метода сжатия Хаффмана был открыт независимо Фоллером [108] и Галлагером [125]. Практические реализации были предложены в работах Cormack and Horspool [62] и Knuth [161]. Точный анализ, позволивший улучшить длину сжатого текста, был выполнен в работе Vitter [236].

Существует множество вариантов кодирования Хаффмана; см., например, [121].

102. Кодирование длинами серий

Двоичное представление (разложение) целого положительного числа x обозначается $r(x) \in \{0,1\}^*$. Кодирование слова $w \in 1\{0,1\}^*$ длинами серий имеет вид $1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}}$, где $s - 2 \geq 0$, p_i – целые положительные числа, $i = 0, \ldots, s - 2$ и $p_{s-1} \geq 0$. Значение s называется длиной серии w.

В этой задаче мы изучим длину серии двоичного представления суммы, разности и произведения двух целых чисел.

Вопрос. Пусть x и y – целые числа, $x \geq y > 0$. Обозначим n суммарную длину серий $r(x)$ и $r(y)$. Показать, что длины серий $r(x + y)$, $r(x - y)$ и $r(x \times y)$ полиномиально зависят от n.

Решение
Пусть $r(x) = 1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}}$, $r(y) = 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}0^{q_{t-1}}$.

Длина серии $r(x + y)$**.** Докажем индукцией по n, что длина серии $r(x + y)$ полиномиально зависит от n.

Очевидно, что предположение индукции справедливо при $n = 2$, когда $s = 1$ или $t = 1$. Предположим, что оно верно, когда суммарная длина серий $r(x)$ и $r(y)$ равна $k < n$, и пусть теперь суммарная длина серий $r(x)$ и $r(y)$ равна n.

- Случай $p_{s-1} \neq 0$ и $q_{t-1} \neq 0$.
  Без ограничения общности можно предположить, что $p_{s-1} \geq q_{t-1}$. Тогда

  $$r(x + y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}-q_{t-1}} + 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}) \cdot 0^{q_{t-1}}.$$

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}-q_{t-1}}$ и $1^{q_0}0^{q_1}\ldots1^{q_{t-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их суммы полиномиально зависит от n.

  Пример. $r(x) = 1^30^21^20^3$ и $r(y) = 1^10^31^30^2$. Тогда $r(x + y) = (1^30^21^20^1 + 1^10^31^3) \cdot 0^2 = 1^10^21^10^11^20^11^10^2$.

  $$\begin{array}{r} 1\,1\,1\,0\,0\,1\,1\,0\,0\,0 \\ +\quad 1\,0\,0\,0\,1\,1\,1\,0\,0 \\ \hline 1\,0\,0\,1\,0\,1\,1\,0\,1\,0\,0 \end{array} \quad = \quad \begin{array}{r} 1\,1\,1\,0\,0\,1\,1\,0 \\ +\quad 1\,0\,0\,0\,1\,1\,1 \\ \hline 1\,0\,0\,1\,0\,1\,1\,0\,1{\cdot}0\,0 \end{array}$$

- Случай $p_{s-1} = 0$ и $q_{t-1} \neq 0$.
  Если $p_{s-2} \geq q_{t-1}$, то

  $$r(x + y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-1}} + 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}) \cdot 1^{q_{t-1}}.$$

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-1}}$ и $1^{q_0}0^{q_1}\ldots1^{q_{t-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их суммы полиномиально зависит от n.

  Пример. $r(x) = 1^30^21^50^0$ и $r(y) = 1^10^31^30^2$. Тогда $r(x + y) = (1^30^21^3 + 1^10^31^3) \cdot 1^2 = 1^10^21^10^11^30^11^2$.

  $$\begin{array}{r} 1\,1\,1\,0\,0\,1\,1\,1\,1\,1 \\ +\quad 1\,0\,0\,0\,1\,1\,1\,0\,0 \\ \hline 1\,0\,0\,1\,0\,1\,1\,1\,0\,1\,1 \end{array} \quad = \quad \begin{array}{r} 1\,1\,1\,0\,0\,1\,1\,1 \\ +\quad 1\,0\,0\,0\,1\,1\,1 \\ \hline 1\,0\,0\,1\,0\,1\,1\,1\,0{\cdot}1\,1 \end{array}$$

  Если $p_{s-2} < q_{t-1}$, то

  $$r(x + y) = (1^{p_0}0^{p_1}\ldots0^{p_{s-3}} + 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}) \cdot 1^{p_{s-2}}.$$

Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots0^{p_{s-3}}$ и $1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}$, по предположению индукции, не превосходит $n-1$, длина серии их суммы полиномиально зависит от n.

Пример. $r(x) = 1^3 0^2 1^3 0^1 1^1 0^0$ и $r(y) = 1^1 0^3 1^3 0^2$. Тогда $r(x+y) = (1^3 0^2 1^3 0^1 + 1^1 0^3 1^3 0^1)\cdot 1^1 = 1^1 0^2 1^1 0^1 1^3 0^2 1^1$.

```
   1 1 1 0 0 1 1 1 0 1          1 1 1 0 0 1 1 1 0
+    1 0 0 0 1 1 1 0 0    =  +    1 0 0 0 1 1 1 0
  1 0 0 1 0 1 1 1 0 0 1        1 0 0 1 0 1 1 1 0 0·1
```

- Случай, когда $p_{s-1} \neq 0$ и $q_{t-1} = 0$, рассматривается аналогично.
- Случай $p_{s-1} = 0$ и $q_{t-1} = 0$.
  Без ограничения общности можно предположить, что $p_{s-2} \geq q_{t-2}$. Тогда

  $r(x+y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-2}} + 1^{q_0}0^{q_1}\ldots0^{q_{t-3}} + 1)\cdot 1^{q_{t-2}-1}0$.

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-2}}$ и $1^{q_0}0^{q_1}\ldots0^{q_{t-3}}$, по предположению индукции, не превосходит $n-1$, длина серии их суммы полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^5 0^0$ и $r(y) = 1^1 0^5 1^3 0^0$. Тогда $r(x+y) = (1^3 0^2 1^2 + 1^1 0^5 + 1)\cdot 1^2 0^1 = 1^1 0^2 1^1 0^1 1^1 0^2 1^2 0^1$.

```
   1 1 1 0 0 0 1 1 1 1          1 1 1 0 0 1 1
+    1 0 0 0 0 0 1 1 1    =  +    1 0 0 0 0 0
                            +              1
  1 0 0 1 0 1 0 0 1 1 0        1 0 0 1 0 1 0 0·1 1 0
```

Итак, во всех случаях утверждение для $r(x+y)$ доказано.

Длина серии $r(x-y)$**.** Докажем индукцией по n, что длина серии $r(x-y)$ полиномиально зависит от n.

Очевидно, что предположение индукции справедливо при $n = 2$. Предположим, что оно верно, когда суммарная длина серий $r(x)$ и $r(y)$ равна $k < n$, и пусть теперь x и y таковы, что суммарная длина серий $r(x)$ и $r(y)$ равна n.

- Случай $p_{s-1} \neq 0$ и $q_{t-1} \neq 0$.
  Без ограничения общности можно предположить, что $p_{s-1} \geq q_{t-1}$. Тогда

  $r(x-y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}-q_{t-1}} - 1^{q_0}0^{q_1}\ldots1^{q_{t-2}})\cdot 0^{q_{t-1}}$.

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}}0^{p_{s-1}-q_{t-1}}$ и $1^{q_0}0^{q_1}\ldots1^{q_{t-2}}$, по предположению индукции, не превосходит $n-1$, длина серии их разности полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^2 0^3$ и $r(y) = 1^1 0^3 1^3 0^2$. Тогда $r(x-y) = (1^3 0^2 1^2 0^1 - 1^1 0^3 1^3)\cdot 0^2 = 1^1 0^2 1^5 0^2$.

```
   1 1 1 0 0 1 1 0 0 0          1 1 1 0 0 1 1 0
-    1 0 0 0 1 1 1 0 0    =  -    1 0 0 0 1 1 1
   1 0 0 1 1 1 1 1 0 0          1 0 0 1 1 1 1 1·0 0
```

- Случай $p_{s-1} = 0$ и $q_{t-1} \neq 0$.
  Если $p_{s-2} \geq q_{t-1}$, то

  $r(x - y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-1}} - 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}) \cdot 1^{q_{t-1}}$.

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-1}}$ и $1^{q_0}0^{q_1}\ldots1^{q_{t-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их разности полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^5 0^0$ и $r(y) = 1^1 0^3 1^3 0^2$. Тогда $r(x - y) = (1^3 0^2 1^3 - 1^1 0^3 1^3) \cdot 1^2 = 1^1 0^2 1^1 0^5 1^2$.

```
    1 1 1 0 0 1 1 1 1 1          1 1 1 0 0 1 1 1
-     1 0 0 0 1 1 1 0 0    =  -     1 0 0 0 1 1 1
  -------------------------    -------------------
    1 0 1 0 0 0 0 0 1 1          1 0 1 0 0 0 0 0·1 1
```

  Если $p_{s-2} < q_{t-1}$, то

  $r(x - y) = (1^{p_0}0^{p_1}\ldots0^{p_{s-3}} - 1^{q_0}0^{q_1}\ldots1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}) \cdot 1^{q_{t-1}}$.

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots0^{p_{s-3}}$ и $1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их разности полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^2 0^1 1^2 0^0$ и $r(y) = 1^1 0^3 1^2 0^3$. Тогда $r(x - y) = (1^3 0^2 1^1 0^1 - 1^1 0^3 1^2 0^1) \cdot 1^2 = 1^1 0^1 1^1 0^5 1^2$.

```
    1 1 1 0 0 1 1 0 1 1          1 1 1 0 0 1 1 0
-     1 0 0 0 1 1 0 0 0    =  -     1 0 0 0 1 1 0
  -------------------------    -------------------
    1 0 1 0 0 0 0 0 1 1          1 0 1 0 0 0 0 0·1 1
```

- Случай, когда $p_{s-1} \neq 0$ и $q_{t-1} = 0$, рассматривается аналогично.
- Случай $p_{s-1} = 0$ и $q_{t-1} = 0$.
  Если $p_{s-2} \geq q_{t-2}$, то

  $r(x - y) = (1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-2}} - 1^{q_0}0^{q_1}\ldots0^{q_{t-3}}) \cdot 0^{q_{t-2}}$.

  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots1^{p_{s-2}-q_{t-2}}$ и $1^{q_0}0^{q_1}\ldots0^{q_{t-3}}$, по предположению индукции, не превосходит $n - 1$, длина серии их разности полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^5 0^0$ и $r(y) = 1^1 0^3 1^2 0^1 1^2 0^0$. Тогда $r(x - y) = (1^3 0^2 1^3 - 1^1 0^3 1^2 0^1) \cdot 0^2 = 1^1 0^1 1^1 0^5 1^2$.

```
    1 1 1 0 0 1 1 1 1 1          1 1 1 0 0 1 1 1
-     1 0 0 0 1 1 0 1 1    =  -     1 0 0 0 1 1 0
  -------------------------    -------------------
    1 0 1 0 0 0 0 1 0 0          1 0 1 0 0 0 0 1·0 0
```

  Если $p_{s-2} < q_{t-2}$, то $r(x - y) = (1^{p_0}0^{p_1}\ldots0^{p_{s-3}} - 1^{q_0}0^{q_1}\ldots1^{q_{t-2}-p_{s-2}}) \cdot 0^{q_{t-2}}$.
  Так как суммарная длина серий $1^{p_0}0^{p_1}\ldots0^{p_{s-3}}$ и $1^{q_0}0^{q_1}\ldots1^{q_{t-2}-p_{s-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их разности полиномиально зависит от n.

Пример. $r(x) = 1^3 0^2 1^1 0^1 1^3 0^0$ и $r(y) = 1^1 0^3 1^5 0^0$. Тогда $r(x - y) = (1^3 0^2 1^1 0^1 - 1^1 0^3 1^2) \cdot 0^3 = 1^1 0^2 1^4 0^3$.

```
   1 1 1 0 0 1 0 1 1 1          1 1 1 0 0 1 0 1
-    1 0 0 0 1 1 1 1 1    =  -    1 0 0 0 1 1 1
   1 0 0 1 1 1 1 0 0 0          1 0 0 1 1 1 1 0·0 0
```

Итак, во всех случаях утверждение для $r(x - y)$ доказано.

Длина серии $r(x \times y)$**.** Докажем индукцией по n, что длина серии $r(x \times y)$ полиномиально зависит от n.

Очевидно, что предположение индукции справедливо при $n = 2$. Предположим, что оно верно, когда суммарная длина серий $r(x)$ и $r(y)$ равна $k < n$, и пусть теперь x и y таковы, что суммарная длина серий $r(x)$ и $r(y)$ равна n.

- Случай $p_{s-1} \neq 0$. Тогда

  $r(x \times y) = (1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}} \times 1^{q_0} 0^{q_1} \ldots 1^{q_{t-2}} 0^{q_{t-1}}) \cdot 0^{p_{s-1}}$.

  Так как суммарная длина серий $1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}}$ и $1^{q_0} 0^{q_1} \ldots 0^{q_{t-1}}$, по предположению индукции, не превосходит $n - 1$, длина серии их произведения полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^2 0^3$ и $r(y) = 1^1 0^3 1^5 0^0$. Тогда $r(x \times y) = (1^3 0^2 1^2 \times 1^1 0^3 1^5) \cdot 0^3$.

```
   1 1 1 0 0 1 1 0 0 0          1 1 1 0 0 1 1
×    1 0 0 0 1 1 1 1 1    =  × 1 0 0 0 1 1 1 1 1
                                              ·0 0 0
```

- Случай, когда $q_{t-1} \neq 0$, рассматривается аналогично.
- Случай $p_{s-1} = 0$ и $q_{t-1} = 0$. Тогда

  $r(x \times y) = (1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}-q_{t-1}} \times 1^{q_0} 0^{q_1} \ldots 0^{q_{t-3}+q_{t-2}}) + (1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}} \times 1^{q_{t-2}})$.

  Так как суммарная длина серий $1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}}$ и $1^{q_0} 0^{q_1} \ldots 0^{q_{t-3}+q_{t-2}}$, по предположению индукции, не превосходит $n - 1$, длина серии их произведения полиномиально зависит от n. А так как длина серий $1^{p_0} 0^{p_1} \ldots 1^{p_{s-2}}$ и $1^{q_{t-2}}$, по предположению индукции, не превосходит n, то длина серии их произведения полиномиально зависит от n.

  Пример. $r(x) = 1^3 0^2 1^2 0^0$ и $r(y) = 1^1 0^2 1^3 0^0$. Тогда $r(x \times y) = (1^3 0^2 1^2 \times 1^1 0^5) + (1^3 0^2 1^2 \times 1^3)$.

```
  1 1 1 0 0 1 1          1 1 1 0 0 1 1          1 1 1 0 0 1 1
×   1 0 0 1 1 1    =   ×   1 0 0 0 0 0    +   ×         1 1 1
```

Итак, во всех случаях утверждение для $r(x \times y)$ доказано.

Примечания

Мы можем также рассмотреть арифметические операции над краткими представлениями чисел в десятичной системе счисления. Например:

$1^{5^n}/41 = 271(00271)^{n-1}$.

Однако это не кодирование длинами серий, а его обобщение.

103. Компактный факторный автомат

Факторным автоматом называется минимальный детерминированный автомат, допускающий все факторы слова. Он также называется ориентированным ациклическим графом слов (directed acyclic word graph – DAWG). Все его состояния заключительные, а ребра помечены одиночными буквами. Для некоторых слов с хорошей структурой автомат можно сильно сжать, удалив вершины, имеющие ровно одного родителя и одну дочернюю вершину, и соответственно пометив ребра факторами слова. Результирующий автомат называют компактным DAWG (CDAWG) или компактным суффиксным автоматом (CSA), если вершины, соответствующие суффиксам, помечены как заключительные.

В этой задаче рассматриваются слова, для которых CDAWG особенно малы, а именно слова Фибоначчи fib_n и их укороченные версии g_n. Слово g_n получается из fib_n удалением двух последних букв, т. е. $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$.

Вопрос. Описать структуру CDAWG слов Фибоначчи fib_n и их укороченных версий g_n. Пользуясь этой структурой, вычислить количество различных факторов, встречающихся в словах.

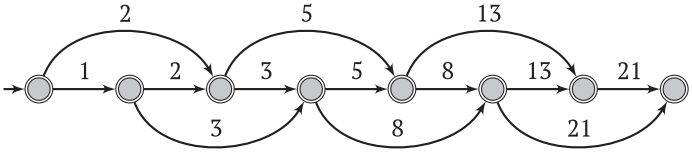
Решение

Решение основано на ленивой системе счисления Фибоначчи и использует тот факт, что любое целое число $x \in [1..F_n - 2]$, $n \geq 4$, единственным образом представляется в виде $x = F_{i_0} + F_{i_1} + \ldots + F_{i_k}$, где $(F_{i_t} : 2 \leq i_t \leq n - 2)$ – возрастающая последовательность чисел Фибоначчи, удовлетворяющая условию

$$i_0 \in \{0,1\} \text{ и } i_t \in \{i_{t-1} + 1, i_{t-1} + 2\} \text{ для } t > 0. \qquad (*)$$

Так, для $n = 8$ числу 13 соответствует последовательность индексов (3,4,6), потому что $F_3 + F_4 + F_6 = 2 + 3 + 8 = 13$.

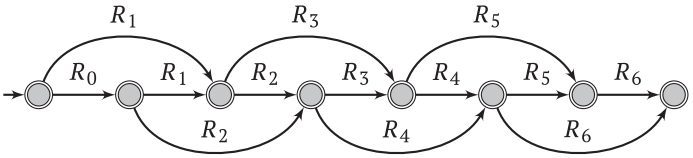
Множество последовательностей $(F_{i_t}: 2 \leq i_t \leq n - 2)$, удовлетворяющих условию $(*)$, допускается простым детерминированным ациклическим автоматом, ребра которого помечены числами Фибоначчи, а все состояния заключительные. На рисунке показан случай $n = 10$ для целых чисел в диапазоне [1..53].



CDAWG укороченных слов Фибоначчи. Показанный выше автомат легко преобразовать в CDAWG чисел g_n, воспользовавшись следующим свойством (см. задачу 56). Обозначим R_i обращение fib_i, и пусть $suf(k, n)$ – k-й суффикс числа g_n, $g_n[k..|g_n| - 1]$.

Свойство. Для $n > 2$ $suf(k, n)$ единственным образом разлагается в произведение $R_{i_0}R_{i_1}...R_{i_m}$, где $i_0 \in \{0,1\}$ и $i_t \in \{i_{t-1} + 1, i_{t-1} + 2\}$ для $t > 0$.

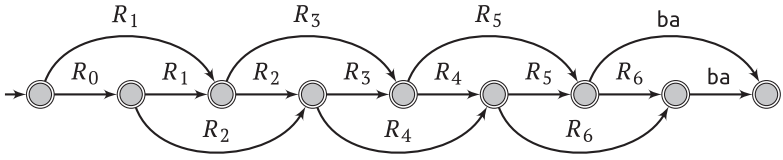
Благодаря этому свойству описанный выше автомат преобразуется в CDAWG(g_n) подстановкой R_i вместо каждого числа Фибоначчи F_i. На рисунке ниже показан CDAWG(g_{10}), полученный преобразованием предыдущего рисунка.



Подсчет факторов в укороченных словах Фибоначчи. CDAWG полезен для вычисления количества различных непустых факторов, встречающихся в соответствующем слове. Действительно, эта величина равна сумме длин ребер, умноженных на число путей, содержащих эти ребра. На самом деле число факторов в g_n равно $F_{n-1}F_{n-2} - 1$, этот результат получается с помощью формулы $F_2^2 + F_3^2 + ... + F_{n-2}^2 = F_{n-1}F_{n-2} - 1$.

Для CDAWG(g_{10}), показанного на рисунке выше, имеем $1^2 + 2^2 + 3^2 + 5^2 + 8^2 + 13^2 + 21^2 = 21 \times 34 - 1 = 713$ непустых факторов.

CDAWG слов Фибоначчи. Компактный DAWG слова Фибоначчи fib_n лишь немного отличается от CDAWG укороченного слова Фибоначчи g_n. Нужно только добавить две последние отброшенные буквы, что ведет к простой модификации CDAWG(g_n), показанной на рисунке на примере получения CDAWG(fib_{10}). Эта сжатая структура представляет все 781 фактор fib_{10}.



Число факторов слова Фибоначчи fib_n немного больше, чем у числа g_n, поскольку нужно учитывать последние две буквы на обоих ребрах, входящих в последнюю вершину. Для $n > 2$ слово fib_n содержит $F_{n-1}F_{n-2} + 2F_{n-1} - 1$ непустых факторов.

В примере для $n = 10$ дополнительное слово равно `ba`. Оно встречается на 34 путях, входящих в последнюю вершину, поэтому нужно добавить $2 \cdot 34 = 68$ факторов. Следовательно, fib_{10} содержит $713 + 68 = 781$ непустой фактор.

Примечания

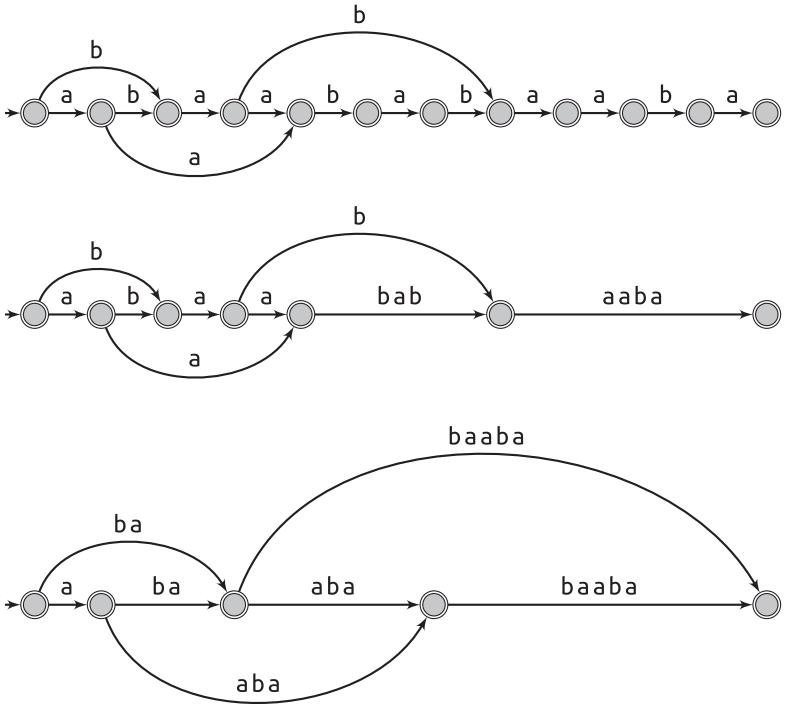
Структура CDAWG слов Фибоначчи описана в работе [213]. Другие сильно сжатые и полезные DAWG встречаются в более общем контексте слов Штурма, см. [27]. Число вершин в структурах отражает объем необходимой для

их хранения памяти, потому что метки можно представить парой индексов позиций в соответствующем слове.

Суффиксный или факторный автомат слова длины l имеет по меньшей мере $l + 1$ состояний. На самом деле над двоичным алфавитом нижняя граница достигается, только когда слово является префиксом слова Штурма, каковыми являются слова Фибоначчи (см. [221]).

Как было сказано в начале раздела, простейшая стратегия сжатия DAWG – удалить вершины, имеющие только одного предшественника и только одного преемника (см. [38, 101, 150]). Описанный выше метод для факторов чисел Фибоначчи дает CDAWG, который не только меньше, но и обладает более полезной структурой.

Ниже показаны суффиксный автомат слова g_7 длины 11 с 12 состояниями, его обыкновенная компактная версия с 7 вершинами и компактная версия, полученная применением описанной выше техники, которая содержит только 5 вершин.



Конечные слова Туэ–Морса тоже имеют очень короткое описание (см. [204]), из которого легко заключить, что число факторов слова Туэ–Морса длины $n \geq 16$ равно $\frac{73}{192}n^2 + \frac{8}{3}$.

104. Сжатое сопоставление в слове Фибоначчи

Под *сжатым сопоставлением* понимается следующая задача: даны компактные представления образца и текста, а требуется быстро найти образец с учетом особенностей сжатия. Размер представления может логарифмически зависеть от размера реальных входных данных, именно так обстоит дело в примере ниже.

Вход зависит от типа сжатого представления. Мы рассмотрим очень простой случай, когда образец является конкатенацией слов Фибоначчи, а его представление – последовательность их индексов. Текстом, в котором производится поиск, является бесконечное слово Фибоначчи $\mathbf{f} = \varphi^{\infty}(\mathtt{a})$, где φ – морфизм, определенный следующим образом: $\varphi(\mathtt{a}) = \mathtt{ab}$, $\varphi(\mathtt{b}) = \mathtt{a}$.

В список слов Фибоначчи добавляется еще слово b, которому присваивается индекс –1: $fib_{-1} = \mathtt{b}$, $fib_0 = \mathtt{a}$, $fib_1 = \mathtt{ab}$, $fib_2 = \mathtt{aba}$, $fib_3 = \mathtt{abaab}$, ...

Вопрос. Для заданной последовательности целых чисел $k_1, k_2, \ldots, k_n$ ($k_i \geq -1$) показать, как за время $O(n + k_1 + k_2 + \ldots + k_n)$ проверить, встречается ли слово $fib_{k_1} fib_{k_2} \ldots fib_{k_n}$ в бесконечном слове Фибоначчи **f**.

Решение

На вход алгоритму подается последовательность $w = (k_1, k_2, \ldots, k_n)$ индексов слов Фибоначчи. Обозначим *first*(w) и *last*(w) первый и последний элементы w соответственно.

```
CompressedMatch(последовательность w индексов ≥ −1)
 1  while |w| > 1 do
 2      if w содержит фактор (i, − 1), i ∉ {0,2} then
 3          return FALSE
 4      if first(w) = −1 then
 5          first(w) ← 1
 6      if last(w) = 2 then
 7          last(w) ← 1
 8      if last(w) = 0 then
 9          удалить последний элемент
10      изменить все факторы (0, − 1) w на 1
11      изменить все факторы (2, − 1) w на (1, 1)
12      уменьшить все элементы w на 1
13  return TRUE
```

Пример. Для входной последовательности $w = (0,1,3,0,1,4)$ алгоритм выполняет пять итераций и возвращает TRUE:

$(0,1,3,0,1,4) \to (-1,0,2,-1,0,3) \to (0,-1,0,0,-1,2)$
$\to (0,-1,0,0) \to (0,-1) \to (0)$.

Алгоритм CompressedMatch просто реализует следующие наблюдения над последовательностью $w = (k_1, k_2, \ldots, k_n)$.

Случай (i). $fib_i fib_{-1}$ является фактором **f** тогда и только тогда, когда $i = 0$ или $i = 2$. Действительно, если $i = 0$, то $fib_i fib_{-1} =$ ab, а если $i = 2$, то $fib_i fib_{-1} =$ abab, и оба слова встречаются в **f**. В противном случае $fib_i fib_{-1}$ имеет суффикс bb или ababab $= \varphi($aaa$)$, который не встречается в **f**.

Случай (ii). Если $k_1 = -1$, то его можно заменить на 1, потому что первой букве b в **f** должна предшествовать a и $fib_1 =$ ab.

Случай (iii). Аналогично, если $k_n = 2$, то его можно заменить на 1, потому что в **f** за каждым вхождением b следует a, так что суффикс aba можно сократить до ab, отчего конечный результат не изменится.

Случай (iv). Фактор $(0,-1)$ можно заменить на 1, потому что $fib_0 fib_{-1} =$ ab $= fib_1$, а фактор $(2,-1)$ на $(1,1)$, потому что $fib_2 fib_{-1} =$ abab $= fib_1 fib_1$.

Случай (v). Единственный нетривиальный случай – когда $k_n = 0$. Это соответствует совпадению с вхождением a в **f**. Доказательство правильности в этом случае снова вытекает из того факта, что за каждым вхождением b в **f** следует буква a. Мы рассмотрим два подслучая в зависимости от последней буквы предпоследнего фактора Фибоначчи.

Случай, когда $fib_{k_{n-1}}$ кончается буквой b. Тогда $fib_{k_1} fib_{k_2} \ldots fib_{k_n}$ встречается в **f** тогда и только тогда, когда $fib_{k_1} fib_{k_2} \ldots fib_{k_{n-1}}$ встречается в **f**, т. к. за каждым вхождением b следует a. Поэтому последняя a избыточна и может быть удалена.

Случай, когда $fib_{k_{n-1}}$ кончается буквой a. После того как 0 удален и выполнена строка 12, алгоритм проверяет, встречается ли $fib_{k_1-1} fib_{k_2-1} \ldots fib_{k_{n-1}-1}$ в **f**. Однако $fib_{k_{n-1}-1}$ теперь заканчивается буквой b, и $fib_{k_1-1} fib_{k_2-1} \ldots fib_{k_{n-1}-1}$ встречается в **f** тогда и только тогда, когда встречается $v = fib_{k_1-1} fib_{k_2-1} \ldots fib_{k_{n-1}-1}$a. Следовательно, если v встречается в **f**, то слово $fib_{k_1} fib_{k_2} \ldots fib_{k_n}$ встречается в $\varphi(v)$. Это показывает, что последний элемент $k_n = 0$ можно удалить, не ставя правильность под сомнение.

Заметим, что когда алгоритм выполняет инструкцию в строке 12, все индексы в последовательности w неотрицательны. Поэтому только что приведенное рассуждение применимо и после выполнения этой строки. На этом доказательство правильности алгоритма заканчивается.

Что до сложности, то заметим, что на каждой паре соседних итераций сумма индексов уменьшается как минимум на 1. Следовательно, алгоритм просит времени $O(n + k_1 + k_2 + \ldots + k_n)$, что и требуется.

Примечания

Описанный алгоритм был предложен Риттером в качестве задачи для Польской олимпиады по информатике. Совершенно другой алгоритм можно най-

ти в работе [238]. И еще один алгоритм можно получить с использованием сжатых факторных графов слов Фибоначчи.

105. Предсказание по частичному совпадению

Предсказание по частичному совпадению (Prediction by Partial Matching – PPM) – метод сжатия без потери информации, при котором кодировщик поддерживает статистическую модель текста. Цель заключается в том, чтобы предсказать буквы, следующие за данным фактором входного текста. В этой задаче мы изучим структуру данных для хранения модели.

Пусть y – текст, подлежащий сжатию, и предположим, что префикс $y[0..i]$ уже закодирован. Метод PPM сопоставляет каждой букве $a \in A$ вероятность $p(a)$, зависящую от числа вхождений $y[i+1-d..i]\cdot a$ в $y[1..i]$, где d – длина контекста. Затем PPM передает $p(y[i+1])$ адаптивному арифметическому кодировщику, который учитывает вероятности букв. Если $y[i+1-d..i+1]$ не входит в $y[0..i]$, то кодировщик уменьшает значение d, пока не будет найдено вхождение $y[i+1-d..i+1]$ или не окажется, что $d=-1$. В последнем случае буква $y[i+1]$ еще не встречалась. При каждом уменьшении d PPM передает адаптивному арифметическому кодировщику вероятность так называемого «отхода».

PPM* – вариант PPM, в котором не учитывается максимальная длина контекста, а хранятся все контексты. На каждом шаге начальным является самый короткий детерминированный контекст – тот, что соответствует кратчайшему повторяющемуся суффиксу, за которым всегда следует одна и та же буква, – или самый длинный контекст, если такого суффикса не существует.

Вопрос. Спроектировать структуру данных, способную динамически поддерживать число вхождений каждого контекста, и такую, что операции с ней производятся за линейное время для алфавита постоянного размера.

Решение

Решение основано на префиксном дереве. Префиксное дерево для $y[0..i]$ строится по префиксному дереву для $y[0..i-1]$ и по существу состоит из суффиксного дерева слова $y[0..i]^R$.

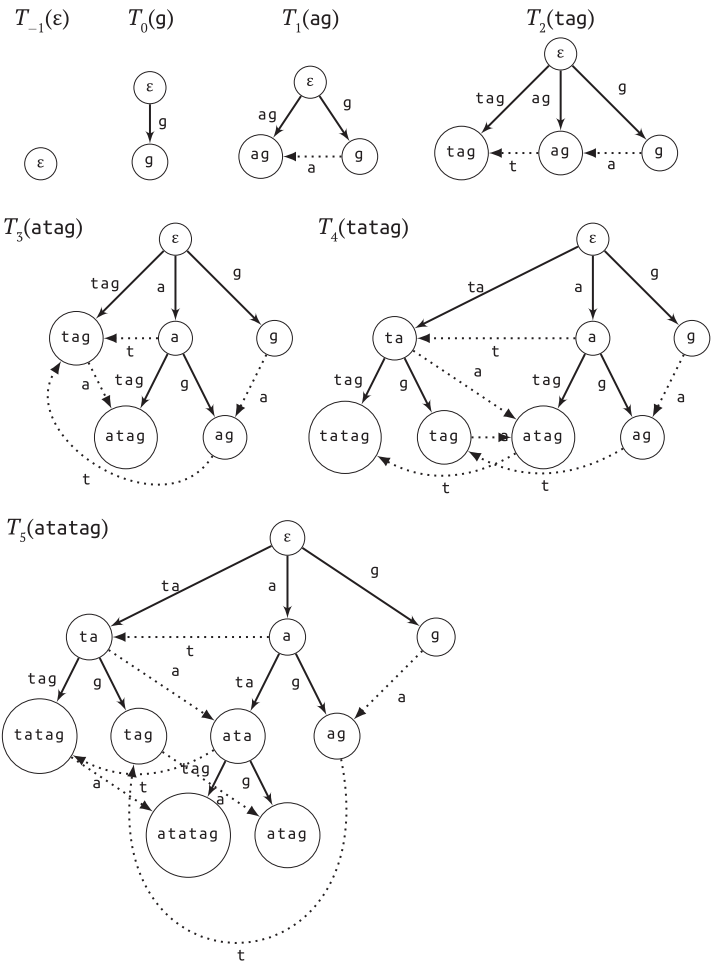
Обозначим T_i префиксное дерево $[0..i]$. Его узлами являются факторы $y[0..i]$. По определению, начальное дерево T_{-1} состоит из одного узла. Для каждого узла T_i, кроме корня и самого недавнего листа, определены префиксные ссылки. Префиксная ссылка, помеченная буквой a, исходящая из узла w, указывает на узел wa или на узел uwa, если любому вхождению wa предшествует u.

Предположим, что префиксное дерево для $y[0..i-1]$ уже построено. Назовем головой w и обозначим $head(w)$ самый длинный суффикс w, для которого имеется внутреннее вхождение в w.

Префиксное дерево обновляется следующим образом. Вставка $y[i]$ начинается с головы $w = y[0..i-1]$ и заканчивается в голове $w' = y[0..i]$. Если $y[i]$ уже встречалась после w, то из узла w исходит префиксная ссылка, помеченная $y[i]$, которая указывает на голову w'. Если из w не исходит префиксной ссылки, помеченной буквой $y[i]$, то алгоритм переходит к родителю w и продолжает поиск, пока либо не найдет префиксную ссылку, помеченную $y[i]$, либо не дойдет до корня дерева. Если достигнутый узел p совпадает с w', то в дерево добавляется только новый лист q. Если достигнутый узел p совпадает с uw' для некоторого $u \in A^+$, то в дерево добавляется новый внутренний узел r и новый лист q.

Все узлы, посещенные в ходе этого процесса, имеют префиксную ссылку, помеченную буквой $y[i]$ и указывающую на новый лист q. Когда создается новый внутренний узел r, некоторые префиксные ссылки, указывавшие на p, возможно, придется обновить, так чтобы они указывали на r.

Пример. На рисунках ниже показана трансформация префиксного дерева при обработке слова $y =$ `gatata`.



***Теорема* 15.** *Описанная процедура правильно вычисляет префиксное дерево T_i по префиксному дереву T_{i-1}.*

Доказательство. T_{i-1} содержит пути, помеченные всеми префиксами слова $w = y[0..i-1]$, и только эти пути. В нем не хватает только пути, помеченного словом $w' = y[0..i]$. Поиск, который начинается с листа s, соответствующего слову w в T_{i-1}, и поднимается вверх до первого узла, имеющего префиксную ссылку, помеченную буквой $a = y[i]$, находит узел t, соответствующий самому длинному суффиксу v слова w – такому, что va является фактором w.

- Если префиксная ссылка, исходящая из t и помеченная a, указывает на узел p, соответствующий va, то в дерево нужно добавить новый лист q, соответствующий w', и ветвь, идущая из p в q, помечается словом u, где $w' = uva$. Все узлы, просмотренные на пути из s в t (кроме самого t), должны теперь иметь префиксную ссылку, помеченную буквой a и указывающую на q.
- Если префиксная ссылка, исходящая из t и помеченная a, указывает на узел p, соответствующий $v'va$, то создается новый внутренний узел r, соответствующий va и имеющий двух преемников: p и новый лист q, соответствующий w'. Ветвь, идущая из r в p, должна быть помечена словом v', а ветвь из r в q – словом u, где $w' = uva$. Все узлы, просмотренные на пути из s в t (кроме самого t), должны теперь иметь префиксную ссылку, помеченную буквой a и указывающую на q. Префиксные ссылки, идущие из узлов v', являющихся суффиксами v, в p, теперь должны указывать на новый внутренний узел r.

В обоих случаях дерево теперь содержит все пути, присутствующие в T_{i-1}, и путь, соответствующий w'. Следовательно, мы построили T_i. ■

***Теорема* 16.** Построение дерева T_{n-1} можно выполнить за время $O(n)$.

Доказательство. Время построения определяется прежде всего числом узлов, посещенных на каждом этапе при вычислении $head(y[0..i])$ для $0 \le i \le n-1$. Обозначим k_i число узлов, посещенных при поиске $head(y[0..i])$. Имеем $|head(y[0..i])| \le |head(y[0..i-1])y[i]| - k_i$. Отсюда $\sum_0^{n-1} k_i = n - |head(y)| \le n$. Таким образом, в процессе построения T_{n-1} посещено не более n узлов, что и доказывает утверждение. ■

Примечания

Алгоритм предсказания по неполному совпадению был спроектирован в работе Cleary and Witten [56] (см. также [190]). Алгоритм PPM* впервые описан в работе [55]. Приведенное здесь построение префиксного дерева взято из работы Effros [107].

106. Сжатие суффиксных массивов

Суффиксные массивы дают простую и экономичную структуру данных для индексирования текстов. К тому же существует много сжатых вариантов суффиксных массивов. В этой задаче обсуждается один из них, в котором

хранится отсортированный (частичный) список суффиксов (точнее, частичный массив рангов) рассматриваемого текста. Это пример применения элементарной теории чисел.

Теоретико-числовые средства. Множество $D \subseteq [0..t-1]$ называется *t-разностным покрытием*, если все элементы отрезка являются разностями элементов D по модулю t:

$$[0..t-1] = \{(x-y) \bmod t : x, y \in D\}.$$

Например, множество $D = \{2,3,5\}$ является 6-разностным покрытием отрезка $[0..6]$, т. к. $1 = 3-2, 2 = 5-3, 3 = 5-2, 4 = 3-5 (\text{mod } 6)$ и $5 = 2-3 (\text{mod } 6)$.

Известно, что для любого целого положительного t существует t-разностное покрытие размера $O(\sqrt{t})$ и что его можно построить за время $O(\sqrt{t})$.

Множество $\mathcal{S}(t) \subseteq [1..n]$ называется *t-покрытием* отрезка $[1..n]$, если $|\mathcal{S}(t)| = O\left(\frac{n}{\sqrt{t}}\right)$ и существует вычислимая за постоянное время функция

$$h : [1..n-t] \times [1..n-t] \to [0..t],$$

удовлетворяющая условиям

$$0 \le h(i,j) \le t \text{ и } i + h(i,j), j + h(i,j) \in \mathcal{S}(t).$$

t-покрытие можно получить из t-разностного покрытия $\mathcal{D}$ отрезка $[0..t-1]$, положив $\mathcal{S}(t) = \{i \in [1..n] : i \bmod t \in \mathcal{D}\}$. Известен следующий факт.

Факт. Для любого $t \le n$ t-покрытие $\mathcal{S}(t)$ можно построить за время $O\left(\frac{n}{\sqrt{t}}\right)$.

Вопрос. Показать, что отсортированный частичный список суффиксов текста длиной n можно представить в памяти объемом всего $O(n^{3/4})$ и при этом сохранить возможность сравнения любых двух суффиксов за время $O(\sqrt{n})$.

[**Указание:** воспользуйтесь понятием t-покрытия отрезков целых чисел.]

Решение

Решение опирается на t-покрытия. Вместо массива SA, в котором хранится отсортированный список суффиксов текста w, мы используем эквивалентный массив Rank, являющийся обращением SA, который дает ранги суффиксов, индексированные начальными позициями. При наличии всего массива сравнение двух суффиксов, начинающихся в позициях i и j, сводится к сравнению их рангов и занимает постоянное время. Но наша цель в этой задаче – оставить только небольшую часть таблицы Rank.

Обозначим $\mathcal{S}$ фиксированное $\sqrt{n}$-покрытие $\{i_1, i_2, \ldots, i_k\}$ отрезка $[1..n]$, где целые числа отсортированы: $i_1 < i_2 < \ldots < i_k$. Его размер имеет порядок $O(n^{3/4})$. Пусть $\mathcal{L}$ – список пар

$((i_1, \text{Rank}[i_1]), (i_2, \text{Rank}[i_2]), \ldots, (i_k, \text{Rank}[i_k]))$.

Поскольку список отсортирован по первой компоненте пары, для проверки того, принадлежит ли $\mathcal{S}$ позиция i, и нахождения ее ранга в $\mathcal{L}$ требуется логарифмическое время.

Пусть требуется лексикографически сравнить суффиксы слова w длины n, начинающиеся в позициях i и j. Обозначим $\Delta = h(i, j)$.

Сначала слова $x[i..i + \Delta - 1]$ и $x[j..j + \Delta - 1]$ сравниваются бесхитростно (побуквенно), что занимает время $O(\Delta)$. Если они совпадают, то остается сравнить суффиксы, начинающиеся в позициях $i + \Delta$ и $j + \Delta$. Это сравнение занимает логарифмическое время, потому что позиции $i + \Delta$ и $j + \Delta$ принадлежат $\mathcal{S}$, и мы можем восстановить ассоциированные с ними ранги из списка $\mathcal{L}$ за логарифмическое время.

В целом сравнение требует времени $O(\sqrt{n})$, т. к. $\Delta = O(\sqrt{n})$.

Пример. Множество $S(6) = \{2,3,5,8,9,11,14,15,17,20,21,23\}$ является 6-покрытием отрезка [1..23], построенным по 6-разностному покрытию $\mathcal{D} = \{2,3,5\}$. В частности, имеем $h(3,10) = 5$, т. к. $3 + 5, 10 + 5 \in \mathcal{S}(6)$.

Если требуется сравнить суффиксы слова w, начинающиеся в позициях 3 и 10, то нужно будет сравнивать только их префиксы длины 5 и, возможно, проверить, верно ли, что $Rank[3 + 5] < Rank[10 + 5]$.

Примечания

Если выбрать $t = n^{2/3}$ вместо $\sqrt{n}$ в определении структуры данных, то объем потребной памяти уменьшится до $O(t)$, но время сравнения двух суффиксов возрастет до $O(t)$.

Построение разностных покрытий описано в работе [178]. Они используются для построения t-покрытий, как сделано, например, в работе [47], где доказан приведенный выше факт.

Похожим методом сжатого индексирования является понятие FM-индекса, основанного на преобразовании Барроуза–Уилера и суффиксных массивах. FM-индексы разработали Феррагина и Манзини (см. [112] и приведенные там ссылки). Их применения к биоинформатике описаны в книге Ohlebusch [196].

107. Коэффициент сжатия жадных суперстрок

В этой задаче рассматривается алгоритм GreedySCS (представленный в других формах в задаче 61), который вычисляет суперстроку $Greedy(X)$ для множества X слов суммарной длины n. Суперстроку можно рассматривать как сжатый текст, представляющий все слова X, и с этой точки зрения интересно получить численную оценку выигрыша от представления X суперстрокой.

Обозначим $GrCompr(X) = n - |Greedy(X)|$ сжатие, достигаемое жадным алгоритмом. Аналогично определим $OptCompr(X) = n - |OPT(X)|$, где OPT – оптимальная (неизвестная) суперстрока для X.

Отношение $GrCompr(X)/OptCompr(X)$ называется коэффициентом сжатия алгоритма GreedySCS.

Вопрос. Показать, что коэффициент сжатия алгоритма GreedySCS не меньше 1/2.

[**Указание:** рассмотрите граф перекрытий входного множества.]

Решение
Удобнее иметь дело с итеративной версией алгоритма GreedySCS из задачи 61.

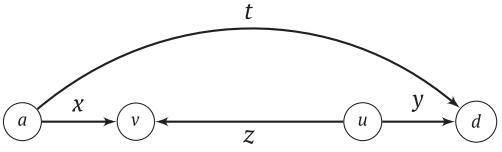
```
IterativeGreedySCS(непустое множество слов X)
1  while |X| > 1 do
2      положить x, y ∈ X, x ≠ y, так что |Overlap(x, y)| максимальна
3      X ← X \ {x, y} ∪ {x ⊗ y}
4  return x ∈ X
```

Начнем с абстрактной задачи для ориентированных графов. Предположим, что G – полный ориентированный граф, для ребер которого определены целые положительные веса. Если $u \to v$ – ребро, то операция стягивания $contract(u, v)$ находит u и v и удаляет ребра, исходящие из u и входящие в v.

Обозначим $OptHam(G)$ максимальный вес гамильтонова пути в G, а $GreedyHam(G)$ – вес гамильтонова пути, неявно порождаемого жадным алгоритмом. На каждом шаге жадный алгоритм на графах выбирает ребро $u \to v$ с максимальным весом и применяет операцию $contract(u, v)$. Алгоритм останавливается, когда в G остается одна вершина. Выбранные ребра образуют гамильтонов путь.

Связь между жадной суперстрокой и жадным гамильтоновым путем. Введем понятие *графа перекрытий* G для множества слов X. Его множество вершин совпадает с X, а весом ребра $x_i \to x_j$ является максимальное перекрытие между словами x_i и x_j. Заметим, что инструкция в строке 3 алгоритма IterativeGreedySCS соответствует операции $contract(x, y)$. Отсюда вытекает следующий факт.

Наблюдение. Жадный гамильтонов путь в графе перекрытий множества X соответствует жадной суперстроке X.



Будем говорить, что взвешенный граф G удовлетворяет условию $(*)$, если в любой конфигурации типа той, что показана на рисунке выше, имеем

$$z \geq x, y \Rightarrow z + t \geq x + y.$$

Кроме того, мы требуем, чтобы это условие удовлетворялось для любого графа, полученного из G применением любого числа стягиваний.

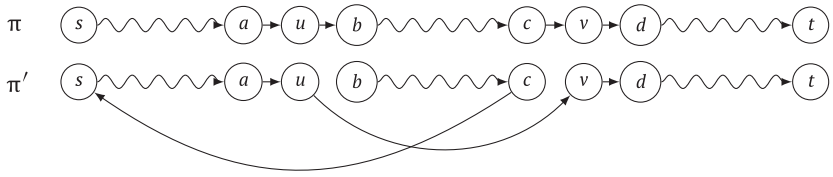
Оставляем читателю несложное доказательство следующего технического утверждения о графах перекрытий (см. примечания).

***Лемма* 17.** *Граф перекрытий удовлетворяет условию* $(*)$.

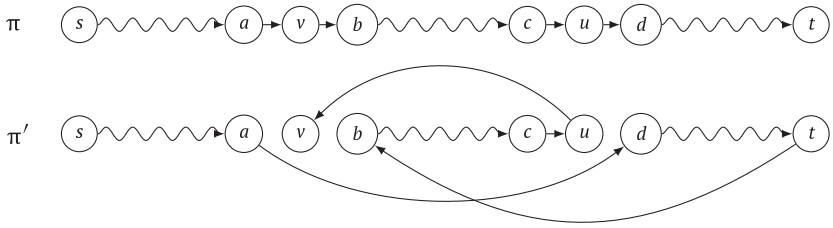
***Лемма* 18.** *Предположим, что G удовлетворяет условию* $(*)$, *e – ребро максимального веса z и G' получен из G стягиванием* (e). Тогда $OptHam(G') \geq OptHam(G) - 2z$.

Доказательство. Пусть $e = u \to v$. Обозначим π оптимальный гамильтонов путь в G, а $|\pi|$ – полный вес π. Достаточно показать, что вес любого гамильтонова пути в G' не меньше $|\pi| - 2z$ или, эквивалентно, что полный вес любого гамильтонова пути π' в G, содержащего ребро $u \to v$, не меньше $|\pi| - z$.

Случай, когда v встречается после u в π. Удалим два ребра (u, b) и (c, v), веса которых не превосходят z, и вставим ребра (u, v) и (c, s), получив тем самым новый гамильтонов путь π' (см. рисунок ниже). Стягивание (u, v) уменьшает полный вес пути на сумму весов (u, b) и (c, v), т. е. не более чем на $2z$. Мы получили гамильтонов путь в G' с полным весом не более $|\pi| - 2z$; см. рисунок ниже. Следовательно, $OptHam(G') \geq OptHam(G) - 2z$.



Случай, когда v встречается раньше u в π. Воспользуемся условием $(*)$, где $x = weight(a, v)$, $y = weight(u, d)$, $z = weight(u, v)$ и $t = weight(a, d)$. Положим $q = weight(v, b)$. Пусть π' получен из π, как на рисунке ниже.



В силу условия $(*)$ и неравенства $q \leq z$ имеем

$$|\pi'| \geq |\pi| - x - y + z + t - q \geq |\pi| - z.$$

Следовательно, $|\pi'| \geq |\pi| - 2z$ и $OptHam(G') \geq OptHam(G) - 2z$. На этом доказательство леммы 18 завершается. ■

***Лемма* 19.** *Если G удовлетворяет условию* $(*)$, *то* $GreedyHam(G) \geq ½\ OptHam(G)$.

Доказательство. Доказательство проведем индукцией по числу вершин G. Пусть z – максимальный вес ребра G, стягивание которого дает G'. С одной стороны, граф G' меньше G, поэтому, по предположению индукции, имеем $GreedyHam(G') \geq ½\ OptHam(G'$. С другой стороны, имеем $OptHam(G') \geq OptHam(G') - 2z$ и $GreedyHam(G) = GreedyHam(G') + z$.

Таким образом, $GreedyHam(\mathrm{G}) \geq ½OptHam(G') + z \geq ½(OptHam(G) - 2z) + z \geq ½OptHam(G)$, что и требовалось доказать. ■

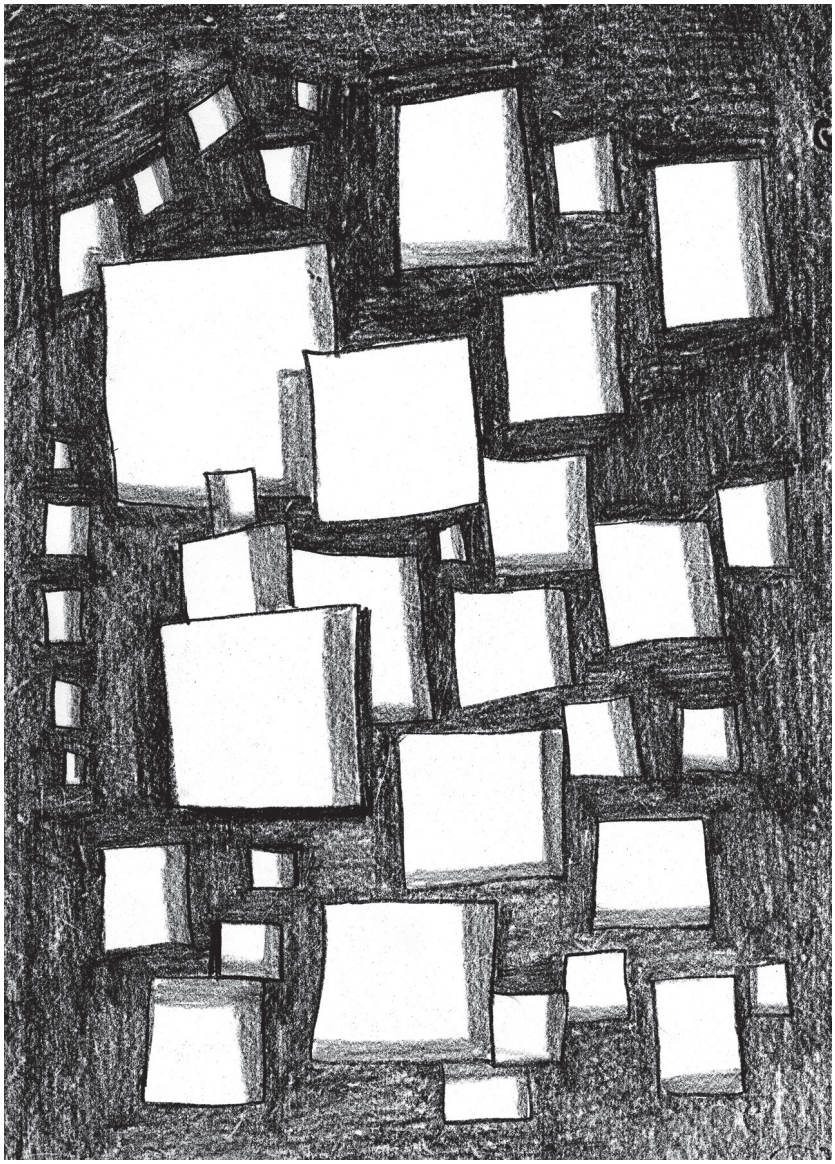
Из приведенного выше наблюдения и лемм 17, 18 и 19 сразу следует, что жадный алгоритм для суперстрок позволяет добиться коэффициента сжатия 1/2.

Примечания

Приведенное доказательство – вариант доказательства из работы Tarhio and Ukkonen [230].

Глава 7

Разное



108. Двоичные слова Паскаля

Треугольник Паскаля содержит биномиальные коэффициенты, вычисляемые по следующему правилу:

$$\binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i}.$$

Благодаря регулярной структуре треугольника доступ к коэффициентам можно осуществить быстро. В этой задаче n-е двоичное слово Паскаля P_n – это n-я строка треугольника Паскаля по модулю 2, т. е. для $0 \le i \le n$:

$$P_n[i] = \binom{n}{i} \bmod 2.$$

Ниже приведены слова P_n для $0 \le n \le 6$:

P_0 = 1
P_1 = 1 1
P_2 = 1 0 1
P_3 = 1 1 1 1
P_4 = 1 0 0 0 1
P_5 = 1 1 0 0 1 1
P_6 = 1 0 1 0 1 0 1

Вопрос. Пусть даны двоичные представления $r_k r_{k-1}...r_0$ числа n и $c_k c_{k-1}...c_0$ числа i. Показать, как за время $O(k)$ вычислить букву $P_n[i]$ и количество вхождений 1 в P_n.

[**Указание:** можно воспользоваться теоремой Лукаса.]

Теорема [Лукас, 1852]. Если p – простое число и r_k, $r_{k-1}...r_0$, $c_k c_{k-1}...c_0$ представления по модулю p целых чисел r и c таких, что $r \ge c \ge 0$, то

$$\binom{r}{c} \bmod p = \prod_{i=0}^{k} \binom{r_i}{c_i} \bmod p.$$

Решение
Следующее свойство позволяет построить алгоритм вычисления букв P_n за время $O(k)$.

Свойство 1. $P_n[i] = 1 \Leftrightarrow \forall j\ 1 \le j \le k\ (r_j = 0 \Rightarrow c_j = 0)$.

Доказательство. Эта эквиваленция – следствие теоремы Лукаса. В нашем случае $= 2$ и $r_j, c_j \in \{0,1\}$. Тогда

$$\binom{r_i}{c_i} \bmod 2 = 1 \Leftrightarrow (r_j = 0 \Rightarrow c_j = 0),$$

откуда сразу следует доказываемое свойство. ■

Пример. $P_6[4] = \binom{6}{4} \bmod 2 = 1$, т. к. двоичные представления 6 и 4 имеют вид `110` и `010` соответственно.

Чтобы ответить на вторую часть вопроса, обозначим $g(n)$ количество вхождений 1 в двоичное представление целого неотрицательного числа n. Следующий факт дает простой алгоритм вычисления количества вхождений 1 в P_n за требуемое время.

Свойство 2. Количество единиц в P_n равно $2^{g(n)}$.

Доказательство. Пусть $r_k r_{k-1} \ldots r_0$ и $c_k c_{k-1} \ldots c_0$ – двоичные представления n и i соответственно. Положим

$$R = \{j : r_j = 1\} \text{ и } C = \{j : c_j = 1\}.$$

В силу свойства 1 имеем

$$\binom{n}{i} \bmod 2 = 1 \Leftrightarrow C \subseteq R.$$

Поэтому искомое число равно количеству подмножеств C множества R, т. е. $2^{g(n)}$, в силу определения $g(n)$. ■

Примечания

Простое доказательство теоремы Лукаса см. в работе Fine [114].

Из многих интересных свойств слов Паскаля рассмотрим следующее. Для слова $w = w[0..k]$ и множества натуральных чисел X определим

$$Filter(w, X) = w[i_1]w[i_2]\ldots w[i_t],$$

где $i_1 < i_2 < \cdots < i_t$ и $\{i_1, i_2, \ldots, i_t\} = X \cap [0..k]$. Тогда для целого положительного n и множества Y степеней 2 имеет место равенство

$$Filter(P_n, Y) = \text{двоичное представление } n, \text{ записанное в обратном порядке.}$$

Простое доказательство следует из структуры i-й диагонали треугольника Паскаля по модулю 2, если считать диагонали слева направо, начиная с 0. Нулевая диагональ состоит из единиц, следующая за ней – из повторений 10 и т. д. Если теперь рассмотреть таблицу, строки которой представляют последовательные числа, то для ее столбцов наблюдаются аналогичные закономерности.

109. Самовоспроизводящиеся слова

Морфизмы часто используются для порождения конечных или бесконечных слов, поскольку они определяются на одиночных буквах. В этой задаче мы рассмотрим другой тип функции, которую можно считать своего рода контекстно-зависимым последовательным преобразователем.

Нашим рабочим алфавитом будет $A = \{0,1,2\}$. Образом h слова $w \in A^+$ является слово

$$h(w) = (0 \oplus a[0])\ (a[0] \oplus a[1])\ (a[1] \oplus a[2])\ldots(a[n-1] \oplus 0),$$

где $\oplus$ – сложение по модулю 3. Итеративное применение h к начальному слову из A^+ порождает все более длинные слова, обладающие весьма специфическими свойствами, которые продемонстрированы на двух примерах ниже.

Пример 1. Если процедура применяется к начальному слову $x = 1221$, то получается список троичных слов, первые элементы которого показаны ниже:

$h^0(x) = 1\ 2\ 2\ 1$
$h^1(x) = 1\ 0\ 1\ 0\ 1$
$h^2(x) = 1\ 1\ 1\ 1\ 1\ 1$
$h^3(x) = 1\ 2\ 2\ 2\ 2\ 2\ 1$
$h^4(x) = 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1$
$h^5(x) = 1\ 1\ 1\ 2\ 2\ 2\ 1\ 1\ 1$
$h^6(x) = 1\ 2\ 2\ 0\ 1\ 1\ 0\ 2\ 2\ 1$
$h^7(x) = 1\ 0\ 1\ 2\ 1\ 2\ 1\ 2\ 1\ 0\ 1$
$h^8(x) = 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1$
$h^9(x) = 1\ 2\ 2\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 2\ 2\ 1$

В частности, $h^9(x) = x \cdot 00000 \cdot x$, т. е. слово x воспроизвелось.

Пример 2. Если процедура применяется к слову $y = 121$, то список начинается следующими элементами:

$h^0(y) = 1\ 2\ 1$
$h^1(y) = 1\ 0\ 0\ 1$
$h^2(y) = 1\ 1\ 0\ 1\ 1$
$h^3(y) = 1\ 2\ 1\ 1\ 2\ 1$

Вопрос. Показать, что для любого слова $w \in A^+$ имеют место два свойства:
(A) существует целое m такое, что $h^m(w)$ состоит из двух копий w, разделенных фактором, состоящим из одних нулей (т. е. фактор принадлежит 0^*);
(B) если $|w|$ – степень 3, то $h^{|w|}(w) = ww$.

Решение
(A). Пусть m – минимальная степень 3, не меньшая длины n слова w, и обозначим

$$\alpha(i) = \binom{m}{i} \bmod 3.$$

Мы воспользуемся следующим фактом.

Наблюдение. Пусть $i \in \{0, 1, \ldots, m\}$. Так как m – степень 3, имеем $\alpha(i) = 1$, если $i \in \{0, m\}$ (это очевидно), иначе (это чуть менее очевидно) $\alpha(i) = 0$.

Отсылаем читателя к задаче 108, где показана связь между описываемым процессом и треугольником Паскаля.

Начав со слова 1, мы после m шагов будем иметь в позиции i слова $h^m(1)$ букву $\alpha(i)$.

В силу наблюдения, после m шагов вклад одной 1 в позиции t в букву в позиции $t + i$ составляет $\alpha(i)$. Поэтому в слове $h^m(w)$ префикс w остается без изменения, т. к. $\alpha(0) = 1$, и копируется на m позиций вправо, поскольку $\alpha(m) = 1$. Буквы в других позициях $h^m(w)$ равны 0, потому что $\alpha(i) = 0$ для $i \notin \{0, m\}$. Это дает ответ на пункт (A) вопроса.

(B). Следуя рассуждению выше, если $|w|$ – степень 3, то слово w копируется на m позиций вправо, что дает слово ww, поскольку префикс размера m не изменяется. Тем самым мы ответили на пункт (B) вопроса.

Примечания
Для алфавита $A_j = \{0, 1, \ldots, j - 1\}$, где j – простое число, мы можем взять $m = \min\{j^i : j^i \geq n\}$, где n – длина начального слова w. Если j не простое, то ситуация сложнее: теперь мы можем взять $m = j \cdot n!$, но в этом случае слово, разделяющее две копии w, может содержать не только нули.

Представленная задача взята из работы [13], где рассмотрена также двумерная (более интересная) версия.

110. Веса факторов

Весом слова над алфавитом {1,2} называется арифметическая сумма его букв. В этой задаче рассматриваются веса всех непустых факторов заданного слова длины n. При таком ограниченном алфавите максимально возможный вес равен $2n$, а максимальное число различных весов факторов равно $2n - 1$.

Например, число весов факторов слова 2221122 равно 10, вот они: 1,2, ..., 8,10,12.

Вопрос. Спроектировать алгоритм, который за линейное время вычисляет количество различных весов непустых факторов слова $x \in \{1,2\}^+$.

Вопрос. Показать, что после предобработки слова x, занимающей линейное время, на любой запрос типа «существует ли непустой фактор x с положительным весом k?» можно ответить за постоянное время. Объем памяти после предобработки должен быть постоянным.

Решение
Прежде чем переходить к решениям, остановимся на некоторых свойствах весов. Для целого положительного числа k положим

$$\text{SameParity}(k) = \{i : 1 \le i \le k \text{ и } (k - i) \text{ четно}\}.$$

Размер этого множества равен $|\text{SameParity}(k)| = \lceil k/2 \rceil$.

Обозначим $sum(i, j)$ вес фактора $x[i..j]$ слова x, $i \le j$. Совсем простые решения поставленных вопросов вытекают из следующего факта.

Факт. Если $k > 0$ – вес некоторого фактора x, то каждый элемент множества $\text{SameParity}(k)$ также является весом некоторого непустого фактора x.

Доказательство. Пусть $sum(i, j) = k$ – вес $x[i..j]$, $i \le j$. Если $x[i] = 2$ или $x[j] = 2$, то отбрасывание первой или последней буквы $x[i..j]$ дает фактор с весом $k - 2$, если он не пуст. В противном случае $x[i] = x[j] = 1$, и после отбрасывания обеих крайних букв мы снова получаем фактор с весом $k - 2$, если он не пуст. Повторяя эту процедуру, мы докажем утверждение. ■

Заметим, что если $x \in 2^+$, то ответы на оба вопроса тривиальны, потому что факторы x имеют $|x|$ различных весов: $2, 4, \ldots, 2|x|$. Далее мы будем предполагать, что x содержит хотя бы одно вхождение буквы 1. Обозначим *first* и *last* соответственно первую и последнюю позиции буквы 1 в x и положим

$$s = sum(0, n - 1) \text{ и } t = \max\{sum(first + 1, n - 1), sum(0, last - 1)\}.$$

Иначе говоря, s – вес всего слова x, а t – максимальный вес префикса или суффикса x, имеющего четность, отличную от s.

Следующее наблюдение является следствием доказанного ранее факта.

Наблюдение. Множество весов всех непустых факторов слова x является объединением $\text{SameParity}(s) \cup \text{SameParity}(t)$.

Число различных весов. Так как число различных весов непустых факторов x равно

$$|\text{SameParity}(s)| + |\text{SameParity}(t)| = \lceil s/2 \rceil + \lceil t/2 \rceil,$$

его вычисление сводится к вычислению s и t, для которого требуется линейное время.

Для слова 2221122 имеем $s = 12$, $t = \max\{5, 7\} = 7$, и число весов его факторов равно $\lceil 12/2 \rceil + \lceil 7/2 \rceil = 10$, что мы уже видели выше.

Постоянное время запроса. Предобработка заключается в вычислении обоих значений s и t, соответствующих слову x, что можно сделать за линейное время. После предобработки память используется только для хранения s и t .

Теперь, чтобы ответить на вопрос «является ли число $k > 0$ весом какого-нибудь непустого фактора x?», достаточно проверить условие

$k \leq t$ или (($s - k$) неотрицательное и четное),

что можно сделать за постоянное время.

Примечания

А как насчет алфавитов с большим числом букв, например {1,2,3,4,5}? Эффективный алгоритм существует, но он далеко не такой простой и красивый, как описано выше.

Пусть x – слово, длина которого равна степени 2. *Заякоренным* отрезком $[i..j]$ называется подотрезок $[0..|x| - 1]$, для которого i находится в левой, а j в правой половине. Соответствующий фактор $x[i..j]$ слова x называется заякоренным фактором. Используя быструю свертку, все различные веса заякоренных факторов слова x можно вычислить за время $O(|x| \log |x|)$. Можно взять характеристические векторы множеств весов суффиксов левой половины и префиксов правой половины x. Длина обоих векторов имеет порядок $O(|x|)$. Тогда свертка этих двух векторов (последовательностей) дает все веса заякоренных факторов.

Применив рекурсивный подход, все различные веса факторов слова длины n над алфавитом {1,2,3,4,5} можно вычислить за время $O(n(\log n)^2)$, потому что время работы алгоритма $T(n)$ удовлетворяет рекуррентному соотношению $T(n) = 2T(n/2) + O(n \log n)$.

111. Разности вхождений букв

Для непустого слова x обозначим $diff(x)$ разность между числом вхождений самой частой и самой редкой буквы в x (это может быть одна и та же буква).

Например:

$diff(\texttt{aaa}) = 0$, а $diff(\texttt{cabbcadbeaebaabec}) = 4$.

Во втором слове `a` и `b` – самые частые буквы, каждая входит по пять раз, а `d` – самая редкая буква с единственным вхождением.

Вопрос. Спроектировать алгоритм, который за время $O(n|A|)$ вычисляет значение $\max\{diff(x) : x$ является фактором $y\}$ для непустого слова y длины n над алфавитом A.

[**Указание:** сначала рассмотрите $A = \{\texttt{a},\texttt{b}\}$.]

Решение

Временно предположим, что $y \in \{\texttt{a},\texttt{b}\}^+$, и поищем фактор x слова y, в котором `b` – самая частая буква. Для этого преобразуем y в Y, подставив -1 вместо `a` и 1 вместо `b`. Теперь задача свелась к вычислению фактора с максимальной

арифметической суммой, содержащего по меньшей мере по одному вхождению 1 и −1.

Прежде чем переходить к алфавиту общего вида, рассмотрим решение для двоичного алфавита {-1,1} и введем некоторые обозначения.

Для данной позиции i в слове $Y \in \{-1,1\}^+$ обозначим sum_i сумму $Y[0] + Y[1] + \ldots + Y[i]$, а $pref_i$ минимальную сумму, соответствующую префиксу $Y[0..k]$ слова Y, для которого $k < i$ и $Y[k+1..i]$ содержит хотя бы одно вхождение -1. Если такого k не существует, положим $pref_i = \infty$.

Следующий алгоритм возвращает искомое значение для слова Y.

```
MaxDifference(непустое слово Y над алфавитом {-1,1})
 1  (maxdiff, prevsum, sum) ← (0,0,0)
 2  pref ←∞
 3  for i ← 0 to |Y| − 1 do
 4      sum ← sum + Y[i]
 5      if Y[i] = −1 then
 6          pref ← min{pref, prevsum}
 7          prevsum ← sum
 8      maxdiff ← max{maxdiff, sum − pref}
 9  return maxdiff
```

Алгоритм MaxDifference реализует следующее наблюдение, чтобы вычислить максимальную разность между факторами поданного на вход слова.

Наблюдение. Предположим, что $pref \neq \infty$. Тогда буква $Y[k]$ равна -1 и разность $diff(Y[k+1..i])$ равна $sum - pref$. Более того, для $Y[k+1..i]$ значение $diff$ максимально среди всех суффиксов $Y[0..i]$.

Таким образом, для слова над двухбуквенным алфавитом задача решается за линейное время.

Для алфавита большего размера применим следующий прием. Для любых двух различных букв a и b слова y обозначим $y_{a,b}$ слово, получающееся удалением всех остальных встречающихся в y букв. После замены $y_{a,b}$ словом $Y_{a,b}$ над алфавитом {-1,1} алгоритм MaxDifference находит максимальную разность среди факторов $Y_{a,b}$, и это значение будет также максимальной разностью среди факторов $y_{a,b}$.

Искомое значение равно максимуму среди всех результатов, полученных применением MaxDifference к $Y_{a,b}$ для всех пар букв a и b по отдельности. Поскольку сумма длин всех слов $y_{a,b}$ равна всего лишь $O(n|A|)$, общее время работы алгоритма составляет $O(n|A|)$ для слова длины n над алфавитом A.

Примечания

Эта задача была предложена на Польской олимпиаде по информатике для школьников в 2010 году.

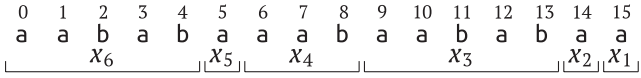
112. Факторизация с префиксами, свободными от границ

Поиск свободных от границ образцов в тексте можно выполнить очень эффективно, не прибегая к сложным решениям на основе алгоритма Бойера–Мура (см. задачу 33), поскольку никакие два вхождения образца не могут перекрываться. Если образец содержит границы, то его разложение на слова, свободные от границ, может привести к эффективным методам поиска.

Мы говорим, что непустое слово u свободно от границ, если ни один его собственный непустой префикс не является также суффиксом, т. е. $\mathrm{Border}(u) = \varepsilon$, или, эквивалентно, если его наименьший период равен его длине, т. е. $per(u) = |u|$.

В настоящей задаче наша цель – показать, как разложить слово на свободные от границ префиксы.

Рассмотрим, к примеру, слово aababaaabaababaa. Для него множество свободных от границ префиксов имеет вид {a,aab,aabab}, а факторизация, состоящая из этих префиксов, такова:



Вопрос. Показать, что непустое слово x единственным образом разлагается в произведение $x_k x_{k-1} \ldots x_1$, где каждый фактор x_i – префикс x, свободный от границ, и что x_1 – самый короткий свободный от границ префикс x.

Факторизацию x можно представить списком длин факторов. В нашем примере он имеет вид (5,1,3,5,1,1), а сами факторы – x[0..4], x[5..5], x[6..8], x[9..13], x[14..14], x[15..15].

Вопрос. Спроектировать алгоритм, который за линейное время вычисляет разложение слова в произведение свободных от границ префиксов, конкретно – список длин факторов.

Решение

Единственность факторизации. Пусть $S(x)$ – множество свободных от границ префиксов x. Это суффиксный код, т. е. если u, $v \in S(x)$ – различные слова, то ни одно из них не является суффиксом другого. Действительно, если бы, например, u было собственным суффиксом v, то, поскольку u – непустой префикс v, слово v не было бы свободно от границ, что противоречит предположению. Таким образом, любое произведение слов из $S(x)$ допускает единственное разложение в такое произведение. Отсюда следует единственность разложения x в произведение слов, принадлежащих $S(x)$, если такое разложение вообще существует.

Докажем, что факторизация существует. Если x свободно от границ, т. е. $x \in S(x)$, то факторизация содержит единственный фактор, само x. В противном случае пусть u – наименьшая непустая граница x. Тогда u свободно от границ, т. е. $u \in S(x)$. Повторив то же рассуждение для слова xu^{-1}, мы в итоге придем к факторизации. Эта факторизация обладает тем свойством, что последний фактор – кратчайший элемент $S(x)$, что нам и нужно.

Вычисление списка длин факторов. Факторизацию непустого слова x можно вычислить по его таблице границ, построив промежуточную *таблицу кратчайших границ shtbord*: элемент *shtbord*[l] равен 0, если слово $x[0..l-1]$ свободно от границ, а в противном случае – длине его самой короткой границы. Таблица вычисляется путем обхода таблицы границ x слева направо.

Ниже приведены таблицы для слова $x =$ `aababaaabaababaa`:

i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$		a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	a
l	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
border[l]	–	0	1	0	1	0	1	2	2	3	4	2	3	4	5	6	7
shtbord[l]	–	0	1	0	1	0	1	1	1	3	1	1	3	1	5	1	1

Длины l свободных от границ префиксов x удовлетворяют условию *border*[l] $= 0$ и в данном примере равны 1, 3 и 5.

Поскольку множество $S(x)$ является суффиксным кодом, естественно вычислять его факторизацию, просматривая x справа налево. Следуя приведенному выше доказательству, длины факторов выбираются из таблицы кратчайших непустых границ, пока не будет найден свободный от границ префикс.

```
FACTORISE(непустое слово x)
 1  border ← BORDERS(x)
 2  for l ← 0 to |x| do
 3      if border[l] > 0 и shtbord[border[l]] > 0 then
 4          shtbord[l] ← shtbord[border[l]]
 5      else shtbord[l] ←border[l]
 6  L ← пустой список
 7  l ← |x|
 8  while border[l] > 0 do
 9      L ← shtbord[l]·L
10      l ← l − shtbord[l]
11  L ← shtbord[l]·L
12  return L
```

Что до времени работы алгоритма FACTORISE, то без учета вычисления таблицы границ оно, очевидно, линейно, а таблицу границ тоже можно вычислить за линейное время (см. задачу 19). Поэтому весь процесс занимает линейное время.

Примечания
Не ясно, можно ли вычислить таблицу кратчайших непустых границ столь же эффективно, применив такую же технику, как для построения таблицы коротких границ в задаче 21.

113. Тест примитивности для унарных расширений

Непустое слово x можно разложить в произведение $x = (uv)^e u$, где u и v – два слова и v непустое, так что $|uv| = per(x)$ – наименьший период x, а e – положительное число. Обозначим $tail(x) = v$ (не u).

Например, $tail(\texttt{abcd}) = \texttt{abcd}$, потому что в соответствующем разложении $u = \varepsilon$, $v = \texttt{abcd}$, $tail(\texttt{abaab}) = \texttt{a}$, поскольку $\texttt{abaab} = (\texttt{aba})^1\texttt{ab}$, а $tail(\texttt{abaababa}) = \texttt{ab}$, так как $\texttt{abaababa} = (\texttt{abaab})^1\texttt{aba}$. Последнее слово является словом Фибоначчи fib_4, и в общем случае $tail(fib_n) = fib_{n-3}$ для $n \geq 3$.

В этой задаче наша цель – проверить, является ли слово xa_k примитивным, если о слове x известно очень мало.

Вопрос. Предположим, что об x имеется лишь следующая информация:

- x не является унарным (содержит хотя бы две буквы);
- $tail(x)$ не является унарным или $tail(x) \in b^*$ для некоторой буквы b;
- $l = |tail(x)|$.

Показать, как за постоянное время ответить на вопрос, является ли слово xa_k примитивным, если заданы целое число k и буква a.

[**Указание:** $tail(x)$ – очевидный кандидат на расширение x до непримитивного слова.]

Решение
Решение опирается на следующее свойство неунарного слова x:

xa^k не является примитивным $\Rightarrow tail(x) = a^k$.

Поскольку обратное, очевидно, верно, это свойство приводит к тесту, требующему постоянного времени, если справедливы сформулированные в вопросе предположения, потому что проверка примитивности xa^k сводится к проверке того, что $tail(x) \in b^*$, т. е. что $a = b$ и $k = l$. Несмотря на простоту формулировки, доказательство этого свойства довольно утомительно. Начнем со вспомогательного факта, а затем перейдем к основной лемме.

Факт. Если x неунарное и $x = (uv)^e u = u'v'u'$, где $|uv| = per(x)$, $e > 0$ и $|u'| < |u|$, то v' неунарное.

Действительно, если v' унарное, то v тоже унарное, откуда следует, что u неунарное (оно не может быть унарным с буквой, отличной от буквы, обра-

зующей v). Так как слово u' является одновременно собственным префиксом и суффиксом u, имеем $u = u'y = zu'$, где y и z – непустые слова, являющиеся соответственно префиксом и суффиксом v', и $|y| = |z| = per(u)$. Тогда оба они унарны, откуда следует, что таковы же u и x, и мы пришли к противоречию.

Лемма об унарном расширении. Если x – неунарное слово, a – буква, k – целое положительное число и $a^k \neq tail(x)$, то xa^k примитивно.

Доказательство. Предположим противное – что xa^k не примитивно, т. е.

$$xa^k = z^j, j \geq 2, |z| = per(xa^k).$$

Отсюда следует, что $|z| > k$, потому что в противном случае x было бы унарным и $|z| \neq per(x)$, т. к. $a^k \neq tail(x)$. Поскольку $|z|$ – период x, $|z| > per(x)$. Не может быть, что $j > 2$, потому что z^2 было бы префиксом x, откуда следовало бы, что $|z| = per(x)$.

Следовательно, остается только одна возможность: $j = 2$, т. е.

$$xa^k = z^2 = u'v'u'v',\ x = (uv)^i u = u'v'u',\ v' = a^k,$$

где $v' \neq v = tail(x)$, $|uv| = per(x)$ и $|u'v'| = per(xa^k)$. Рассмотрим два случая.

Случай $|u'| < |u|$. Это невозможно, в силу установленного ранее факта, из которого следует, что слово $v' = a^k$ неунарное.

Случай $|u'| > |u|$. Рассмотрим только случай $i = 2$, т. е. $x = (uv)^2u$. Общий случай $i > 1$ разбирается аналогично.

Утверждение. $|u'| < |uv|$.

Доказательство (утверждения). Предположим противное – что $|u'| \geq |uv|$. Тогда слово x допускает периоды $p = |uv|$ и $q = |x| - |u'| = |u'v'|$, где $p + q \leq |x|$. По лемме о периодичности, p (будучи наименьшим периодом) делит q. Но тогда p также является периодом слова xa^k (которое имеет период q). Следовательно, x и xa^k имеют одинаковый наименьший период, что невозможно. ■

Пусто w – такое слово, что $u'v' = uvw$. В силу утверждения выше, w – суффикс v' и, следовательно, w унарное. Слово u' является префиксом uvu (будучи префиксом x). Отсюда следует, что $|w|$ – период uvu. Так как w унарное, uv тоже унарное, и все слово x унарное – противоречие.

В обоих случаях мы пришли к противоречию. Поэтому слово xa^k примитивное, как и утверждалось. ■

Примечания

Решение этой задачи взято из работы Rytter [215]. Оптимальный по времени и памяти тест примитивности (с линейным временем работы и постоянной памятью) описан в задаче 39, но приведенное здесь решение гораздо быстрее в рассмотренном частном случае.

114. Частично коммутативные алфавиты

Интерес к изучению слов над частично коммутативным алфавитом связан с представлением конкурентных процессов, в котором буквы – это имена процессов, а коммутативность соответствует неконкурентности двух процессов.

Мы будем рассматривать алфавит A, в котором некоторые пары букв коммутируют. Это означает, что слово $uabv$ можно преобразовать в $ubav$ для любой коммутирующей пары (a, b). Соответствующее частичное отношение коммутативности на A обозначается $\approx$ и предполагается симметричным.

Два слова называются эквивалентными относительно отношения коммутативности (обозначается $u \equiv v$), если одно можно преобразовать в другое посредством серии перестановок соседних коммутирующих букв. Заметим, что $\equiv$ является отношением эквивалентности, тогда как $\approx$ обычно таковым не является.

Например, над алфавитом $A = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$

$$\mathtt{a} \approx \mathtt{b} \approx \mathtt{c} \approx \mathtt{d} \approx \mathtt{a} \Rightarrow \mathtt{abcdabcd} \equiv \mathtt{badbdcac}$$

в силу следующих перестановок:

a b c d a b c d
b a c d a b c d
b a d c a b c d
b a d c b a c d
b a d b c a c d
b a d b c a d c
b a d b c d a c
b a d b d c a c

Вопрос. Спроектировать тест эквивалентности, который за время $O(n|A|)$ проверяет, верно ли, что $u \equiv v$, где u и v – слова длины n, принадлежащие A^*.

[**Указание:** рассмотрите проекции слов на пары букв.]

Решение

Для двух букв $a, b \in A$ обозначим $\pi_{a,b}(w)$ проекцию слова w на пару (a, b), т. е. слово, получающееся удалением из w всех букв, кроме этих двух. Обозначим $|w|_a$ количество вхождений буквы a в слово w. Следующее свойство является основой нашего решения.

Свойство. Для двух слов u и v эквивалентность $u \equiv v$ имеет место тогда и только тогда, когда выполнены следующие два условия:

(i) $|u|_a = |v|_a$ для любой буквы $a \in A$;
(ii) $\pi_{a,b}(u) = \pi_{a,b}(v)$ всегда, когда a и b не коммутируют.

Доказательство. Ясно, что условия выполняются, если $u \equiv v$. Обратно, предположим, что условия (i) и (ii) выполнены. Доказательство проведем индукцией по общей длине двух слов.

Предположим, что $u = au'$, где $a \in A$. Мы утверждаем, что можно переместить первое вхождение буквы a в v в первую позицию, применяя отношение $\approx$. Действительно, если это невозможно, то существует какая-то не коммутирующая с a буква b, встречающаяся в v раньше a. Тогда $\pi_{a,b}(u) \neq \pi_{a,b}(v)$, что противоречит условию (ii).

После перемещения a в начало v мы получаем слово av' такое, что $av' \equiv v$, и условия (i) и (ii) выполнены для u' и v'. По предположению индукции $u' \equiv v'$ и, следовательно, $u = au' \equiv av' \equiv v$. Что и требовалось доказать. ■

Тест эквивалентности состоит в проверке двух сформулированных выше условий. Для проверки первого, очевидно, нужно время $O(n|A|)$ (или даже $O(n \log |A|)$ без каких-либо предположений об алфавите).

Что касается второго условия, то нужно проверить, что $\pi_{a,b}(u) = \pi_{a,b}(v)$ для всех пар некоммутирующих букв a, b. На первый взгляд, для этого нужно время $O(n|A|^2)$. Однако сумма длин всех слов вида $\pi_{a,b}(u)$ равна лишь $O(n|A|)$, это и есть верхняя граница времени работы алгоритма.

Примечания

Материал этой задачи основан на свойствах частичного коммутирования, изученных в работе Cori and Perrin [61].

Существует другой алгоритм решения задачи об эквивалентности. Мы можем определить каноническую форму слова – его лексикографически наименьшую эквивалентную версию. Тогда, имея два слова, мы можем вычислить их канонические формы и проверить, эквивалентны ли они. Вычисление канонических форм представляет самостоятельный интерес.

115. Наибольшее ожерелье фиксированной плотности

Слово называется ***ожерельем***, если оно является лексикографически наименьшим в своем классе сопряженности. ***Плотностью*** слова над алфавитом {0,1} называется число вхождений 1. Обозначим $N(n, d)$ множество всех двоичных ожерелий длины n и плотности d.

Задача заключается в том, чтобы найти лексикографически наибольшее ожерелье в множестве $N(n, d)$, где $0 \leq d \leq n$. Например, 00100101 – наибольшее ожерелье в $N(8,3)$:

{00000111,00001011,00001101,00010011,00010101,00011001,00100101}.

А 01011011 – наибольшее ожерелье в $N(8,5)$:

{00011111,00101111,00110111,00111011,00111101,01010111,01011011}.

Следующее интуитивно понятное свойство характеризует структуру наибольших ожерелий.

***Лемма* 20.** *Пусть C – наибольшее ожерелье в $N(n, d)$:*

(i) *если $d \le n/2$, то $C = 0^{c_0}10^{c_1}1\ldots0^{c_{d-1}}1$, где $c_0 > 0$ и для любого $i > 0$, $c_i \in \{c_0, c_0 - 1\}$;*

(ii) *если $d > n/2$, то $C = 01^{c_0}01^{c_1}\ldots01^{c_{n-d-1}}$, где $c_0 > 0$ и для любого $i > 0$, $c_i \in \{c_0, c_0 + 1\}$;*

(iii) *в обоих случаях двоичная последовательность $w = (0, |c_1 - c_0|, |c_2 - c_0|, \ldots)$ является наибольшим ожерельем среди всех ожерелий такой же длины и плотности.*

Вопрос. На основе леммы 20 спроектировать алгоритм, который за линейное время вычисляет наибольшее ожерелье в $N(n, d)$.

Решение

Принимая во внимание лемму, определим следующие две величины для двоичного слова w, являющегося ожерельем длины l:

$$\varphi_t(w) = 0^{t-w[0]}10^{t-w[1]}1 \ldots 0^{t-w[l-1]}1,$$
$$\psi_t(w) = 01^{t+w[0]}\,01^{t+w[1]} \ldots 01^{t+w[l-1]}.$$

Следующие два факта – почти прямые следствия леммы, показывающие, что функции φ_t и ψ_t сохраняют лексикографический порядок. Они также служат обоснованием рекурсивной структуры приведенного ниже алгоритма.

Факт 1. Двоичное слово w длины l является ожерельем тогда и только тогда, когда $\varphi_t(w)$ является ожерельем для всех $t > 0$.

Факт 2. Двоичное слово w длины l является ожерельем тогда и только тогда, когда $\psi_t(w)$ является ожерельем для всех $t \ge 0$.

С учетом этих фактов следующий алгоритм решает задачу. Пункт (iii) леммы позволяет свести задачу к гораздо меньшей с помощью одного рекурсивного вызова.

```
GreatestNecklace(натуральные числа n, d такие, что d ≤ n)
 1  if d = 0 then
 2      return 0^n
 3  elseif d ≤ n/2 then
 4      (t, r) ← (⌊n/d⌋, n mod d)
 5      w ← GreatestNecklace(d, d − r)
 6      return φ_t(w)
 7  elseif d < n then
 8      (t, r) ← (⌊n/(n − d)⌋, n mod (n − d))
 9      w ← GreatestNecklace(n − d, r)
10      return ψ_{t−1}(w)
11  else return 1^n
```

Пример. Для $n = 8$ и $d = 3$ имеем $t = 2$, $r = 2$, и рекурсивный вызов дает GreatestNecklace(3, 1) $= w =$ 001. В конечном итоге GreatestNecklace(8, 3) $=$ $0^{2-0}10^{2-0}10^{2-1}1$, т. е. 00100101, как мы уже видели раньше.

Пример. Для $n = 8$ и $d = 5$ также имеем $t = 2$, $r = 2$ (т. к. $n - d = 3$), и рекурсивный вызов дает GreatestNecklace(3, 2) $= w =$ 011. В конечном итоге GreatestNecklace(8, 5) $= 01^{1+0}01^{1+1}01^{1+1}$, т. е. 01011011, как мы уже видели раньше.

Правильность алгоритма GreatestNecklace вытекает из леммы и обоих фактов.

Что касается времени работы GreatestNecklace(n, d), заметим, что рекурсивные вызовы получают слова, длина которых не превышает $n/2$. Следовательно, полное время генерирования конечного слова линейно.

Примечания

Материал для этой задачи взят из работы Sawada and Hartman [218].

116. Двоичные слова, эквивалентные по периодам

Говорят, что два слова эквивалентны по периодам, если они имеют одинаковый набор периодов, или, эквивалентно, одинаковую длину и одно и то же множество длин границ. Например, слова abcdabcda и abaaabaaa эквивалентны по периодам, т. к. у них общее множество периодов {4,8,9}, хотя между буквами нет взаимно однозначного соответствия.

В этой задаче наша цель – показать, что множество периодов слова можно реализовать двоичным словом.

Вопрос. Пусть w – слово над сколь угодно большим алфавитом. Показать, как за линейное время построить двоичное слово x, эквивалентное w по периодам.

[**Указание:** рассмотрите длины границ вместо периодов.]

Решение

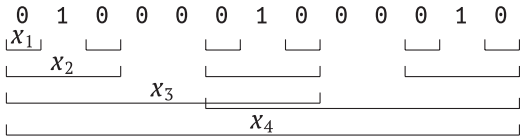
Работать с длинами границ w, а не с его периодической структурой удобнее при решении поставленного вопроса и описании соответствующего алгоритма. Структура границ описывается возрастающим списком $\mathcal{B}(w) = (q_1, q_2, \ldots, q_n)$ длин непустых границ w, в который добавлен элемент $q_n = |w| = N$. Например, со словом abcdabcda ассоциирован список (1,5,9).

Для ответа на вопрос мы итеративно построим по списку $\mathcal{B}(w)$ последовательность слов $(x_1, x_2, \ldots, x_n)$, в которой x_i – двоичное слово, ассоциированное со списком границ $(q_1, \ldots, q_i)$. Двоичным словом, эквивалентным по периодам w, является $x = x_n$.

Пусть x_1 – свободное от границ слово длины q_1. Слово x_i длины q_i с наибольшей границей x_{i-1} либо имеет вид $x_{i-1}y_ix_{i-1}$, если это укладывается в его

длину, либо строится путем перекрытия x_{i-1} с самим собой. Слово y_i унарное, а образующая его буква выбирается так, чтобы не создавать нежелательных границ.

Пример. Пусть $(1,3,8,13) = (q_1, q_2, q_3, q_4) = \mathcal{B}(\texttt{abacdabacdaba})$. Начав со свободного от границ слова $x_1 = \texttt{0}$, мы строим x_2, вставляя $y_2 = \texttt{1}$ между двумя вхождениями x_1. Слово x_3 строится аналогично из x_2 с унарным словом $y_3 = \texttt{00}$, для которого образующая буква отличается от буквы, образующей y_2. Наконец, $x_4 = \texttt{0100001000010}$ строится путем перекрытия двух экземпляров x_3.



В алгоритме Alternating, реализующем этот метод, *prefix*(z, k) означает префикс длины k слова z, когда $k \le |z|$.

```
Alternating(непустое слово w)
1  (q_1, q_2, ..., q_n) ← B(w)
2  (x_1, a) ← (01^(q_1 − 1), 0)
3  for i ← 2 to n do
4      gap_i ← q_i − 2q_(i−1)
5      if gap_i > 0 then
6          a ← 1 − a
7          x_i ← x_(i−1) · a^(gap_i) · x_(i−1)
8      else x_i ← prefix(x_(i−1), q_i − q_(i−1)) · x_(i−1)
9  return x_n
```

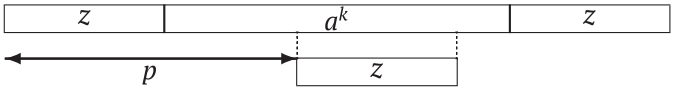
Почему алгоритм Alternating работает? Доказательство опирается на следующий результат, который прямо следует из леммы об унарном расширении (см. задачу 113). Если слово z записать в виде $(uv)^e u$ и $|uv|$ – его наименьший период, то, по определению, $tail(z) = v$.

***Лемма* 21.** *Пусть z – не унарное двоичное слово, k – целое положительное число, a – буква такая, что $a^k \ne tail(z)$. Тогда слово $x = za^kz$ не имеет периода, меньшего, чем $|za^k|$, т. е. не имеет границы, более длинной, чем у z.*

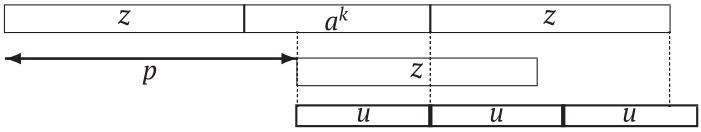
Доказательство. Доказательство проведем от противного. Предположим, что $x = za^kz$ имеет период $p < |za^k|$, и рассмотрим два случая.

Случай $p \le |z|$**.** Слово x имеет два периода, $|za^k|$ и p, удовлетворяющих условию $|za^k| + p \le |x|$. Применив к ним лемму о периодичности, получаем, что $gcd(|za^k|, p)$ – тоже период x. Отсюда следует, что ua^k непримитивное, т. к. $gcd(|za^k|, p) \le p < |ua^k|$. Но это противоречит лемме об унарном расширении, в силу которой при данном предположении ua^k должно быть примитивным.

Случай $|u| < p < |ua^k|$.



Из этих неравенств следует, что существует внутреннее вхождение z в za^kz, а именно в позиции p. Если $p \le k$, т. е. $p + |z| \le |za^k|$ (см. рисунок), то это вхождение внутреннее по отношению к a^k, что противоречит неунарности z.



В противном случае $p > k$, т. е. $p + |z| > |za^k|$ (см. рисунок). Тогда два последних вхождения z перекрываются, а значит, $|za^k| - p$ является периодом z. Будучи суффиксом a^k, префиксный период $u = z[0..|za^k| - p - 1]$ слова z унарный, откуда следует, что и само z унарное, и мы снова пришли к противоречию.

Таким образом, никакой период za^kz не может быть меньше $|za^k|$. ■

Для доказательства правильности алгоритма Alternating мы должны показать, что заполнение промежутка *gap* между вхождениями x_{i-1} унарным словом в строке 7 не порождает избыточного периода.

Для этого предположим, что $q_1 > 1$. Случай $q_1 = 1$ можно рассмотреть аналогично, начав с первого i, для которого x_i неунарное. Алгоритм Alternating обладает следующим свойством: если $gap_i > 0$, то $x_i = x_{i-1}y_ix_{i-1}$, где $y_i = a^{gap_i} \ne tail(x_{i-1})$ и x_{i-1} неунарное. По лемме, $x_{i-1}y_ix_{i-1}$ не имеет границы, более длинной, чем у x_{i-1}. Таким образом, никакой избыточной границы не создается, что и доказывает правильность алгоритма Alternating.

Вычисление списка длин границ, например, с помощью алгоритма Borders из задачи 19 и выполнение самого алгоритма Alternating занимает линейное время $O(N) = O(|w|)$, что нам и нужно.

Примечания

Приведенный алгоритм, а также более сложный алгоритм для двоичных лексикографически первых слов описаны в работе Rytter [215].

Заметим, что отсортированный список $\mathcal{B} = (q_1, q_2, \ldots, q_n)$ соответствует списку длин границ слова тогда и только тогда, когда $\delta_i = q_i - q_{i-1}$ при $i > 1$,

$$\delta_{i-1} \mid \delta_i \Rightarrow \delta_{i-1} = \delta_i \text{ и } q_i + \delta_i \le n \Rightarrow q_i + \delta_i \in \mathcal{B}.$$

Это вариант теоремы 8.1.11 из книги [176]. Описанная выше техника дает сжатое представление выходного слова длины N, имеющее размер $O(n)$, который может быть порядка $\log N$.

117. Динамическое генерирование слов де Брёйна

Двоичным словом де Брёйна порядка n называется слово длины 2^n над алфавитом {0,1}, в котором все двоичные слова длины встречаются циклически ровно один раз. Например, 00010111 и 01000111 – (несопряженные) слова де Брёйна порядка 3.

Слово де Брёйна можно сгенерировать, начав с двоичного слова длины n и повторяя описанную ниже операцию Next(w). Эта операция вычисляет следующий бит искомого слова де Брёйна и обновляет слово w. Процесс останавливается, когда w становится равным начальному слову.

```
deBruijn(целое положительное число n)
1  (x, w_0) ← (ε, двоичное слово длины n)
2  w ← w_0
3  do w ← Next(w)
4     x ← x·w[n − 1]
5  while w ≠ w_0
6  return x
```

Операция Next нужна только для получения подходящего алгоритма динамического генерирования. Обозначим $\bar{b}$ инверсию бита b.

```
Next(непустое слово w длины n)
1  if w[1..n − 1]·1 наименьшее в своем классе сопряженности then
2     b ← w[0]‾
3  else b ← w[0]
4  return w[1..n − 1]·b
```

Вопрос. Показать, что в результате выполнения алгоритма deBruijn(n) генерируется двоичное слово де Брёйна длины 2^n.

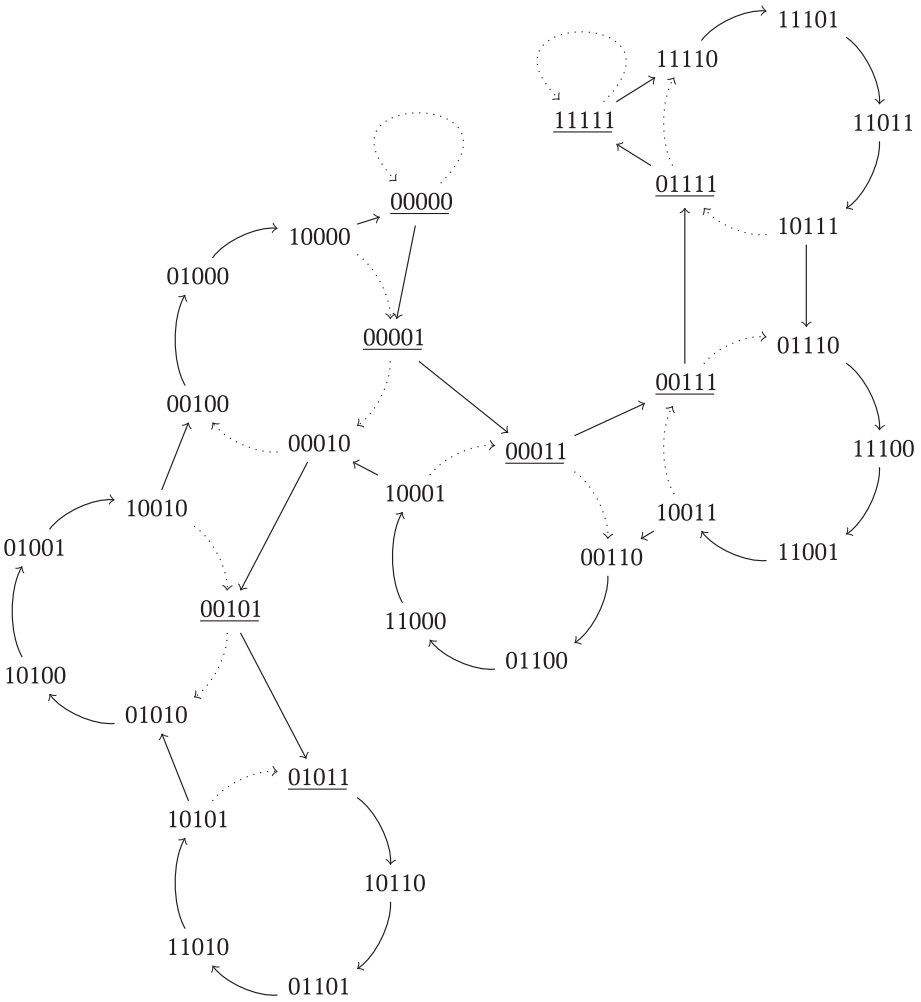
Пример. Пусть $n = 3$ и $w_0 = 111$. Слово w в строке 4 алгоритма deBruijn последовательно принимает следующие значения: 11$\underline{0}$, 10$\underline{1}$, 01$\underline{0}$, 10$\underline{0}$, 00$\underline{0}$, 00$\underline{1}$, 01$\underline{1}$, 11$\underline{1}$. Подчеркнутые биты образуют слово де Брёйна 01000111.

Для $n = 5$ и $w_0 = 11111$ последовательно порождаются значения 1111$\underline{0}$, 1110$\underline{1}$, 1101$\underline{1}$, 1011$\underline{1}$, 0111$\underline{0}$, 1110$\underline{0}$, 1100$\underline{1}$, 1001$\underline{1}$, 0011$\underline{0}$, 0110$\underline{0}$, 1100$\underline{0}$, 1000$\underline{1}$, 0001$\underline{0}$, 0010$\underline{1}$, 0101$\underline{1}$, 1011$\underline{0}$, 0110$\underline{1}$, 1101$\underline{0}$, 1010$\underline{1}$, 0101$\underline{0}$, 1010$\underline{0}$, 0100$\underline{1}$, 1001$\underline{0}$, 0010$\underline{0}$, 0100$\underline{0}$, 1000$\underline{0}$ 0000$\underline{0}$, 0000$\underline{1}$, 0001$\underline{1}$, 0011$\underline{1}$, 0111$\underline{1}$, 1111$\underline{1}$, а подчеркнутые биты образуют слово де Брёйна

01110011000101101010010000011111.

Решение

Правильность алгоритма deBruijn можно доказать, интерпретируя его выполнение как обход дерева, узлами которого являются циклы сдвига, соединенные двусторонними «мостами». Вершинами циклов сдвига являются слова длины n, принадлежащие одному классу сопряженности. Представителем цикла служит лексикографически минимальное слово в нем (ожерелье или слово Линдона, если оно примитивно). Ребра циклов соответствуют сдвигам вида $au \to ua$, где a – один бит, а u – слово длины $n-1$. Циклы сдвига образуют граф G_n.



Граф G_n преобразуется в граф G'_n (см. рисунок, где показан граф G'_5) путем добавления мостов, соединяющих дизъюнктные циклы, и удаления некоторых ребер из циклов (на рисунке они изображены пунктиром) с помощью следующей процедуры.

```
BRIDGES(граф циклов G)
1  for каждой вершины u графа G do
2      if слово u1 наименьшее в своем классе сопряженности then
3          удалить ребра 1u → u1 и 0u → u0
4          создать ребра 1u → u0 и 0u → u1
5  return модифицированный граф
```

Все сплошные ребра в G'_n ассоциированы с функцией NEXT и используются при обходе графа. Граф G'_n состоит из одного гамильтонова цикла, содержащего все слова длины n. Мосты, соединяющие циклический класс слова с другим циклическим классом слов, содержащих больше вхождений `0`, образуют дерево (без корня), узлами которого являются циклы.

Наблюдение. Для слова w длины n $v = \text{NEXT}(w)$ тогда и только тогда, когда $w \to v$ является ребром G'_n.

Таким образом, алгоритм неявно обходит граф по гамильтонову циклу. Тем самым его правильность доказана.

Примечания

Этот алгоритм взят из работы Sawada et al. [219], но приведенное доказательство совершенно другое. Цикл for в функции Bridges можно заменить таким:

```
1  for каждой вершины u of графа G do
2      if слово 0u наименьшее в своем классе сопряженности then
3        удалить ребра 0u → u0 и 1u → u1
4        создать ребра 0u → u1 и 1u → u0
```

В таком случае мы получим другой граф циклов сдвига, соединенных мостами иного типа, и другой вариант операции NEXT, которому соответствует иной гамильтонов цикл.

118. Рекурсивное генерирование слов де Брёйна

Продолжая тему задачи 117, в этой задаче предлагается другой метод генерирования слова де Брёйна над алфавитом B = {0,1}. Этот метод рекурсивный, и для его описания потребуется несколько предварительных определений.

Сначала определим функцию *Shift*. Если слово u циклически встречается в слове x ровно один раз и $|u| < |x|$, то *Shift*(x, u) обозначает слово, сопряженное x, в котором u является суффиксом. Например, *Shift*(`001011101`, `0111`) = `010010111` и *Shift*(`001011101`, `1010`) = `010111010`.

Пусть имеется два слова $x, y \in \mathrm{B}^N$, $N = 2^n$, таких, что суффикс u длины n слова x единственным образом входит циклическим фактором в слово y, тогда определим $x \oplus y$ как $x \cdot \mathit{Shift}(y, u)$. Например, для $n = 2$, $\mathtt{0010} \oplus \mathtt{1101} = \mathtt{0010\,1110}$, потому что $\mathit{Shift}(\mathtt{1101}, \mathtt{10}) = \mathtt{1110}$.

Наконец, для двоичного слова w обозначим $\Psi(w)$ двоичное слово v длины $|w|$, определенное следующим образом: для $0 \le i \le |w| - 1$

$$v[i] = (w[0] + w[1] + \ldots + w[i]) \bmod 2.$$

Например, $\Psi(\mathtt{0010111011}) = \mathtt{0011010010}$. И будем обозначать $\bar{x}$ дополнение к x, т. е. побитовое отрицание.

Вопрос. Показать, что если x – двоичное слово де Брёйна длины 2^n, то $\Psi(x) \oplus \overline{\Psi(x)}$ – слово де Брёйна длины 2^{n+1}.

[**Указание:** подсчитайте циклические факторы $\Psi(x) \oplus \overline{\Psi(x)}$.]

Пример. Для слова де Брёйна $\mathtt{0011}$ имеем $\Psi(\mathtt{0011}) = \mathtt{0010}$ и $\overline{\Psi(\mathtt{0011})} = \mathtt{1101}$. Тогда $\mathtt{0010} \oplus \mathtt{1101} = \mathtt{0010\,1110}$, т. е. слово де Брёйна длины 8.

Решение

Обозначим $\mathit{Cfact}_k(z)$ множество циклических факторов длины k слова z. Начнем со следующего факта.

Наблюдение 1. Если два слова x и y длины N имеют общий суффикс длины $n < N$, то $\mathit{Cfact}_{n+1}(x \cdot y) = \mathit{Cfact}_{n+1}(x) \cup \mathit{Cfact}_{n+1}(y)$.

Второе наблюдение является следствием первого.

Наблюдение 2. Пусть x и y – двоичные слова длины $N = 2^n$. Если $\mathit{Cfact}_{n+1}(x) \cup \mathit{Cfact}_{n+1}(y) = \mathrm{B}^{n+1}$ и суффикс x длины n принадлежит $\mathit{Cfact}_n(y)$, то $x \oplus y$ является словом де Брёйна.

Доказательство. Можно предполагать, что x и y имеют общий суффикс длины n, потому что это не изменяет результат операции $\oplus$, и тогда имеем $x \oplus y = x \cdot y$. Тогда из наблюдения 1 следует, что $\mathit{Cfact}_{n+1}(x \oplus y) = \mathit{Cfact}_{n+1}(x \cdot y) = \mathit{Cfact}_{n+1}(x) \cup \mathit{Cfact}_{n+1}(y) = \mathrm{B}^{n+1}$. Так как длина $x \oplus y$ равна 2^{n+1}, каждое двоичное слово длины $n + 1$ циклически входит в него ровно один раз. Это и означает, что $x \oplus y$ является словом де Брёйна. ■

Для ответа на вопрос обозначим x слово де Брёйна длины 2^n. Операция Ψ обладает следующими двумя свойствами:

(i) $\mathit{Cfact}_{n+1}(\Psi(x)) \cap \mathit{Cfact}_{n+1}(\overline{\Psi(x)}) = \emptyset$;

(ii) $|\mathit{Cfact}_{n+1}(\Psi(x))| = 2^n$ и $|\mathit{Cfact}_{n+1}(\overline{\Psi(x)})| = 2^n$.

Из свойств (i) и (ii) следует, что слова $\Psi(x)$ и $\overline{\Psi(x)}$ удовлетворяют предположениям наблюдения 2. Следовательно, $\Psi(x) \oplus \overline{\Psi(x)}$ – двоичное слово де Брёйна длины 2^{n+1}.

Примечания

Рекурсивный подход к построению слов де Брёйна взят из работы [206]. Он является синтаксической версией алгоритма Лемпеля, в котором используется специальный тип гомоморфизма графов, см. [174].

Это пример простого приложения алгебраических методов к алгоритмам обработки текста. Более сложное применение алгебраических методов – генерирование слов де Брёйна на основе так называемых *линейных сдвиговых регистров* и *примитивных полиномов*, см. [131].

У этого алгоритма имеется удивительное теоретико-графовое свойство. Предположим, что мы начали со слова $w_2 = \mathtt{0011}$, и определим для $n \geq 3$

$$w_n = \Psi(w_{n-1}) \oplus \overline{\Psi(w_{n-1})}.$$

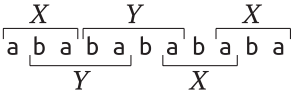
Тогда в графе де Брёйна G_{n+1} порядка $n+1$, имеющем 2^n вершин, w_n соответствует гамильтонову циклу C. После удаления C и отбрасывания двух одновершинных петель граф G_{n+1} становится большим простым циклом длины $2^n - 2$.

119. Уравнения в словах с заданными длинами переменных

Уравнением в словах называется уравнение между словами, буквы которых представляют собой постоянные или переменные. Постоянные принадлежат алфавиту $\mathtt{A} = \{\mathtt{a},\mathtt{b},\ldots\}$, а переменные – дизъюнктному алфавиту неизвестных $\mathtt{U} = \{\mathtt{X},\mathtt{Y},\ldots\}$. Уравнение записывается в виде $L = R$, где $L, R \in (\mathtt{A} \cup \mathtt{U})^*$, а его решением является морфизм $\psi : (\mathtt{A} \cup \mathtt{U})^* \to \mathtt{A}^*$, оставляющий постоянные неизменными, для которого имеет место равенство $\psi(L) = \psi(R)$ holds.

В этой задаче мы предполагаем, что для каждой переменной X, встречающейся в уравнении, задана длина $|\psi(X)|$, которая обозначается просто $|X|$.

Например, уравнение $XY\mathtt{b}X = \mathtt{a}Y\mathtt{b}X\mathtt{ba}$ с $|X| = 3$ и $|Y| = 4$ допускает (единственное) решение ψ, определенное как $\psi(X) = \mathtt{aba}$ и $\psi(Y) = \mathtt{baba}$:



С другой стороны, уравнение $\mathtt{a}XY = Y\mathtt{b}X$ не имеет решений, потому что слова $\mathtt{a}\psi(X)$ и $\mathtt{b}\psi(X)$ должны быть сопряженными, что несовместимо с условиями $|\mathtt{a}\psi(X)|_\mathtt{a} = 1 + |\psi(X)|_\mathtt{a}$ и $|\mathtt{b}\psi(X)|_\mathtt{a} = |\psi(X)|_\mathtt{a}$. Уравнение же $\mathtt{a}XY = Y\mathtt{a}X$ имеет $\mathtt{A}^{|X|}$ решений, когда $|X| = |Y| - 1$.

Вопрос. Пусть дано уравнение в словах с известными длинами переменных. Показать, как проверить существование решения за время, линейно зависящее от суммы длины уравнения и длины результата.

Решение

Пусть ψ – потенциальное решение уравнения $L = R$. Если $|\psi(L)| \neq |\psi(R)|$ для заданных длин переменных, то, очевидно, решений не существует. Поэтому будем считать, что длины переменных совместимы, и положим $n = |\psi(L)| = |\psi(R)|$.

Пусть $G = (V, E)$ – неориентированный граф, определенный следующим образом:

- $V = \{0, 1, \ldots, n - 1\}$, множество позиций слова $x = \psi(L) = \psi(R)$;
- E – множество ребер (i, j), где i и j соответствуют одной и той же относительной позиции в двух вхождениях $\psi(X)$ в $\psi(L)$ или в $\psi(R)$ для некоторой переменной X. Например, i и j могут быть первыми позициями вхождений.

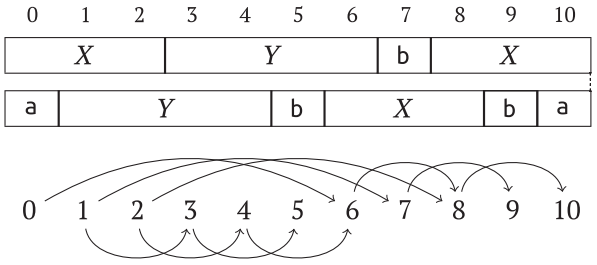
Для построения этого графа можно предварительно вычислить список позиций в слове x, покрытых вхождением $\psi(X)$ в $\psi(L)$ или в $\psi(R)$.

Будем говорить, что две позиции *конфликтуют*, если они индексируют две разные постоянные.

Наблюдение. Уравнение разрешимо тогда и только тогда, когда в одной компоненте связности графа G нет конфликтующих позиций.

Алгоритм с квадратичным временем работы. После вычисления графа строятся его компоненты связности и проверяется условие на конфликтующие позиции. Стандартный эффективный алгоритм вычисления компонент связности займет время $O(n^2)$, потому что размер графа имеет порядок $O(n^2)$.

Пример. Рассмотрим вышеупомянутое уравнение $XY\mathtt{b}X = \mathtt{a}Y\mathtt{b}X\mathtt{ba}$ при $|X| = 3$ и $|Y| = 4$. Граф имеет две компоненты связности: {0,2,4,6,8,10} и {1,3,5,7,9}. Позиции 0 и 10 соответствуют букве a, а позиции 5, 7 и 9 – букве b. Конфликтующих позиций не существует, и имеется единственное решение, приводящее к слову abababababa.



Алгоритм с линейным временем работы. Алгоритм можно ускорить, уменьшив число ребер в G, и это довольно просто. Достаточно рассматривать ребра (i, j), $i < j$, для которых i и j – позиции последовательных вхождений $\psi(X)$ в $\psi(L)$ или в $\psi(R)$ (см. рисунок), быть может, объединив два списка. Компоненты связности нового графа удовлетворяют наблюдению, и алгоритм теперь работает за линейное время, потому что размер графа линеен, т. к. из каждой позиции исходит не более двух ребер.

Примечания

В работе Makanin [181] показано, что задача разрешима, когда длины, ассоциированные с переменными, не заданы. Известно, что она NP-трудная, но вопрос о том, к какому именно классу NP-задач она относится, открыт.

Самые быстрые из известных алгоритмов работают за экспоненциальное время (см. [176, глава 12] и имеющиеся там ссылки). Если бы мы знали, что самое короткое решение имеет лишь однократно экспоненциальную длину, то можно было бы предложить простой NP-алгоритм его нахождения. Неизвестен пример уравнения, для которого длина самого короткого решения больше однократно экспоненциального, но до сих пор не доказано, что так обстоит дело всегда.

120. Разнородные факторы над трехбуквенным алфавитом

Слово w называется разнородным (diverse), если количества вхождений букв в него попарно различны (некоторые буквы могут не встречаться в w). В этой задаче рассматриваются разнородные факторы, встречающиеся в слове $x \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$.

Пример. Слово `aab` разнородное, но слово `aa` и пустое слово таковыми не являются. Слово `abbccc` разнородно, но слово `abcabcabc` не имеет ни одного разнородного фактора. Самым длинным разнородным фактором слова `cbaababacccbba` является `cbaabaccc`.

Очевидно, что любое слово длины, не большей 2, не имеет разнородных факторов, а слово длины 3 не является разнородным, если представляет собой перестановку трех букв. Имеет место следующее наблюдение.

Наблюдение 1. Слово $x \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$, $|x| \geq 3$, не имеет разнородных факторов тогда и только тогда, когда его префикс длины 3 не является разнородным (т. е. представляет собой перестановку трех букв) и является периодом x.

Вопрос. Спроектировать алгоритм, который за линейное время находит самый длинный разнородный фактор слова x над трехбуквенным алфавитом.

[**Указание:** примите во внимание доказанное ниже ключевое свойство.]

Ключевое свойство. Если слово $x[0..n-1] \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ имеет разнородный фактор, то оно имеет самый длинный разнородный фактор $w = x[i..j]$, для которого либо $0 \leq i < 3$, либо $n-3 \leq j < n$, т. е.

$$w \in \bigcup_{i=0}^{2} Pref(x[i..n-1]) \cup \bigcup_{j=n-3}^{n-1} Suff(x[0..j]).$$

Решение
Поскольку проверка условия в наблюдении 1 занимает линейное время, остается рассмотреть случай, когда слово x содержит разнородный фактор. Прежде чем обсуждать алгоритм, докажем ключевое свойство.

Доказательство проведем от противного. Предположим, что x имеет самый длинный разнородный фактор w такой, что $x = uwv$, где $|u| \geq 3$ и $|v| \geq 3$. Иначе говоря, x имеет фактор вида *abcwdef*, где a, b, c, d, e и f – буквы. Рассмотрим все случаи, соответствующие разным количествам вхождений букв в w в три соседние с w позиции справа и слева. Без ограничения общности можно предположить, что

$$|w|_a < |w|_b < |w|_c.$$

Следующее наблюдение существенно ограничивает число случаев, подлежащих рассмотрению.

Наблюдение 2. Если w – самый длинный разнородный фактор *abcwdef*, где a, b, c, d, e и f – буквы, и $|w|_a < |w|_b < |w|_c$, то

$$c \notin \{c, d\} \text{ и } |w|_a + 1 = |w|_b = |w|_c - 1.$$

К сожалению, все-таки придется рассмотреть несколько случаев для букв от a до f, но все они приводят к противоречию с нерасширяемостью разнородного фактора w в локальном окне *abcwdef*. В конечном итоге вопрос сводится к шести случаям, доказательство которых мы оставляем читателю.

Из ключевого свойства следует, что задача сводится к нескольким применениям более простой версии: задач о разнородном префиксе и суффиксе, т. е. к вычислению самого длинного разнородного префикса и самого длинного разнородного суффикса слова. Чтобы получить результат, их необходимо вычислить для всех трех суффиксов $x[i\,..n-1]$, $0 \leq i < 3$, и для всех трех префиксов $x[0..j]$, $n - 3 \leq j < n$ слова x.

Решение задачи о самом длинном разнородном префиксе за линейное время. Мы опишем только вычисление самого длинного разнородного префикса x, поскольку остальные случаи либо аналогичны, либо симметричны.

Пример. Самым длинным разнородным префиксом слова $y =$ `cbaabacccbba` является `cbaabaccc`; как видно из таблицы, он заканчивается в позиции 8. На самом деле можно проверить, что это самый длинный разнородный фактор y.

i	0	1	2	3	4	5	6	7	8	9	10	11	
$y[i]$		c	b	a	a	b	a	c	c	c	b	b	a
$\|y\|_a$	0	0	0	1	2	2	3	3	3	**3**	3	3	4
$\|y\|_b$	0	0	1	1	1	2	2	2	2	**2**	3	4	4
$\|y\|_c$	0	1	1	1	1	1	1	2	3	**4**	4	4	4

Нахождение самого длинного разнородного префикса x производится в онлайновом режиме, по мере поступления x. Количества вхождений всех трех букв вычисляются для последовательных префиксов. Наибольший индекс, для которого эти величины попарно различны, дает искомый префикс.

Примечания

Заметим, что самый длинный разнородный фактор может быть гораздо короче слова, как, например, `ccbaaaaaa` в слове `aaaabccbaaaaaa`, и что протяженность границы 3 в ключевом свойстве нельзя уменьшить до 2: контпримером служит слово `abcacbacba`, для которого самый длинный разнородный фактор равен `cacbac`.

Эта задача предлагалась на 25-й Польской олимпиаде по информатике под названием «Три башни».

121. Наибольшая возрастающая подпоследовательность

В этой задаче мы рассмотрим слово x над алфавитом целых положительных чисел. Возрастающей подпоследовательностью x называется подпоследовательность $x[i_0]x[i_1]...x[i_{l-1}]$, где $0 \le i_0 < i_1 < ... < i_{l-1} < |x|$ и $x[i_0] \le x[i_1] \le \cdots \le x[i_{l-1}]$.

Пример. Пусть $x =$ `3 6 4 10 1 15 13 4 19 16 10`. Наибольшей (самой длинной) возрастающей подпоследовательностью x является $y =$ `3 4 10 13 16` длины 5. Есть и другая такая подпоследовательность – $z =$ `3 4 10 13 19`. Если добавить к x число 21, то длина возрастающих подпоследовательностей y и z увеличится до 6. Но если добавить число 18, то только y `18` останется наибольшей подпоследовательностью, поскольку z `18` уже не является возрастающей.

Вопрос. Показать, что алгоритм Lis на месте вычисляет максимальную длину возрастающей подпоследовательности слова x за время $O(|x| \log |x|)$.

```
Lis(непустое слово x над алфавитом целых положительных чисел)
 1  l ← 1
 2  for i ← 1 to |x| − 1 do
 3      (a, x[i]) ← (x[i], ∞)
 4      k ← min{j : 0 ≤ j ≤ l и a < x[j]}
 5      x[k] ← a
 6      if k = l then
 7          l ← l + 1
 8  return l
```

Продолжение примера. В таблицах ниже показано слово x до и после применения алгоритма Lis.

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	3	6	4	10	1	15	13	4	19	16	10

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	1	4	4	10	16	∞	∞	∞	∞	∞	∞

Внимательно изучив алгоритм Lis, можно заметить, что слово $x[0..l-1]$, вычисленное к моменту остановки алгоритма, возрастающее, но обычно не является подпоследовательностью x, что хорошо видно на примере. Это подводит нас к следующему вопросу.

Вопрос. Спроектировать алгоритм, который вычисляет наибольшую возрастающую подпоследовательность слова x за время $O(|x| \log |x|)$.

Решение

Сложность. Заметим, что значения, хранящиеся в префиксе $x[0..l-1]$ слова x, удовлетворяют условиям $x[0] \le x[1] \le \cdots \le x[l-1]$, а за ними следуют значения ∞ (эти значения могут отличаться от начальных значений в $x[0..l-1]$). Таким образом, инструкцию в строке 4 можно реализовать за время $O(\log |x|)$ и даже за время $O(\log l)$, где l – длина наибольшей возрастающей подпоследовательности x. В итоге получаем время работы $O(|x| \log |x|)$ или $O(|x| \log l)$. Ясно, что объем памяти сверх необходимой для входных данных постоянен.

Правильность алгоритма Lis опирается на следующий инвариант цикла for: для любого j, $0 \le j < l$, $x[j]$ является наименьшим значением (обозначим его $best[j]$), которым может заканчиваться возрастающая подпоследовательность длины j в $x[0..i]$.

Это утверждение, очевидно, верно в самом начале, когда $j = 0$ и $x[0]$ еще не изменилось. В результате выполнения строк 4–7 либо уменьшается $best[k]$, либо расширяется ранее вычисленная наибольшая возрастающая подпоследовательность.

Наибольшая возрастающая подпоследовательность. Алгоритм Lis можно модифицировать так, чтобы он вычислял интересующую нас подпоследовательность. Заведем отдельный массив, в котором для каждой позиции в $x[0..l]$ хранится позиция ее предшественника в возрастающей подпоследовательности. После инициализации нужно будет при обработке текущего элемента $x[i]$ записывать в соответствующий ему элемент массива предшественника элемент, который он заменяет. После завершения алгоритма для нахождения наибольшей возрастающей последовательности нужно будет проследовать по ссылкам на предшественников, начиная с самого большого (самого правого) значения в массиве.

Ниже показан массив предшественников для индексов в примере выше. Две наибольшие возрастающие подпоследовательности извлекаются путем прохода справа налево, начиная с индекса 9 или 8.

j	0	1	2	3	4	5	6	7	8	9	10
$pred[j]$	–	0	0	2	–	3	3	2	6	6	3

Примечания

Заметим, что этот алгоритм решает двойственную задачу и вычисляет наименьшее число непересекающихся строго возрастающих подпоследовательностей, на которые можно разбить заданное слово.

Вычисление наибольшей возрастающей подпоследовательности – хрестоматийный пример динамического программирования (см., например, [226]).

Если слово является перестановкой первых n целых положительных чисел, то вопрос оказывается связан с представлением перестановок таблицами Юнга; см. главу, написанную Lascoux, Leclerc и Thibon в книге [176, глава 5], где описан алгоритм Шенстеда в этом контексте. В этом случае время вычислений можно уменьшить до $O(n \log\log n)$ (см. [241]) и даже до $O(n \log\log l)$ (см. [95] и имеющиеся там ссылки).

122. Неустранимые множества и слова Линдона

Слова Линдона часто возникают совершенно неожиданно в, казалось бы, никак не связанных с ними задачах. Мы покажем, как они появляются в качестве основных компонент некоторых декомпозиций.

Задача касается неустранимых множеств слов. Множество $X \subseteq \{0,1\}^*$ называется неустранимым, если любое бесконечное двоичное слово имеет фактор, принадлежащий X.

Обозначим $\mathcal{N}_k$ множество ожерелий длины $k > 0$. Ожерельем называется лексикографически наименьшее слово в своем классе сопряженности. Любое ожерелье является степенью числа Линдона.

Пример. Множество $\mathcal{N}_3 = \{000,001,011,111\}$ устранимо, т. к. слово $(01)^\infty$ не имеет факторов, принадлежащих $\mathcal{N}_3$. Но после перемещения последней 1 в начало мы получаем неустранимое множество $\{000,100,101,111\}$.

Наблюдение 1. Если $X \subseteq \{0,1\}^k$ неустранимо, то $|X| \geq |\mathcal{N}_k|$.

Действительно, для любого $w \in \{0,1\}^k$ факторами длины k слова w^∞ являются все слова, сопряженные w, и по меньшей мере одно из них должно принадлежать неустранимому множеству X. Наш первый вопрос дает более гибкое условие неустранимости.

Вопрос. Показать, что если для любого ожерелья $y \in \{0,1\}^*$, $|y| \geq 2k$, слово y^2 содержит слово, принадлежащее $X \subseteq \{0,1\}^k$, то X неустранимо.

Изобретательное построение неустранимого множества основано на понятии предпростых слов. Предпростое слово w – это префикс ожерелья.

Наблюдение 2. Предпростое слово является префиксом степени слова Линдона.

Это наблюдение является обоснованием специальной декомпозиции предпростого слова w в виде $u^e v$, где u – слово Линдона, $e \geq 1$, $|v| < |u|$ и u – самое короткое из возможных. Начало и конец w определяются как $head(w) = u^e$ и $tail(w) = v$.

Пример. $w = \texttt{0101110}$ разлагается в произведения $\texttt{01}^2 \cdot \texttt{110}$, $\texttt{01011} \cdot \texttt{10}$ и $\texttt{010111} \cdot \texttt{0}$ с префиксами Линдона 01, 01011 и 010111 соответственно. Его специальная декомпозиция – $\texttt{01011} \cdot \texttt{10}$, $head(w) = \texttt{01011}$ и $tail(w) = \texttt{10}$.

Заметим, что v – необязательно собственный префикс u, т. е. $|u|$ обычно не является периодом w. Ясно, что такая факторизация предпростого слова всегда существует и является основным инструментом в рассматриваемой задаче.

Вопрос. Показать, что существует неустранимое множество $X_k \subseteq \{\texttt{0},\texttt{1}\}^k$ размера $|\mathcal{N}_k|$. Отсюда следует, что $|\mathcal{N}_k|$ – наименьший размер такого множества.

[**Указание:** рассмотрите слова вида $tail(w) \cdot head(w)$.]

Решение

Сначала докажем утверждение первого вопроса, которое дает условие неустранимости подмножества $\{\texttt{0},\texttt{1}\}^k$, позволяющее избавиться от бесконечности.

Предположим противное – что существует бесконечное слово x, не имеющее фактора из множества $X \subseteq \{\texttt{0},\texttt{1}\}^k$. Рассмотрим слово u, которое два раза входит в x без перекрытий, т. е. uvu – фактор x и $|u|, |v| \geq k$. Пусть y – ожерелье, сопряженное uv. Из нашего предположения следует, что существует слово $w \in X$, являющееся фактором y^2; однако, в силу неравенств $|u|, |v| \geq k$, это слово входит также в uvu и потому в x. Это противоречит предположению о том, что x не содержит ни одного слова, принадлежащего X, и тем самым завершает доказательство.

Наименьшее неустранимое множество. Искомое неустранимое множество имеет вид

$$X_k = \{tail(w) \cdot head(w) : w \in \mathcal{N}_k\}.$$

Например, $X_4 = \{\texttt{0000},\texttt{0001},\texttt{1001},\texttt{0101},\texttt{1011},\texttt{1111}\}$ и X_7 содержат все 20 слов (концы *tail* подчеркнуты):

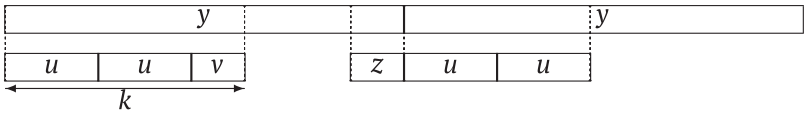
0000000,0000001,1000001,0100001,1100001,0010001,0110001,
1010001,1110001,1001001,0100101,1100101,0110011,1010011,
1110011,1010101,1101011,1011011,1110111,1111111.

Прежде чем доказывать, что X_k – ответ на второй вопрос, сформулируем полезное свойство ожерелий и специальных декомпозиций, его формальное доказательство оставляем читателю (см. примечания).

Наблюдение 2. Пусть $w \in \{0,1\}^k$ – предпростой префикс ожерелья y длины не менее $2k$ с декомпозицией $u^e \cdot v$, и пусть z – суффикс y длины $|v|$. Тогда $u^e \cdot z$ – ожерелье, имеющее декомпозицию $u^e \cdot z$.

Теорема. Множество X_k неустранимо.

Доказательство. В силу утверждения из первого вопроса достаточно показать, что для каждого ожерелья y размера не менее $2k$ слово y^2 имеет фактор, принадлежащий X_k. Зафиксируем такое y, и пусть $u^e \cdot v$ – декомпозиция предпростого префикса y длины k. Наша цель – найти фактор y^2, который принадлежит X_k.



Пусть z – суффикс y длины $|v|$. Согласно наблюдению 2, слово $w = u^e z$ является ожерельем и $u^e \cdot z$ – его декомпозиция. Следовательно, $z \cdot u^e \in X_k$. Так как z – суффикс y, а u^e – префикс y, zu^e является фактором y^2 (см. рисунок). Это завершает доказательство.

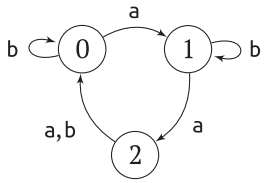
Примечания
Эта задача решена в работе Champarnaud et al. [53]. Вопрос о проверке предпростоты слова рассматривается в задаче 42.

123. Синхронизация слов

В этой задаче рассматривается синхронизация композиции функций, отображающих множество $I_n = \{0, 1, \ldots, n-1\}$ в себя для целого положительного n. Для двух таких функций f_a и f_b будем говорить, что слово $w \in \{a,b\}^+$ кодирует их композицию $f_w = h_0 \circ h_1 \circ \ldots \circ h_{|w|-1}$, где $h_i = f_a$, если $w[i] = a$, и $h_i = f_b$, если $w[i] = b$. (Заметим, что функции применяются к элементам I_n в порядке убывания i, как обычно, т. е. справа налево.) Говорят, что слово w является *синхронизирующим*, если множество $f_w(I_n)$ содержит всего один элемент.

Полезно также понятие синхронизатора пары: слово $u \in \{a,b\}^*$ является синхронизатором пары (i, j), $i, j \in I_n$, если $f_u(i) = f_u(j)$.

Пример. Рассмотрим две функции: $g_a(i) = (i + 1) \bmod n$, $g_b(i) = \min\{i, g_a(i)\}$. Для $n = 3$ они иллюстрируются автоматом ниже. Как показано в таблице, слово $w = baab$ является синхронизирующим, потому что образ множества $\{0,1,2\}$ относительно g_w – одноэлементное множество $\{0\}$. Это слово, очевидно, синхронизирует каждую пару, но в таблице присутствуют и дополнительные синхронизаторы, например b для пары (0,2), baa для пары (0,1) и ba для пары (1,2).



w			b		a		a		b	
$g_w(0)$	=	0	←	2	←	1	←	0	←	0
$g_w(1)$	=	0	←	0	←	2	←	1	←	1
$g_w(2)$	=	0	←	2	←	1	←	0	←	2

Для любого целого положительного n слово $w = \mathtt{b}(\mathtt{a}^{n-1}\mathtt{b})^{n-2}$ синхронизирующее для функций $g_\mathtt{a}$ и $g_\mathtt{b}$. Труднее увидеть, что в этом частном случае оно еще и кратчайшее. Это дает нам квадратичную нижнюю границу длины синхронизирующих слов.

Вопрос. Показать, что пара функций допускает синхронизирующее слово тогда и только тогда, когда существует синхронизатор для каждой пары (i, j) элементов I_n.

[**Указание:** вычислите синхронизирующее слово по синхронизаторам.]

Вопрос. Показать, как за квадратичное время проверить, допускает ли пара функций синхронизирующее слово.

[**Указание:** проверить синхронизаторы пар.]

Решение

Часть «только тогда» в утверждении из первого вопроса очевидна, потому синхронизирующее слово является синхронизатором каждой пары элементов I_n. Остается только доказать часть «тогда», т. е. показать, что синхронизирующее слово существует, если существует синхронизатор для каждой пары (i, j) элементов I_n. Алгоритм SyncWord строит глобальное синхронизирующее слово.

```
SyncWord(функции f_a и f_b, отображающие I_n в себя)
 1  J ← I_n
 2  w ← ε
 3  while |J| > 1 do
 4      i, j ← любые два различных элемента J
 5      u ← синхронизатор (i, j)
 6      J ← f_u(J)
 7      w ← u·w
 8  return w
```

Существование синхронизаторов пар. Чтобы ответить на второй вопрос, мы должны проверить, имеет ли каждая пара элементов синхронизатор.

Для этого рассмотрим ориентированный граф G, вершинами которого являются пары (i, j), $i, j \in I_n$, а ребра имеют вид $(i, j) \to (p, q)$, где $p = f_a(i)$ и $q = f_a(j)$ для некоторой буквы $a \in \{\texttt{a},\texttt{b}\}$.

Тогда пара (i, j) имеет синхронизатор тогда и только тогда, когда существует путь из (i, j) в вершину вида (p, p). Проверить это можно с помощью стандартного алгоритма обхода графа.

Если для какой-то пары не существует синхронизатора, то, в силу первого утверждения, функции не имеют синхронизирующего слова.

Время обработки графа имеет порядок $O(n^2)$, что нам и нужно. Но для нахождения синхронизирующего слова с помощью алгоритма SyncWord требуется кубическое время, при условии что операции над словами занимают постоянное время.

Примечания

Когда обе функции являются буквами, воздействующими на множество состояний конечного автомата, синхронизирующее слово называется также словом сброса; см., например, работу [29].

Хотя описанный выше метод требует квадратичного времени (достаточно проверить существование локальных синхронизаторов), фактическое генерирование синхронизирующих слов может занимать кубическое время. Связано это с тем, что длина сгенерированного слова может быть кубической. Так называемая гипотеза Черны утверждает, что верхняя граница синхронизирующего слова всего лишь квадратичная, но лучшая из известных оценок – лишь $\frac{114}{685}n^3 + O(n^2)$ (улучшает предыдущую оценку $\frac{114}{684}n^3 + O(n^2)$); см. работу [229] и имеющиеся в ней ссылки.

124. Сейфооткрывающие слова

В этой задаче рассматривается специальная недетерминированная версия синхронизирующих слов. Задан граф G, ребра которого помечены символами и в котором имеется единственная вершина-сток s такая, что все исходящие из нее ребра являются петлями. Требуется найти синхронизирующее слово S такое, что любой случайно выбранный путь, помеченный словом S, приходит в s независимо от начальной вершины.

В общем случае это трудная задача, но мы рассмотрим очень специальный случай *сейфооткрывающих слов*. В нем демонстрируются некоторые удивительные операции со словами над алфавитом $\texttt{B}_n = \{\texttt{0},\texttt{1}\}^n$.

Игровое описание задачи. *Поворотный сейф* запирается на круговой замок, по окружности которого на равном расстоянии друг от друга расположено n неразличимых кнопок. Каждая кнопка соединена с переключателем на другой стороне дверцы, который снаружи невидим. Переключатель может находиться в состоянии 0 (выключен) или 1 (включен). На каждом ходе разре-

шается нажать несколько кнопок одновременно. Если в результате все переключатели окажутся включены, то сейф откроется и останется открытым. Непосредственно перед каждым ходом круговой замок проворачивается в случайное положение, при этом состояния переключателей не изменяются. Начальная конфигурация неизвестна.

Цель состоит в том, чтобы найти последовательность ходов $A_i \in \mathrm{B}_n$

$$\mathcal{S}(n) = A_1 \cdot A_2 \ldots A_{2^n-1},$$

которая называется *сейфооткрывающим словом*, такую, что некоторый префикс этого слова открывает сейф.

В предположении, что позиции кнопок пронумерованы от 1 до n, начиная с верхней по часовой стрелке, ход описывается n-разрядным словом $b_1 b_2 \ldots b_n$, означающим, что кнопка в позиции i нажата тогда и только тогда, когда $b_i = 1$. Позиции фиксированы, но сами кнопки могут перемещаться, т. е. изменять свою позицию.

Пример. Можно проверить, что единственное кратчайшее слово, открывающее сейф с двумя кнопками, – $\mathcal{S}(2) = 11 \cdot 01 \cdot 11$.

Вопрос. Пусть n – степень 2. Построить сейфооткрывающее слово $\mathcal{S}(n)$ длины $2^n - 1$ над алфавитом $\mathtt{B}_n$.

Абстрактное описание задачи. Каждый ход A_i рассматривается как двоичное слово длины n. Обозначим ≡ отношение сопряженности (циклического сдвига), являющееся отношением эквивалентности. Пусть $G_n = (V, E)$ – ориентированный граф, в котором вершинами (множество V) являются двоичные слова длины n, представляющие конфигурации кругового замка, а для $A \in \mathtt{B}_n$ проведено ребро

$$u \xrightarrow{A} (v \text{ xor } A)$$

для каждого $v \equiv u$, если $u \neq 1^n$, и ребро $1^n \xrightarrow{A} 1^n$ в противном случае.

Пример. Для $u = \mathtt{0111}$ и $A = \mathtt{1010}$ существуют четыре вершины v, сопряженные u: `0111`, `1110`, `1101`, `1011`, – и соответственно ребра:

$$u \xrightarrow{A} \mathtt{1101},\ u \xrightarrow{A} \mathtt{0100},\ u \xrightarrow{A} \mathtt{0111},\ u \xrightarrow{A} \mathtt{0001}.$$

Цель заключается в том, чтобы найти слово $\mathcal{S} = A_1 \cdot A_2 \ldots A_{2^n-1}$, принадлежащее $\mathtt{B}_n^*$, – такое, что каждый случайно выбранный путь в G_n, помеченный словом $\mathcal{S}$, ведет в сток 1^n независимо от начальной вершины.

Решение
Над словами $X, Y \in \mathtt{B}_n^*$ определим две операции:

$$X \odot Y = X \cdot Y[0] \cdot X \cdot Y[1] \cdot X \ldots X \cdot Y[N-1] \cdot X$$

является словом в $\mathtt{B}_n^*$, а

$$X \otimes Y = X[0]Y[0] \cdot X[1]Y[1] \ldots X[N-1]Y[N-1]$$

является словом в $\sum_{2n}^*$, при условии что $|X| = |Y| = N$.
Например, $(\mathtt{01 \cdot 11 \cdot 10}) \otimes (\mathtt{10 \cdot 11 \cdot 00}) = \mathtt{0110 \cdot 1111 \cdot 1000}$.

Рекуррентные соотношения. Обозначим $Z(n) = (0^n)^{2^n-1}$ слово длины $2^n - 1$ над B_n. Пусть $\mathcal{S}(n)$ – сейфооткрывающее слово для $n \geq 2$. Тогда $\mathcal{S}(2n)$ можно вычислить следующим образом:
(i) $\mathcal{B}(n) = \mathcal{S}(n) \otimes \mathcal{S}(n)$, $\mathcal{C}(n) = Z(n) \otimes \mathcal{S}(n)$;
(ii) $\mathcal{S}(2n) = \mathcal{B}(n) \odot \mathcal{C}(n)$.

Пример. Зная, что $\mathcal{S}(2) = \mathtt{11 \cdot 01 \cdot 11}$, получаем

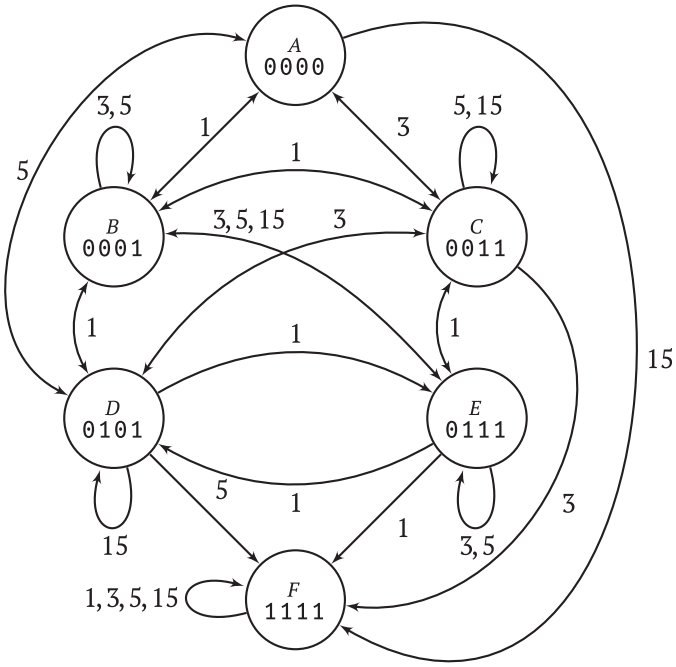
$\mathcal{B}(2) = \mathtt{1111 \cdot 0101 \cdot 1111}$,
$\mathcal{C}(2) = \mathtt{0011 \cdot 0001 \cdot 0011}$ и
$\mathcal{S}(4) = \mathtt{1111 \cdot 0101 \cdot 1111 \cdot}\mathbf{\mathtt{0011}}\mathtt{\cdot 1111 \cdot 0101 \cdot 1111 \cdot}\mathbf{\mathtt{0001}}\mathtt{\cdot 1111 \cdot 0101 \cdot 1111}$
$\mathtt{\cdot}\mathbf{\mathtt{0011}}\mathtt{\cdot 1111 \cdot 0101 \cdot 1111}$.

Утверждение. Если n – степень 2, то рекуррентное соотношение (ii) правильно генерирует сейфооткрывающие слова.

Доказательство. Слово $\mathcal{B}(n)$ означает применение в точности одинаковых действий к кнопкам, расположенным в противоположных позициях на окружности. Иначе говоря, кнопки в позициях i и $i + n/2$ либо одновременно нажаты, либо одновременно не нажаты. Поэтому в какой-то момент слово $\mathcal{B}(n)$ достигает требуемой конфигурации, если начальной является конфигурация, в которой для каждой пары $(i, i + n/2)$ соответствующие кнопки синхронизированы, т. е. находятся в одинаковом состоянии.

Но именно в этом и состоит роль слова $\mathcal{C}(n)$. После выполнения его префикса $C_1 \cdot C_2 \cdots C_i$ для некоторого i все пары противоположных кнопок будут синхронизированы. Затем применение всего слова $\mathcal{B}(n)$ открывает сейф, что нам и требуется. ■

Иллюстрация на сжатом графе. Размер алфавита $\mathtt{B}_n$ экспоненциально велик, но в решении используется лишь малая его часть, обозначаемая $\mathtt{B}'_n$. Вместо G_n рассмотрим его сжатую версию G'_n, вершинами которой являются наименьшие представители классов сопряженности, а ребра помечены словами только из $\mathtt{B}'_n$. На рисунке показан граф G'_4, в котором буквы 0001, 0011, 0101, 1111 алфавита $\mathtt{B}'_4$ записаны сокращенно – 1, 3, 5, 15, а вершины 0000, 0001, 0011, 0101, 0111, 1111 являются ожерельями, представляющими классы сопряженности, и для краткости обозначены соответственно A, B, C, D, E, F.



Заметим, что независимо от начальной вершины в G'_4 каждый путь, помеченный 15,5, 15,3, 15,5, 15,1, 15,5, 15,3, 15,5, 15, ведет в вершину 1111.

В правильности этой последовательности можно убедиться, начав со всего множества узлов и применяя соседние переходы. В конце получится множество $\{F\}$. Действительно, имеем:

$$\{A,B,C,D,E,F\} \xrightarrow{15} \{B,C,D,E,F\} \xrightarrow{5} \{A,B,C,E,F\} \xrightarrow{15}$$
$$\{B,C,E,F\} \xrightarrow{3} \{A,B,E,D,F\} \xrightarrow{15} \{B,E,D,F\} \xrightarrow{5} \{A,B,E,F\}$$
$$\xrightarrow{15} \{B,E,F\} \xrightarrow{1} \{A,D,C,F\} \xrightarrow{15} \{D,C,F\} \xrightarrow{5} \{A,C,F\}$$
$$\xrightarrow{15} \{C,F\} \xrightarrow{3} \{A,D,F\} \xrightarrow{15} \{D,F\} \xrightarrow{5} \{A,F\} \xrightarrow{15} \{F\}.$$

Примечания

Материал этой задачи основан на оригинальной работе Guo, опубликованной по адресу https://www.austms.org.au/Publ/Gazette/2013/Mar13/Puzzle.pdf. Длина сейфооткрывающих слов не изучалась, но показано, что такого слова не существует, если n не является степенью 2.

Существует альтернативное описание сейфооткрывающей последовательности. Предположим, что двоичные слова стандартным образом представлены неотрицательными целыми числами. Тогда $\mathcal{S}(2) = 3 \cdot 1 \cdot 3$ и

$$\mathcal{S}(4) = 15 \cdot 5 \cdot 15 \cdot \mathbf{3} \cdot 15 \cdot 5 \cdot 15 \cdot \mathbf{1} \cdot 15 \cdot 5 \cdot 15 \cdot \mathbf{3} \cdot 15 \cdot 5 \cdot 15.$$

Рекуррентные соотношения (i) и (ii) теперь выглядят гораздо короче:

$$\mathcal{S}(2\mathrm{n}) = (2^n \times \mathcal{S}(n) + \mathcal{S}(n)) \odot \mathcal{S}(n),$$

где операции $+$, $\times$ – покомпонентные арифметические операции над последовательностью целых чисел.

125. Суперслова укороченных перестановок

Слово над алфавитом натуральных чисел называется n-перестановкой, если каждое число из множества $\{1, 2, \ldots, n\}$ встречается в нем ровно один раз (см. задачи 14 и 15). Слово называется укороченной n-перестановкой, если это n-перестановка, из которой удален последний элемент. Наличие биективного отображения между стандартными и укороченными перестановками для данного n означает, что всего существует $n!$ укороченных n-перестановок.

В этой задаче наша цель – построить самое короткое суперслово для всех укороченных n-перестановок. Их длина равна $n! + n - 2$, что дает очевидную нижнюю границу. Например, 3213123 – самое короткое суперслово укороченных 3-перестановок, потому что оно содержит все укороченные 3-перестановки

32, 21, 13, 31, 12 и 23.

Вопрос. Показать, как построить суперслово длины $n! + n - 2$, содержащее все укороченные n-перестановки, за время, линейно зависящее от длины результата.

[**Указание:** рассмотрите эйлеров цикл в подходящем графе.]

Решение

Задача сводится к нахождению эйлерова цикла в ориентированном графе $\mathcal{J}_n$ (графе Джексона), который очень похож на граф де Брёйна. Множество V_n вершин $\mathcal{J}_n$ состоит из всех слов, являющихся сочетаниями $n - 2$ элементов из множества $\{1, 2, \ldots, n\}$. Из каждой вершины $w = a_1a_2\ldots a_{n-2} \in V_n$ исходит два ребра:

$$a_1a_2\cdots a_{n-2} \xrightarrow{\ b\ } a_2\cdots a_{n-2}b,$$

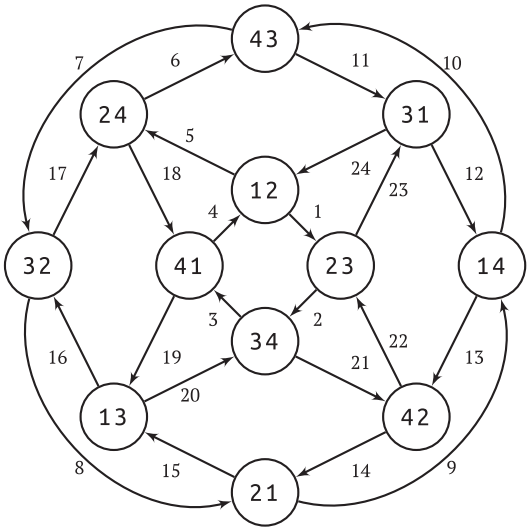
где $b \in \{1, 2, \ldots, n\} - \{a_1, a_2, \ldots, a_{n-2}\}$. Каждое такое ребро помечено числом b и соответствует укороченной перестановке $a_1a_2\ldots a_{n-2}b$. Граф $\mathcal{J}_4$ показан на рисунке ниже.

Наблюдение. Если $b_1b_2\ldots b_m$ – метка эйлерова цикла, начинающегося в вершине $a_1a_2\ldots a_{n-2}$, то $a_1a_2\ldots a_{n-2}b_1b_2\ldots b_m$ – кратчайшее суперслово укороченных n-перестановок.

Пример. В графе $\mathcal{J}_4$ эйлеров цикл 12 → 23 → 34 → 41 → 12 → 24 → 43 → 32 → 21 → 14 → 43 → 31 → 14 → 42 → 21 → 13 → 32 → 24 → 41 → 13 → 34 → 42 → 23 → 31 → 12 порождает суперслово длины $26 = 4! + 4 - 2$ с префиксом 12:

12 341243214314213241342312.

Чтобы ответить на вопрос, достаточно показать, что граф $\mathcal{J}_n$ является эйлеровым графом.



***Лемма* 22.** *Если имеются две укороченные n-перестановки с одинаковым множеством элементов, то одна достижима из другой в графе Джексона* $\mathcal{J}_n$.

Доказательство. Достаточно показать, что для каждой укороченной n-перестановки $a_1a_2\ldots a_{n-2}$ существует путь в $\mathcal{J}_n$ к ее циклическому сдвигу $a_2\ldots a_{n-2}a_1$ и к $a_2a_1a_3\ldots a_{n-2}$ (транспозиция первых двух элементов), потому что любая перестановка может быть разложена в последовательность таких перестановок.

Мы приведем набросок доказательства для укороченной перестановки вида $123\ldots n-2$ на репрезентативном примере перестановки 12345 для $n=7$. Положим $a=6$ и $b=7$. В графе $\mathcal{J}_7$ имеется путь

$$12345 \to 2345a \to 345ab \to 45ab2 \to 5ab23 \to ab234 \to b2345 \to 23451,$$

который показывает, что существует путь $12345 \xrightarrow{*} 23451$ из 12345 в циклический сдвиг 23451. Существует также путь $12345 \xrightarrow{*} 345ab$. Используя оба, получаем путь

$$12345 \xrightarrow{*} 345ab \xrightarrow{*} ab345 \to b3452 \to 34521 \xrightarrow{*} 21345,$$

который соответствует транспозиции первых двух элементов. Тем самым набросок доказательства закончен. ■

Из этой леммы следует, что граф Джексона сильно связный. Кроме того, он регулярный, т. е. полустепени захода и исхода для каждой вершины равны 2. Известно, что при таких условиях граф является эйлеровым. Поэтому, следуя приведенному выше наблюдению, мы можем построить искомое суперслово по эйлеровому циклу.

Примечания

Задача о построении кратчайшего суперслова всех укороченных n-перестановок была полностью решена в работах Jackson [151, 211].

Эта задача эквивалентна нахождению гамильтонова цикла в *реберном графе* $\mathcal{H}_n$ графа $\mathcal{J}_n$. Вершины $\mathcal{H}_n$, идентифицируемые укороченными перестановками, соответствуют ребрам $\mathcal{J}_n$: ребро (e, e') существует в $\mathcal{H}_n$ тогда и только тогда, когда начальная вершина ребра e' в $\mathcal{J}_n$ является конечной вершиной e.

Ребра $\mathcal{H}_n$ можно пометить следующим образом. Для каждой вершины $a_1a_2...a_{n-1}$ графа $\mathcal{H}_n$ в графе имеется два помеченных ребра:

$$a_1a_2...a_{n-1} \xrightarrow{1} a_2...a_{n-1}a_1 \text{ и } a_1a_2...a_{n-1} \,{*}88{*}\, a_2...a_{n-1}a_1a_n,$$

где $a_n \notin \{a_1, a_2, ..., a_{n-1}\}$. Эйлеров обход $\mathcal{J}_n$ соответствует гамильтонову циклу в $\mathcal{H}_n$. Например, эйлеров цикл в графе $\mathcal{J}_4$ из предыдущего примера ассоциирован с гамильтоновым циклом в $\mathcal{H}_4$ после добавления одного ребра в конце:

$$\begin{aligned}
&\texttt{123} \xrightarrow{0} \texttt{234} \xrightarrow{0} \texttt{341} \xrightarrow{0} \texttt{412} \xrightarrow{1} \texttt{124} \xrightarrow{0} \texttt{243} \xrightarrow{1} \texttt{432} \xrightarrow{0}\\
&\texttt{321} \xrightarrow{0} \texttt{214} \xrightarrow{0} \texttt{143} \xrightarrow{1} \texttt{431} \xrightarrow{1} \texttt{314} \xrightarrow{0} \texttt{142} \xrightarrow{1} \texttt{421} \xrightarrow{0}\\
&\texttt{213} \xrightarrow{1} \texttt{132} \xrightarrow{0} \texttt{324} \xrightarrow{0} \texttt{241} \xrightarrow{0} \texttt{413} \xrightarrow{1} \texttt{134} \xrightarrow{0} \texttt{342} \xrightarrow{1}\\
&\texttt{423} \xrightarrow{0} \texttt{231} \xrightarrow{1} \texttt{312} \xrightarrow{1} \texttt{123}.
\end{aligned}$$

Гамильтонов цикл, начинающийся в вершине `123`, идентифицируется словом меток $x = \texttt{000101000110101000101011}$.

Любопытно, что существует семейство слов x_n, описывающее гамильтонов цикл в графе $\mathcal{H}_n$ и имеющее очень компактное описание. Мы используем интересную операцию $\odot$ над словами, определенную следующим образом. Для двух двоичных слов u и $v = v[0..k-1]$ положим

$$u \odot v = u\,v[0]\,u\,v[1]\,u\,v\,|2]\,...\,u\,v[k-1].$$

Обозначим $\bar{u}$ операцию отрицания всех символов u. Тогда для $n \geq 2$ положим

$$x_2 = 00 \text{ и } x_{n+1} = \texttt{001}^{n-2} \odot \overline{x_n}.$$

Например, $x_3 = \texttt{00} \odot \texttt{11} = \texttt{001001}$ и $x_4 = \texttt{001} \odot \texttt{110 110} = (\texttt{0011 0011 0010})^2$. В работе [211] показано, что для $n \geq 2$ слово x_n описывает гамильтонов цикл в $\mathcal{H}_n$, начинающийся с вершины $123...n-1$.

Описанная здесь связь между словами и гамильтоновыми циклами похожа на связь между словами де Брёйна и гамильтоновыми циклами в графах де Брёйна.

Литература

[1] Z. Adamczyk and W. Rytter. A note on a simple computation of the maximal suffix of a string. *J. Discrete Algorithms*, 20:61–64, 2013.

[2] D. Adjeroh, T. Bell and A. Mukherjee. *The Burrows-Wheeler Transform*. Springer, 2008.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[4] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] C. Allauzen, M. Crochemore and M. Raffinot. Factor oracle: A new structure for pattern matching. In J. Pavelka, G. Tel and M. Bartosek, eds., *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics*, Milovy, Czech Republic, 27 November– 4 December 1999, Lecture Notes in Computer Science, vol. 1725, pp. 295–310. Springer, 1999.

[6] J. Allouche and J. O. Shallit. The ubiquitous Prouhet-Thue-Morse sequence. In C. Ding, T. Helleseth and H. Niederreiter, eds., *Sequences and Their Applications, proceedings SETA 98*, pp. 1–16. Springer-Verlag, 1999.

[7] J. Allouche and J. O. Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, 2003.

[8] J. Almeida, A. Costa, R. Kyriakoglou and D. Perrin. Profinite semigroups and symbolic dynamics. Personal communication, 2018.

[9] M. Alzamel, M. Crochemore, C. S. Iliopoulos, et al. How much different are two words with different shortest periods? In L. S. Iliadis, I. Maglogiannis and V. P. Plagianakos, eds., *Artificial Intelligence Applications and Innovations AIAI 2018 IFIP WG12.5 InternationalWorkshops, SEDSEAL, 5G-PINE, MHDW, and HEALTHIOT*, Rhodes, Greece, 25–27 May 2018, IFIP Advances in Information and Communication Technology, vol. 520, pp. 168–178. Springer, 2018.

[10] A. Amir and M. Farach. Efficient matching of nonrectangular shapes. *Ann. Math. Artif. Intell.*, 4:211–224, 1991.

[11] A. Amir, M. Farach and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994.

[12] A. Amir, C. S. Iliopoulos and J. Radoszewski. Two strings at Hamming distance 1 cannot be both quasiperiodic. *Inf. Process. Lett.*, 128:54–57, 2017.

[13] S. Amoroso and G. Cooper. Tessellation structures for reproduction of arbitrary patterns. *J. Comput. Syst. Sci.*, 5(5):455–464, 1971.

[14] A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991.

[15] A. Apostolico and R. Giancarlo. Pattern matching machine implementation of a fast test for unique decipherability. *Inf. Process. Lett.*, 18(3):155–158, 1984.

[16] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[17] G. Assayag and S. Dubnov. Using factor oracles for machine improvisation. *Soft Comput.*, 8(9):604–610, 2004.

[18] G. Badkobeh. Infinite words containing the minimal number of repetitions. *J. Discrete Algorithms*, 20:38–42, 2013.
[19] G. Badkobeh and M. Crochemore. Fewest repetitions in infinite binary words. *RAIRO Theor. Inf. Applic.*, 46(1):17–31, 2012.
[20] G. Badkobeh, G. Fici and S. J. Puglisi. Algorithms for anti-powers in strings. *Inf. Process. Lett.*, 137:57–60, 2018.
[21] R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
[22] H. Bai, A. Deza and F. Franek. On a lemma of Crochemore and Rytter. *J. Discrete Algorithms*, 34:18–22, 2015.
[23] B. S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In S. R. Kosaraju, D. S. Johnson and A. Aggarwal, eds., *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, 16–18 May 1993, San Diego, CA, pp. 71–80. ACM, 1993.
[24] B. S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.
[25] H. Bannai, M. Giraud, K. Kusano, W. Matsubara, A. Shinohara and J. Simpson. The number of runs in a ternary word. In J. Holub and J. Zdárek, eds., *Proceedings of the Prague Stringology Conference 2010*, Prague, Czech Republic, 30 August–1 September 2010, pp. 178–181. Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2010.
[26] H. Bannai, I. Tomohiro, S. Inenaga, Y. Nakashima, M. Takeda and K. Tsuruta. The «runs» theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. [27] P. Baturo, M. Piatkowski and W. Rytter. Usefulness of directed acyclic subword graphs in problems related to standard sturmian words. *Int. J. Found. Comput. Sci.*, 20(6):1005–1023, 2009. [28] M. Béal, F. Mignosi and A. Restivo. Minimal forbidden words and symbolic dynamics. In C. Puech and R. Reischuk, eds., *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science*, Grenoble, France, 22–24 February 1996, Lecture Notes in Computer Science, vol. 1046, pp. 555–566. Springer, 1996.
[29] M. Béal and D. Perrin. Synchronised automata. In V. Berthé and M. Rigo, eds., *Combinatorics, Words and Symbolic Dynamics. Encyclopedia of Mathematics and Its Applications*, pp. 213–240. Cambridge University Press, 2016.
[30] M.-P. Béal, M. Crochemore, F. Mignosi, A. Restivo and M. Sciortino. Forbidden words of regular languages. *Fundam. Inform.*, 56(1,2):121–135, 2003.
[31] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In M. E. Saks, ed., *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, LA, 5–7 January 1997, New Orleans, pp. 360–369. ACM/SIAM, 1997.
[32] J. Berstel. Langford strings are square free. *Bull. EATCS*, 37:127–128, 1989.
[33] J. Berstel and A. de Luca. Sturmian words, Lyndon words and trees. *Theor. Comput. Sci.*, 178(1–2):171–203, 1997.
[34] J. Berstel and J. Karhumäki. Combinatorics on words: A tutorial. *EATCS*, 79:178, 2003.
[35] J. Berstel, A. Lauve, C. Reutenauer and F. Saliola. *Combinatorics on Words*, CRM Monograph Series, vol. 27. Université de Montréal et American Mathematical Society, 2008.
[36] J. Berstel and D. Perrin. *Theory of Codes*. Academic Press, 1985.

[37] J. Berstel and A. Savelli. Crochemore factorization of sturmian and other infinite words. In *Mathematical Foundations of Computer Science 2006, 31st International Symposium*, *MFCS 2006*, Stará Lesná, Slovakia, 28 August–1 September 2006, Lecture Notes in Computer Science, vol. 4162, pp. 157–166. Springer, 2006.
[38] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.
[39] K. S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980.
[40] J. Bourdon and I. Rusu. Statistical properties of factor oracles. *J. Discrete Algorithms*, 9(1):57–66, 2011.
[41] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
[42] F. Brandenburg. Uniformly growing k-th power-free homomorphisms. *Theor. Comput. Sci.*, 23:69–82, 1983.
[43] D. Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.
[44] D. Breslauer, L. Colussi and L. Toniolo. Tight comparison bounds for the string prefix-matching problem. *Inf. Process. Lett.*, 47(1):51–57, 1993.
[45] D. Breslauer, R. Grossi and F. Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013.
[46] S. Brlek, D. Jamet and G. Paquin. Smooth words on 2-letter alphabets having same parity. *Theor. Comput. Sci.*, 393(1–3):166–181, 2008.
[47] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In R. A. Baeza-Yates, E. Chávez and M. Crochemore, eds., *Combinatorial Pattern Matching, CPM 2003*, Lecture Notes in Computer Science, vol. 2676, pp. 55–69. Springer, 2003.
[48] A. Carayol and S. Göller. On long words avoiding Zimin patterns. In H. Vollmer and B. Vallée, eds., *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017*, 8–11 March, 2017, Hannover, Germany, vol. 66 of LIPIcs, pp. 19:1–19:13. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.
[49] Y. Césari and M. Vincent. Une caractérisation des mots périodiques. *C. R. Acad. Sci.*, 286:1175, 1978.
[50] S. Chairungsee and M. Crochemore. Efficient computing of longest previous reverse factors. In Y. Shoukourian, ed., *Seventh International Conference on Computer Science and Information Technologies*, CSIT 2009, pp. 27–30. The National Academy of Sciences of Armenia Publishers, Yerevan, Armenia, 2009.
[51] S. Chairungsee and M. Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450(1):109–116, 2012.
[52] S. Chairungsee and M. Crochemore. Longest previous non-overlapping factors table computation. In X. Gao, H. D. and M. Han, eds., *Combinatorial Optimization and Applications – 11th International Conference, COCOA 2017*, Shanghai, China, 10–18 December, 2017, Proceedings, Part II, vol. 10628 *Lecture Notes in Computer Science*, pp. 483–491. Springer, 2017.
[53] J. Champarnaud, G. Hansel and D. Perrin. Unavoidable sets of constant length. *IJAC*, 14(2):241–251, 2004.
[54] S. Cho, J. C. Na, K. Park and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.*, 115(2):397–402, 2015.
[55] J. G. Cleary, W. J. Teahan and I. H. Witten. Unbounded length contexts for PPM. In J. A. Storer and M. Cohn, eds., *Proceedings of the IEEE Data Compression Con-*

ference, DCC 1995, Snowbird, UT, 28–30 March, 1995, pp. 52–61. IEEE Computer Society, 1995.

[56] J. G. Cleary and I. H. Witten. A comparison of enumerative and adaptive codes. *IEEE Trans. Inf. Theory*, 30(2):306–315, 1984.

[57] J. Clément, P. Flajolet and B. Vallée. The analysis of hybrid trie structures. In H. J. Karloff, ed., *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 25–27 January 1998, San Francisco, 25–27 January 1998, pp. 531–539. ACM/SIAM, 1998.

[58] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007.

[59] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.

[60] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM J. Comput.*, 26(3):803–856, 1997.

[61] R. Cori and D. Perrin. Automates et commutations partielles. *ITA*, 19(1):21–32, 1985.

[62] G. V. Cormack and R. N. Horspool. Algorithms for adaptive Huffman codes. *Inf. Process. Lett.*, 18(3):159–165, 1984.

[63] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, 3rd Edition. MIT Press, 2009.

[64] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[65] M. Crochemore. Sharp characterization of square-free morphisms. *Theor. Comput. Sci.*, 18(2):221–226, 1982.

[66] M. Crochemore. *Régularités évitables*. Thèse d'état, Université de Haute-Normandie, 1983.

[67] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[68] M. Crochemore. Longest common factor of two words. In H. Ehrig, R. A. Kowalski, G. Levi and U. Montanari, eds., *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, 23–27 March, 1987, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'87), vol. 249, Lecture Notes in Computer Science, pp. 26–36. Springer, 1987.

[69] M. Crochemore. String-matching on ordered alphabets. *Theor. Comput. Sci.*, 92(1):33–47, 1992.

[70] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[71] M. Crochemore, C. Epifanio, R. Grossi and F. Mignosi. Linear-size suffix tries. *Theor. Comput. Sci.*, 638:171–178, 2016.

[72] M. Crochemore, S. Z. Fazekas, C. S. Iliopoulos and I. Jayasekera. Number of occurrences of powers in strings. *Int. J. Found. Comput. Sci.*, 21(4):535–547, 2010.

[73] M. Crochemore, R. Grossi, J. Kärkkäinen and G. M. Landau. Computing the Burrows-Wheeler transform in place and in small space. *J. Discrete Algorithms*, 32:44–52, 2015.

[74] M. Crochemore, C. Hancart and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[75] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis and Y. Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 2019. In press.

[76] M. Crochemore and L. Ilie. Computing longest previous factors in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.
[77] M. Crochemore and L. Ilie. Maximal repetitions in strings. *J. Comput. Syst. Sci.*, 74(5):796–807, 2008.
[78] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter and T. Walén. Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26, 2013.
[79] M. Crochemore, L. Ilie and E. Seid-Hilmi. The structure of factor oracles. *Int. J. Found. Comput. Sci.*, 18(4):781–797, 2007.
[80] M. Crochemore, L. Ilie and L. Tinta. The «runs» conjecture. *Theor. Comput. Sci.*, 412(27):2931–2941, 2011.
[81] M. Crochemore, C. S. Iliopoulos, T. Kociumaka et al. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016.
[82] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, et al. The maximum number of squares in a tree. In J. Kärkkäinen and J. Stoye, eds., *Combinatorial Pattern Matching: 23rd Annual Symposium, CPM 2012*, Helsinki, Finland, 3–5 July, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7354, pp. 27–40. Springer, 2012.
[83] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, et al. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In S. Inenaga, K. Sadakane and T. Sakai, eds., *String Processing and Information Retrieval – 23rd International Symposium, SPIRE 2016*, Beppu, Japan, 18–20 October, 2016, Proceedings, vol. 9954, *Lecture Notes in Computer Science*, pp. 22–34, 2016.
[84] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, K. Stencel and T. Walen. New simple efficient algorithms computing powers and runs in strings. *Discrete Appl. Math.*, 163:258–267, 2014.
[85] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter and T. Walen. On the maximal number of cubic runs in a string. In A. Dediu, H. Fernau, and C. Martín-Vide, eds., *Language and Automata Theory and Applications, 4th International Conference, LATA 2010*, Trier, Germany, 24–28 May, 2010. Proceedings, vol. 6031, *Lecture Notes in Computer Science*, pp. 227–238. Springer, 2010.
[86] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter and T. Walen. The maximal number of cubic runs in a word. *J. Comput. Syst. Sci.*, 78(6):1828–1836, 2012.
[87] M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter and T. Walén. Efficient algorithms for three variants of the LPF table. *J. Discrete Algorithms*, 11:51–61, 2012.
[88] M. Crochemore, G. M. Landau and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.
[89] M. Crochemore and T. Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Inf. Process. Lett.*, 63(4):195–203, 1997.
[90] M. Crochemore, M. Lerest and P. Wender. An optimal test on finite unavoidable sets of words. *Inf. Process. Lett.*, 16(4):179–180, 1983.
[91] M. Crochemore and R. Mercas. On the density of Lyndon roots in factors. *Theor. Comput. Sci.*, 656:234–240, 2016.
[92] M. Crochemore, F. Mignosi and A. Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998.
[93] M. Crochemore, F. Mignosi, A. Restivo and S. Salemi. Text compression using antidictonaries. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, eds., *International Conference on Automata, Languages an Programming* (Prague, 1999), Lecture Notes in Computer Science, pp. 261–270. Springer-Verlag, 1999. Rapport I.G.M. 98–10, Université de Marne-la-Vallée.

[94] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):651–675, 1991.
[95] M. Crochemore and E. Porat. Fast computation of a longest increasing subsequence and application. *Inf. Comput.*, 208(9):1054–1059, 2010.
[96] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
[97] M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995.
[98] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing, Hong-Kong, 2002.
[99] M. Crochemore and G. Tischler. Computing longest previous non-overlapping factors. *Inf. Process. Lett.*, 111(6):291–295, 2011.
[100] M. Crochemore and Z. Tronícek. On the size of DASG for multiple texts. In A. H. F. Laender and A. L. Oliveira, eds., *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002*, Lisbon, Portugal, 11–13 September, 2002, Proceedings, vol. 2476, *Lecture Notes in Computer Science*, pp. 58–64. Springer, 2002.
[101] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, eds., *Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht*, vol. 1261 *Lecture Notes in Computer Science*, pp. 192–211. Springer, 1997.
[102] A. Deza and F. Franek. A *d*-step approach to the maximum number of distinct squares and runs in strings. *Discrete Appl.Math.*, 163(3):268–274, 2014.
[103] A. Deza, F. Franek and A. Thierry. How many double squares can a string contain? *Discrete Appl. Math.*, 180:52–69, 2015.
[104] F. Durand and J. Leroy. The constant of recognizability is computable for primitive morphisms. *CoRR*, abs/1610.05577, 2016.
[105] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
[106] J. Duval, R. Kolpakov, G. Kucherov, T. Lecroq and A. Lefebvre. Linear-time computation of local periods. *Theor. Comput. Sci.*, 326(1–3):229–240, 2004.
[107] M. Effros. PPM performance with BWT complexity: A new method for lossless data compression. In *Data Compression Conference, DCC 2000*, Snowbird, UT, 28–30 March, 2000, pp. 203–212. IEEE Computer Society, 2000.
[108] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pp. 593–597, 1973.
[109] H. Fan, N. Yao and H. Ma. Fast variants of the Backward-Oracle-Marching algorithm. In *ICICSE '09, Fourth International Conference on Internet Computing for Science and Engineering*, pp. 56–59. IEEE Computer Society, 2009.
[110] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach*, FL, 19–22 October, 1997, pp. 137–143. IEEE Computer Society, 1997.
[111] S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. *Int. J. Found. Comput. Sci.*, 20(6):967–984, 2009.
[112] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
[113] G. Fici, A. Restivo, M. Silva and L. Q. Zamboni. Anti-powers in infinite words. *J. Comb. Theory, Ser. A*, 157:109–119, 2018.
[114] N. J. Fine. Binomial coefficients modulo a prime. *Am. Math. Mon.*, 54(10, Part 1): 589–592, December 1947.
[115] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQproblem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, eds.,

Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, 5–7 July, 2006, Proceedings, vol. 4009 *Lecture Notes in Computer Science*, pp. 36–48. Springer, 2006.

[116] J. Fischer, Š. Holub, T. I. and M. Lewenstein. Beyond the runs theorem. In *22nd SPIRE*, Lecture Notes in Computer Science, vol. 9309, pp. 272–281, 2015.

[117] A. S. Fraenkel and J. Simpson. How many squares must a binary sequence contain? *Electr. J. Comb.*, 2, 1995.

[118] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998.

[119] F. Franek, A. S. M. S. Islam, M. S. Rahman and W. F. Smyth. Algorithms to compute the Lyndon array. *CoRR*, abs/1605.08935, 2016.

[120] H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de bruijn sequences. *Discrete Math.*, 23(3):207–210, 1978.

[121] A. Fruchtman, Y. Gross, S. T. Klein and D. Shapira. Weighted adaptive huffman coding. In A. Bilgin, M. W. Marcellin, J. Serra-Sagrista and J. A. Storer, eds., Data Compression Conference, DCC 2020, Snowbird, UT, 24–27 March 2020, pp. 368–385. IEEE, 2020. http://arxiv.org/abs/2005.08232vl.

[122] Z. Galil. On improving the worse case running time of the Boyer-Moore string matching algorithm. *Commun. ACM*, 22(9):505–508, 1979.

[123] Z. Galil and R. Giancarlo. On the exact complexity of string matching: Upper bounds. *SIAM J. Comput.*, 21(3):407–437, 1992.

[124] Z. Galil and J. I. Seiferas. Time-space-optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.

[125] R. G. Gallager. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory*, 24(6):668–674, 1978.

[126] J. Gallant, D. Maier and J. A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980.

[127] P. Gawrychowski, T. Kociumaka, J. Radoszewski, W. Rytter and T. Walen. Universal reconstruction of a string. In F. Dehne, J. Sack and U. Stege, eds, *Algorithms and Data Structures: 14th International Symposium, WADS 2015*, Victoria, BC, Canada, 5–7 August, 2015. Proceedings, vol. 9214, *Lecture Notes in Computer Science*, pp. 386–397. Springer, 2015.

[128] P. Gawrychowski, T. Kociumaka, W. Rytter and T. Walen. Faster longest common extension queries in strings over general alphabets. In R. Grossi and M. Lewenstein, eds., *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, 27–29 June, 2016, Tel Aviv, Israel, vol. 54, LIPIcs, pp. 5:1–5:13. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

[129] P. Gawrychowski and P. Uznanski. Order-preserving pattern matching with k mismatches. In A. S. Kulikov, S. O. Kuznetsov and P. A. Pevzner, eds., *Combinatorial Pattern Matching: 25th Annual Symposium, CPM 2014*, Moscow, Russia, 16–18 June, 2014. Proceedings, vol. 8486, *Lecture Notes in Computer Science*, pp. 130–139. Springer, 2014.

[130] A. Glen, J. Justin, S. Widmer and L. Q. Zamboni. Palindromic richness. *Eur. J. Comb.*, 30(2):510–531, 2009.

[131] S. W. Golomb. *Shift Register Sequences* 3rd rev. ed. World Scientific, 2017.

[132] J. Grytczuk, K. Kosinski and M. Zmarz. How to play Thue games. *Theor. Comput. Sci.*, 582:83–88, 2015.

[133] C. Guo, J. Shallit and A. M. Shur. On the combinatorics of palindromes and antipalindromes. *CoRR*, abs/1503.09112, 2015.

[134] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
[135] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.
[136] C. Hancart. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2): 95–99, 1993.
[137] T. Harju and T. Kärki. On the number of frames in binary words. *Theor. Comput. Sci.*, 412(39):5276–5284, 2011.
[138] A. Hartman and M. Rodeh. Optimal parsing of strings. In A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, vol. 12, NATO ASI Series F: Computer and System Sciences, pp. 155–167, Springer, 1985.
[139] M. M. Hasan, A. S. M. S. Islam, M. S. Rahman and M. S. Rahman. Order preserving pattern matching revisited. *Pattern Recogn. Lett.*, 55:15–21, 2015.
[140] D. Hendrian, T. Takagi and S. Inenaga. Online algorithms for constructing linearsize suffix trie. *CoRR*, abs/1901.10045, 2019.
[141] D. Hickerson. There are at most 2*n* distinct twins in any finite string of length *n*. Personal communication by D. Gusfield, 2003.
[142] C. Hohlweg and C. Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003.
[143] R. Houston. Tackling the minimal superpermutation problem. *CoRR*, abs/1408.5108, 2014.
[144] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. I.R.E.*, 40:1098–1101, 1951.
[145] R. M. Idury and A. A. Schäffer. Multiple matching of parameterized patterns. *Theor. Comput. Sci.*, 154(2):203–224, 1996.
[146] L. Ilie. A simple proof that a word of length *n* has at most 2*n* distinct squares. *J. Comb. Theory, Ser. A*, 112(1):163–164, 2005.
[147] L. Ilie. A note on the number of squares in a word. *Theor. Comput. Sci.*, 380(3):373–376, 2007.
[148] L. Ilie and W. F. Smyth. Minimum unique substrings and maximum repeats. *Fundam. Inform.*, 110(1–4):183–195, 2011.
[149] C. S. Iliopoulos, D. Moore andW. F. Smyth. A characterization of the squares in a Fibonacci string. *Theor. Comput. Sci.*, 172(1–2):281–291, 1997.
[150] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri and G. Pavesi. On-line construction of compact directed acyclic word graphs. In A. Amir and G. M. Landau, eds., *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*, Jerusalem, Israel, 1–4 July 2001, Proceedings, vol. 2089, *Lecture Notes in Computer Science*, pp. 169–180. Springer, 2001.
[151] B. W. Jackson. Universal cycles of k-subsets and k-permutations. *Discrete Math.*, 117(1–3):141–150, 1993.
[152] N. Johnston. All minimal superpermutations on five symbols have been found. www.njohnston.ca/2014/08/all-minimal-superpermutations-on-five-symbol-shave-been-found/, 2014.
[153] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow and G. J. Woeginger, eds., *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, Eindhoven, The Netherlands, 30–4 June, 2003. Proceedings, vol. 2719, *Lecture Notes in Computer Science*, pp. 943–955. Springer, 2003.
[154] J. Kärkkäinen P. Sanders and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[155] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park. Linear-time longest- common-prefix computation in suffix arrays and its applications. In A. Amir and G. M. Landau, eds., *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*, Jerusalem, Israel, 1–4 July 2001, Proceedings vol. 2089, *Lecture Notes in Computer Science*, pp. 181–192. Springer, 2001.

[156] J. Katajainen, A. Moffat and A. Turpin. A fast and space-economical algorithm for length-limited coding. In J. Staples, P. Eades, N. Katoh and A. Moffat, eds., *Algorithms and Computation, 6th International Symposium, ISAAC '95*, Cairns, Australia, 4–6 December 1995, Proceedings vol. 1004, *Lecture Notes in Computer Science*, pp. 12–21. Springer, 1995.

[157] D. Kempa, A. Policriti, N. Prezza and E. Rotenberg. String attractors: Verification and optimization. *CoRR*, abs/1803.01695, 2018.

[158] A. J. Kfoury. A linear-time algorithm to decide whether A binary word contains an overlap. *ITA*, 22(2):135–145, 1988.

[159] D. K. Kim, J. S. Sim, H. Park and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2–4):126–142, 2005.

[160] J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi and T. Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014.

[161] D. E. Knuth. Dynamic Huffman coding. *J. Algorithms*, 6(2):163–180, 1985.

[162] D. E. Knuth, J. H. M. Jr. and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[163] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2–4):143–156, 2005.

[164] T. Kociumaka, J. Pachocki, J. Radoszewski, W. Rytter and T. Walen. Efficient counting of square substrings in a tree. *Theor. Comput. Sci.*, 544:60–73, 2014.

[165] T. Kociumaka, J. Radoszewski, W. Rytter, J. Straszynski, T. Walen and W. Zuba. Efficient representation and counting of antipower factors in words. In C. Martín-Vide, A. Okhotin and D. Shapira, eds., *Language and Automata Theory and Applications: 13th International Conference, LATA 2019*, St. Petersburg, Russia, 26–29 March, 2019, Proceedings, vol. 11417, *Lecture Notes in Computer Science*, pp. 421–433. Springer, 2019.

[166] W. Kolakoski. Problem 5304. *Am. Math. Mon.*, 72(674), 1965.

[167] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, 17–18 October 1999, New York, pp. 596–604. IEEE Computer Society, 1999.

[168] R. M. Kolpakov and G. Kucherov. Finding approximate repetitions under Hamming distance. In F. Meyer auf der Heide, ed., *Algorithms – ESA 2001, 9th Annual European Symposium*, Aarhus, Denmark, 28–31 August 2001, Proceedings, vol. 2161, *Lecture Notes in Computer Science*, pp. 170–181. Springer, 2001.

[169] K. Kosinski, R. Mercas and D. Nowotka. Corrigendum to 'a note on thue games' [*Inf. Process. Lett.* 118 (2017) 75–77]. *Inf. Process. Lett.*, 130:63–65, 2018.

[170] M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter and T. Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013.

[171] P. Kurka. *Topological and Symbolic Dynamics*. Société Mathématique de France, 2003.

[172] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *J. ACM*, 37(3):464–473, 1990.

[173] A. Lefebvre and T. Lecroq. Compror: On-line lossless data compression with a factor oracle. *Inf. Process. Lett.*, 83(1):1–6, 2002.

[174] A. Lempel. On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers. *IEEE Trans. Comput.*, 19(12):1204–1209, 1970.
[175] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983. Reprinted in 1997.
[176] M. Lothaire. *Algebraic Combinatorics on Words. Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 2002.
[177] M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, 2005.
[178] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.
[179] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for recognizing repetition. Report CS-79-056, Washington State University, Pullman, 1979.
[180] M. G. Main and R. J. Lorentz. Linear time recognition of square-free strings. In A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, vol. 12, *Series F: Computer and System Sciences*, pp. 271–278. Springer, 1985.
[181] G. S. Makanin. The problem of solvability of equations in a free semi-group. *Math. Sb.*, 103(2):147–236, 1977. In Russian. English translation in: *Math. USSR-Sb*, 32, 129–198, 1977.
[182] A. Mancheron and C. Moan. Combinatorial characterization of the language recognized by factor and suffix oracles. *Int. J. Found. Comput. Sci.*, 16(6):1179–1191, 2005.
[183] S. Mantaci, A. Restivo, G. Romana G. Rosone and M. Sciortino. String attractors and combinatorics on words. *CoRR*, abs/1907.04660, 2019.
[184] S. Mantaci, A. Restivo, G. Rosone and M. Sciortino. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014.
[185] S. Mantaci, A. Restivo and M. Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003.
[186] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.
[187] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
[188] J. Mendivelso and Y. Pinzón. Parameterized matching: Solutions and extensions. In J. Holub and J. Žd'árek, eds., *Proceedings of the Prague Stringology Conference 2015*, pp. 118–131, Czech Technical University in Prague, Czech Republic, 2015.
[189] T. Mieno, Y. Kuhara, T. Akagi, et al. Minimal unique substrings and minimal absent words in a sliding window, *International Conference on Current Trends in Theory and Practice of Informatics*, Springer, 2019.
[190] A. Moffat. Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, 38(11):1917–1921, 1990.
[191] S. P. Mohanty. Shortest string containing all permutations. *Discrete Math.*, 31:91–95, 1980.
[192] E. Moreno and D. Perrin. Corrigendum to 'on the theorem of fredricksen and maiorana about de bruijn sequences'. *Adv. Appl. Math.*, 33(2):413–415, 2004.
[193] G. Navarro and N. Prezza. Universal compressed text indexing. *Theor. Comput. Sci.*, 762:41–50, 2019.
[194] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical Online Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.
[195] J. Nilsson. Letter frequencies in the Kolakoski sequence. *Acta Phys. Pol. A*, 126(2):549–552, 2014.

[196] E. Ohlebusch. *Bioinformatics Algorithms*. Oldenbusch Verlag, 2013.
[197] R. Oldenburger. Exponent trajectories in symbolic dynamics. *Trans. AMS*, 46:453–466, 1939.
[198] T. Ota, H. Fukae and H. Morita. Dynamic construction of an antidictionary with linear complexity. *Theor. Comput. Sci.*, 526:108–119, 2014.
[199] T. Ota and H. Morita. On a universal antidictionary coding for stationary ergodic sources with finite alphabet. In *International Symposium on Information Theory and Its Applications, ISITA 2014*, Melbourne, Australia, 26–29 October 2014, pp. 294–298. IEEE, 2014.
[200] T. Ota and H. Morita. A compact tree representation of an antidictionary. *IEICE Trans.*, 100-A(9):1973–1984, 2017.
[201] J. Pansiot. Decidability of periodicity for infinite words. *ITA*, 20(1):43–46, 1986.
[202] N. Prezza. String attractors. *CoRR*, abs/1709.05314, 2017.
[203] S. J. Puglisi, J. Simpson and W. F. Smyth. How many runs can a string contain? *Theor. Comput. Sci.*, 401(1–3):165–171, 2008.
[204] J. Radoszewski and W. Rytter. On the structure of compacted subword graphs of Thue-Morse words and their applications. *J. Discrete Algorithms*, 11:15–24, 2012.
[205] N. Rampersad, J. Shallit and M. Wang. Avoiding large squares in infinite binary words. *Theor. Comput. Sci.*, 339(1):19–34, 2005.
[206] D. Repke and W. Rytter. On semi-perfect de Bruijn words. *Theor. Comput. Sci.*, 720:55–63, 2018.
[207] A. Restivo and S. Salemi. Overlap-free words on two symbols. In M. Nivat and D. Perrin, eds., *Automata on Infinite Words*, Ecole de Printemps d'Informatique Théorique, Le Mont Dore, 14–18 May, 1984, vol. 192, *Lecture Notes in Computer Science*, pp. 198–206. Springer, 1985.
[208] C. Reutenauer. *From Christoffel Words to Markov Numbers*. Oxford University Press, 2018.
[209] G. Rozenberg and A. Salomaa. *The Mathematical Theory of L Systems*. Academic Press, 1980.
[210] M. Rubinchik and A. M. Shur. *Eertree: An efficient data structure for processing palindromes in strings*. CoRR, abs/1506.04862, 2015.
[211] F. Ruskey and A. Williams. An explicit universal cycle for the (n – 1)-permutations of an n-set. *ACM Trans. Algorithms*, 6(3):45:1–45:12, 2010.
[212] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J. Comput.*, 9(3):509–512, 1980.
[213] W. Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theor. Comput. Sci.*, 363(2):211–223, 2006.
[214] W. Rytter. The number of runs in a string. *Informat. Comput.*, 205(9):1459–1469, 2007.
[215] W. Rytter. Two fast constructions of compact representations of binary words with given set of periods. *Theor. Comput. Sci.*, 656:180–187, 2016.
[216] W. Rytter. Computing the *k*-th letter of Fibonacci word. Personal communication, 2017.
[217] A. A. Sardinas and G. W. Patterson. A necessary and sufficient condition for the unique decomposition of coded messages. Research Division Report 50–27, Moore School of Electrical Engineering, University of Pennsylvania, 1950.
[218] J. Sawada and P. Hartman. Finding the largest fixed-density necklace and Lyndon word. *Inf. Process. Lett.*, 125:15–19, 2017.

[219] J. Sawada, A. Williams and D. Wong. A surprisingly simple de Bruijn sequence construction. *Discrete Math.*, 339(1):127–131, 2016.
[220] B. Schieber. Computing a minimum weight-link path in graphs with the concave Monge property. *J. Algorithms*, 29(2):204–222, 1998.
[221] M. Sciortino and L. Q. Zamboni. Suffix automata and standard sturmian words. In T. Harju, J. Karhumäki and A. Lepistö, eds., *Developments in Language Theory, 11th International Conference, DLT 2007*, Turku, Finland, 3–6 July, 2007, Proceedings, vol. 4588, *Lecture Notes in Computer Science*, pp. 382–398. Springer, 2007.
[222] J. Shallit. On the maximum number of distinct factors of a binary string. *Graphs Combinat.*, 9(2–4):197–200, 1993.
[223] Y. Shiloach. A fast equivalence-checking algorithm for circular lists. *Inf. Process. Lett.*, 8(5):236–238, 1979.
[224] R. M. Silva, D. Pratas, L. Castro, A. J. Pinho and P. J. S. G. Ferreira. Three minimal sequences found in ebola virus genomes and absent from human DNA. *Bioinformatics*, 31(15):2421–2425, 2015.
[225] I. Simon. String matching algorithms and automata. In R. Baeza-Yates and N. Ziviani, eds., *Proceedings of the 1st South American Workshop on String Processing*, pp. 151–157, Belo Horizonte, Brasil, 1993. Universidade Federal de Minas Gerais.
[226] S. S. Skiena. *The Algorithm Design Manual*. 2nd ed., Springer, 2008.
[227] T. Skolem. On certain distributions of integers in pairs with given differences. *Math. Scand.*, 5:57–68, 1957.
[228] B. Smyth. *Computing Patterns in Strings*. Pearson Education Limited, 2003.
[229] M. Szykula. Improving the upper bound on the length of the shortest reset word. In R. Niedermeier and B. Vallée, eds., *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, 28 February – 3 March 2018, Caen, France, vol. 96, LIPIcs, pp. 56:1–56:13. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018.
[230] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57:131–145, 1988.
[231] Z. Tronícek and B. Melichar. Directed acyclic subsequence graph. In J. Holub and M. Simánek, eds., *Proceedings of the Prague Stringology Club Workshop 1998*, Prague, Czech Republic, 3–4 September, 1998, pp. 107–118. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 1998.
[232] Z. Tronícek and A. Shinohara. The size of subsequence automaton. *Theor. Comput. Sci.*, 341(1–3):379–384, 2005.
[233] K. Tsuruta, S. Inenaga, H. Bannai and M. Takeda. Shortest unique substrings queries in optimal time. In V. Geffert, B. Preneel, B. Rovan, J. Stuller and A. M. Tjoa, eds., *SOFSEM 2014: Theory and Practice of Computer Science: 40th International Conference on Current Trends in Theory and Practice of Computer Science*, Nový Smokovec, Slovakia, 26–29 January 2014, Proceedings, vol. 8327, Lecture Notes in Computer Science, pp. 503–513. Springer, 2014.
[234] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
[235] J. van Leeuwen. On the construction of Huffman trees. In *ICALP*, pp. 382–410, 1976.
[236] J. S. Vitter. Design and analysis of dynamic Huffman codes. *J. ACM*, 34(4): 825–845, 1987.

[237] N. Vörös. On the complexity measures of symbol-sequences. In A. Iványi, ed., *Conference of Young Programmers and Mathematicians*, pp. 43–50, Budapest, 1984. Faculty of Sciences, Eötvös Loránd University.
[238] B. Walczak. A simple representation of subwords of the Fibonacci word. *Inf. Process. Lett.*, 110(21):956–960, 2010.
[239] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.
[240] W. A. Wythoff. A modification of the game of Nim. *Nieuw Arch. Wisk.*, 8:199–202, 1907/1909.
[241] I.-H. Yang, C.-P. Huang and K.-M. Chao. A fast algorithm for computing a longest increasing subsequence. *Inf. Process. Lett.*, 93(5):249–253, 2005.
[242] E. Zalinescu. Shorter strings containing all k-element permutations. *Inf. Process. Lett.*, 111(12):605–608, 2011.
[243] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Максим Крошемор, Тьерри Лекрок, Войцех Риттер

Алгоритмы обработки текста

125 задач с решениями

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 23,35. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Сопоставление строк – одна из самых старых тем в теории алгоритмов, но по-прежнему занимает важное место в информатике. За прошедшие 20 лет мы видели технологические прорывы в таких, например, приложениях, как информационный поиск и сжатие информации. Эта книга, представляющая собой богатое собрание задач и упражнений по важнейшим вопросам алгоритмов обработки текстов и комбинаторных свойств слов, предлагает приятный и прямой путь к их изучению и практическому освоению.
Задачи взяты из многочисленных публикаций – как уже ставших классическими, так и сравнительно новых. Начав с основ, авторы рассматривают все более сложные задачи по комбинаторным свойствам слов (включая слова Фибоначчи и Туэ–Морса), поиску строк в тексте (включая алгоритмы Кнута–Морриса–Пратта и Бойера–Мура), эффективным структурам данных для представления текстов (включая суффиксные деревья и суффиксные массивы) и сжатия текста (включая методы Хаффмана, Лемпеля–Зива и Барроуза–Уилера).

Издание будет полезно студентам, преподавателям, школьникам для подготовки к олимпиадам по информатике, а также широкому кругу разработчиков программного обеспечения.

Максим Крошемор – заслуженный профессор университета Гюстава Эйфеля и Королевского колледжа Лондона. Является почетным доктором Хельсинкского университета. Автор более 200 статей по алгоритмам на строках и их приложениям, а также соавтор нескольких книг по этой теме.

Тьерри Лекрок – профессор факультета информатики в университете Руана-Нормандии, Франция. В настоящее время возглавляет исследовательскую группу по обработке информации в биологии и здравоохранении, созданную на базе лаборатории компьютерных наук, обработки информации и систем. Более десяти лет являлся одним из координаторов рабочей группы по стрингологии во Французском национальном центре научных исследований.

Войцех Риттер – профессор факультета математики, информатики и механики Варшавского университета. Автор многочисленных публикаций по автоматам, формальным языкам, параллельным алгоритмам и алгоритмам обработки текста. Соавтор нескольких книг на эти темы, в т. ч. «Efficient Parallel Algorithms», «Text Algorithms» и «Analysis of Algorithms and Data Structures». Член Европейской академии.

ISBN 978-5-97060-952-1