



● РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

# АЛГОРИТМЫ ОБРАБОТКИ СТРОК

С. М. Окулов

# АЛГОРИТМЫ ОБРАБОТКИ СТРОК

4-е издание, электронное



Москва  
Лаборатория знаний  
2020

УДК 519.85(023)  
ББК 22.18  
0-52

*Серия основана в 2008 г.*

**Окулов С. М.**

0-52 Алгоритмы обработки строк / С. М. Окулов. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 258 с. — (Развитие интеллекта школьников). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-658-8

На материале задачи поиска подстроки в строке, решению которой посвящены работы многих профессионалов за последние 20–30 лет, показано, как построить занятия по информатике, чтобы побудить школьника к творчеству, развить у него вкус к решению исследовательских проблем.

Для школьников, преподавателей информатики, а также для студентов, выбравших информатику в качестве основной специальности. Книга может быть использована как в обычных школах при проведении факультативных занятий, так и в образовательных учреждениях с углубленным изучением информатики и математики.



УДК 519.85(023)  
ББК 22.18

**Деривативное издание на основе печатного аналога:** Алгоритмы обработки строк / С. М. Окулов. — 2-е изд. — М. : БИНОМ. Лаборатория знаний, 2015. — 255 с. : ил. — (Развитие интеллекта школьников).

ISBN 978-5-9963-1931-2.

**В соответствии со ст. 1299 и 1301 ГК РФ при устранении ограничений, установленных техническими средствами защиты авторских прав, правообладатель вправе требовать от нарушителя возмещения убытков или выплаты компенсации**

ISBN 978-5-00101-658-8

© Лаборатория знаний, 2015

---

---

# Оглавление

---

---

Предисловие .....	5
<b>Глава 1. Строки .....</b>	<b>9</b>
1.1. Основные понятия .....	9
1.2. Методы предварительного анализа строк .....	13
<b>Глава 2. Классические алгоритмы решения задач     обработки строк .....</b>	<b>28</b>
2.1. Алгоритм Д. Кнута – Дж. Морриса – В. Пратта ....	28
2.2. Алгоритм Р. Бойера – Дж. Мура .....	36
2.3. Алгоритм Р. Карпа – М. Рабина .....	52
2.4. Алгоритм Shift-And .....	57
2.5. Использование элементов теории автоматов в решении задач обработки строк .....	73
2.6. Алгоритм М. Крочемора .....	81
2.7. Алгоритм М. Мейна – Р. Лоренца .....	88
<b>Глава 3. Деревья суффиксов .....</b>	<b>103</b>
3.1. Основные понятия. Простые алгоритмы построения дерева суффиксов .....	103
3.2. Алгоритм Э. Укконена .....	118
3.3. Алгоритм Е. Мак-Крейга .....	127
3.4. Суффиксные массивы .....	136
3.5. Алгоритм А. Ахо – М. Корасик .....	147
<b>Глава 4. Вычисление расстояния между строками .....</b>	<b>155</b>
4.1. Основной алгоритм .....	155
4.2. Алгоритм Э. Укконена – Ю. Майерса .....	165
4.3. Задача о наибольшей общей подпоследовательности двух строк .....	174

<b>Глава 5. Алгоритмы приближенного поиска подстрок ..</b>	<b>198</b>
5.1. Простой алгоритм .....	198
5.2. Алгоритм С. Ву – Ю. Менбера .....	201
5.3. Задача о $k$ -несовпадениях .....	205
5.4. Алгоритм Ю. Майерса .....	215
Вместо заключения .....	225
Приложения .....	234



## Предисловие

---

Все должно быть изложено так просто, как только возможно, — но не проще.



Альберт Эйнштейн

Что может быть эффективнее для развития творческих возможностей школьника и его интеллекта, чем решение задач, казалось бы, очень простых, но «тянущих» за собой проблемы, исследованием которых занимались ведущие специалисты по информатике в последние 20–30 лет? Одной из таких задач является *задача поиска подстроки в строке*, которая так или иначе затрагивается в любом учебнике по информатике. Длительность ее решения с помощью самого простого алгоритма пропорциональна произведению длин строки и подстроки, и, несмотря на возросшую производительность компьютера, она оказывается слишком большой для многих приложений. Можно ли найти такие алгоритмы решения этой задачи, чтобы произведение заменялось хотя бы суммой? Оказывается, да, и эта замена является сутью работ лучших умов в информатике, многие из которых продолжают свою деятельность и в настоящее время.

Данная книга конструктивно построена в виде занятий, ее материал апробирован в ходе проведения реальных уроков<sup>1)</sup>. Особенность изложения этого материала заключается в том, что он не приводится в виде конечных результатов (лемм, теорем, фактов и т. д.). Напротив, автором сделана попытка показать сам процесс получения нового результата, а он не появляется сразу доказательно оформленным. Конеч-

<sup>1)</sup> Первая попытка изложения части предлагаемого здесь материала была сделана в 1997 г. (Бабушкина И. А., Бушмелева Н. А., Окулов С. М., Черных С. Ю. Конспекты занятий по информатике (практикум по Турбо Паскалю). — Киров: Изд-во ВГПУ, 1997).

ное оформление по принятым «правилам игры» при написании книг такого типа обычно скрывает способ его «рождения» — в каких «муках» и как он получен. «Ни один ученый не мыслит формулами», — любил говорить Альберт Эйнштейн. В основе любого алгоритма лежит какая-то образная картинка или ясная идея. Воссоздать эту картинку, взять на себя смелость утверждения о «рождении» результата именно так, как описывается в книге (через примеры, через эксперименты — а они обязательны — и ошибки с «набросками» кода) — страшно. Но если читатель подвергнет все сомнению, то автор будет считать, что он уже достиг цели, ибо специалист по информатике обязан все подвергать сомнению и ничего не принимать на веру! Такое сомнение в правильности результатов и самостоятельная работа приведут вас к моментам «Эврика!», к рождению нового, и, может быть, ваша фамилия так же войдет в историю информатики, как и фамилии авторов рассматриваемых здесь алгоритмов...

В предмете «информатика», а именно в олимпиадных соревнованиях по информатике, сложилась уникальная ситуация, которой нет ни в одном школьном предмете. Например, тематика заданий олимпиад по математике опирается на школьный курс, что вполне естественно. В олимпиадной же информатике (назовем ее так) существует как минимум три направления: первое, поддерживаемое и развиваемое международным сообществом, связано с алгоритмами и программированием; второе — это олимпиады по базовому курсу информатики; третье — различного рода соревнования по использованию информационных технологий. Связь первого направления со школьным курсом информатики ограничена несколькими разделами, второе направление полностью адекватно курсу, а третье — лишь частично. Для достижения результатов в рамках первого направления знать и уметь требуется много, очень много, и любой школьный учебник не входит в этот необходимый минимум. Если выразиться точнее, то победитель соревнований первого типа не всегда сможет выразить свои мысли тем языком, который задается учебниками, хотя, конечно, он знает их материал, но только на совершенно другом уровне понимания и осознания.

Рассматриваемые в книге алгоритмы (основные) входят в примерную программу по олимпиадной информатике всего одной строкой — *«алгоритмы поиска подстроки в строке*

за  $O(n+m)$ »<sup>1)</sup>. Эти алгоритмы относятся к дидактическим единицам, «изучение которых формирует у школьников ключевые умения в области олимпиадной подготовки, открывает перед участником олимпиадного состязания возможность проявить свой творческий потенциал на достойном уровне ... — победителей и призеров заключительных этапов Всероссийской олимпиады школьников»<sup>2)</sup>.

### Структура книги

Первая глава, с одной стороны, является вводной, а с другой — дает основы предварительного анализа строк, без которых понимание многих изложенных далее алгоритмов невозможно. Но, вероятно, главным в ней следует считать «очерчивание» одного из основных способов построения эффективных алгоритмов, который заключается в тщательном анализе исходных данных с целью выявления в них закономерностей, а затем — в использовании этих закономерностей при решении основной задачи.

Во второй главе рассматриваются ставшие уже классикой алгоритмы Д. Кнута – Дж. Морриса – В. Пратта; Р. Бойера – Дж. Мура; Р. Карпа – М. Рабина; *Shift-And* (Б. Дёмёлки – Р. Беза-Йетс – Г. Гоннет); М. Крочемора и М. Мейна – Р. Лоренца. Если первые четыре алгоритма посвящены проблеме поиска подстроки в строке, то последние два являются основополагающими в задаче анализа свойств строки (текста). Во второй главе кратко показано, как использовать аппарат теории автоматов при описании алгоритмов на строках.

Третья глава целиком посвящена *деревьям суффиксов* — структуре данных, в которой фиксируются особенности строки (текста), позволяющие эффективно решать многочисленные задачи обработки строк. Рассмотрены два алгоритма — Э. Укконена и Е. Мак-Крейга, однако их рассмотрению предшествует анализ простых методов построения дерева суффиксов, что позволяет сделать изложение доступным и ясным (с точки зрения автора). На остальные известные алгоритмы решения этой задачи в данной книге приводятся только ссылки.

В четвертой главе рассматриваются задачи вычисления расстояния между строками и нахождения наибольшей общей подпоследовательности двух строк. Анализ первой за-

<sup>1)</sup> Кирюхин В. М. Информатика: всероссийские олимпиады. — М.: Просвещение, 2008. С. 71.

<sup>2)</sup> Там же. С. 67.

дачи ограничивается основным алгоритмом и результатом Э. Укконена – Ю. Майерса. Вторая проблема анализируется через достижения С. Нудельмана – К. Вунша, Д. Ханта – Т. Зиманского и Л. Эллисона – Т. Дикса.

Пятая глава посвящена достаточно значимой задаче данной проблематики — приближенному поиску подстрок в тексте. Даны простые алгоритмы, а также алгоритм С. Ву – Ю. Менбера, алгоритм решения задачи о  $k$ -несовпадениях и идейные основы алгоритма Ю. Майерса.

Вместо заключения читателям предлагается небольшое эссе о предмете «информатика», из которого становятся яснее роль и место как материала данной книги, так и результата, получаемого в итоге деятельности по освоению рассматриваемых алгоритмов.

В приложениях раскрываются некоторые моменты организации углубленного экспериментального исследования алгоритмов и приводится ряд проблем (задач) для индивидуальной творческой работы<sup>1)</sup>. Рассмотренные в данной книге алгоритмы — это, если можно так выразиться, только первый «пласт» указанного раздела информатики. Проблемы, приведенные в приложении, лишь частично восполняют этот пробел, полностью устраняемый, вероятно, лишь последующими работами.

### Благодарности

Я благодарю коллег: Евгения Вячеславовича Котельникова, Андрея Васильевича Лялина и многих других за интеллектуальную помощь, без которой вряд ли состоялся бы этот труд. Особая признательность — Дмитрию Юрьевичу Усенкову, сотруднику издательства «БИНОМ. Лаборатория знаний», оперативность и доброжелательность работы которого вызывают восхищение. Глубочайшая благодарность — директору издательства Михаилу Николаевичу Бородину, который не только поверил в «пришедшего с улицы» автора (в 2001 г.), но и все эти годы профессионально поддерживает его деятельность.

<sup>1)</sup> На русском языке имеются только две фундаментальные переводные книги по данной проблематике: *Смит Б.* Методы и алгоритмы вычисления на строках: пер. с англ. — М.: ООО «И. Д. Вильямс», 2006 и *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология: пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. Автор, конечно же, использовал их (как и англоязычные статьи) при написании данной книги, которая, выражаясь педагогическим языком, является пропедевтикой к изучению названных книг, представляющих обширный материал по проблемам для индивидуальной творческой работы как школьника, так и студента.

И приходят мне в голову сказки  
Мудрецами отмеченных дней,  
И блуждаю я в них по указке  
Удивительной птицы моей.

*Николай Заболоцкий*

### 1.1. Основные понятия

Стену можно пробить только головой.  
Все остальное — только орудия.

*Лешек Кумор*

Обычно говорят, что *строка* ( $S$ ) — это последовательность символов, взятых из заранее определенного *алфавита*. При этом сразу возникают как минимум два вопроса, заключенные в понятиях «последовательность» и «алфавит».

*Последовательность* — это значит строка как одномерная структура, в которой каждый элемент (*символ*) имеет уникальную метку в виде номера, а рассматриваются только конечные элементы строки (первый и последний). Далее, каждый элемент строки, кроме первого (самого левого), имеет единственный предшествующий элемент, а каждый элемент, кроме последнего (самого правого), имеет единственный следующий элемент. Проводя аналогию с языком программирования Паскаль, можно сказать, что для строки работают операции  $\text{pred}(i)$  и  $\text{succ}(i)$ , где  $i$  — номер символа в строке, и справедливы равенства:  $i = \text{succ}(\text{pred}(i))$  (за исключением самого левого символа) и  $i = \text{pred}(\text{succ}(i))$  (за исключением самого правого символа).

*Алфавит* — это интуитивно достаточно ясное понятие; определим его как конечное множество  $A$  различных элементов, на котором определено *отношение порядка*. Традиционные примеры алфавитов: латинский; русский; все символы в соответствии с их кодировкой ASCII. Алфавит из двух символов (а компьютер работает с данными, представленными

ми в таком алфавите!) называют *бинарным (двоичным)*. Отношение порядка для символов из алфавита говорит о том, что для любых  $x \in A$  и  $y \in A$  можно сделать вывод, какой элемент больше другого, т. е. что  $x < y$  или  $y < x$  при  $x \neq y$ .

Понятие «подстрока» строки  $S$  определяется как  $S[i..j]$  для любой пары таких чисел  $i$  и  $j$ , что  $1 \leq i \leq j \leq n$ , где  $n$  — количество символов в  $S$  (*длина строки*, обозначим ее как  $|S|$ ). Если же  $i > j$ , то мы считаем подстроку  $S[i..j]$  *пустой*. Другими словами, подстрока — это часть строки, состоящая из некоторого количества смежных символов исходной строки, и в данном случае «некоторое количество» определяется как  $j - i + 1$ . Можно выделить точно  $n - k + 1$  подстрок длины  $k$  из строки  $S$  длины  $n$ : это подстроки  $S[1..k]$ ,  $S[2..k+1]$ , ...,  $S[n-k+1..n]$ . Общее количество подстрок с длинами от 1 до  $n$  определяется суммой —  $\sum_{k=1}^n (n-k+1) = \frac{n^2+n}{2}$ , т. е. имеет порядок  $O(n^2)$ .

Строки (естественно, на одном алфавите)  $S_1$  и  $S_2$  равны, если:

- 1) совпадают их длины ( $n$ );
- 2)  $S_1[i] = S_2[i]$  для всех  $i = 1, \dots, n$ .

На множестве строк на упорядоченном алфавите  $A$  отношение порядка (лексикографического) вводится естественным образом. Пусть имеется две строки  $S_1[1..n]$  и  $S_2[1..m]$ , тогда мы говорим, что  $S_1 < S_2$  ( $S_1$  лексикографически меньше  $S_2$ ), если выполняется одно из следующих условий (взаимоисключающих):

- а)  $n < m$  и  $S_1[1..n] = S_2[1..n]$ ;
- б) существует такое целое число  $i$  ( $i \in 1..\min\{n, m\}$ ), что  $S_1[1..i-1] = S_2[1..i-1]$  и  $S_1[i] < S_2[i]$  (или, другими словами,  $i$  — первая позиция слева, в которой элементы  $S_1$  и  $S_2$  различны).

### Примеры

$abc < abcd$  ( $n = 3, m = 4$ );  $abdef < ada$  ( $i = 2, b < d$ ).

*Префикс* строки  $S$ , заканчивающийся в позиции  $i$ , — это подстрока  $S[1..i]$ .

*Суффикс* строки  $S$ , начинающийся в позиции  $i$ , — это подстрока  $S[i..n]$ .

Префиксы и суффиксы называют *собственными*, если они не являются пустыми и не совпадают с  $S$ .

**Основная задача.** Дана строка  $P$  (ее чаще всего называют *образцом*) и строка  $T$  (*текст*). Требуется найти все вхождения образца  $P$  в текст  $T$ . Длины строк обозначим как  $m = |P|$  и  $n = |T|$ .

*Пример*

$P = aab$ ,  $T = aacbaabaatabaabaaw$ .  $P$  входит в  $T$ , начиная с позиций 5 и 13.

Простой, или «наивный», алгоритм решения задачи показан на рис. 1.1. Здесь найдено два вхождения (оба подчеркнуты) образца в текст.

```

123456789012345678
aacbaabaatabaabaaw
aab

```

Рис. 1.1. Пример простого поиска образца в тексте

Как нетрудно видеть, для решения поставленной задачи образец  $P$  «прикладывается» левым концом к тексту  $T$ , начиная с его первой позиции, а затем осуществляется посимвольное сравнение образца и соответствующих символов текста. При этом возможны два исхода: произойдет несовпадение символов в какой-то позиции либо будет найдено вхождение  $P$  в  $T$  (все символы  $P$  сравнены). Во втором случае фиксируется факт совпадения, но в обоих случаях затем  $P$  сдвигается на одну позицию вправо и процесс проверки совпадения повторяется. В данном алгоритме не учитываются как результаты предыдущих сравнений, так и структура образца или текста.

Формализованная запись рассмотренного алгоритма имеет следующий вид.

```

Procedure Solve;
{P, T - глобальные величины типа String}
  Var i, j:Integer;
  Begin
    For i:=1 To n-m+1 Do Begin
      j:=1;
      While (j<=m) And (P[j]=T[i+j-1]) Do j:=j+1;
      If j=m+1 Then WriteLn('найденно вхождение P в T,
                               начиная с позиции ', i);
    End;
  End;

```

Оценим время работы этого алгоритма в количестве операций сравнения. Оно очевидно и имеет значение  $O(n \cdot m)$ . Такая оценка достигается при  $n - m + 1$  совпадениях, т. е. когда и  $P$ , и  $T$  состоят из одного символа. При значениях  $m > 10^3$ ,  $n > 10^9$  время работы такого алгоритма становится неприемлемым для многих приложений.

Целью специалистов по информатике за последние 30 лет является разработка алгоритмов поиска вхождения образца в текст (иногда — удивительных и неожиданных, как сказка ☺), а также решение целого ряда родственных задач, с временной оценкой порядка  $O(n + m)$ . Большинство результатов при этом основано на предварительном анализе образца или текста (в зависимости от задачи), направленном на выявление, если можно так выразиться, его структуры, с последующим использованием этой информации для решения задачи. Разумеется, смысл имеют только алгоритмы с временной оценкой  $O(n)$  или  $O(m)$ . Скажем, такой анализ  $P$  позволил бы решить задачу из приведенного на рис. 1.1 примера не за 16 сдвигов, а за гораздо меньшее их число (например, 9).



### Упражнения

1. Сформулируйте отличие следующей реализации простого алгоритма поиска вхождения  $P$  в  $T$  от ранее приведенной.

```

Procedure Solve;
{P, T - глобальные величины типа String}
  Var i, j:Integer;
  Begin
    For i:=1 To n-m+1 Do Begin

```

```
j:=m;  
While (j>=1) And (P[j]=T[i+j-1]) Do j:=j-1;  
If j=0 Then WriteLn('найден входение P  
в T, начиная с позиции ', i);  
End;  
End;
```

2. В формализованной записи алгоритма простого поиска считалось, что как  $P$ , так и  $T$  имеют длины, допускающие использование типа данных `String`. Снимите это ограничение. Предположите, например, что  $m = 100$ , а  $n = 10000$ . Соответственно измените реализацию метода и оцените (экспериментально) время его работы. *Примечание.* Для этого потребуется дополнительно написать генератор случайных строк заданной длины.
3. Даны две строки —  $S_1$  и  $S_2$ . Определите, написав соответствующую программу, можно ли одну (любую) из этих строк получить циклическим сдвигом другой строки.

## 1.2. Методы предварительного анализа строк

Каждый из нас лишь выиграет, создавая время от времени «игрушечные» программы с заданными искусственными ограничениями, заставляющими нас до предела напрягать свои способности... Искусство решения мини-задач на пределе своих возможностей оттачивает наше умение для *реальных задач*.

Дональд Кнут

Методы предварительного анализа строк позволяют выявить их *структуру* — закономерности расположения символов в строке. Решение задачи за линейное время (от длины строки) создает предпосылки для нахождения эффективных алгоритмов поиска подстроки в строке. Известны две схемы предварительного анализа — нахождение *граней*<sup>1)</sup> и *блоков* строк<sup>2)</sup>. Рассмотрим их.

<sup>1)</sup> Смит Б. Методы и алгоритмы вычисления на строках: пер. с англ. — М.: ООО «И. Д. Вильямс», 2006.

<sup>2)</sup> Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология: пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003.

### 1.2.1. Грани строки

На грани марта и апреля,  
 границе дня и темноты —  
 С сознанием, что в конце тоннеля  
 на самом деле только ты.

Дмитрий Быков

#### Определение

*Гранью* (*border, verge, brink*) *br* строки *S* называется любой собственный префикс этой строки, равный суффиксу *S*.

Строка  $S = \text{abaababaabaab}$  имеет две грани (не пустые) — *ab* и *abaab*. Строка  $S = \text{abaabaab}$  также имеет две грани — *ab* и *abaab*, но вторая грань — перекрывающаяся. Строка длины  $n$  из повторяющегося символа, например  $\text{aaaaaaaa}$  (или  $a^8$ ), имеет  $n - 1$  грань. Для  $S = a^8$  это грани: *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, *aaaaaa* и *aaaaaaa*.

Понятие «собственный префикс» исключает грань, совпадающую с самой строкой.

*Длина грани* — это количество символов в ней.

Естественным обобщением понятия «грань» является понятие «*наибольшей грани*» — это наибольший (по количеству символов) собственный префикс строки, равный ее суффиксу.

Простым алгоритмом вычисления наибольшей грани строки *S* является последовательная проверка совпадения префиксов  $S[1]$ ,  $S[1..2]$ ,  $S[1..3]$ , ...,  $S[1..n - 1]$  с соответствующими суффиксами  $S[n]$ ,  $S[n - 1..n]$ ,  $S[n - 2..n]$ , ...,  $S[2..n]$ :

```
Function MaxBorder (S:String) :Word;
  Var i, j, br, n:Word;
  Begin
    n:=Length(S); {Вычисляем длину строки W}
    br:=0;
    For i:=1 To n-1 Do Begin {Цикл по длине грани}
      j:=n-i+1;
      While (j<=n) And (S[i+j-n]=S[j]) Do j:=j+1;
      {Префикс и суффикс длины i совпадают?}
      If j=n+1 Then br:=i;
    End;
    MaxBorder:=br;
  End;
```

Временная сложность этого алгоритма —  $O(n^2)$ .

Сделаем обобщение задачи. Пусть необходимо вычислить значения наибольших граней для всех подстрок  $S[1..i]$  ( $i = 1..n$ ) и сохранить их в массиве  $br[1..n]$ . Очевидно, значение  $br[1]$  равно 0, ибо подстрока  $S[1]$  не имеет собственных подстрок. Последовательное применение предыдущей логики к каждой подстроке приводит к следующему алгоритму с временной сложностью  $O(n^3)$ :

**Procedure** MaxBorderArray (S:String) ;

{Массив  $br$  — глобальный}

**Var** i,n,t:Word;

**Begin**

n:=Length(S) ;

br[1]:=0;

**For** i:=2 **To** n **Do**

br[i]:=MaxBorder(Copy(S,1,i)) ;

{ "Вырезаем" из  $S$  подстроку длины  $i$ ,  
начиная с первого символа }

**End;**



### Примеры

В табл. 1.1 приведены примеры вычисления массива граней  $br$  для различных строк  $S$ .



Таблица 1.1

№	$S$	$Br$
1	aaaaaa	0, 1, 2, 3, 4, 5
2	abcdef	0, 0, 0, 0, 0, 0
3	abaababaabaab	0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6, 4, 5
4	abcabcabcabc=(abc) <sup>4</sup>	0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
5	abcabdabcabeabcabd abcabc	0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 3

Какие наблюдения можно сделать на основании этих данных? Во-первых,  $br[i + 1] \leq br[i] + 1$  для любых  $i$  от 1 до  $n - 1$ , и если значения элементов  $br$  возрастают, то они возрастают с шагом 1. Во-вторых, когда  $br[i + 1] = br[i] + 1$ ? Ответ однозначен: при  $S[i + 1] = S[br[i] + 1]$ . Другими словами, если символ  $S$  в позиции  $i + 1$  совпадает с символом,

следующим за максимальным префиксом  $S[1..i]$ , совпадающим с суффиксом  $S[1..i]$ , то наибольшая грань  $br[i + 1]$  для  $S[1..i + 1]$  просто увеличивается на 1. А если  $S[i + 1] \neq S[br[i] + 1]$ ? Ситуация для строки  $S[1..12]$  третьего примера из табл. 1.1 поясняется на рис. 1.2а, а для строки  $S[1..24]$  пятого примера из табл. 1.1 — на рис. 1.2б.

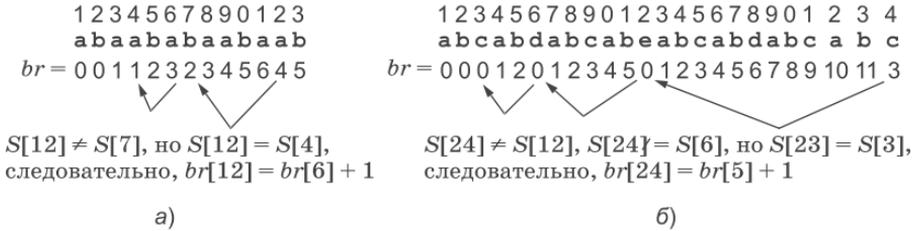


Рис. 1.2. Примеры вычислений значений  $br$  при  $S[i + 1] \neq S[br[i] + 1]$

Итак, если нет совпадения  $S[i + 1] \neq S[br[i] + 1]$ , то рассматривается наибольшая грань подстроки  $S[1..br[i]]$  и проверяется равенство  $S[i + 1] = S[br[br[i]] + 1]$ . Для примера на рис. 1.2а оно выполняется:

$$\begin{aligned}
 S[12] &\neq S[br[11] + 1] = S[7] \quad (a \neq b); \\
 S[12] &= S[br[br[11]] + 1] = S[br^2[11] + 1] = \\
 &= S[br[6] + 1] = S[3 + 1] = S[4] \quad (a = a) \\
 \text{и } br[12] &= br[6] + 1,
 \end{aligned}$$

а для примера на рис. 1.2б оно (на втором шаге) не выполняется, и требуется еще один шаг аналогичного «скачка» по строке.

В формировании массива граней ключевую роль играет факт, что если  $t$  (строка) — грань строки  $S$ , то  $t'$  (грань подстроки  $t$ ) является гранью строки  $S$ . Массив граней можно «развернуть» в двумерную структуру (ее часть для примера на рис. 1.2б представлена в табл. 1.2). Естественно, что на формирование значений двумерного массива потребуется время  $O(n^2)$ .

Если ввести обозначение  $br^j[i]$ ,  $j = 1, 2, \dots, k$ , являющееся значением длины  $j$ -й по величине грани подстроки  $S[1..i]$ , то  $S[1..br^j[i]]$  есть грань подстроки  $S[1..br^{j-1}[i]]$ . Другими словами,  $br^j[i] = br[br^{j-1}[i]]$ , или  $j$ -я по величине грань подстроки  $S[1..i]$ , является гранью  $(j - 1)$ -й по величине



Таблица 1.2

<i>I</i>	Длина граней
24	3, 0
23	11, 5, 2, 0
22	10, 4, 1, 0
21	9, 3, 0
20	8, 2, 0
...	...

грани подстроки  $S[1..i]$ , и существует наименьшее значение  $k$  такое, что  $br^k[i] = 0$ .

Формализованная запись алгоритма вычисления  $br$  для  $S$  может быть представлена так:

```

Procedure MaxBorderArray(S:String);
{Массив br – глобальный}
  Var i, n, t:Word;
  Begin
    n:=Length(S);
    br[1]:=0;
    For i:=1 To n-1 Do Begin
      t:=br[i];
      While (t>0) And (S[i+1]<>S[t+1]) Do t:=br[t];
      If S[i+1]=S[t+1] Then br[i+1]:=t+1
      Else br[i+1]:=0;
    End;
  End;

```

Оценим временные параметры этого алгоритма. Цикл **For** выполняется  $n - 1$  раз. Количество шагов вложенного цикла **While** различно. Время выполнения алгоритма пропорционально общему количеству присваиваний значений переменной  $t$ . Оно равно  $n - 1$  (в цикле **For**) плюс количество этих операций внутри цикла **While**. В цикле **While** происходит уменьшение значения переменной  $t$ . На каждой же итерации **For** значение  $t$  (а оно всегда неотрицательное) либо остается равным нулю, либо увеличивается на единицу. Таким образом, количество увеличений пропорционально  $(n - 1)$ , но общее количество уменьшений в цикле **While** (а оно всегда осуществляется не менее чем на 1) не может

превосходить количества увеличений. Следовательно,  $t$  изменяется внутри цикла While не более  $(n - 1)$  раз, так что полное количество присваиваний  $t$  ограничено сверху величиной  $2(n - 1) = O(n)$ . Итак, массив  $br$  для  $S$  формируется не за время  $O(n^3)$ , как было рассмотрено ранее, а за время  $O(n)$ .

Как можно видеть, в массиве  $br$  фиксируются грани всех подстрок  $S[1..i]$ ,  $i = 1..n$ , или, другими словами, вычисляются грани всех префиксов строки  $S$ . Аналогичную задачу можно решить и для суффиксов строки  $S$ , например в массиве  $bw$  сформировать грани  $S[i..n]$ ,  $i=1..n$ . На рис. 1.3 показано, в чем заключается отличие этого способа от предыдущего.

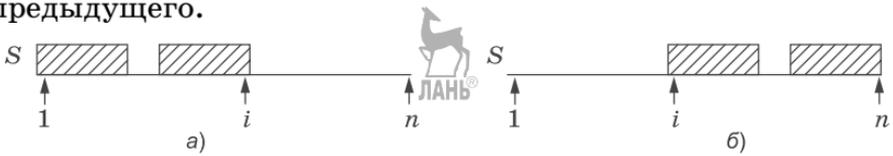


Рис. 1.3. Грани:

а) грань префикса  $S[1..i] - br[i]$ ; б) грань суффикса  $S[i..n] - bw[i]$

*Пример*

В табл. 1.3 показана строка  $S$  и массивы граней префиксов  $br$  и суффиксов  $bw$  для нее.

Таблица 1.3

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
<b>S</b>	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
<b>br</b>	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8
<b>bw</b>	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1	3	2	1	0	0

Логика формирования значений элементов массива  $bw$  аналогична ранее рассмотренной в процедуре MaxBorderArray и имеет вид:

```

Procedure BorderRigth(S:String);
  Var i,t:Word;
  Begin
    n:=Length(S);
    bw[n]:=0;
    For i:=n DownTo 2 Do Begin
      t:=bw[i];
      While (t>0) And (S[i-1]<>S[n-t]) Do t:=bw[n-t];
  
```

```

If S[i-1]=S[n-t] Then bw[i-1]:=t+1
      Else bw[i-1]:=0;
End;
End;

```

Обратимся теперь к основной задаче, сформулированной в п. 1.1, — к поиску вхождения образца  $P$  в текст  $T$ . Пусть имеется символ  $\$,$  не принадлежащий алфавиту  $A$ . Сформируем из  $P$  и  $T$  строку  $S = P\$T$  и вычислим  $br$  для  $S$ .

### Примеры

На рис. 1.4 приведены два примера такого решения задачи. В первой строке указаны номера позиции символа в  $S$ , во второй — строка  $S$ , а в третьей — значения элементов массива  $br$ .

1 2 3 4 5 6 7 8 9 0 1 2	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
aba\$abaabaab	ba\$abbabaabbaababba
$br = 001012312312$	$br = 0000112120112012112$

Рис. 1.4. Два примера вычисления  $br$  для строк типа  $S=P\$T$

Решение задачи достаточно очевидно. В сформированном массиве  $br$  требуется найти значения, равные длине  $P$ , а номер позиции  $i$  указывает на правый конец места вхождения  $P$  в  $T$ . Задача решается за время  $O(n + m)$ .

Итак, в массиве граней фиксируется некая информация о структурной организации  $S$ , позволяющая решать исходную задачу более эффективно, т. е. за меньшее время. Подчеркнем следующую мысль об этом приеме (или методе) информатики: *введение дополнительных структур данных, эффективно формируемых и либо описывающих исходную задачу несколько иначе, либо структурирующих информацию об исходных данных, — это путь к нахождению более быстродействующих методов решения задач.*

## 1.2.2. Блоки строки



Сложность — это сумма простых трудностей.

Георгий Александров

### Определение

**Блоком** (*bloc*) строки  $S$  в позиции  $i$  ( $bl[i]$ ) назовем длину наибольшей подстроки  $S$ , которая начинается в  $i$  и совпадает с префиксом  $S$ .

Для хранения блоков строки  $S$  наиболее естественной структурой данных является массив. Определим его как массив блоков ( $bl$ ).

### Примеры



На рис. 1.5 приведено восемь примеров вычисления массива блоков строк.

- |    |   |    |  |
|----|---|----|--|
| 1) | 123456789012<br>$S = \text{abcabdabcabd}$<br>$bl = 000200600200$    | 2) | 12345678901<br>$S = \text{aabcaabyaaz}$<br>$bl = 01003100210$                |
| 3) | 12345678<br>$S = \text{abababab}$<br>$bl = 00604020$                | 4) | 12345678<br>$S = \text{abaabaab}$<br>$bl = 00150120$                         |
| 5) | 1234567890123<br>$S = \text{abaababaabaab}$<br>$bl = 0013060150120$ | 6) | 123456789<br>$S = \text{aaabaabab}$<br>$bl = 021021010$                      |
| 7) | 12345678901<br>$S = \text{abcabcabcab}$<br>$bl = 00080050020$       | 8) | 1234567890123456<br>$S = \text{abbabaabbaababba}$<br>$bl = 0002014001204001$ |

Рис. 1.5. Примеры вычисления  $bl$  для различных строк  $S$

Формализованная запись алгоритма вычисления блоков строки с временной сложностью  $O(n^2)$  имеет вид:

```

Procedure Bloc( $S$ :String); {Массив  $bl$  - глобальный}
  Var  $i, j, n$ :Word;
  Begin
     $n := \text{Length}(S)$ ;
     $bl[1] := 0$ ;
    For  $i := 2$  To  $n$  Do Begin
       $j := i$ ;
      While ( $j \leq n$ ) And ( $S[j-i+1] = S[j]$ ) Do  $j := j+1$ ;
       $bl[i] := j-i$ ;
    End;
  End;

```



Поставим вопрос: можно ли вычислять блоки строки алгоритмом с линейной временной сложностью  $O(n)$ ? Вероятно, единственный возможный путь — это учет при вычислении очередного блока информации о ранее вычисленных блоках. В приведенном выше алгоритме этого не делается, и каждый блок определяется как бы с нулевой отметки.

Введем функцию сравнения (*comparison*) подстрок строки, которая получает номера первых символов подстрок ( $p_1$  и  $p_2$ ) и определяет длину (количество символов) совпадающей части:

**Function** Cmp ( $p_1, p_2$ :Word) :Word;

{Символы строки  $S$  сравниваются, начиная с позиций  $p_1$  и  $p_2$ :  $S[p_1]=S[p_2]$ ,  $S[p_1+1]=S[p_2+1]$ , ..., до первого неравенства символов или до достижения конца строки}

**Var** j, t:Word;

**Begin**

**If** ( $p_1 > n$ ) **Or** ( $p_2 > n$ ) **Then** Cmp:=0

**Else Begin**

**If**  $n-p_1 < n-p_2$  **Then**  $t:=n-p_1$  **Else**  $t:=n-p_2$ ;

{Определяем минимальное значение до конца строки}

$j:=0$ ;

**While** ( $j \leq t$ ) **And** ( $S[p_1+j]=S[p_2+j]$ ) **Do**  $j:=j+1$ ;

Cmp:=j;

**End;**

**End;**

При наличии этой функции предыдущий алгоритм записывается в следующем виде:

**Procedure** Bloc ( $S$ :String); {Массив  $bl$  - глобальный}

**Var** i, n:Word;

**Begin**

$n:=\text{Length}(S)$ ;

$bl[1]:=0$ ;

**For**  $i:=2$  **To**  $n$  **Do**

$bl[i]:=Cmp(1, i)$ ;

**End;**

Правда, для решения задачи нами пока ничего не сделано, а проведена чисто техническая правка алгоритма. Поэтому продолжим.

Для любой позиции  $i > 1$  значение  $bl[i] > 0$  определяет правую границу  $i + bl[i] - 1$  блока, совпадающего с префиксом  $S$ , начинающегося в  $i$  и заканчивающегося в  $i + bl[i] - 1$ . Но позиция  $i$  строки может «покрываться» несколькими блоками, поэтому вполне естественно будет определить самое правое (наибольшее) значение  $r[i] = \text{Max}(j + bl[j] - 1)$  по всем  $1 < j \leq i$ . В качестве левой гра-

ницы  $l[i]$  мы возьмем левую границу любого блока, содержащего позицию  $i$ .

### Пример

В табл. 1.4 приведены строка  $S$  и вычисленные для нее значения  $bl[i]$ , а также  $r[i]$  и  $l[i]$ .

Таблица 1.4

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$S$	a	b	c	a	b	d	a	b	c	a	b	e	a	b	c	a	b	d	a	b	c	a	b	c
$bl$	0	0	0	2	0	0	5	0	0	2	0	0	11	0	0	2	0	0	5	0	0	3	0	0
$r$	0	0	0	5	5	0	11	11	11	11	11	0	23	23	23	23	23	23	23	23	23	24	24	24
$l$	0	0	0	4	4	0	7	7	7	10	10	0	13	13	13	13	13	13	19	19	19	22	22	22

Пусть вычисляется  $bl[i]$ , и позиция  $i$  не входит ни в один ранее вычисленный блок (фактически нам потребуются только последние значения  $r$  и  $l$ , так что в использовании массивов нет необходимости).

Пусть  $i > r$ . В этом случае прямым сравнением до несовпадения подстрок, начинающихся с позиции 1 и с позиции  $i$  (или до исчерпания строки), определяется значение  $bl[i]$ . Тогда  $r = i + bl[i] - 1$  и  $l = i$ .

Рассмотрим теперь второй случай:  $i \leq r$ , т. е. позиция  $i$  находится в блоке. О чем это говорит? Такая ситуация показана на рис. 1.6, и из нее следует, что принадлежность  $S[i]$  блоку дает основание утверждать о наличии префикса  $S$ , совпадающего с данным блоком (на рис. 1.6 он выделен пунктирной линией). А значит, в позиции  $k = i - l + 1$  находится символ  $S[k]$ , равный  $S[i]$ . Более того, подстрока  $S[i..r]$  должна совпадать с подстрокой  $S[k..bl[i]]$ . Но значение  $bl[k]$  уже вычислено, и известно, с какой частью префикса  $S$  совпадает подстрока, начинающаяся с позиции  $k$ . Тогда получается, что у нас есть все основания утверждать о совпадении подстроки, начинающейся с позиции  $i$ , и префикса  $S$  длиной, равной минимуму значений  $bl[k]$  и  $r - i + 1$ .



Рис. 1.6. Позиция  $i$  находится в блоке  $bl[i]$  с границами  $[l, r]$

Логически при нахождении  $i$  в блоке ненулевой длины возможны следующие ситуации.

Первая — когда  $bl[k] < r - i + 1$ , т. е. длина префикса  $S$ , совпадающего с подстрокой, начинающейся с позиции  $k$ , меньше, чем «остаток» блока  $bl[i]$ . Тогда  $bl[i] = bl[k]$  и значения  $l, r$  не изменяются.

Вторая ситуация — при  $bl[k] \geq r - i + 1$  — более интересна. Имеет место полное совпадение подстрок  $S[k..bl[i]]$  и  $S[i..r]$ , но оно может быть продолжено, начиная с позиций  $r + 1$  и  $bl[i] + 1$ . Следовательно, необходимо непосредственным сравнением проверить совпадение подстрок, начинающихся с этих позиций. И если при этом произойдет  $q$  совпадений, то  $bl[i] = r - i + 1 + q$ ,  $r = r + q$ ,  $l = i$ .

В формализованном виде этот алгоритм можно представить в следующем виде:



```

Procedure Bloc (S:String) ;
{Массив bl - глобальный}
  Var n, r, l, i, k, q:Word;
  Begin
    n:=Length (S) ;
    r:=0;
    l:=0;
    bl[1]:=0;
  For i:=2 To n Do Begin
    bl[i]:=0;
    If i>r Then Begin
      bl[i]:=Cmp (l, i) ;
      {Вычисляем длину совпадения подстрок
      и корректируем, если есть совпадение,
      значения l и r}
      If bl[i]>0 Then Begin
        r:=i+bl[i]-1;
        l:=i;
      End
    End
    Else Begin {Позиция i входит в блок}
      k:=i-1+1;
      If bl[k]<r-i+1 Then bl[i]:=bl[k]
      Else Begin

```

```

{Размер блока в позиции  $k$  превышает
"остаток" блока, "покрывающего" позицию  $i$ }
  bl[i]:=r-i+1;
  l:=i;
  q:=Cmp(r-i+2,r+1);
  If q>0 Then Begin
    {Есть дальнейшее совпадение - уточняем
    размер блока и его правую границу}
    bl[i]:=bl[i]+q;
    r:=i+bl[i]-1;
  End;
End;
End;
End;
End;
End;

```

Время работы этого алгоритма вычисления блоков равно  $O(n)$ . При поверхностном анализе нетрудно увидеть, что количество итераций (цикл For) пропорционально  $n$ , и есть внутренний цикл While при реализации сравнения подстрок, т. е. временная оценка должна была бы равняться  $O(n^2)$ . Но цикл While заканчивается при первом несовпадении символов. Если допустить одни несовпадения (когда вся строка состоит из различных символов), то количество несовпадений не превышает значения  $n - 1$ . Разберемся теперь с совпадением символов. Следует понять, что каждый символ строки участвует в совпадении один раз, т. е. повторных проверок, несмотря на вложенный цикл While, нет. Верхняя граница  $r$  — это количество проверенных (совпавших) символов. Ее значение не уменьшается, а только увеличивается от итерации к итерации:  $r_i \geq r_{i-1}$  для любого значения  $i$ . Пусть на очередной итерации  $i$  символ  $S[i]$  находится в блоке, и выполнено  $q$  сравнений, тогда  $r_i$  увеличивается на значение  $q$ . Однако верхняя граница блоков не превосходит  $n$ , следовательно, общее количество совпадений (на всех итерациях) также не превосходит  $n$ . Таким образом, количество совпадений и несовпадений имеет оценку  $O(n)$ .

Обратимся в очередной раз к основной задаче, сформулированной в п. 1.1, — к поиску вхождения образца  $P$  в текст  $T$ . Алгоритм ее решения с использованием массива граней был рассмотрен ранее. Можно ли использовать для этой цели массив блоков?

Пусть имеется символ  $\$,$  не принадлежащий алфавиту  $A.$  Сформируем из  $P$  и  $T$  строку  $S = P\$,T$  и вычислим  $bl$  для  $S.$

### Примеры

На рис. 1.7 приведены два примера. В первой строке указаны номера позиции символа в  $S,$  во второй — строка  $S,$  а в третьей — значения элементов массива  $bl.$

1 2 3 4 5 6 7 8 9 0 1 2	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
aba\$abaabaab	ba\$abbabaabbaababba
$bl = 001030130120$	$bl = 0000120200120020120$

Рис. 1.7. Два примера вычисления  $bl$  для строк типа  $S=P\$,T$

Решение этой задачи достаточно очевидно. В сформированном массиве  $bl$  требуется найти значения, равные длине  $P.$  Номер позиции  $i$  ( $i > n$ ) при этом указывает на левый конец места вхождения  $P$  в  $T.$  Задача решается за время  $O(n + m).$

*Примечание.* Казалось бы, зачем нам продолжать рассматривать проблему, если для нее уже есть два простых варианта решения? Однако задачи многочисленных приложений, сводящихся к обработке строк, не исчерпываются этой простой схемой! Например, при поиске в реальном времени или оперативной обработке данных (в режиме on-line), когда действия обязаны выполняться по мере поступления информации, возвращаться к повторным операциям с отдельным символом текста (или, в зависимости от задачи, образца) не представляется возможным.



### Упражнения

1. В алгоритме нахождения наибольшей грани строки  $S$  с временной сложностью  $O(n^2)$  поиск начинался с наименьшего возможного значения грани. Измените этот алгоритм так, чтобы поиск начинался с максимально возможного значения и при нахождении первого совпадения завершал работу.
2. Вычислите массив граней префиксов и суффиксов для строк:
  - ab;
  - abba;
  - abbabaab;



### Методический комментарий

При рассмотрении задачи точного поиска образца в известных автору книгах<sup>1)</sup> на русском языке обычно используется один из методов предварительного анализа образца. Рассмотрение обоих методов и их использование при обсуждении особенностей алгоритмов не нарушает целостности изложения; наоборот, такой подход позволяет в очередной раз подчеркнуть основное положение процесса получения эффективных алгоритмов — «из ничего не рождается нечто». Другими словами, для достижения результата требуется выявить закономерности в исходных данных и представить их в новых структурах данных, использование которых приводит к эффекту (по затратам времени либо по расходу ресурса памяти) при реализации основной обработки (логики алгоритма).



---

<sup>1)</sup> *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003; *Смит Б.* Методы и алгоритмы вычислений на строках. — М.: ООО «И. Д. Вильямс», 2005.

# Классические алгоритмы решения задач обработки строк

---

Нам выпало — стремиться за пределы.  
Зря, что ли, существуют небеса?!

*Роберт Браунинг*



## 2.1. Алгоритм Д. Кнута – Дж. Морриса – В. Пратта

Храни в памяти имена великих  
людей и в своих походах и действиях  
с благоразумием следуй их примеру.

*Александр Суворов*

Это — один из самых известных алгоритмов решения задачи поиска образца  $P$  в тексте  $T$ , имеющий временную оценку  $O(n)$ , т. е. в нем поиск образца осуществляется за время, пропорциональное длине текста. В какой-то мере этот результат можно считать «точкой отсчета» в стремлении специалистов по информатике создать новые алгоритмы решения данной классической задачи.

Здесь образец, как и в простом алгоритме, последовательно «прикладывается» к тексту и осуществляется пошаговое сравнение символов. Но если в простом алгоритме после несовпадения в какой-то позиции осуществляется сдвиг на одну позицию, то в рассматриваемом, за счет предварительного анализа  $P$ , сдвиг выполняется в некоторых случаях более чем на один символ.

Пусть вычислен массив граней  $P$  (см. п. 1.2), или, другими словами, для каждой позиции  $i$  в  $P$  определена  $br[i]$  — длина наибольшего собственного суффикса  $P[1..i]$ , совпадающего с префиксом  $P$ . В табл. 2.1 дан пример  $P$  и его массива граней  $br$ .

Таблица 2.1

<i>i</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
<i>P</i>	a	b	c	a	e	a	b	c	a	b	c	a					
<i>br</i>	0	0	0	1	0	1	2	3	4	2	3	4					
						a	b	c	a	e	a	b	c	a	b	c	a

Предположим, что произошло несовпадение символов при  $i = 9$  и при некоторой позиции  $k$  текста  $T$ . Значение  $br[8]$  говорит о наличии суффикса длиной 3, совпадающего с префиксом  $P$ . Значит, чтобы этот префикс совпал с суффиксом, следует сдвинуть  $P$  на  $(8 - 3) = 5$  позиций. При этом гарантируется совпадение  $br[8]$  (т. е. трех) символов  $P$  с соответствующими символами  $T$ , так что следующее сравнение следует выполнять между символами  $T[k]$  и  $P[br[8] + 1]$ .

В этом заключается вся суть рассматриваемого алгоритма: благодаря знанию структуры образца  $P$ , анализ которой выполняется за линейное время, можно выполнять сдвиг при поиске вхождения  $P$  в  $T$  более чем на одну позицию. При этом символ  $T[k]$  участвует в сравнении как минимум два раза.

Пример

Пусть требуется в  $T = ababcxabdababcxabcabcde$  найти вхождения  $P = abcxabcde$ . Массив граней  $br = (0, 0, 0, 0, 1, 2, 3, 0, 0)$ . Процесс поиска представлен в табл. 2.2.

Таблица 2.2

<i>i</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
<i>T</i>	a	b	a	b	c	x	a	b	d	a	b	c	x	a	b	c	x	a	b	c	d	e
<i>P</i>	a	b	c	x	a	b	c	d	e													
			a	b	c	x	a	b	c	d	e											
							a	b	c	x	a	b	c	d	e							
									a	b	c	x	a	b	c	d	e					
												a	b	c	x	a	b	c	d	e		

После первого «прикладывания» происходит два совпадения символов, затем в третьей позиции — несовпадение ( $a \neq c$ ). Так как  $br[3] = 0$ , то осуществляется сдвиг на  $(2 - 0) = 2$  позиции. При втором «прикладывании» несов-

падение происходит в 7-й позиции  $P$ , тогда значение нового сдвига определяется как  $(6 - br[6]) = (6 - 2) = 4$ . Действительно, у подстроки  $P[1..6]$  есть суффикс длины 2, совпадающий с префиксом  $P$ , и этот префикс следует «приложить» к части текста  $T[7..8]$ . Затем осуществляется сравнение символов  $T[9]$  (повторное) и  $P[3]$ . Имеет место несовпадение и сдвиг на  $(2 - br[2]) = (2 - 0) = 2$  позиции вправо. Теперь символ  $T[9]$  сравнивается с  $P[1]$  и выполняется сдвиг на одну позицию. Последовательное сравнение символов  $T$  и  $P$  при данном «прикладывании» достигает 8-й позиции. Значение  $br[7]$  равно 3, что говорит о наличии суффикса из трех символов, совпадающего с префиксом  $P$ , и этот префикс следует «разместить» под соответствующими символами  $T[14..16]$ , а для этого требуется выполнить сдвиг на  $(7 - 3) = 4$  позиции. Далее находится вхождение  $P$  в  $T$ , и следует заметить, что если бы текст  $T$  продолжался, то следующий сдвиг (после фиксации вхождения) выполнялся бы на всю длину  $P$ , ибо  $br[9] = 0$ .

*Примечание.* Мы здесь говорим о «сдвигах» и «прикладываниях», но надо понимать, что фактически нет никаких реальных сдвигов и прикладываний строк! Изменяются лишь индексы соответствующих элементов  $T$  и  $P$ , поскольку предполагается, что  $T$  и  $P$  представлены в памяти компьютера как массивы символьного типа. (При другом способе представления  $T$  и  $P$  изменяется схема адресации к символам, но суть от этого не меняется.)

### Пример 1

Пусть  $T = \text{aaaaaaaaaaaa}$  и  $P = \text{aaaaa}$ . Очевидно, что  $P$  входит в  $T$   $(n - m + 1) = (12 - 5 + 1) = 8$  раз. Подсчитаем количество сравнений. При простом методе мы имеем  $(n - m + 1) \cdot m = 40$  сравнений. Массив граней  $br$  для  $P$  равен  $(0, 1, 2, 3, 4)$ . Схема поиска вхождений по алгоритму Д. Кнута – Дж. Морриса – В. Пратта показана в табл. 2.3.

После нахождения первого вхождения вычисляем значение сдвига. Оно равно  $(5 - br[5]) = (5 - 4) = 1$ . Другими словами, мы сдвигаем подстроку при обнаружении каждого вхождения на одну позицию. Но сравниваем мы после сдвигов только последний символ  $P[5]$  с очередным символом  $T$ :  $P[5]$  с  $T[6]$ ;  $P[5]$  с  $T[7]$ ;  $P[5]$  с  $T[8]$  и т. д. Таким образом, для поиска всех вхождений требуется лишь  $n$  сравнений, в данном случае — 12, а не 40.

Таблица 2.3

<i>i</i>	1	2	3	4	5	6	7	8	9	10	11	12
<b>T</b>	a	a	a	a	a	a	a	a	a	a	a	a
<b>P</b>	a	a	a	a	a							
		a	a	a	a	a						
			a	a	a	a	a					
				a	a	a	a	a				
					a	a	a	a	a			
						a	a	a	a	a		
							a	a	a	a	a	
								a	a	a	a	a

**Пример 2**

Пусть  $T = \text{aaaaaaaaaaaa}$  и  $P = \text{aaaab}$ . Как нетрудно видеть,  $P$  не входит в  $T$ . Простым алгоритмом этот факт устанавливается за 40 сравнений. Массив граней для  $P$  равен (0, 1, 2, 3, 0). Схема поиска вхождения  $P$  в  $T$  по алгоритму Д. Кнута – Дж. Морриса – В. Пратта приведена в табл. 2.4.

Таблица 2.4

<i>i</i>	1	2	3	4	5	6	7	8	9	10	11	12
<b>T</b>	a	a	a	a	a	a	a	a	a	a	a	a
<b>P</b>	a	a	a	a	b							
		a	a	a	a	b						
			a	a	a	a	b					
				a	a	a	a	b				
					a	a	a	a	b			
						a	a	a	a	b		
							a	a	a	a	b	
								a	a	a	a	b

Так как несовпадения происходят в позиции  $P[5]$ , то величина сдвига всегда равна  $(4 - br[4]) = (4 - 3) = 1$ . В каждом случае символы  $T[5]$ ,  $T[6]$ , ...,  $T[11]$  повторно сравниваются с символом  $P[4]$ , т. е. дважды участвуют в сравнениях. Общее количество сравнений равно  $(m - 1 + 2 \cdot (n - m + 1) - 1) = (2n - m)$ . Последнее вычитание единицы обусловлено тем, что символ  $T[n]$  сравнивается только один раз. Для рассматриваемого примера количество сравнений равно 19, а не 40, как при простом алгоритме (см. п. 1.2).

Формализованная запись алгоритма Д. Кнута – Дж. Морриса – В. Пратта поиска  $P$  в  $T$  имеет вид:

```

Procedure KMP( $T, P$ :String);
  Var  $n, m, i, q$ :Word;
  Begin
     $n$ :=Length( $T$ );
     $m$ :=Length( $P$ );
    MaxBorderArray( $P$ );
    {Вычисляем значения элементов массива границ  $br$ 
      (п. 1.2)}
     $q$ :=0;
    {Индекс сравниваемого символа образца  $P$ }
    For  $i$ :=1 To  $n$  Do Begin
      {Индекс символа текста  $T$ }
      While ( $q > 0$ ) And ( $P[q+1] \neq T[i]$ ) Do  $q$ := $br[q]$ ;
      {Имитация сдвига}
      If  $P[q+1]=T[i]$  Then  $q$ := $q+1$ ;
      If  $q=m$  Then Begin
        WriteLn('Найдено вхождение  $P$  в  $T$  с позиции ',
           $i-m+1$ );
         $q$ := $br[m]$ ;
      End;
    End;
  End;

```

В примере, представленном в табл. 2.2, жирным шрифтом был выделен случай, дающий основания для дальнейшего усовершенствования алгоритма. В этот момент символ  $T[9]$  сравнивается с  $P[7]$  ( $i = 9, q = 6$ ). Работает цикл **While** (см. соответствующую формализованную запись), и значение  $q$  становится равным 2 ( $br[6] = 2$ ). Происходит сравнение  $T[9]$  с  $P[3]$ . Результат заведомо известен, ибо  $P[3] = P[7]$ , и уже было зафиксировано несовпадение символа  $P[7]$  с  $T[9]$ . Затем  $q$  присваивается значение 0 ( $br[2] = 0$ ) и осуществляется сравнение  $T[9]$  с  $P[1]$ , но это уже следующая строка таблицы! Можно ли исключить эти лишние сдвиги?

Уточним понятие грани. Для каждой позиции  $i$  строки  $S$  определим  $brs[i]$  как длину наибольшего собственного суффикса  $S[1..i]$ , совпадающего с префиксом  $S$  и такого, что  $S[i+1] \neq S[brs[i]+1]$ . Другими словами, следующий сим-

вол за префиксом, равным суффиксу, не должен совпадать с символом  $S[i + 1]$ .

В табл. 2.5 приведены три примера вычисления значений уточненных граней. (Конечно, значения элементов массива *brs* получены на основе вычисленного ранее массива *br*.)

Таблица 2.5

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
<i>S</i>	a	b	c	x	a	b	c	d	e												
<i>br</i>	0	0	0	0	1	2	3	0	0												
<i>brs</i>	0	0	0	0	0	0	3	0	0												
<i>S</i>	a	b	a	a	b	a	b	a	a	b	a	a	b								
<i>br</i>	0	0	1	1	2	3	2	3	4	5	6	4	5								
<i>brs</i>	0	0	1	0	0	3	0	1	0	0	6	0	5								
<i>S</i>	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
<i>br</i>	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8
<i>brs</i>	0	0	1	0	0	3	0	1	0	0	6	0	0	3	0	1	0	0	11	0	8

*Примечание.* Значение  $brs[n]$  получено при условии, что к *S* приписывается символ, которого нет в алфавите.

Если в алгоритме Д. Кнута – Дж. Морриса – В. Пратта для определения величин сдвигов использовать не *br*, а *brs*, то сдвиги, о которых шла речь выше, исключаются. В рассматриваемом примере (см. табл. 2.2)  $brs[6] = 0$ , и сдвиг, выделенный жирным шрифтом в табл. 2.2, уже не выполняется.

Получение *brs* из *br* требует линейного времени  $O(n)$  и реализуется следующим образом:

```

Procedure Brst (S:String) ;
  Var i, n:Word;
  Begin
    n:=Length(S) ;
    brs[1]:=0;
    For i:=2 To n Do
      If S[br[i]+1]<>S[i+1] Then brs[i]:=br[i]
      Else brs[i]:=brs[br[i]];
      {Символы совпадают - берем ранее
      сформированное значение}
  End;

```

В случае использования как  $br$ , так и  $brs$  при несовпадении в позиции  $q + 1$  образца  $P$  и позиции  $i$  текста  $T$  при сдвиге  $P$  на  $q - br[q]$  вправо левые  $br[q]$  символов  $P$  совпадут с соответствующими символами в  $T$ . Следующее сравнение следует выполнять между символами  $T[i]$  и  $P[br[q] + 1]$ . Для полноты обоснования нужно также показать, что образец  $P$  не сдвигается слишком далеко, или, другими словами, что не будут пропущены вхождения  $P$  в  $T$ . Это интуитивно ясное утверждение (оно следует из определения массива граней и логики сдвига) доказывается методом «от противного»: предполагается, что пропущено такое вхождение и находится грань, превосходящая по длине значение  $br[q]$ .

Алгоритм Д. Кнута – Дж. Морриса – В. Пратта выполняется за время  $O(n)$ ; при этом время, необходимое для предварительной обработки образца  $P$  (для формирования массива граней), равно  $O(m)$ . Тогда общее время —  $O(n+m)$ .

Действительно, для оценки временной сложности алгоритма Д. Кнута – Дж. Морриса – В. Пратта требуется подсчитать количество сравнений. Если результат сравнения  $P[q + 1] = T[i]$  оказывается положительным, то значения  $q$  и  $i$  увеличиваются на единицу и образец  $P$  не сдвигается относительно  $T$ . Если же результат сравнения отрицательный (символы не совпадают), то при  $q = 0$  значение  $i$  увеличивается на единицу, а  $P$  сдвигается вправо на одну позицию. При  $q > 0$  значение  $i$  не меняется и находится грань (путем присвоения  $q := br[q]$  или  $q := brs[q]$ ), дающая совпадение  $P[q + 1] = T[i]$ , что обеспечивает требуемый сдвиг. Таким образом, общее количество сравнений символов не превосходит  $n + t$ , где  $t$  — количество сдвигов, выполненных алгоритмом. Но значение  $t \leq n - m + 1$ , поэтому общее количество сравнений не превосходит значения  $2n$ .

### Упражнения

1. Выполните трассировку алгоритма Д. Кнута – Дж. Морриса – В. Пратта с использованием как массива граней  $br$ , так и массива  $brs$  для следующих примеров:
  - $T=abbabaabbaababba$ ,  $P=abbab$ ;
  - $T=abcabdabcabeabcabdabcabc$ ,  $P=abda$ ;
  - $T=abcabdabcabcabd$ ,  $P=abcabc$ ;
  - $T=abcabcabdabcabcabc$ ,  $P=abcabcabc$ .

2. Докажите утверждение, что алгоритм Д. Кнута – Дж. Морриса – В. Пратта не пропускает вхождений образца  $P$  в текст  $T$ .
3. Пусть образец  $P$  и текст  $T$  имеют вид:  $P = (abcd)^t$  и  $T = (abcd)^r$ , где  $t$  и  $r$  — натуральные числа, причем  $t < r$ . (Например, пусть  $t = 2$ ,  $r = 5$ , тогда  $P = abcdabcd$ ,  $T = abcdabcdabcdabcdabcd$ .) Сколько сравнений потребует алгоритм Д. Кнута – Дж. Морриса – В. Пратта для нахождения всех вхождений  $P$  в  $T$ ?
4. Разработайте полную версию программы поиска образца в тексте по алгоритму Д. Кнута – Дж. Морриса – В. Пратта.
5. В алгоритме Д. Кнута – Дж. Морриса – В. Пратта образец  $P$  «прикладывается» к подстроке  $T$ , и они посимвольно сравниваются слева направо. При несовпадении в каких-либо позициях  $q + 1$  в  $P$  и  $i$  в  $T$  при  $brs[q] > 0$  образец  $P$  сдвигается вправо на  $q - brs[q]$  мест. Этот сдвиг гарантирует совпадение префикса  $P[1..brs[q]]$  с прилегающей подстрокой  $T$ , и далее будут сравниваться символы  $T[i]$  и  $P[brs[q] + 1]$ . Сдвиг, основанный не на  $br[q]$ , а на  $brs[q]$ , гарантирует, что  $P[q + 1] <> P[brs[q] + 1]$ , но он не обеспечивает равенство  $T[i]$  и  $P[brs[q] + 1]$ , поэтому символ  $T[i]$  вновь участвует в сравнении. Если ввести ограничение, что каждый символ  $T$  может участвовать в сравнении только один раз, то модифицированный алгоритм Д. Кнута – Дж. Морриса – В. Пратта называют *алгоритмом реального времени*. Измените алгоритм Д. Кнута – Дж. Морриса – В. Пратта указанным образом.

*Примечание.* Для этого нужно преобразовать одномерный массив граней  $brs[1..m]$  в двумерный массив  $brsa[1..m, |A|]$ , где  $|A|$  — мощность конечного используемого алфавита  $A$ , следующим образом. Пусть  $x \in A$ . Для каждой позиции  $q$  образца  $P$  вычислим  $brsa[q, x]$  как такую грань  $P[1..q]$ , что  $P[brs[q] + 1] = x$ . Это преобразование осуществляется за линейное время. Далее предположим, что произошло несовпадение символов  $T[i]$  и  $P[q + 1]$  и что  $T[i] = x$ . Тогда осуществляется сдвиг  $P$  на  $q - brsa[q, x]$  позиций вправо. В этом случае не только префикс  $P[1..brsa[q, x]]$  совпадает с соответствующими символами  $T$ , но и символ  $T[i]$  совпадает

с символом  $P[brsa[q, x] + 1]$ , поэтому следующее сравнение следует выполнять между символами  $P[brsa[q, x] + 2]$  и  $T[i + 1]$ .

## 2.2. Алгоритм Р. Бойера – Дж. Мура

Но как вы спелись!

*Из к/ф «Покровские ворота»*

В алгоритме Р. Бойера – Дж. Мура образец перемещается по тексту слева направо, но (и это отличительная черта!) сравнение символов выполняется *справа налево*, т. е. первыми (при первом прикладывании) сравниваются символы  $P[m]$  и  $T[m]$ . Данный алгоритм после его появления имеет, вероятно, наибольшее количество «продолжателей»: многочисленные авторы развивали его и предложили свои модификации этого алгоритма, являющегося, если так можно выразиться, передовым, «пионерским» в описываемой проблематике.

Первая эвристика алгоритма Р. Бойера – Дж. Мура заключается в том, что если символа  $T[m]$  и любого другого, участвующего в сравнении, нет в  $P$ , то образец можно сдвинуть *за* этот символ. Так, если при первом прикладывании  $P[m] \neq T[m]$  и  $T[m]$  нет в  $P$ , то  $P$  сдвигается сразу на  $m$  позиций, и следующими символами, подлежащими сравнению, являются  $P[m]$  и  $T[2 \cdot m]$ . Если же символ  $T[m]$  есть в образце  $P$ , то  $P$  следует сдвинуть вправо так, чтобы под этим символом оказался самый правый символ  $P$ , совпадающий с  $T[m]$ . И если самое правое вхождение  $T[m]$  находится в позиции  $q$  образца, то величина сдвига определяется как  $m - q$ .

### Пример

В табл. 2.6 (строки 1 и 2) приведена иллюстрация описанных выше случаев. Символа  $\bar{a}$  ( $T[5]$ ) нет в образце, и последний сдвигается за этот символ. Символ же  $b$  ( $T[10]$ ) есть в образце, поэтому осуществляется сдвиг на  $5 - 2 = 3$  позиции.

Предположим, что произошло сравнение  $P[m]$  и  $T[i]$  (например, как в строке 3 табл. 2.6). Тогда далее следует сравнивать предшествующие символы в образце и тексте, пока весь образец не совпадет с подстрокой текста (будет найдено

Таблица 2.6

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
<i>T</i>	a	x	b	c	f	a	b	a	x	b	a	d	a	b	a	x	a	x	a	a	d
1	<i>P</i>	a	b	a	x	a															
2						a	b	a	x	a											
3									a	b	a	x	a								
4													a	b	a	x	a				
5														a	b	a	x	a			
6															a	b	a	x	a		
7																a	b	a	x	a	
8																	a	b	a	x	a

вхождение) или пока не будет встречено несовпадение. В первом случае (строка 4 табл. 2.6) фиксируется факт вхождения, а затем образец сдвигается на одну позицию вправо (пока предполагаем этот, наихудший, вариант сдвига), и процесс поиска продолжается. Во втором случае возможны два варианта. Если символа текста, на котором произошло несовпадение, нет в образце, то последний сдвигается вправо за этот символ. В ситуации же, когда символ текста есть в образце, под ним надо расположить самый правый аналогичный символ образца, обеспечив соответствующий сдвиг (строка 5 табл. 2.6), но только тогда, когда этот символ находится *слева* от места несовпадения в образце. При его нахождении *справа* (символ *x* в строке 6 табл. 2.6) образец сдвигается вправо на одну позицию. Описанная эвристика носит название «правило плохого символа».

### Пример

В табл. 2.7 приведены сдвиги образца  $P = \text{abcxabcde}$  в тексте  $T = \text{ababcxcdedeaxaabcxabcde}$ .

При первом «прикладывании» (строка 1 в табл. 2.7) несовпадение произошло между символами  $T[6] = x$  и  $P[6] = b$ . Символ *x* есть в *P* на четвертом месте, поэтому величина сдвига равна  $6 - 4 = 2$ . Во второй строке произошло несовпадение  $T[9] \neq P[7]$ . Символ  $T[9] = e$  есть в *P*, но он находится справа от  $P[7]$ , — значит, образец сдвигаем на одну позицию вправо. В третьей строке произошло несовпадение  $T[12] \neq P[9]$ . Самый правый символ *a* в *P* находится на пятом

Таблица 2.7

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	<b>T</b>	a	b	a	b	c	x	c	d	e	d	e	a	x	a	a	b	c	x	a	b	c	d	e
<b>1</b>	<b>P</b>	a	b	c	x	a	b	c	d	e														
<b>2</b>				a	b	c	x	a	b	c	d	e												
<b>3</b>					a	b	c	x	a	b	c	d	e											
<b>4</b>								a	b	c	x	a	b	c	d	e								
<b>5</b>											a	b	c	x	a	b	c	d	e					
<b>6</b>															a	b	c	x	a	b	c	d	e	

месте, следовательно, величина сдвига —  $9 - 5 = 4$ . В четвертой строке вновь произошло несовпадение  $T[16] \neq P[9]$ . Самый правый символ  $b$  в  $P$  находится на шестом месте — сдвигаем  $P$  на  $9 - 6 = 3$  позиции. Аналогичные действия представлены и в пятой строке. После совпадения  $P$  с подстрокой из  $T$  (строка 6) образец, если бы текст не закончился, сдвинулся бы на одну позицию вправо (как в худшем случае).

Перейдем теперь к формальной реализации описанной эвристики. Для фиксации самого правого вхождения символов алфавита (предположим, что это буквы латинского алфавита от  $a$  до  $z$ ) в  $P$  нам необходим массив — обозначим его как  $bs$ . Формирование его значений может осуществляться следующим образом.

```

Procedure ShiftBadSymbol;
{Массив  $bs$  ( $bs: \text{Array}['a'.. 'z']$  Of Integer),
так же как образец  $P$  и количество символов  $m$  в  $P$ , -
глобальные переменные}
Var  $i: \text{Integer}$ ;
     $q: \text{Char}$ ;
Begin
    For  $q := 'a'$  To  $'z'$  Do  $bs[q] := 0$ ;
    {Начальная инициализация}
    For  $i := 1$  To  $m$  Do  $bs[P[i]] := i$ ;
End;

```

Простой алгоритм поиска образца в тексте (см. п. 1.1) с вводом рассмотренной эвристики изменяется не принципиально. Цикл **For** заменяется на цикл **While** (что естест-

венно), и в случае несовпадения символов при очередном прикладывании величина изменения индекса  $i$  формируется с помощью сформированного массива  $bs$  — массива позиций самого правого вхождения символов алфавита в образец  $P$ :

```

Procedure BadSymbol;
  Var i, j: Word;
  Begin
    i:=1;
    While i<=(n-m+1) Do Begin
      j:=m;
      While (j>=1) And (P[j]=T[i+j-1]) Do j:=j-1;
      If j=0 Then Begin
        WriteLn('образец ', P, ' входит в ', T,
          ' с позиции ', i);
        i:=i+1;
      End
      Else i:=i+Max(1, j-bs[t[i+j-1]]);
      {Функция Max – нахождение максимального
        из двух целых чисел}
    End;
  End;

```

«Правило плохого символа» можно улучшить. В массиве  $bs$  хранятся только позиции самого правого вхождения символов в образец  $P$ , поэтому это правило обеспечивает сдвиг более чем на одну позицию при несовпадениях, близких к правому концу образца. Если же правое вхождение символа пройдено, произошло несовпадение левее его правого вхождения в образец и символом  $T$  является этот символ, то сдвиг выполняется на одну позицию. Если теперь массив  $bs$  преобразовать в двумерную структуру для хранения места ближайшего появления символов в  $P$  слева от каждой позиции  $i$ , то та же эвристика даст возможность выполнять бóльшие сдвиги при поиске.

### Пример

Пусть  $P = abcdabdcscbad$ . Двумерный массив  $bs$  (табл. 2.8) содержит данные о ближайших левых вхождениях символов в образец. Так,  $P[12] = d$ , причем слева в  $P$  символ  $d$  повторяется на 7-м месте.

Таблица 2.8

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>
a	0				1						5	
b		0				2				6		
c			0					3	8			
d				0			4					7
...												

Массив *bs* можно организовать и несколько иначе — так, как показано в табл. 2.9, когда формируются списки вхождения каждого символа в образец.

Таблица 2.9

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
a	11	5	1	0
b	10	6	2	0
c	9	8	3	0
d	12	7	4	0

Логика формирования массива *bs* для расширенного «правила плохого символа» претерпевает незначительное изменение:

**Procedure** ShiftBadSymbol;    
 {Массив *bs* (*bs:Array['a'..'z',0..NMax]* *Of Integer*),  
 так же как образец *P* и количество символов *m* в *P*, —  
 глобальные переменные}

**Var** *i:Integer*;

*q:Char*;

**Begin**

**For** *q:= 'a' To 'z' Do*

**For** *i:=0 To m Do* *bs[q,i]:=0*;

{Начальная инициализация}

**For** *i:=m DownTo 1 Do Begin*

*bs[P[i],0]:=bs[P[i],0]+1*;

*bs[P[i],bs[P[i],0]]:=i*;

**End**;

**End**;

Формализованная запись расширенного «правила плохого символа» имеет вид:

```

Procedure BadSymbol;
  Var i, j, k: Word;
      w: Char;
  Begin
    i:=1;
    While i<=(n-m+1) Do Begin
      j:=m;
      While (j>=1) And (P[j]=T[i+j-1]) Do j:=j-1;
      If j=0 Then Begin
        WriteLn('образец ', P, ' входит в ', T,
          ' с позиции ', i);
        i:=i+1;
      End
      Else Begin
        k:=1;
        w:=T[i+j-1];
        While (k<=bs[w,0]) And (bs[w,k]>j) Do k=k+1;
        i:=i+Max(1, j-bs[w,k]);
        {Функция Max – нахождение максимального
          из двух целых чисел}
      End;
    End;
  End;
End;

```

Прежде чем перейти к обсуждению второй эвристики, разберем случай вхождения  $P$  в  $T$ . До этого момента предполагалось, что сдвиг в этой ситуации осуществляется на одну позицию. А можно ли сделать это более эффективно?

*Пример*

Пусть  $T = \text{abaabaabaabaabaaba}$ , а  $P = \text{abaaba}$ . В табл. 2.10 показаны значения сдвигов при поиске вхождения  $P$  в  $T$ .

При первом прикладывании  $P$  совпало с подстрокой из  $T$ . Сдвиг на одну позицию нерационален, если сформирован массив граней ( $br$ , см. п. 1.2.1) для  $P$ , — в данном примере  $br = (0, 0, 1, 1, 2, 3)$ . Правда, в данном случае достаточно только значения  $br[m]$ , которое говорит о том, что у  $P$  есть суффикс этой длины, совпадающий с собственным префикс-

Таблица 2.10

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	<b>T</b>	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	b	a	a	b	a
<b>1</b>	<b>P</b>	a	b	a	a	b	a														
<b>2</b>					a	b	a	a	b	a											
<b>3</b>								a	b	a	a	b	a								
<b>4</b>										a	b	a	a	b	a						
<b>5</b>											a	b	a	a	b	a					
<b>6</b>													a	b	a	a	b	a			
<b>7</b>																a	b	a	a	b	a

сом, и как этот суффикс, так и этот префикс сравнивались с соответствующими символами  $T$ , ибо у нас зафиксировано совпадение. Следовательно, при сдвиге на  $m - br[m]$  позиций (а меньшие сдвиги приводят так или иначе к несовпадению  $P$  с соответствующей подстрокой  $T$ ) первые  $br[m]$  символов  $P$  будут совпадать с символами из  $T$ . В нашем примере в строках 1, 2, 3, 6 табл. 2.10  $P$  сдвигается относительно  $T$  на три позиции. Но из этого примера прямо «бьет в глаза» еще одно возможное улучшение алгоритма! Так, в строке 1 найдено вхождение  $P$  в  $T$ , и мы выполнили сдвиг на  $m - br[m] = 6 - 3 = 3$  позиции. Символы префикса  $P$  длиной  $br[m]$  совпадают с соответствующими символами  $T$ . Тогда в строке 2 следует сравнивать только символы  $T[9]$  с  $P[6]$ ;  $T[8]$  с  $P[5]$  и  $T[7]$  с  $P[4]$ . В общем же случае, если после сдвига  $P[m]$  «приложен» к  $T[k]$ , надо сравнивать символы  $P[m] \dots P[m - br[m] + 1]$  с  $T[k], \dots, T[k - br[m] + 1]$ .

Пусть на стадии предварительной обработки у нас сформирован массив граней  $br$  (п. 1.2.1, процедура `MaxBorderArray`). Тогда реализация разобранного выше случая обработки (при вхождении  $P$  в  $T$ ) приводит к следующим изменениям первого варианта процедуры `BadSymbol` (назовем модифицированную логику процедурой `BadSymbolJump`). В переменной `jump` мы фиксируем место в  $P$ , до которого следует осуществлять сравнение символов  $P$  и  $T$ , начиная с последнего символа в соответствии с логикой метода Р. Бойера – Дж. Мура.

```

Procedure BadSymbolJump;
  Var i, j, jump: Integer;
  Begin
    i:=1;
    jump:=1;
    While i<=(n-m+1) Do Begin
      j:=m;
      While (j>=jump) And (P[j]=T[i+j-1]) Do j:=j-1;
      If j=jump-1 Then Begin
        WriteLn('образец', P, ' входит в ',
                T, ' с позиции ', i);
        i:=i+br[m];
        jump:=m-br[m]+1;
        {При новом прикладывании сравниваем P и T
         только до этой позиции}
      End
      Else Begin
        i:=i+Max(1, j-bs[t[i+j-1]]);
        jump:=1;
        {Пока нам неизвестно, как работать
         с переменной jump в этом случае}
      End;
    End;
  End;

```

Перейдем теперь к обсуждению второй эвристики алгоритма Р. Бойера – Дж. Мура — к «*правилу хорошего суффикса*». В чем состоит его суть и как обеспечить наибольшие из возможных сдвиги образца относительно текста?

Пусть образец  $P$  приложен к тексту  $T$  и осуществляется сравнение символов слева направо. Пусть при этом успешно, т. е. с положительным результатом, выполнены сравнения для подстроки  $w$ , и символы не совпадают в позиции  $q$  образца (рис. 2.1). Требуется выполнить сдвиг  $P$  относительно  $T$  на как можно большее количество позиций, но так, чтобы не пропустить вхождения  $P$  в  $T$ .



Рис. 2.1. Основная идея «правила хорошего суффикса»

Очевидно, что если бы мы знали самое правое вхождение  $w$  в  $P$ , т. е. позицию  $j$  (рис. 2.1), то  $P$  можно было бы сдвинуть вправо на  $q + 1 - j$  позиций (или на  $m - (j + |w| - 1)$ ). Таким образом, основная идея сдвигов по «правилу хорошего суффикса» в алгоритме Р. Бойера – Дж. Мура заключается в том, чтобы на стадии предварительной обработки за линейное время (от  $m$ ) проанализировать структуру  $P$  на предмет знания о вхождении суффиксов в образец и использовать эти знания на стадии поиска вхождений  $P$  в  $T$ .

На рис. 2.1 в позиции  $q$  указан символ  $a$ , а в позиции  $j - 1$  — символ  $b$ . Эти символы предшествуют подстрокам  $w$  в  $P$ . Если бы эти символы совпадали, то смысл сдвига на  $q + 1 - j$  позиций терялся бы, поскольку  $P[q]$  не совпадает с соответствующим символом  $T$ . При  $P[q] \neq P[j - 1]$  обычно говорят о «*сильном правиле хорошего суффикса*», а если это условие не рассматривается (как в первоначальном варианте алгоритма Р. Бойера – Дж. Мура), то просто о «*правиле хорошего суффикса*».

Напомним методы предварительной обработки образца за линейное время, рассмотренные в п. 1.2 (рис. 2.2).

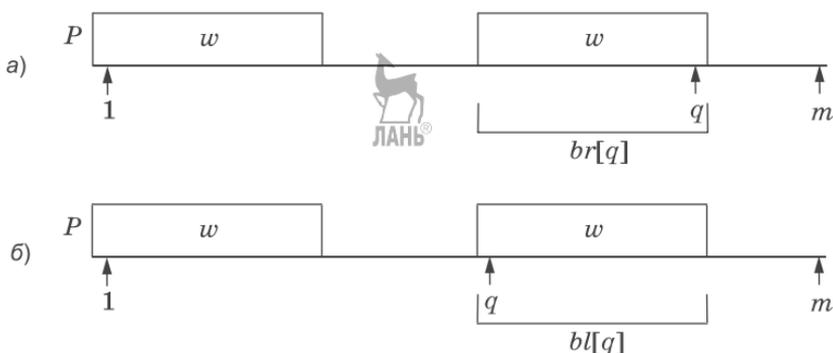


Рис. 2.2. Общие схемы методов предварительной обработки образца: а) вычисление граней; б) вычисление блоков

Гранью строки  $S$  мы называем любой собственный префикс, равный суффиксу  $S$ . Естественным обобщением при этом является понятие наибольшей грани. Вычисление граней для всех подстрок  $S[1..i]$  ( $i=1..m$ ) приводит к понятию массива граней  $br$ . Значение  $br[q]$  говорит о том, что имеется наибольший суффикс  $w$  подстроки  $S[1..q]$ , совпадающий с префиксом  $S$  (рис. 2.2а).

Блок строки  $S$  в позиции  $q$  есть длина наибольшей подстроки  $S$ , начинающейся в позиции  $q$  и совпадающей с префиксом  $S$  (рис. 2.2б).

Как грани строки, так и ее блоки вычисляются за линейное время, но в «чистом» виде не отвечают требованиям алгоритма Р. Бойера – Дж. Мура, ибо в данном случае речь должна идти о вхождении суффиксов в  $S$ .

Возможен следующий переход от имеющихся методов предварительной обработки к требованиям алгоритма Р. Бойера – Дж. Мура. «Перевернем»  $P$  в  $P'$ , например строку  $P = abcdef$  преобразуем в  $P' = fedcba$  (эта операция требует линейного времени). Для  $P'$  подсчитаем массив блоков  $bl$  (рис. 2.3). Какая информация относительно исходной строки будет при этом заложена в  $bl$ ? Относительно исходной строки  $P$  из данных по блокам перевернутой строки можно получить информацию о длинах наибольших суффиксов подстрок  $P[1..i]$ , совпадающих с суффиксом полной строки  $P$ . Сформируем эти длины для каждой подстроки  $P[1..i]$  в массиве  $bsuf$ . Для этого требуется:

- 1) получить  $P'$  из  $P$ ;
- 2) вычислить массив  $bl[i]$  ( $i = 1..m$ ) для  $P'$ ;
- 3) выполнить фрагмент логики:

```
For i:=1 To m Do bsuf[i]:=bl[m-i+1]
```

На рис. 2.3 приведен пример строки  $P$ . Массив  $bl$  сформирован для строки  $P'$ . Значение  $bl[4] = 2$ . С одной стороны (см. верхнюю часть рис. 2.3), это говорит о том, что с позиции 4 в  $P'$  есть подстрока длины 2, совпадающая с префиксом  $P'$ , а с другой — что в  $P$  есть подстрока длины 2, правый конец которой заканчивается в позиции 12, и эта подстрока

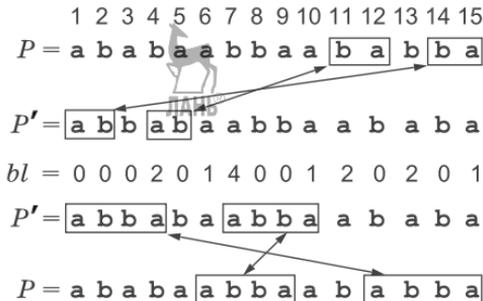


Рис. 2.3. Пример, иллюстрирующий, как блоки перевернутой строки описывают вхождение суффиксов исходной строки

совпадает с суффиксом  $P$  из двух символов. В нижней части рис. 2.3 эта схема иллюстрируется для значения  $bl[7]$ . В  $P'$  есть подстрока из четырех символов с началом в позиции 7, совпадающая с префиксом  $P'$ , а в  $P$  суффикс длиной 4 повторяется, начиная с позиции 9 влево. При этом символы, предшествующие суффиксу и подстроке, не совпадают, что следует из правила формирования массива блоков  $bl$ . Массив  $bsuf$  для примера на рис. 2.3 равен (1, 0, 2, 0, 2, 1, 0, 0, 4, 1, 0, 2, 0, 0, 0).

Однако для реализации логики Р. Бойера – Дж. Мура требуется указание не на длины вхождений суффиксов, а на крайние правые (или левые) позиции вхождения копий этих суффиксов (см. рис. 2.1). Поэтому массив длин следует преобразовать в массив позиций ( $psuf$ ), причем из всех возможных длин требуется выбирать ту, которая дает наибольшее значение правой границы вхождения суффикса. Другими словами, для каждой позиции  $i$  значение  $psuf[i]$  определяет наибольшую позицию, такую, что  $P[i..m]$  совпадает с суффиксом подстроки  $P[1..psuf[i]]$ .

Для примера на рис. 2.3 массив  $psuf = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 0, 12, 10)$ . Это значит, что суффикс  $P[1..10] = P[15..15]$ , суффикс  $P[1..12] = P[14..15]$ , а суффикс  $P[1..9] = P[12..15]$ . Действительно,  $bsuf[9] = 4$  ( $j = 9$ ), т. е. подстрока из четырех символов влево от позиции 9 совпадает с суффиксом  $P$ . Левая граница суффикса  $P$  вычисляется как  $i = m - bsuf[9] + 1 = 15 - 4 + 1 = 12$ . Таким образом,  $psuf[i] := j$ , или  $psuf[12] = 9$ .

Итак, формирование  $psuf$  осуществляется по следующей логике:

```

For i:=1 To m Do psuf[i]:=0;
For j:=1 To m-1 Do Begin
  i:=m-bsuf[j]+1;
  psuf[i]:=j;
End;

```

Резюмируем первую часть «сильного правила хорошего суффикса». Пусть в нашем примере на рис. 2.3 после прикладывания  $P$  к тексту несовпадение произошло в одиннадцатой позиции ( $q = 11$ , см. рис. 2.1). Значение  $psuf[12] = 9$ . После сдвига  $P$  на  $m - psuf[q+1]$  позиций на месте суффикса  $P$  размещается его самая правая копия (выражение  $q - i + 1$  — см. рис. 2.1 — преобразуется в  $m - psuf[q+1]$ ).

А можно ли построить логику сдвигов в этом случае, используя идею вычисления граней строки? В п. 1.2.1 вычислялись грани как подстрока  $P[1..i]$  — грани префиксов, так и подстрока  $P[i..m]$  — грани суффиксов ( $i = 1..m$ ). Очевидно, поскольку сравнения в алгоритме Р. Бойера – Дж. Мура идут слева направо, более приемлем второй вариант.

### Пример

В табл. 2.11 приведен пример строки (образца)  $P$ . В массиве  $bw$  фиксируются грани суффиксов строки. Так,  $bw[6] = 4$  говорит о том, что с шестой позиции начинается подстрока  $P[6..9]$  длиной в четыре символа, которая совпадает с суффиксом  $P[12..15]$ . Массив граней суффиксов вычисляется с помощью процедуры `BorderRigth` (см. п. 1.2.1). В массиве  $bwt$  фиксируются грани суффиксов, но с дополнительным условием: символы, предшествующие подстроке  $P[i..m]$ , и грани этой подстроки не совпадают, или, другими словами,  $P[i - 1] \neq P[m - bw[i]]$ . Тогда логика вычислений имеет следующий вид:

```

Procedure BorderGRigth(P:String);
  Var i:Word;
  Begin
    bwt[m]:=0;
    For i:=m-1 DownTo 1 Do
      If P[m-bw[i]]<>P[i-1]
        Then bwt[i]:=bw[i]
        Else bwt[i]:=bwt[bw[i]];
  End;

```

Таблица 2.11

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>P</b>	a	b	a	b	a	a	b	b	a	a	b	a	b	b	a
<b>bw</b>	1	2	1	2	1	4	3	2	1	1	2	1	0	0	0
<b>bwt</b>	1	2	0	2	0	4	0	0	0	1	2	0	0	0	0
<b>pbwt</b>	0	0	0	0	0	0	0	0	0	0	0	6	0	11	10

После рассмотрения этого примера становится ясно, что необходимые величины сдвигов в «правиле хорошего суффикса» (массив  $pbwt$ ) формируются из значений  $bwt$  за один шаг. При этом для каждой позиции  $i$  требуется указать, есть

ли суффикс  $P[i..m]$  в  $P$ , и если есть, то указать границу (нижнюю или верхнюю) самого правого его вхождения в  $P$ , причем символы, предшествующие суффиксу и его копии, должны не совпадать. Логика этого шага:

```

Procedure GudSuf (S:String) ;
  Var i, j:Word;
  Begin
    For i:=1 To m Do pbwt[i]:=0;
    For j:=1 To m Do Begin
      i:=m-bwt[j]+1;
      pbwt[i]:=j;
    End;
  End;

```

В примере (табл. 2.11)  $pbwt[12] = 6$ . Это означает, что  $P[12..15] = P[6..9]$  и символы  $P[11] \neq P[5]$ . Тогда если при сравнении  $P$  с  $T$  совпадение не произошло в позиции  $q = 11$ , то  $P$  можно сдвинуть относительно  $T$  на  $q + 1 - pbwt[q + 1]$  позиций вправо.

Теперь у нас остался открытым еще один логически возможный случай (по крайней мере, на текущий момент рассмотрения алгоритма): когда вхождения копии суффикса в образец  $P$  нет, т. е. соответствующий элемент массива  $psuf$  или  $pbwt$  равен нулю. Сдвиг на одну позицию при этом вряд ли рационален. Здесь требуется найти наибольшее значение  $j$  (рис. 2.4), такое, что  $P[1..j]$  является суффиксом строки  $P[q + 1..m]$ . Другими словами, ищется наибольшая грань  $P$  при условии  $j < m - q$ . Так как подстрока  $P[q + 1..m]$  совпала с соответствующей подстрокой  $T$ , то:

- 1) мы имеем право сдвинуть образец  $P$  на  $m - j$  позиций, так как меньшие сдвиги, исходя из структуры  $P$  и факта произошедших сравнений с  $T$ , приведут к несовпадениям;
- 2) после сдвига первые  $j$  символов  $P$  и те символы  $T$ , к которым они приложены, будут совпадать.



**Рис. 2.4.** Наибольший префикс  $P$ , совпадающий с суффиксом  $P$ , при условии, что  $j < m - q$

Найти значение  $j$  для каждого  $P[q]$  (сформировать элементы массива, который мы назовем  $brsuf$ ) следует на стадии предварительной обработки. Пусть массив граней  $br$  получен. Для формирования  $brsuf$  требуется знать элементы убывающей последовательности  $br[m], br^2[m], \dots, br^k[m] = 0$ . Тогда для каждого такого  $q$ , что  $0 < q \leq m - br[m]$ , положим  $brsuf[q] = br[m]$ . Так, для значений  $q$  из интервала  $m - br[m] < q \leq m - br^2[m]$  мы имеем  $brsuf[q] = br^2[m]$  и т. д. Очевидно, что массив  $brsuf$  при известном  $br$  вычисляется за линейное время.

```

Procedure BrSuff;
  Var l, t, q: Word;
  Begin
    l:=0;
    t:=br[m];
    For q:=1 To m Do Begin
      If (q>l) And (q<=m-t) Then brsuf[q]:=t;
      If q=m-t Then Begin
        l:=m-t;
        t:=br[t];
      End;
    End;
End;

```

Достаточно очевидно, что в этом случае, по аналогии с вхождением  $P$  в  $T$ , переменной  $jump$ , определяющей, до какой позиции  $P$  необходимо выполнять сравнения при следующем прикладывании к  $T$ , следует присвоить значение  $brsuf[q]$ .

С учетом всего вышесказанного требуется изменить следующий фрагмент рассмотренной ранее процедуры `Search`:

```

Begin
  i:=i+Max(1, j-bs[t[i+j-1]]);
  jump:=1;
  {Пока нам не известно, как работать
   с переменной  $jump$  в этом случае}
End;

```

Выберем второй способ реализации «правила хорошего суффикса» — через массив граней — и будем считать, что на

стадии предварительной обработки  $P$  вычислены значения элементов массивов  $bs$ ,  $br$ ,  $pbwt$ ,  $brsuf$ . Тогда мы получим следующую логику этого фрагмента:

**Begin**

**If**  $pbwt[j+1]=0$  **Then**  $v:=m-brsuf[j+1]$

**Else**  $v:=j+1-pbwt[j+1]$ ;

$q:=\text{Max}(v, j-bs[t[i+j-1]]]$ ;

$i:=i+q$ ;

**If**  $q=m-brsuf[j+1]$  **Then**  $jump:=q$

**Else**  $jump:=1$ ;

**End;**

Оценка времени работы алгоритма Р. Бойера – Дж. Мура — это более сложная задача, чем для алгоритма Д. Кнута – Дж. Морриса – В. Пратта. Ей посвящены многочисленные теоретические и практические исследования. Показано, что среднее количество выполняемых буквенных сравнений является почти линейной функцией от длины текста. Для текстов на естественном языке эта оценка имеет вид  $0,3 \cdot n$  (при условии, что длина образца больше 10, но намного меньше значения  $n$ ). Алгоритм же Д. Кнута – Дж. Морриса – В. Пратта в этой ситуации требует  $1,0 \cdot n$  буквенных сравнений. Однако алгоритм Р. Бойера – Дж. Мура «плохо себя ведет» на кратных строках, а в худшем случае имеет точно такую же временную оценку, что и простой алгоритм поиска образца в тексте, т. е.  $O(m \cdot n)$ .



### Упражнения

1. Разработайте программу поиска  $P$  в  $T$  только с использованием «правила плохого символа» (рассмотрите как простой случай, так и улучшенный). Оцените время ее работы на различных тестовых примерах, включая случай, когда  $P = a^m$ ,  $T = a^n$  ( $m < n$ ). Сравните время работы этой программы с временными характеристиками обычного алгоритма поиска образца в тексте.
2. Измените решение упражнения 1 так, чтобы при нахождении  $P$  в  $T$  сдвиг  $P$  относительно  $T$  осуществлялся не на одну позицию, а на значение, определяемое максимальной длиной префикса  $P$ , совпадающего с суффиксом  $P$ .

3. Приведите пример строки  $P$  и выполните вручную следующие действия:
- «переверните»  $P$  в  $P'$ ;
  - вычислите массив блоков  $bl$  для  $P'$ ;
  - преобразуйте массив  $bl$  в  $bsuf$  — массив длин наибольших суффиксов подстрок  $P[1..i]$ , совпадающих с суффиксом полной строки  $P$ ;
  - преобразуйте массив  $bsuf$  в массив позиций  $psuf$ , определяющий крайние правые позиции вхождения суффиксов  $P$  в образец  $P$  (при этом необходимо учесть, что если позиция  $i$  входит в несколько вхождений, то следует выбирать вхождение, дающее наибольшее значение правой границы).
4. Приведите пример строки  $P$  и выполните вручную следующие действия:
- вычислите массив  $bw$  граней суффиксов строки;
  - преобразуйте массив  $bw$  в массив  $bwt$ , определяющий грани суффиксов строки, но с дополнительным условием, что символы, предшествующие подстроке  $P[i..m]$ , и грани этой подстроки не совпадают;
  - преобразуйте массив длин  $bwt$  в массив номеров позиций  $pbwt$ .
5. Определите, имеются ли ошибки в следующем варианте реализации алгоритма Р. Бойера – Дж. Мура. В любом случае найдите возможность оптимизировать программный код:

```

Procedure Search;
  Var i, j, jump, v, q: Integer;
  Begin
    jump:=1;
    i:=1;
    While (i<=n-m+1) Do Begin
      j:=m;
      While (j>=jump) And (P[j]=T[i+j-1]) Do
        j:=j+1;
      If j=jump-1 Then Begin
        WriteLn('образец входит в текст
                с позиции: ',i);
        i:=i+Max(1, br[m]);
        jump:=m-br[m]+1;
      End
    End

```

```

End
Else Begin
  If pbwt[j+1]#0b Then v:=m-brsuf[j+1]
    Else v:=j+1-pbwt[j+1];
  q:=Max(v, j-bs[t[i+j-1]]);
  i:=i+q;
  If q=m-brsuf[j+1] Then jump:=q
    Else jump:=1;
End;
End;
End;

```

## 2.3. Алгоритм Р. Карпа – М. Рабина

Математика может открыть  
определенную последовательность  
даже в хаосе.

*Гертруда Стайн*

Алгоритм Р. Карпа – М. Рабина в «идейном» плане кардинально отличается от рассмотренных ранее методов, и его изучение — это необходимая составляющая образования специалиста по информатике, показывающая, как, казалось бы, «на ровном месте» можно находить новые результаты по преодолению ограниченности компьютера.

Ограничение на алфавит —  $A = \{0, 1\}$ . Это не столь принципиально, ибо, в конечном счете, строку вполне можно трактовать как двоичную последовательность. Пусть  $P$  есть двоичная последовательность длины  $m$ , а  $T$  — двоичная последовательность длины  $n$ . Как можно реализовать поиск вхождения  $P$  в  $T$ ?

Образец  $P$  преобразуем в десятичное число традиционным образом (перевод числа из двоичной системы счисления в десятичную назовем преобразованием  $H$ ) —  $H(P) = \sum_{i=1}^m 2^{m-i} P[i]$ .

Аналогичную операцию выполним и для  $T$ , начиная с позиции  $r$ , —  $H(T_r) = \sum_{i=1}^m 2^{m-i} P[r+i-1]$ . Очевидно, что если  $H(P) = H(T_r)$ , то  $P$  входит в  $T$ , начиная с позиции  $r$ .

*Пример*

$P = 10101$ ,  $m = 5$  и  $H(P) = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$ . Если  $T = 1011010101$ ,  $r = 2$ , то  $H(T_2) = 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$ , а при  $r = 6$  значение  $H(T_6) = 21$ . Вывод: образец  $P$  входит в  $T$  с шестой позиции (и с третьей).

Таким образом, задача поиска вхождения  $P$  в  $T$  здесь фактически сведена к вычислительной задаче, в которой сравниваются два числа  $H(P)$  и  $H(T_r)$ . Но определение  $T_r$  для каждого значения  $r$  — трудоемкая задача, а кроме того, степени двойки быстро возрастают, поэтому преобразование  $H$  для реальных образцов  $P$  становится малоэффективным.

Развитием этой идеи — преобразования строки в число — стало использование одного из классических разделов теории чисел, а именно *модульной арифметики*. (В принципе, и компьютерное выполнение арифметических операций — это вычисления по модулю максимально допустимого числа в интервале, определенном типом данных.) При этом вместо работы со слишком большими числами  $H(P)$  и  $H(T_r)$  предлагается выполнять действия с остатками по модулю некоторого числа  $q$  (относительно небольшого, что позволяет использовать малое количество битов). Но в этом случае возникает другая проблема: различные числа могут иметь один и тот же остаток, что неизбежно приводит к ложным совпадениям  $P$  и  $T$ . Поэтому следует выбирать  $q$  так, чтобы вероятность ложного совпадения была минимальной. Итак,  $H(P)$  переводится в  $H_q(P)$  путем нахождения  $H(P) \bmod q$  и  $H_q(T_r) = H(T_r) \bmod q$ . Однако такая последовательность действий, как вычисление  $H(P)$  и  $H(T_r)$ , а затем нахождение остатков по модулю  $q$ , еще не решает проблемы, ибо промежуточные результаты остаются прежними (большими), а не в диапазоне от 0 до  $q - 1$ . Как же быть? Рассмотрим простой пример.

*Пример*

Пусть  $q = 7$ . Вычислим  $2^5 \bmod 7 = 32 \bmod 7 = 4$ . Это прямое вычисление. А теперь вычислим  $((((2 \cdot 2 \bmod 7 \cdot 2) \bmod 7) \cdot 2) \bmod 7 \cdot 2) = 4$ , т. е. будем выполнять операцию дальнейшего возведения в степень над остатком предыдущего

вычисления, Как видим, промежуточные результаты при этом существенно меньше.

Модульная арифметика<sup>1)</sup> дает возможность оперировать при арифметических вычислениях (любых, не обязательно только с возведением в степень) с остатками по модулю  $q$ , поэтому промежуточные результаты никогда не превысят значение  $2 \cdot q$ .

Второй используемый математический факт — всем известная *схема Горнера* для вычисления значений полиномов:

$$a_{m-1} \cdot x^{m-1} + \dots + a_2 \cdot x^2 + a_1 \cdot x^1 + a_0 = (((((a_{m-1} \cdot x + a_{m-2}) \cdot x + a_{m-3}) \cdot x) + \dots + a_1) \cdot x + a_0).$$

Другими словами, возведения в степень здесь уже нет. Схема Горнера позволяет обойтись операциями умножения и сложения, а кроме того, промежуточные результаты в процессе вычислений остаются небольшими.

В рассматриваемой нами задаче  $x$  равно 2, а коэффициенты  $a_i$  — это двоичные разряды в представлении  $P$  и  $T_r$ . Интегрируя оба указанных выше математических факта в единое целое, вычисление  $H(P)$  можно осуществить по схеме:  $H(P) = (((((P[1] \cdot 2 \text{ Mod } q + P[2]) \cdot 2 \text{ Mod } q + P[3]) \cdot 2 \text{ Mod } q + P[4]) \dots) \cdot 2 \text{ Mod } q + P[m]) \text{ Mod } q$ . Отметим, что операция умножения на 2 при компьютерной реализации заменяется простым сдвигом числа на один разряд влево, необходимо только исключить потерю старшего разряда в результате такого сдвига.

### Пример

Если  $P = 1010111$  и  $q = 11$ , то  $H(P) = 87$  и  $H(P) = 87 \text{ Mod } 11 = 10$ . Вычисления по приведенной выше схеме имеют вид:

$$\begin{aligned} 1 \cdot 2 \text{ Mod } 11 + 0 &= 2; & 2 \cdot 2 \text{ Mod } 11 + 1 &= 5; \\ 5 \cdot 2 \text{ Mod } 11 + 0 &= 10; & 10 \cdot 2 \text{ Mod } 11 + 1 &= 10; \\ 10 \cdot 2 \text{ Mod } 11 + 1 &= 10; & 10 \cdot 2 \text{ Mod } 11 + 1 &= 10. \end{aligned}$$

Аналогично вычисляется и  $H(T_r)$  для любого значения  $r$ . Однако схему вычисления при  $r > 1$  можно упростить (повысить ее эффективность), получая значение  $H(T_r)$  из  $H(T_{r-1})$  за постоянное небольшое количество операций. Тогда  $H(T_r) = H(T_r) \text{ Mod } q$  и  $H(T_r) = 2 \cdot H(T_{r-1}) - 2^m \cdot T[r-1] + T[r+m-1]$ . Подобная ситуация показана на рис. 2.5.

<sup>1)</sup> См., например, *Виноградов И. М.* Основы теории чисел. — М.: Наука, 1972.

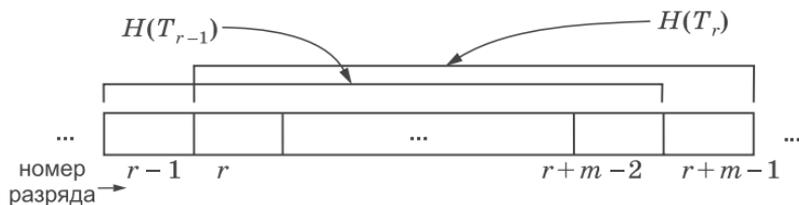


Рис. 2.5. Схема преобразования  $H(T_{r-1})$  в  $H(T_r)$

Если каждую последующую степень двойки брать по модулю 2, т. е.  $(2^m \bmod q) = 2 \cdot (2^{m-1} \bmod q) \bmod q$ , то  $H(T_r) = (2 \cdot H(T_{r-1}) \bmod q) - (2^m \bmod q) \cdot [T[r-1] + T[r+m-1]] \bmod q$  вычисляется из  $H(T_{r-1})$  за постоянное время.

Очевидно, что если  $P$  входит в  $T$  начиная с позиции  $r$ , то  $H_q(P) = H_q(T_r)$ , но обратное верно не всегда. В данной схеме поиска  $P$  в  $T$  возможны ложные совпадения.

#### Пример

$P = 1010111$ ,  $q = 5$ ,  $H(P) \bmod q = 2$ ,  $T = 010101110010110$ . Получаем, что  $H_5(P) = H_5(T_2) = H_5(T_9)$ . В первом случае совпадение истинное, во втором — ложное.

Ключ к решению проблемы ложных совпадений опять-таки дает теория чисел: следует в качестве  $q$  выбирать простое число и использовать свойства простых чисел. Значение  $\pi(q)^{1)}$  определим как количество простых чисел, не превосходящих положительного целого  $q$ . Для  $\pi(q)$  известна такая оценка:  $\frac{q}{\ln q} \leq \pi(q) \leq 1.26\pi(q)^{2)}$ .

Приведем (без доказательства) основную теорему Р. Карпа – М. Рабина<sup>3)</sup>.

**Теорема.** Пусть  $P$  и  $T$  — некоторые строки, причем  $n \cdot m \geq 29$ , где  $m = |P|$  и  $n = |T|$ . Пусть  $I$  — некоторое положительное число. Если  $q$  — случайно выбранное простое число, не превосходящее  $I$ , то вероятность ложного совпадения  $P$  и  $T$  не превосходит  $\frac{\pi(nm)}{\pi(I)}$ .

1) Здесь греческая буква  $\pi$  использована только как условное обозначение и не имеет отношения к числу  $\pi = 3,1415926\dots$  — Прим. ред.

2) Бухштаб А. А. Теория чисел. — М.: Учпедгиз, 1960.

3) Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 110–111.

Удивительным является тот факт, что данная теорема верна при любом выборе образца  $P$  и текста  $T$ , таких, что  $m \cdot n \geq 29$ . «Вероятность» в этой теореме относится не к случайному выбору  $P$  и  $T$ , а к случайному выбору простого  $q$ , не превосходящего значение  $I$ .

Из приведенной теоремы нетрудно видеть, что при возрастании  $I$  вероятность ложного совпадения убывает. Однако в этом случае увеличивается и значение  $q$ , что приводит к увеличению затрат времени на вычисление  $H_q(P)$  и  $H_q(T_r)$ .

А вот еще один любопытный факт<sup>1)</sup>. Если  $I = m \cdot n^2$ , то вероятность ложного совпадения не превосходит  $2.53/n$ .

Действительно:

$$\frac{\pi(mn)}{\pi(mn^2)} \leq 1.26 \frac{mn}{mn^2} \frac{\ln(mn^2)}{\ln(mn)} = 1.26 \frac{1}{n} \left( \frac{\ln m + 2 \ln n}{\ln m + \ln n} \right) \leq \frac{2.53}{n}.$$

*Пример*

Пусть  $m = 100$ , а  $n = 2000$ , тогда  $I = 10^2 \cdot (2 \cdot 10^3)^2 = 4 \cdot 10^8 < 4 \cdot (10^3)^{2.7} = 2^2 \cdot (2^{10})^{2.7} = 2^{29} < 2^{32}$ . Получаем, что 32 битов достаточно для представления простого числа  $q$ , причем модульная арифметика будет эффективно работать, а вероятность ложного совпадения не будет превосходить  $2.53/2000 < 0,001265$ .

Итак, алгоритм Р. Карпа – М. Рабина имеет временную оценку  $O(n)$ . На стадии предварительной обработки следует выбрать простое число  $q$ , не превосходящее положительно-го целого  $I$ , и вычислить  $H(P)$ . Дальнейшие действия сводятся к последовательному вычислению  $H(T_r)$  и его сравнению с  $H(P)$ . Из-за возможности ложного совпадения при равенстве  $H(P)$  и  $H(T_r)$  можно организовать побитовое сравнение этой подстроки  $T$  с  $P$ .

## Упражнения

1. При заданном  $m$  (например, 100) определите минимально возможное значение  $n$ , при котором 32 разрядов не будет хватать для представления  $I$ .

<sup>1)</sup> Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 112.

2. Разработайте программную реализацию алгоритма Р. Карпа – М. Рабина. Выбор простого числа осуществите с помощью известных вам алгоритмов. На случайных тестах оцените количество ложных совпадений.

ЛАНЬ®

## 2.4. Алгоритм Shift-And

Ты не забывай, что у меня в голове опилки, и длинные слова меня только огорчают...

*Алан Милн. «Винни-Пух и все-все-все»*

Это — удивительный алгоритм, являющийся основой целого класса методов решения задач на строках, включая приближенный поиск образца в тексте. Единственное принципиальное его ограничение — небольшое значение  $m$ , т. е. образец не должен быть очень длинным.

Итак, пусть образец  $P$  имеет длину  $m$ , текст  $T$  —  $n$ . Будем искать несколько больше, чем от нас требуется, — не только все вхождения  $P$  в  $T$ , но и вхождения в  $T$  всех возможных префиксов  $P$ .

### Пример

Пусть образец  $P = \text{acacd}$ , а текст  $T = \text{acacdfafacacda}$ . При этом  $P$  имеет пять префиксов:  $a$ ,  $ac$ ,  $aca$ ,  $acac$  и  $acacd$ . Для каждой позиции текста необходимо знать, является ли она концом вхождения не только образца, но и любого из его префиксов. С этой целью для каждой позиции текста  $i$  мы построим  $m$ -элементный массив из нулей и единиц, в котором  $j$ -й элемент равен 1, если  $j$ -й префикс слова входит в предыдущий текст и заканчивается на этой позиции, а иначе  $j$ -й элемент равен 0. В итоге мы получим двумерный массив  $R$  из  $n$  строк и  $m$  столбцов, который представлен в табл. 2.12. (Введение нулевого столбца  $R$ , заполненного единицами, будет объяснено чуть позже.)

Рассмотрим формирование четвертой строки  $R$  (для четвертой позиции текста), в табл. 2.12 она выделена курсивом. Первый префикс  $a$  не входит в предыдущий текст, заканчиваясь на данной позиции, поэтому первый элемент массива равен 0. Второй префикс  $ac$  входит в текст и заканчивается на данной позиции, поэтому второй элемент ра-

Таблица 2.12

$R$	Символ $T \downarrow$	Символ $P \rightarrow$	a	c	a	c	d
Номер символа $T \downarrow$	Номер символа $P \rightarrow$	0	1	2	3	4	5
1	a	1	1	0	0	0	0
2	c	1	0	1	0	0	0
3	a	1	1	0	1	0	0
4	c	1	0	1	0	1	0
5	d	1	0	0	0	0	1
6	f	1	0	0	0	0	0
7	a	1	1	0	0	0	0
8	f	1	0	0	0	0	0
9	a	1	1	0	0	0	0
10	c	1	0	1	0	0	0
11	a	1	1	0	1	0	0
12	c	1	0	1	0	1	0
13	d	1	0	0	0	0	1
14	f	1	0	0	0	0	0

вен 1. Далее аналогично рассматриваются префиксы аса (не входит), асас (входит) и асасd (не входит) и заполняются третий, четвертый и пятый элементы строки, которые равны 0, 1 и 0 соответственно. Единицы же в пятой и тринадцатой строках табл. 2.12 означают, что в текст  $T$  входит весь образец  $P$ .

Осталось научиться быстро формировать  $R$ , строка за строкой. Пусть  $R[i]$  —  $i$ -я строка таблицы. На самом деле каждая строка  $R[i]$  зависит только от предыдущей строки  $R[i - 1]$ , от  $P$  и от текущего символа текста  $T[i]$ . Итак, пусть у нас уже имеется строка  $R[i - 1]$  и на обработку поступил символ текста  $T[i]$ . Строим следующую строку  $R[i]$ . Так, в рассматриваемом примере  $R_i[5] = 1$ , т. е. пятый префикс асасd входит в текст и заканчивается на текущей позиции  $i$ , только если одновременно выполняются два условия:

- 1) в текст входит и заканчивается на предыдущей позиции  $i - 1$  меньший префикс асас, или  $R[i - 1, 4] = 1$ ;
- 2) текущий символ текста  $T[i]$  совпадает с пятым символом  $P$ , или  $T[i] = P[5]$ .

Первое условие означает, что все предыдущие символы префикса *asad* до последнего, который рассматривается в данный момент, входят в текст, а второе — что и последний символ префикса *asad* входит в текст. Аналогично,  $R[i,4] = 1$  только тогда, когда одновременно  $R[i-1,3] = 1$  и  $T[i] = P[4]$  и т. д. Другими словами, очередная строка  $R$  для всех  $1 \leq j \leq m$  строится из предыдущей по формуле:

$$R[i, j] = \begin{cases} 1, & \text{если } R[i-1, j-1] = 1 \text{ и } T[i] = P[j]; \\ 0, & \text{в остальных случаях.} \end{cases}$$

Единственное исключение возникает при заполнении первой позиции  $R[i,1]$ , для которой первое условие, очевидно, всегда истинно, поскольку предыдущих символов попросту нет. Чтобы указанная формула работала и в этом случае, будем хранить в дополнительном нулевом элементе каждой строки единицу, т. е.  $R[i,0] = 1$  для всех  $1 \leq i \leq n$  (это тот самый нулевой столбец, существование которого мы обещали объяснить в предыдущем примере).

В результате мы получим следующий алгоритм поиска подстроки в строке:

- просматриваем текст;
- на основании предшествующей информации для каждой его позиции  $i$  строим битовый массив;
- проверяем, появилась ли единица в конце нового массива: если да, то слово входит в текст и заканчивается на текущей позиции  $i$ , а начинается с позиции  $i - m + 1$ .

```

Procedure Search(P, T:String);
  Const Wsize=...;
  {Максимальный размер слова w}
  Var R:Array [0..Wsize] of Byte;
      {Битовый массив}
      n, m, i, j:Byte;
Begin
  n:=Length(T);
  m:=Length(P);
  R[0]:=1;
  For j:=1 To m Do R[j]:=0;
  For i:=1 To n Do Begin
    For j:=m DownTo 1 Do

```

{Вычисляем битовый массив для текущего символа текста, используя старый массив. Кстати, самостоятельно ответьте на вопрос: почему мы вычисляем его, начиная с последнего элемента?}

```
If (R[j-1]=1) And (T[i]=P[j]) Then R[j]:=1
Else R[j]:=0;
```

```
If R[m]=1 Then WriteLn(i-m+1);
```

{Если последний элемент - единичный,  
то слово найдено}

```
End;
```

```
End;
```

Итак, на каждом шаге такой алгоритм позволяет получить информацию о вхождении не только всего слова, но и любого из его префиксов. Однако во временной эффективности выигрыша пока нет: для каждого значения  $i$  всегда выполняется  $m$  сравнений, так что временная сложность в любом случае составляет  $O(m \cdot n)$ .

На первый взгляд, попытка оказалась неудачной, но давайте сделаем следующий шаг. Если длина искомого слова не превышает 32, то любую из строк таблицы (а это — массив из нулей и единиц) можно представить в виде одной целочисленной переменной типа `LongInt`: целое число ведь тоже состоит из нулей и единиц (в его двоичном представлении)! Оказывается, что тогда очередную битовую строку можно вычислять всю сразу, без цикла по ней, за несколько битовых операций.

Рассмотрим, например, как по третьей строке 10100 строится четвертая — 01010. Напомним, что единица в четвертой строке на некоторой позиции  $j$  может появиться только при одновременном выполнении двух условий:

- 1) в третьей строке слева от позиции  $j$  стоит единица (т. е. в текст входят все первые символы  $j$ -го префикса до последнего);
- 2) текущий символ текста (для нашего примера это  $c$ ) совпадает с  $j$ -м символом образца (т. е. в текст входит и последний символ  $j$ -го префикса).

Для проверки первого условия достаточно сдвинуть вправо третью строку 10100. А так как в дополнительном нулевом элементе всегда хранится единица, получаем 11010. Вспомним, что обычный сдвиг `ShR` заполняет освободившийся разряд нулем, поэтому после сдвига еще необхо-

димо установить единицу в первом элементе. В тех позициях получившейся новой строки, где стоят единицы, возможно, будут стоять единицы и искомой нами четвертой строки: для всех префиксов, помеченных этими единицами, их первые символы до последнего уже точно входят в текст.

Для проверки второго условия для каждого символа алфавита мы заранее подготовим *характеристический вектор* длиной  $m$  (обозначим его как  $V$ ). В нем  $j$ -й элемент равен 1, если данный символ совпадает с последним символом  $j$ -го префикса (или просто совпадает с  $j$ -м символом образца), и равен 0 в противном случае. Для рассматриваемого примера характеристические векторы представлены в табл. 2.13

Таблица 2.13



$v$	a	с	a	с	d
a	1	0	1	0	0
с	0	1	0	1	0
d	0	0	0	0	1

Для всех остальных символов алфавита характеристические векторы будут нулевыми (00000), так как этих символов вообще нет в слове.

Пусть нам известен характеристический вектор текущего символа текста. Для символа с это 01010. В тех позициях вектора, где стоят единицы, возможно, будут стоять единицы и искомой четвертой строки. Для всех префиксов, помеченных этими единицами, последний символ совпадает с текущим символом текста.

Осталось объединить оба этих условия — ведь они должны выполняться одновременно. При этом позиции, для которых и в третьей строке, сдвинутой вправо, и в векторе стоит единица (т. е. префикс полностью входит в текст), получают также единичное значение, а остальные позиции обнуляются. Используется для этого, конечно же, битовая операция And. А получаемая в результате строка и становится четвертой:  $11010 \text{ And } 01010 = 01010$ .

Итак, у нас все готово, чтобы сформулировать окончательный вариант *алгоритма Shift-And*. Вначале для каждого символа алфавита мы вычисляем его характеристический вектор — целое число типа LongInt. Далее просматриваем текст: как и раньше, для каждой его позиции  $i$  строим

битовую строку (вектор), но храним ее уже в одной переменной типа `LongInt`. Сдвигаем старую строку вправо, в ее первый элемент устанавливаем единицу и объединяем через `And` с характеристическим вектором текущего символа текста. Опять же, но с помощью битовых операций, проверяем, появилась ли 1 в конце новой строки. Если да, то слово входит в текст и заканчивается на текущей позиции  $i$ , а значит, начинается с позиции  $i - m + 1$ . (Первоначально же битовая строка нулевая.)

```

Procedure SearchShiftAnd(P,T:String);
Var V:Array [Chr(0)..Chr(255)] Of LongInt;
  {Массив для хранения характеристических векторов
   всех символов таблицы ASCII, от 0-го до 255-го}
  n,m,i:Byte;
  R,first:LongInt;
  j:Char;
Begin
  n:=Length(T);
  m:=Length(P);
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
  For i:=1 To m Do V[P[i]]:=V[P[i]] Or
    (1 ShL (m-i));
  {Вычисляем характеристические векторы}
  first:=1 ShL (m-1);
  {Число вида 10..0, используется для установки 1
   в первую позицию битовой строки}
  R:=0;
  For i:=1 To n Do Begin
    R:=(R ShR 1) Or first) And V[T[i]];
    {Вычисляем битовую строку для текущей позиции}
    If R And 1=1 Then WriteLn(i-m+1);
    {Если последний бит единичный, то слово
     найдено}
  End;
End;

```



Алгоритм, по существу, получился линейным, с временной сложностью  $O(n)$ . Внутренний цикл в нем заменен тремя битовыми операциями, а они выполняются даже быстрее обычных арифметических операций! Ограничение же на длину слова (максимум 32 символа) приемлемо для ряда практических применений.

Рассмотрим теперь одну из возможных модификаций алгоритма *Shift-And* для приближенного поиска  $P$  в  $T$ .

**Задача.** Предположим, что искомый образец в тексте содержится, возможно, с одним несовпадающим символом. Необходимо, несмотря на это несовпадение, найти вхождение образца.

*Пример*

Пусть  $P = \text{acacd}$  и  $T = \text{acacafacacda}$ . Тогда вхождения  $P$  в  $T$  будут начинаться с первого и седьмого символов, причем первое из них будет неточным, а второе — точным вхождением.

Для поиска точных вхождений сформируем массив  $R_0$  (табл. 2.14), аналогичный тому, что представлен в алгоритме *Shift-And*. Для поиска же неточных вхождений определим массив  $R_1$  (табл. 2.15), который похож на первый с той лишь разницей, что отражает как точные вхождения, так и вхождения при одном измененном символе.

Таблица 2.14

$R_0$	Символ $T \downarrow$	Символ $P \rightarrow$	a	c	a	c	d
Номер символа $T \downarrow$	Номер символа $P \rightarrow$	0	1	2	3	4	5
1	a	1	1	0	0	0	0
2	c	1	0	1	0	0	0
3	a	1	1	0	1	0	0
4	c	1	0	1	0	1	0
5	a	1	1	0	1	0	0
6	f	1	0	0	0	0	0
7	a	1	1	0	0	0	0
8	c	1	0	1	0	0	0
9	a	1	1	0	1	0	0
10	c	1	0	1	0	1	0
11	d	1	0	0	0	0	1
12	a	1	1	0	0	0	0

*Примечание.* Мы говорим о массивах  $R_0$  и  $R_1$  для наглядности изложения. Фактически, как и ранее, для формирования очередной строки  $R[i]$  необходимо знать только  $R[i - 1]$ .

Как и прежде, для каждой позиции текста мы составляем битовую строку (вектор) длиной  $m$ . В ней  $j$ -й элемент равен 1, если  $j$ -й префикс образца входит в предыдущий текст (возможно, с одним измененным символом) и заканчивается на этой позиции, иначе  $j$ -й элемент равен 0. Единица в конце пятой и одиннадцатой строк  $R_1$  означает, что в текст входит (возможно, с одним измененным символом) весь образец, и данная позиция является концом его вхождения.

Таблица 2.15

$R_1$	Символ $T \downarrow$	Символ $P \rightarrow$	а	с	а	с	д
Номер символа $T \downarrow$	Номер символа $P \rightarrow$	0	1	2	3	4	5
1	а	1	1	0	0	0	0
2	с	1	1	1	0	0	0
3	а	1	1	0	1	0	0
4	с	1	1	1	0	1	0
5	а	1	1	0	1	0	1
6	ф	1	1	1	0	1	0
7	а	1	1	0	1	0	0
8	с	1	1	1	0	1	0
9	а	1	1	0	1	0	0
10	с	1	1	1	0	1	0
11	д	1	1	0	1	0	1
12	а	1	1	0	0	0	0

Как можно быстро построить таблицу  $R_1$ ?

Пусть у нас уже имеются строки  $R_1[i - 1]$  и  $R_0[i - 1]$ , а на обработку поступил символ текста  $T[i]$ . Построим следующую строку  $R_1[i]$ . Единица в новой строке  $R_1$  для некоторого  $j$ -го префикса может появиться только в двух случаях:

- если все первые символы  $j$ -го префикса, кроме последнего символа, точно совпадают с текстом, т. е.  $R_0[i-1, j-1] = 1$  (проверять на совпадение последний символ не требуется, он может быть и отличающимся — это допустимо);
- если в первых символах  $j$ -го префикса уже имеется один измененный символ, но последний символ этого префикса совпадает с текущим символом текста, т. е.  $R_1[i-1, j-1] = 1$  и  $T[i] = P[j]$ .

Если сдвинуть вправо (с дополнением единицей) предыдущую строку таблицы  $R_0$ , то мы получим все единицы новой строки  $R_1$  по первому случаю, и для всех префиксов, помеченных этими единицами, изменение может быть только в последнем символе. Если же сдвинуть вправо (с дополнением единицей) предыдущую строку  $R_1$  и соединить ее через And с характеристическим вектором текущего символа текста, то мы получим все единицы новой строки  $R_1$  по второму случаю. Для всех префиксов, помеченных этими единицами, изменение символа, возможно, произошло где-то внутри. Остается только собрать все единицы в новой строке  $R_1$  при помощи операции Or.

Рассмотрим, как в обсуждаемом примере строится восьмая строка  $R_1$ . Седьмая строка  $R_0$  равна 10000, после сдвига и дополнения единицей она переходит в 11000. Часть единиц восьмой строки  $R_1$  уже есть. Седьмая строка  $R_1$  10100 после сдвига, дополнения единицей и операции And с характеристическим вектором текущего символа с (01010) становится равной 01010. Теперь у нас есть и другая часть единиц. Соединим полученные строки через Or, так как нам нужны единицы из обеих строк: 11000 Or 01010 = 11010. Это и будет восьмая строка  $R_1$ .

```
Procedure SearchShiftCh(P, T:String);
```

```
  Var V:Array [Chr(0)..Chr(255)] Of LongInt;
      n,m,i:Byte;
      R0,R1,first: LongInt;
      j:Char;
```

```
Begin
```

```
  n:=Length(T);
```

```
  m:=Length(P);
```

```
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
```



```

For i:=1 To m Do V[P[i]]:=V[P[i]] Or
                                     (1 ShL (m-i));
R0:=0;
R1:=0;
first:=1 ShL (m-1);
For i:=1 To n Do Begin
  {Вычисляем битовые строки}
  R0:=(R0 ShR 1) Or first;
  R1:= R0 Or ((R1 ShR 1) And V[T[i]]);
  {Дополнение единицей можно не делать,
   так как она уже есть в R0}
  R0:=R0 And V[T[i]];
  If R1 And 1 = 1 Then WriteLn(i-m+1);
End;
End;

```

Можно ли обобщить рассмотренную задачу на случай более чем одного несовпадения символов?

Предположим, что задано возможное количество несовпадений  $w$  и необходимо найти все вхождения  $P$  в  $T$  с точностью до этих  $w$  несовпадений.

Определим значение  $R_k[i, j]$  равным единице, если  $j$ -й префикс  $P$  совпадает с точностью до  $k$  символов с подстрокой  $T$ , заканчивающейся в позиции  $i$ . В противном случае  $R_k[i, j]$  равно нулю. Очевидно, что массивы  $R$  вычисляются последовательно:  $R_0, R_1, \dots, R_k, \dots, R_w$ . При этом при вычислении значения  $R_k[i, j]$  логически возможны следующие варианты:

- первые  $j - 1$  символов  $P$  совпадают с подстрокой  $T$ , кончающейся в позиции  $i - 1$  с не более чем  $k$  несовпадениями, и  $P[j]$  равно  $T[i]$ ; это условие записывается как  $R_k[i - 1]_{\rightarrow 1}$  разряд **And**  $V[T[i]]$ ;
- первые  $j - 1$  символов  $P$  совпадают с подстрокой  $T$ , кончающейся в позиции  $i - 1$ , с не более чем  $k - 1$  несовпадениями; это условие записывается как  $R_{k-1}[i - 1]_{\rightarrow 1}$  разряд.

Остается объединить указанные условия с помощью операции **Or**.

### Пример

Пусть  $P = abcd$  и  $T = abceabfeakrt$ . Последовательное вычисление массивов  $R$  показано в табл. 2.16.

Таблица 2.16

Сим-вол	Номер символа	$R_0$				$R_1$				$R_2$				$R_3$			
a	1	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
b	2	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
c	3	0	0	1	0	1	0	1	0	1	1	1	0	1	1	1	0
e	4	0	0	0	0	1	0	0	1	1	1	0	1	1	1	1	1
a	5	1	0	0	0	1	0	0	0	1	1	0	0	1	1	1	0
b	6	0	1	0	0	1	1	0	0	1	1	0	0	1	1	1	0
f	7	0	0	0	0	1	0	1	0	1	1	1	0	1	1	1	0
e	8	0	0	0	0	1	0	0	0	1	1	0	1	1	1	1	1
a	9	1	0	0	0	1	0	0	0	1	1	0	0	1	1	1	0
k	10	0	0	0	0	1	1	0	0	1	1	0	0	1	1	1	0
r	11	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0
t	12	0	0	0	0	1	0	0	0	1	1	0	0	1	1	1	1

Единица в последнем столбце  $R_k$  говорит о вхождении  $P$  в  $T$  с не более чем  $k$  несовпадениями.

### Упражнения

1. Выполните трассировку алгоритма *Shift-And*, например, при  $P = abac$  и  $T = aabaccababacab$ .
2. Экспериментально сравните время работы простого алгоритма поиска образца в тексте и алгоритма *Shift-And*.
3. В алгоритме *Shift-And* на вхождение префиксов указывает 1. Изменим условие: пусть на вхождение префикса указывает значение 0. Тогда это уже будет алгоритм не *Shift-And*, а «*Shift-Or*». Разберите приведенный ниже листинг и проверьте его работоспособность.

```

Procedure SearchShiftOr(P, T:String);
  Var V:Array [Chr(0)..Chr(255)] Of LongInt;
      n,m,i:Byte;
      h,R: LongInt;
      j:Char;

```

```

Begin
  n:=Length(T);
  m:=Length(P);
  h:=1 ShL m-1;
  For j:=Chr(0) To Chr(255) Do V[j]:=h;
  For i:=1 To m Do V[P[i]]:=V[P[i]] And
    (Not(1 ShL (m-i)));
  R:=h;
  For i:=1 To n Do Begin
    R:=(R ShR 1) Or V[T[i]];
    If R And 1 = 0 Then WriteLn(i-m+1);
  End;
End;

```

*Примечание.* В основном цикле на одну битовую операцию здесь стало меньше, так как дополнение единицей при сдвиге не требуется. (Объясните почему.)

4. Разработайте метод «регистронезависимого» (т. е. вне зависимости от регистра букв) поиска образца в тексте.

*Примечание.* Для этого достаточно для одних и тех же букв разного регистра построить одинаковые характеристические векторы. Основная же часть алгоритма *Shift-And* останется без изменений.

5. Известно, что в тексте без цифр некоторые буквы были заменены цифрами. Найдите вхождение образца в таком «испорченном» тексте.

*Примечание.* Для этого следует характеристические векторы цифр сделать единичными: это будет означать, что цифра может появиться в любой позиции слова.

6. Автомобильные номера имеют вид: «буква, три цифры, две буквы», например «м815тк». Найдите в тексте все такие автомобильные номера.

*Примечание.* Характеристические векторы для всех букв нужно положить равными 100011, для цифр — 011100, а для всех остальных символов — нулевыми.

7. В тексте некоторые буквы, например о, были заменены буквой а. Как найти слово в таком «испорченном» тексте?

*Примечание.* Нужно сформировать характеристические векторы для всех символов алфавита, а затем скор-

ректировать вектор для буквы а, соединяя его через Or с вектором для буквы о. Тем самым определяется, что буква а может стоять в слове и на месте буквы о.

8. Необходимо найти в тексте не конкретный образец, а *образец по некоторому шаблону*, например по шаблону  $c^*c^*s$ , в котором символ \* заменяет любой символ. То есть требуется найти все слова из пяти букв, в первой и третьей позициях которых находится символ с, а в последней — символ s.

*Примечание.* После формирования характеристических векторов их следует подвергнуть корректировке: во второй и четвертой позициях (т. е. в позициях с символом \*) в этих векторах следует поставить единицы. Это означает, что в этих позициях может находиться любой символ.

9. Пусть квадратные скобки указывают группу символов для поиска. Например, по запросу  $b[ai]ржа$  должны быть найдены слова баржа и биржа, т. е. слова, начинающиеся с символа б и заканчивающиеся на ржа, где вторым символом может быть как а, так и и. Выполните соответствующую модификацию алгоритма *Shift-And*.

*Примечание.* Квадратные скобки определяют одну некоторую позицию слова. В характеристических векторах для всех символов, записанных в квадратных скобках, надо установить единицу в данной позиции.

10. Пусть фигурные скобки указывают группу символов, которых *не должно* содержаться в искомом образце. Например, по запросу  $\{abc\}гпу$  необходимо найти все слова из четырех букв, не начинающиеся на а, в или с и заканчивающиеся на гпу. Выполните соответствующую модификацию алгоритма *Shift-And*.

*Примечание.* Фигурные скобки тоже определяют одну некоторую позицию слова. В характеристических векторах для всех символов, записанных в фигурных скобках, надо записать ноль в данной позиции.

11. Предположим, что искомый образец в тексте содержится, возможно, с одним лишним символом. Разработайте логику построения соответствующих таблиц по анало-

гии с табл. 2.14 и 2.15. Проверьте работоспособность следующего листинга:

```

Procedure SearchShiftIns (P, T:String);
  Var V:Array [Chr(0)..Chr(255)] Of LongInt;
      n,m,i:Byte;
      R0,R1,first:LongInt;
      j:Char;
Begin
  n:=Length(T);
  m:=Length(P);
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
  For i:=1 To m Do V[P[i]]:=V[P[i]] Or
      (1 ShL (m-i));

  R0:=0;
  R1:=0;
  first:=1 ShL (m-1);
  For i:=1 To n Do Begin
    R1:= R0 Or ((R1 ShR 1) Or first) And
      V[T[i]];
    R0:=((R0 ShR 1) Or first) And V[t[i]];
    If R1 And 1 = 1 Then WriteLn(i-m);
  End;
End;

```

12. Предположим, что искомый образец в тексте содержится, возможно, с одним удаленным символом. Как, не смотря на эту «ошибку удаления», найти вхождение образца? Проверьте правильность следующего листинга:

```

Procedure SearchShiftDel (w, t:String);
  Var V:Array [Chr(0)..Chr(255)] Of LongInt;
      n,m,i:Byte;
      R0,R1,first:LongInt;
      j:Char;
Begin
  n:=Length(T);
  m:=Length(P);
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
  For i:=1 To m Do V[P[i]]:=V[P[i]] Or
      (1 ShL (m-i));

  R0:=0;
  R1:=0;

```

```

first:=1 ShL (m-1);
For i:=1 To n Do Begin
  R0:=((R0 ShR 1) Or first) And V[T[i]];
  R1:=((R0 ShR 1) Or first) Or ((R1 ShR 1)
                                Or first) And V[T[i]];
  If R1 And 1 = 1 Then WriteLn(i-m+2);
End;
End;

```

13. *Расстояние P. Хемминга* между двумя строками одинаковой длины определяется как количество позиций, в которых символы не совпадают. Дано слово и некоторое число  $d$ . Необходимо найти все слова в тексте, удаленные от данного слова не более чем на расстояние  $d$ . Разберите приведенное решение и проверьте его правильность.

*Примечание.* Переформулируем задачу: требуется найти слова, которые содержатся в тексте не более чем с  $d$  «ошибками замены». Случай с одной «ошибкой замены» (с одним измененным символом) уже был рассмотрен. Необходимо ввести по одной дополнительной таблице на каждую «ошибку замены» и проводить аналогичные преобразования одной таблицы в другую.

```

Procedure SearchShiftChk(P, T:String; d:Byte);
Const Maxd=...;
Var V:Array [Chr(0)..Chr(255)] Of LongInt;
    R:Array [0..Maxd] Of LongInt;
    n, m, k, i:Byte;
    first, OldR0, tmp:LongInt;
    j:Char;
Begin
  n:=Length(T);
  m:=Length(P);
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
  For i:=1 To m Do V[P[i]]:=V[P[i]]
                                Or (1 ShL (m-i));
  For k:=0 To d Do R[k]:=0;
  first:=1 ShL (m-1);
  For i:=1 To n Do Begin
    OldR0:=R[0];
    R[0]:=((R[0] ShR 1) Or first) And V[T[i]];

```

```

For k:=1 To d Do Begin
  tmp:=R[k];
  R[k]:=((R[k] Shr 1) And V[T[i]])
    Or (OldR0 Shr 1) Or first;
  OldR0:=tmp;
End;
If R[d] And l=1 Then WriteLn(i-m+1);
End;
End;

```

14. Как изменится решение в упражнении 13, если необходимо найти все слова, удаленные от данного слова точно на расстояние  $d$ ?
15. В слове возможна одна замена, одно удаление и одна вставка символа, т. е. все три «ошибки» сразу, но только по одной каждого типа. Какие изменения потребуется внести в алгоритм *Shift-And* для поиска таких слов?
16. Во всех вариантах алгоритма *Shift-And* на вхождение префиксов указывает значение 1. Пусть теперь на вхождение указывает значение 0. Измените решения в упражнениях 8, 11, 12, 13 и 15 с учетом замены 1 на 0.  
*Примечание.* Пример подобной модификации был рассмотрен для точного поиска. При этом дополнение единицей оказалось лишним. Быть может, и здесь это приведет к сокращению количества битовых операций?
17. Пусть угловые скобки указывают группу символов, в которых, возможно, есть одна «ошибка замены». Например, пусть необходимо найти автомобильный номер м815тк и есть уверенность в том, что буквы правильны, а в цифрах возможна одна ошибка, тогда запрос на поиск можно записать как  $m\langle 815 \rangle тк$ . Как модифицировать алгоритм *Shift-And* для поиска слов по такому шаблону?
18. Одним из стандартных знаков в шаблонах является символ #, означающий, что на его место может быть подставлено любое слово. Например, шаблон  $ab\#cd$  означает, что ищется подстрока  $ab$ , за ней следует любой набор символов, а затем на любом расстоянии идет подстрока  $cd$ . Можно ли изменить алгоритм *Shift-And* так, чтобы осуществлялась проверка наличия в тексте слов, соответствующих подобным шаблонам?

## 2.5. Использование элементов теории автоматов в решении задач обработки строк

Нет ничего практичнее хорошей теории.

*Роберт Курцгоф*



### 2.5.1. Основной формализм теории автоматов

Приехал из Германии Войнович. Поселился в гостинице на Бродвее. Понадобилось ему сделать копии. Зашли они с женой в специальную контору. Протянули копировщику несколько страниц. Тот спрашивает:

— One of each? («Каждую по одной?»)

Войнович говорит жене:

— Ирка, ты слышала? Он спросил: «Войнович?» Он меня узнал! Ты представляешь? Вот она, популярность!

*Сергей Довлатов*

Пусть  $A$  — конечное множество символов (алфавит). Определим  $A^*$  как множество всех строк алфавита  $A$ . Как и ранее, символом  $\varepsilon$  обозначим пустую строку (она принадлежит  $A^*$ ). По своей сути, автомат  $M$  — это некое устройство, распознающее определенные элементы из  $A^*$ . Для заданного  $A$  автомат  $M$  состоит из множества состояний  $Q$  и функции переходов  $F: A \cdot Q \rightarrow Q$ . Автомат  $M$  имеет начальное состояние  $q_0 \in Q$  и одно или несколько конечных состояний ( $E$  — множество конечных состояний,  $E \subseteq Q$ ).

Итак,  $M = (A, Q, q_0, E, F)$ . Входом функции  $F$  является пара — символ  $a$ , состояние  $q$ . Автомат находится в состоянии  $q$  и читает символ  $a$ ; в результате он переходит в новое состояние, определяемое  $F[a, q]$ .

*Пример*

$A = \{a, b\}$ ,  $Q = \{q_0, q_1, q_2\}$ ,  $E = \{q_2\}$ . Функция  $F$  приведена в табл. 2.17.

Таблица 2.17

$F$	$a$	$b$
$q_0$	$q_1$	$q_0$
$q_1$	$q_1$	$q_2$
$q_2$	$q_1$	$q_2$

Тракуются данные табл. 2.17 следующим образом: автомат  $M$  находится в состоянии  $q_0$  и читает символ  $a$ , в результате чего он переходит в состояние  $q_1$  ( $F(a, q_0) = q_1$ ), либо автомат  $M$  находится в состоянии  $q_2$  и читает символ  $b$ , в результате чего он переходит в состояние  $q_2$ , так как  $F(b, q_2) = q_2$ .

Работу автомата  $M$  можно наглядно изобразить в виде ориентированного графа (*диаграммы состояний*), у которого метками вершин являются состояния автомата  $q$ , а метками дуг — символы алфавита  $A$  (рис. 2.6).

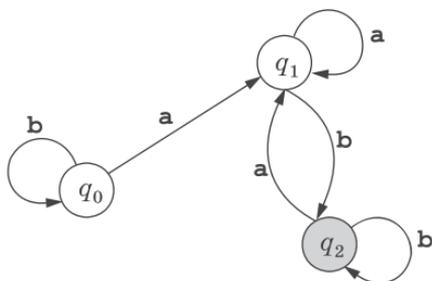


Рис. 2.6. Пример диаграммы состояний автомата

Если автомат  $M$  обрабатывает строки  $ab$ ,  $aab$ ,  $aabb$ ,  $a \dots ab \dots b$ ,  $bab$ ,  $b \dots ba \dots ab \dots b$ , то в итоге  $M$  перейдет в конечное состояние  $q_2$ , и эти строки считаются допустимыми или распознаваемыми. При обработке же строки  $aba$   $M$  в итоге окажется в состоянии  $q_1$ , поэтому такая строка недопустима или нераспознаваема данным автоматом. (Аналогична ситуация и со строками типа  $b \dots b$  или  $b \dots ba \dots a$ .)

Автоматы  $M$ , у которых для любого состояния  $q$  и любого символа алфавита  $A$ , поступающего на обработку, существует одно и только одно состояние — один возможный переход, определяют как *детерминированные автоматы*. Свойство детерминированности следует из того, что  $F$  —

функция. Если же рассматривать  $F$  как множество правил, то в этом случае для некоторого  $a \in A$  и  $q \in Q$  правила может не существовать. Автомат как бы «зависает» и не может работать; о таких автоматах говорят, что они недетерминированны.

На рис. 2.7 приведена диаграмма недетерминированного автомата ( $q_2$  — конечное состояние). Он корректно обрабатывает строки типа  $ab\dots bc$ , но, например, строка  $aa$  приводит к его «зависанию».

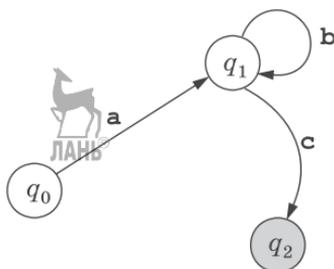


Рис. 2.7. Пример недетерминированного автомата

Известно, что для каждого недетерминированного автомата существует эквивалентный ему детерминированный автомат, решающий ту же задачу<sup>1)</sup>.

Конечный автомат  $M$  фактически определяет некую функцию  $\varphi: A^* \rightarrow Q$ . Значение  $\varphi(S)$  есть состояние, в которое перейдет автомат после обработки строки  $S$ . Автомат допускает строку  $S$ , если  $\varphi(S) \in E$ .

Функция  $\varphi$  допускает рекуррентное определение:

$$\varphi(\epsilon) = q_0;$$

$$\varphi(Sa) = F[\varphi(S), a], \text{ для любых } S \in A^* \text{ и } a \in A \text{ (где } Sa \text{ — «склейка» строки } S \text{ и символа } a \text{).}$$



### Упражнения

1. Для каждого из приведенных на рис. 2.8 автоматов (диаграмм состояний) определите множество допустимых слов.

*Примечание.* Начальное состояние —  $q_0$ , конечные состояния выделены серым цветом.

<sup>1)</sup> Андерсон Д. Дискретная математика и комбинаторика. — М.: Издательский дом «Вильямс», 2003. С. 736–737.

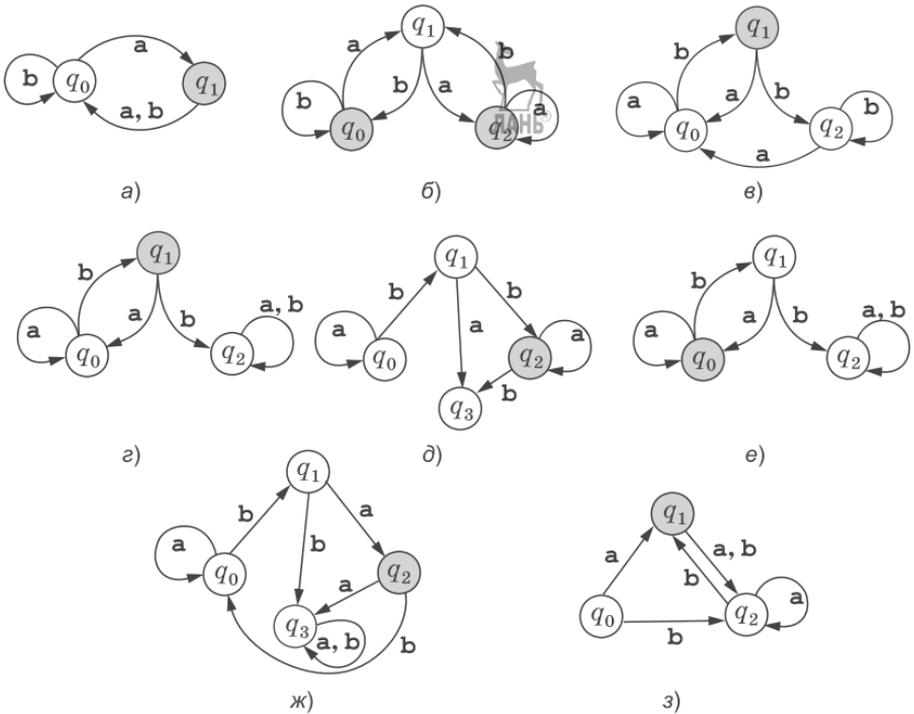


Рис. 2.8. Примеры диаграмм состояний автоматов

2. Постройте автоматы, для которых множества допустимых слов имеют вид:

- $aa\dots abb\dots bcc\dots c$ ;
- $a\dots aba\dots aba\dots ab$ ;
- $a\dots ab$  и  $b\dots ba$ ;
- $aa\dots abac\dots c$  и  $bb\dots baac\dots c$ .

*Примечание.* Через многоточие между символами (например  $a\dots a$ ), обозначается любое количество таких же символов, вплоть до их отсутствия.

## 2.5.2. Конечный автомат для поиска образца



Случайные открытия делают только подготовленные умы.

*Блез Паскаль*

Для поиска образца  $P$  в тексте  $T$  конечный автомат  $M$  конструируется следующим образом. Пусть  $P = aabbabcaaa$ . Диаграмма переходов такого конечного автомата (ориенти-

рованный граф) для заданного  $P$  приведена на рис. 2.9. Состояние  $q_0$  здесь обозначено кружком (вершина ориентированного графа) с меткой  $q_0$  и является начальным. Конечное состояние — вершина с меткой  $q_9$ . Дуги, идущие слева направо, имеют метки, соответствующие символам  $P$ , и направлены от вершин с метками  $q_j$  к вершинам с метками  $q_{j+1}$ ; эти дуги соответствуют успешным этапам поиска  $P$  в  $T$ . Нахождение в состоянии  $q_j$  (вершине с меткой  $q_j$ ) говорит о том, что  $j$  символов  $P$  совпали с  $j$  последними символами  $T$ . Переход в состояние  $q_{j+1}$  говорит об успешности сравнения  $j + 1$  символов образца  $P$  и соответствующего символа  $T$ . Достижение конечного состояния соответствует успешному поиску  $P$  в  $T$ .

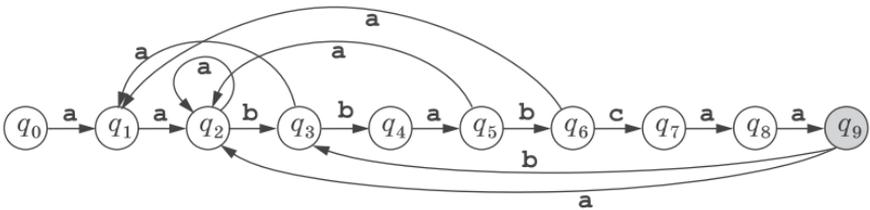


Рис. 2.9. Диаграмма переходов конечного автомата для  $P = aabbabca$

Осталось понять назначение дуг, идущих справа налево (обратных дуг). Пусть мы находимся в состоянии  $q_6$ . Если на обработку из  $T$  поступил символ  $c$ , то мы переходим в состояние  $q_7$  (что очевидно из вышеизложенного текста). А если на обработку поступил символ  $a$ ? У строки  $aabbabca$  длина максимального суффикса, совпадающего с префиксом  $P$ , равна единице (это символ  $a$ ), поэтому и дуга идет в состояние  $q_1$ . Если же поступил символ  $b$ , то это будет дуга, идущая в состояние  $q_0$ , — на рис. 2.9 она не приведена, так как длина максимального суффикса  $aabbabb$ , совпадающего с префиксом  $P$ , равна нулю — пустая строка. (Соответственно, и все дуги, идущие в состояние  $q_0$ , не приведены на рис. 2.9.)

Переход в состояние  $q_1$  говорит о том, что следующее сравнение очередного символа  $T$  будет осуществляться с символом  $P[2]$ . Переход по обратной дуге говорит о наличии префикса  $P$ , совпадающего с суффиксом пройденной части текста  $T$ .

Пусть сравнивался символ текста  $T[i]$ , и автомат перешел в состояние  $j$  — фактически эти параметры определяют величину сдвига  $P$  относительно  $T$ . К  $j$  последним символам  $T$  следует приложить  $j$  первых символов  $P$ , а следующее сравнение (если его делать) следует выполнять между символами  $T[i + 1]$  и  $P[j + 1]$ .

Тем самым мы приблизились к пониманию функции переходов  $F$  описываемого автомата. Для рассматриваемого примера она приведена в табл. 2.18.

Таблица 2.18

$F$		Состояние									
		0	1	2	3	4	5	6	7	8	9
Входной символ	а	1	2	2	1	5	2	1	8	9	2
	б	0	0	3	4	0	6	0	0	0	3
	с	0	0	0	0	0	0	7	0	0	0

Логика формирования функции переходов (если не касаться вопроса ее эффективности) здесь достаточно очевидна. Для каждой позиции  $j$  образца  $P$  и символа алфавита  $A$  мы определяем максимальную длину префикса  $P$ , совпадающего с суффиксом строки ( $P[1..j] + \langle \text{символ алфавита} \rangle$ ). Эта «фраза» в формализованном виде записывается так:

**Procedure** Automat;

**Var**  $j, k$ :Integer;

$q$ :Char;

**Begin**

**For**  $j:=0$  **To**  $m$  **Do**

**For**  $q \in A$  **Do** **Begin**

      {Цикл по символам алфавита  $A$ }

$k := \text{Min}(m, j+1)$ ;

      {Начинаем с максимального значения  $k$ }

**While Not**  $\text{MaxSuf}(k, j, q)$  **And**  $(k > 0)$  **Do**  $k := k - 1$ ;

      {Если префикс  $P[1..k]$  не является суффиксом  $P[1..j] + q$ , то уменьшаем значение  $k$ }

$F[j, q] := k$ ;

      { $F$  — массив типа  $F : \text{Array} [1..Length$

      ( $\text{MaxString}$ ),  $A$ ] **Of** Integer. Первый его индекс

изменяется от 1 до  $\langle \text{длина максимальной возможной строки} \rangle$ , а второй индекс — по символам алфавита  $A$

**End;**

**End;**

Проверка совпадения символов возложена на функцию MaxSuf:

```

Function MaxSuf(k, j:Integer; q:Char) : Boolean;
  Var l:Integer;
      W:String;
  Begin
    {Проверяем: является ли префикс P[1..k] суффиксом
     строки P[1..j]+q}
    W:=Copy(P, 1, j);
    W:=W+q;
    l:=1;
    While (P[k-l+1]=W[j+1-l+1]) And (l<=k) Do l:=l+1;
    If l=k+1 Then MaxSuf:=True
      Else MaxSuf:=False;
  End;

```

Временная сложность логики —  $O(m^3 \cdot |A|)$  — вложенные циклы по  $P$  и символам  $A$ , а затем для каждой пары (позиция, символ) —  $O(m^2)$  на поиск длины максимального префикса  $P$ , совпадающего с суффиксом строки. Оставим пока этот способ вычисления функции  $F$ , ибо мы сознательно не вспоминали про материал из п. 1.2 (понятия грани и массива граней), и приведем алгоритм поиска  $P$  в  $T$  с помощью конечного автомата  $M$ , построенного для образца  $P$ .

```

Procedure SearchAutomat;
  Var i, q:Integer;
  {Идентифицируем состояние qj с его номером j}
  Begin
    n:=Length(T);
    q:=0;
    For i:=1 To n Do Begin
      q:=F[q, T[i]];
      If q=m Then  $\langle \text{образец } P \text{ входит в } T \text{ с позиции } i-m+1 \rangle$ 
    End;
  End;

```

Как видим, при наличии автомата  $M$  для  $P$  время поиска  $P$  в  $T$  есть  $O(n)$ .

### Пример

Пусть  $P = \text{abaababa}$  и  $T = \text{abaabababababaa}$ . Диаграмма переходов  $M$  приведена на рис. 2.10, а функция  $F$  — в табл. 2.19.

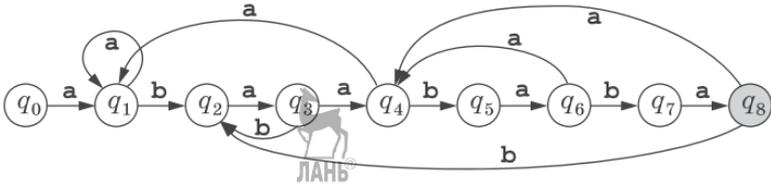


Рис. 2.10. Диаграмма переходов конечного автомата для строки  $P = \text{abaababa}$

Таблица 2.19

$F$		Состояние									
		0	1	2	3	4	5	6	7	8	
Входной символ	а	1	1	3	4	1	6	4	8	4	
	б	0	2	0	2	5	0	7	0	2	

Изменение значения переменной  $q$  (состояния автомата) в процессе поиска  $P$  в  $T$  приведено в табл. 2.20.

Таблица 2.20

$T$		а	б	а	а	а	б	а	б	а	а	б	а	б	а	а
$q$	0	1	2	3	4	1	2	3	2	3	4	5	6	7	8	4

А теперь вспомним о массиве граней  $br$  (см. п. 1.2.1). Он вычисляется за время  $O(m)$  и фактически содержит данные о длинах суффиксов строк  $P[1..i]$ , совпадающих с префиксом  $P$ . (Так, для последнего примера  $br = (0, 0, 1, 1, 2, 3, 2, 3)$ .) Значение  $br[j]$  говорит о том, что у строки  $P[1..j]$  суффикс этой длины совпадает с префиксом  $P$ , или, другими словами, если в состоянии  $j$  автомата обрабатывается символ  $w$ , то нам следует просто узнать, как этот символ обрабатывался в состоянии  $br[j]$ , поскольку его обработка идентична для этих состояний. (В рассматриваемом примере  $br[6] = 3$  и  $F[6, a] = F[3, a]$ .) Тогда логика формирования функции переходов  $F$  автомата  $M$  за время  $O(m \cdot |A|)$  будет следующей:

```

Procedure Automat;
  Var j:Integer;
      q:Char;
  Begin
    For q∈A Do
      If P[1]=q Then F[0,q]:=1 Else F[0,q]:=0;
    For j:=1 To m Do
      For q∈A Do
        {Цикл по символам алфавита A}
        If P[j+1]=q Then F[j,q]:=j+1
          Else F[j,q]:=F[br[j],q];
  End;

```

*Примечание.* Предполагается, что к концу  $P$  приписан символ, отсутствующий в  $A$ , например  $\$$ .



### Упражнения

1. Приведите пример образца и постройте для него автомат распознавания.
2. Постройте автомат распознавания образца в тексте на основе массива граней суффиксов.
3. Сконструируйте автомат (если это возможно) для распознавания образца в тексте с использованием данных, представленных в массиве блоков.



## 2.6. Алгоритм М. Крочемора

Невозможно решить проблему на том же уровне, на котором она возникла. Нужно стать выше этой проблемы, поднявшись на следующий уровень.

*Альберт Эйнштейн*

Пусть имеется строка  $S$  из  $m$  символов. Понятие грани определено в п. 1.2.2 — это любой суффикс  $S[i..m]$  ( $i \neq 1$ ), совпадающий с префиксом  $S$ . Обозначим длину наибольшей грани строки  $S$  через  $z$ . Тогда величина  $r = m \text{ Div } (m - z)$  определяет *кратность строки*, а подстроку  $U = S[1..m - z]$  называют *образующей подстрокой*. Если для строки  $S$  параметр  $r$  — целое число и  $r \geq 2$ , то  $S$  считается кратной строкой.

## Примеры

- 1)  $S = \text{abcabcabcabc}$ ;  $z = 9$ ;  $\Rightarrow r = 4$ ;  $U = \text{abc}$ ;
- 2)  $S = \text{abab}$ ;  $z = 2$ ;  $\Rightarrow r = 2$ ;  $U = \text{ab}$ ;
- 3)  $S = \text{bbbbbb}$ ;  $z = 4$ ;  $\Rightarrow r = 5$ ;  $U = \text{b}$ ;
- 4)  $S = \text{ababa}$ ;  $z = 3$ ;  $\Rightarrow r = 5/2$ , строка не является кратной.

Кратную строку можно записать в виде  $S = (U)^r$ . У любой кратной строки  $S$  образующая  $U$  не является кратной.

**Постановка задачи.** Пусть дан текст  $T[1..n]$ . Требуется найти все кратные подстроки  $T$ .

Описывать вхождение кратной строки  $S$  в  $T$  можно тройкой чисел  $(i, p, r)$ , где  $i$  — индекс позиции символа в  $T$ , с которого начинается  $S$ ;  $p$  — количество символов в образующей;  $r$  — кратность.

## Пример

$S = \text{aababbbbaa}$ .

Ответ:  $(1, 1, 2)$ ,  $(2, 2, 2)$ ,  $(5, 1, 3)$ ,  $(8, 1, 2)$ .

**Алгоритм М. Крочемора** подвергает текст *декомпозиции*. Для этого на каждом шаге (уровне)  $L$  алгоритм находит уникальные подстроки длиной  $L$ . При  $L = 1$  вычисляются последовательности позиций, в которых встречаются одинаковые символы. Далее для уровней  $L > 1$  находятся последовательности позиций, с которых начинаются одинаковые подстроки длиной  $L$ .

## Пример

Текст  $T$  приведен в табл. 2.21 (в первой строке указаны номера позиции символов в тексте).



Таблица 2.21

$i$	1	2	3	4	5	6	7	8	9	10
$T$	a	a	b	a	b	b	b	a	b	§

На рис. 2.11 показана разбивка текста на уникальные подстроки.

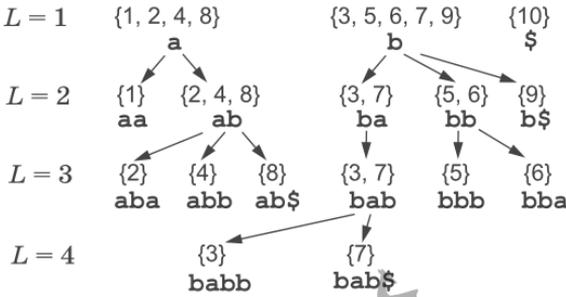


Рис. 2.11. Разбивка текста  $T$  на уникальные подстроки

Выскажем утверждение о том, что переход от  $L$  к  $L + 1$  возможен только на основе последовательностей текущего уровня.

По очевидной логике, для декомпозиции некоторой последовательности уровня  $L$ , представленной последовательностью индексов  $\{i_1, i_2, \dots, i_k\}$ , требуется проверить совпадение символов  $T[i_1 + L], T[i_2 + L], \dots, T[i_k + L]$ . Тогда при совпадении какой-либо пары символов (например,  $T[i_1 + L] = T[i_2 + L]$ ) индексы  $i_1$  и  $i_2$  помещаются в одну и ту же последовательность уровня  $L + 1$  (рис. 2.12). Такая логика имеет временную сложность  $O(n^2)$ .

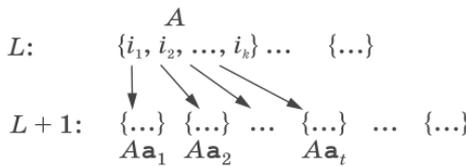


Рис. 2.12. Обычная схема перехода от одного уровня к другому.  $A$  — некоторая подстрока;  $Aa_i$  — concatenация подстроки  $A$  с символом  $a_i$ . Все символы  $a_i$  ( $i = 1..t$ ) различны

**Наблюдение 1.**

Возьмем индексы из одной последовательности уровня  $L$  (например,  $i_1$  и  $i_2$ ) и прибавим к ним по единице:  $j_1 = i_1 + 1, j_2 = i_2 + 1$ . Если окажется, что  $j_1$  и  $j_2$  принадлежат одной последовательности уровня  $L$ , то на уровне  $L + 1$  индексы  $i_1$  и  $i_2$  следует включить в одну подпоследовательность.

Обоснование этого факта «лежит на поверхности». Действительно, пусть  $A$  — строка, соответствующая одной по-

следовательности уровня  $L + 1$ . Убрав из нее первый символ (он общий), мы получаем последовательность длины  $L$ , характеризуемую своей последовательностью индексов. И эти индексы обязаны иметь одну принадлежность; в противном случае нарушается логика декомпозиции текста.

Вернемся к примеру на рис. 2.11. Здесь есть последовательность  $\{2, 4, 8\}$  на уровне 2. Прибавление единицы дает числа 3, 5 и 8. Ни одна пара из этих чисел не принадлежит одной из последовательностей второго уровня. Тогда исходная последовательность разбивается на три подпоследовательности. Еще одна последовательность —  $\{3, 7\}$ . Аналогичная операция прибавления единицы дает числа 4 и 8. Они принадлежат одной последовательности, поэтому исходную последовательность мы оставляем без изменений. Проверить это наблюдение можно с помощью еще одного примера, приведенного на рис. 2.13 (на подчеркнутые последовательности и пунктирные линии пока не обращаем внимания).

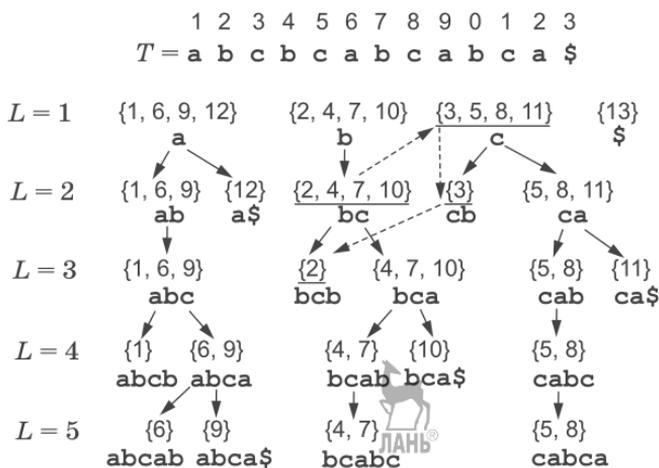


Рис. 2.13. Пример декомпозиции

## Наблюдение 2.

Снова обратимся к примеру на рис. 2.13. Пусть необходимо разбить последовательность  $\{2, 4, 7, 10\}$  на втором уровне. Эта последовательность соответствует подстроке  $bc$ , имеющей суффикс  $c$ . Последовательность, соответствующая суффиксу  $c$  на первом уровне, была разбита на две подпоследовательности:  $\{3\}$  и  $\{5, 8, 11\}$ . Одна из них — корот-

кая, а другая — длинная. Поэтому, во-первых, можно утверждать, что последовательность индексов  $\{2, 4, 7, 10\}$  тоже будет разбита на две подпоследовательности при переходе на третий уровень, а во-вторых, что это разбиение можно получить, используя только короткую подпоследовательность  $\{3\}$  (малую), уже имеющуюся на втором уровне. С ее помощью получается короткая подпоследовательность  $\{2\}$ , а индексы длинной (большой) подпоследовательности —  $\{4, 7, 10\}$ , — если это необходимо, просто переписываются из исходной последовательности. Разумеется, допустимо получать и подпоследовательность  $\{4, 7, 10\}$  с использованием последовательности  $\{5, 8, 11\}$ , а «в остатке» иметь  $\{2\}$ , но это — более затратный вариант.

В декомпозиции последовательности уровня  $L$  на подпоследовательности уровня  $L + 1$  одну подпоследовательность с наибольшим количеством элементов мы определим как *большую* и состоящую из большего количества индексов, а остальные подпоследовательности — как *малые* (*малые индексы*). Для первого уровня все последовательности состоят из малых индексов.

Основная идея алгоритма М. Крочемора заключается в том, чтобы на каждом уровне выполнять декомпозицию только малых индексов. При использовании только их (при условии, что обработка каждого требует константного времени) весь текст подвергается декомпозиции за время  $O(n \cdot \log_2(n))$ .

Действительно, подсчитаем, сколько раз за всю декомпозицию индекс  $i$  будет принадлежать малой последовательности, т. е. будет малым. Пусть на уровне  $L = 1$  он принадлежит малой последовательности, в которой содержится  $n$  индексов. В наихудшем случае последовательность, которой принадлежит  $i$ , будет разбита на две равные последовательности, и индекс  $i$  попадает в одну из них. Отсюда следует, что индекс  $i$  не входит в больше чем  $\log_2(n) + 1$  малых последовательностей. Всего у нас  $n$  индексов, следовательно, в малых последовательностях на всех уровнях содержится  $O(n \cdot \log_2(n))$  индексов.

Общий вид алгоритма М. Крочемора:

$L := 1;$

<вычисление всех последовательностей первого уровня>;

**While** <есть малые индексы на уровне  $L$ > **Do Begin**

<вывод кратных строк с периодом  $L$  (при их наличии)>;

<с использованием только малых индексов проведение декомпозиции уровня  $L$  в уровень  $L+1$ >;

$L := L+1$ ;

<вычисление малых индексов уровня  $L$ >;

**End**;

Показанная формализация логики оговаривает только общее управление вычислительным процессом. Получение временной оценки  $O(n \cdot \log_2(n))$  зависит от искусства синтеза структур данных и управляющей логики того, что называют программой. В данном алгоритме это нетривиальная задача, частично она вынесена в упражнения, но в целом это предмет достойного исследования и проверки уровня профессионализма читателя!



### Упражнения

1. Дана строка  $S$  из  $n$  символов (алфавит  $A$  упорядочен, каждый символ имеет индекс). За время  $O(n)$  подсчитайте количество вхождений каждого символа в строку  $S$ .
2. Разработайте алгоритм разбивки текста на уникальные подстроки с временной сложностью  $O(n^2)$ .
3. Приведите пример текста. Выполните его декомпозицию с использованием только последовательностей индексов на каждом уровне (к символам текста можно обращаться только на первом уровне). Проверьте результат.
4. Приведите пример текста. Выполните его декомпозицию так, как это указано в предыдущем упражнении, а также с учетом принципа малых индексов.
5. Для вывода кратных строк с периодом  $L$  необходимо в каждой последовательности индексов уровня  $L$  найти группы идущих подряд индексов с периодом  $L$ . Например, на рис. 2.6 при  $L = 1$  это 1, 2 ( $a^2$ ) из первой последовательности и 5, 6, 7 ( $b^3$ ) из второй, а при  $L = 2$  — 2, 4 ( $ab^2$ ). Предложите структуру данных для хранения индексов последовательностей и разработайте программу вывода кратных подстрок.
6. Предположим, что для хранения последовательностей на каждом уровне используется массив  $sq$  вида

Array[1..n] Of Record num, mark End (для примера на рис. 2.6 при  $L = 1$  и  $L = 2$  он приведен в табл. 2.22). В поле *num* хранится индекс символа, а в поле *mark* — номер последовательности.

Таблица 2.22

$L = 1$	<i>num</i>	1	2	3	4	5	6	7	8	9	10
	<i>mark</i>	1	1	2	1	2	2	2	1	2	3
$L = 2$	<i>num</i>	1	2	3	4	5	6	7	8	9	–
	<i>mark</i>	1	2	3	2	4	4	3	2	5	–

Возможна ли при выборе такой структуры данных реализация первого наблюдения за время  $O(n)$ ?

7. Предположим, что для хранения последовательностей на каждом уровне используется массив *sq* из записей вида:

Record

<метка последовательности (*mark*)>;  
 <количество элементов в последовательности>;  
 <указатель на список пар вида (индекс элемента (*num*), связь по индексу), где "связь по индексу" – это метка последовательности, к которой принадлежит следующий индекс –  $mark(num+1)$ >;

**End.**

Возможна ли при выборе описанной структуры данных реализация второго наблюдения за время  $O(n)$ ?

8. Для связи индексов последовательностей одного уровня использовалась операция прибавления единицы. Возможно ли использование для этого операции вычитания единицы?
9. При переходе от уровня к уровню появляются неиспользуемые индексы. Какую структуру данных следует выбрать для их хранения?
10. Какую структуру данных следует выбрать для хранения малых индексов?
11. Разработайте полную программную реализацию алгоритма М. Крочемора.

## 2.7. Алгоритм М. Мейна – Р. Лоренца

...делить каждую из рассматриваемых мною трудностей на столько частей, насколько потребуется, чтобы лучше их разрешить...

*Рене Декарт.  
Рассуждение о методе*

*Алгоритм М. Мейна – Р. Лоренца* предназначен для вычисления всех квадратов подстрок строки  $S$  ( $|S| = n$ ).

### Пример

$S = \text{babaabab}$ . Разобьем  $S$  на две равные подстроки (половины):  $u = \text{baba}$  и  $v = \text{abab}$ , т. е.  $S = uv$ . Тогда в  $S$  существуют квадраты трех типов:

- полностью находящиеся в  $u$  –  $(ba)^2$ ;
- полностью находящиеся в  $v$  –  $(ab)^2$ ;
- начинающиеся в  $u$  и заканчивающиеся в  $v$  –  $(aba)^2$ ;  $(a)^2$ .

### Идея алгоритма.

Вычислим все квадраты строки  $S$  путем рекурсивной разбивки строки  $S$  пополам и вычисления квадратов третьего типа. (Очевидно, что в этом случае определяются квадраты и первых двух типов.) Если при этом шаг рекурсии будет выполняться за линейное время, то общее время работы такого алгоритма будет иметь оценку  $O(n \cdot \log_2 n)$  — аналогично бинарному поиску.

Запишем сказанное в другом виде.

```

Procedure Solve( $S$ :String); { $n$ =Length( $S$ )}
Begin
  If  $n > 1$  Then Begin
    <вычисление квадратов третьего типа
    для  $u=S[1..[n/2]]$  и  $v=S[[n/2]+1..n]$ >;
    Solve( $u$ );
    Solve( $v$ );
  End;
End;

```

Введем два новых понятия: *квадраты правого* и *левого вида* (рис. 2.14).

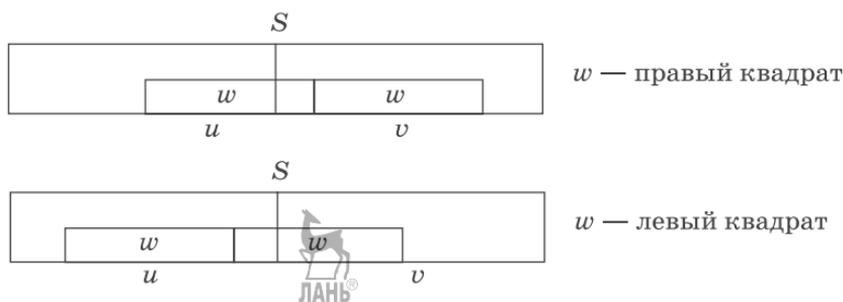


Рис. 2.14. Иллюстрация понятий правого и левого квадратов

Пусть  $w^2$  — квадрат с образующей  $w$ , а  $u$  и  $v$  — подстроки строки  $S$  ( $u = S[1..\lfloor n/2 \rfloor]$ ,  $v = S[\lfloor n/2 \rfloor + 1..n]$ ). Тогда *правый квадрат* — это подстрока  $w^2$ , в которой второе вхождение  $w$  в строку  $S$  полностью находится в подстроке  $v$ . *Левый квадрат* — это подстрока  $w^2$ , в которой второе вхождение  $w$  в строку  $S$  имеет непустой префикс в подстроке  $u$ .

*Пример*

$S = ababaa$ ,  $u = aba$ ,  $v = baa$ . Имеются правый квадрат  $(ba)^2$  и левый —  $(ab)^2$ .

**Первая схема рассуждений.**

Разберемся со структурой правого квадрата (рис. 2.15). Пусть дана позиция  $i$  в  $v$  ( $i = 1..n - \lfloor n/2 \rfloor$ ). Является ли эта позиция окончанием квадрата  $w^2$  (где  $w$  — длина квадрата)? Предположим, что да, и сначала вычислим значения индексов, определяющих вхождение  $w$  в  $S$ .

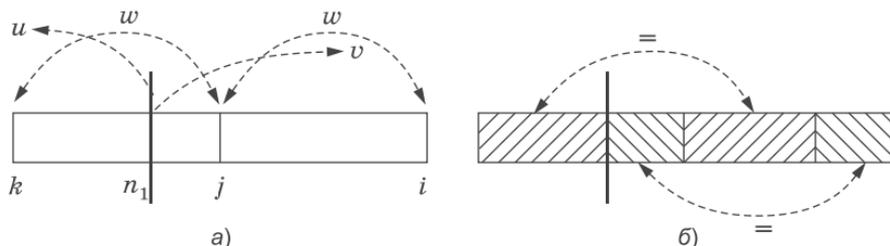


Рис. 2.15. Структура правого квадрата

Правое вхождение  $w$  начинается с позиции  $j + 1$  в  $v$ , где  $j = i - w$  (рис. 2.15а). Левое вхождение  $w$  начинается с пози-

ции  $k$  в  $u$ :  $k = n_1 + j - w + 1 = n_1 + i - w - w + 1 = n_1 - (2w - i - 1)$ . Чтобы в позиции  $i$  заканчивался квадрат длиной  $w$ , требуется выполнение равенства:  $u[k..n_1]v[1..j] = v[j + 1..i]$ , а оно разбивается на две части (рис. 2.15б):

- $v[1..j] = v[i - j + 1..i]$  или  $v[1..i - w] = v[w + 1..i]$ ;
- $u[k..n_1] = v[j + 1..i - j]$  или  $u[n_1 - (2w - i - 1)..n_1] = v[i - w + 1..w]$ .

Первое равенство — это не что иное, как грань строки  $v$  в позиции  $i$  (см. п. 1.2.1), которая вычисляется за линейное время. (Напомним, что грань  $v[1..i]$  — это длина наибольшего префикса строки, совпадающего с ее суффиксом.) Итак, нам необходимо вычислить массив граней для  $v$  (обозначим его как  $br_v$ ). Вторая же часть равенства требует знания общих суффиксов строки  $u$  и подстроки  $v$  (обозначим массив общих суффиксов как  $cs_v$ ), т. е.  $cs_v[t]$  — это длина наибольшего суффикса строки  $u$  и подстроки  $v[1..t]$ . Предположим, что за линейное время вычисляется не только массив граней  $br_v$ , но и  $cs_v$ . Как на основе этой информации определить наличие правых квадратов?

### Примеры

В табл. 2.23 и 2.24 даны строки  $S = uv$ , для которых подсчитаны значения элементов массивов  $br_v$  и  $cs_v$ .

Таблица 2.23

u												v											
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4
a	b	c	a	b	d	a	b	c	a	b	d	a	b	c	a	b	d	a	b	c	a	b	c
<i>i</i>												1	2	3	4	5	6	7	8	9	0	1	2
$br_v$												0	0	0	1	2	0	1	2	3	4	5	3
$cs_v$												0	0	0	0	0	6	0	0	0	0	0	0

Таблица 2.24

u						v						
1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	a	b	a	b	a	a	b	a	b	a
<i>i</i>						1	2	3	4	5	6	7
$br_v$						0	0	0	1	2	1	2
$cs_v$						0	2	1	0	5	0	3

Выскажем предположение: чтобы правый квадрат заканчивался в позиции  $i$ , необходимо выполнение условия  $2br_v[i] + cs_v[i - br_v[i]] \geq i$ .

Для первой строки имеем шестикратное выполнение этого условия:

- $2br_v[6] + cs_v[6 - br_v[6]] = 2 \cdot 0 + 6 = 6 \geq 6$  — квадрат `abcabdabcabd`;
- $2br_v[7] + cs_v[7 - br_v[7]] = 2 \cdot 1 + 6 = 8 \geq 7$  — квадрат `bcabdabcabda`;
- $2br_v[8] + cs_v[8 - br_v[8]] = 2 \cdot 2 + 6 = 10 \geq 8$  — квадрат `cabdabcabdab`;
- $2br_v[9] + cs_v[9 - br_v[9]] = 2 \cdot 3 + 6 = 12 \geq 9$  — квадрат `abdabcabdabc`;
- $2br_v[10] + cs_v[10 - br_v[10]] = 2 \cdot 4 + 6 = 14 \geq 10$  — квадрат `bdabcabdabca`;
- $2br_v[11] + cs_v[11 - br_v[11]] = 2 \cdot 5 + 6 = 16 \geq 11$  — квадрат `dabcabdabcab`.

Для второй строки указанное выше условие выполняется четыре раза:

- $2br_v[2] + cs_v[2 - br_v[2]] = 2 \cdot 0 + 2 = 2 \geq 2$  — квадрат `baba`;
- $2br_v[5] + cs_v[5 - br_v[5]] = 2 \cdot 2 + 1 = 5 \geq 5$  — квадрат `abaaba`;
- $2br_v[6] + cs_v[6 - br_v[6]] = 2 \cdot 1 + 5 = 7 \geq 6$  — квадрат `aababaabab`;
- $2br_v[7] + cs_v[7 - br_v[7]] = 2 \cdot 2 + 5 = 9 \geq 7$  — квадрат `ababaababa`.

Разберемся теперь со структурой левого квадрата (рис. 2.16). Пусть дана позиция  $i$  в  $u$  ( $i = 1..n_1, n_1 = n \text{ Div } 2$ ). Является ли эта позиция началом квадрата  $w^2$  (где  $w$  — длина квадрата), заканчивающегося в подстроке  $v$  (рис. 2.16а)? Опять-таки предположим, что да, и сначала вычислим значения индексов, определяющих вхождение  $w$  в  $S$ .

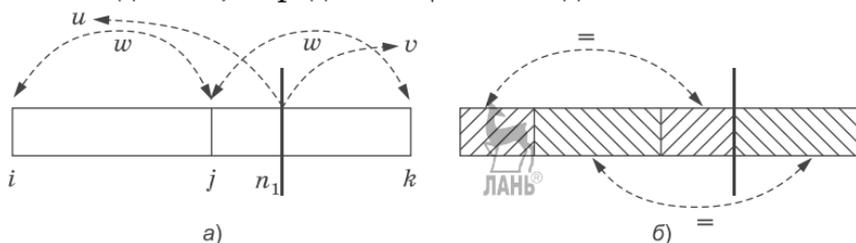


Рис. 2.16. Структура левого квадрата

Правое вхождение  $w$  начинается с позиции  $j$  в  $u$ , где  $j = i + w$  (рис. 2.16а), заканчивается в позиции  $k$  в  $v$ , где  $k = j + w - 1 - n_1 = i + 2w - 1 - n_1$ . Чтобы в позиции  $i$  начинался квадрат длины  $w$ , требуется выполнение равенства:  $u[i..j-1] = u[j..n_1]v[1..k]$ , а оно разбивается на две части (рис. 2.16б):

- $u[i..j-k-1] = u[j..n_1]$  или  $u[i..i+w-i-2w+1+n_1-1] = u[i..n_1-w] = u[i+w..n_1]$ ;
- $u[i+w-k..j-1] = v[1..k]$  или  $u[n_1-w+1..i+w-1] = v[1..i+2w-1-n_1]$ .

Известно (см. п. 1.2.1), что грани можно вычислять относительно не только префиксов, но и суффиксов строки. В п. 1.2.1 массив граней суффиксов (мы обозначали его как  $bw$ ) определяется за линейное время, а первая часть равенства — это не что иное, как массив граней суффиксов подстроки  $u$  (обозначим его как  $bw_u$ ). Вторая часть равенства требует знания общих префиксов строки  $v$  и подстрок  $u$  (обозначим массив общих префиксов как  $cp_u$ ), т. е.  $cp_u[t]$  — это длина наибольшего префикса строки  $v$  и подстроки  $u[t..n_1]$ . Предположим, что за линейное время вычисляется не только массив граней  $bw_u$ , но и  $cp_u$ .

### Примеры

В табл. 2.25 и 2.26 даны строки  $S = uv$ , для которых подсчитаны значения элементов массивов  $bw_u$  и  $cp_u$ .

Таблица 2.25

u												v											
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4
a	b	a	b	c	a	b	a	d	a	b	c	a	b	a	d	a	a	b	c	a	b	c	a
1	2	3	4	5	6	7	8	9	0	1	2	$i$											
0	0	3	2	1	0	0	0	0	0	0	0	$bw_u$											
3	0	2	0	0	5	0	1	0	2	0	0	$cp_u$											

Таблица 2.26

u						v						
1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	a	b	a	b	a	a	b	a	b	a
1	2	3	4	5	6	$i$						
1	2	1	1	0	0	$bw_u$						
0	5	0	0	2	0	$cp_u$						

Выскажем предположение: чтобы левый квадрат начался в позиции  $i$ , необходимо выполнение условия  $2bw_u[i] + cp_u[i + bw_u[i]] \geq n_1 - i + 1$ .

Для первой строки мы имеем двукратное выполнение этого условия:

- $2bw_u[3] + cp_u[3 + bw_u[3]] = 2 \cdot 3 + 5 = 11 \geq 12 - 3 + 1 = 10$  — квадрат `abcbadabcbad`;
- $2bw_u[4] + cp_u[4 + bw_u[4]] = 2 \cdot 2 + 5 = 9 \geq 12 - 4 + 1 = 9$  — квадрат `bcbadabcbada`.

А вот при  $i = 5$  условие уже не выполняется:  
 $2bw_u[5] + cp_u[5 + bw_u[5]] = 2 \cdot 1 + 5 = 7 \geq 12 - 5 + 1 = 8$ .

Для второй строки (см. табл. 2.26):

- $2bw_u[1] + cp_u[1 + bw_u[1]] = 2 \cdot 1 + 5 = 7 \geq 6 - 1 + 1 = 6$  — квадрат `abaababaab`;
- $2bw_u[4] + cp_u[4 + bw_u[4]] = 2 \cdot 1 + 2 = 4 \geq 6 - 4 + 1 = 3$  — квадрат `abab`;
- $2bw_u[5] + cp_u[5 + bw_u[5]] = 2 \cdot 0 + 2 = 2 \geq 6 - 5 + 1 = 2$  — квадрат `baba`.

*Примечание.* Перед первым символом  $u$  добавляется несуществующий в алфавите символ.

Если исходить из верности принятых нами предположений, то для завершения обсуждения необходимо понимание того, как за линейное время вычислять значения элементов массивов  $cs_v$  — длин наибольших суффиксов строки  $u$  и подстрок  $v[1..i]$  ( $i = 1..n_2$ ), и  $cp_u$  — длин наибольших префиксов строки  $v$  и подстроки  $u[i..n_1]$  ( $i = 1..n_1$ ).

Начнем с последнего массива (рис. 2.17). Предположим, что для некоторого значения  $i$  вычислен общий префикс, и он заканчивается в позиции  $t$ , т. е. известно значение  $cp_u[i]$ . Структура этого префикса определяется значением массива граней  $br_v$  в позиции  $t - i + 1$ , что позволяет, во-первых, вычислять следующим значение  $cp_u[q]$  (все остальные сдвиги не имеют смысла), а во-вторых (и это главное!), выполнять сравнение символов  $u[t + 1]$  и  $v[br_v[t - i + 1] + 1]$ . Таким образом, возвратов по строке  $u$  нет, что дает предпосылки для получения линейного алгоритма.

Так что принцип использования  $br_v$  для получения  $cp_u$  уже просматривается, осталось проработать детали.

Аналогичная ситуация — и с массивом  $cs_v$  — длинами наибольших суффиксов строки  $u$  и подстрок  $v[1..i]$

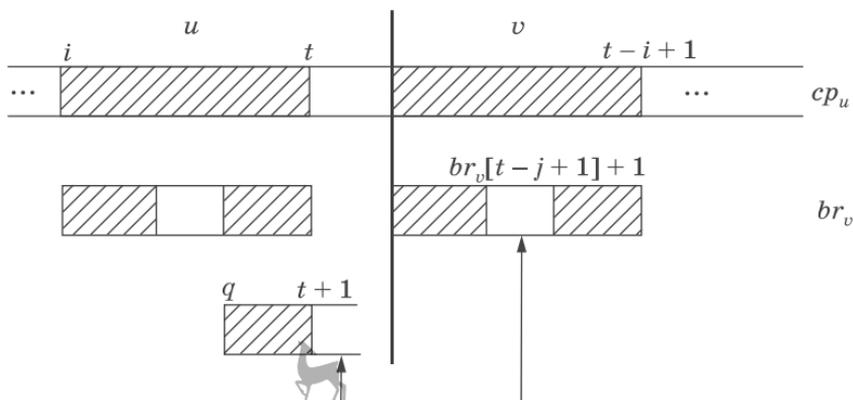


Рис. 2.17. Принцип вычисления массива  $cp_u$

( $i = 1..n_2$ ). Она приведена на рис. 2.18; при этом используется уже вычисленный массив  $bw_u$  — грани суффиксов подстроки  $u$ . Предположим, что вычислено значение  $cs_v[i]$ , и общий суффикс заканчивается в позиции  $t$ . Структура суффикса определена соответствующим элементом массива  $bw_u$ , что позволяет перейти к вычислению значения  $cs_v[q]$  и выполнять сравнение элементов  $v[t-1]$  и  $u[n_1 - bw_u[n_1 - (i-t)]]$ .

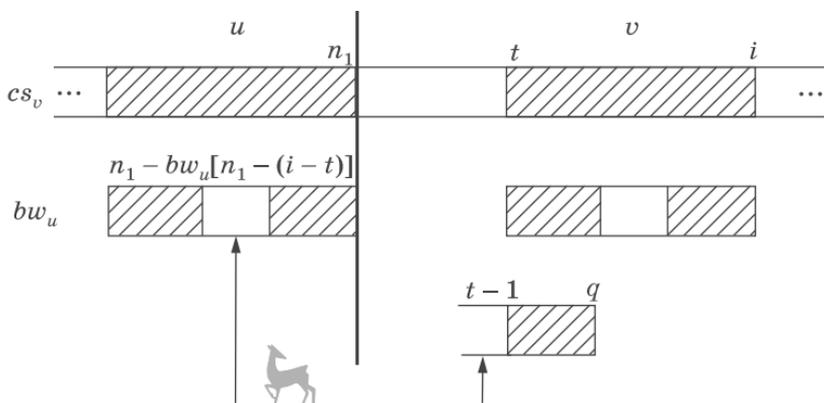


Рис. 2.18. Принцип вычисления массива  $cs_v$

### Вторая схема рассуждений.

Казалось бы, логика решения задачи уже выстроена. Однако вспомним, что следует все подвергать сомнению — это один из принципов работы специалиста по информатике, — уж очень (в данном случае) проста получаемая «конструкция» для достаточно сложной проблемы!

## Пример

В табл. 2.27 представлены данные еще по одной строке, для которой требуется найти все квадраты. Проверим наличие правых квадратов, — а хотя бы один таковой у нас явно есть — `abcdcabcbcdcabс`.

Таблица 2.27

u									v								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	a	a	b	c	d	c	a	b	c	a	b	c	d	c	a	b	c
<i>i</i>									1	2	3	4	5	6	7	8	9
<i>br<sub>v</sub></i>									0	0	0	1	0	1	2	3	4
<i>cs<sub>v</sub></i>									0	0	3	0	0	0	0	7	0

Проверим наше предположение относительно правых квадратов:  $2br_v[i] + cs_v[i - br_v[i]] \geq i$ . Для  $i = 9$ :  $2 \cdot br_v[9] + cs_v[9 - br_v[9]] = 2 \cdot 4 + 0 = 8$ , — это условие не выполняется, но для  $2 \cdot br_v[br_v[9]] + cs_v[9 - br_v[br_v[9]]] = 2 \cdot 1 + 7 = 9 \geq 9$  оно верно! Получается, что следует проверять не только грань, заканчивающуюся в позиции  $i$ , но и ее грани, затем — грани этих граней, и так пока мы не дойдем до нулевого значения. Аналогичный контрпример можно найти и для левых квадратов.

Вернемся к анализу правого квадрата (см. рис. 2.15). Выполнение равенства  $u[k..n_1]v[1..j] = v[j + 1..i]$  необходимо. Но равенство  $v[1..j] = v[i - j + 1..i]$  можно записать как  $v[1..j] = v[w + 1..i]$ , а это почти формула для блока строки  $v$  в позиции  $w + 1$  (в формуле блока — длина максимальной подстроки, совпадающей с префиксом строки). Попробуем вычислить не грани строки  $v$ , а блоки — массив  $bl_v$  (см. п. 1.2.2). Пусть массивы  $bl_v$  и  $cs_v$  вычислены. Чтобы квадрат длиной  $w$  заканчивался в позиции  $i$  строки  $v$ , необходимо выполнение неравенств  $bl_v[w + 1] \geq l_2$  и  $cs_v[w] \geq l_1$ , иначе в квадрате будет «разрыв» (рис. 2.19), и подстрока не будет кратной. На основе этих неравенств появляется возможность определить границы  $i$ , в которых может заканчиваться квадрат.

Здесь в строке  $S = uv$  имеется квадрат длиной  $2w$ , заканчивающийся в позиции  $i$  подстроки  $v$  при условии, что

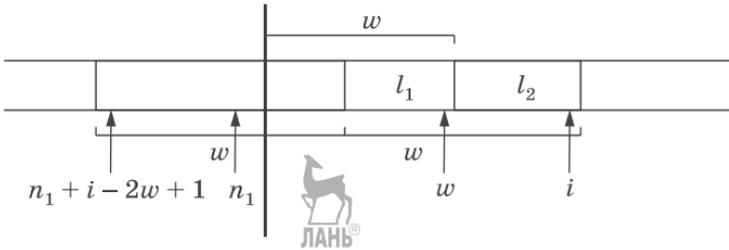


Рис. 2.19. Правый квадрат

$2w - cs_v[w] \leq i \leq w + bl_v[w + 1]$ . Действительно, чтобы в позиции  $i$  заканчивался квадрат длиной  $w$ , требуется выполнение равенства:  $u[k..n_1]v[1..j] = v[j + 1..i]$ , а оно разбивается на две части:

- $u[n_1 - (2w - i - 1)..n_1] = v[i - w + 1..w]$ ;
- $v[1..i - w] = v[w + 1..i]$ ; при  $i = w$  квадрат формируется из суффикса  $u$  и префикса  $v$ , и эта часть «вырождается».

Первая часть данного равенства говорит о том, что строка  $v[i - w + 1..w]$  длиной  $2w - i$  является суффиксом строки  $u$ , и оно выполняется только тогда, когда  $cs_v[w] \geq 2w - i$ , т. е.  $2w - cs_v[w] \leq i$ . Аналогично, для второй части равенства: строка  $v[w + 1..i]$  длиной  $i - w$  является блоком строки  $v$ , откуда следует, что  $bl_v[w + 1] \geq i - w$ , т. е.  $i \leq w + bl_v[w + 1]$ .

Итак, для каждого допустимого значения  $w \in 1..n_2$  соответствующие значения  $i$  могут находиться только в интервале  $2w - cs_v[w]..w + bl_v[w + 1]$ . Поэтому при вычисленных массивах  $bl_v$  и  $cs_v$  фактические значения  $i$  определяются за константное время путем вывода границ этих интервалов при выполнении условия  $bl_v[w + 1] + cs_v[w] \geq w$ .

Рассмотрим теперь левый квадрат (см. рис. 2.16). Для него необходимо выполнение равенства  $u[i..i + w - 1] = u[i + w..n_1]v[1..i + 2w - 1 - n_1]$ , которое разбивается на две части:

- $u[i..n_1 - w] = u[n_1 - (i + w - 1) + 1..n_1]$ ;
- $u[(n_1 - w) + 1..n_1 - (i + w - 1)] = v[1..i + 2w - 1 - n_1]$ .

Будем считать, что массивы  $bw_u$  и  $cr_u$  вычислены. Массив  $bw_u$  дает наибольший общий суффикс подстроки  $u[1..i]$  и строки  $u$ , т. е. значение  $bw_u[n_1 - w]$  (рис. 2.20) должно быть больше  $l_1$ . Массив же  $cr_u$  дает наибольший общий префикс

подстроки  $u[i..n_1]$  и строки  $v$ , т. е. значение  $cp_u[n_1 - w + 1]$  должно быть больше  $l_2$  (рис. 2.20).

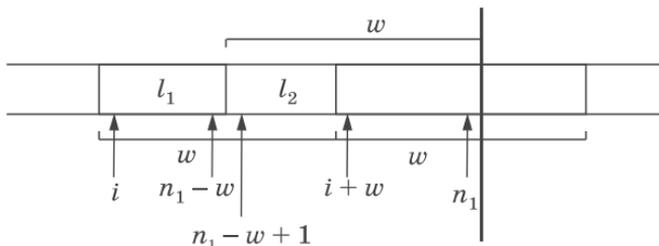


Рис. 2.20. Левый квадрат

Чтобы в строке  $S = uv$  имелся квадрат длиной  $2w$ , начинающийся с позиции  $i$  подстроки  $u$ , необходимо выполнение условия  $(n_1 - w) - bw_u[n_1 - w] \leq i - 1 \leq (n_1 - 2w) + cp_u[n_1 - w + 1]$ . Действительно, первая часть равенства говорит о том, что строка  $u[i..n_1 - w]$  длиной  $n_1 - (i + w - 1)$  является суффиксом строки  $u$ , т. е.  $bw_u[n_1 - w] \geq n_1 - (i + w - 1)$  или  $(n_1 - w) - bw_u[n_1 - w] \leq i - 1$ . Вторая же часть равенства требует (для строки  $u[(n_1 - w) + 1..n_1 - (i + w - 1)]$  длиной  $i + 2w - 1 - n_1$ ) выполнения условия  $cp_u[n_1 - w + 1] \geq i + 2w - 1 - n_1$ , или  $i - 1 \leq (n_1 - 2w) + cp_u[n_1 - w + 1]$ . Следовательно, для каждого допустимого значения  $w \in 1..n_1$  соответствующие значения  $i$  могут находиться только в интервале от  $(n_1 - w) - bw_u[n_1 - w]$  до  $(n_1 - 2w) + cp_u[n_1 - w + 1]$ . Поэтому при вычисленных массивах  $bw_u$  и  $cp_u$  фактические значения  $i$  определяются за константное время путем вывода границ этих интервалов при выполнении условия  $cp_u[n_1 - w + 1] + bw_u[n_1 - w] \geq w$ .

На этом мы завершим обсуждение алгоритма, вынеся оставшуюся часть (в том числе вычисление массивов  $cs_v$ ,  $cp_u$  и  $bw_u$  на основе массива  $bl_v$ ) в упражнения.

### Упражнения

1. Приведите пример строки  $S$ . Выпишите все ее квадраты в той очередности, в которой они будут выведены при рекурсивной реализации алгоритма М. Мейна – Р. Лоренца.

2. Модифицируйте процедуры вычисления массивов граний префиксов и суффиксов ( $br_v$ ,  $bw_u$ ), приведенные в п. 1.2.1, для их использования в алгоритме М. Мейна – Р. Лоренца. Можно ли довести первый способ рассуждений до алгоритма с линейной временной оценкой работы  $O(n)$ ?
3. Для строки  $S$  вычислите значения элементов массивов  $cs_v$  и  $cp_u$  вручную и с использованием  $br_v$ ,  $bw_u$ . Сравните результаты.
4. Приведите контрпример, показывающий некорректность определения левого квадрата при первом способе рассуждений.
5. Проверьте правильность вычисления массива блоков строки с помощью следующей процедуры. Оцените время ее работы.

```

Procedure Bloc (s:String);
  Var i, k, l:Byte;
  Begin
    s:=s+'$';
    blv[2]:=0;
    While (s[blv[2]+1]=s[blv[2]+2]) Do
      blv[2]:=blv[2]+1;
    k:=2;
    For i:=3 To Length(s)+1 Do Begin
      l:=k+blv[k]+i;
      If blv[i-k+1] < 1 Then blv[i]:=blv[i-k+1]
        Else blv[i]:=max(0, l);
      While (s[blv[i]+1]=s[blv[i]+i]) Do
        blv[i]:=blv[i]+1;
      k:=i;
    End;
  End;

```

6. На основании приведенной процедуры  $Cpu$  определите логику формирования массива  $cp_u$  — длин наибольших префиксов строки  $v$  и подстрок  $u[t..n_1]$  ( $t = 2..n_1$ ).

```

Procedure Cpu (v, u:String);
  Var i, k, l:Byte;
  Begin
    v:=v+'$'; u:=u+'@';

```

```

cpu[2]:=0;
While (v[cpu[2]+1]=u[cpu[2]+2]) Do
    cpu[2]:=cpu[2]+1;
k:=2;
For i:=3 To Length(u) Do Begin
    l:=k+cpu[k]-i;
    If blv[i-k+1]<l Then cpu[i]:=blv[i-k+1]
    Else cpu[i]:=max(0,l);
    While (v[cpu[i]+1]=u[cpu[i]+i]) Do
        cpu[i]:=cpu[i]+1;
    k:= i;
End;
End;

```

7. С помощью процедуры *Bwu* вычисляются значения элементов массива  $bw_u$  — грани суффиксов подстроки  $u$ . Проверьте ее работоспособность. Сравните данную логику вычисления с приведенной в п. 1.2.1.

```

Procedure Bwu(s:String);
Var i, k, l, n1:Byte;
Begin
    s:='@'+s;
    n1:=Length(s);
    While s[n1-bwu[(n1-1)-1]]=
        s[(n1-bwu[(n1-1)-1])-1] Do
        bwu[(n1-1)-1]:=bwu[(n1-1)-1]+1;
    k:=n1-1;
    For i:=n1-2 DownTo 1 Do Begin
        l:=bwu[k-1]-(k-i);
        If bwu[(n1-(k-i)+1)-1]<l Then
            bwu[i-1]:=bwu[(n1-(k-i)+1)-1]
        Else bwu[i-1]:=max(0,l);
        While (s[n1-bwu[i-1]]=s[i-bwu[i-1]]) Do
            bwu[i-1]:=bwu[i-1]+1;
        k:=i;
    End;
End;

```

8. Определите по процедуре *Csv* логику формирования массива  $cs_v$  — длин наибольших общих суффиксов строки  $u$  и подстрок  $v[1..t]$  ( $t = 1..n_2$ ).

```

Procedure Csv(u,v:String);
  Var i, k, l, n1, n2:Byte;
  Begin
    v:='$'+v;
    u: '@'+u;
    n1:=Length(u);
    n2:=Length(v);
    While u[n1-csv[n2-1]]=v[n2-csv[n2-1]] Do
      csv[n2-1]:=csv[n2-1]+1;
    k:=n2;
    For i:=n2-1 DownTo 1 Do Begin
      l:=csv[k-1]-(k-i);
      If bwu[n1-(k-i)+1]<l Then
        csv[i-1]:=bwu[n1-(k-i)+1]
      Else csv[i-1]:=max(0,l);
      While (u[n1-csv[i-1]]=v[i-csv[i-1]]) Do
        csv[i-1]:=csv[i-1]+1;
      k:=i;
    End;
  End;

```

9. Приведенный ниже фрагмент программного кода обеспечивает вывод квадратов. Значение переменной с равно 1 либо  $n_1 + 1$  (в зависимости от того, вызывается логика с  $u$  или с  $v$ ). Улучшите этот программный код.

```

w:=1;
While w<=n2 Do Begin
  If (blv[w+1]+csv[w]>=w) Then Begin
    WriteLn ('(',w,',',',',((2*w-csv[w])+n1+c-1)-
      2*w+1,',',',',((w+blv[w+1])+n1+c-1)-
      2*w+1,',') R');
    For i:=((2*w-csv[w])+n1+c-1)-2*w+1 To
      ((w+blv[w+1])+n1+c-1)-2*w+1 Do
      WriteLn ('Правый квадрат ',Copy(S,i,2*w),
        ' с длиной=',w,' и началом
        в позиции ',i);
  End;
  If (cpu[(n1-w)+1]+bwu[n1-w]>=w) And (w<n1)
  Then Begin
    If (((n1-w)-bwu[n1-w])+1)=
      ((n1-2*w)+cpu[(n1-w)+1]+1)
  End;

```

```

Then WriteLn (' ', w, ' ', ' ', ((n1-w)-bwu[n1-w])
              +1)+c-1, ' ', ' ', ((n1-2*w)+
              cпу[(n1-w)+1]+1)+c-1, ' ') L')
Else WriteLn (' ', w, ' ', ' ', ((n1-w)-bwu[n1-w])
              +1)+c-1, ' ', ' ', ((n1-2*w)+bwu[(n1
              -p)+1]+1)+c-2, ' ') L');
For i:=(((n1-w)-bwu[n1-w])+1)+c-1 To
      ((n1-2*w)+cпу[(n1-w)+1]+1)+c-2 Do
  WriteLn ('Левый квадрат ', Copy(S, i, 2*w) ,
           ' с длиной=', w, ' и началом
           в позиции ', i);
End;
w:=w+1;
End;

```

10. Разработайте полную программную реализацию алгоритма М. Мейна – Р. Лоренца.

### Методический комментарий

Алгоритмы Д. Кнута (*Donald Knuth*) – Дж. Морриса (*James Morris*) – В. Пратта (*Vaughan Pratt*)<sup>1)</sup> и Р. Бойера (*Robert Boyer*) – Дж. Мура (*James Moore*)<sup>2)</sup> являются классикой данного раздела информатики (*computer science*). В той или иной мере они затрагиваются во многих учебниках по информатике при изучении темы поиска данных, включая классическую работу Н. Вирта (*Niklaus Wirth*)<sup>3)</sup>.

Р. Карп (*Richard Karp*) и М. Рабин (*Michael Rabin*)<sup>4)</sup> опубликовали свой метод поиска образца в тексте в 1987 г. В работе Д. Гасфилда (с. 108), однако, утверждается, что он изобретен десятилетием ранее. Если алгоритмы Д. Кнута – Дж. Морриса – В. Пратта и Р. Бойера – Дж. Мура, а также их многочисленные модификации, сделанные после 1977 г., основаны на сравнении символов и сдвигах, то в данном методе заложен совершенно другой принцип — «арифметический». Образец и строка текста преобразуются в числа, и происходит сравнение не символов, а чисел. При этом все определяется эффективностью преобразования подстроки в число.

<sup>1)</sup> Knuth D. E., Morris J. H., Pratt V. R. Fast pattern matching in strings // SIAM J. Comput. 6–2, 1977. P. 323–350.

<sup>2)</sup> Boyer R. S., Moore J. S. A fast string searching algorithm // CACM, 20–10, 1977. P. 762–772.

<sup>3)</sup> Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989.

<sup>4)</sup> Karp R. M., Rabin M. O. Efficient randomized pattern – matching algorithms // IBM J. Res. Develop. 31–2, 1987. P. 85–103.

Р. Беза-Йетс (*Ricardo Baeza-Yates*) и Г. Гоннет (*Gaston Gonnet*)<sup>1)</sup> в 1992 г. предложили простой битовый метод (алгоритм *Shift-And*), который очень эффективно решает задачу точного поиска для относительно малых образцов (например, длиной в типичное английское слово). Однако, как указано в работе Б. Смита (*Bill Smith*) (с. 240), первоначально этот алгоритм еще в 1968 г. был описан Б. Дёмёлки (*Balint Domolki*)<sup>2)</sup>.

Формализм теории автоматов является достаточно конструктивным инструментом описания алгоритмов обработки строк. В данной главе затронуты только первичные понятия этой хорошо разработанной теории. Ссылки на источники не приводятся, ибо при необходимости их нетрудно найти самостоятельно.

Алгоритм М. Крочемора (*Maxime Crochemore*)<sup>3)</sup> является одним из основных в задаче поиска кратных строк. Если идея данного алгоритма понимается достаточно просто, то этого нельзя сказать о его программной реализации, которая (в полном варианте и с учетом тестирования), безусловно, служит предметом проверки профессионального уровня читателя. Причем обращение к книге Б. Смита не прояснит ситуацию с программной реализацией, а скорее запутает ее. (*Примечание.* Приводить в данной книге полную реализацию программы и ее разбор автор не стал, так как это влечет значительное увеличение объема книги и вряд ли целесообразно, исходя из поставленных целей.)

Алгоритм же М. Мейна (*Michael Main*) и Р. Лоренца (*Richard Lorentz*)<sup>4)</sup> интересен не только с точки зрения использования классического принципа «разделяй и властвуй» в решении задач, но и эффективным применением ранее рассмотренных методов предварительного анализа строк.

---

<sup>1)</sup> *Baeza-Yates R. A., Gonzaio N. H.* A new approach to text searching // *CACM*, 35–70, 1992. P. 74–82.

<sup>2)</sup> *Domolki B.* A universal computer system based on production rules // *BITS*, 1968. P. 262–275.

<sup>3)</sup> *Crochemore M.* An optimal algorithm for computing all the repetitions in a word // *IPL*, 12–5, 1981. P. 244–248.

<sup>4)</sup> *Main M. G., Lorentz R. J.* An  $O(n \cdot \log n)$  algorithm for finding all repetitions in a string // *J. Algs.*, 5, 1984. P. 422–432.

## Деревья суффиксов

---

Деревья принадлежат к числу самых лучших вещей, какие я только знаю. Вода, деревья и девушки.

*Эрленд Лу*



Знания о структуре образца (из каких частей, например, повторяющихся фрагментов и как они связаны между собой) позволяют эффективно решать задачу его поиска в тексте. Этот вывод вполне естественен после изучения алгоритмов из второй главы. Теперь же мы рассмотрим новую структуру данных — *дерево суффиксов*, которое строится на стадии предварительной обработки для текста, а не для образца, и за линейное время (иначе смысл его построения теряется). После построения дерева суффиксов задача определения вхождения образца в текст также решается за линейное время.

### 3.1. Основные понятия. Простые алгоритмы построения дерева суффиксов

В окне звезда, деревья за окном,  
Как стражники, мой охраняют дом.

*Борис Рыжий*

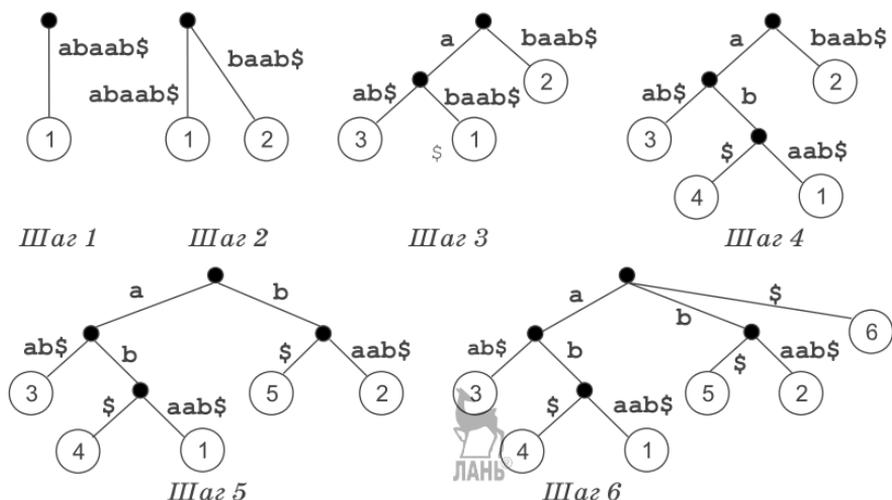
*Дерево суффиксов* строки  $S$  длины  $n$  — это не что иное, как способ компактного описания множества ее суффиксов. Это дерево является ориентированным; вся его специфика заключается в том, как оно строится. Строка  $S$  дополняется *терминальным символом* (обычно  $\$$ ), которого, по предположению, нет в алфавите  $A$ . Дерево суффиксов имеет корень,  $n + 1$  листьев (конечных вершин, помечаемых целыми числами от 1 до  $n + 1$ ) и не более  $n$  внутренних вершин.



тем осуществляем дальнейшие посимвольные сравнения  $P$  с метками дуг. При этом возможны три варианта. Первый — если произошло несовпадение, в этом случае  $P$  нет в  $T$ . Во втором же и в третьем случаях сравнение  $P$  выполнено успешно (все символы  $P$  совпали), но закончилось оно, соответственно, или в вершине (последний символ  $P$  совпал с последним символом очередной метки дуги), или на каком-то символе метки дуги. Эти два последних случая практически идентичны, ибо в подобных ситуациях нам известна вершина дерева. Если она — лист, то имеет место единственное вхождение  $P$  в  $T$ . Если же это внутренняя вершина, то имеет место множественное вхождение  $P$  в  $T$ , и тогда фрагмент дерева, начиная с этой вершины, обходится методом поиска в глубину, чтобы найти метки листьев, указывающие на начальные позиции вхождений  $P$  в  $T$ . При известном (построенном) дереве суффиксов время поиска  $P$  имеет очевидную оценку  $O(m)$  (либо  $O(m + k)$  при множественном вхождении, где  $k$  — количество вхождений).

**Простой алгоритм построения дерева суффиксов** (иллюстрация дана на рис. 3.2).

Пусть дана строка  $S$ . Дополним ее символом  $\$,$  отсутствующим в алфавите  $A$ . На первом шаге поместим в дерево дугу с меткой  $S\$$  ( $S[1..n]\$$ ), а затем будем последовательно вставлять суффиксы  $S[i..n]\$$ .



**Рис. 3.2.** Пример работы простого алгоритма построения дерева суффиксов для строки  $abaab\$$

Переход от дерева  $T_i$  к дереву  $T_{i+1}$  при этом выполняется так. Есть суффикс  $S[i + 1..n]\$$ . Его символы последовательно сравниваются с метками дуг, начиная с дуг, выходящих из корня. По совпадениям символов будет пройдена часть дерева (в том числе, может быть, и не будет пройдено ни одного символа, как на шаге 2, — рис. 3.2). Этот пройденный путь — единственный, в силу того что дуги, выходящие из любой вершины дерева, имеют метки, начинающиеся с различных символов, и произошло несовпадение символов в какой-то позиции  $q$  суффикса. Причем факт такого несовпадения обязателен, ибо каждый суффикс уникален — он заканчивается символом  $\$$ , отсутствующим в алфавите. При этом создается новая вершина с двумя сыновьями; к одному из сыновей идет дуга с оставшейся частью суффикса  $S[q..n]\$$ , а к другому — с оставшейся частью метки дуги. Если же несовпадение произошло в вершине дерева, то эта вершина получает одного нового сына. В любом случае создается новая вершина — лист с меткой  $i$  (номером суффикса).

*Пример (рис. 3.2)*

Обрабатывается суффикс  $aab\$$  (шаг 3). Символ  $a$  совпадает с первым символом метки дуги  $abaab\$$ , но для второго символа  $a$  нет совпадения. Создается новая внутренняя вершина, где к первому сыну идет дуга с оставшейся частью метки  $baab\$$ , а ко второму сыну (листу с меткой 3) идет дуга с оставшейся частью суффикса —  $ab\$$ .

Рассмотрим особенности реализации этого алгоритма. Описание данных (оно минимально) имеет вид:

**Type**

```

PList = ^node;
node = Record
  cnt: Integer;
  {Количество сыновей у вершины дерева}
  d: Array[|A|] Of Record
  {Предполагаем, что алфавит конечен,
   есть отношение порядка}
  first, last: Integer;
  {Номера первого и последнего символов
   подстроки - метки дуги}
  next: PList;
  {Указатель на вершину - сына}

```

```

num: Integer;
{Метка листа. Для внутренней вершины, включая
корень, она имеет значение 0}
End;

```

End;

Var

```

S: String;
n: Integer;
tree: PList;

```

В каждой вершине дерева мы вынуждены вводить массив  $d$ , поскольку максимальное количество дуг, выходящих из вершины, равно мощности алфавита  $A$ . Описание строки  $S$ , такое, что ограничение на ее размер определяется системой программирования, взято здесь для простоты.

Приведем один из возможных способов реализации рассмотренного алгоритма.

**Procedure** TreeSuf1;

```

Var w, v: PList;
    i, j, l, t: Integer;
    bl: Boolean;

```

**Begin**

```

S:=S+'$';
n:=Length(S);

```

{Первый шаг алгоритма - в корне дерева дается описание дуги, идущей в вершину с меткой "1"}  
st:=1; {Номер листа}

```

New(tree);
tree^.cnt:=1;
tree^.d[1].first:=1;
tree^.d[1].last:=n;
tree^.d[1].next:=Nil;
tree^.d[1].num:=st;

```

{Суффикс  $S[i..n]$  описываем (представляем) в дереве}

**For** i:=2 **To** n **Do Begin**

```

w:=tree;
t:=i;
bl:=False;

```

**While Not** bl **Do Begin**

{Пока не сформирован лист, соответствующий суффиксу, выполняем действия цикла}

```

j:=1;
l:=0;
While (l=0) And (j<=w^.cnt) Do Begin
{Просматриваем дуги, выходящие из вершины,
 на которую "смотрит" w^}
    If S[w^.d[j].first]=S[t] Then l:=j;
{Дуга найдена - первый символ суффикса совпадает
 с первым символом метки дуги}
    j:=j+1;
End;
If l=0 Then Begin
{Дуга не найдена - создаем новую вершину-лист,
 отцом которой является вершина w^}
    w^.cnt:=w^.cnt+1;
    st:=st+1;
    bl:=True; {Суффикс обработан}
    w^.d[w^.cnt].next:=Nil;
    w^.d[w^.cnt].num:=st;
    w^.d[w^.cnt].first:=t;
    w^.d[w^.cnt].last:=n;
End
Else Begin
{Дуга найдена - сравниваем символы суффикса
 и метки дуги}
    j:=1;
    While (w^.d[l].first+j<=w^.d[l].last) And
        (S[t+j]=S[w^.d[l].first+j]) Do
        j:=j+1;
    If w^.d[l].first+j<=w^.d[l].last Then
        Begin
        {Создаем новую внутреннюю вершину}
        st:=st+1;
        New (v);
        v^.cnt:=2; {У вершины два сына}
        v^.d[1].next:=w^.d[l].next;
{Новая метка дуги - совпадающая часть суффикса
 и старой метки дуги}
        v^.d[1].num:=w^.d[l].num;
        v^.d[1].first:=w^.d[l].first+j;
        v^.d[1].last:=w^.d[l].last;
        v^.d[2].next:=Nil;

```

```

{Оставшаяся часть суффикса является меткой дуги,
 идущей к листу с номером обрабатываемого суффикса}
    v^.d[2].num:=st;
    v^.d[2].first:=t+j;
    v^.d[2].last:=n;
{Корректируем описание вершины - отца}
    w^.d[1].next:=v;
    w^.d[1].last:=w^.d[1].first+j-1;
    w^.d[1].num:=0;
    bl:=True; {Суффикс обработан}
End
Else Begin
{Переходим к следующей вершине дерева - часть
 суффикса полностью совпала с меткой дуги}
    t:=t+w^.d[1].last-w^.d[1].first+1;
    w:=w^.d[1].next;
End;
End; {Конец обработки найденной дуги}
End; {Конец цикла по обработке суффикса}
End; {Конец цикла по суффиксам}
End;

```

Временная сложность алгоритма  $O(n^2)$  — это в предположении, что алфавит конечен, поэтому время обработки символа (или выбора дуги, выходящей из вершины) имеет постоянное значение (является константой). Действительно, мы обрабатываем  $n$  суффиксов, и для обработки каждого суффикса требуется выполнить количество операций, пропорциональное его длине.

А теперь поставим перед собой другую задачу — можно ли создать дерево суффиксов, «перевернув» схему построения этого дерева (описание вершин дерева оставляем прежним)? То есть будем начинать не с полной строки, а с ее первого символа и, последовательно расширяя дерево, получим тот же результат. Другими словами, начнем с символа  $S[1]$ ; затем, изменяя дерево, опишем все суффиксы  $S[1..2]$ , затем все суффиксы  $S[1..3]$  и т. д., вплоть до суффиксов  $S[1..n]$ \$. Схема такого построения для строки  $abaab\$$  показана на рис. 3.3.

Приведем еще один пример. Для строки  $abcabdabcabdab\$$  процесс построения дерева суффиксов

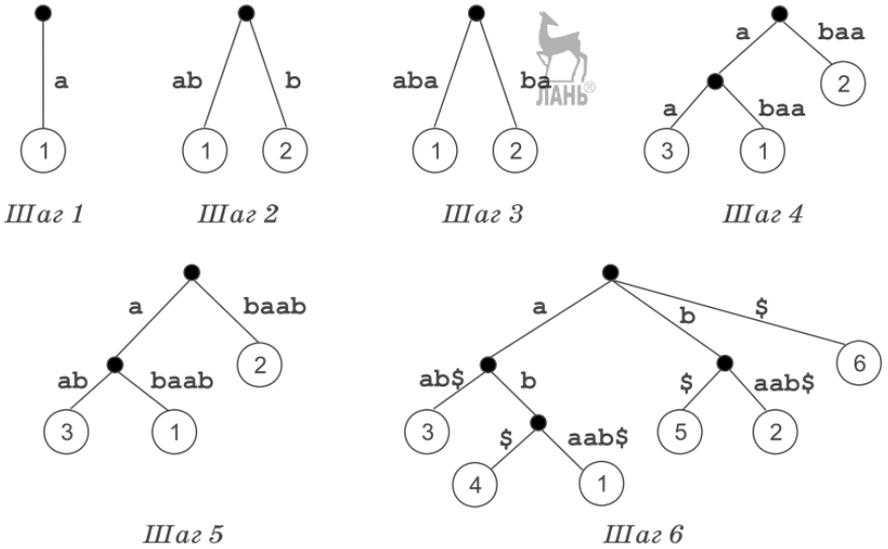


Рис. 3.3. Пример второго способа реализации алгоритма построения дерева суффиксов для строки abaab\$

приведен на рис. 3.4. На последнем шаге под номером 15 (его нет на рис. 3.4) получается дерево суффиксов, показанное ранее на рис. 3.1.

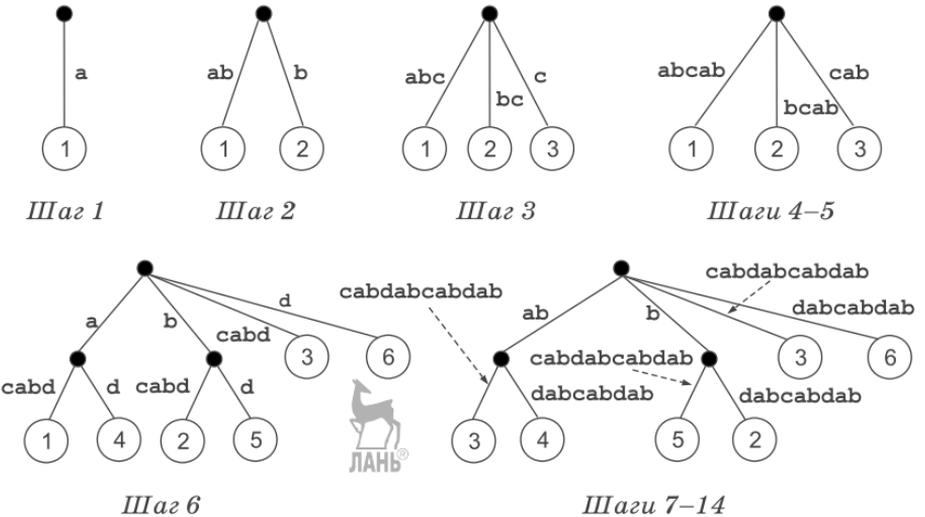


Рис. 3.4. Иллюстрация второго способа построения дерева суффиксов для строки abcabdabcbadab\$

В чем же заключаются особенности построения дерева суффиксов по данной схеме? Обратимся к последнему примеру.

*Шаг 1.* Создается лист с номером 1 и дуга, идущая из корня с меткой  $a$ .

*Шаг 2.* Подстрока  $ab$  — ее суффиксы  $ab$  и  $b$ . Первый суффикс описывается путем изменения метки  $a$  на  $ab$ . Вторым суффикс — создается лист с номером 2 и меткой дуги  $b$ .

*Шаг 3.* Подстрока  $abc$  — ее суффиксы  $abc$ ,  $bc$ ,  $c$ . Обработка первых двух суффиксов сводится к изменению меток дуг:  $ab$  на  $abc$  и  $b$  на  $bc$ . Третий суффикс — создается лист с номером 3 и меткой дуги  $c$ .

*Шаг 4.* Подстрока  $abca$  — ее суффиксы  $abca$ ,  $bca$ ,  $ca$ ,  $a$ . Первые три суффикса вводятся в дерево путем приписывания (назовем это так) символа  $a$  к уже существующим меткам дуг. Четвертый суффикс  $a$  составляет часть метки дуги  $abca$ , т. е. он уже представлен в дереве.

*Шаг 5.* Подстрока  $abcab$  — ее суффиксы  $abcab$ ,  $bcab$ ,  $cab$ ,  $ab$ ,  $b$ . Первые три суффикса тоже обрабатываются путем приписывания символа  $b$  к уже существующим меткам дуг, а последние два суффикса уже представлены в дереве.

*Шаг 6.* Подстрока  $abcabd$  — ее суффиксы  $abcabd$ ,  $bcabd$ ,  $cabd$ ,  $abd$ ,  $bd$ ,  $d$ . Для первых трех суффиксов действия аналогичны тем, что выполняются на шаге 5, а для последних трех создаются новые внутренние вершины и вершины — листья с соответствующими номерами 4, 5, 6.

*Шаг 7.* Подстрока  $abcabda$ . Первые шесть ее суффиксов обрабатываются путем приписывания символа  $a$  к соответствующим меткам дуг, а седьмой суффикс представлен в дереве как часть метки дуги, идущей к листу с номером 1.

*Шаги 8–14.* Аналогичны шагу 7. Первые шесть суффиксов обрабатываются путем приписывания символов, а остальные суффиксы на каждом шаге уже представлены в дереве.

*Шаг 15.* Полная строка  $S\$$ . Первые шесть суффиксов обрабатываются путем приписывания, а остальные девять суффиксов — путем создания новых внутренних вершин и вершин — листьев с соответствующими номерами 7, 8, ..., 15.

В итоге мы и получаем дерево суффиксов, представленное на рис. 3.1.

Итак, в этой схеме явно просматриваются три типа действий. Назовем действия по приписыванию символа к метке дуги *правилом 1*, по созданию новых вершин дерева — *правилом 2*, а по обнаружению факта присутствия суффикса в дереве (когда он есть в неявном виде — без вершины листа) — *правилом 3*. В табл. 3.2 представлены последовательности применения названных правил на каждом шаге построения дерева суффиксов для строки  $abcabdabcabdab\$.$

Таблица 3.2

Номер символа	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Шаг	a	b	c	a	b	d	a	b	c	a	b	d	a	b	§
1	2														
2	1	2													
3	1	1	2												
4	1	1	1	3											
5	1	1	1	3	3										
6	1	1	1	2	2	2									
7	1	1	1	1	1	1	3								
8	1	1	1	1	1	1	3	3							
9	1	1	1	1	1	1	3	3	3						
10	1	1	1	1	1	1	3	3	3	3					
11	1	1	1	1	1	1	3	3	3	3	3				
12	1	1	1	1	1	1	3	3	3	3	3	3			
13	1	1	1	1	1	1	3	3	3	3	3	3	3		
14	1	1	1	1	1	1	3	3	3	3	3	3	3	3	
15	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2

Какие выводы (или предположения для дальнейшего обоснования) можно сделать из анализа этого примера?

1. Если на каком-либо шаге при обработке очередного суффикса  $S[j..i]$  сработало правило 3, то оно будет работать и дальше (на этом шаге) при обработке суффиксов  $S[j + 1..i]$ ,  $S[j + 2..i]$ , ...,  $S[i..i]$ . Это следует из структуры дерева суффиксов. Действительно, в дереве уже представлены суффиксы  $S[j..i - 1]$ ,  $S[j + 1..i - 1]$ , ...,  $S[i - 1..i - 1]$ . Если найдено продолжение суффикса  $S[j..i - 1]$  символом  $S[i]$ , или, други-

ми словами, наличие в дереве суффикса  $S[j..i]$ , то очевидно, что будут найдены продолжения и остальных суффиксов символом  $S[i]$ . Правило 3 не требует выполнения никаких операций по обработке, и после его первого срабатывания этот шаг построения дерева можно заканчивать.

Естественно, что при реализации алгоритма следует ввести некую логическую переменную (например,  $bl$ ), для фиксации факта срабатывания правила 3 и по ее значению выходить из данной итерации обработки шага, т. е. переходить к следующему значению  $i$ .

2. Пусть к моменту выполнения шага  $i$  уже было создано  $q$  листов. Тогда на шаге  $i$  первые  $q$  суффиксов обрабатываются по правилу 1, а оно сводится к изменению поля  $last$  (его увеличению на единицу) у  $q$  соответствующих меток дуг.

А если не изменять это поле? Если при создании листа ввести признак того, что он — лист, тем более что такой признак уже есть в описании вершины — это поле  $num$  (внутренние вершины дерева, включая корень, не имеют номеров), то на каждом шаге обработку суффиксов можно будет начинать не с первого —  $S[1..i]$ , — а с  $(q + 1)$ -го, т. е. с суффикса  $S[q + 1..i]$ . Реализация этого положения требует учета при поиске конца очередного суффикса того, что метка дуги, идущей к листу, не прописана явно.

3. Количество срабатываний правила 2 равно  $n$ . Его распределение по шагам (цифры над отрезками) в рассматриваемом выше примере обработки строки  $abcabdabcabdab\$$  приведено на рис. 3.5.

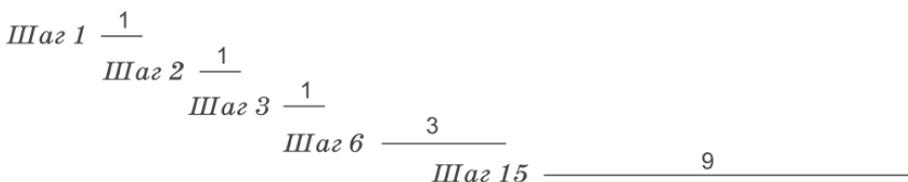


Рис. 3.5. Срабатывание правила 2 при построении дерева суффиксов

Правило 2 заключается в создании новой внутренней вершины и вершины — листа с номером, определяющим начало обрабатываемого суффикса в строке. Если найдено требуемое место в дереве (конец суффикса), то реализация этого правила требует константного времени (оно не зависит от длины строки).

Время построения дерева суффиксов по данному методу имеет оценку  $O(n^3)$ . Действительно, здесь имеется циклическая конструкция по шагам ( $i$ ), где в теле цикла обрабатывается  $i$  суффиксов, и на обработку каждого суффикса  $j$  требуется  $O(i - j)$  операций. Правила 1 и 3 позволяют избежать обработки части суффиксов, но в целом они не изменяют временную оценку. Следовательно, необходим эффективный способ определения окончания суффикса в дереве и метод обработки каждого суффикса (когда локализовано его место в дереве). В этом случае у данного алгоритма появится возможность перейти в разряд линейных по времени.

Один из возможных способов реализации вышеприведенной схемы построения дерева суффиксов (без правила 1) приводится ниже (процедура TreeSuf2). Она максимально приближена к предыдущей версии построения дерева суффиксов. Наиболее трудный момент — выявление правил при анализе суффикса, поэтому вначале приведем его словесное описание:

```

If <суффикс закончился> Then <правило 3>
  Else If <метка дуги не закончилась> Then <правило 2>
    Else If <лист> Then <правило 1>
      Else <переадресация по дереву>;

```

```

Procedure TreeSuf2;
  Var w,v: PList;
      i,j,l,t,k:Integer;
      bl,bl1:Boolean;
  Begin
    S:=S+'$';
    n:=Length(S);
    st:=1;
    New(tree); {Создаем первую вершину}
    tree^.cnt:=1;
    tree^.d[1].first:=1;
    tree^.d[1].last:=1;
    tree^.d[1].next:=Nil;
    tree^.d[1].num:=st;
    For i:=2 To n Do Begin
  {Обрабатываем строки S[1..i] - изменяем дерево
  суффиксов}
    k:=1;

```

{При реализации правила 1 значение  $k$  начинает изменяться не с единицы}

**bl:=False;**

{Признак того, что следующие суффиксы для больших значений  $k$  обрабатывать не следует (правило 3)}

**While (k<=i) And (Not bl) Do Begin**

{Обрабатываем суффикс  $S[k..i]$ }

w:=tree;

t:=k;

**bl1:=False;**

{Признак обработки суффикса}

**While Not bl1 Do Begin**

j:=1;

l:=0;

{Поиск дуги, начинающейся с символа  $S[t]$ }

**While (l=0) And (j<=w^.cnt) Do Begin**

**If**  $S[w^.d[j].first]=S[t]$  **Then** l:=j;

j:=j+1;

**End;**

**If** l=0 **Then Begin**

{Дуга не найдена – формируем новую вершину-сына у  $w^$  (правило 2)}

**bl1:=True;** {Суффикс обработан}

$w^.cnt:=w^.cnt+1$ ;

st:=st+1;

$w^.d[w^.cnt].next:=Nil$ ;

$w^.d[w^.cnt].num:=st$ ;

$w^.d[w^.cnt].first:=t$ ;

$w^.d[w^.cnt].last:=i$ ;

**End**

**Else Begin**

j:=0;

{Дуга найдена. Сравниваем символы суффикса и метки дуги}

**While**  $(w^.d[l].first+j<=w^.d[l].last)$

**And**  $((t+j)<=i)$  **And**  $(S[t+j]=$

$S[w^.d[l].first+j])$

**Do** j:=j+1;

**If** t+j=i+1 **Then Begin**

{Сравнение завершилось полным исчерпанием суффикса (правило 3). Суффикс обработан;

более того, обработка следующих суффиксов для данного значения  $i$  лишена смысла - они уже представлены в дереве суффиксов}

```

    b1:=True;
    b11:=True;
End {3}
Else If w^.d[l].first+j<=w^.d[l].last
Then Begin

```

{Если метка дуги не закончилась, то надо создать новую внутреннюю вершину (правило 2)}

```

    st:=st+1;
New(v);
    v^.cnt:=2;
    v^.d[1].next:=w^.d[l].next;
    v^.d[1].num:=w^.d[l].num;
    v^.d[1].first:=w^.d[l].first+j;
    v^.d[1].last:=w^.d[l].last;
    v^.d[2].next:=Nil;
    v^.d[2].num:=st;
    v^.d[2].first:=t+j;
    v^.d[2].last:=i;
    w^.d[l].next:=v;
    w^.d[l].last:=w^.d[l].first+j-1;
    w^.d[l].num:=0;
    b11:=True;

```

**End**

```

Else If w^.d[l].num<>0 Then Begin

```

{Метка дуги закончилась, а суффикс - нет. Значит, мы имеем дело с вершиной - листом. В этом случае добавляется символ к метке дуги (правило 1)}

```

    w^.d[l].last:=w^.d[l].last+1;
    b11:=True;

```

**End**

```

Else Begin

```

{Не лист - переадресация по дереву}

```

    t:=t+w^.d[l].last-w^.d[l].first+1;
    w:=w^.d[l].next;

```

**End;**

**End;**

**End;**

k:=k+1;



*Примечание.* При этом возможны два способа реализации этого правила.

1. Если дуга идет к вершине-листу, то сравнение в цикле по шагам ( $i$ ) осуществляется до символа с номером  $i$ .

2. К вершинам-листьям добавляется метка « $\infty$ ». Если вершина-лист  $j$  приходится сыном некоей вершины  $v$ , путь до которой имеет метку  $S[j..h]$ , то дуга от вершины  $v$  до листа  $j$  помечается как  $(h + 1, \infty)$ , что указывает на суффикс  $S[h + 1..n]$ .

## 3.2. Алгоритм Э. Укконена

На высокую башню можно подняться лишь по винтовой лестнице.

Фрэнсис Бэкон

Алгоритм создания дерева суффиксов, предложенный Э. Укконеном, основан на втором простом способе его построения (см. п. 3.1). На первом шаге строится дерево  $T_1$  из корня и одной дуги, выходящей из него в лист с номером 1, с меткой дуги — символом  $S[1]$ . На втором шаге  $T_1$  преобразуется в  $T_2$ , содержащее описание всех суффиксов подстроки  $S[1..2]$ . На третьем шаге из  $T_2$  мы получаем  $T_3$  для подстроки  $S[1..3]$  и т. д. — до получения дерева  $T_{n+1}$  для строки  $S[1..n]$ §. При этом при переходе от дерева  $T_{i-1}$  к  $T_i$  (где  $i$  — номер шага) используются правила 1, 2 и 3, которые требуют константного времени. Поэтому, чтобы у нас была надежда на получение линейного по времени алгоритма построения дерева суффиксов, нам необходимо за постоянное время искать концы подстрок (суффиксов) в дереве.

Пусть  $u$  — некоторая вершина дерева суффиксов. Метка пути до  $u$  от корня или какого-либо предка  $u$  получается склейкой меток дуг.

Определим *суффиксную связь*  $s$  (это функция; при ее реализации в описании каждой вершины появляется поле для хранения адреса другой вершины дерева) для любой вершины  $u$  дерева следующим образом:

- для корня дерева  $s(u) = u$ ;
- для любой вершины дерева  $u$  с меткой пути от корня  $xt$ , где  $x$  — первый символ метки, а  $t$  — ее оставшаяся часть, существует вершина  $v$  с меткой пути, равной  $t$ , то  $s(u) = v$ .

## Пример

Дана строка, представленная в табл. 3.3.

Таблица 3.3

Номер позиции	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b	\$

Построим для нее дерево с суффиксными связями. Первые 5 шагов приводят к дереву, показанному в левой части рис. 3.6. Пока здесь имеется только одна суффиксная связь из корня в корень. (Суффиксные связи от листьев изображать не будем, чтобы не перегрузить рисунок — они очевидны и идут от одного листа к другому: от первого листа ко второму, от второго к третьему и т. д.)

Рассмотрим шаг 6. В нем обрабатывается подстрока bacbab. Первые три суффикса попадают под правило 1, т. е. фактически не обрабатываются. Остались суффиксы bab, ab, b. Начинаем с суффикса bab.

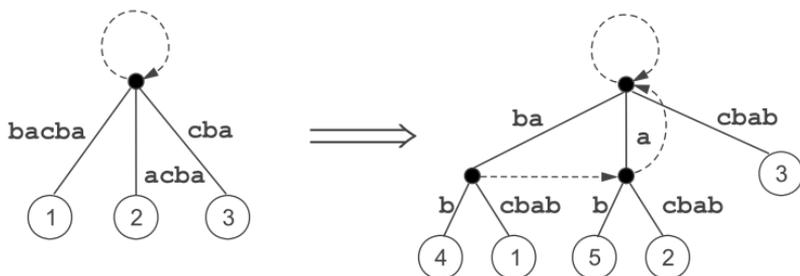


Рис. 3.6. Шаг 6 построения дерева суффиксов

Создается внутренняя вершина  $u$ , из которой исходит дуга в лист с меткой 4. При обработке следующего суффикса  $ab$  (именно на этом шаге!) вновь создается внутренняя вершина  $v$  и оказывается, что  $s(u) = v$ . Последний суффикс  $b$  обрабатывается по правилу 3, а суффиксная связь (такие связи на рисунках изображаются пунктирными линиями) из  $v$  идет в корень дерева.

Пока нам еще не очень понятно, что дают суффиксные связи, но отложим это мини-открытие до рассмотрения следующих шагов.

Шаги 7, 8, 9 не дают новых листьев, т. е. не приводят к срабатыванию правила 2. На каждом шаге здесь проверяются суффиксы, начинающиеся с шестой позиции (ba, bab, baba), срабатывает правило 3 и действия для данного шага заканчиваются.

Шаг 10. Обработке подлежат суффиксы babaa, abaa, baa, aa, a, и мы начинаем шаг именно с шестого суффикса. Последовательность изменений дерева при переходе от обработки одного суффикса к обработке другого приведена на рис. 3.7.

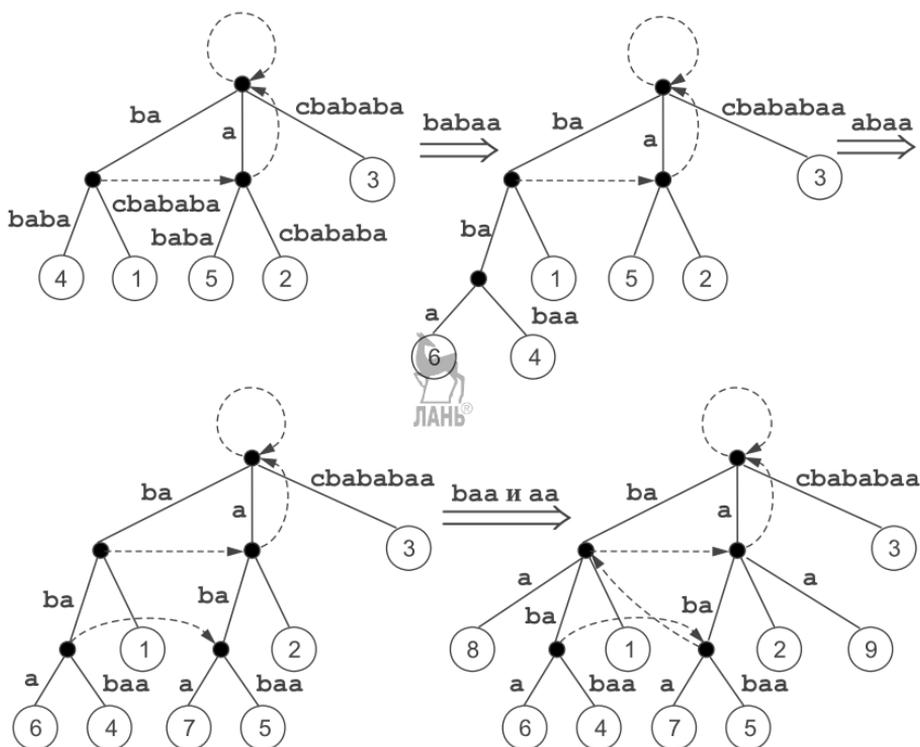


Рис. 3.7. Шаг 10 построения дерева суффиксов

Остановимся на обработке суффикса abaa. Требуется найти конец суффикса aba в дереве. При обработке предыдущего суффикса была создана внутренняя вершина. Поднимемся от нее вверх по дереву к ее отцу. Эта вершина-отец имеет суффиксную связь. Перейдем по ней (вот он, момент «эврика»!), а не будем начинать с корня дерева, как в простом алгоритме. Часть суффикса aba пройдена, и заведомо известно, что есть

оставшаяся часть. Осталось выбрать дугу, идущую из вершины, в которую мы перешли по суффиксной связи. Оставшаяся часть суффикса может быть «исчерпана» меткой найденной дуги (как в данном случае), а может быть и «не исчерпана». В первом случае срабатывает правило 2, а во втором — выполняется переход вниз по дереву до тех пор, пока суффикс не будет обработан полностью. В любом случае создаются новая внутренняя вершина и лист, и к этой новой внутренней вершине пойдет суффиксная связь от вершины, образованной при обработке предыдущего суффикса. При анализе суффиксов *baa* и *aa* новых внутренних вершин не создается (только вершины-листья), но суффиксные связи тоже определяются. В итоге каждая внутренняя вершина дерева имеет суффиксную связь.

Шаг 11 не изменяет структуры дерева — новых вершин и суффиксных связей в нем не появляется.

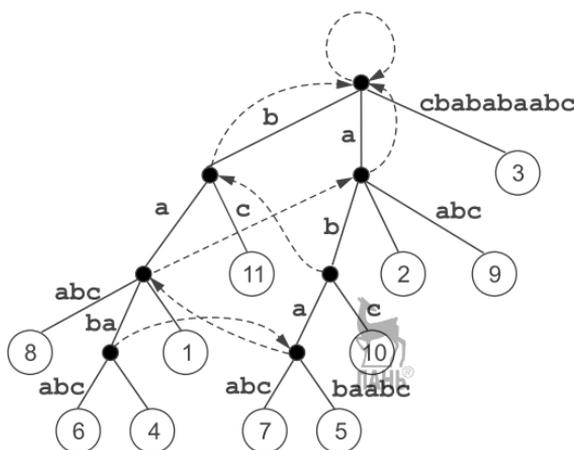


Рис. 3.8. Шаг 12 построения дерева суффиксов

В результате выполнения шага 12 мы получаем дерево, показанное на рис. 3.8. Требуется обработать суффиксы *adc*, *bc* и *c*. Обработка первых двух из них выполняется по правилу 2 и приводит к появлению новых вершин с соответствующими суффиксными связями.

*Примечание.* На некоторых дугах на рис. 3.8 метки не приводятся. Это делается, чтобы не загружать рисунок лишними деталями, тем более что фактически нам надо хранить не сами метки, а номера их первого и последнего символов.

Шаги 13, 14, 15 не приводят к изменению структуры дерева.

В результате работы шага 16 — обработки суффиксов  $cbab\$, bab\$, ab\$, b\$$  и  $\$$  — мы получаем дерево, показанное на рис. 3.9.

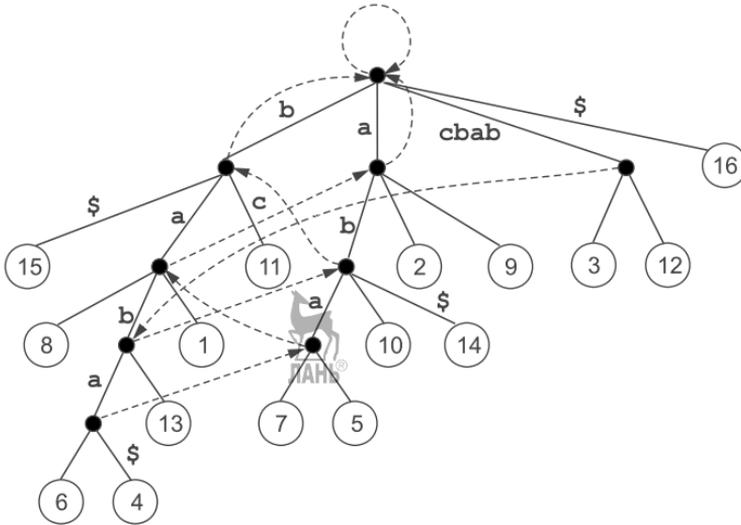


Рис. 3.9. Шаг 16 построения дерева суффиксов

Итак, любая внутренняя вершина, создаваемая в процессе построения дерева суффиксов, будет иметь суффиксную связь. Пусть мы создали вершину  $u$  (правило 2) при обработке суффикса  $S[j-1..i]$ , и метка пути (склейка меток дуг) от корня к  $u$  есть  $xt$ , где  $x$  — первый символ. Это означает, что существующий путь  $S[j-1..i-1]$  продолжался в дереве не символом  $S[i]$  (тогда сработало бы правило 3). В дереве существует путь с меткой  $t$  и он соответствует подстроке  $S[j..i-1]$ . Этот путь заканчивается или на дуге, или во внутренней вершине  $t$ . В первом случае будут созданы внутренняя вершина  $v$ , вершина-лист с номером  $j$  и меткой дуги  $S[i]$ , а также установлена суффиксная связь  $s(u) = v$ . Во втором же случае создается только вершина-лист (но это тоже правило 2!), а суффиксная связь имеет вид  $s(u) = t$ .

Суффиксные связи облегчают (являясь идейной основой) нахождение окончаний следующих суффиксов, но не очевидно, что их использование дает алгоритм с линейной временной оценкой. Дело в том, что после перехода по суф-

фиксированной связи остается посимвольное сравнение части суффикса (окончания) с меткой дуги (или дуг). И пока мы не найдем способа уйти от этого метода анализа окончания суффикса, ситуация не изменится. С этой целью используется прием, который Иосиф Владимирович Романовский в переводе книги Д. Гасфилда<sup>1)</sup> назвал «скачком по счетчику». Его смысл заключается в переходе от сравнения символов к сравнению длины меток дуг и оставшейся части суффикса, что позволяет осуществлять переходы (скачком, без анализа символов) от вершины-отца дерева к вершине-сыну до тех пор, пока не будет исчерпан суффикс.

Поясним суть операции «скачок по счетчику» с помощью примера (рис. 3.10).

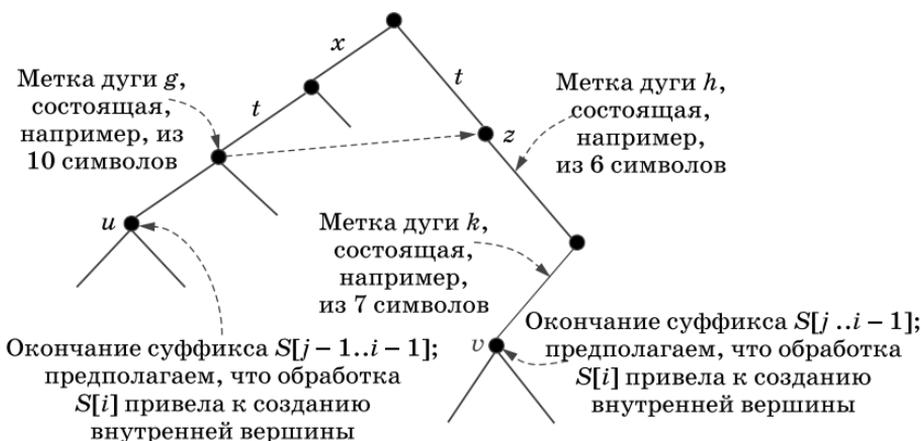


Рис. 3.10. Пример, поясняющий суть операции «скачок по счетчику»

Пусть при обработке суффикса  $S[j-1..i]$  найдено окончание подстроки  $S[j-1..i-1]$  и в результате срабатывания правила 2 создана внутренняя вершина  $u$ . Метка пути к  $u$  есть  $xtg$  (это и есть суффикс  $S[j-1..i-1]$ ). Что мы теперь делаем? Мы поднимаемся вверх по дереву от вершины  $u$  к ее вершине-отцу и переходим по суффиксной связи к вершине  $z$ . По обычной логике после выбора дуги происходит посимвольное сравнение символов метки  $h$  с оставшейся частью суффикса  $S[j..i-1]$ , — пусть это будет часть  $S[q..i-1]$  (пер-

<sup>1)</sup> Гасфилд Д. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 134.

вые символы, соответствующие метке дуги  $t$ , исключены из обработки). То есть  $S[q]$  сравнивается с  $h[1]$ ;  $S[q + 1]$  — с  $h[2]$  и т. д. При «скачке по счетчику» сравниваются длины  $S[q..i - 1]$  и  $h$ . В данном примере длина  $S[q..i - 1]$  больше, и мы переходим (делаем скачок!) к вершине-сыну  $z$ . В этой вершине выполняются аналогичные действия, но уже с оставшейся частью суффикса  $S[q + 6..i - 1]$ . Оказывается, что длина метки  $k$  больше длины  $S[q + 6..i - 1]$  — суффикс  $S[j..i - 1]$  не оканчивается в вершине дерева. Предположим, что символа  $S[i]$  нет далее в метке дуги (правило 3), тогда создается новая внутренняя вершина  $v$  и вершина-лист с номером  $j$ , а между вершинами  $u$  и  $v$  устанавливается суффиксная связь.

В результате время поиска окончания суффикса в дереве становится пропорциональным не количеству символов в суффиксе, не длине этого суффикса, а количеству пройденных вершин дерева, что и является условием получения линейного по времени алгоритма.

Введем понятие *вершинной глубины* заданной вершины, обозначающее количество вершин на пути до нее от корня, и сделаем попытку неформального обоснования линейного времени работы алгоритма Э. Укконена. Для этого необходимо оценить общее количество обработок суффиксов строки и время обработки суффикса. Правила 1 и 3 исключают обработку части суффиксов. Само правило 2 также выполняется за постоянное время при условии, что найдено окончание суффикса, поиск которого обеспечивается суффиксными связями и «скачками по счетчику». Напомним, что на каждом шаге обработка суффиксов подстроки  $S[1..i]$  начинается не с первого, а с некоторого суффикса  $q S[q..i]$ , где  $q$  — номер последнего срабатывания правила 2 (листа). Вершина-лист создается в обязательном порядке, а внутренняя вершина — в зависимости от того, заканчивается суффикс на дуге или в ранее созданной внутренней вершине. Значение  $q$  не убывает при работе алгоритма и не изменяется при переходе от шага к шагу.

Пример изменения  $q$  приведен на рис. 3.11. Каждая линия здесь обозначает шаг алгоритма, а число — значение  $q$ , т. е. обрабатываемый на данном шаге суффикс  $S[q..i]$ . Очевидно, что количество обработок суффиксов не может превосходить значения  $2 \cdot n$ .

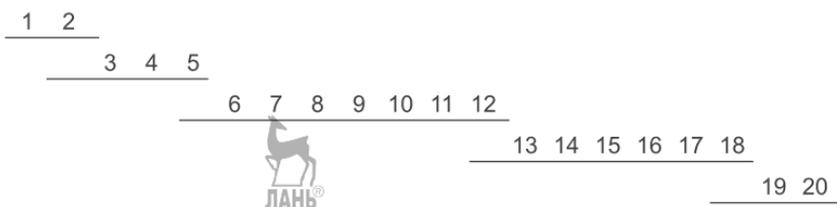


Рис. 3.11. Пример изменения значения переменной  $q$

Осталось оценить количество срабатываний операции «скачок по счетчику» в целом при исполнении алгоритма. Посмотрим, как изменяется при этом значение вершинной глубины. Каждый шаг алгоритма начинается с обработки суффикса  $q$  (это — последний обработанный суффикс на предыдущем шаге), т. е. вершинная глубина не изменяется при переходе от шага к шагу. При поиске окончания суффикса вершинная глубина вначале уменьшает свое значение не более чем на 2 (вверх по дереву на одну дугу плюс переход по суффиксной связи), а затем увеличивается на 1 при каждом «скачке». Максимальное значение вершинной глубины —  $n$ , а количество срабатываний правила 2 не превосходит  $2 \cdot n$ ; отсюда следует, что количество «скачков по счетчику», выполненных на всех шагах алгоритма, имеет порядок  $O(n)$ .

Другими словами, цикл по префиксам ( $i$ ) неизбежен. Но для всех значений  $i$  количество срабатываний правила 2 линейно, ибо перекрытие между шагами  $i$  и  $i + 1$  возможно не более чем по одному суффиксу. Из всего множества суффиксов (всех префиксов текста — порядок  $O(n^2)$ ) только для части суффиксов требуется константное время при их включении в дерево, и эта часть не превосходит значения  $2 \cdot n$ . Далее, при срабатывании правила 2 необходимо найти окончание суффикса. Зная место в дереве окончания предыдущего срабатывания правила 2, следует идти вверх по дереву на одну вершину, затем по суффиксной связи (а она уже есть в дереве) и вниз по дереву с помощью «скачка по счетчику». В итоге оказывается, что суммарное количество (при всех значениях  $i$ ) таких «прыжков» по дереву линейно. В итоге общая оценка времени есть  $O(n)$ .

Особенности реализации описываемого алгоритма вынесем в упражнения. Приведем здесь только возможное





суффиксов  $S[2..8]$  и  $S[5..8]$  на рис. 3.12 наибольший общий префикс есть строка  $ba$ , а для суффиксов  $S[2..8]$  и  $S[3..8]$  —  $\epsilon$  (пустая строка). Нетрудно заметить, что наибольший общий префикс совпадает с меткой пути в дереве до вершины, являющейся наименьшим общим предком вершин-листьев, соответствующих рассматриваемым суффиксам. Этот факт, если так можно выразиться, заложен в структуре дерева суффиксов.

Разобьем каждый суффикс  $S[i..n]$  (которому соответствует вершина-лист с номером  $i$ ) на две части: подстроку  $head[i]$  — «голову» суффикса и подстроку  $tail[i]$  — «хвост» суффикса. Значение  $head[i]$  суффикса  $S[i..n]$  мы определим как общий префикс максимальной длины, вычисленный для всех  $S[j..n]$ ,  $1 \leq j \leq i - 1$  ( $head[1]=\epsilon$ ). Значение же  $tail[i]$  — это оставшаяся часть суффикса. В табл. 3.4 приведены примеры строк и значений  $head[i]$  для них.

Таблица 3.4

Строка	abaaaabaa aab\$	abcabdabcbab dab\$	abaabababab aab\$	bacbabababab bab\$
$i$	$head[i]$	$head[i]$	$head[i]$	$head[i]$
1	$\epsilon$	$\epsilon$	$\epsilon$	$\epsilon$
2	$\epsilon$	$\epsilon$	$\epsilon$	$\epsilon$
3	a	$\epsilon$	a	$\epsilon$
4	aaa	ab	aba	ba
5	aa	b	ba	a
6	abaaaab	$\epsilon$	abaaba	baba
7	baaaaab	abcabdab	baaba	aba
8	aaaab	bcabdab	aaba	ba
9	aaab	cabdab	abaab	a
10	aab	abdab	baab	ab
11	ab	bdab	aab	b
12	b	dab	ab	cbab
13	$\epsilon$	ab	b	bab
14		b	$\epsilon$	ab
15		$\epsilon$		b
16				$\epsilon$

В чем должна состоять идея построения дерева суффиксов за линейное время? Напомним, что в описываемом способе построения дерева первым определяется суффикс  $S[1..n]$ , затем  $S[2..n]$  и т.д., вплоть до суффиксов  $S[n]$  и  $\$$ . На каждом шаге алгоритма создается лист с номером  $i$ . Если при этом создается внутренняя вершина, то метка пути к ней, как оказывается, равна  $head[i]$ , а дуга, выходящая к вершине-листу  $i$ , имеет метку  $tail[i]$ . С помощью суффиксных связей (а они здесь определяются точно так же, как и в алгоритме Э. Укконена) следует искать конец очередного суффикса начиная не с корня, а с некоторой внутренней вершины. Другими словами, нам требуется эффективный переход от суффикса  $S[i-1..n] = head[i-1]tail[i-1]$  к суффиксу  $S[i..n] = head[i]tail[i]$ , т.е. от внутренней вершины, определяемой  $head[i-1]$ , к вершине с меткой пути  $head[i]$ . Если проанализировать значения  $head$  (см. табл. 3.4; необходимо рассматривать и случаи пустой строки  $\epsilon$ ), то просматривается зависимость: если  $w = head[i-1]$  — строка из  $k$  символов ( $k \geq 0$ ), то подстрока  $w[2..k]$  является префиксом строки  $head[i]$  (а подстрока пустой строки есть пустая строка). Или, другими словами,  $head[i]$  есть (особые случаи мы исключаем) наибольший собственный суффикс  $head[i-1]$ . Но суффиксная связь  $s$ , согласно ее определению, это не что иное, как указатель от вершины дерева с меткой пути  $w$  к вершине дерева с меткой, равной наибольшему собственному суффиксу  $w$ , т.е.  $s(head[i-1]) = head[i]$ . Однако суффиксная связь устанавливается только после того, как определена вершина дерева с меткой  $head[i]$ , поэтому следует в качестве отправной точки ее поиска взять вершину  $s(parent(head[i-1]))$  — сделать переход по суффиксной связи (ранее установленной) от отца  $head[i-1]$  и от этой вершины искать окончание метки  $head[i]$ .

Попытаемся на уровне примера конкретизировать схему перехода от окончания одного суффикса к окончанию другого.

#### Пример

Построим дерево суффиксов для строки  $abaaaaabaaaab\$$ . (Шаги 3–8 показаны на рис. 3.13, а шаги 9–12 — на рис. 3.14.)

Процесс формирования дерева начинается с шага 1 — создается корень с суффиксной связью на себя самого и вершина-лист с номером 1; дуга, идущая из корня, имеет метку  $abaaaabaaaab\$.$

Шаг 2 призван обработать суффикс  $baaaabaaaab\$;$  действия при этом аналогичны.

На шаге 3 после выбора дуги, выходящей из корня, производится сравнение метки дуги и суффикса  $aaaabaaaab\$.$  На втором символе совпадения заканчиваются, создается внутренняя вершина, входящая дуга (из корня) имеет метку  $a,$  а выходящая в вершину-лист с номером 3 — метку  $aaabaaaab\$.$  Суффиксная связь идет при этом из корня в корень ( $head[i - 1]$  есть корень).

Шаг 4 аналогичен предыдущим действиям, и пока ключевая идея алгоритма у нас еще не просматривается. Суффиксная связь идет из  $head[i - 1] = a$  в корень (рис. 3.13).

Шаг 5. Логика этого шага показана на рис. 3.13. Суффиксная связь создается от вершины с меткой  $aaa$  до вновь образованной внутренней вершины. Сделаем предположение (наблюдение): от вершины  $s(parent(head[i - 1]))$  (переход по суффиксной связи вершины-отца) с меткой, равной метке дуги от  $parent(head[i - 1]),$  до вершины  $head[i - 1]$  в дереве можно сделать «скачок по счетчику». Другими словами, эта метка совпадает с префиксом суффикса  $S[i..n],$  окончание которого в дереве нам необходимо найти, и этот префикс уже представлен в дереве.

На шаге 5 от отца  $head[i - 1]$  суффиксная связь идет к корню. Из него «скачком по счетчику» мы идем с меткой  $aa.$  Операция заканчивается на дуге, что приводит к ее разбивке на две дуги, т. е. к образованию новой внутренней вершины, причем к ней пойдут суффиксная связь от  $head[i - 1],$  а из нее новая дуга с меткой  $tail[i]$  к вершине-листу с номером  $i.$

Шаг 6 подтверждает это наше наблюдение. Суффиксная связь от отца  $head[i - 1]$  опять в корень. «Скачок по счетчику» с меткой  $a$  и дальнейшая обработка суффикса  $abaaaab\$$  приводят к созданию внутренней вершины и вершины-листа с номером 6.

Шаг 7. Из корня выполняется «скачок по счетчику» с меткой  $baaaaab.$

Шаг 8. Из корня выполняется «скачок по счетчику» с меткой  $aaaab.$  При этом мы проходим три вершины. На

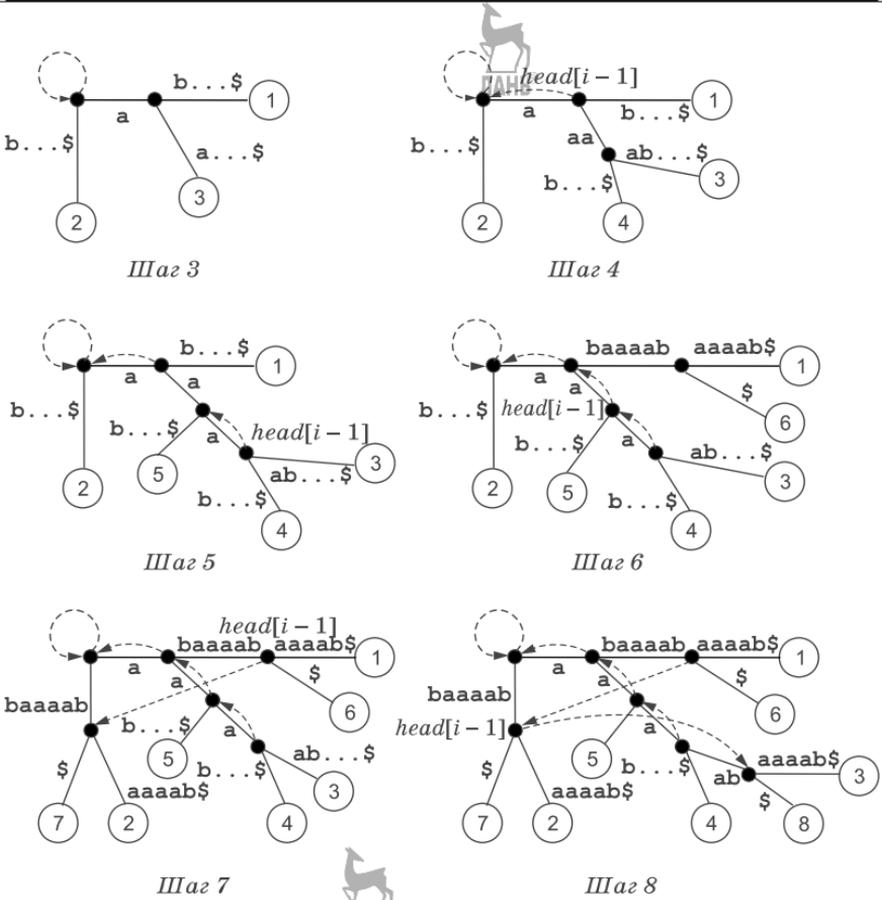


Рис. 3.13. Шаги 3–8 построения дерева суффиксов для строки  $abaaaabaaaab\$$

дуге с меткой  $abaaaab\$$  создается новая внутренняя вершина, а суффиксная связь идет из  $head[i - 1]$  (рис. 3.13) в образованную вершину.

Шаг 9. Наше предположение начинает работать! Суффиксная связь от вершины  $parent(head[i - 1])$  приводит к вершине  $u$  (рис. 3.14), от которой «скачок по счетчику» с меткой  $ab$  быстро приводит к образованию новой вершины, т. е. к определению окончания суффикса  $aaaab$  в дереве. При этом префикс  $aa$  суффикса в работе не участвовал.

Шаг 10. Выполняется аналогично предыдущему шагу, от вершины  $u$ , но с меткой  $b$ .

Шаг 11. Суффикс —  $ab\$$ . Операция  $s(parent(head[i - 1]))$  дает новое значение  $u$ .

Шаг 12. Идем от корня ( $u$  есть корень) с меткой  $b$ .

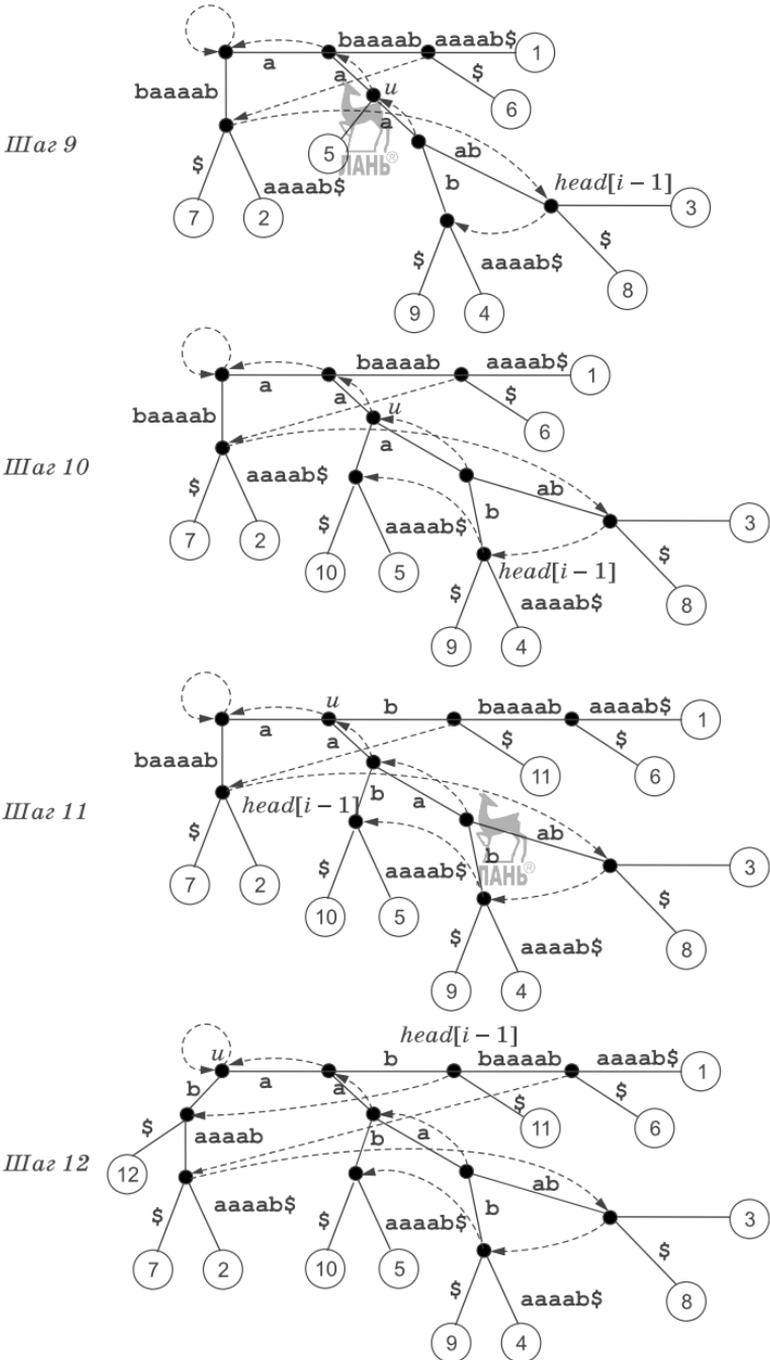


Рис. 3.14. Шаги 9–12 построения дерева суффиксов для строки  $abaaaaabaaaaab\$$

Описание алгоритма в общем виде может быть следующим:

```

<создать дерево  $T_1$ >;
q:=Nil;
For i:=2 To n+1 Do Begin
  <переменной u присвоить значение указателя на отца
  вершины дерева ( $q^{\wedge}.parent$ )>;
  <переменной v присвоить значение метки дуги,
  идущей от отца к сыну ( $от\ u^{\wedge}\ к\ q^{\wedge}$ )>;
  If u<>Nil Then Begin
    <по суффиксной связи отца перейти к вершине t>;
    <от вершины t с меткой v идти "скачком
    по счетчику" по дереву и определить вершину w>;
  End
  Else <идти от корня с меткой, начинающейся
  с символа  $v[2]$ , и определить вершину w путем
  обычного (простой алгоритм) просмотра вершин
  и дуг дерева>;
  If <w новая вершина> Then <w есть  $head[i]$  -
  добавить эту вершину в дерево>;
  <установить суффиксную связь от вершины
   $head[i-1]$  к вершине w>;
  <добавить лист с номером i в дерево и дугу от w
  к листу с меткой  $tail[i]$ >;
End;

```

Техника обоснования временной оценки алгоритма Е. Мак-Крейга практически совпадает с той, что была рассмотрена при анализе результата Э. Укконена.

### Упражнения

1. Измените строку  $abaaaaabaaaab\$$  на  $abaaaaabaaaabaaaab\$$  и продолжите построение дерева суффиксов, показанного на рис. 3.13 и рис. 3.14.
2. Напишите программу вычисления значений  $head[i]$  для произвольной строки  $S$ .
3. Предположим, что задано следующее описание вершины дерева суффиксов:

```

tree=^node;
node=Record
  cnt:Word;
  {Количество сыновей}

```

```

parent:tree;
  {Ссылка на родителя}
first,last:Word;
  {Индексы начала и конца метки дуги}
suf:tree;
  {Суффиксная связь}
num:Word;
  {Метка вершины}
ln:Word;
  {Количество символов в строке, получаемой
   "склежкой" меток дуг на пути от корня
   до вершины}
next:Array[1..|A|] Of tree;
  {Массив указателей на сыновей вершины}
End;

```

Здесь в переменной *pt* хранится указатель на корень дерева (на момент вызова процедуры). Найдите возможность улучшить программный код процедуры Search — поиска образца *P* в тексте *T* при построенном дереве суффиксов.

```

Procedure Search;
  Var i,j,r:Word;
      fn:Boolean; {Признак завершения работы}
Begin
  r:=1; {Номер символа в P}
  fn:=False;
  Repeat
    i:=1;
    While (i<=pt^.cnt) And
      (T[pt^.next[i]^first]<>P[r]) Do i:=i+1;
    If (i>pt^.cnt) Then Begin
      WriteLn('P нет в тексте');
      fn:=True;
    End
    Else Begin
      pt:=pt^.next[i];
      j:=1;
      While (T[pt^.first+j]=P[r+j]) And
        (pt^.first+j<=pt^.last) And
        (r+j<=m) Do j:=j+1;

```

```

If (r+j>m) Then Begin {Вывод}
  If (pt^.num0) Then WriteLn(pt^.num)
  Else <Обход части дерева (поиск
    в глубину) с вершины pr>;
  fn:=True;
End
Else
  If (pt^.first+j>pt^.last) Then r:=r+j
  {Переход к следующей вершине}
  Else Begin
    WriteLn('Строки нет в тексте');
    fn:=True;
  End;
End;
Until fn;
End;

```

Один из возможных вариантов реализации обхода части дерева имеет вид:

```

Procedure Pg(t:tree);
  Var i:Word;
  Begin
    If (t^.cnt=0) Then WriteLn(t^.num)
    Else
      For i:=1 To t^.cnt Do Pg(t^.next[i]);
  End;

```

4. Определите назначение функции Scan. Можно ли улучшить данную ее реализацию (все ли параметры в описании вершины дерева суффиксов используются эффективно)?

```

Function Scan(Var vf:Word;w:tree):tree;
  Var i,j:Word;
  fn:Boolean;
  tt:tree;
  Begin
    fn:=False;
    Repeat {Ищем сына}
      i:=1;
      While (i<=w^.cnt) And
        (T[w^.next[i]^first]<>T[vf])
        Do i:=i+1;

```

```

If (i>w^.cnt) Then fn:=True
Else Begin {Сын найден}
  w:=w^.next[i];
  j:=1;
  While (T[w^.first+j]=T[vf+j]) And
    (w^.first+j<=w^.last) Do j:=j+1;
  If (w^.first+j>w^.last) Then vf:=vf+j
    Else Begin
    {Совпадения закончились внутри дуги -
     разбиаем дугу}
    <Создаем новую вершину и корректируем
     поля вершины w^>;
    w:=<значение указателя на вновь
     созданную вершину>;
    vf:=vf+j;
    fn:=True;
    End;
  End;
Until fn;
  scan:=w;
End;

```

5. Разработайте программную реализацию алгоритма Е. Мак-Крейга.
6. Экспериментально исследуйте эффективность алгоритмов Э. Укконена и Е. Мак-Крейга на различных типах входных данных (кратные строки, строки на естественном языке, строки на небольшом алфавите и т. д.).

### 3.4. Суффиксные массивы

Как же это вы без гравицаппы пепелац выкатываете из гаража? Это непорядок...

(К/ф «Кин-дза-дза»)

Пусть дан текст  $T$  ( $n = |T|$ ), и для него построено дерево суффиксов. Предположим, что в этом дереве суффиксов дуги, выходящие из каждой внутренней вершины и из корня, лексически упорядочены, т. е. дуга  $(v, u)$  лексически меньше дуги  $(v, w)$ , если первый символ дуги  $(v, u)$  лексически меньше первого символа  $(v, w)$ . Тогда при обходе в глуби-

ну такого дерева (модификация процедуры Print из упражнения 2 к п. 3.1) и выписывании суффиксов в массиве (например, *ArSuf*) мы получим  $n$  лексически упорядоченных суффиксов  $T$  (точнее, их номеров). Дерево суффиксов строится за время  $O(n)$ ; обход в глубину также требует времени  $O(n)$ . Таким образом, и суффиксный массив мы тоже получаем за время  $O(n)$ .

### Пример

Для строк из табл. 3.5 массивы суффиксов *ArSuf* приведены в табл. 3.6.

Таблица 3.5

Номер позиции	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
<i>T</i>	a	b	a	a	b	a	b	a	a	b	a	a	b			
<i>T</i>	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b	a

Таблица 3.6

Номер элемента массива <i>ArSuf</i>	Номер суффикса	Суффикс	Номер суффикса	Суффикс
1	11	aab	16	a
2	8	aabaab	9	aabcbaba
3	3	aababaabaab	14	aba
4	12	ab	7	abaabcbaba
5	9	abaab	5	ababaabcbaba
6	6	abaabaab	10	abcbaba
7	1	abaababaabaab	2	acbababaabcbaba
8	4	ababaabaab	15	ba
9	13	b	8	baabcbaba
10	10	baab	13	baba
11	7	baabaab	6	babaabcbaba
12	2	baababaabaab	4	bababaabcbaba
13	5	babaabaab	1	bacbababaabcbaba
14			11	bcbaba
15			12	cbaba
16			3	cbababaabcbaba

Для второй строки в табл. 3.5 дерево суффиксов показано на рис. 3.15 (некоторые метки дуг, в силу их очевидности, на этом рисунке не приведены).

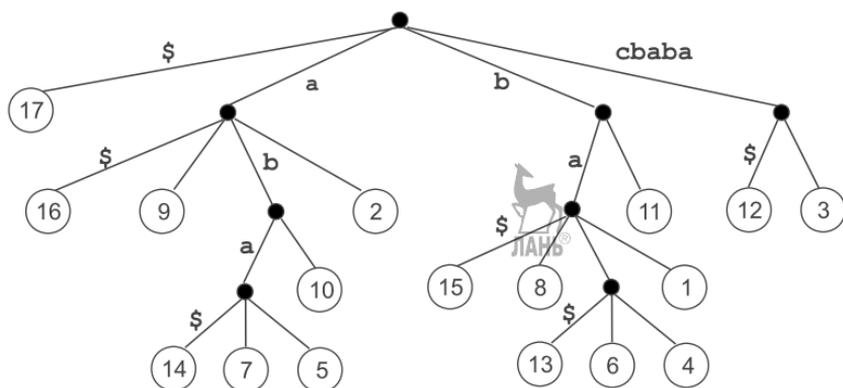


Рис. 3.15. Дерево суффиксов для строки bacbababaabcbaba

Путь от корня дерева до листа определяет суффикс (склежкой меток дуг).

Итак, при обходе в глубину дуги, выходящие из каждой внутренней вершины, просматриваются в лексическом порядке. Следовательно, и листья (а они соответствуют суффиксам) также проходятся в соответствии с этим же отношением порядка. Массив  $ArSuf$  — это не что иное, как упорядоченные (лексически) номера суффиксов  $T$ . Естественно, что терминальный символ  $\$$ , если мы выписываем не только номера, но и сами суффиксы путем склейки меток дуг, следует исключить из операции.

При наличии массива  $ArSuf$  задача поиска вхождения образца  $P$  в текст  $T$  решается путем использования классического алгоритма двоичного (бинарного) поиска. Действительно, при вхождении  $P$  в  $T$  все начальные позиции в массиве расположены последовательно, начиная с какого-то места, и этот факт является основанием для использования логики двоичного поиска.

Рассмотрим три варианта этого алгоритма. Первые два имеют временную оценку  $O(m \cdot \log_2 n)$ , а третий —  $O(m + \log_2 n)$ .

**Первый алгоритм** начнем рассматривать с примера.

*Пример*

Пусть  $T$  — это вторая строка из табл. 3.5, и требуется найти  $P = \text{baba}$ . Видим (см. табл. 3.6), что в 10-м, 11-м и 12-м элементах массива  $ArSuf$  указаны номера суффиксов, начинающихся с  $P$ . При этом имеет место три вхождения  $P$  в  $T$ .

Напомним логику двоичного поиска.

```

Function PSearch (P:String) :Integer;
  Var m,j,l,r,q:Integer;
      w:String;
  Begin
    m:=Length(P);
    j:=0;
    l:=1;
    r:=n;
  Repeat
    q:=(l+r) Div 2;
    w:=Copy(ArSuf[q],1,m);
    If P=w Then j:=q
    Else If P>w Then l:=q Else r:=q;
  Until (l=r) Or (j<>0);
  PSearch:=j;
End;

```

Данная функция возвращает номер элемента массива  $ArSuf$ , в котором указано начало суффикса  $T$ , содержащего  $P$  в качестве своего префикса. Она обеспечивает нахождение единственного вхождения  $P$  в  $T$  (если оно есть). Для поиска всех вхождений  $P$  в  $T$  необходимо проанализировать соседние суффиксы  $T$ .

Оценка времени поиска здесь очевидна: это  $O(m \cdot \log_2 n)$  (считаем, что осуществляется посимвольное сравнение  $P$  и префикса суффикса, определяемого элементом  $ArSuf[t]$ , а не сравнение строк, как приведено в функции PSearch).

**Второй алгоритм.** Особенность функции PSearch — в том, что на каждой итерации поиска  $P$  сравнивается с префиксом очередного суффикса, начиная с первого символа. Но суффиксный массив таков, что в нем рядом расположены суффиксы, имеющие наибольшие по длине совпадающие префиксы. Можно ли использовать этот факт и тем самым исключить явную избыточность (по количеству сравнений символов) функции PSearch? Пусть  $l$  и  $r$  определяют

границы поиска на очередной итерации, а значения  $ll$  и  $lr$  определяют длины общих префиксов: в первом случае — для суффикса с начальной позицией  $ArSuf[l]$  и  $P$ , а во втором —  $ArSuf[r]$  и  $P$ . Если через  $lt$  определить  $\min(ll, lr)$ , то в интервале от  $l$  до  $r$  все суффиксы  $T$  имеют префиксы длиной  $lt$ , совпадающие с префиксом этой же длины  $P$  и между собой. Получается, что сравнение символов можно осуществлять не с первого символа, а с символа с номером  $lt + 1$ . И если значения  $ll$  и  $lr$  корректировать вместе со значениями  $l$  и  $r$ , то мы уменьшаем количество сравнений (явно ненужных) символов в функции  $PSearch$ .

Модифицированный вариант функции  $PSearch$  тогда имеет вид:

```

Function PMSearch(P:String):Integer;
  Var m, j, l, r, q:Integer;
      ll, lr, lq, lt:Integer;
Begin
  m:=Length(P);
  j:=0;
  l:=1;
  r:=n;
  <вычисление ll и lr путем сравнения P
  с суффиксами ArSuf[l] и ArSuf[r]>;
  Repeat
    q:=(l+r) Div 2;
    lt:=Min(ll, lr);
  {Функция Min вычисляет минимальное из двух чисел}
  lq:=<значение длины совпадающей части
  P[lt+1..m] и T[ArSuf[q]+lt..n]>;
  lt:=lt+lq;
  If lt=m Then j:=q
  Else If P[lt+1]>T[ArSuf[q]+lt] Then Begin
    l:=q;
    ll:=lt;
  End
  Else Begin
    r:=q;
    lr:=lt;
  End;
  Until (l=r) Or (j<>0);
  PMSearch:=j;
End;

```

## Пример

Пусть дан текст  $T = \text{абаааабаааабааба}$ . Массив суффиксов  $ArSuf$  для  $T$  представлен в табл. 3.7.

Таблица 3.7

Номер элемента массива $ArSuf$	Номер суффикса	Суффикс
1	16	а
2	3	аааабаааабааба
3	8	аааабааба
4	9	аааабааба
5	4	ааааааабааба
6	13	ааба
7	5	аабаааабааба
8	10	аабааба
9	14	аба
10	1	абаааабаааабааба
11	6	абаааабааба
12	11	абааба
13	15	ба
14	2	баааабаааабааба
15	7	баааабааба
16	12	бааба

Логика поиска  $P$  «с привязкой» к переменным функции  $PMSearch$  приведена в табл. 3.8.

Таблица 3.8

	$l$	$r$	$ll$	$lr$	$q$	$lt$	$lq$	$lt$	Сравнение
$P = \text{аааба}$	1	16	1	0	8	0	2	2	$P[3] < T[15]$
	1	8	1	2	4	1	3	4	Элемент найден
$P = \text{аабаа}$	1	16	1	0	8	0	4	4	$P[5] < T[17]$
	1	8	1	4	4	1	1	2	$P[3] > T[11]$
	4	8	2	4	6	2	3	5	Элемент найден

Пусть  $P = aaaba$ . На первой итерации  $P$  сравнивается с суффиксом, находящимся на восьмом месте в массиве  $ArSuf$ . Начальное значение количества совпадающих символов ( $lt$ ) равно нулю, конечное — двум (новое значение  $lt$ ). Так как  $P[3]$  меньше  $T[15]$ , то изменяем верхнюю границу поиска ( $r$ ) и значение  $lr$ . Сравнение  $P$  с суффиксом, определяемым четвертым элементом  $ArSuf$ , начинается со второго элемента  $P$  (этот элемент повторно участвует в сравнениях, — он сравнивался и на первой итерации). Совпало три символа. Образец исчерпан и найдено его вхождение в  $T$ .

Пусть теперь  $P = aabaa$ . На второй итерации (см. табл. 3.8) второй символ  $P$  вновь участвует в сравнениях. Значение  $P[3]$  больше  $T[11]$  — изменяется нижняя граница поиска и значение  $ll$ . На третьей итерации сравнения начинаются с третьего символа  $P$ . Произошло три совпадения, и образец  $P$  найден.

Анализируя функцию  $PMSearch$ , мы видим, что часть избыточных сравнений в ней исключена, но временная оценка остается прежней —  $O(m \cdot \log_2 n)$ . Можно ли свести ее к  $O(m + \log_2 n)$ ? Это означало бы, что на всех итерациях двоичного поиска количество сравнений символов (суммарное) имеет такой порядок, или, другими словами, количество избыточных сравнений по отношению к длине  $P$  равно  $\log_2 n$ .

Из всего предшествующего изложения ясно, что если это возможно, то только за счет введения дополнительных структур данных, описывающих какой-то аспект (особенность) задачи и формируемых на предварительной стадии обработки. Другого нам не дано, и это очередной раз подтверждает тезис Н. Вирта: программа (метод обработки) есть синтез данных и управления!

**Третий алгоритм.** Рассмотрим более подробно схему двоичного поиска и попытаемся использовать идею исключения лишних сравнений, заложенную в функции  $PMSearch$ . Начнем опять-таки с примера.

### Пример

Во второй строке табл. 3.5 была приведена строка  $T = bacbababaabcbaba$ . Массив суффиксов для нее был приведен в табл. 3.6, а дерево суффиксов (без суффиксных связей) показано на рис. 3.15. Рассмотрим возможные значения  $l$  и  $r$  — границ интервала в схеме двоичного поиска (они приведены на рис. 3.16 в круглых скобках у каждой вершины дерева).

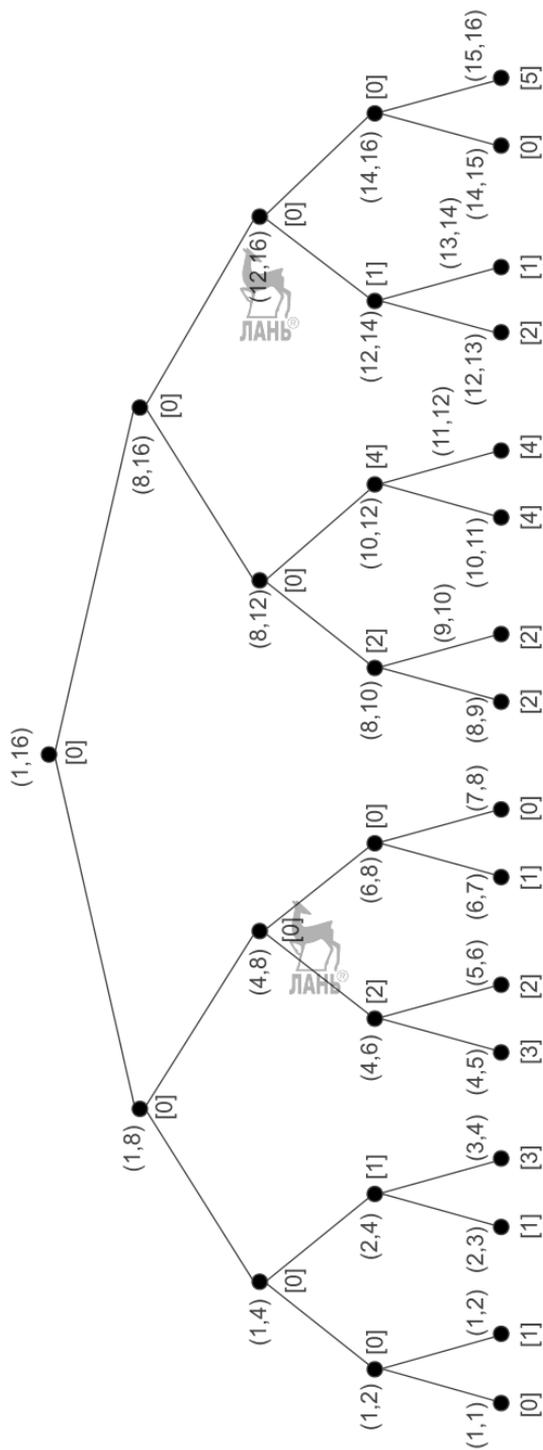


Рис. 3.16. Интервалы схемы двойного поиска

Количество возможных границ интервалов поиска равно количеству вершин дерева, т. е.  $2n - 1$ . Видим, что некую характеристику (параметр), связанную со схемой двоичного поиска, можно на стадии предварительной обработки вычислить за время  $O(n)$ . Этой характеристикой может быть  $lcp[l, r]$  — длина *наибольшего общего префикса* (*longest common prefix*) суффиксов, определенных позициями  $l$  и  $r$  массива  $ArSuf$ . Так, в нашем примере  $ArSuf[10] = 13$ , а  $ArSuf[12] = 4$ , и  $lcp[10, 12] = 4$  (суффиксы *baba* и *bababaabcbaba*). На рис. 3.16 значение  $lcp$  для всех границ поиска указано в квадратных скобках у вершин дерева.

Предположим, что значения  $lcp$  вычислены на стадии предварительной обработки. Что они могут дать в процессе поиска? (Мы, конечно же, в рассуждениях отталкиваемся от функции  $PMSearch$ .)

Если на какой-то итерации длины совпадающих префиксов  $P$ ,  $ArSuf[l]$  ( $ll$ ) и  $P$ ,  $ArSuf[r]$  ( $lr$ ) совпадают ( $ll = lr$ ), то префиксы такой длины всех суффиксов из этого интервала равны префиксу  $P$ , и сравнение следует начать с символа  $ll + 1 = lr + 1$ .

Значение  $q = (l + r) \text{ Div } 2$ , и известно значение  $lcp[l, q]$  — это длина наибольшего общего префикса суффиксов, представленных в позициях  $l$  и  $q$  массива  $ArSuf$ .

Если  $lcp[l, q] > ll$  (общий префикс суффиксов  $ArSuf[l]$  и  $ArSuf[q]$  длиннее, чем общий префикс  $P$  и  $ArSuf[l]$ ), то символ  $ll + 1$  образца  $P$  лексически больше символа  $ll + 1$  у суффиксов  $ArSuf[l]$  и  $ArSuf[q]$  (он у них один и тот же). Эта ситуация показана на рис. 3.17: (а) — для  $ll > lr$ ; (б) — для  $ll < lr$ . В данном случае на любой итерации двоичного поиска проверок  $P$  не требуется,  $l$  заменяется на  $q$ , а  $ll$  и  $lr$  остаются неизменными — вхождения  $P$  в  $T$  должны находиться справа от позиции  $q$  в  $ArSuf$ .

Выполнение неравенства  $lcp[l, q] < ll$  говорит о том, что общий префикс суффикса  $ArSuf[l]$  и  $P$  больше (длиннее) общего префикса суффиксов  $ArSuf[l]$  и  $ArSuf[q]$ . Следовательно, символ  $lcp[l, q] + 1$  у  $P$  и суффикса  $ArSuf[l]$  один и тот же, и он лексически меньше, чем у суффикса  $ArSuf[q]$ , поэтому вхождения  $P$  в  $T$  при их существовании возможны только в суффиксы, представленные до позиции  $q$  в  $ArSuf$ . При возникновении такой ситуации никаких проверок  $P$  не

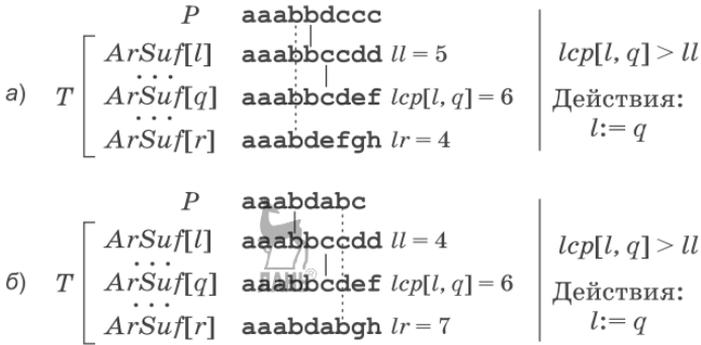


Рис. 3.17. Поиск *P* в *T* — случай, когда *lcp*[*l*, *q*] > *ll*

требуется, *lr* заменяется на *lcp*[*l*, *q*], а *r* заменяется на *q* (рис. 3.18).

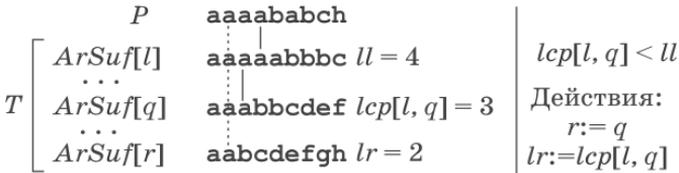


Рис. 3.18. Поиск *P* в *T* — случай, когда *lcp*[*l*, *q*] < *ll*

*Примечание.* Вариант *lr* > *ll* при *lcp*[*l*, *q*] < *ll* становится невозможным.

Выполнение равенства *lcp*[*l*, *q*] = *ll* — это случай, когда префиксы всех суффиксов (длиной *ll*) в интервале от *l* до *q* совпадают с префиксом *P*. Необходимо сравнивать *P* с суффиксом *ArSuf*[*q*] начиная с позиции *ll* + 1 и по результатам сравнения изменять *l* или *r* с соответствующим изменением *ll* и *lr* (рис. 3.19).

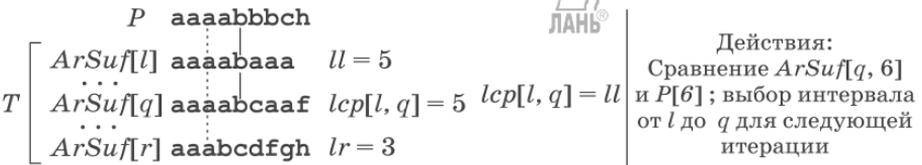


Рис. 3.19. Поиск *P* в *T* — случай, когда *lcp*[*l*, *q*] = *ll*

Реализация данной логики приводит к алгоритму с временной оценкой  $O(m + \log_2 n)$ .

Действительно, на каждой итерации двоичного поиска у нас или не проверяется ни одного символа, а изменяются только границы поиска, или итерация заканчивается первым несовпадением, а сравнения начинаются с символа  $\max(l, r)$ , совпадает некоторое количество символов  $w - 1$  и на символе с номером  $w$  происходит несовпадение. Тогда на значение  $w - 1$  в конце итерации увеличивается или величина  $l$ , или  $r$ , а следующая итерация начинается с символа с номером  $w$ . Таким образом, на каждой итерации делается не более одного избыточного сравнения, т. е. всего  $\log_2 n$  сравнений, а количество совпадений (успешных сравнений) не превосходит  $m$ .

Однако у нас остался открытым вопрос о вычислении значений элементов массива  $lcp$ , выполняемом на стадии предварительной обработки при создании массива суффиксов  $ArSuf$ .

На рис. 3.16 было показано полное двоичное дерево с  $n$  листьями (листья имеют метки  $(i, i + 1)$ , а один лист — метку  $(1, 1)$ ) для строки  $T = \text{bacbababaabcbaba}$  (вторая строка табл. 3.5), задающее все возможные интервалы  $(l, r)$  при двоичном поиске. Дерево суффиксов же было приведено на рис. 3.15. При обходе в глубину дерева суффиксов между просмотрами вершин-листьев, соответствующих суффиксам  $ArSuf[i]$  и  $ArSuf[i + 1]$ , посещаются некоторые внутренние вершины. Ближайшая из них к корню определяет значение  $lcp[i, i + 1]$ ; точнее, длина подстроки, получающаяся путем склейки меток дуг от корня до нее (строковая глубина вершины), равна  $lcp[i, i + 1]$ . Строковая глубина вершины как ее параметр формируется при создании дерева суффиксов, тогда при получении  $ArSuf$  определяются и значения  $lcp[i, i + 1]$ , что требует времени  $O(n)$ .

Для вычисления оставшихся значений  $lcp[i, j]$  заметим, что  $lcp[i, i + 2]$  равно минимуму из  $lcp[i, i + 1]$  и  $lcp[i + 1, i + 2]$ . Продолжая рассуждения, определяем, что  $lcp[i, j] = \min\{lcp[k, k + 1] : i \leq k < j\}$ . Если полное двоичное дерево обходить по принципу «левый сын, правый сын, а затем отец сыновей» и на выходе из рекурсивного обхода сравнивать значения  $lcp$  сыновей, то время формирования  $lcp[i, j]$  пропорционально количеству вершин дерева, т. е.  $O(n)$ .



## Упражнения

1. Напишите процедуру формирования суффиксного массива из дерева суффиксов.
2. С помощью функции `PSearch` обеспечивается нахождение *единственного* вхождения  $P$  в  $T$ . Разработайте логику поиска *всех* вхождений  $P$  в  $T$ .
3. Приведите пример (значения  $T$  и  $P$ ), в котором функция `PMSearch` не уменьшает количества сравнений символов по отношению к функции `PSearch`.
4. Приведите пример для случая, когда количество несопадений в алгоритме (втором) в точности равно  $\log_2 n$ .
5. Найдите способ хранения значений  $lcp$  в одномерном массиве.
6. Приведите пример строки. Вычислите для нее массивы  $ArSuf$  и  $lcp$ .
7. Разработайте программную реализацию третьего алгоритма поиска  $P$  в  $T$ .

## 3.5. Алгоритм А. Ахо – М. Корасик

Представляешь, вчера ходил в лес и у поляны встретил огромную толпу деревьев-сверстников. Зорик, 4 класс.

М. Дымов «Дети пишут Богу»

Пусть в тексте  $T$  ( $n = |T|$ ) нам необходимо найти все вхождения не одного образца  $P$ , а нескольких —  $P_1, P_2, \dots, P_t$ . Конечно, такая задача может быть решена путем последовательного поиска образцов и, соответственно, иметь временную оценку  $O(t \cdot n + m)$ . Однако А. Ахо и М. Корасик предложили алгоритм с временной оценкой  $O(n + m + k)$ , где  $k$  — суммарное количество вхождений образцов  $P_i$  ( $1 \leq i \leq t$ ) в текст  $T$ , а  $m$  — суммарная длина образцов.

Введем понятие «*дерево ключей*». Оно отличается от дерева суффиксов (см. п. 3.1) только тем, что меткам дуг в нем приписываются не значения подстрок, а отдельные символы. Дерево ключей строится для всего множества образцов, причем свойство дерева суффиксов — что любые две дуги,

выходящие из одной и той же вершины, имеют разные метки, — в нем сохраняется. Тогда каждому образцу  $P_i$  соответствует такая вершина дерева ключей, что склейка символов на пути из корня дерева в эту вершину в точности составляет  $P_i$ .

Логика построения дерева ключей достаточно «прозрачна». Мы начинаем из одной вершины — корня, а затем обрабатываем образцы  $P_i$  один за другим: идем из корня по дугам, отмеченным символами из  $P_i$ , до тех пор, пока это возможно. Если  $P_i$  заканчивается в какой-то вершине  $v$ , то мы сохраняем идентификатор  $P_i$  (например,  $i$ ) в  $v$  — нумеруем вершину. В случае же, когда дуги, помеченной очередным символом  $P_i$ , нет, мы создаем новые дуги и вершины для всех оставшихся символов  $P_i$ . При выборе дуги из вершины за константное время мы получаем очевидную оценку —  $O(m)$ . За это время дерево ключей будет построено.

Пример

Пусть дано множество образцов:  $P_1 = abcdccb$ ,  $P_2 = bcddccb$ ,  $P_3 = dccb$ ,  $P_4 = bc$ ,  $P_5 = dc$ ,  $P_6 = cb$ . Дерево ключей для  $P_i$  ( $i = 1, \dots, 6$ ) показано на рис. 3.20.

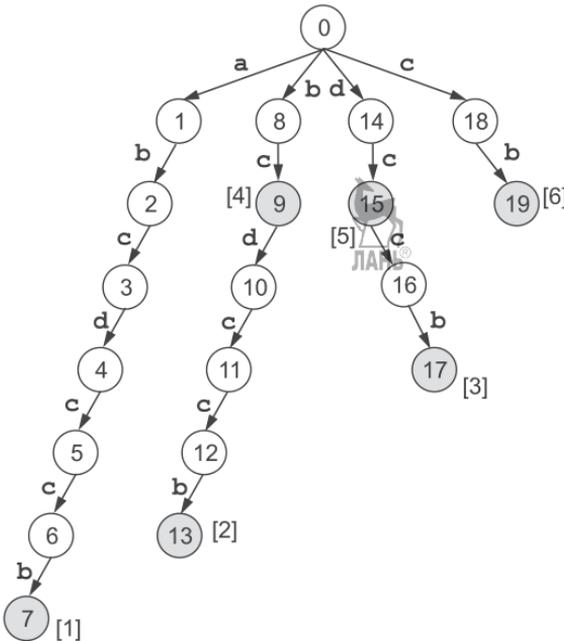


Рис. 3.20. Пример дерева ключей

Темным фоном на рис. 3.20 выделены вершины  $v$ , соответствующие образцам с номером  $i$ . Значение  $i$  указано рядом с каждой такой вершиной в квадратных скобках.

Напомним, что автомат — это пятерка  $M = (A, Q, q_0, E, F)$ , где  $A$  — алфавит (для рассматриваемого примера это  $\{a, b, c, d\}$ );  $Q$  — множество состояний (вершин дерева ключей);  $q_0$  — начальное состояние (корень дерева);  $E$  — множество конечных состояний (вершин дерева с темным фоном на рис. 3.20, или вершин дерева, имеющих метки в виде номера образца),  $F$  — функция переходов. Частично функция переходов  $F: A \cdot Q \rightarrow Q$  определена при построении дерева ключей: если автомат при обработке некоторого символа текста  $T[i]$  находится в состоянии  $q$  и  $T[i]$  совпадает с символом образца, то новое состояние очевидно — это переход по дереву ключей. Чтобы работала стандартная логика автомата с линейной временной оценкой, требуется сделать, как минимум (опять же за линейное время на стадии предварительной обработки), две вещи (ответить на два вопроса).

**Procedure** Search;

**Var**  $i, q$ :Integer;

**Begin**

$n := \text{Length}(T)$  ;

$q := 0$ ;

**For**  $i := 1$  **To**  $n$  **Do Begin**

$q := F[q, T[i]]$ ;

**If**  $q \in E$  **Then** *<вывести все образцы  $P_t$ , связанные с состоянием  $q$ ; начало их вхождения определяется как  $i - \text{Length}(P_t) + 1$ >*

*{Индекс  $t$  принимает значение из некоторого подмножества множества номеров образцов}*

**End**;

**End**;



Первый вопрос заключается в определении состояния автомата, в которое следует переходить при несовпадении символов. Будем рассуждать по аналогии с понятием суффиксной связи (см. п. 3.2). Каждой вершине  $v$  (состоянию) дерева ключей соответствует префикс (путем конкатенации меток дуг от корня) некоего образца; пусть это  $P_a[1..j]$ . Следует найти такую вершину дерева  $w$  (или образец  $P_b$ ), что префикс  $P_b[1..k]$  совпадает с самым длинным суффиксом

$P_a[1..j]$ . Другими словами, требуется найти самый длинный суффикс  $P_a[1..j]$ , совпадающий с префиксом какого-то образца (т. е. мы делаем обобщение понятия граней строки). В силу свойств дерева ключей — все дуги, выходящие из вершин, имеют различные значения меток, — такая вершина (префикс) — единственная (если она существует) и, соответственно, такая связь  $v \rightarrow w$ . Что дает эта связь и как ее строить?

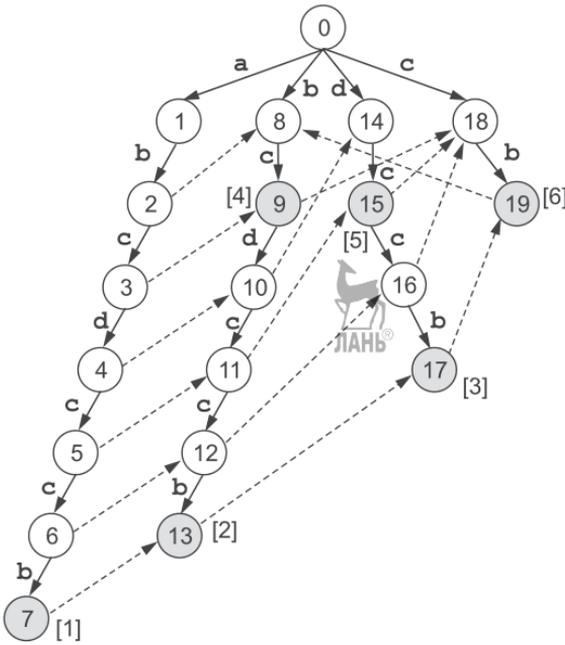
Пусть мы находились в состоянии  $v$ , и произошло несоответствие символа  $T[i]$  с очередным символом какого-то образца. Автомат осуществляет переход  $v \rightarrow w$ . Значение  $i$  остается прежним (нет возврата назад по тексту  $T$ , что является необходимым условием получения линейного по времени алгоритма), и нет необходимости сравнивать символы на пути от корня до вершины  $w$ . Начало вхождения образца в текст пересчитывается при  $v \rightarrow w$  очевидным образом — по значению  $i$ , длине префикса и известному до перехода началу вхождения.

На рис. 3.21 представлена диаграмма переходов автомата для рассматриваемого примера, а в табл. 3.9 — функция переходов  $F$ . Скажем, если мы находимся в состоянии 12,  $i = 18$  и префикс `bedcc` входит в текст с 13-й позиции, то начало вхождения `ccc` определяется как  $18 - 3 = 15$ .

Таблица 3.9

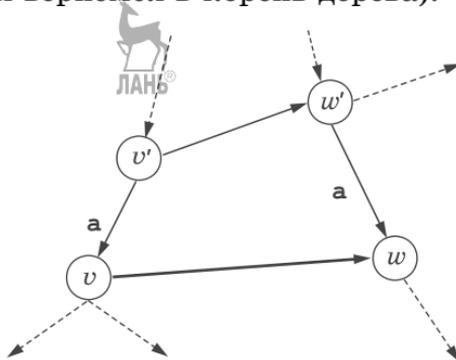
$F$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	1	0	8	9	10	11	12	13	0	18	14	15	16	17	0	18	18	19	0	8
б	8	2	8	9	10	11	7	13	0	18	14	15	13	17	0	18	17	19	19	8
с	18	0	3	9	5	6	12	13	9	18	11	12	16	17	15	16	18	19	0	8
д	14	0	8	4	10	11	12	13	0	10	14	15	16	17	0	18	18	19	0	8

Метод определения пунктирных связей (рис. 3.21) за линейное время известен нам по теме «деревья суффиксов». Напомним его (рис. 3.22). Пусть мы находимся в вершине  $v$ . Тогда мы идем к отцу этой вершины  $v'$  (дуга между  $v'$  и  $v$  помечена символом  $a$ ), а затем переходим к вершине  $w'$ . Связь, обеспечивающая этот переход, уже имеется в дереве. Если из вершины  $w'$  есть дуга, помеченная тем же символом  $a$ , и она идет в вершину  $w$ , то связь между  $v$  и  $w$  установлена (она



**Рис. 3.21.** Диаграмма переходов конечного автомата для поиска в тексте образцов  $P_1 = abcdccb$ ,  $P_2 = bcdccb$ ,  $P_3 = dccb$ ,  $P_4 = bc$ ,  $P_5 = dc$ ,  $P_6 = cb$

выделена толстой линией на рис. 3.22). Если нет, то мы идем дальше по дереву (пунктирная стрелка из вершины  $w'$  на рис. 3.22). В любом случае этот процесс конечен (в худшем случае мы вернемся в корень дерева).



**Рис. 3.22.** Формирование связи между вершинами  $v$  и  $w$

Осталось ответить на второй вопрос — как выводить вхождения образцов в текст? Обратимся к рис. 3.21. Состоя-

ния автомата, соответствующие окончаниям образцов, выделены на этом рисунке темным фоном и имеют метки в виде номера образца. Естественно, что для них известны и длины образцов. Что нам требуется сделать? Да просто собрать для каждого состояния автомата «темные кружочки»! Результат этой сборки (а она осуществляется в процессе формирования функции  $F$  автомата и не влияет на временную оценку) приведен для нашего «сквозного» примера в табл. 3.10.

Таблица 3.10

Номер состояния автомата	Список номеров образцов	Номер состояния автомата	Список номеров образцов
3	4	13	2, 3, 6
5	5	15	5
7	1, 2, 3, 6	17	3, 6
9	4	19	6
11	5		

Обсуждение алгоритма А. Ахо и М. Корасик завершим рассмотрением следующего примера. Пусть дан текст  $T = abcdccbcddccbbcabcdccb$ . Работа автомата (она однозначно отслеживается с помощью  $F$  (табл. 3.9) или диаграммы переходов на рис. 3.21), созданного для приведенного множества образцов, дает результат, представленный в табл. 3.11.

Таблица 3.11

Номер позиции в тексте (i)	Номер образца	Начальная позиция образца в T	Номер позиции в тексте (i)	Номер образца	Начальная позиция образца в T
3	4	2	12	3	9
5	5	4	12	6	11
7	1	1	14	4	13
7	2	2	17	4	16
7	3	4	19	5	18
7	6	6	21	1	15
8	4	7	21	2	16
10	5	9	21	3	18
12	2	7	21	6	20

Нетрудно при этом отследить последовательность переходов из одного состояния автомата в другое.



### Упражнения

1. Разработайте простой алгоритм поиска образцов в тексте с временной оценкой  $O(t \cdot n \cdot m)$ , где  $m$  — суммарная длина образцов.
2. Приведите пример множества образцов, для которого диаграмма переходов автомата не будет иметь пунктирных связей (см. рис. 3.21). Как в этом случае работает алгоритм А. Ахо – М. Корасик?
3. Приведите пример множества образцов. Постройте для него дерево ключей и преобразуйте его в диаграмму работы автомата. Сформируйте для каждого состояния список образцов, которые следует вывести при его достижении.
4. Выясните, как алгоритм А. Ахо – М. Корасик работает с периодическими строками. Пример периодической строки:  $(abc)^t$ , где  $t$  — натуральное число.
5. Реализуйте алгоритм А. Ахо – М. Корасика (напишите программу). Основная часть логики может выглядеть следующим образом:

**Begin**

```
Init;      {Инициализация данных}
KeyTree;  {Построение дерева ключей}
Automat;  {Построение автомата}
Search;   {Поиск вхождения образцов в текст
           с выводом результата}
```

**End;**

*Примечание.* Изящество реализации будет, как обычно, зависеть от выбора структур данных.

### Методический комментарий

Первый алгоритм построения дерева суффиксов (дерева позиций) за линейное время разработан П. Вайнером<sup>1)</sup> (*Peter Weiner*) в 1973 г. Затем в 1976 г. Е. Мак-Крейгом<sup>2)</sup> (*Edward McCreight*) был предложен другой алгоритм. Вариант Е. Мак-Крейга более экономичен по памяти. В 1995 г. Э. Укко-

<sup>1)</sup> *Weiner P.* Linear pattern matching algorithms // Proc. of the 14<sup>th</sup> IEEE Symp. on Switching and Automata Theory. 1973. P. 1–11.

<sup>2)</sup> *McCreight E. M.* A space-economical suffix tree constructor algorithm // J. ACM. 1976. Vol. 23. P. 262–272.

нен<sup>1)</sup> (*Esko Ukkonen*) разработал новый вариант алгоритма. Его принципиальное отличие заключается в возможности обработки текста в реальном масштабе времени — дерево суффиксов строится по мере поступления символов текста. В 1997 г. М. Фарах<sup>2)</sup> (*Martin Farach*) для индексированных алфавитов создал другой вариант алгоритма построения дерева суффиксов. Описание алгоритма П. Вайнера можно найти в книге Д. Гасфилда, а алгоритма М. Фараха — в книге Б. Смита.

Суффиксные массивы как новая структура данных, рационально использующая память в задаче поиска подстроки в тексте, предложена У. Манбер (*Udi Manber*) и Г. Майерсом (*Gene Myers*)<sup>3)</sup>. Известно, что суффиксные массивы могут быть сформированы без использования дерева суффиксов, но за несколько большее время<sup>4)</sup>.

Алгоритм А. Ахо (*Alfred Aho*) и М. Корасик (*Margaret Corasick*)<sup>5)</sup>, с одной стороны, развивает тему использования методов теории автоматов в решении задач обработки строк, а с другой — наглядно демонстрирует применение идей рассматриваемой структуры данных — деревьев суффиксов.

---

<sup>1)</sup> *Ukkonen E.* On-line construction of suffix-trees // *Algorithmica*. 1995. Vol. 14. P. 249–260.

<sup>2)</sup> *Farach M.* Optimal suffix tree construction with large alphabets // *Proc. 38<sup>th</sup> Annual IEEE Symp. Foundations of Computer Science*, 1997. P. 137–143.

<sup>3)</sup> *Manber U., Myers G.* Suffix arrays: a new method for on-line string searches // *Proc. First Annual ACM – SIAM Symp. Discrete Algs.*, 1990. P. 319–327.  
*Manber U., Myers G.* Suffix arrays: a new method for on-line string searches // *SIAM J. Comput.* 22–5, 1993. P. 935–948

<sup>4)</sup> *Sadakane K.* A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation // *Proc. IEEE Data Compression Conference*, 1998. P. 129–138.

<sup>5)</sup> *Aho A., Corasick M.* Efficient string matching: an aid to bibliographic search // *Comm. ACM*. 1975. Vol. 18. P. 333–340.

## Вычисление расстояния между строками

---



Какая дальность расстоянья!  
В одной из городских квартир  
В столовой — речь о Ляояне,  
А в детской — тушь и транспортир.

*Борис Пастернак*

В изучаемом разделе теории построения алгоритмов на строках решается задача *приближенного* сравнения двух строк. Но для измерения приближенности требуется введение меры, или «расстояния» между объектами, в данном случае — строками. Только после этого можно сказать, насколько сильно отличаются строки друг от друга. В данной главе рассматривается задача нахождения расстояния между строками и родственная ей задача о нахождении наибольшей общей подпоследовательности. В основе методов вычисления расстояния лежат идеи динамического программирования, а решениями являются алгоритмы динамического программирования<sup>1)</sup>.

### 4.1. Основной алгоритм

Главное всюду — начать; начало  
важнейшая часть дел.

*Авсоний*

*Мера близости (расстояние)* должна удовлетворять следующим утверждениям. Пусть  $S_1$ ,  $S_2$  и  $S_3$  — строки, а  $d$  — расстояние, тогда:

- $d(S_1, S_2) \geq 0$  — условие неотрицательности;
- из  $d(S_1, S_2) = 0$  следует, что  $u = v$  и наоборот — условие единственности;

---

<sup>1)</sup> Считаем, что с методом динамического программирования читатель знаком, например, по книге: *Окулов С. М. Программирование в алгоритмах.* — М.: БИНОМ. Лаборатория знаний, 2007.

- $d(S_1, S_2) = d(S_2, S_1)$  — условие симметричности;
- $d(S_1, S_3) \leq d(S_1, S_2) + d(S_2, S_3)$  — неравенство треугольника.

Различие строк  $S_1$  и  $S_2$  между собой обычно определяют в количестве операций, которые следует выполнить для преобразования  $S_1$  в  $S_2$ . К таким операциям относят: вставку (In — Insert) символа; удаление (Dl — Delete) символа; подстановку или замену (Re — Replace) символа; буквосочетанием же Ma (от слова Match) обозначим отсутствие операций над символом строки. Все такие операции обычно определяют как *операции редактирования*.

*Расстояние Р. Хемминга* для строк  $S_1$  и  $S_2$  одинаковой длины  $n$  определяется как минимальное количество операций подстановок (Re), необходимых для преобразования  $S_1$  в  $S_2$ .

#### Пример

Пусть  $S_1 = abcd$ ,  $S_2 = acbd$ . Тогда расстояние Р. Хемминга для этих строк  $d(u_1, u_2) = 2$ .

*Расстояние В. Левенштейна*<sup>1)</sup> для строк  $u_1$  и  $u_2$  определяется как минимальное количество операций вставок (In), удалений (Dl) и подстановок (Re), необходимых для преобразования  $u_1$  в  $u_2$ .

#### Пример

Пусть  $S_1 = abbca$ , а  $S_2 = aca$ . Последовательность операций MaReReDlDl над символами первой строки (первый символ не изменяется, второй и третий заменяются, четвертый и пятый — удаляются) дает  $d(S_1, S_2) = 4$ . Аналогично последовательность операций MaReDlDlMa приводит к расстоянию  $d(S_1, S_2) = 3$ , а последовательность MaDlDlMaMa —  $d(S_1, S_2) = 2$ .

Таким образом, если речь идет о минимальном расстоянии, то следует понимать, что определяется не только конкретное числовое значение, но и последовательность операций, применение которых обеспечивает его достижение. Кроме того, первая строка преобразуется во вторую, однако очевидно, что вставка символа в одну строку может рас-

<sup>1)</sup> Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады АН СССР. 1965. Т. 163. С. 707–710.

смагиваться как удаление символа из другой строки, и наоборот.

*Примечание.* Далее в основном мы будем подразумевать расстояние В. Левенштейна<sup>1)</sup>. Если же речь пойдет о другой мере близости, то это будет специально оговорено.

Пусть даны две строки  $S_1$  ( $|S_1| = n$ ) и  $S_2$  ( $|S_2| = m$ ), и требуется вычислить расстояние между ними. Введем массив  $D[0..n, 0..m]$ , где элемент  $D[i, j]$  определяется как расстояние между подстроками  $S_1[1..i]$  и  $S_2[1..j]$ . Тогда  $d(S_1, S_2) = D[n, m]$ .

Сделаем очевидные утверждения:

- $D[0, 0] = 0$  — пустая строка преобразуется в пустую строку за нулевое количество операций;
- $D[0, j] = j$  для  $j$  от 1 до  $m$  — пустая строка преобразуется в строку  $S_2[1..j]$  за  $j$  операций вставки, и это — минимальное количество таких операций;
- $D[i, 0] = i$  для  $i$  от 1 до  $n$  — строка  $S_1[1..i]$  преобразуется в пустую строку за  $i$  операций удаления, и это — минимальное количество таких операций.

Осталось определить следующее рекуррентное соотношение для  $D[i, j]$  при  $1 \leq i \leq n$  и  $1 \leq j \leq m$ :

$$D[i, j] = \min\{D[i - 1, j] + 1, D[i, j - 1] + 1, D[i - 1, j - 1] + t\},$$

где  $t = 1$ , если  $S_1[i] \neq S_2[j]$ , и  $t = 0$ , если  $S_1[i] = S_2[j]$ .

Указанное рекуррентное соотношение отражает следующие факты:

- *аддитивность задачи* — для получения минимального расстояния между подстроками  $S_1[1..i]$  и  $S_2[1..j]$  достаточно знать минимальные расстояния между подстроками:  $S_1[1..i - 1]$  и  $S_2[1..j]$ ;  $S_1[1..i]$  и  $S_2[1..j - 1]$ ;  $S_1[1..i - 1]$  и  $S_2[1..j - 1]$ , — ничего другого не требуется;
- *достижимость значения  $D[i, j]$*  — конкретные операции, переводящие каждое из минимальных расстояний  $D[i - 1, j]$ ,  $D[i, j - 1]$ ,  $D[i - 1, j - 1]$  в  $D[i, j]$ .

<sup>1)</sup> Расстояние В. Левенштейна в различных источниках определяют по-разному. В работе Б. Смита в определении фигурируют только операции вставки и удаления, а для расстояния, определенного так, как это сделано в тексте данной книги, вводится новое понятие — *расстояние преобразования*.

Приостановим пока попытки обоснования этих утверждений и рассмотрим несколько примеров, тем более что логика формирования массива  $D$  абсолютно «прозрачна» (как и в большинстве задач динамического программирования, если осознана суть аддитивности):

```

Procedure CreateDist;
  Var i, j, t: Word;
  Begin
    D[0,0] := 0;
    For i := 1 To n Do D[i,0] := i;
    For j := 1 To m Do D[0,j] := j;
    For i := 1 To n Do
      For j := 1 To m Do Begin
        If S1[i] = S2[j] Then t := 0 Else t := 1;
        D[i,j] := Min(D[i-1,j]+1, D[i,j-1]+1,
                     D[i-1,j-1]+t);
        {Функция вычисления минимального из трех чисел}
      End;
    End;
  End;

```

### Пример 1

$S_1 = \text{abcdbad}$ ,  $S_2 = \text{bacaba}$ . Массив  $D$  (табл. 4.1) заполняется последовательно: сначала элементы первой строки, затем — второй и т. д. Значение  $D[6, 7]$  говорит о том, что строка  $S_1$  преобразуется в строку  $S_2$  за четыре операции. Так, подстрока  $S_1[1..6] = \text{abcdba}$  преобразуется в  $S_2[1..6] = \text{bacaba}$  за три операции ( $D[6, 6] = 3$ ), а удаляя из  $S_1[1..7]$  последний символ (одна операция), мы получаем четыре операции. Подстрока  $S_1[1..6] = \text{abcdba}$  преобразуется в  $S_2[1..5] = \text{bacab}$  за четыре операции. Выполняя замену символа  $S_1[7] = \text{d}$  на символ  $S_2[6] = \text{a}$ , мы имеем одну операцию, а в итоге — пять. Подстрока  $S_1[1..7] = \text{abcdbad}$  преобразуется в  $S_2[1..5] = \text{bacab}$  за пять операций. Выполняя вставку в  $S_1$  символа  $\text{a}$ , мы получаем шесть операций. Минимальное из трех чисел  $\{3, 4, 5\}$  есть 3. По такой же логике в соответствии с установленной очередностью формируются и другие элементы  $D$ .

Таблица 4.1

$D$	Номер символа	0	1	2	3	4	5	6	7
Номер символа	Символ		a	b	c	d	b	a	d
0	Символ	0	1	2	3	4	5	6	7
1	b	1	1	1	2	3	4	5	6
2	a	2	1	2	2	3	4	4	5
3	c	3	2	2	2	3	4	5	6
4	a	4	3	3	3	3	4	4	5
5	b	5	4	3	4	4	3	4	5
6	a	6	5	4	4	5	4	3	4

## Пример 2

Строка  $S_1 = aababab$  преобразуется в  $S_2 = abbaa$  за три операции (массив  $D$  приведен в табл. 4.2). Последней операцией по преобразованию  $S_1$  в  $S_2$  может быть как замена символа  $b$  на символ  $a$  — мы переходим из  $D[4, 6]$  в  $D[5, 7]$ , так и удаление символа  $b$  из  $S_1$  — переходим из  $D[5, 6]$  в  $D[5, 7]$ . Но и в том, и в другом случае преобразование ( $S_1[1..6]$  в  $S_2[1..5]$  и  $S_1[1..6]$  в  $S_2[1..4]$ ) осуществляется за две операции.

Таблица 4.2

$D$	Номер символа	0	1	2	3	4	5	6	7
Номер символа	Символ		a	a	b	a	b	a	b
0	Символ	0	1	2	3	4	5	6	7
1	a	1	0	1	2	3	4	5	6
2	b	2	1	1	1	2	3	4	5
3	b	3	2	2	1	2	2	3	4
4	a	4	3	2	2	1	2	2	3
5	a	5	4	3	3	2	2	2	3

Вернемся к обсуждению рекуррентного соотношения. Нам необходимо вычислить  $D[i, j]$  (минимальное количество операций по преобразованию  $S_1[1..i]$  в  $S_2[1..j]$ ), и известны значения  $D[i - 1, j]$ ,  $D[i, j - 1]$  и  $D[i - 1, j - 1]$ . При этом логически возможны четыре случая:

- вставка символа  $S_2[j]$  на место  $i$  в  $S_1$  (переход от  $D[i - 1, j]$  к  $D[i, j]$  за одну операцию);
- удаление символа  $S_1[i]$  из  $S_1$  (переход от  $D[i, j - 1]$  к  $D[i, j]$  за одну операцию);
- замена символа  $S_1[i]$  на символ  $S_2[j]$  (переход от  $D[i - 1, j - 1]$  к  $D[i, j]$  за одну операцию);
- отсутствие операции —  $S_1[i] = S_2[j]$  (переход от  $D[i - 1, j - 1]$  к  $D[i, j]$  за нулевое количество операций).

Выбор минимального из этих чисел дает искомое значение количества операций по преобразованию  $S_1[1..i]$  в  $S_2[1..j]$ . Другие способы преобразований (а они, конечно, есть, например удаление всех символов из  $S_1[1..i]$ , а затем вставка символов, присутствующих в  $S_2[1..j]$ , — количество операций  $i + j$ : мы идем вначале по горизонтали до столбца с номером  $j$ , а затем по вертикали до строки  $i$ ) лишают нас свойства минимальности.

Временная сложность такого алгоритма —  $O(n \cdot m)$ , она пропорциональна размеру заполняемого массива  $D$ , который фактически определяет и объем памяти, необходимый для решения задачи.

Заметим, что при всем этом мы вычисляем расстояние, но не последовательность операций по преобразованию  $S_1$  в  $S_2$ ! В последнем же случае используются два стандартных приема (считаем, что массив  $D$  сформирован), ставших уже традиционными. Определим их как обратный просмотр массива расстояний  $D$ .

В первом случае дополнительная память не используется, а работа производится только с массивом  $D$ . Определим в качестве «соседей» элемента  $D[i, j]$  элементы  $D[i - 1, j]$ ,  $D[i, j - 1]$  и  $D[i - 1, j - 1]$ , ( $i > 0, j > 0$ ). Начнем с элемента  $D[n, m]$  и выберем его «соседа» с минимальным значением  $D$ . Если несколько «соседей» имеют одно и то же (минимальное) значение  $D$  и среди них есть  $D[i - 1, j - 1]$ , то выбираем его (осуществляем переход к этому элементу  $D$ ).

Если же  $D[i - 1, j - 1]$  нет, то выбираем любой соседний элемент — при решении задачи о выводе одной последовательности операций по преобразованию  $S_1$  в  $S_2$ . (При поиске же *всех* последовательностей операций по преобразованию  $S_1$  в  $S_2$  эту «развилку» следует запомнить, чтобы затем, после того как будет найдена очередная последовательность операций, вернуться к ней.) Для выбранного элемента  $D$  действия аналогичны, и этот процесс продолжается до тех пор, пока не будет достигнут элемент  $D[1, 1]$ . В табл. 4.1 жирным шрифтом и серым цветом фона был выделен этот обратный путь по  $D$ . Переход от  $D[i, j]$  к  $D[i, j - 1]$  при этом соответствует удалению (Dl) символа  $S_1[i]$ ; переход от  $D[i, j]$  к  $D[i - 1, j]$  — вставке (In) символа  $S_2[j]$  после  $S_1[i]$ ; наконец, переход от  $D[i, j]$  к  $D[i - 1, j - 1]$  соответствует подстановке (Re) символа  $S_1[i]$  на символ  $S_2[j]$  или отсутствию операции (Ma). Так, в примере, показанном в табл. 4.1, строка  $S_1 = abcdbad$  преобразуется в  $S_2 = bacaba$  последовательностью операций ReReMaReMaMaDl, и эта последовательность минимальна. Во втором рассматриваемом примере (см. табл. 4.2) строка  $S_1 = aababab$  преобразуется в  $S_2 = abbaa$  последовательностями операций MaReMaMaDlMaDl и MaDlMaDlMaMaRe.

Во втором же случае используется дополнительная память — массив указателей  $Dp[0..n, 0..m]$ , формируемый при заполнении массива  $D$ . Фактически в нем в каждом элементе  $Dp[i, j]$  запоминается (так или иначе) информация о том, от какого «соседа» был осуществлен переход к  $D[i, j]$ . Так, если считать, что от  $D[i, j - 1]$  к  $D[i, j]$  это значение равно единице (операция Dl), от  $D[i - 1, j]$  к  $D[i, j]$  — двум (операция In), а от  $D[i - 1, j - 1]$  к  $D[i, j]$  — четырем (операции Re или Ma), то логику получения  $D$  следует дополнить фрагментом:

```

If  $D[i, j-1]=D[i, j]-1$  Then  $Dp[i, j]:=Dp[i, j]+1;$ 
If  $D[i-1, j]=D[i, j]-1$  Then  $Dp[i, j]:=Dp[i, j]+2;$ 
If  $D[i-1, j-1]=D[i, j]-4$  Then  $Dp[i, j]:=Dp[i, j]+4;$ 

```

После того как массивы  $D$  и  $Dp$  сформированы, остается выполнить обратный просмотр от элемента  $Dp[n, m]$  по указателям до элемента  $Dp[1, 1]$ , записывая при этом операции, соответствующие переходам. Время работы этого фрагмента логики —  $O(n + m)$ .



## Упражнения

1. Приведите примеры строк  $S_1$  и  $S_2$ . Найдите все способы преобразования  $S_1$  в  $S_2$ . Выберите из них те, которые решают задачу за минимальное количество операций.
2. Приведите примеры строк  $S_1$  и  $S_2$ . Сформируйте вручную массив расстояний  $D$ . Выполните обратный просмотр массива  $D$ .
3. Разработайте алгоритм вычисления расстояния Хемминга между двумя строками длины  $n$ .
4. При известном массиве  $D$  найдите последовательность операций по преобразованию  $S_1$  в  $S_2$ .
5. При известных массивах  $D$  и  $D_p$  найдите все последовательности операций по преобразованию  $S_1$  в  $S_2$ .
6. *Взвешенное расстояние.* Обозначим через  $ds$  «стоимость» операций вставки и удаления символа, через  $rs$  — «стоимость» операции подстановки или замены, а через  $es$  — «стоимость» совпадения. Ставится задача перевода строки  $S_1$  в строку  $S_2$  с минимальной суммарной «стоимостью» операций (с минимальным взвешенным расстоянием). Проверьте приведенную ниже логику и убедитесь (на конкретном примере), что в этом случае вычисляется как взвешенное расстояние, так и массив указателей  $D_p$ .

**Procedure** Created;

**Var** i, j, t:Word;

**Begin**

  D[0, 0]:=0;

**For** i:=1 **To** n **Do Begin**

    D[i, 0]:=i\*ds; Dp[i, 0]:=2; **End;**

**For** j:=1 **To** m **Do Begin**

    D[0, j]:=j\*ds; Dp[0, j]:=1; **End;**

**For** i:=1 **To** n **Do**

**For** j:=1 **To** m **Do Begin**

**If** S<sub>1</sub>[i]=S<sub>2</sub>[j] **Then** t:=es **Else** t:=rs;

      D[i, j]:=Min(D[i-1, j]+ds, D[i, j-1]+ds,

        D[i-1, j-1]+t);



```

If D[i,j-1]=D[i,j]-ds Then
    Dp[i,j]:=Dp[i,j]+1;
If D[i-1,j]=D[i,j]-ds Then
    Dp[i,j]:=Dp[i,j]+2;
If D[i-1,j-1]=D[i,j]-t Then
    Dp[i,j]:=Dp[i,j]+4;
End;
End;

```

7. Выясните, какая задача решается с помощью приведенной логики (и как именно). Упростите решение (например, написав рекурсивный вариант реализации).

```

Procedure FindAllWay;
Var s:String;
    ok:Boolean;
    St:Array[1..2*MaxN,1..2] Of Word; {Стек}
    Stm:Array[1..2*MaxN] Of Set Of 1..3;
    yk:Word; {Указатель стека}
    it,jt,k,mask:Word;
Begin
    s:='';
    yk:=1;
    St[yk,1]:=n;
    St[yk,2]:=m;
    Stm[yk]:=[1..3];
    While yk<>0 Do Begin {Пока стек не пуст}
        it:=St[yk,1];
        jt:=St[yk,2];
        If (it=0) And (jt=0) Then Begin
            {Нашли одно из решений}
            WriteLn(s);
            yk:=yk-1;
            Delete(s,1,1);
            Continue;
        End;
        k:=0;
        ok:=False;
        Repeat
            {Выбор направления перемещения}
            k:=k+1;

```

```

mask:=(1 Shl (k-1));
If (Dp[it,jt]) (And Mask = Mask)
                And (k In Stm[yk])
                    Then ok:=True;
Until ok Or (k>=3);
If ok Then Begin {Перемещение найдено}
    Stm[yk]:=Stm[yk]-[k];
    yk:=yk+1;
    Stm[yk]:=[1..3];
    Case k Of
        1: Begin
            St[yk,1]:=it;St[yk,2]:=jt-1;
            s:='In'+s;
            End;
        2: Begin
            St[yk,1]:=it-1;St[yk,2]:=jt;
            s:='Dl'+s;
            End;
        3: Begin
            St[yk,1]:=it-1;St[yk,2]:=jt-1;
            If S1[it]<>S2[jt] Then s:='Re'+s
                Else s:='Ma'+s;
            End;
    End;
End Else Begin
    yk:=yk-1;
    Delete(s,1,1);
    End;
End;
End;

```

8. *Сходство строк.* Пусть алфавит  $A$  дополнен символом пробела (обозначим его как « $\_$ »), и для любых двух символов  $x, y \in A^*$  ( $A^* = A + \text{«}\_ \text{»}$ ) определена оценка  $rs[x, y]$  (от слова *resemblance* — «сходство»), задающая сходство символов  $x$  и  $y$  при размещении их друг под другом при выравнивании. *Выравнивание* двух строк  $S_1$  и  $S_2$  получается вставкой символа пробела в  $S_1$  и  $S_2$  и размещением строк друг под другом так, чтобы каждому символу первой строки (включая пробелы) соответствовали символы второй строки, и наоборот.

Пример

$A = \{a, b, c\}$ ,  $S_1 = aabbcc$ ,  $S_2 = aacb$ ,  $\Rightarrow$

$$rs = \begin{pmatrix} 1 & -2 & -4 & -3 \\ & 3 & -1 & -2 \\ & & 2 & -5 \\ & & & 0 \end{pmatrix} \cdot \begin{matrix} \text{ЛАНЬ} \\ \text{ЛАНЬ} \\ \text{ЛАНЬ} \\ \text{ЛАНЬ} \end{matrix}$$

Выравнивание  $a a b b c c$  имеет суммарную оценку  
 $a a \_ \_ c c$

$1 + 1 - 2 - 2 + 2 - 5 = -5$ , а выравнивание  $a a b b c c$  по-  
 $a a c b \_ \_$   
 лучает суммарную оценку  $1 + 1 - 1 + 3 - 5 - 5 = -6$ .

Сходство строк  $S_1$  и  $S_2$  определяется как максимальное суммарное значение оценок, получающееся в результате выполнения различных способов выравнивания  $S_1$  и  $S_2$  относительно друг друга.

Обозначим через  $R[i, j]$  значение максимальной оценки для префиксов  $S_1[1..i]$  и  $S_2[1..j]$ . Тогда значение  $R[n, m]$  дает максимальную оценку в целом для строк  $S_1$  и  $S_2$ .

1. Обоснуйте справедливость рекуррентного соотношения:

$$R[0, j] = \sum_{1 \leq k \leq j} rs[_ , S_2[k]] \text{ и } R[i, 0] = \sum_{1 \leq k \leq i} rs[S_1[k], _];$$

$$R[i, j] = \max \left\{ \begin{array}{l} R[i-1, j-1] + rs[S_1[i], S_2[j]], \\ R[i-1, j] + rs[S_1[i], _], \\ R[i, j-1] + rs[_ , S_2[j]] \end{array} \right\}.$$

2. Разработайте программу вычисления сходства строк и нахождения всех способов выравнивания, при которых обеспечивается это сходство.

## 4.2. Алгоритм Э. Укконена – Ю. Майерса

Вперед, в рукопашную!

(М/ф «Остров сокровищ»)

Уточним меру близости между строками  $S_1$  и  $S_2$ , введенную в п. 4.1. Заменим (условно) операцию подстановки или замены символа (Replace) на две операции — вставку

(Insert) и удаление (Delete) символов. Другими словами, операция Re будет иметь вес 2. Вычислим матрицу расстояний  $D$ . Напомним рекуррентное соотношение для  $D[i, j]$  при  $1 \leq i \leq n$  и  $1 \leq j \leq m$ :

$$D[i, j] = \min\{D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + t\},$$

где  $t = 2$ , если  $S_1[i] \neq S_2[j]$ , и  $t = 0$ , если  $S_1[i] = S_2[j]$  (инициализация  $D$  осуществляется так же, как и в п. 4.1).

Для строк  $S_1 = \text{abcaadbdbd}$  и  $S_2 = \text{bcadabbbdc}$  матрица  $D$  представлена в табл. 4.3. Значение  $D[n, m]$  здесь равно 4, т. е. за четыре операции строка  $S_1$  может быть преобразована в строку  $S_2$ , и это — минимальное количество операций.

Таблица 4.3

$D$	0	b	c	a	d	a	b	b	d	c
0	0	1	2	3	4	5	6	7	8	9
a	1	2	3	2	3	4	5	6	7	8
b	2	1	2	3	4	5	4	5	6	7
c	3	2	1	2	3	4	5	6	7	6
a	4	3	2	1	2	3	4	5	6	7
a	5	4	3	2	3	2	3	4	5	6
d	6	5	4	3	2	3	4	5	4	5
b	7	6	5	4	3	4	3	4	5	6
b	8	7	6	5	4	5	4	3	4	5
d	9	8	7	6	5	6	5	4	3	4

Представим эту информацию в виде графа  $G = (V, E)$ , где  $V$  — конечное множество вершин, которые соответствуют индексам  $(i, j)$ ,  $i \in 0 \dots n$ ,  $j \in 0 \dots m$ , а  $E$  — дуги графа. Каждая вершина  $(i, j)$  имеет не более трех входящих дуг (кроме вершины  $(0, 0)$ ), направленных из вершины  $(i-1, j)$ , из вершины  $(i, j-1)$  и из вершины  $(i-1, j-1)$ . Каждой такой дуге мы припишем вес, равный разности между  $D[i, j]$  и  $D[i', j']$ . Вес же самой вершины  $(i, j)$  определим как значение  $D[i, j]$ .

Вес вершины  $(i, j)$  определяет минимальное количество операций по преобразованию префикса  $S_1[1..i]$  в префикс  $S_2[1..j]$  и равен сумме весов дуг ориентированного пути из  $(0, 0)$  в  $(i, j)$ . Тогда путь с минимальным суммарным весом из

$(0, 0)$  в  $(n, m)$  определяет наилучший способ преобразования  $S_1$  в  $S_2$ . Заметим, что в подобном графе нет дуг от вершин с большим весом к вершинам с меньшим весом. На рис. 4.1 показан граф  $G$ , соответствующий матрице  $D$  для рассматриваемого примера.

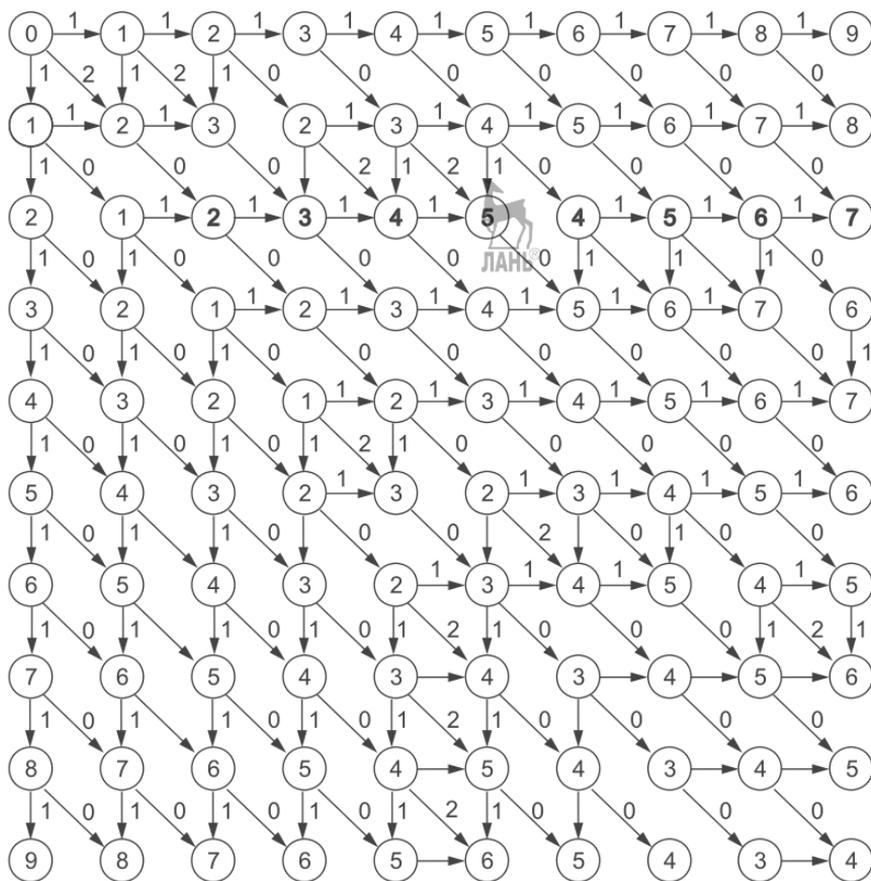


Рис. 4.1. Граф  $G$ , соответствующий матрице  $D$

Пока мы представили исходные данные задачи в другом виде, и только. Конечно, у нас есть возможность использовать алгоритмы теории графов по поиску кратчайших путей<sup>1)</sup>, но их временная сложность сопоставима с временем формирования  $D$ , так что на этом пути ускорение не достигается.

<sup>1)</sup> Окулов С. М. Дискретная математика. Теория и практика решения задач по информатике. — М.: БИНОМ. Лаборатория знаний, 2008.

Обратимся вновь к графу  $G$  и воспользуемся очевидным наблюдением. На пути с минимальным суммарным весом из  $(0, 0)$  в  $(n, m)$  не может быть таких вершин  $(i, j)$ , что  $D[i, j] > D[n, m]$ . Если теперь из  $G$  убрать все вершины с таким весом и все пути, ведущие только к ним, то получится сокращенный (уменьшенный) граф. Для рассматриваемого примера он приведен на рис. 4.2. Что он дает? Очередное наблюдение: вершины «концентрируются» вокруг главной диагонали, которая содержит не более чем  $n$  дуг (считаем, что  $n \leq m$ ). Интуитивно ясно, что вершины сокращенного графа не могут отступать от этой диагонали более чем на  $d = D[n, m]$  шагов (дуг). Даже более точно — на  $\left\lfloor \frac{d}{2} \right\rfloor$  шагов.

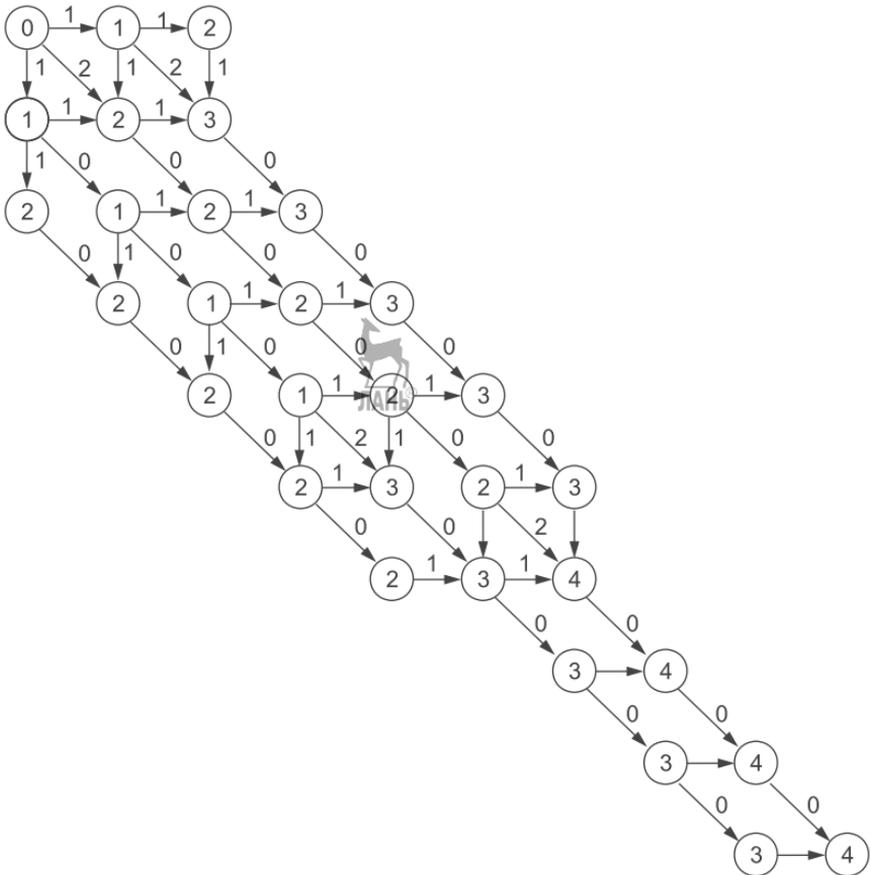


Рис. 4.2. Сокращенный вариант графа  $G$

Тогда общее количество вершин в сокращенном графе имеет порядок  $O(n \cdot d)$ . И вот у нас очередной момент «эврика»: что если строить сразу сокращенный граф, тогда время его построения (естественно, с получением оценок) будет пропорционально количеству вершин, а задача решается за  $O(n \cdot d)$ ? Проверим эту идею для «крайних» случаев.

### Пример 1

Пусть  $S_1 = abcde$ ,  $S_2 = fghijk$ . Матрица  $D$  представлена в табл. 4.4.



Таблица 4.4

$D$	0	f	g	h	i	j	k
0	0	1	2	3	4	5	6
a	1	2	3	4	5	6	7
b	2	3	4	5	6	7	8
c	3	4	5	6	7	8	9
d	4	5	6	7	8	9	10
e	5	6	7	8	9	10	11



Сокращенный граф  $G$  здесь совпадает с исходным, но если известна верхняя оценка  $d$ , то, пройдя по главной диагонали один раз (по соответствующему пути в графе), можно найти значение  $d$ .

### Пример 2

Пусть  $S_1 = aaaaa$ ,  $S_2 = aaaaaa$ . Матрица  $D$  приведена в табл. 4.5.

Таблица 4.5

$D$	0	a	a	a	a	a	a
0	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
a	2	1	0	1	2	3	4
a	3	2	1	0	1	2	3
a	4	3	2	1	0	1	2
a	5	4	3	2	1	0	1

А здесь — эффект максимальный! От главной диагонали, идущей от  $(0, 0)$  до  $(n, n)$ , мы отступаем только на один шаг вверх и вниз (считаем, что  $n \leq m$ ).

Но как строить сокращенный граф  $G$ ? А если его вообще не строить, а просто заполнять матрицу  $D$  лишь частично в той мере, в которой она будет соответствовать сокращенному графу, точнее, предположению об его структуре при известной оценке  $d$ ? В этом случае при заполнении  $D$  для каждого значения  $i$  (номера строки) следует формировать не все элементы строки (индекс  $j$ ), а только от нижнего значения (обозначим его как  $a$ ) до верхнего значения (обозначим его как  $b$ ).

Величины  $a$  и  $b$  определяются оценкой  $d$  и размерами матрицы  $n$  и  $m$ . Пусть  $m$  равно  $n$ . Тогда при  $i = 1$  значение  $j$  должно изменяться от 1 до  $1 + d$  (на  $d$  диагоналей вверх), а при  $i = n$  — от  $n - d$  (на  $d$  диагоналей вниз) до  $n$ . Обобщая, получаем:  $a = \max(1, i - d)$  и  $b = \min(n, i + d)$ . Осталось учесть случай неравенства  $m$  и  $n$ . Но тогда при достижении последней строки по главной диагонали необходимо сделать  $m - n$  ( $n < m$ ) шагов вправо по строке. Эта разность и определяет окончательные границы индекса  $j$  при каждом значении  $i$ .

Оформим эту логику в виде функции Trail.

```
Function Trail(d:Integer):Integer;
  Var i,j,q,t:Integer;
  Begin
    q:=|d-(m-n) Div 2|;
    For i:=1 To n Do
      For j:= Max(1,i-q) To Min(m,i+q+m-n) Do Begin
        D[i,j]:=Min(D[i-1,j]+1,D[i,j-1]+1);
        If S1[i]=S2[j] Then t:=0 Else t:=2;
        D[i,j]:=Min(D[i,j],D[i-1,j-1]+t);
      End;
    Trail:=D[n,m];
  End;
```

В основной процедуре остается только инициализировать  $D$  и вычислять  $D[n, m]$  до тех пор, пока это значение не станет больше оценки  $d$ , которую мы увеличиваем вдвое на каждой итерации:

**Procedure** Dist;

**Var**  $i, j, d$ :Integer;

**Begin**

$D[0, 0] := 0$ ;

**For**  $i := 1$  **To**  $n$  **Do**  $D[i, 0] := D[i-1, 0] + 1$ ;

**For**  $j := 1$  **To**  $m$  **Do**  $D[0, j] := D[0, j-1] + 1$ ;

$d := 1$ ;

**While**  $d < \text{Trail}(d)$  **Do**  $d := 2 * d$ ;

**WriteLn**( $D[n, m]$ );

**End**;

Время выполнения функции `Trail` составляет  $O(n \cdot d)$ , а выполняется она для значений  $d$ , равных  $1, 2, \dots, 2^t$ , где  $t$  — наименьшее целое, такое, что  $d \leq 2^t$ . Следовательно, общее время работы `Trail` имеет такую же оценку.



### Упражнения

1. Приведите пример двух строк. Подсчитайте для них матрицу  $D$ . Постройте граф  $G$  и его сокращенный вариант.
2. Выполните программную реализацию алгоритма Э. Укконена и Ю. Майерса.
3. Разработайте модификацию алгоритма Э. Укконена и Ю. Майерса, обеспечивающую не только поиск наибольшей общей подпоследовательности, но и вывод самой этой подпоследовательности.
4. Пусть известна верхняя оценка для  $d$ . Модифицируйте алгоритм Э. Укконена и Ю. Майерса для этого случая.
5. Представим граф, соответствующий строкам  $S_1$  и  $S_2$ , несколько иначе. Пусть в нем диагональное ребро от вершины  $(i-1, j-1)$  к вершине  $(i, j)$  идет только в том случае, если  $S_1[i] = S_2[j]$ . Для строк  $S_1 = \text{abcaadbdbd}$  и  $S_2 = \text{bcadabbdbc}$  такой граф представлен на рис. 4.3. Путь длиной  $L$  — это путь, содержащий  $L$  вершин графа  $(i_1, j_1), (i_2, j_2), \dots, (i_L, j_L)$ , для которых  $S_1[i_k] = S_2[j_k]$  ( $k = 1..L$ ) и  $i_{k-1} < i_k, j_{k-1} < j_k$ . Тогда проблема нахождения наибольшей общей подпоследовательности  $S_1$  и  $S_2$  эквивалентна нахождению пути из вершины  $(0, 0)$  в вершину  $(n, m)$  наибольшей длины или содержащего наибольшее количество диагональных ребер. Обозначим

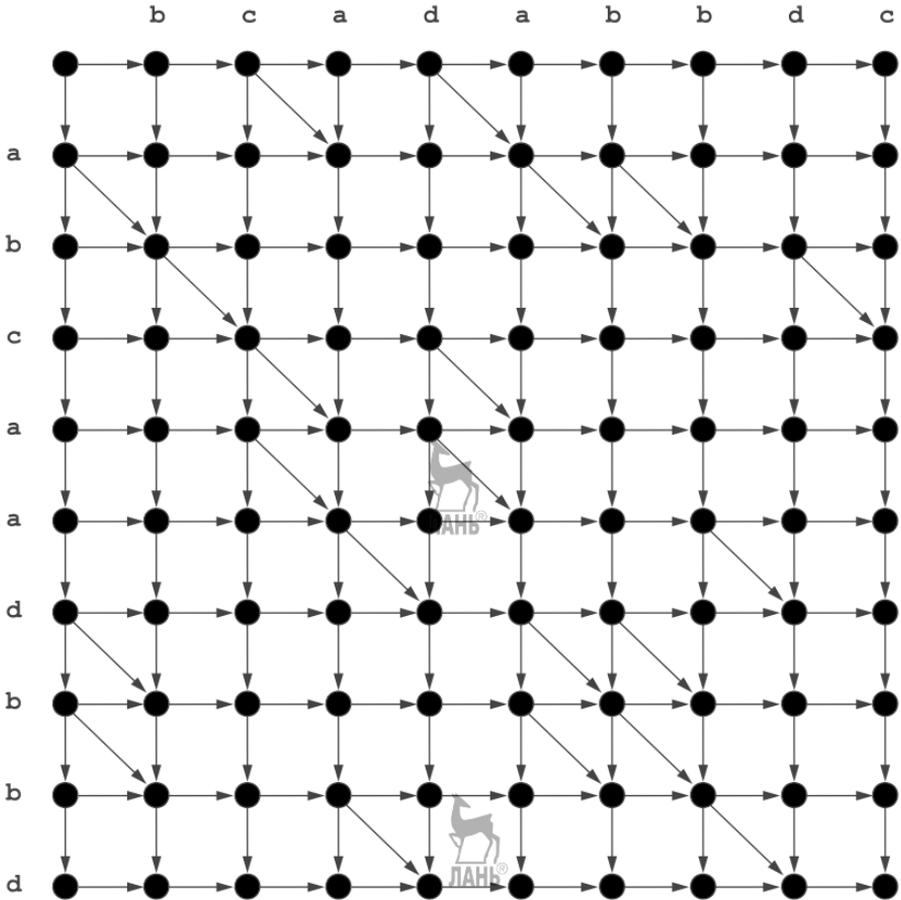


Рис. 4.3. Граф, отражающий связи между символами строк  $S_1 = abcaadbbd$  и  $S_2 = bcadabbdc$

как  $d$ -путь некоторый путь, начинающийся в  $(0, 0)$  и содержащий  $d$  недиагональных ребер. Тогда  $0$ -путь состоит только из диагональных ребер, а  $d$ -путь строится из  $(d - 1)$ -пути добавлением еще одного недиагонального ребра и возможной последовательности диагональных ребер. Процесс построения  $d$ -путей заканчивается при достижении вершины  $(n, m)$ . Значение  $d$  при этом дает количество операций по преобразованию  $S_1$  в  $S_2$ , а количество диагональных ребер — длину наибольшей общей подпоследовательности.

Выясните, какая задача решается с помощью нижеприведенной процедуры *Solve*. Исследуйте вопрос о ее улучшении (например, с точки зрения возможности вывода наибольшей общей подпоследовательности, а не только количества операций по преобразованию  $S_1$  в  $S_2$  или ее длины):

```

Procedure Solve;
Var d, j, x, y: Integer;
Begin
  Res[1]:=0;
  {Массив Res имеет тип
   TArray=Array[-NMax..NMax] Of Integer;}
  For d:=0 To Max(n,m) Do Begin
    {Max - функция определения максимального
     из двух чисел}
    For j:=-d To d Do
      If (Abs(j) Mod 2)=(d Mod 2) Then Begin
        If (j=-d) Or (j<>d) And (Res[j-1]<Res[j+1])
          Then x:=Res[j+1]
          Else x:=Res[j-1]+1;
        y:=x-j;
        {Идем по диагонали от конца предыдущего
         участка на диагонали j}
        While (x<n) And (y<m)
          And (S1[x+1]=S2[y+1]) Do
            Begin
              x:=x+1;
              y:=y+1;
            End;
        Res[j]:=x;
        {Запоминаем конец диагонального участка}
        If (x>=n) And (y>=m) Then Begin
          WriteLn(d); {WriteLn(Max(n,m)-d);}
          Exit;
        End;
      End;
    End;
  End;
End;

```

### 4.3. Задача о наибольшей общей подпоследовательности двух строк

Нужно иметь что-то общее, чтобы понимать друг друга, и чем-то отличаться, чтобы любить друг друга.

*Поль Жеральди*

#### 4.3.1. Простой алгоритм решения



А супруга у меня простая, из народа.

*Сергей Довлатов*

Определим по имеющейся строке  $S$  строку  $S'$  как  $S' = S[i_1]S[i_2]...S[i_k]$ , где  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  и  $1 \leq k \leq n$ . О строке  $S'$  мы в этом случае будем говорить как о подпоследовательности  $S$  и считать, что пустая строка  $\varepsilon$  также является подпоследовательностью  $S$ .

Если даны две строки  $S_1$  и  $S_2$ , то возникает задача о нахождении *наибольшей общей подпоследовательности* этих строк —  $lcs(S_1, S_2)$  (*Longest Common Subsequence* — «наибольшая общая подпоследовательность»).

*Пример*

$$S_1 = abcdefgk, S_2 = bdqfk \Rightarrow lcs(S_1, S_2) = bdfk.$$

Определим  $V[i, j]$  как длину наибольшей общей подпоследовательности префиксов  $S_1[1..i]$  и  $S_2[1..j]$  —  $|lcs(S_1[1..i], S_2[1..j])|$ . Как она формируется? Если известно значение  $V[i-1, j-1]$  и  $S_1[i] = S_2[j]$ , то очевидно равенство  $V[i, j] = V[i-1, j-1] + 1$  — длина наибольшей общей подпоследовательности увеличивается на единицу. Если же нет —  $S_1[i] \neq S_2[j]$  и определены значения  $V[i-1, j]$  и  $V[i, j-1]$  ( $|lcs(S_1[1..i-1], S_2[1..j])|$  и  $|lcs(S_1[1..i], S_2[1..j-1])|$ ), то с той же очевидностью следует, что  $V[i, j]$  равно максимальному из чисел  $V[i-1, j]$  и  $V[i, j-1]$ .

Итак:

- $V[0, j] = 0, 0 \leq j \leq m;$
- $V[i, 0] = 0, 0 \leq i \leq n;$
- $V[i, j] = \begin{cases} V[i-1, j-1] + 1, \text{ если } S_1[i] = S_2[j]; \\ \max(V[i-1, j], V[i, j-1]) & \text{— в противном случае.} \end{cases}$

Запись этого алгоритма в формализованном виде проста и не требует пояснений.

```

Procedure CreateV;
  Var i, j: Word;
  Begin
    For i:=1 To n Do V[i, 0] := 0;
    For j:=1 To m Do V[0, j] := 0;
    For i:=1 To n Do
      For j:=1 To m Do
        If S1[i]=S2[j] Then V[i, j] := V[i-1, j-1] + 1
        Else V[i, j] := max(V[i-1, j], V[i, j-1]);
    {Функция находит максимальное из двух чисел}
  End;

```

### Пример

Пусть  $S_1 = \text{abcaadbdbd}$  и  $S_2 = \text{bcadabdbdc}$ . В табл. 4.6 показан массив  $V$ . Длина наибольшей общей подпоследовательности равна 7. Но их — две:  $\text{bcaabdbd}$  и  $\text{bcadbdbd}$ . Для их вывода в процессе получения  $V$  следует формировать и массив указателей  $Vp$  (аналогично массиву  $Dp$  из п. 4.1), а затем выполнить обратный просмотр от элемента  $Vp[n, m]$  к  $Vp[1, 1]$ . В табл. 4.6 последовательность переходов при обратном просмотре выделена серым фоном.

Таблица 4.6

V	Номер символа	0	1	2	3	4	5	6	7	8	9
Номер символа	Символ		a	b	c	a	a	d	b	b	d
0		0	0	0	0	0	0	0	0	0	0
1	b	0	0	1	1	1	1	1	1	1	1
2	c	0	0	1	2	2	2	2	2	2	2
3	a	0	1	1	2	3	3	3	3	3	3
4	d	0	1	1	2	3	3	4	4	4	4
5	a	0	1	1	2	3	4	4	4	4	4
6	b	0	1	2	2	3	4	4	5	5	5
7	b	0	1	2	2	3	4	4	5	6	6
8	d	0	1	2	2	3	4	5	5	6	7
9	c	0	1	2	3	3	4	5	5	6	7

Время работы этого алгоритма пропорционально  $O(n \cdot m)$ , так же как и объем используемой памяти. Если речь идет только о длине наибольшей общей подпоследовательности, то достаточно очевидно его упрощение. Длина  $lcs$  равна  $V[n, m]$ , и при формировании очередной строки  $V$  достаточно знания только предыдущей строки, поэтому требование к памяти может быть уменьшено до значения  $O(\min(n, m))$ . С этой целью в логику формирования  $V$  вводится дополнительная переменная  $q$ , с помощью которой осуществляется переадресация между двумя одномерными массивами размерности  $\min(n, m)$ . Приведем формальную запись (процедура CreateM), поскольку она потребуется нам в дальнейших поисках улучшений первоначального решения:



```
Const NMax=...;
```

```
Type
```

```
TArray=Array[0..NMax] Of Integer;
```

```
FArray=Array[0..1] Of TArray;
```

```
...
```

```
Procedure CreateM(S1,S2:String;Var V:TArray);
```

```
{Параметры этой процедуры потребуются  
в дальнейшем изложении}
```

```
Var Vs:FArray;
```

```
i, j, q, n, m: Integer;
```

```
Begin
```

```
n:=Length(S1);
```

```
m:=Length(S2);
```

```
For j:=0 To m Do Vs[0, j]:=0;
```

```
{Инициализация данных}
```

```
Vs[1, 0]:=0;
```

```
q:=0;
```

```
{Переключатель строк массива Vs}
```

```
For i:=1 To n Do Begin
```

```
q:=1-q;
```

```
For j:=1 To m Do
```

```
  If S1[i]=S2[j] Then Vs[q, j]:=Vs[1-q, j-1]+1
```

```
    Else Vs[q, j]:=Max(Vs[1-q, j], Vs[q, j-1]);
```

```
End;
```

```
V:=Vs[q];
```

```
{Результат - последняя сформированная строка}
```

```
End;
```



Итак, требования по памяти сокращены, результат определяется значением  $V[q, m]$ , но возможности обратного просмотра с целью вывода самой  $lcs$  мы лишились. А можно ли и в этом случае найти не только длину  $lcs$ , но и вывести саму последовательность? Естественный путь поиска: разбивка строк на подстроки, решение задач для подстрок, рекурсивная реализация этой схемы и склейка результатов на выходе из рекурсивного спуска. Вся сложность в том, как разбить строки  $S_1$  и  $S_2$ . Следует выбрать пару чисел  $(is, js)$  так, чтобы решение задачи для подстрок  $S_1[1..is]$ ,  $S_2[1..js]$  и  $S_1[is + 1..n]$ ,  $S_2[js + 1..m]$  (их склейка) давало результат для  $S_1$  и  $S_2$ . Аналогичную пару чисел следует находить и для каждой следующей пары подстрок. Такая задача, при успешности нахождения  $(is, js)$ , обладала бы свойством аддитивности.

Пусть, как и ранее,  $S_1 = abcaadbdbd$  и  $S_2 = bcadabbdbc$ . Разобьем  $S_1$  на две подстроки  $S_{11}$  и  $S_{12}$  (делением пополам —  $is := n \text{ Div } 2$ ). Для каждой пары  $(S_{11}, S_2)$  и  $(S_{12}, S_2)$  «прогоним» логику CreateM. Результаты представлены в табл. 4.7 и табл. 4.8.

Таблица 4.7

V	Номер символа	0	1	2	3	4	5	6	7	8	9
Номер символа	Символ		b	c	a	d	a	b	b	d	c
0		0	0	0	0	0	0	0	0	0	0
1	a	0	0	0	1	1	1	1	1	1	1
2	b	0	1	1	1	1	1	2	2	2	2
3	c	0	1	2	2	2	2	2	2	2	3
4	a	0	1	2	3	3	3	3	3	3	3

Подстроки  $abca$  и  $bcadabbdbc$  имеют наибольшую общую подпоследовательность длины 3, и в нашем распоряжении осталась только последняя строка с результатом (обозначим ее как  $V_1$ ). Подстроки  $adbdb$  и  $bcadabbdbc$  имеют  $lcs$  длины 5, и у нас есть массив с результатом  $V_2$ . А что дальше? Можно разбивать  $S_2$  по такому же принципу, но ощущения гарантированности результата все же не возникает.



Таблица 4.8

V	Номер символа	0	1	2	3	4	5	6	7	8	9
Номер символа	Символ		b	c	a	d	a	b	b	d	c
0		0	0	0	0	0	0	0	0	0	0
1	a	0	0	0	1	1	1	1	1	1	1
2	d	0	0	0	1	2	2	2	2	2	2
3	b	0	1	1	1	2	2	3	3	3	3
4	b	0	1	1	1	2	2	3	4	4	4
5	d	0	1	1	1	2	2	3	4	5	5

Проанализируем  $V_1$  (последняя строка в табл. 4.7). В ней неявным образом заложена информация о способах выравнивания  $S_{12}$  относительно  $S_2$  — точнее, о последнем символе  $S_{12}$  относительно  $S_2$ . Способы такого выравнивания приведены в табл. 4.9. Значение максимума в  $V_1$  достигается уже на элементе  $V_1[3]$ , что соответствует последней строке в табл. 4.9.

Таблица 4.9

	b	c	a	d	a	b	b	d	c	
					a	—	b	—	c	a
					a	b	—	—	c	a
			a	—	—	—	b	—	c	a
			a	—	—	b	—	—	c	a
a	b	c	—	—	a					
a	b	c	a							

Итак, значение 3 достигается уже на подстроках  $abca$  и  $bca$  —  $V_1[3]$ . Если из  $S_2$  убрать  $bca$ , то останется строка  $dabbbdc$ . Длина наибольшей общей подпоследовательности  $S_{12}$  ( $adbdbd$ ) и  $dabbbdc$  равна 4 —  $V_2[m] - V_2[3]$ . Возникает предположение о том, что в качестве  $js$  следует взять значение  $j$ , при котором достигается максимум величины  $V_1[j] + V_2[m] - V_2[j]$ . Проверим его (и идею в целом), ибо его формализация не представляет собой сложности

(функция Lcs). Если она работоспособна<sup>1)</sup>, то весь основной код сведется к вводу строк и выводу результатов работы функции.

```

Function Lcs(S1,S2:String):String;
  Var n,m,i,j,is,js,max:Integer;
      V1,V2:TArray;
Begin
  n:=Length(S1);
  m:=Length(S2);
  If (n=0) Or (m=0) Then Lcs:=''
  Else If S1=S2 Then Lcs:=S1
  Else
    If n=1 Then Begin
      {Находим символ S1 в S2}
      j:=1;
      While (j<=m) And (S1[1]<>S2[j]) Do j:=j+1;
      If j>=m+1 Then Lcs:=''
      Else Lcs:=S2[j];
    End
  Else Begin
    is:=n Div 2;
    CreateM(Copy(S1,1,is),S2,V1);
    CreateM(Copy(S1,is+1,n-is),S2,V2);
    js:=0; max:=0;
    For j:=1 To m Do
      If (V1[j]<>0) And ((V1[j]+V2[m]-
                          V2[j])>max)
        Then Begin
          js:=j;
          max:=V1[j]+V2[m]-V2[j];
        End;
    Lcs:=Lcs(Copy(S1,1,is),Copy(S2,1,js)) +
          Lcs(Copy(S1,is+1,n-is),
              Copy(S2,js+1,m-js));
  End;
End;

```

<sup>1)</sup> В книге Б. Смита (с. 288–291) доказывается (леммы, теорема) правильность данного результата, но нас интересует не конечный результат как таковой, оформленный доказательно, а процесс его «рождения» (предполагаемый) у специалиста по информатике. Можно с определенной долей уверенности утверждать, что леммы и теоремы оформляются уже после его получения (и не всегда удачно).

Для рассматриваемого примера (и ему подобных) приведенная выше логика дает правильный результат. Разбивка на подстроки происходит так, как описано ранее, и мы получаем одну из общих подпоследовательностей, а именно `bcadbdbd`. Однако не будем торопиться. Возьмем строки из одной буквы  $a$  разной длины, например  $S_1 = aaaaaa$  и  $S_2 = aaaaa$ . При первом вызове функции `Lcs` значение  $is$  будет равно  $3$ , а  $js = 1$ , и рекурсивные вызовы `Lcs` выполняются с подстроками: первый —  $S_{11} = aaa$  и  $S_{21} = a$ ; второй —  $S_{12} = aaa$  и  $S_{22} = aaaa$ , что приводит к явно неточному результату, а именно  $lcs = aaaa$ . Вопрос о возможности доведения функции `Lcs` до работоспособного состояния вынесен в упражнения.

### Упражнения

1. Приведите примеры двух строк и вычислите вручную длину их наибольшей общей подпоследовательности. Выполните обратный просмотр по значениям из  $V$  и найдите все  $lcs$  (если их несколько).
2. Исключим в расстоянии В. Левенштейна операцию подстановки `Re`. Точнее, будем считать, что эта операция имеет вес  $2$  и выполняется путем удаления символа из  $S_1$  и последующей вставки символа, равного соответствующему символу из  $S_2$ . Покажите, что в этом случае  $d(S_1, S_2) = n + m - 2 \cdot lcs(S_1, S_2)$ .
3. Дополните логику вычисления длины наибольшей общей подпоследовательности  $S_1$  и  $S_2$  вычислением массива указателей и последующим выводом всех  $lcs(S_1, S_2)$ .
4. Исследуйте возможность доведения рекурсивной схемы вычисления  $lcs$  до работоспособного состояния.

#### 4.3.2. Алгоритм Д. Ханта – Т. Зиманского

Воображение важнее, чем знания. Знания ограничены, тогда как воображение охватывает целый мир, стимулируя прогресс, порождая эволюцию.

*Альберт Эйнштейн*

Данный алгоритм основан (как это, вероятно, и бывает с новыми результатами) на достаточно простой идее. Поясним ее с помощью примера. Представим рассматриваемые строки  $S_1 = abcaadbdbd$  и  $S_2 = bcadabbbdc$  на квадратной сет-

ке, как показано на рис. 4.4. Точками выделены узлы сетки, соответствующие совпадающим символам в  $S_1$  и  $S_2$ . Эти точки можно соединять друг с другом отрезками и получать различные ломаные линии. Совпадающим подпоследовательностям в  $S_1$  и  $S_2$  тогда соответствуют строго убывающие диагональные (без горизонтальных и вертикальных отрезков) линии, а наибольшей общей подпоследовательности соответствует линия, состоящая из наибольшего количества таких отрезков.

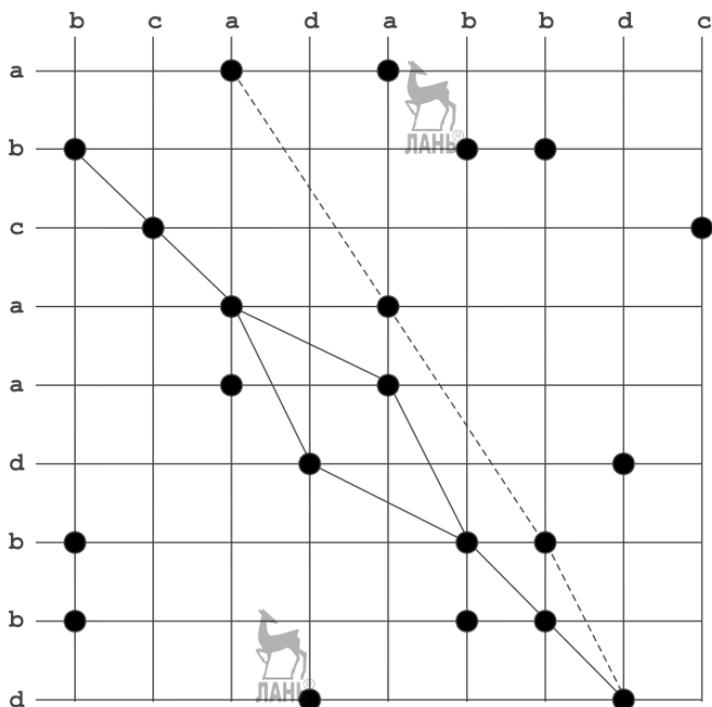


Рис. 4.4. Сетка для представления строк

Таким образом, задача о нахождении наибольшей общей подпоследовательности эквивалентна поиску максимальной монотонно возрастающей последовательности таких пар чисел  $(i, j)$ , что  $S_1[i] = S_2[j]$ . По рис. 4.4 для рассматриваемого примера ее очень просто выписать, хотя номера символов строк не приведены на рис. 4.4:  $(2, 1)$ ,  $(3, 2)$ ,  $(4, 3)$ ,  $(5, 5)$ ,  $(7, 6)$ ,  $(8, 7)$ ,  $(9, 8)$ . Но кроме максимальной последовательности на рис. 4.4 явно просматриваются и другие общие подпоследовательности  $S_1$  и  $S_2$ , одна из которых изображена пунктирной линией.

Следующим шагом является новый способ описания взаимосвязи символов в строках  $S_1$  и  $S_2$ . Введем двумерный массив  $V$ . Первым индексом ( $i$ ) в нем является номер символа в  $S_1$ , вторым ( $q$ ) — длина общей подпоследовательности префикса  $S_1[1..i]$  и  $S_2$ , а значением элемента массива  $V[i, q]$  является минимальное значение  $j$  (номера символа в  $S_2$ ), на котором длина наибольшей общей подпоследовательности префиксов  $S_1[1..i]$  и  $S_2[1..j]$  равна значению  $q$ .

Подсчитаем вручную значения, например для  $V[7]$ , т. е. для префикса строки  $S_1$ , равного  $abcaadb$ . Пусть  $q = 1$ . Минимальное значение  $j$ , для которого длина общей последовательности  $S_1[1..i]$  и  $S_2[1..j]$  равна  $q$ , есть единица —  $S_2[1..1]$ . Аналогично, для  $q = 2$  ( $bc$ ),  $3$  ( $bca$ ),  $4$  ( $bcad$ ). Для  $q = 5$  требуется взять префикс  $S_2$  из шести символов —  $bcadab$ . Если проделать такую же работу для всех оставшихся префиксов  $S_1$ , то мы получим массив следующего вида (при условии, что первоначально его элементам присваиваются значения  $m + 1$ , а элементам нулевого столбца — значения  $0$ ):

$$V = \begin{pmatrix} 0 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 3 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 1 & 6 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 9 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 10 & 10 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 5 & 10 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 4 & 8 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 4 & 6 & 10 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 4 & 6 & 7 & 10 & 10 & 10 \\ 0 & 1 & 2 & 3 & 4 & 6 & 7 & 8 & 10 & 10 \end{pmatrix}$$

Имея в своем распоряжении массив  $V$ , выписать максимальную монотонно возрастающую последовательность таких пар чисел  $(i, j)$ , что  $S_1[i] = S_2[j]$ , не составляет труда.

Что отражается в массиве  $V$ ? В целом — некое подобие пути наибольшей общей подпоследовательности двух строк  $S_1$  и  $S_2$ , где индекс  $i$  (номер строки) определяет номер символа  $S_1$ , индекс  $q$  (номер столбца) — номер символа в  $lcs$ , а значение  $V[i, q]$  — минимальное значение длины префикса  $S_2$ , при котором получается эта  $lcs$ .

Итак,  $V[i, q] = \min_{1 \leq j \leq |S_2|} (j)$ . Как формируется элемент  $V[i, q]$ ?

Можно утверждать, что значение  $V[i, q]$  находится в интервале от  $V[i - 1, q - 1]$  до  $V[i - 1, q]$  (проверьте это утверждение на приведенном массиве  $V$ ). Действительно, если  $S_1[i] \neq S_2[j]$ , то  $lcs$  не может быть продолжена символом  $S_1[i]$ , и  $V[i, q] = V[i - 1, q]$ . Рассмотрим теперь случай равенства  $S_1[i] = S_2[j]$  и разберемся с левой границей интервала. Значение  $j$  для наибольшей общей последовательности длины  $q$  не может быть меньше значения  $j$  для  $lcs$  длины  $q - 1$ . Действительно,  $lcs(S_1[1..i], S_2[1..V[i, q]])$  образуется путем добавления одного символа из  $S_1$  и  $S_2$  к  $lcs(S_1[1..i - 1], S_2[1..V[i - 1, q]])$ , т. е.  $V[i - 1, q - 1] < V[i, q]$ . Определим правую границу:  $V[i, q] \leq V[i - 1, q]$ . Так как наибольшая общая подпоследовательность длины  $q$  уже построена, то при формировании лучшей  $lcs$  этой длины ( $q$ ) необходимо взять символ из  $S_2$ , обладающий большим «потенциалом» создания  $lcs$  (т. е. символ с меньшим индексом), тогда в суффиксе  $S_2[j..m]$  окажется больше символов, которые могут входить в  $lcs$ .

Формализованная запись этой логики имеет вид:

**Procedure** Build;

**Var**  $i, j, q, t$ : Integer;

**Begin**

$t := 1$ ;

**For**  $i := 1$  **To**  $n$  **Do Begin**

**For**  $j := m$  **DownTo**  $1$  **Do**

{Сканируем вторую строку в обратном направлении.

В массиве  $V$  ищутся индексы  $j$ , удовлетворяющие

условию:  $V[i-1, q-1] < j_m < j_{m-1} < \dots < j_1 < V[i-1, q]$ }

**If**  $S_1[i] = S_2[j]$  **Then Begin**

$q := \text{Find}(V[i-1], j)$ ;

{Поиск "самого левого" индекса,  
удовлетворяющего условию}

**If**  $q > 0$  **Then Begin**  $V[i, q] := j$ ;  $t := q$ ; **End**;

{Запоминаем значение  $q$ , так как в этой строке потребуются корректировка значений (путем переписывания из предыдущей строки)  $V$  с меньшими индексами}

**End**;



```

For j:=1 To t-1 Do
  If V[i,j]>=m Then V[i,j]:=V[i-1,j];
  {Значением m+1 были проинициализированы все
  элементы V, кроме элементов из нулевого столбца}
End;
End;

```

Назначение функции Find строго определено: найти число в интервале от  $V[i-1, q-1]$  до  $V[i-1, q]$ , определяющее длину  $lcs$ , при заданном номере символа  $S_2$  (этим символом  $lcs$  может быть увеличена на единицу, поскольку Find вызывается при условии равенства  $S_1[i]$  и  $S_2[j]$ ).

```

Function Find(A:TArray;w:Integer):Integer;
{В разделе типов (Type) определяем:
TArray=Array[0..NMax] Of Integer;
FArray=Array[0..NMax] Of TArray, тогда в нотации
Паскаля строку массива можно передать как параметр}
Var i:Integer;
Begin
  i:=1;
  While (i<m+1) And (A[i]<w) Do i:=i+1;
  If A[i]=w Then Find:=0
  {Требуется найти меньшее, чем w, значение индекса, -
  только в этом случае lcs можно улучшить}
  Else Find:=i;
End;

```

По окончании работы процедуры Build значение длины наибольшей общей подпоследовательности строк  $S_1$  и  $S_2$  определяется максимальным определенным (не равным  $m+1$ ) значением  $V[n, q]$ .

Временная сложность алгоритма —  $O(n \cdot m \cdot q)$ , где  $q$  — длина наибольшей общей подпоследовательности. Действительно, пусть строки  $S_1$  и  $S_2$  равны и состоят из одного символа (например,  $a^n$ ). Тогда поиск позиции  $j$  будет выполняться  $n \cdot m$  раз, а время поиска будет пропорционально длине уже имеющейся  $lcs$ .

Можно ли улучшить этот алгоритм? Самым очевидным улучшением является оптимизация функции Find — поиска позиции в массиве общих последовательностей. Значения элементов массива  $V$  (в строке) являются возрастающей

последовательностью чисел, поэтому линейный поиск индекса может быть заменен на бинарный поиск. Временная сложность алгоритма в этом случае оценивается как  $O(n \cdot m \cdot \log_2 q)$ . Новая версия Find выглядит так:

```

Function Find(A:TArray;w:Integer):Integer;
  Var up, down, mid:Integer;
  Begin
    up:=m+1; down:=0;
    While down+1<up Do Begin
      mid:=(down+up) Shr 1;
      If A[mid]<=w Then down:=mid Else up:=mid;
    End;
    If A[down]<>w Then Find:=up Else Find:=0;
  End;

```

Второе улучшение тоже явно просматривается из ранее приведенной логики — это поиск совпадающих символов:

```

For i:=1 To n Do Begin
  For j:=m DownTo 1 Do
    If S1[i]=S2[j] Then ...

```

Эта конструкция дает нам квадратичную временную оценку —  $O(n \cdot m)$ . Вероятно, единственный путь решения проблемы заключается в вынесении части работы на стадию предварительной обработки. Возьмем более длинную строку и представим номера позиции с одинаковыми символами в виде связного списка (здесь эта процедура не приводится). Тогда, зная символ, мы однозначно выбираем список, содержащий номера позиций, в которых он находится. Возможное описание данных для реализации этой идеи — следующее:

```

Type
  PListEl=^TListEl;
  TListEl=Record
    x:Integer;
    next:PListEl;
  End;
  TListAr=Array[Byte] Of PListEl;
Var AListEl:TListEl;

```

Измененная версия процедуры Build (преобразуем ее в функцию) выглядит так:

```
Function Build:Integer;
```

```
  Var i,j,q,t:Integer;
      p:PListEl;
```



```
Begin
```

```
  t:=1;
```

```
  For i:=1 To n Do Begin
```

```
    p:=AListEl[Byte(S1[i])];
```

{Функция *Byte* преобразует символ в его двоичное представление — число. Списки номеров позиций символов в  $S_2$  организованы в убывающем порядке}

```
    While p<>Nil Do Begin
```

```
      q:=Find(V[i-1],p^.x);
```

```
      If q>0 Then Begin V[i,q]:=p^.x; t:=q; End;
```

```
      p:=p^.next;
```

```
    End;
```

```
    For j:=1 To t-1 Do
```

```
      If V[i,j]>=m Then V[i,j]:=V[i-1,j];
```

```
    End;
```

```
  Build:=t-1;
```

```
End;
```

Время работы такого алгоритма уменьшается с оценки  $O(n \cdot m)$  до  $O(n + r)$ , где  $r$  — количество пар совпадающих символов в  $S_1$  и  $S_2$ . Однако есть еще конструкция

```
For j:=1 To t-1 Do
```

```
  If V[i,j]>=m Then V[i,j]:=V[i-1,j];.
```

Таким образом, общая временная оценка остается в виде  $O((n + r) \cdot \log_2 q + n \cdot q)$ . Для худшего случая (совпадающих строк) мы имеем  $O((n + n \cdot n) \cdot \log_2 n + n \cdot n)$ , а для строк из разных символов —  $O(n)$ .

Способ уменьшения требований к используемой памяти (уход от массива  $V$ ) также очевиден. Для вычисления значений элементов строки  $V[i]$  используется только предыдущая строка  $V[i - 1]$ , причем обработка ведется по столбцам, что дает возможность выполнить замену двумерного массива на одномерный. Но это позволяет заодно исключить «добавку» к логике в виде заполнения неопределенных значений  $V[i, q]$ , что сводит временную оценку к  $O((n + r) \cdot \log_2 q)$ . Однако при этом мы лишаемся данных для вывода самой наибольшей общей последовательности и определяем только ее длину. Чтобы не исключать этой возможности, следу-

ет изменить конструкцию (при истинности условия) **If**  $q > 0$  **Then Begin**  $V[i, q] := p^x$ ;  $t := q$ ; **End**; — необходимо запоминать все пары одинаковых символов в том порядке, в котором они проходились в условии, а также номер позиции этих символов в *lcs*. Для этой цели следует использовать еще один связный список типа:

**Type**

```
PSeqEl = ^TSeqEl;
```

```
TSeqEl = Record
```

```
  xv: Integer;
```

```
  yv: Integer;
```

```
  sv: Integer;
```

```
  next: PSeqEl;
```

```
End;
```

```
Var pt: PSeqEl;
```

{Указатель на первый элемент списка – глобальная переменная}



В качестве справочного материала приведем процедуру вставки в начало списка<sup>1)</sup>:

```
Procedure AddSeq (Var sq: PSeqEl; Const a, b, c: Integer);
```

```
  Var temp: PSeqEl;
```

```
  Begin
```

```
    New(temp);
```

```
    temp^.xv := a;
```

```
    temp^.yv := b;
```

```
    temp^.sv := c;
```

```
    temp^.next := sq;
```

```
    sq := temp;
```

```
  End;
```



После этих размышлений новая версия функции *Build* будет выглядеть так:

```
Function Build: Integer;
```

```
  Var i, j, q, t: Integer;
```

```
  p: PListEl;
```

```
  Begin
```

```
    t := 1; {Количество элементов в массиве V}
```

<sup>1)</sup> Окулов С. М. Развитие интеллекта школьника. Абстрактные типы данных. — М.: БИНОМ. Лаборатория знаний, 2009.



```

For i:=1 To n Do Begin
  p:=AList[Byte(S1[i])];
  {Выбираем список номеров позиций вхождения
  символа  $S_1[i]$  в строку  $S_2$ }
  While p<>Nil Do Begin
    q:=Find(V,p^.x,t);
    {Массив V имеет тип:
     TArray=Array[0..NMax] Of Integer;}
    If q>0 Then Begin
      t:=Max(t,q+1);
      AddSeq(pt,i,p^.x,q);
      V[q]:=p^.x;
    End;
    p:=p^.next;
  End;
End;
Build:=t-1;
End;

```

Осталось привести процедуру вывода наибольшей общей подпоследовательности (Path) и на этом закончить работу по оптимизации алгоритма Д. Ханта и Т. Зиманского:



```

Procedure Path;
Var temp:PSeqEl;
      i,j,ln:Integer;
Begin
  temp:=pt;
  {pt - указатель на первый элемент
  списка символов, образующих наибольшую
  общую подпоследовательность}
  ln:=Build;
  ln:=ln+1;
  lcs:='';
  {Глобальная переменная строкового типа -
  наибольшая общая подпоследовательность
  строк  $S_1$  и  $S_2$ }
  i:=MaxInt; j:=MaxInt;
  {MaxInt - максимальное целое число}
  While (temp<>Nil) And (ln>0) Do Begin
    If (temp^.xv<i) And (temp^.yv<j) Then Begin

```

```
i:=temp^.xv;  
j:=temp^.yv;  
lcs:=S1[i]+lcs;  
ln:=ln-1;  
End;  
temp:=temp^.next;
```

**End;**

**End;**

Временная оценка работы оптимизированного варианта алгоритма имеет вид  $O((n + r) \cdot \log_2 q)$ , где  $n$  — длина меньшей строки,  $r$  — количество совпадающих символов, а  $q$  — количество символов в общей подпоследовательности наибольшей длины.



### Упражнения

1. Приведите примеры двух строк. Путем формирования массива  $V$  подсчитайте для них длину наибольшей общей подпоследовательности.
2. Путем обратного просмотра элементов  $V$  выпишите наибольшую общую подпоследовательность строк.
3. Разработайте программную реализацию алгоритма Д. Ханга и Т. Зиманского (неоптимизированный вариант).
4. Дана строка  $S$ . Разработайте программу формирования списков из номеров позиций с одинаковыми символами строки.
5. Разработайте программную реализацию алгоритма Д. Ханга и Т. Зиманского (оптимизированный вариант). Оцените время работы в наихудшем и наилучшем случаях.

#### 4.3.3. Алгоритм Л. Эллисона – Т. Дикса

Неизвестно, что человек еще выдумает: голова — круглая.

*Хенрик Ягодзиньский*

Весь смысл этого алгоритма заключается в построении двоичной матрицы  $R$ , отражающей связи между символами строк  $S_1$  и  $S_2$ , длина наибольшей общей последовательности ( $lcs$ ) которых ищется. Построение начинается с последней строки  $R[m]$ . Если в алгоритме Р. Бойера – Дж. Мура обра-

зец сравнивался с фрагментом строки справа налево (от последнего символа до первого), то и здесь, по аналогии, ищется длина  $lcs$  последнего символа  $S_2[m]$  и  $S_1$ , затем — суффикса  $S_2[m - 1..m]$  и  $S_1$  и т. д. По окончании этого процесса в первой строке  $R$  представлена информация о длине  $lcs$ .

Почему мы начинаем с конца строки  $S_2$ ? Ответ заключается в том, что на каждом шаге ищется самое правое вхождение  $lcs$  суффикса  $S_2[i..m]$  и строки  $S_1$ . (Пока мы не говорим о том, как именно представлена информация о длине  $lcs$  в строке.)

Вторая идея заключается в использовании *характеристических векторов*, описывающих вхождение символов в одну из строк (в рассматриваемом случае  $S_1$ , т. е.  $n \ll m$  и  $n < 32$ ), и в применении идеологии алгоритма *Shift-And* (см. п. 2.4) для осуществления перехода от обработки одного суффикса  $S_2$  к другому.

«Привяжем» изложение материала этого параграфа к сквозному примеру.

### Пример

Алфавит  $A$  состоит из символов  $\{a, b, c, d\}$ . Даны строки  $S_1 = bcadabbdc$ ,  $S_2 = abcaadbdd$ . Требуется найти длину наибольшей общей подпоследовательности этих строк. В табл. 4.10 представлены характеристические векторы длиной  $n$  бит, описывающие вхождение символов в  $S_1$  (матрица  $V$ ).

Таблица 4.10

V	b	c	a	d	a	b	b	d	c
a	0	0	1	0	1	0	0	0	0
b	1	0	0	0	0	1	1	0	0
c	0	1	0	0	0	0	0	0	1
d	0	0	0	1	0	0	0	1	0

Решим простую задачу. Требуется оставить в характеристическом векторе одну единицу, описывающую только самое правое вхождение символа в строку. Пусть это будет символ  $a$ . Требуемая последовательность действий приведена в табл. 4.11 и имеет вид:  $((V[a] - 1) \text{Xor } V[a]) \text{And } V[a]$ .

Таблица 4.11

$V[a]$	0	0	1	0	1	0	0	0	0
Единица	0	0	0	0	0	0	0	0	1
Вычитание: $V[a] - 1$	0	0	1	0	0	1	1	1	1
$(V[a] - 1) \text{Xor } V[a]$	0	0	0	0	1	1	1	1	1
$((V[a] - 1) \text{Xor } V[a]) \text{And } V[a]$	0	0	0	0	1	0	0	0	0

А теперь чуть-чуть усложним задачу, временно отвлекшись от нашего сквозного примера. Пусть  $S_1 = \text{aaaaa}$  и  $S_2 = \text{aaaa}$ ,  $n = 5$ ,  $m = 4$  (значение  $m$  может быть и увеличено — смысл от этого не изменится). Единственный характеристический вектор  $V[a]$  для символа  $a$  состоит из одних единиц. Строка матрицы  $R[5]$  — нулевая. Строки  $R$  формируются в очередности 4, 3, 2, 1. Последовательность действий приведена в табл. 4.12.

Таблица 4.12

$R$	Действие	Разряды				
$R[4]$	$x = V[a] \text{Or } R[5]$	1	1	1	1	1
	$t = R[5]_{\leftarrow \text{на } 1 \text{ разряд}} + 1$	0	0	0	0	1
	$x - t$	1	1	1	1	0
	$x \text{Xor } (x - t)$	0	0	0	0	1
	$x \text{And } (x \text{Xor } (x - t))$	0	0	0	0	1
$R[3]$	$x = V[a] \text{Or } R[4]$	1	1	1	1	1
	$t = R[4]_{\leftarrow \text{на } 1 \text{ разряд}} + 1$	0	0	0	1	1
	$x - t$	1	1	1	0	0
	$x \text{Xor } (x - t)$	0	0	0	1	1
	$x \text{And } (x \text{Xor } (x - t))$	0	0	0	1	1
$R[2]$	$x = V[a] \text{Or } R[3]$	1	1	1	1	1
	$t = R[3]_{\leftarrow \text{на } 1 \text{ разряд}} + 1$	0	0	1	1	1
	$x - t$	1	1	0	0	0
	$x \text{Xor } (x - t)$	0	0	1	1	1
	$x \text{And } (x \text{Xor } (x - t))$	0	0	1	1	1
$R[1]$	$x = V[a] \text{Or } R[2]$	1	1	1	1	1
	$t = R[2]_{\leftarrow \text{на } 1 \text{ разряд}} + 1$	0	1	1	1	1
	$x - t$	1	0	0	0	0
	$x \text{Xor } (x - t)$	0	1	1	1	1
	$x \text{And } (x \text{Xor } (x - t))$	0	1	1	1	1

Таким образом,  $R = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ . В строке  $R[4]$  одна

единица, и она говорит о том, что суффиксы  $a$  строк  $S_2$  и  $S_1$  имеют  $lcs$  длины 1 и указывается самое правое вхождение  $lcs$ . Аналогично — для строк  $R[3]$  и  $R[2]$ . Количество же единиц в  $R[1]$  дает длину  $lcs$  для строк  $S_1$  и  $S_2$ .

Рассмотрим еще один пример. Пусть  $R[i] = 000110010$ , а очередным,  $(i - 1)$ -м символом  $S_2$  является символ  $b$  (его характеристический вектор —  $100001100$ ). Разделим  $R[i]$  на блоки (разделитель обозначен символом « $\uparrow$ »)  $000\uparrow 1\uparrow 100\uparrow 10$  по следующему правилу: идем по строке слева направо; блок заканчивается *перед* первой единицей справа (условно считаем, что строка  $R$  всегда начинается с единицы — нулевой столбец), а следующий начинается с этой единицы; блок может закончиться нулем, если он — последний в строке. Объединением  $R[i]$  с  $V[b]$  с помощью операции  $\text{Or}$  фиксируется присутствие символов  $b$  в каждом блоке:  $x = R[i] \text{ Or } V[b] = 100111110$ . Сдвиг на один разряд влево строки  $R[i]$  и добавление единицы к младшему разряду ( $t = R[i]_{\leftarrow 1 \text{ разряд}} + 1 = 001100101$ ) сохраняет структуру разбиения строки на блоки и вводит один (возможный) блок в следующей  $R[i - 1]$ -й строке. Остальные операции:  $((x - t) \text{ Xor } x) \text{ And } x = ((100111110 - 001100101) \text{ Xor } 100111110) \text{ And } 100111110 = 100\uparrow 100\uparrow 1\uparrow 10$  — решают задачу фиксации старшей единицы, но уже в каждом блоке. Строка  $R[i - 1]$  содержит четыре единицы, т. е. у суффикса  $S_2[i - 1..m]$  и  $S_1$  существует  $lcs$  длины 4 (обратите внимание, что здесь не говорится о том, из каких символов состоит  $lcs$ , — говорится лишь о том, что она состоит из четырех символов!).

Обратимся вновь к нашему сквозному примеру. Окончательный вид матрицы  $R$  приведен в табл. 4.13.

Пусть формируется  $R[4]$ . Имеем:  $R[5] = 00\uparrow 10\uparrow 1\uparrow 1\uparrow 1\uparrow 10$  (разделили на блоки), а характеристический вектор символа  $a$  равен  $001010000$ . Логическая сумма равна  $x = 001011110$ . Значение  $R[5]$ , сдвинутое на один разряд влево с прибавлением единицы, равно  $010111101$ . Разность  $t = 110100001$  (обратите внимание, что пришлось заимство-

Таблица 4.13

<i>R</i>	<i>S</i> <sub>1</sub>		<i>b</i>	<i>c</i>	<i>a</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>d</i>	<i>c</i>
<i>S</i> <sub>2</sub>	<i>i/j</i>	0	1	2	3	4	5	6	7	8	9
<i>a</i>	1	1	1	1	1	0	1	1	1	0	1
<i>b</i>	2	1	1	1	1	0	1	1	1	0	1
<i>c</i>	3	1	0	1	1	0	1	1	1	0	1
<i>a</i>	4	1	0	0	1	0	1	1	1	1	0
<i>a</i>	5	1	0	0	1	0	1	1	1	1	0
<i>d</i>	6	1	0	0	0	1	0	1	1	1	0
<i>b</i>	7	1	0	0	0	0	0	1	1	1	0
<i>b</i>	8	1	0	0	0	0	0	0	1	1	0
<i>d</i>	9	1	0	0	0	0	0	0	0	1	0
		0	0	0	0	0	0	0	0	0	0

вать единицу из нулевого столбца). Дальнейшие действия (операции Xor и And) приводят к значению  $R[4]$ , равному  $00\uparrow 10\uparrow 1\uparrow 1\uparrow 1\uparrow 10$ . Структура *lcs* не изменилась.

На следующем шаге обрабатывается суффикс caadbdd. Имеем  $x = 011011111$ ,  $t = 011011111 - 010111101 = 000100010$ . Операции Xor и And приводят к  $R[3] = 0\uparrow 1\uparrow 10\uparrow 1\uparrow 1\uparrow 10\uparrow 1$ . Значение длины *lcs* для этого суффикса и  $S_1$  равно шести.

Очевидно, что при подсчете длины *lcs* на каждом шаге достаточно знать только предыдущую строку *R*. Формализованная запись этого линейного по времени варианта логики имеет следующий вид:

```

Procedure Search(S1,S2:String);
  Var V:Array[Chr(0)..Chr(255)] Of LongInt;
      n,m,i:Byte;
      r,t,x:LongInt;
      j:Char;
Begin
  n:=Length(S1);
  m:=Length(S2);
  For j:=Chr(0) To Chr(255) Do V[j]:=0;
  For i:=1 To n Do V[S1[i]]:=V[S1[i]] Or
      (1 ShL (n-i));
  t:=0; {Значение r на предыдущем шаге обработки}

```

```

For i:=m DownTo 1 Do Begin
  x:=V[S2[i]] Or t; {Рабочая переменная}
  r:=x And ((x-((t ShL 1)+1)) Xor x);
  t:=r; {Запоминаем r}
End;
i:=0;
While (r>0) Do
  {Подсчитываем количество единиц в r}
  If (r Mod 2=1) Then Begin
    i:=i+1;
    r:=r ShR 1;
  End;
WriteLn('длина lcs равна ', i);
End;

```

Здесь найдена длина  $lcs$  строк, но не сама подпоследовательность! Однако если матрица  $R$  полностью определена, то для вывода самой  $lcs$  у нас есть исходная информация. В табл. 4.13 единицы, соответствующие  $lcs$ , выделены жирным курсивом. Для вывода подпоследовательности начиная с первой строки  $R$  следует найти последнюю строку с максимальным значением количества единиц, а затем совершить «угловой обход». Суть этого обхода в том, что в каждой строке мы находим первую единицу и обходим ее, т. е. «спускаемся» в следующую строку, ищем с этой позиции первую единицу и т. д., пока не будет достигнут последний столбец. Символы из  $S_1$ , соответствующие выделенным единицам, как раз и составляют  $lcs$ .

### Упражнения

1. Для строк  $S_1 = \text{tgcsaaag}$  и  $S_2 = \text{gtacstc}$  убедитесь, что матрица  $R$  имеет вид:

$$R = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix},$$

т. е. длина  $lcs$  равна 4. Используя «угловой обход», найдите самую эту подпоследовательность.

2. Пусть  $S_1 = \text{aaaaaaa}$  и  $S_2 = \text{aaaaab}$ . Сформируйте для них матрицу  $R$ .
3. Приведите пример двух строк на заданном алфавите. Вычислите длину их наибольшей общей подпоследовательности.
4. При известной матрице  $R$  разработайте процедуру поиска наибольшей общей подпоследовательности.
5. Используя материал предыдущего раздела, разработайте логику вывода  $lcs$  для случая, когда для подсчета длины  $lcs$  используется только предыдущая строка матрицы  $R$  (а она сама не формируется).
6. Ниже представлен вариант логики, реализующей поиск длины наибольшей общей подпоследовательности строк ( $n = |S_1|$ ,  $m = |S_2|$ ) при  $n > 32$ . Проверьте его работоспособность и найдите возможность улучшить приведенный ниже программный код.

```

Procedure SearchLong (S1, S2: String) ;
  Const nmax=10;
  {Значение 10 условно, оно определяется
  конкретной задачей. Остальные константы
  оставлены в листинге для наглядности}
  Type mas=Array[1..nmax] Of Longint;
  Var V:Array [Chr(0) .. Chr(255), 1..nmax]
  Of Longint;

  n, m, i, j, q: Byte;
  R, T: mas;
  y: Char;
Begin
  n:=Length(S1);
  q:=n Div 31;
  If (n Mod 31>0) Then q:=q+1;
  For y:=Chr(0) To Chr(255) Do
    For j:=1 To q Do V[y, j]:=0;
  For i:=1 To n Do
    If (i Mod 31=0)
      Then V[S1[n-i+1], i Div 31]:=
        W[S1[n-i+1], i Div 31] Or
        (1 ShL(i-31*(i Div 31-1)-1))

```

```

Else V[S1[n-i+1],i Div 31+1]:=
      V[S1[n-i+1],i Div 31+1] Or
      (1 ShL (i-31*(i Div 31)-1));
m:=Length(S2);
For i:=m DownTo 1 Do Begin
  For j:=1 To q Do Begin
    R[j]:=T[j] Or V[S2[i],j];
    If (j=1) Or ((T[j-1] Div 1073741824=1)
      And (j>1)) Then T[j]:= (T[j] ShL 1)+1
    Else T[j]:= (T[j] ShL 1);
    T[j]:=R[j] And ((R[j]-T[j]) Xor R[j]);
  End;
End;
i:=0;
For j:=1 To q Do
  While (T[j]>0) Do Begin
    If (T[j] Mod 2=1) Then i:=i+1;
    T[j]:=T[j] ShR 1;
  End;
WriteLn('длина lcs равна',i);
End;

```

### Методический комментарий

Алгоритмы вычисления расстояния между строками и нахождения подпоследовательностей являются, если так можно выразиться, «преамбулой» к целому классу алгоритмов приближенного сопоставления строк. Последние имеют огромное значение как в молекулярной биологии, так и в других областях знания, включая *computer science*.

Алгоритм нахождения расстояния между строками достаточно известен, хотя его описание можно встретить не так часто (в силу его простоты и очевидности). Б. Смит называет его «алгоритмом Р. Вагнера (*Robert Wagner*) и М. Фишера (*Michael Fischer*)». Простой алгоритм нахождения наибольшей общей подпоследовательности приписывают различным авторам, но на русском языке он стал известен как «алгоритм С. Нудельмана (*Saul Needleman*) – К. Вунша (*Christian Wunsch*)»<sup>1)</sup>. Его изменения, направленные на уменьшение объема используемой памяти, определяют как алгоритм

<sup>1)</sup> *Needleman Saul B., Wunsch Christian D. A general method applicable to the search for similarities in the amino-acid principle of two proteins // J. Molecular Biology 48, 1970. P. 443–453.*

Д. Хешберга (*Dan Hirschberg*)<sup>1)</sup>. С алгоритмами Д. Ханта (*James Hunt*) – Т. Зиманского (*Thomas Szymanski*)<sup>2)</sup> и Э. Укконена (*Esko Ukkonen*)<sup>3)</sup> – Ю. Майерса (*Gene Myers*)<sup>4)</sup> для обоснования некоторых деталей можно поработать и по книге Б. Смита. Алгоритм Л. Эллисона (*Lloyd Allison*) и Т. Дикса (*Trevor Dix*)<sup>5)</sup> интересен синтезом идей методов Р. Бойра – Дж. Мура и Р. Беза-Йетс и Г. Гоннет (алгоритм *Shift-And*).

Представленный в данной главе материал, конечно же, не полностью охватывает рассматриваемую проблематику. Наиболее перспективным направлением ее развития, видимо, является использование идей алгоритма *Shift-And* для решения поставленной задачи. Примером тому является не только рассмотренный алгоритм Л. Эллисона – Т. Дикса, но и модификация алгоритмов Д. Хешберга и Д. Ханта – Т. Зиманского<sup>6)</sup>, оставленная за пределами рассмотрения.

- 
- 1) *Hirschberg Dan S.* Algorithms for the longest common subsequence problem // JACM 24–4, 1977. P. 664–675.
  - 2) *Hunt James W., Szymanski Thomas G.* A fast algorithm for computing longest common subsequences // SACM 20–5, 1977. P. 350–353.
  - 3) *Ukkonen Esko.* Algorithms for approximate string matching // Information & Control, 64, 1985. P. 100–118.
  - 4) *Myers Gene W.* An O(ND) difference algorithm and its variations // Algorithmica I, 1986. P. 251–266.
  - 5) *Allison Lloyd, Dix Trevor.* A bit string longest subsequence algorithm // IPL, 23–6, 1986. P. 305–310.
  - 6) *Crochemore Maxime, Iliopoulos Costas, Pinzon Yoan.* Speeding up Hirschberg and Hunt–Szymanski LCS algorithms // Proc. Eighth IEEE Internal. Symp. String Processing & Information Retrieval. IEEE Computer Science Press, 2001. P. 59–67.

# Алгоритмы приближенного поиска подстрок

---

- Какой у него телефон?
- Не помню.
- Ну, хотя бы приблизительно?



*Сергей Довлатов*

Количество алгоритмов приближенного поиска подстрок, предложенных с 1980-х гг., сопоставимо с количеством алгоритмов точного поиска подстроки. Их изучение преследует цель не только анализа хода мыслей ученых-информатиков, но и формирования базы для решения многочисленных прикладных задач.

## 5.1. Простой алгоритм

Молодость мне много обещала,  
Было мне когда-то двадцать лет,  
Это было самое начало,  
Я был глуп, и это не секрет.



*Борис Рыжий*

В п. 4.1 были введены различные меры близости между строками  $S_1$  и  $S_2$ : расстояние Р. Хемминга и расстояние В. Левенштейна. В некоторых случаях операция подстановки имеет вес 1; в других она заменяется двумя операциями — вставкой и удалением — и имеет вес 2.

Пусть дан текст  $T$  ( $n = |T|$ ) и образец  $P$  ( $m = |P|$ ). Определим расстояние  $d$  между префиксами  $T[1..i]$  и  $P[1..j]$  как  $d(i, j) = \min_{0 \leq i' \leq i} (d[T[i', i], P[1..j]])$  для всех  $i \in 0..n$ ,  $j \in 1..m$ .

Будем хранить значения  $d$  в массиве  $D$ . Главное, на что следует обратить внимание в этой формуле, заключается в необязательности равенства  $i - i' + 1 = j$ . Ищется минимальное значение расстояния между суффиксами префикса  $T[1..i]$  и префиксом  $P[1..j]$ .

Определим зависимости:

- $D[0, 0] = 0$  — пустая строка преобразуется в пустую строку за нуль операций;
- $D[0, j] = j$  для  $j$  от 1 до  $m$  — пустая строка преобразуется в префикс  $P[1..j]$  за  $j$  операций вставки, и это минимальное количество операций;
- $D[i, 1] = \min(D[i - 1, 1] + 1, D[T[i], P[j]])$  — ищется минимальное расстояние между  $P[1]$  и подстрокой  $T[i', i]$ . Если  $T[i] = P[j]$ , то  $D[T[i], P[j]] = 0$ ; при  $T[i] \neq P[j]$  мы имеем  $D[T[i], P[j]] = 2$ .

Рекуррентное соотношение для  $D[i, j]$  при  $1 \leq i \leq n$  и  $1 < j \leq m$  имеет вид:

$$D[i, j] = \min\{D[i - 1, j] + 1, D[i, j - 1] + 1, D[i - 1, j - 1] + t\},$$

где  $t = 2$ , если  $T[i] \neq P[j]$ , и  $t = 0$ , если  $T[i] = P[j]$ .

После вычисления массива расстояний  $D$  для любого неотрицательного целого числа  $k$  и префикса  $P[1..j]$  у нас есть возможность ответить на вопрос о том, входит ли  $P[1..j]$  в  $T$  с точностью до  $k$  операций редактирования (*k-приближенное вхождение  $P$  в  $T$* ). Для этого достаточно посмотреть столбец  $j$  и найти значения  $D[i, j] \leq k$ . Найденное значение  $D[i, j]$  говорит о том, что существует подстрока  $T[i'..i]$ , отличающаяся от  $P[1..j]$  не более чем на значение  $k$  (*k-приближение*). Или, другими словами, эта подстрока и префикс  $P[1..j]$  могут быть преобразованы друг в друга операциями редактирования с суммарным весом не более чем  $k$ .

### Пример

Пусть  $T = \text{abcaadbdbd}$  и  $P = \text{bcadab}$ . Получаемый для них массив  $D$  представлен в табл. 5.1.

Возьмем префикс  $P[1..4] = \text{bcad}$ . Проанализируем данные из столбца № 4:

- $T[1..1] = \text{a}$  — вставка трех символов;
- $T[1..2] = \text{ab}$  — подстрока  $\text{b}$ , вставка трех символов;
- $T[1..3] = \text{abc}$  — подстрока  $\text{bc}$ , вставка двух символов;
- $T[1..4] = \text{abca}$  — подстрока  $\text{bca}$ , вставка одного символа;
- $T[1..5] = \text{abcaa}$  — подстрока  $\text{bcaa}$ , подстановка одного символа;
- $T[1..6] = \text{abcaad}$  — подстрока  $\text{bcaad}$ , удаление одного символа  $\text{a}$ ;

Таблица 5.1

$D$	Номер символа	1	2	3	4	5	6
Номер символа	Символ	b	c	a	d	a	b
0	Символ	1	2	3	4	5	6
1	a	2	3	2	3	4	5
2	b	0	1	2	3	4	4
3	c	1	0	1	2	3	4
4	a	2	1	0	1	2	3
5	a	2	2	1	2	1	2
6	d	2	3	2	1	2	3
7	b	0	1	2	2	3	2
8	b	0	1	2	3	4	3
9	d	1	2	3	2	3	4

- $T[1..7] = abcaadb$  — подстрока  $bcaadb$ , удаление двух символов;
- $T[1..8] = abcaadbb$  — подстрока  $bcaadbb$ , удаление трех символов;
- $T[1..9] = abcaadbbd$  — подстрока  $bd$ , вставка двух символов.

Данный алгоритм практически совпадает с тем, что был описан в п. 4.1. Единственное отличие заключается в возможности его использования для расстояний, определенных различным образом. В данном случае операции вставки и удаления имели вес 1, а операция подстановки — вес 2, но эти значения могут быть и другими.

Для определения подстроки  $T[i'..i]$ , совпадающей с префиксом  $P[1..j]$ , следует (так же, как это описано в п. 4.1) ввести дополнительные структуры данных, в которых предполагается фиксировать координаты предшествующей позиции, обеспечивающей минимальное значение расстояния.

Алгоритм затратный: с точки зрения как времени —  $O(n \cdot m)$ , так и объема памяти, имеющего аналогичную оценку. Однако требование к памяти можно уменьшить, поскольку значения в строке  $i$  зависят только от значений в строке  $i - 1$ . В этом случае требования к объему памяти сокращаются до  $O(m)$ .

 **Упражнения**

1. Приведите примеры строк  $P$  и  $T$ . Вычислите для них матрицу расстояний  $D$ . Для каждого префикса  $P$  определите подстроку  $T$ , на которой достигается полученная оценка.
2. Разработайте программу формирования массива  $D$  и поиска такого значения  $i'$ , что  $d(T[i'..n], P) = D[n, m]$ .
3. Для заданного значения  $k$  найдите все такие значения  $i'$  и  $i$ , что  $d(T[i'..i], P) \leq k$ .



## 5.2. Алгоритм С. Ву – Ю. Менбера

Бесценен тот, кто подал ценную идею.

*Георгий Александров*

Алгоритм С. Ву и Ю. Менбера работоспособен при различных весах операций вставки, удаления и подстановки (замены) и опирается на базовые рекуррентные соотношения, приведенные в п. 5.1. Однако матрица  $D$  здесь не вычисляется, а вместо этого используются идеи алгоритма *Shift-And*, сводящие задачу к манипуляциям с битовыми векторами. Для определенности и простоты изложения примем значения весов, равные 1 для операций вставки и удаления и 2 для операции замены. Речь в этом случае идет не о  $k$ -несовпадениях, а о  $k$ -приближении ( $k \geq 1$ ). Другими словами, требуется найти вхождение  $P$  в  $T$  с точностью до  $k$  — когда суммарный вес операций по преобразованию  $P$  в какую-то подстроку  $T[i'..i]$  не превышает значения  $k$ . Вычислив матрицу  $D$  (п. 5.1) и просмотрев последний столбец, ответ задачи можно найти, но суть алгоритма именно в том, чтобы не вычислять ее в прямом виде.

«По традиции» рассмотрим пример.

### *Пример*

Пусть  $P = \text{bacaba}$ ,  $T = \text{abcdabd}$  и  $k = 3$ . Приведем матрицу  $D$  (табл. 5.2), но только для понимания идейной основы алгоритма.

Таблица 5.2

<i>D</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	2	1	2	3	4	5
<i>b</i>	0	1	2	3	3	4
<i>c</i>	1	2	1	2	3	4
<i>d</i>	2	3	2	3	4	5
<i>b</i>	0	1	2	3	3	4
<i>a</i>	1	0	1	2	3	3
<i>d</i>	2	1	2	3	4	4

Предположим, что каким-то способом вычислены матрицы  $R_q$ , где  $q = 0, 1, 2$  и  $3$  (табл. 5.3). Элемент  $R_q[i, j]$  равен нулю тогда и только тогда, когда префикс  $P[1..j]$  с точностью  $q$  входит в какую-то подстроку ( $T[i'..i]$ ) префикса  $T[1..i]$ , иначе  $R_q[i, j] = 1$ . При наличии матрицы  $D$  элементы  $R_q$  для всех значений  $q$  выписываются просто. Для  $R_0$ : если  $D[i, j] = 0$ , то  $R_0[i, j] = 0$ , иначе  $R_0[i, j] = 1$ . Для  $R_1$ : если  $D[i, j] \leq 1$ , то  $R_0[i, j] = 0$ , иначе  $R_0[i, j] = 1$ . Аналогично — и для оставшихся значений  $q$ .

Таблица 5.3

	$R_0$						$R_1$						$R_2$						$R_3$					
	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1
<i>b</i>	0	1	1	1	1	1	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1
<i>c</i>	1	1	1	1	1	1	0	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	1
<i>d</i>	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	0	0	0	1	1
<i>b</i>	0	1	1	1	1	1	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1
<i>a</i>	1	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0
<i>d</i>	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1

О чем говорит, например, значение  $R_3[6, 6] = 0$ ? Да только о том, что строка  $P = bacaba$  может быть преобразована в подстроку  $T[2..6] = bcdba$  за операции с суммарной стоимостью 3. Вот эти операции: вставка символа *a* после первого символа *b* и замена символа *d* на символ *a*. При весах операций, о которых мы говорили в начале этого разде-

ла, это как раз составляет значение 3. Правда, мы здесь чуть-чуть слукавили: значение  $R_3[6, 6] = 0$  говорит только о самом этом факте, но для того, чтобы определить подстроку  $T$  и сами требуемые операции, необходимы дополнительные усилия.

*Примечание.* Обратите внимание, что по отношению к предыдущему варианту матриц  $R$  (обобщение алгоритма *Shift-And*) значения 1 и 0 инвертированы. Забегая вперед, скажем, что в данном алгоритме и значения элементов матрицы  $V$  (характеристические векторы), отражающих вхождение символов алфавита  $A$  в образец  $P$ , тоже инвертированы. Так, для рассматриваемого примера  $V$  имеет вид, приведенный в табл. 5.4.



Таблица 5.4

V	b	a	c	a	b	a
a	1	0	1	0	1	0
b	0	1	1	1	0	1
c	1	1	0	1	1	1
d	1	1	1	1	1	1

И вот, наконец, пришло время задать главный вопрос: можно ли значения  $R$  вычислять без знания  $D$ ?

Бинарными векторами в данном случае являются строки  $R_q[i]: R_0[i], R_1[i], \dots, R_k[i]$  ( $0 \leq q \leq k$ ). Задача заключается в том, чтобы на основе известных значений  $R_q[i-1]$  вычислить  $R_q[i]$ . Другими словами, вычисляются первые строки всех матриц  $R$ , затем вторые, третьи и т. д. Появление нуля в столбце  $m$  ( $m$  — это компонент какого-то из  $q+1$  векторов) говорит о том, что образец  $P$  входит в префикс  $T$  (если на последнем шаге, то в текст  $T$ , а иначе — в  $T[1..i]$ ) с точностью  $k$ . Естественным является и ввод нулевой строки и нулевого столбца: значение  $R_q[0, j] = (0, 1, 1, \dots, 1)$  для  $j = 0, 1, 2, \dots, m$ , а  $R_q[i, 1] = 0$  для  $i = 0, 1, 2, \dots, n$ .

Что необходимо знать для вычисления компоненты  $j$  вектора  $R_q[i]$ , т. е.  $R_q[i, j]$  (определяется вхождение префикса  $P[1..j]$  в префикс  $T[1..i]$  с точностью  $q$ )? Во-первых, входит ли префикс  $P[1..j]$  в префикс  $T[1..i-1]$  с точностью  $q-1$ . Во-вторых, входит ли префикс  $P[1..j-1]$  в префикс  $T[1..i]$  с точностью  $q-1$ . И в-третьих, входит ли префикс

$P[1..j - 1]$  в префикс  $T[1..i - 1]$  с точностью  $q - 2$  (а если бы операция подстановки имела вес, например, 3, то было бы  $q - 3$ ). Выполнение хотя бы одного из этих трех условий (равенство нулю элемента соответствующей матрицы  $R$ ) говорит о том, что совпадение символов  $P[j]$  и  $T[i]$  не принципиально, ибо гарантировано, что  $P[1..j]$  входит в  $T[1..i]$  с точностью  $q$ . Отсюда следует необходимость вычисления выражения  $R_{q-1}[i - 1, j]$  And  $R_{q-1}[i, j - 1]$  And  $R_{q-2}[i - 1, j - 1]$ . Но это — еще не окончательный результат. Если префикс  $P[1..j - 1]$  входит в  $T[1..i - 1]$  с точностью  $q$  ( $R_q[i - 1, j - 1] = 0$ ) и символы совпадают:  $P[j] = T[i]$ , то мы имеем вхождение  $P[1..j]$  в  $T[1..i]$  с точностью  $q$ . Таким образом, мы получаем необходимость вычисления выражения:

$$R_q[i, j] := R_{q-1}[i - 1, j] \text{ And } R_{q-1}[i, j - 1] \text{ And } R_{q-2}[i - 1, j - 1] \text{ And } (R_q[i - 1, j - 1] \text{ Or } V[T[i], j]).$$

Вернемся к нашему примеру. В табл. 5.3 элемент  $R_3[6,6]$  выделен жирным шрифтом, а значения  $R_2[5,6]$ ,  $R_2[6,5]$ ,  $R_1[5,5]$  и  $R_3[5,5]$  — курсивом. Выражение  $R_2[5,6]$  And  $R_2[6,5]$  And  $R_1[5,6]$  равно единице, т. е. образец  $P$  не может быть преобразован в подстроку префикса  $T[1..6]$  операциями с суммарным весом 3. Однако префикс  $P[1..5] = \text{bacab}$  может быть преобразован в подстроку суффикса  $T[1..5] = \text{abcdb}$  (вставкой символа  $a$  после первого символа  $b$  и заменой символа  $d$  на символ  $a$  в подстроке  $\text{bcd}b$ ) и символ  $T[6] = P[6]$ . Следовательно, образец  $P$  преобразуется в подстроку префикса  $T[1..6]$  за операции с суммарным весом 3.

Эффект по времени вычислений, как и в алгоритме *Shift-And*, достигается за счет исключения цикла по компонентам бинарных векторов: он заменяется логическими операциями в целом с бинарными векторами (строками матриц  $R$  и  $T$ ), а выравнивание позиций (по индексу  $j$ ) осуществляется с помощью операции сдвига на один разряд вправо. При этом нулевой разряд всех векторов равен 0. Имеем:

$$R_q[i] := R_{q-1}[i - 1] \text{ And } R_{q-1}[i] \rightarrow_1 \text{ And } R_{q-2}[i - 1] \rightarrow_1 \text{ And } (R_q[i - 1] \rightarrow_1 \text{ Or } V[T[i]]).$$

Время выполнения такого алгоритма имеет порядок  $O(k \lceil m/w \rceil n)$ , где  $w$  — длина (в битах) представления данных в компьютере (скажем, для типа `LongInt` это 32 бита).



## Упражнения

1. Приведите пример  $P$  и  $T$ . Вычислите вхождение  $P$  в  $T$  с точностью до трех несовпадений.
2. Пусть операция вставки имеет вес 2, операция удаления — вес 3, а операция замены — также вес 3. Проанализируйте работу алгоритма С. Ву – Ю. Менбера для этого случая путем поэтапного вычисления бинарных векторов при конкретных значениях  $P$  и  $T$ .
3. Разработайте программную реализацию алгоритма С. Ву – Ю. Менбера.

## 5.3. Задача о $k$ -несовпадениях

Наняли тридцать корректоров, чтобы избежать ошибки, и все равно на титульном листе издания стояло «Британская энциклопедия».

*Илья Ильф и Евгений Петров*

Дан текст  $T$  ( $n = |T|$ ) и образец  $P$  ( $m = |P|$ ). Мера близости определяется как расстояние между  $P$  и подстроками  $T$  длины  $m$ . (Напомним, что расстояние Р. Хемминга между строками одинаковой длины определяется как минимальное количество операций подстановок, необходимых для преобразования одной строки в другую.) Задано неотрицательное целое число  $k$ . Требуется найти все вхождения  $P$  в  $T$  с точностью до  $k$  (включительно) несовпадений.

Рассмотрим два варианта решения этой задачи: модификацию алгоритма *Shift-And* и, вероятно, исторически первый алгоритм Г. Ландау – Ю. Вишкина.

### 5.3.1. Модификация алгоритма Shift-And

В алгоритмах *Shift-And* и С. Ву – Ю. Менбера рассматривались бинарные векторы, отражающие характер взаимосвязи символов префиксов образца  $P$  и подстроки  $T$ . Основная идея первого из рассматриваемых алгоритмов заключается в том, чтобы использовать для этих целей не один бит, а  $t$  битов, где  $t = \lceil \log_2 k \rceil + 1$  (назовем эти  $t$  битов

«обобщенным битом»). Это значение позволяет фиксировать все несовпадения от 0 до  $k$ . Факт же большего количества несовпадений нас не интересует с точки зрения их количества, но наличие этого факта фиксируется в специальном векторе переполнения  $R'$ . Если проводить аналогию с ранее рассмотренными алгоритмами, то  $R[i, j]$  состоит из  $t$  битов, а его значение  $q$  ( $q = 0..k$ ) говорит о том, что префикс  $P[1..j]$  и подстрока  $T[j+1..i]$ , заканчивающаяся в позиции  $i$ , имеют  $q$  несовпадений. Логические операции в алгоритмах *Shift-And* и *S. Бу – Ю. Менбера* преобразуются в этом случае в операции сложения по модулю  $k + 1$  для каждого обобщенного бита векторов  $R[i]$  и  $V[T[i]]$  и сдвига вектора  $R[i]$  на  $t$  битов вправо (на «обобщенный бит»). Очевидно, что в этом случае и факт вхождения символов алфавита в образец  $P$  требует для каждой позиции характеристических векторов  $t$  битов — «обобщенного бита». Например, табл. 5.4 для  $P = \text{bacaba}$ ,  $A = [a, b, c, d]$  и  $t = 2$  выглядит так, как представлено в табл. 5.5.

Таблица 5.5

V	b	a	c	a	b	a
a	01	00	01	00	01	00
b	00	01	01	01	00	01
c	01	01	00	01	01	01
d	01	01	01	01	01	01

Проиллюстрируем логику работы этого алгоритма традиционным методом.

### Пример

$P = \text{bacaba}$ ,  $T = \text{badabaacbabaca}$ ,  $k = 3$ . Образец  $P$  входит в подстроку:

- $\text{badaba}$  — с одним несовпадением;
- $\text{baacba}$  — с двумя несовпадениями;
- $\text{acbaba}$  — с тремя несовпадениями;
- $\text{babaca}$  — с двумя несовпадениями.

В табл. 5.6 эти вхождения выделены полужирным шрифтом в позиции  $R'[m]$ .

Таблица 5.6

Символ $T$	Вектор $R$	Вектор $R'$	Символ $T$	Вектор $R$	Вектор $R'$
b	000000	011111	c	011133	000000
	011101			110111	
	011101	011111		121200	000011
a	001110	001111	b	012120	000001
	101010			011101	
	102120	001111		023221	000001
d	010212	000111	a	002322	000000
	111111			101010	
	121323	000111		103332	000000
a	012132	000011	b	010333	
	101010			011101	
	113102	000011		021030	000101
b	011310	000001	a	002103	000010
	011101			101010	
	022011	000101		103113	000010
a	002201	000010	c	010311	000001
	101010			110111	
	103211	000010		120022	000101
a	010321	000001	a	012002	000010
	101010			101010	
	111331	000001		113012	000010

Начальное состояние вектора  $R = 000000$  (из «обобщенных битов»), а вектора  $R' = 011111$ . Обработывается символ  $T[1]$ :  $R[1] = R[0] + V['b'] = 011101$  (табл. 5.6). Затем вектор  $R$  сдвигается вправо на  $t$  битов, а вектор  $R'$  — на один бит (слева идет, как обычно, дополнение нулями). Затем обрабатывается символ  $T[2] = 'a'$ . При обработке символа  $T[4]$  в пятом разряде (или  $m - 1$ ) происходит сложение  $3 + 1 = 0 \pmod{4}$  — переполнение, которое фиксируется в соответствующем бите  $R'$ , но в данном случае там уже единица, и ничего не изменяется. Однако в аналогичной ситуации при обработке  $T[5] = 'b'$  в четвертом разряде  $R'$  появляется единица, отмечая факт переполнения.

При обработке символа  $T[6] = 'а'$  мы находим первое 1-несовпадение:  $R'[m] = 0$  и  $R[m] = 1$ . Однако уже при обработке следующего символа мы получаем пять несовпадений —  $R'[m] = 1$ , а это уже четыре (или более в общем случае) несовпадения, плюс одно несовпадение дает  $R[m] = 1$ . Обобщая, можно утверждать, что величина  $R[j] + 2^t R'[j]$  дает нижнюю оценку количества несовпадений префикса  $P[1..j]$  с соответствующей подстрокой  $T$ . Продолжая обработку  $T$ , находим еще три указанных выше несовпадения, удовлетворяющих условиям задачи.

Возможен также следующий вариант работы с вектором переполнения. Каждый обобщенный бит вектора  $R$  дополняется одним разрядом. После выполнения операции сложения эти разряды извлекаются из вектора (логическая операция `And` с соответствующим образом подобранной константой), а затем инвертируются в нулевое состояние. По извлеченным данным может быть получен  $R'$  и его  $m$ -й бит.

*Оценка времени работы алгоритма.* Предположим, что вектор  $R$  может быть размещен в машинном слове ( $w = 32$ ). Например, для случая  $k \leq 3$ ,  $m \leq 10$  мы имеем:  $m \cdot (\lceil \log_2 k \rceil + 1) < w$ . Тогда операции по обработке каждого символа  $T$  будут выполняться за константное время, и мы получаем оценку  $O(n)$ .

### 5.3.2. Алгоритм Г. Ландау – Ю. Вишкина

В алгоритме Г. Ландау – Ю. Вишкина задача о  $k$ -несовпадениях решается с несколько других позиций. Используется метод предварительного анализа образца, но он отличается от ранее рассмотренных (от выделения граней префиксов или блоков строки).

Обратимся для «вычленения» ключевой идеи алгоритма к примеру. Пусть  $T$ ,  $P$  и  $k$  те же, что и в ранее рассмотренной модификации алгоритма *Shift-And*. В табл. 5.7 представлена обычная логика поиска образца в тексте. Единственное отличие — при каждом прикладывании образца к тексту в соответствующей строке матрицы несовпадений ( $pc$ ) фиксируются номера позиций несовпавших символов  $P$ . Первоначально  $pc$  заполняется значением  $m + 1$ . В последнем столбце таблицы приводится количество несов-

падений ( $cnt$ ) при каждом прикладывании  $P$  к  $T$  (на каждом шаге). Решение задачи достигается при 0, 4, 6 и 8 прикладываниях.

Таблица 5.7

Номер шага	b	a	d	a	b	a	a	c	b	a	b	a	c	a	$pc$	1	2	3	4	$cnt$
0	b	a	c	a	b	a									0	3	7	7	7	1
1		b	a	c	a	b	a								1	1	2	3	4	4
2			b	a	c	a	b	a							2	1	3	5	6	4
3				b	a	c	a	b	a						3	1	2	3	5	4
4					b	a	c	a	b	a					4	3	4	7	7	2
5						b	a	c	a	b	a				5	1	4	5	6	4
6							b	a	c	a	b	a			6	1	2	3	7	3
7								b	a	c	a	b	a		7	1	2	3	4	4
8									b	a	c	a	b	a	8	3	5	7	7	2

Приведем обычную логику поиска, чтобы пояснить суть проводимых изменений:

**Procedure** Solve;

**Var**  $i, j, cnt$ : Integer;

**Begin**

**For**  $i:=0$  **To**  $n-m$  **Do** **Begin**

$cnt:=0$ ;

$j:=1$ ;

**While** ( $j \leq m$ ) **And** ( $cnt \leq k$ ) **Do** **Begin**

**If**  $P[j] \neq T[i+j]$  **Then** **Begin**

$cnt:=cnt+1$ ;

$pc[i, cnt]:=j$ ;

**End**;

$j:=j+1$ ;

**End**;

**End**;

**End**;

Наша очевидная цель — изменение данного принципа формирования матрицы  $pc$ . Но она (матрица) должна формироваться, поэтому первым «кирпичиком здания» является логика получения значений части конкретной строки  $pc$

(индекс  $i$ ) при сравнениях, проводимых с конкретной позиции в  $P$  (индекс  $j$ ). Другими словами, мы должны уметь «вытягивать» («extend») при необходимости строку  $pc$ . Приведем соответствующую логику:

```

Procedure Extend(i:Integer; Var j, cnt:Integer);
Begin
  While (cnt<k+1) And (j-i<m) Do Begin
    j:=j+1;
    If T[j]<>P[j-i] Then Begin
      cnt:=cnt+1;
      pc[i,cnt]:=j-i;
    End;
  End;
End;

```

Как видим, вызов `Extend(0, 0, 0)` приводит к формированию первой строки  $pc$ , т. е. отражает результат нулевого прикладывания  $P$  к  $T$  (см. табл. 5.7).

И вот подошло время для рассмотрения ключевого вопроса. Первоначально упростим его. Какую предварительную работу с образцом  $P$  следует выполнить для исключения посимвольного сравнения  $P$  с  $T$  на первом шаге, зная результаты нулевого шага? Обратимся к табл. 5.7. Процедурой `Solve` на этом шаге подстрока  $T = adabaa$  сравнивается с  $P = bacaba$ . Но на предыдущем шаге фрагмент подстроки  $adaba$  уже сравнивался с  $acaba$  и результат этих действий уже зафиксирован в  $pc[0]$ ! Вывод уже почти напрашивается: если на стадии предварительной обработки в некоторой структуре данных будут сформированы результаты посимвольного сравнения  $bacab$  с  $acaba$ , то они могут быть использованы для сопоставления подстрок  $adaba$  и  $bacab$ . Действительно, например, первые символы  $bacab$  и  $acaba$  не совпадают, а первые символы  $acaba$  и  $adaba$  совпадают. Значит, первые символы  $bacab$  и  $adaba$  не совпадают. Осталось продумать все возможные варианты сравнений, но ключевая идея предварительной обработки у нас уже есть — следует зафиксировать результаты сравнения подстрок  $P$  между собой в некоторой структуре данных (матрице  $pn[1..m - 1, 1..2k + 1]$ ). Результаты этих действий для рассматриваемого примера приведены в табл. 5.8.

Таблица 5.8

Подстроки $P$	Строка матрицы $pn$	Номер
bacab	1 2 3 4 5 7 7	1
acaba		
baca	1 3 7 7 7 7 7	2
caba		
bac	1 2 3 7 7 7 7	3
aba		
ba	7 7 7 7 7 7 7	4
ba		
b	1 7 7 7 7 7 7	5
a		

Длительность такой предварительной обработки  $P$  имеет оценку  $O(m \cdot k)$ .

**Procedure** Form\_pn;

**Var** i, j, t: Integer;

**Begin**

**For** i:=1 **To** m-1 **Do Begin**

j:=0;

t:=0;

**While** (j<2\*k+1) **And** (j<m-i) **Do Begin**

j:=j+1;

**If** P[j]<>P[j+i] **Then Begin**

t:=t+1;

pn[i, t]:=j;

**End;**

**End;**

**End;**

**End;**

А сейчас сравним  $pn[1]$  из табл. 5.8 и  $pc[1]$  из табл. 5.7. Оказывается, что можно переписать  $k + 1$  значение из строки  $pn[1]$  в строку  $pc[1]$ , и при этом будет выполнено только одно дополнительное сравнение символов из  $P$  и  $T$ , а именно  $T[3]$  и  $P[2]$ . Строка  $pc$  как бы «поглотила» результаты предыдущей работы. В данном случае она сформировалась полностью, но возможно и ее только частичное формирование (если не набирается  $k + 1$  несовпадение и строка  $P$  не

пройдена до конца при очередном прикладывании). В этом случае следует запустить процедуру `Extend` с конкретными параметрами. Если вынести действия по «поглощению» в отдельную процедуру `Merge`, то разрабатываемый вариант решения будет иметь следующий вид:

```

Procedure Solve;
  Var i, j, cnt, r: Integer;
  Begin
    r:=0;
    j:=0;
    For i:=0 To n-m Do Begin
      cnt:=0;
      If i<j Then Merge(i, r, j, cnt);
      If cnt<k+1 Then Begin
        r:=i;
        {Запоминаем номер последнего расширяемого
         прикладывания или номер строки,
         на которой будет достигнуто значение j}
        Extend(i, j, cnt);
      End;
    End;
  End;

```

Что же происходит в `Solve`? Очевидно, что индекс  $i$  «идет по тексту» и указывает на левую границу сравниваемого участка, а значение  $j$  — это самая правая позиция текста, достигнутая на предыдущих шагах. Образец  $P$  просматривается одновременно с текстом. На шаге  $i$  с  $P$  сравнивается часть текста  $T[i + 1..i + m]$  (где  $i$  — номер строки матрицы  $pc$ ). Значение  $j$  равно  $r + pc[r, k + 1]$  (или  $r + m$  в случае, когда  $pc[r, k + 1] = m + 1$ ). А значение  $r$  — это номер строки, при анализе которой было достигнуто значение  $j$ . При  $i < j$  вызываемая процедура `Merge` находит несовпадения между  $P[1..j - i]$  и  $T[i + 1..j]$  путем, как мы образно называли, «поглощения». При этом определяется значение  $cnt$  — количество несовпадений, или номер столбца в  $pc$ . Если в результате окажется, что значение  $cnt < k + 1$ , то вызывается процедура `Extend`, заканчивающая формирование строки  $i + 1$  матрицы  $pc$  (изменяются  $r, j$  и, возможно,  $cnt$ ). Если же по-прежнему останется  $cnt < k + 1$ , то будет найдено очередное  $k$ -несовпадение.

Оценим время работы `Solve` (время анализа текста). Если исключить вызовы процедур `Merge` и `Extend`, то каждая из  $n - m + 1$  итераций цикла выполняется за фиксированное время, что дает в общей сложности оценку  $O(n)$ . Общее число операций, выполняемых процедурой `Extend` при всех вызовах, есть  $O(n)$ , — каждый символ текста проверяется не более одного раза. Если же окажется, что процедура `Merge` работает за время  $O(k)$ , то общее время анализа текста составит  $O(n \cdot k)$ .

Рассмотрим логику работы процедуры `Merge` (рис. 5.1). Формируется строка  $i$  матрицы  $pc$ . Известно, что на шаге  $r$  достигнута позиция  $j$  текста и результаты этого прикладывания отражены в  $pc[r]$ . Кроме того, на стадии предварительной обработки сформирована матрица  $pn$ , в частности строка  $pn[i - r]$ . Другими словами, нам известны результаты посимвольного сравнения подстроки  $P[1..j - i]$  с подстрокой  $P[j - i + 1..m]$ . На основании этих данных мы должны сделать вывод о результате сравнения  $P[1..j - i]$  и  $T[j - i]$ . Если количество несовпадений «наберется» (будет равно  $k + 1$ ), а их количество фиксируется в счетчике  $cnt$ , то следует перейти на следующий шаг, а если нет, то надо задействовать в анализе оставшуюся часть образца (она заштрихована на рис. 5.1). Но это задействование будет осуществляться уже не в процедуре `Merge`.

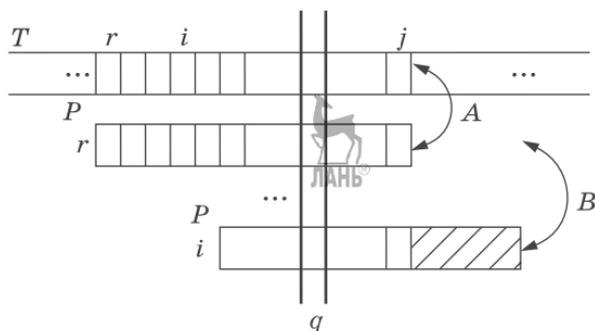


Рис. 5.1. Пояснение к логике работы процедуры `Merge`

Пусть анализируется конкретная позиция  $q$  (рис. 5.1). Имеются результаты сравнений  $A$  и  $B$ , на основании которых необходимо сделать заключение о сравнимости символов  $P[q]$  и  $T[i + 1 + q]$ . Все возможные варианты представлены в табл. 5.9.

Таблица 5.9

A	B	Комментарий
=	=	Совпадение; позиции $q$ нет в матрицах $pc$ и $pn$ . Переход к следующему значению $q$
=	$\neq$	Несовпадение; позиции $q$ нет в $pc$ , но она есть в $pn$ . Номер позиции $q$ следует записать в $pc[i, cnt]$
$\neq$	=	Несовпадение; позиция $q$ есть в $pc$ , но ее нет в $pn$ . Номер позиции $q$ следует записать в $pc[i, cnt]$
$\neq$	$\neq$	Неопределенность. Требуется сравнение символов $P[q]$ и $T[i + q + 1]$ , по результатам которого принимается решение

Завершая обсуждение алгоритма Г. Ландау – Ю. Вишкина, приведем диаграмму срабатываний процедур `Extend` и `Merge` при формировании матрицы  $pc$  для рассматриваемого ранее примера (рис. 5.2). Значения элементов  $pc$  даны в табл. 5.7, а на рис. 5.2 указано, с помощью какой процедуры они формируются (обозначения «E» и «M»). Длина строки диаграммы при этом пропорциональна количеству элементов в строке  $pc$  (их четыре, так как  $k = 3$ ).

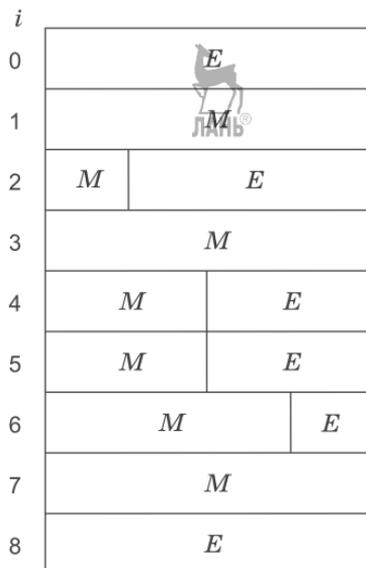


Рис. 5.2. Диаграмма срабатываний процедур `Extend` и `Merge` при формировании матрицы  $pc$

 **Упражнения**

1. Приведите примеры строк  $P$  и  $T$ , задайте их значение  $k$  и выполните ручную трассировку работы модифицированного алгоритма *Shift-And* поиска  $k$ -несовпадений (по типу табл. 5.6).
2. Разработайте программную реализацию модифицированного алгоритма *Shift-And* с временной оценкой  $O(n)$ .
3. Задайте значения  $P$ ,  $T$  и  $k$ . Процедура `Extend` алгоритма Г. Ландау – Ю. Вишкина вызывается с определенными параметрами. Укажите то место текста, с которого будет происходить сравнение символов.
4. Пусть  $P = a^m$  и  $T = a^n$  ( $n > m$ ). Проанализируйте работу обоих рассмотренных алгоритмов при этих исходных данных.
5. Пусть в  $P$  нет повторяющихся символов и ни один символ из  $P$  не совпадает с символами из  $T$  (т. е.  $P$  и  $T$  состоят из разных символов). Проанализируйте работу рассмотренных алгоритмов при этих исходных данных.
6. Задайте значения  $P$ ,  $T$  и  $k$ . Без использования программного кода алгоритма Г. Ландау – Ю. Вишкина составьте диаграмму срабатываний процедур `Extend` и `Merge` (по типу приведенной на рис. 5.2).
7. Разработайте программную реализацию алгоритма Г. Ландау – Ю. Вишкина.



## 5.4. Алгоритм Ю. Майерса

Всё, в том числе и ложь, служит истине. Тени не гасят солнца.

Франц Кафка

Алгоритм Ю. Майерса для приближенного поиска образца в тексте считается одним из самых эффективных. Рассмотрим его идейную основу на «сквозном» примере<sup>1)</sup>.

<sup>1)</sup> Изложение максимально приближенно к тексту книги Б. Смита (с. 325–332) по причине, которая станет ясна в конце этого раздела.

Пример

$S_1 = \text{aabcab}$  ( $m = |S_1|$ ),  $S_2 = \text{caabb}$  ( $n = |S_2|$ ).

Введем матрицу  $V$ :

$$V[i, j] = \begin{cases} 1, & \text{при } S_1[i] \neq S_2[j], \\ 0, & \text{при } S_1[i] = S_2[j]. \end{cases}$$

Для  $S_1$  и  $S_2$  она приведена в табл. 5.10.

Таблица 5.10



<b>V</b>	<b>a</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>
<b>c</b>	1	1	1	0	1	1
<b>a</b>	0	0	1	1	1	1
<b>a</b>	0	0	1	1	1	1
<b>b</b>	1	1	0	1	1	0
<b>b</b>	1	1	0	1	1	0

Подсчитаем (так, как это сделано в п. 4.1) матрицу расстояний (табл. 5.11):  $D[0, 0] = 0$ ;  $D[0, j] = j$  для  $j$  от 1 до  $m$ ;  $D[i, 0] = i$  для  $i$  от 1 до  $n$ ;  $D[i, j] = \min\{D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + V[i, j]\}$  при  $1 \leq i \leq n, 1 \leq j \leq m$ . Если окажется, что  $D[i, m] \leq k$ , где  $k$  — целое неотрицательное число, то решается и задача поиска  $S_1$  в  $S_2[1..i]$  с заданным приближением  $k$ .

Таблица 5.11

<b>D</b>		<b>a</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>
	0	1	2	3	4	5	6
<b>c</b>	1	1	2	3	3	4	5
<b>a</b>	2	1	1	2	3	3	4
<b>a</b>	3	2	1	2	3	3	4
<b>b</b>	4	3	2	1	2	3	3
<b>b</b>	5	4	3	2	2	3	3

Из способа построения  $D$  мы видим, что значение разности  $D[i, j] - D[i-1, j]$  принадлежит множеству из трех элементов  $\{-1, 0, 1\}$ ; аналогично дела обстоят для  $D[i, j] - D[i, j-1]$  и  $D[i, j] - D[i-1, j-1]$ .

Введем, исходя из этой особенности  $D$ , две новых матрицы —  $\Delta$  и  $\Delta'$  (табл. 5.12 и 5.13):

- $\Delta[i, j] = D[i, j] - D[i, j - 1]$  — горизонтальные разности;
- $\Delta'[i, j] = D[i, j] - D[i - 1, j]$  — вертикальные разности.

Таблица 5.12

$\Delta$		а	а	б	с	а	б
	0	1	1	1	1	1	1
с	0	0	1	1	1	1	1
а	0	-1	0	1	1	0	1
а	0	-1	-1	1	1	0	1
б	0	-1	-1	-1	1	1	0
б	0	-1	-1	-1	0	1	0

Таблица 5.13

$\Delta'$		а	а	б	с	а	б
	0	0	0	0	0	0	0
с	1	0	0	0	-1	-1	-1
а	1	0	-1	-1	0	-1	-1
а	1	1	0	0	0	0	0
б	1	0	1	-1	-1	0	-1
б	1	1	1	1	0	0	0

Следующие зависимости составляют основу алгоритма Г. Майерса:

- $\Delta[i, j] = \min\{D[i - 1, j]; \Delta'[i, j - 1]; V[i, j] - 1\} + (1 - \Delta'[i, j - 1]);$
- $\Delta'[i, j] = \min\{\Delta[i - 1, j]; \Delta'[i, j - 1]; V[i, j] - 1\} + (1 - \Delta[i, j - 1]).$

Как их получить? Проверить просто: подставляем данные и получаем равенство. Обратный же процесс, вероятно, идет из практики. У нас есть цель — разработать алгоритм на основе бинарных операций, и аналог — алгоритм *Shift-And*. Есть наблюдение:  $D$  разбивается на две матрицы, в которых элементы принимают значения из интервала от  $-1$  до  $1$ . Далее из практики (из примеров) мы получаем требуемые зависимости и делаем следующий шаг, а именно избавляемся от  $-1$ , ибо в бинарных операциях ее присутствие нежелательно.

Схема исключения  $-1$  основана на замене каждой из матриц  $\Delta$  и  $\Delta'$  на очередные две матрицы. Для  $\Delta$  — это матрицы  $P$  и  $M$ , а для  $\Delta'$  — матрицы  $P'$  и  $M'$ .

Вводим эти матрицы:

$$P[i, j] = \begin{cases} 1, & \text{если } \Delta[i, j] = 1; \\ 0 & \text{— в противном случае} \end{cases}$$

и

$$M[i, j] = \begin{cases} 1, & \text{если } \Delta[i, j] = -1; \\ 0 & \text{— в противном случае.} \end{cases}$$

Тогда  $\Delta[i, j] = \begin{cases} -1 \Leftrightarrow M[i, j] = 1; \\ 0 \Leftrightarrow M[i, j] = 0 \text{ и } P[i, j] = 0; \\ 1 \Leftrightarrow P[i, j] = 1. \end{cases}$

Аналогично, матрица  $\Delta'$  разбивается на две матрицы  $P'$  и  $M'$ . В табл. 5.14–5.17 приведены значения элементов матриц  $P$ ,  $M$ ,  $P'$  и  $M'$  для рассматриваемого в данном разделе сквозного примера.

Таблица 5.14

<i>P</i>	<b>a</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>
<b>c</b>	0	1	1	1	1	1
<b>a</b>	0	0	1	1	0	1
<b>a</b>	0	0	1	1	0	1
<b>b</b>	0	0	0	1	1	0
<b>b</b>	0	0	0	0	1	0

Таблица 5.15

<i>P'</i>	<b>a</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>
<b>c</b>	0	0	0	0	0	0
<b>a</b>	0	0	0	0	0	0
<b>a</b>	1	0	0	0	0	0
<b>b</b>	0	1	0	0	0	0
<b>b</b>	1	1	1	0	0	0

Таблица 5.16

<i>M</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>
<i>c</i>	0	0	0	0	0	0
<i>a</i>	1	0	0	0	0	0
<i>a</i>	1	1	0	0	0	0
<i>b</i>	1	1	1	0	0	0
<i>b</i>	1	1	1	0	0	0

Таблица 5.17

<i>M'</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>
<i>c</i>	0	0	0	1	1	1
<i>a</i>	0	1	1	0	1	1
<i>a</i>	0	0	0	0	0	0
<i>b</i>	0	0	1	1	0	1
<i>b</i>	0	0	0	0	0	0

Цель достигнута: матрицы  $\Delta$  и  $\Delta'$  сведены к бинарным матрицам  $P$ ,  $M$ ,  $P'$  и  $M'$ . Осталось сделать следующий шаг — логику вычисления  $\Delta$  и  $\Delta'$  преобразовать в логику вычисления  $P$ ,  $M$ ,  $P'$  и  $M'$ .

Введем формулы ( $K[j]$  — вспомогательная величина, смысловой нагрузки она не несет):

- $K[j] = \text{Not}(V[i, j]) \text{ Or } M[i - 1, j]$ ;
- $P[i, j] = M'[i, j - 1] \text{ Or } \text{Not}(K[j] \text{ Or } P'[i, j - 1])$ ;
- $M[i, j] = P'[i, j - 1] \text{ And } K[j]$ .

Выскажем утверждение о том, что  $\Delta[i, j] = P[i, j] - M[i, j]$ . Естественно возникновение вопроса — а как они (эти формулы) получены? Попытаемся восстановить этот процесс. Зависимость  $\Delta[i, j] = \min\{\Delta[i - 1, j]; \Delta'[i, j - 1]; V[i, j] - 1\} + (1 - \Delta'[i, j - 1])$  уже выведена. В правой части этого равенства величины  $\Delta[i - 1, j]$  и  $\Delta'[i, j - 1]$  принимают значения из множества  $\{-1, 0, 1\}$ , а величина  $V[i, j]$  — из множества  $\{0, 1\}$ . Всего может быть восемнадцать различных значений ( $3 \cdot 3 \cdot 2 = 18$ ). Выпишем их (левая часть табл. 5.18) и подсчитаем значение  $\Delta[i, j]$ . А теперь —

внимание, вопрос: как из элементов матриц  $P$ ,  $M$ ,  $P'$  и  $M'$  получить точно такое же значение  $\Delta[i, j]$ ? Вероятно, только настойчивая работа с примерами позволяет «углядеть» эти зависимости. Правая часть табл. 5.18 дает право считать истинным утверждение  $\Delta[i, j] = P[i, j] - M[i, j]$ .

Таблица 5.18

$\Delta[i - 1, j]$	$\Delta[i, j - 1]$	$V[i, j]$	$\Delta[i, j]$	$K[i, j]$	$P[i, j]$	$M[i, j]$	$\Delta[i, j]$
-1	-1	0	1	1	1	0	1
-1	-1	1	1	1	1	0	1
-1	0	0	0	1	0	0	0
-1	0	1	0	1	0	0	0
-1	1	0	-1	1	0	1	-1
-1	1	1	-1	1	0	1	-1
0	-1	0	1	1	1	0	1
0	-1	1	1	0	1	0	1
0	0	0	0	1	0	0	0
0	0	1	1	0	1	0	1
0	1	0	-1	1	0	1	-1
0	1	1	0	0	0	0	0
1	-1	0	1	1	1	0	1
1	-1	1	1	0	1	0	1
1	0	0	0	1	0	0	0
1	0	1	1	0	1	0	1
1	1	0	-1	1	0	1	-1
1	1	1	0	0	0	0	0

Аналогично устанавливаются зависимости и для  $\Delta'[i, j]$ :

- $K'[j] = \text{Not}(V[i, j]) \text{ Or } M'[i, j - 1]$ ;
- $P'[i, j] = M[i - 1, j] \text{ Or } \text{Not}(K'[j]) \text{ Or } P[i - 1, j]$ ;
- $M'[i, j] = P[i - 1, j] \text{ And } K'[j]$ ,

и утверждение  $\Delta'[i, j] = P'[i, j] - M'[i, j]$ , проверяемое при всех возможных исходных данных (табл. 5.19).

Таблица 5.19

$\Delta[i-1, j]$	$\Delta'[i, j-1]$	$t[i, j]$	$\Delta[i, j]$	$K'[i, j]$	$P[i, j]$	$M[i, j]$	$\Delta[i, j]$
-1	-1	0	1	1	1	0	1
-1	-1	1	1	1	1	0	1
-1	0	0	1	1	1	0	1
-1	0	1	1	0	1	0	1
-1	1	0	1	1	1	0	1
-1	1	1	1	0	1	0	1
0	-1	0	0	1	0	0	0
0	-1	1	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	1	1	0	1	0	1
0	1	0	0	1	0	0	0
0	1	1	1	0	1	0	1
1	-1	0	-1	1	0	1	-1
1	-1	1	-1	1	0	1	-1
1	0	0	-1	1	0	1	-1
1	0	1	0	0	0	0	0
1	1	0	-1	1	0	1	-1
1	1	1	0	0	0	0	0

Весь смысл алгоритма заключается в исключении цикла по  $j$  — в его замене на некую последовательность логических операций, выполняемых с машинным словом длиной  $w$  бит. Обычно  $w$  равно 32 или 64. Если  $m > w$ , то используется

$\left\lfloor \frac{m}{w} \right\rfloor$  смежных слов.

Введем переменную  $cost$  с начальным значением  $m$  и выпишем полученные зависимости:

- $K' := \text{Not}(V[i]) \text{ Or } \text{ShR}(M'[i], 1)$ ;
- $P'[i] := (M[i-1]) \text{ Or } \text{Not}(K' \text{ Or } P[i-1])$ ;
- $M'[i] := P[i-1] \text{ And } K'$ ;
- $cost := cost + P'[i, m] - M'[i, m]$ ;
- $K := \text{Not}(V[i]) \text{ Or } M[i-1]$ ;
- $P[i] := \text{ShR}(M'[i], 1) \text{ Or } \text{Not}(K \text{ Or } \text{ShR}(P'[i], 1))$ ;
- $M[i] := \text{ShR}(P'[i]) \text{ And } K$ .

Использование операции сдвига вправо на один разряд стандартно. Этот прием использовался и в алгоритме *Shift-And* для обращения к  $j - 1$  элементам бинарного вектора.

Однако при внимательном анализе зависимостей мы видим, что они неработоспособны: значение  $K'$  вычисляется через  $M'[i]$ , а значение  $M'[i]$  — с использованием  $K'$ ! Б. Смит<sup>1)</sup> эту неточность разъясняет простой ссылкой на работу Ю. Майерса<sup>2)</sup>, говоря, что выкладки, приводящие к эквивалентной формуле  $K' = \text{Not}(V[i]) \text{ Or } (((\text{Not}(V[i]) \text{ And } P[i - 1]) + P[i - 1]) \text{ Xor } (P[i - 1])))$ , достаточно сложны. Проверка же этого соотношения путем прямого подсчета значений  $K'$  для сквозного примера приведена в табл. 5.20.

Таблица 5.20

$i$	$K' = \text{Not}(V[i])$ Or $\text{ShR}(M'[i], 1)$	$K' = \text{Not}(V[i]) \text{ Or } (((\text{Not}(V[i]) \text{ And } P[i - 1]) + P[i - 1]) \text{ Xor } (P[i - 1])))$
1	000111	000100
2	111111	111111
3	110010	110010
4	001111	011011
5	001001	001001

В первой и четвертой строках таблицы здесь имеет место несовпадение, что говорит о неэквивалентности формул!



### Упражнения

1. Приведите пример двух строк. Вычислите для них матрицу расстояний  $D$  и матрицы разностей  $\Delta$  и  $\Delta'$ .
2. Напишите программу для восстановления матрицы  $D$  по заданным матрицам  $\Delta$  и  $\Delta'$ .
3. Приведите примеры двух строк. Вычислите для них матрицы  $\Delta$  и  $\Delta'$  и проверьте путем подстановки значений истинность зависимостей:

<sup>1)</sup> Смит Б. Методы и алгоритмы вычислений на строках. — М.: ООО «И. Д. Вильямс», 2006. С. 332.

<sup>2)</sup> Myers G. W. A fast bit-vector algorithm for approximate string matching based on dynamic programming // JACM 46-3, 1999. P. 395-415.

- $\Delta[i, j] = \min\{\Delta[i - 1, j]; \Delta[i, j - 1]; V[i, j] - 1\} + (1 - \Delta[i, j - 1]);$
  - $\Delta'[i, j] = \min\{\Delta[i - 1, j]; \Delta'[i, j - 1]; V[i, j] - 1\} + (1 - \Delta[i, j - 1]).$
4. Напишите программу для восстановления матрицы  $D$  по заданным матрицам  $P$ ,  $M$ ,  $P'$  и  $M'$ .
  5. По аналогии с табл. 5.18 и 5.19 составьте для конкретного примера таблицы проверки истинности утверждений  $\Delta[i, j] = P[i, j] - M[i, j]$  и  $\Delta'[i, j] = P'[i, j] - M'[i, j]$ .
  6. Разработайте программную реализацию алгоритма с использованием формулы Ю. Майерса для вычисления значений  $K'$ . Найдите (если они есть) примеры, для которых результат вычислений будет неверным.
  7. Найдите корректный способ вычисления  $K'$  или обоснуйте корректность проведенной замены. (*Примечание.* Автор не смог этого сделать.)

### Методический комментарий

Простой алгоритм решения рассмотренной в этой главе задачи основан на идеях динамического программирования, он достаточно хорошо известен и может быть использован как задача средней сложности при изучении этой темы.

С. Ву (*Sun Wu*) и Ю. Манбер (*Udi Manber*)<sup>1)</sup> в 1992 г. разработали метод, обобщающий алгоритм *Shift-And* на нахождение неточного вхождения образца в тексте.

Решение задачи о  $k$ -несовпадениях представлено модификацией алгоритма С. Ву и Ю. Манбер, а также алгоритмом Г. Ландау и Ю. Вишкина.

Алгоритм Г. Ландау (*Gad Landau*) – Ю. Вишкина (*Uzi Vishkin*)<sup>2)</sup>, кроме того что он относится к числу первых, предназначенных для решения задачи о  $k$ -несовпадениях, имеет свой, отличающийся от ранее рассмотренных, метод предварительной обработки образца  $P$ .

Алгоритм Ю. Майерса (*Gene Myers*)<sup>3)</sup> (точнее, его модификация) считается одним из самых эффективных в решении задачи поиска неточного вхождения образца в текст. Идея моди-

<sup>1)</sup> Wu S., Manber U. Fast text searching allowing errors // CACM 35–10, 1992. P. 83–91.

<sup>2)</sup> Landau G. M., Vishkin U. Efficient string matching with k mismatches // TCS 43, 1986. P. 239–249.

<sup>3)</sup> Myers G. W. A fast bit-vector algorithm for approximate string matching based on dynamic programming // JACM 46–3, 1999. P. 395–415.

фикации заключается в вычислении не полных бинарных векторов длины  $m$ , а только их частей, определяемых константой  $k$ . В этом случае временная оценка преобразуется из  $O\left(\left\lceil \frac{m}{w} \right\rceil \cdot n\right)$  в  $O\left(\left\lceil \frac{k}{w} \right\rceil \cdot n\right)$ .

Проблематика этой главы не исчерпывается описанными алгоритмами. Известны также алгоритмы Э. Укконена<sup>1)</sup>, В. Чанга (*William Chang*) – Е. Лоулера (*Eugene Lawler*)<sup>2)</sup>, Р. Коула (*Richard Cole*) – Р. Харихарана (*Ramesh Hariharan*)<sup>3)</sup> и т. д. Их сравнительный анализ может быть предметом самостоятельного исследования. В работах Р. Беза-Йетс (*Ricardo Baeza-Yates*) и Г. Наварро (*Gonsaio Navarro*)<sup>4)</sup> задача приближенного поиска образца или множества образцов решается с использованием аппарата теории автоматов и опять же идей битового представления данных, т. е. путем сведения определенной части действий алгоритма к логическим операциям с машинными словами.



<sup>1)</sup> *Ukkonen E.* Finding approximate patterns in strings // *J. Algs.* 6, 1985. P. 132–137.

<sup>2)</sup> *Chang W. I., Lawler E. L.* Approximate string matching in sublinear expected time // *Proc. 31st Annual IEEE Symp. Foundations of Computer Science.* Vol. 1. 1990. P. 116–124.

<sup>3)</sup> *Cole R., Hariharan R.* Faster approximate string matching // *Proc. Ninth Annual ACM-SIAM Symp. Discrete Algs.*, 1998. P. 463–472.

<sup>4)</sup> *Baeza-Yates R., Navarro G.* Multiple approximate string matching // *Proc. Fifth Annual Workshop on Algorithms & Data Structures.* F. Dehne et al. (eds.), 1997. P. 174–184.

---

## Вместо заключения

---

- А на той планете есть охотники?  
— Нет.  
— Как интересно! А куры там есть?  
— Нет.  
— Нет в мире совершенства! —  
вдохнул лис.

*Антуан де Сент-Экзюпери,  
«Маленький принц»*

Взгляд на информатику как на предмет в общеобразовательной школе за немногим более чем двадцатилетнее развитие обладал некоей «аномальной» изменчивостью. С одной стороны, эта ситуация обусловлена объективными причинами (динамичность развития данной области действительности и т. д.), а с другой — все ли ладно в нашем «Датском королевстве»? Попробуем как бы забыть на время обо всех существующих концепциях, позициях, учебниках<sup>1)</sup> и т. д. и, опираясь только на более чем тридцатилетний опыт работы как в разработке реальных систем обработки информации, так и в образовательной информатике, сформулировать точку зрения прагматика.

Зададим себе первый наивный вопрос — «что такое информатика»? И оказывается, что дать простой и ясный ответ на него не очень просто! Такие области, как, например, математика или физика, имеют достаточно четкие границы, а информатика? Конечно, допустимо сказать: это — что-то связанное с информационными процессами, с обработкой информации и т. д., но тогда одно понятие будет определяться через другие, образуется длинная понятийная цепочка, часто — тавтологическая, и прагматику не очень ясно, где начало и где конец этого понятийного многообразия.

Упростим (или усложним?) первоначальный вопрос: «Возможна ли информатика без компьютера»? Если ответить «да», то границы этой области становятся еще более не-

---

<sup>1)</sup> Автор не играет здесь в игру «кто прав, кто виноват» или «что такое хорошо, что такое плохо», а рассматривает ситуацию как данность, сложившуюся в ходе исторического развития явления, естественно, в результате объективных и субъективных факторов.

определенными, понятие «информатика» переходит в разряд «сверхоткрытых», и впору задать вопрос — «а что *не есть* информатика»? Но предположим, что мы ответили «нет». В этом случае у нас уже появляется граница, пусть пока только «пунктирная». Но появляется и новое понятие — «компьютер», и тут же возникают вопросы как минимум о взаимосвязи понятий и о том, *что* есть компьютер — объект изучения в информатике или средство (инструмент) изучения информатики?

Определимся с условными границами понятия «компьютер». Самый низший уровень — это «железо» (*hardware*). Безусловно, понимание того, как функционирует аппаратное обеспечение, необходимо. Хорошо, например, знание того, что для реализации всех возможностей этого уровня компьютеру достаточно уметь выполнять только операции инверсии, сдвига на один разряд и прибавления единицы, а все остальные операции конструируются из этих примитивов. Но полезность компьютера при таком ограничении меньше, чем у молотка: последним хотя бы можно гвозди забивать ☺. Таким образом, в понятие «компьютер» следует вложить как минимум операционную систему (ее основные функциональные возможности и структуру) и простейшую систему программирования. После этого компьютер уже выполняет свое основное предназначение — вычисляет и перебирает, т. е. происходит то чудесное «оживление», которое делает компьютер универсальным устройством, универсальным вычислителем и переборщиком вариантов.

Итак, граница очерчена, и просматривается, в каком объеме (хотя бы минимальном) компьютер является объектом изучения. А что дальше? Дальше у нас два пути. Первый — расширяя понятие «компьютер», добавляя к нему уже некий готовый функционал (например, графический редактор), мы можем изучать его и утверждать, что изучается информатика — редакторы же связаны с компьютером! Второй путь — ответить на вопрос, что произошло в цепочке от потребности человека рисовать к появлению у компьютера такого функционала, удовлетворяющего эту потребность.

Оставим на время второй путь и закончим рассуждения о нашем понимании компьютера. Повторим мысль, что возможности компьютера ограничены. Простейший пример (чуть-чуть абсурдный): любой из нас, если потребует, выполнит операцию вычисления  $10^{1000}$ , просто записывая соответствующее количество нулей, а вот компьютер (без спе-

циальных ухищрений) этого не сделает! Естественно, что возможности компьютера увеличиваются со временем — сравним хотя бы компьютеры двадцатилетней давности и нынешние. Расширение возможностей позволяет решать все новые проблемы и совершенствовать решения старых, но ограниченность компьютера остается.

Однако вернемся к нашим рассуждениям об информатике и продолжим их с помощью примера. Человек и до появления компьютера работал с текстом. В зависимости от вида его деятельности, эта работа была различной (писатель, редактор и т. д.). Другими словами, есть *потребность* работы с текстом и есть *проблема* работы с текстом. С появлением компьютера человек «углядел», что последний может быть полезен для удовлетворения этой потребности, и разработал вначале простейший текстовый редактор, а затем текстовый процессор с огромными (если сравнивать с первыми версиями) возможностями. Отметим, что в данном случае (как и в других) совершенство инструмента, а текстовый процессор является именно инструментом, оказывает обратное влияние на процесс работы с текстом (обратная связь). Но не это главное, а продолжение нашего наивного вопроса — «где здесь информатика»?

Итак, в самом общем виде (см. рис. 1) есть действительность, есть проблемы этой действительности и есть отображение этих проблем в некие их решения, использующие компьютер, в результате чего получают некие продукты (причем так называемые «новые информационные технологии» — это лишь небольшая часть этих продуктов), удовлетворяющие потребностям этой действительности.

Примеры можно продолжить. Вот еще один из них для наглядности, связанный, в частности, и с обработкой текстов. В молекулярной биологии и в других областях знаний существует проблема поиска в больших объемах текстовых данных входящих в них образцов. Прообразом этой проблемы является элементарная задача поиска слова в тексте. Постоянное наращивание возможностей уровня *B* (см. рис. 1) не решало запросы действительности. И поэтому в последние 30 лет сформировался новый подраздел информатики — «методы обработки строк».

Вернемся к первоначальному вопросу. Информатика — это уровни *A*, *B*, *C* или *A* и *B*? (Мы оставляем только самые логически разумные сочетания.) Отметим, что при положительном ответе на вопрос о взаимосвязи информатики и



Рис. 1

компьютера («возможна ли информатика без компьютера») в нашей схеме, а она применима не только к информатике, появляется еще один трудно определяемый «прямоугольник».

Далее можно было бы рассуждать следующим образом. Можно взять образовательный стандарт школьного курса, соотнести его положения с приведенной схемой, а затем на основе некоего анализа с учетом привнесенных извне положений сделать определенные выводы и ответить на поставленный вопрос. Очевидно, что в зависимости от этих «привнесенных извне положений» могут получиться различные результаты — вплоть до того, что мы рассуждаем вовсе не о том, или до появления некоей «новой информатики» (например, «асоциальной»). Но мы попробуем пойти другим путем — спроецируем ситуацию на высшую школу, пойдем, как отвечают на поставленный вопрос в ней, и сделаем «обратную проекцию» на общеобразовательную школу. (Такие операции «проецирования», как прямого, так и обратного, не приводят к искажению предмета анализа, ибо речь идет об одном и том же, просто в разных объемах.)

Область высшей школы также поражает своим многообразием направленности подготовки специалистов по информатике, но имеет более определенные границы. Вычленим три основополагающих российских образовательных стандарта.

*Специальность 010200 — «Прикладная математика и информатика», квалификация — математик, системный программист.* Естественно, что здесь осуществляется фундаментальная подготовка по математике. В блоке общеобразовательных дисциплин информатика представлена здесь курсами «Языки программирования и методы трансляции», «Системное и прикладное программное обеспечение» и «Базы данных и экспертные системы». Остальные составляющие подготовки зависят от выбранной в вузе специализации, но в основном, если последняя связана с информатикой, рассматриваются вопросы программирования и функционирования операционных систем разного назначения. Таким образом, согласно нашей схеме (см. рис. 1), это уровни *A* и *B*, причем уровень *A* трактуется чисто с математических позиций. Согласно логике стандарта, подготовка по математике обеспечивает реализацию отображения *A*, с чем, конечно, трудно согласиться. Математическая составляющая в *A*, безусловно, есть, в некоторых проблемах она имеет принципиальное значение, но сведение методов решения проблем только к ней не соответствует реальному состоянию дел. И здесь в информатике имеет место парадоксальная ситуация, образно характеризуемая фразой «сапожник без сапог». Специалисты по информатике создают системы обработки информации различного назначения, в частности автоматизирующие различные виды деятельности, но собственная деятельность по отображению проблемы в некий программный продукт (уровень *A*) соответствующей поддержки практически не имеет! Структурное, объектно-ориентированное проектирование, а также различные технологии типа CASE (Computer Aided Software Engineering) или RAD (Rapid Application Development) лишь частично решают эту проблему, причем последние относят скорее к уровню *C*, а не *A*. Естественно, что эта часть проблематики уровня *A* слабо представлена в образовательных стандартах или вообще никоим образом там не затрагивается.

*Специальность 351400 — «Прикладная информатика (по областям)», квалификация — информатик (квалификация в области).* Для определенности будем считать такой

областью экономику (самый распространенный вариант). Общеобразовательный блок здесь представлен стандартным набором курсов: вычислительные системы, сети и телекоммуникации; информационные системы; базы данных; высокоуровневые методы информатики и программирования; операционные системы, среды и оболочки; информационные технологии. К специальным дисциплинам относятся проектирование информационных систем; интеллектуальные информационные системы; информационная безопасность, а также информационные системы в бухгалтерском учете и аудите, в банковском деле и т. д. Итого, в данном стандарте четко представлен уровень *B* (его программная составляющая) и та часть уровня *C*, которая относится к выбранной отрасли. Уровень *A* дается скорее в технологическом плане, связанном с общими вопросами проектирования информационных систем.

*Специальность 654600 — «Информатика и вычислительная техника»*, квалификация — инженер. Общеобразовательный блок здесь с точностью до деталей совпадает с аналогичным блоком предыдущей специальности, а в специальных дисциплинах (в зависимости от специализации) более детально представлен уровень *B*, например курсами «Теория вычислительных процессов», «Архитектура вычислительных систем» и т. д. Уровень *A* отождествляется с программированием.

С некоторыми оговорками можно считать, что подготовка специалистов по уровню *A* осуществляется в рамках специальности «Прикладная математика и информатика» (квалификация выпускника — математик, системный программист), по уровню *B* — «Информатика и вычислительная техника» (квалификация выпускника — инженер), по уровню *C* — «Прикладная информатика (по отраслям)» (квалификация выпускника — например, информатик-экономист).

Итак, если уровни *B* и *C* достаточно четко очерчиваются, то с уровнем *A* все обстоит несколько сложнее. Так или иначе, в нем присутствует то, что обозначают понятием «программирование». Но само по себе программирование как знание систем программирования и умение что-то на них записывать, пусть даже в формализованном виде, не решает проблемы отображения *A* (см. рис. 1), как не решает ее и чисто математическая составляющая подготовки!

Приведем примеры в подкрепление точки зрения «за» и «против». Новый раздел информатики — методы обработки строк, имеющий огромное значение для многочисленных приложений. Его математическая составляющая минимальна. Другой раздел — методы защиты информации, где без знания математики не достигается даже первичное понимание сути пионерской работы У. Диффи и М. Э. Хеллмана. Но и в том, и в другом случае ключом является понятие «алгоритм» и нечто (вспомним ситуацию «сапожник без сапог»), что мы условно обозначим как *искусство перевода решения проблемы* (заметим — оно не сводится к алгоритму!) на язык, понятный на уровне *B*.

Подведем определенную черту под вышеизложенным. Уровни *B* и *C* четко прописываются в стандартах, чего не скажешь об уровне *A*: общепринятого понимания уровня *A* нет, а различные его трактовки опираются на не очень понятную аксиоматику. Использование же нами понятия «искусство» — вынужденное. Человек, когда его не удовлетворяет определенный язык как инструмент формализации, создает некий «метаязык». Но в данном случае все попытки создания его в информатике трудно считать успешными, и популярное в настоящее время объектно-ориентированное проектирование вряд ли в полном объеме решает проблему.

А в школе, в школьном образовании, делается попытка интеграции уровней в единое целое в рамках ограниченного времени на изучение. И естественно, возникает вопрос — возможно ли решение данной проблемы? Как собрать в малом (по времени) большое, и не просто большое, а постоянно растущее, особенно на уровне *C*, целое?

Помимо теоретической возможности такой интеграции, есть еще один вопрос — а надо ли это делать? Если следовать логике одного из государственных мужей, утверждающего, что основной задачей школы является «подготовка грамотного пользователя технологий» (естественно, с приставкой — «новых информационных»), то, конечно, «да», — и основной акцент должен быть сделан на уровне *C*. Кстати, изучение уровня *C* имеет еще одну особенность, которую просто необходимо отметить. Научно доказано<sup>1)</sup>, что существующие методики изучения уровня *C* (во всяком слу-

<sup>1)</sup> Окулов С. М., Суворова Т. Н. О традиционной методике изучения информационных технологий // Информатика и образование. 2006. № 11. С. 93–96; Суворова Т. Н. Совершенствование методики изучения информационных технологий в школьном курсе информатики: Автореф. дис. ... канд. пед. наук. — М., 2007.

чае те, которые представлены в современных школьных учебниках) не развивают или очень слабо развивают интеллектуальный потенциал школьника. (Мы не говорим, что не *учат*, но это несколько разные срезы явления.) Сумма запоминаемого фактографического материала огромна, так же как и навыки манипулирования мышью. Но ведь святая святых образования — это *развитие* школьника, развитие его умственных способностей и возможностей, позволяющих решать любую встречающуюся проблему, опираясь не только на чувственный аппарат, данный человеку, но и используя мощь всего аналитического аппарата ума, который формируется у человека, в частности, в процессе его обучения. Доказательная база «греховности» традиционных методик изучения информационных технологий предельно проста. Вот одна из возможных схем такого доказательства. Есть, например, *теория поэтапного развития умственных способностей школьника* одного из ведущих психологов и педагогов XX столетия, Петра Яковлевича Гальперина. Опуская многое (что естественно для данного материала), зафиксируем «итога», т. е. главное. Непременным условием умственного развития школьника является определенная познавательная деятельность (вообще говоря, слово «познавательная» можно опустить), включающая в себя действия, классифицируемые как исполнительские, ориентировочные и контрольные. Не будем раскрывать их, ибо это только отяготит текст, но не пояснит суть доказательства. Упрощая, скажем, что в развитии умственных способностей ключевую роль играет преобладание и синтез именно действий второго и третьего типов, и зафиксируем этот момент как некий «оселок» нашего понимания проблемы. Таким образом, выстраиваются необходимые условия развития. А затем мы «расчлняем» методику изучения информационных технологий школьных учебников до уровня конкретных действий — тех, которые должен выполнить школьник, — и сопоставляем (устанавливаем соответствие) их с действиями по П. Я. Гальперину. Если оказывается, что большинство действий относится к первому типу, а не второму и третьему, как необходимо, то доказательство завершено (необходимые условия не выполнены), а мы получаем обучение в соответствии с «принципом банана» (известный образ по выработке условного рефлекса у обезьяны по доставанию банана)...

В заключение необходимо вспомнить об огромной ответственности современного общества перед детьми за их буду-

щее. Любое достижение цивилизации кроме позитивной несет в себе и негативную составляющую. Компьютер — это хорошо или плохо для образования, для развития школьника? Современный школьник не знает таблицы умножения — зачем, если есть калькулятор? Так в чем же заключается негативная составляющая компьютера (а она обязательно есть)? Любому педагогу, связанному с информатикой, приходилось (и, вероятно, не один раз) отвечать на вопрос родителей: «Мы купили ребенку компьютер в надежде на то, что он повлияет на его развитие, а ребенок день и ночь играет — что делать»? Приведем почти абсурдный пример из другой области, чтобы пояснить эту мысль. Потребность человека убивать себе подобных хорошо известна. Человек придумал атомную бомбу, чтобы эффективно убивать. Предположим, что она стала предметом общего пользования. Что тогда произойдет — предсказать нетрудно. А что убьет компьютер? Компьютер — при его бездумном использовании — убьет в человеке творческое начало и сделает из него нечто как начинающееся, так и заканчивающееся примитивным потреблением кем-то созданных услуг! И пусть это несколько гипертрофированное утверждение, но это лучше, чем очередная эйфория, связанная с всеобщей компьютеризацией...



---

---

# Приложения

---



## Приложение 1

Не хватайте меня за палец, а смотрите, куда я указываю.

*У. Маккалок*

### Об организации экспериментального исследования алгоритмов

Стоило только попросить мужчину помочь вымыть посуду — и тут же появилась автоматическая посудомойка.

*Сирил Норткот Паркинсон*

Организация работы по *экспериментальному исследованию алгоритмов (ЭИА)* требует в методическом плане (кроме четко формулируемых проблем) разработки некоего инструментального оборудования, некой программной среды (обозначим ее как *СЭИА*). Она (эта среда) не относится к разряду сложного программного продукта, но создает базу для следующего этапа творческой деятельности (где первый этап — это осознание и понимание того, как авторы алгоритма шли к своему результату).

Чтобы понять ее функциональные возможности, представим себе обычную (традиционную) работу школьника (студента) с алгоритмом. Она проста и выражается одним предложением: после осознания сути алгоритма пишется программный код и проверяется на нескольких случайно подобранных тестах. Затем, конечно, могут проводиться не-

кие модификации или дополнения, но схема работы при этом не изменяется.

Взглянем на поставленную задачу (разработку СЭИА как инструмента творческой деятельности) несколько с другой позиции. Если обратиться к англоязычной периодике по рассматриваемой нами в этой книге проблеме (методам обработки строк), то окажется, что значительная часть статей (не «пионерских») пишется по следующей схеме. Берется алгоритм или несколько алгоритмов одного типа. В лучшем случае (но не всегда) проводится его модификация, а затем даются результаты экспериментальной оценки. Эксперимент же строится опять традиционным образом: серия тестов, на которых оцениваются, например, время работы и используемая память алгоритма (или алгоритмов).

При поддержке такого вида деятельности требования к СЭИА минимальны:

- наличие библиотеки тестов;
- наличие библиотеки модулей вывода результатов в различных форматах;
- наличие управляющей программы по обеспечению процесса тестирования<sup>1)</sup>.

Такая СЭИА должна создаваться для каждого алгоритма, а схема ее работы незначительно отличается от профессиональной проверки (тестирования) обычной задачи.

Опишем коротко функциональные возможности СЭИА, которую желательно иметь учителю для более продуктивной работы со школьниками. Это, конечно же, один из вариантов, — учитель в процессе ее создания (совместно с программистом) и в ходе ее эксплуатации, естественно, внесет свои коррективы, отражающие его специфику работы.

### *1. Общие требования:*

- наличие двух версий: сетевой и локальной;
- возможность управления деятельностью школьников (постановка задач, организация индивидуальной и групповой работы, ведение статистических данных о работе школьника и т. д.);

---

<sup>1)</sup> Организация такой работы (с примером управляющей программы) рассмотрена в книге: *Окулов С. М.* Программирование в алгоритмах. — М.: БИНОМ. Лаборатория знаний, 2002. С. 325–328.

- возможность логической организации проблем (задач) в группы (для конкретного школьника); при этом одна задача может находиться сразу в нескольких группах;
- возможность добавления, удаления и редактирования групп, задач, тестов, модулей оценки результатов; централизованное хранение тестов и результатов тестирования;
- возможность фиксировать и регулировать время работы программы, а также выставлять максимально допустимый объем памяти, который может использовать задача.

## *2. Требования к интерфейсу:*

- простой и понятный интерфейс;
- одинаковый интерфейс для обеих версий системы — локальной и сетевой.

## *3. Требования к сетевой версии:*

- наличие системы аутентификации — запроса имени пользователя и пароля;
- поддержка двух режимов:
  - школьник может отправить учителю на проверку любую из задач и получить результат проверки в целом;
  - школьник может исследовать любую из задач, получать доступ к библиотеке тестов и модулям обработки результатов, создавать собственные тесты и вспомогательные программы.

*4. Требования к локальной версии* — обеспечивается второй режим работы школьника (из указанных выше для сетевой версии).

*5. Возможности СЭИА, предоставляемые учителю* (как минимум):

- управление группами задач и отдельными задачами (добавление, редактирование, удаление);
- управление библиотекой тестов и модулей оценки результатов (создание, добавление, редактирование, удаление);
- управление работой школьников (статистические данные, создание групп задач и т. д.).

### 6. Основные компоненты СЭИА (сетевая версия):

- серверная часть, обеспечивающая всю функциональность по работе со школьниками;
- клиентская часть, обеспечивающая пересылку решений школьников на сервер и отображение (фиксацию) результатов работы;
- управляющая часть, обеспечивающая настройку системы.

*Локальная версия* — это (в основном) третий компонент сетевой версии.

### 7. Требования к задачам, добавляемым в СЭИА.

При добавлении новой задачи в систему необходимо учитывать, что она может как быть исследовательской, так и использоваться затем для проверки знаний школьника путем его индивидуального тестирования или организации, например, турниров типа олимпиад. В зависимости от этого школьнику предоставляются различные права работы с ресурсами системы. В любом случае результаты решения задачи должны оформляться в виде выходного файла, формат которого заранее оговаривается. Это позволит, в частности, при наличии эталонного решения учителя (или другого школьника) осуществлять простое сравнение файлов. Этот момент особенно важен при работе с исходными данными большого объема.

Выше «очерчены» лишь «контуры» СЭИА с целью обоснования утверждения о необходимости использования определенного формализма в деятельности как учителя, так и школьника. Так или иначе, но при проведении серьезной индивидуальной работы даже школьник создает в процессе нее различного рода вспомогательные программы, наборы файлов для анализа решения и т. п. Определенная унификация (систематизация) этого аспекта работы повышает общую продуктивность, ибо освобождает от многих технических деталей как школьника, так и учителя. Но это — действительно только «контуры», поскольку в них не обозначены многие детали (например, необходимость библиотеки для работы со строками большой длины и пр.). Главное — захотеть, результат придет.

## Приложение 2

Человек начинает жить лишь тогда, когда ему удастся превзойти самого себя.

*Альберт Эйнштейн*



### Проблемы и задачи

Порой задача столь сложна, что возникает желание посмотреть в глаза ее автору.

*Георгий Александров*

Создание СЭИА (см. приложение 1) — это как бы технический аспект деятельности. Но главное все же — это проблемы и задачи (содержание), которые необходимо ставить перед школьником (студентом), ибо их решение и исследование не только дают то состояние интеллектуальной радости, которое делает человека человеком, но и может стать «осью», «стержнем» вызревания профессионала в информатике. Претензий на некую полноту проблем и задач, приведенных в данном приложении и охватывающих рассматриваемую проблематику, конечно же, нет, — речь идет только о примерах<sup>1)</sup>.

**Проблема.** Ниже приведен программный код Р. Коула, реализующий предварительную обработку образца  $P$  для сильного правила хорошего суффикса<sup>2)</sup> (алгоритм Р. Бойера – Дж. Мура). Требуется из этого кода извлечь идею и алгоритм обработки и оценить время его работы, — но не только! Необходимо исследовать возможность применения выявленного метода предварительной обработки для создания алгоритмов поиска образца в тексте. Другими словами, следует попытаться разработать свой алгоритм и оценить его эффективность.

<sup>1)</sup> Обработка задачного материала по данной (равно как и родственной) проблематике и его классификация — это содержание отдельной работы, которой еще нет в учебной литературе по информатике.

<sup>2)</sup> *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ–Петербург, 2003. С. 56–57.

```

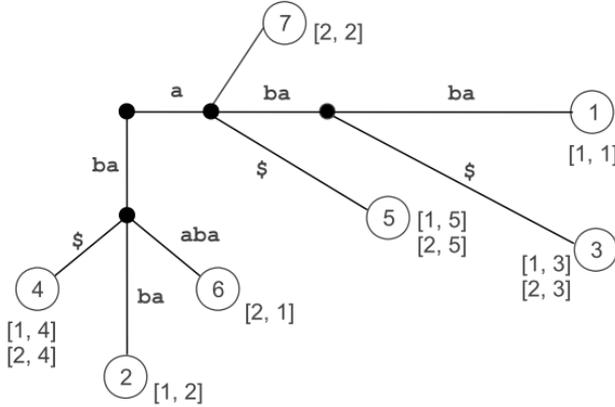
Procedure GSshift (P:String; Var gs_shift:IndexArray;
                    m:Integer);
Var i,j,j_old,k:Integer;
    kmp_shift:IndexArray;
    {Тип используемых массивов имеет вид:
     IndexArray=Array[1..NMax] Of Integer;}
    go_on:Boolean;
Begin
  For j:=1 To m Do gs_shift[j]:=m;
  kmp_shift[m]:=1;
  j:=m;
  For k:=m-1 DownTo 1 Do Begin
    go_on:=True;
    While (P[j]<>P[k]) And go_on Do Begin
      If (gs_shift[j]>j-k) Then gs_shift[j]:=j-k;
      If (j<m) Then j:=j+kmp_shift[j+1]
      Else go_on:=False;
    End;
    If (p[k]=p[j]) Then Begin
      kmp_shift[k]:=j-k;
      j:=j-1;
    End
    Else kmp_shift[k]:=j+1;
  End;
  j:=j+1;
  j_old:=1;
  While (j<=m) Do Begin
    For i:=j_old To j-1 Do
      If (gs_shift[i]>j-1) Then gs_shift[i]:=j-1;
      j_old:=j;
      j:=j+kmp_shift[j];
    End;
  End;

```

**Задача.** Дано  $k$  строк  $S_1, S_2, \dots, S_k$ . Требуется построить для них обобщенное дерево суффиксов<sup>1)</sup>, т. е. дерево, в котором представлены суффиксы всех строк.

<sup>1)</sup> Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология: Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 151–152.

**Комментарий.** Пусть  $S_1 = ababa\$, S_2 = baaba\$$  (к строкам добавлен конечный символ « $\$$ »). Тогда получаемое обобщенное дерево суффиксов представлено на рис. П2.1. Для листьев в квадратных скобках приведены метки, состоящие из двух чисел: [номер слова, номер суффикса].



**Рис. П2.1.** Обобщенное дерево суффиксов (без суффиксных связей) для строк  $S_1 = ababa$  и  $S_2 = baaba$

**Логика построения.** Строится дерево для  $S_1$  (например, с помощью алгоритма Э. Укконена), а затем дерево достраивается для  $S_2$  обычными действиями алгоритма Э. Укконена (с использованием суффиксных связей и скачков по счетчику). В общем случае процесс продолжается до тех пор, пока не будут обработаны все строки. Здесь требуют уточнения лишь два момента. Во-первых, лист дерева может относиться к суффиксам различных строк, поэтому для каждого листа следует хранить список строк и номеров суффиксов. Во-вторых, метки дуг, в виде индексов начального и конечного символов в строке, претерпевают изменение. Необходимо связывать эти данные с номером строки.

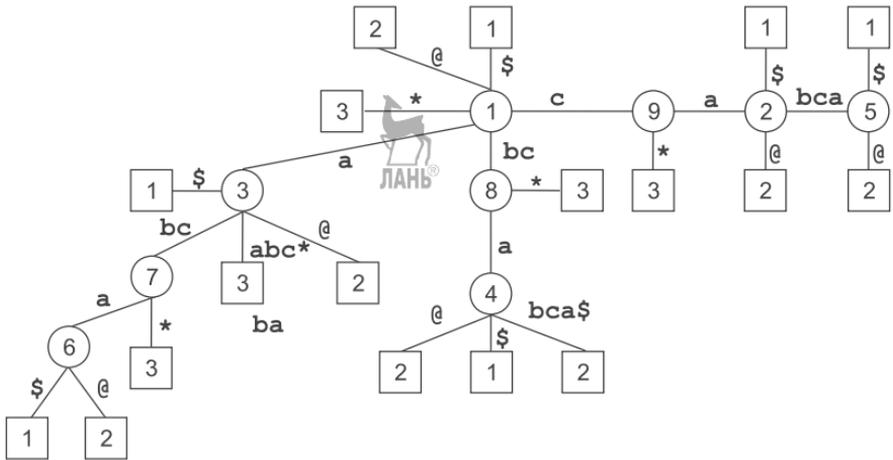
Обобщенное дерево суффиксов для заданного множества строк с суммарной длиной  $n$  может быть инструментом для решения задач поиска. Предположим, что дерево построено, и необходимо ответить на вопрос, есть ли строка  $S$  в заданном множестве строк. Тогда мы обычным образом «идем» по дереву со строкой  $S$ . Если просмотр закончен в листе дерева и строка  $S$  исчерпана, то ответ положительный. А если нет? Если поиск завершен во внутренней вершине дерева или на дуге? В этом случае просмотр «нижней», оставшейся части дерева дает строки, в которые  $S$  входит в качестве подстроки. Остался логически последний возможный случай — найдено несовпадение. Но и из этой ситуации извлекается информация, ибо можно указать самый длинный префикс  $S$ , который есть среди заданного множества строк в качестве подстроки.

**Задача.** Требуется найти наибольшую общую подстроку двух строк на основе данных, представленных в их обобщенном дереве суффиксов.

**Комментарий.** Внутренние вершины  $v$  обобщенного дерева должны иметь метки. Если в поддереве с корнем в вершине  $v$  есть лист, относящийся к первой строке, то значение метки равно 1, если ко второй — 2, и наконец, если он относится и к первой, и ко второй строкам, то метки 1 и 2. После этого остается найти наиболее удаленную от корня дерева вершину со значением метки последнего типа.

**Задача.** Дано  $k$  строк. Для каждого  $i$  от 2 до  $k$  требуется найти длину самой длинной подстроки, общей не менее чем для  $i$  строк. Сложность алгоритма должна иметь порядок  $O(k \cdot n)^1$ .

**Комментарий.** На рис. П2.2 показано обобщенное дерево суффиксов для строк  $S_1 = cabca\$, S_2 = bcabca@$  и  $S_3 = aabc*$ , где к каждой строке приписан некоторый конечный символ (отсутствующий в алфавите). Листья (они изображены квадратами) имеют метки в виде номера строки.



**Рис. П2.2.** Обобщенное дерево суффиксов для строк  $S_1 = cabca\$, S_2 = bcabca@, S_3 = aabc*$ . Квадратами обозначены листья с метками, равными номеру строки. Кружками — внутренние вершины с номерами, соответствующими очередности их создания при построении дерева

<sup>1)</sup> По информации из книги Д. Гасфилда (с. 165), эта задача может быть решена за время  $O(n)$ . При этом дается ссылка на следующий источник: *Hui L. Color set size problem with applications to string matching // Proc. 3rd Symp. on Combinatorial Pattern Matching Lecture Notes in Computer Science. Springer. 1992. Vol. 664. P. 227–240.*

Для каждой внутренней вершины  $v$  (их номера заданы в той очередности, в которой они образуются при создании дерева) подсчитывается значение  $cnt[v]$  — количество различных меток на листьях поддерева с корнем  $v$ . Для рассматриваемого примера  $cnt = (3, 2, 3, 2, 2, 2, 3, 3, 3)$ . Предположим, что для каждой внутренней вершины определена *строковая глубина* (количество символов в подстроке, получающейся склейкой меток дуг на пути от корня до вершины). Выпишем для наглядности строковые глубины вершин в нашем примере:  $vg = (0, 2, 1, 3, 5, 4, 3, 2, 1)$ . Осталось для каждого  $i$  от 2 до  $k$  найти такой максимальный элемент в  $vg$ , что  $cnt[v] = i$ . Например, при  $i = 2$  максимальный элемент равен 5 и соответствует вершине 5. Общая подстрока для двух строк (элементы  $cnt$  и  $vg$  выделены полужирным шрифтом) с максимальной длиной есть *cabca*, а для трех строк — *abc*. Осталось отметить, что реально для решения этой задачи массив  $vg$  не требуется, а необходим массив для хранения максимальных элементов, элементы которого формируются при обычном обходе дерева.

**Проблема.** Исследование текста на наличие кратных подстрок. Используются алгоритмы М. Крочемора, М. Мейна – Р. Лоренца и, возможно, ряд других. Экспериментально оценивается время их работы.

**Комментарий.** Среди используемых тестов должны иметься строки Фибоначчи разной длины, ибо известно, что последние имеют значительное количество кратных подстрок. В этом случае необходима разработка *генератора строк Фибоначчи*, который по заданному значению  $n$  формирует строку  $f_n$  и записывает ее в библиотеку тестов.

*Строки Фибоначчи* строятся на бинарном алфавите  $A = \{a, b\}$  и удовлетворяют рекуррентному соотношению:

$$f_0 = b, \quad f_1 = a, \quad f_n = f_{n-1}f_{n-2} \quad (n \geq 2).$$

Первые строки Фибоначчи и их длины ( $l$ ):

$$f_0 = b, \quad l = 1;$$

$$f_1 = a, \quad l = 1;$$

$$f_2 = ab, \quad l = 2;$$

$$f_3 = aba, \quad l = 3;$$

$$f_4 = abaab, \quad l = 5;$$

$$f_5 = abaababa, \quad l = 8;$$

$$f_6 = abaababaabaab, \quad l = 13;$$

$$f_7 = abaababaabaababaababa, \quad l = 21;$$

$$f_8 = abaababaabaababaabaabaababaabaab, \quad l = 34;$$

...

Нетрудно заметить, что длины строк  $f_n$  удовлетворяют классическому соотношению:  $f_n = f_{n-1} + f_{n-2} \quad (n \geq 2)$ .

**Задача.** По заданному образцу  $P$  требуется найти все его вхождения в текст  $T$ . Образец  $P$  на определенных местах содержит специальный символ (например, @), который может означать любой символ текста  $T$ . Временная сложность алгоритма —  $O(k \cdot m)$ , где  $m = |P|$ , а  $k$  — количество подстрок  $P$  без символа @.

**Комментарий.** Образец  $P = bc@a@@abc@$  входит в  $T = abcdadaabcbcaaadabcd$  два раза — со второй и одиннадцатой позиций. Разобьем  $P$  на  $P_1 = ab, P_2 = a, P_3 = abc, k = 3$ . Первый образец  $P_1$  входит в  $P$  с позиции  $l_1=1$ ; второй —  $l_2 = 4$ ; третий —  $l_3 = 7$ . Используя, например, алгоритм А. Ахо – М. Корасик (см. п. 3.5), ищем вхождения  $P_1, P_2, P_3$  в текст. Результат поиска фиксируется в некотором массиве  $cnt[1..n]$  следующим образом. Если найдено вхождение образца  $P_i$  в  $T$  с позиции  $j$ , то к элементу массива  $cnt[j - l_i + 1]$  прибавляется единица. Так, для рассматриваемого текста  $T$  массив  $cnt = (0, 3, 0, 1, 1, 0, 0, 0, 1, 1, 3, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$ . Остается найти элементы  $cnt$ , равные 3.

**Задача.** Пусть некое изображение описывается в виде матрицы  $T$ , элементами которой являются положительные целые числа, определяющие, например, цвет и яркость точек. Часть изображения описывается матрицей  $P$  меньшей размерности. Требуется найти все вхождения  $P$  в  $T$ .

**Комментарий.** Пусть  $T = \begin{pmatrix} 2 & 5 & 1 & 1 & 4 \\ 3 & 1 & 4 & 2 & 1 \\ 4 & 5 & 1 & 9 & 6 \\ 5 & 4 & 1 & 4 & 7 \\ 5 & 1 & 5 & 1 & 2 \end{pmatrix}$ , а  $P = \begin{pmatrix} 1 & 4 \\ 5 & 1 \end{pmatrix}$ .

Преобразуем  $T$  в  $T' = 25114\$31421\$45196\$54147\$51512\$$ , а  $P$  в  $P_1 = 14$  и  $P_2 = 51$  (в общем случае —  $k$  образцов). Результаты поиска  $P_1$  и  $P_2$  в  $T'$  фиксируются в матрице  $cnt$ . Если  $P_i$  входит в  $T'$  с позиции  $j$ , то преобразование  $j$  в пару  $(p, q)$ , определяющую номер строки и номер столбца  $cnt$ , очевидно: элементу  $cnt[p, q]$  присваивается значение  $i$ . Так, в нашем случае

$$cnt = \begin{pmatrix} 0 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \end{pmatrix}.$$

Остается в столбцах  $cnt$  найти числа  $1, 2, \dots, k$  (именно в такой очередности!), записанные в последовательных ячейках одного столбца.

**Проблема.** Текст  $T$  назовем *исключаемым образцом*  $P$ , если отсутствует вхождение  $P$  в  $T$ . Например, в строках Фибоначчи отсутствуют подстроки  $P = bb$ , т. е. кратная подстрока  $bb$  порядка 2.

Пусть алфавит  $A$  состоит из  $n$  символов. Требуется исследовать возможность «построения строк (возможно — и бесконечных) на алфавите из  $n$  символов, свободных от кратных подстрок порядка  $r$ , но содержащих подстроки порядков  $2, 3, \dots, r - 1$ »<sup>1)</sup>. Обозначение проблемы —  $(n, r)$ -*исключение*.

**Комментарий.** Решение этой проблемы как минимум следует начать с анализа строк А. Туе (*Axel Thue*)<sup>2)</sup>. Следует написать генераторы строк, являющихся  $(2, 3)$ -исключениями и  $(3, 2)$ -исключениями. Строки А. Туе, являющиеся  $(2, 3)$ -исключениями, на алфавите  $A = \{a, b\}$  задаются отображением:

$$a \rightarrow ab;$$

$$b \rightarrow ba.$$

Первые строки А. Туе и их длины ( $l$ ):

$$t_0 = a, l = 1;$$

$$t_1 = ab, l = 2;$$

$$t_2 = abba, l = 4;$$

$$t_3 = abbabaab, l = 8;$$

$$t_4 = abbabaabbaababba, l = 16;$$

$$t_5 = abbabaabbaababbabaababbaabbabaab, l = 32;$$

...

Если обозначить  $\bar{a} = b, \bar{b} = a$ , то для любого  $n \geq 1$ :  $t_n = t_{n-1} \bar{t}_{n-1}$ .

Строки А. Туе, являющиеся  $(3, 2)$ -исключениями, на алфавите  $A = \{a, b, c\}$  задаются отображением:

$$a \rightarrow abcab;$$

$$b \rightarrow acabcb;$$

$$c \rightarrow acbcsbc.$$

Первые строки А. Туе<sup>3)</sup>:

$$h_0 = a;$$

$$h_1 = abcab;$$

$$h_2 = abcab acabcb acbcsbc abcab acabcb;$$

$$h_3 = abcab acabcb acbcsbc abcab acabcb abcab acbcsbc abcab acabcb acbcsbc acabcb abcab acbcsbc acabcb abcab acabcb abcab acabcb$$

1) Смит Б. Методы и алгоритмы вычисления на строках. — М.: ООО «И. Д. Вильямс», 2006. С. 86.

2) Там же.

3) Знак пробела в записи  $h_2$  и  $h_3$  используется только для удобства чтения.

acbcacb abcab acabcb abcab acbcacb abcab acabcb  
acbcacb acabcb;

...

**Задача.** Известно, что деревья суффиксов являются достаточно затратной по используемой памяти структурой данных. Требуется преобразовать дерево суффиксов в ориентированный ациклический граф.

**Комментарий.** На рис. П2.3 представлено дерево суффиксов для строки  $S = abacabbac\$$ . Суффиксные связи изображены пунктирными линиями.

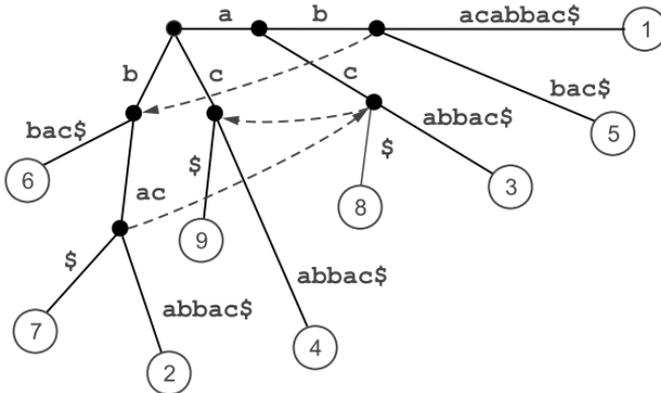


Рис. П2.3. Дерево суффиксов для строки  $S = abacabbac\$$

На рис. П2.4 показан ориентированный ациклический граф, построенный по этому дереву суффиксов. Номера меток у вершин-листьев сохранены, а в квадратных скобках указаны метки листьев дерева суффиксов, «стянутых» в эту вершину. Часть дуг, почти соответствующих суффиксным связям, имеют пустые метки.

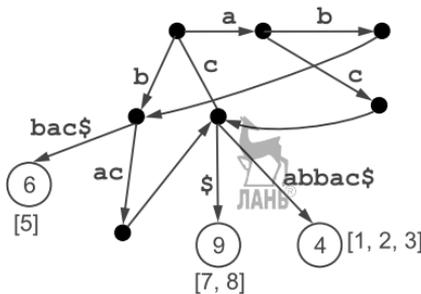


Рис. П2.4. Ориентированный ациклический граф для распознавания подстрок  $S = abacabbac$

Каждому суффиксу  $S$  соответствует путь из корня в лист в ориентированном ациклическом графе, и наоборот, любой суффикс  $S$  представлен каким-либо из таких путей.

Алгоритм подобного преобразования может быть описан так:

- Шаг 1. Построить дерево суффиксов и выписать все пары вершин  $(p, q)$ , между которыми существует суффиксная связь, если количество вершин-листьев в поддеревьях, определяемых вершинами  $p$  и  $q$ , совпадает.
- Шаг 2. Пока существуют пары  $(p, q)$  и обе вершины принадлежат текущему ациклическому графу, объединить вершины  $p$  и  $q$ .

Между результатом работы такого алгоритма и примером на рис. П2.4 есть небольшое несоответствие, но оно устранимо при работе над задачей.

**Задача.** Среди всех строк, получающихся циклическим сдвигом строки  $S$  ( $n = |S|$ ), требуется найти лексикографически наименьшую.

**Комментарий.** Естественно, что строка  $S$  построена на конечном и упорядоченном алфавите  $A$ . Пусть  $S = bacda$  ( $n = 5$ ). Ее циклические сдвиги:  $bacda, acdab, cdaba, dabac, abacd$ . Наименьшей в лексикографическом порядке является строка  $abacd$ .

Возьмем строку  $SS\$ = bacdabacda\$$  (символ  $\$$  считается лексикографически наибольшим) и построим по ней дерево суффиксов (рис. П2.5). Далее идем по дереву, выбирая в каждой вершине лексикографически наименьшую метку дуги. Обход заканчивается, если будет достигнута строковая глубина  $n$  (на рис. П2.5 это место выделено подчеркиванием). Склейка меток дуг пройденного пути дает искомую строку.

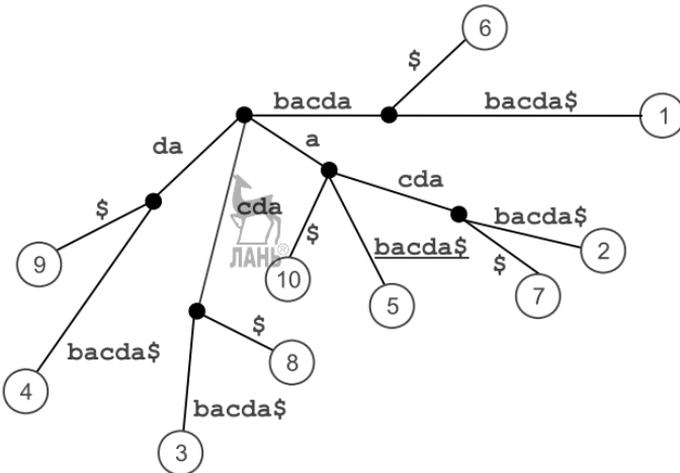
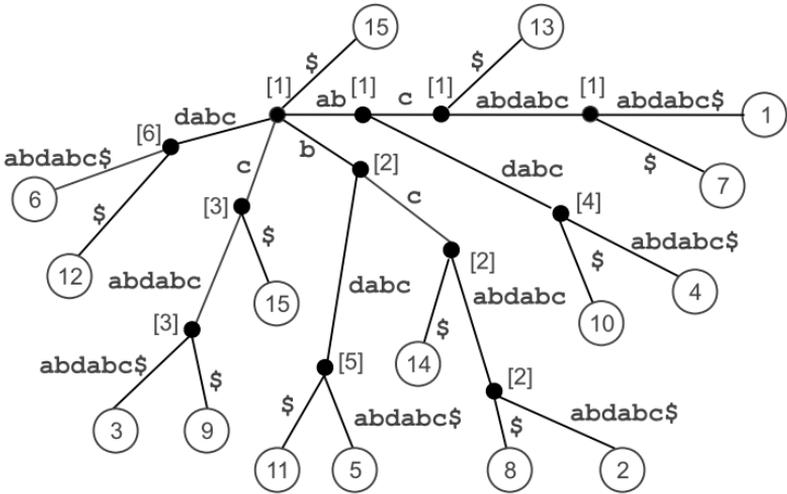


Рис. П2.5. Дерево суффиксов для строки  $bacdabacda\$$

**Задача.** Для любой позиции  $i$  строки  $S$  требуется найти в подстроке строки  $S[1..i - 1]$  начало крайнего левого вхождения ( $r_i$ ) и длину ( $l_i$ ) самого длинного префикса  $S[i..n]$ .

**Комментарий 1.** Для строки  $S = \text{abcabdabcabdabc}\$$  на рис. П2.6 представлено дерево суффиксов. В результате обхода дерева, — а он осуществляется за время  $O(n)$ , — каждой внутренней вершине  $v$  присваивается метка, равная наименьшему номеру листа в поддереве с корнем в  $v$ . Эти метки на рис. П2.6 приведены в квадратных скобках.



**Рис. П2.6.** Дерево суффиксов для строки  $S = \text{abcabdabcabdabc}\$$  (без суффиксных связей)

Пусть  $i$  равно 8, т. е. требуется найти префикс строки  $\text{bcabdabc}$ , встречающийся в  $\text{abcabda}$ . Идем по дереву суффиксов. Из корня выбрана дуга с меткой  $b$ , затем —  $c$ , а потом — дуга с меткой  $\text{abdabc}$  (рис. П2.6). Но мы должны достичь только строковой глубины 6 ( $8 - 2$ , где 2 — метка первой внутренней вершины пути) или меньшей. Просмотр заканчивается не во внутренней вершине, а на дуге. Берем метку следующей внутренней вершины — она равна 2, и это есть начало префикса ( $r_i$ ), а длина — достигнутая строковая глубина ( $l_i$ ). Для рассматриваемого примера значения  $r_i$  и  $l_i$  представлены в табл. П2.1.

Таблица П2.1

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S$	a	b	c	a	b	d	a	b	c	a	b	d	a	b	c
$l_i$	0	0	0	2	1	0	6	6	6	6	5	4	3	2	1
$r_i$	0	0	0	1	2	0	1	2	3	4	5	6	1	2	3

**Комментарий 2.** Данный алгоритм является основой метода сжатия данных Д. Зива – А. Лемпеля<sup>1)</sup>. Например, рассмотренную строку в сжатом виде можно представить как  $abc(1, 2)d(1, 6)(1, 3)$ . Первое число в круглых скобках есть  $r_i$ , а второе —  $l_i$ .

**Проблема.** В п. 3.5. приводится алгоритм А. Ахо – М. Корасик одновременного поиска нескольких образцов в тексте. Его идейной основой является алгоритм Д. Кнута – Дж. Морриса – В. Пратта. Разработка метода поиска нескольких образцов в тексте на основе синтеза идей алгоритмов Р. Бойера – Дж. Мура и А. Ахо – М. Корасик — решаемая, но не тривиальная проблема.

**Комментарий.** Данная проблема решена Б. Комменц-Уолтер<sup>2)</sup>, но от самостоятельного поиска получение интеллектуального удовлетворения гарантировано. Кроме того, ваша работа может завершиться созданием более простого метода.

**Задача.** Дано полное двоичное дерево (каждая вершина дерева имеет двух сыновей). Для каждой пары вершин  $x$  и  $y$  требуется найти вершину  $z$ , являющуюся их наименьшим общим предшественником.

**Комментарий.** На рис. П2.7 показано полное двоичное дерево из 15 вершин. Номера вершин (рядом с их десятичным представлением приводится и двоичная запись) указаны в порядке обхода дерева в глубину или симметричного обхода — корень нумеруется после присвоения номеров вершинам левого поддерева, но перед нумерацией вершин правого поддерева<sup>3)</sup>.

Данной нумерации вершин можно дать и другую интерпретацию. Для дерева с  $p$  листьями запись номера вершины требует  $d + 1$  битов, где  $d = \log_2 p$ .

<sup>1)</sup> Ziv J., Lempel A. A universal algorithm for sequential data compression // IEEE Trans. on Info. Theory. 1977. Vol. 23. P. 337–343; Ziv J., Lempel A. Compression of individual sequences via variable length coding // IEEE Trans. on Info. Theory. 1978. Vol. 24. P. 530–536; Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 208–212.

<sup>2)</sup> Commentz-Walter B. A string matching algorithm fast on average // Proc. of the 6th Int. Colloq. on Automata, Languages and Programming. 1979. P. 118–132; Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 200–208; Смит Б. Методы и алгоритмы вычисления на строках: пер. с англ. — М.: ООО «И. Д. Вильямс», 2006. С. 364–366 (правда, здесь алгоритм называется «алгоритмом Комменц-Вальтера», но это — специфика перевода различными людьми).

<sup>3)</sup> Окулов С. М. Развитие интеллекта школьника. Абстрактные типы данных. — М.: БИНОМ. Лаборатория знаний, 2009. С. 96.

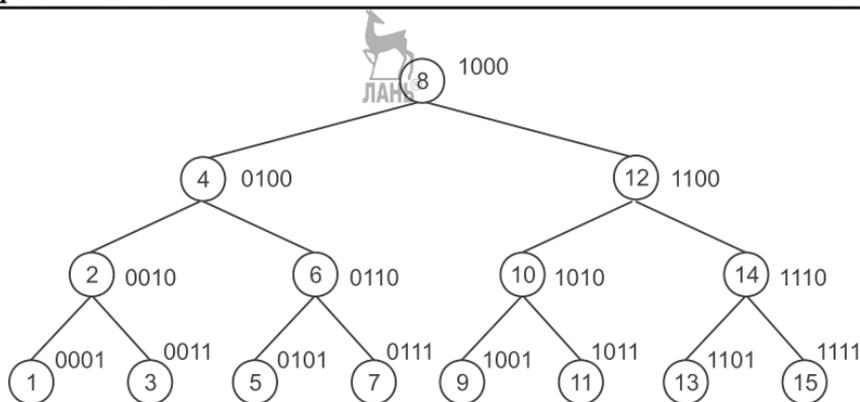


Рис. П2.7. Полное двоичное дерево из 15 вершин

Рассмотрим путь от корня к какой-либо вершине, например 5. Путь влево обозначим как 0, а вправо — как 1. До пятой вершины мы идем налево (0), затем направо (1) и, наконец, опять налево (0). Получили метку пути 0101. Припишем к ней единицу — имеем 01011.

Еще пример — от корня до вершины 10. Метка пути имеет вид 10. Приписываем единицу и оставшиеся до  $(d + 1)$ -го бита нули, — получаем 1010. Корень дерева всегда имеет метку в виде единицы на первом месте (приписанную) и оставшиеся нули.

Итак, к полученным в результате прохода по ребрам дерева битам мы приписываем единицу и дополняем нулями до требуемого количества битов.

Вернемся к поставленной задаче. Необходимо найти наименьшего общего предшественника вершин  $x$  и  $y$ . Исключим пока случай, когда одна из вершин предшествует другой. Пусть  $x = 9$  (1001), а  $y = 13$  (1101). Выполним операцию  $\text{Xor}$  над двоичными номерами:  $1001 \text{ Xor } 1101 = 0100$ . До первой единицы слева (точнее, номера ее позиции  $k$ ) результаты операции  $\text{Xor}$  пути до вершин  $x$  и  $y$  из корня совпадают, т. е. первые  $k - 1$  ребер одинаковы. Путь по этим ребрам из корня как раз и дает вершину, являющуюся наименьшим общим предшественником вершин  $x$  и  $y$ . Другими словами, номер этой вершины состоит из  $k - 1$  бита  $x$  или  $y$ , затем — единицы и  $d + 1 - k$  нулей. Для его получения следует выполнить операцию  $\text{Xor}$ , затем найти  $k$ , сдвинуть  $x$  (или  $y$ ) вправо на  $d + 1 - k$ , положить крайний правый бит равным единице и сдвинуть результат влево на  $d + 1 - k$  позиций. Все эти операции выполняются за константное время и, таким образом, наименьший общий предшественник вершин в полном двоичном дереве находится за постоянное время.

Остался нерассмотренным случай, когда  $x$  предшествует  $y$  или  $y$  предшествует  $x$ . Выполним операцию  $\text{Xor}$  и найдем значение  $k$ . Вершина  $x$  предшествует вершине  $y$  тогда и только тогда,

когда значение  $k$  строго больше, чем количество ребер в пути от корня до вершины  $x$ .

**Проблема.** Требуется найти метод предварительной обработки дерева суффиксов с линейной временной оценкой — такой, что для получения ответа на запрос о наименьшем общем предшественнике любых двух вершин потребуется константное время. Другими словами, когда за константное время (при построенном и предварительно обработанном дереве суффиксов за общее время  $O(n)$ ) определяется наибольший общий префикс двух произвольных суффиксов строки.

**Комментарий.** Если найти отображение дерева суффиксов в полное двоичное дерево, выполняемое за линейное время и сохраняющее отношение наследования в дереве суффиксов, то проблема решена<sup>1)</sup>.

**Задача.** Даны две строки  $S_1$  ( $n = |S_1|$ ) и  $S_2$  ( $m = |S_2|$ ). Для каждой пары значений  $(i, j)$  требуется найти длину наибольшего общего префикса суффиксов  $S_1[i..n]$  и  $S_2[j..m]$ .

**Комментарий.** Следует построить обобщенное дерево суффиксов для строк и выполнить его предварительную обработку так, как это указано в описании к предыдущей проблеме. После этого задача сводится к нахождению строковой глубины наименьшего общего предшественника двух вершин  $(i, j)$  дерева.

**Проблема.** Анализ текстов на русском языке.

По определению К. Шеннона, энтропию источника сообщения можно определить как меру содержащегося в нем объема информации. Пусть у нас есть язык  $A$ , состоящий из  $n$  разных символов  $A = \{a_1, a_2, \dots, a_n\}$ . Предположим, что источник  $S$  может генерировать эти символы с независимыми вероятностями  $p[a_1], p[a_2], \dots, p[a_n]$ , причем последние удовлетворяют условию:  $\sum_{i=1}^n p[a_i] = 1$ . Тогда энтропия источ-

ника  $S$  равна  $H(S) = \sum_{i=1}^n p[a_i] \cdot \log_2 \left( \frac{1}{p[a_i]} \right)$ . Ее обычно опреде-

<sup>1)</sup> Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. С. 227–244; Harel D., Tarjan R. E. Fast algorithms for finding nearest common ancestors // SIAM J. Comput. 1984. Vol. 13. P. 338–355; Schieber B., Vishkin U. On finding lowest common ancestors: simplifications and parallelization // SIAM J. Comput. 1988. Vol. 17. P. 1253–1262.

ляют как среднее количество битов в сообщении, порожденном в источнике. Предположим, что мы имеем дело с текстами на русском языке и алфавит состоит из 32 символов («е» и «ё» не различаем).

**Задача 1.** Требуется произвести экспериментальные оценки значений  $p[a_i]$  по текстам на русском языке и подсчитать величину  $H(S(A)) = \sum_{i=1}^n p[a_i] \cdot \log_2 \left( \frac{1}{p[a_i]} \right)$ .

**Комментарий.** Следует взять  $t$  независимых текстов. По каждому тексту  $j$  (эксперимент) подсчитываются (для всех символов  $a_i$ ) величины  $p_j[a_i] = \frac{\text{количество символов } a_i}{\text{длина текста}}$  и находят-

ся значения  $p[a_i] = \frac{\sum_{j=1}^t p_j[a_i]}{t}$ .

Величину  $r(|A|) = \log_2 |A|$  называют *абсолютной энтропией языка* с алфавитом  $A$ . Для русского языка эта величина равна  $\log_2 32 = 5$  битов на букву (для английского языка  $\log_2 26 \approx 4,7$  битов на букву).

**Задача 2.** Предположим, что в среднем каждое слово на русском языке состоит из  $k$  символов. Требуется выполнить экспериментальную оценку величины  $k$  по текстам на русском языке.

Считаем, что источник  $S$  порождает слова в виде последовательности, состоящей из  $k$  символов. Пусть  $A_k$  обозначает минимальное ожидаемое количество битов, требуемых для записи слова из  $k$  символов.

**Теорема К. Шеннона** формулируется в виде утверждения:  $\lim_{k \rightarrow \infty} \frac{A_k}{k} = H(S)$ . Другими словами, минимальное среднее количество битов, необходимое для записи слова, равно  $H(S)$ . Величину  $r = \frac{H(S(A))}{k}$  называют *энтропией языка* с алфавитом  $A$ . В свое время К. Шеннон показал, что энтропия английского языка лежит в интервале от 1,0 до 1,5 битов на букву<sup>1)</sup>.

<sup>1)</sup> Мао В. Современная криптография: теория и практика. — М.: Издательский дом «Вильямс», 2005. С. 114.

**Задача 3.** Требуется выполнить экспериментальную оценку энтропии русского языка.

*Избыточность* языка с алфавитом  $A$  называют величину  $r(A) - r$ . Так, для английского языка она равна  $4,7 - 1,5 = 3,2$  битов на букву или, в процентном выражении,  $3,2/4,7 \approx 68\%$ . Другими словами, английский текст без потери информации можно сократить на  $32\%$ .

**Задача 4.** Требуется экспериментально определить избыточность русского языка.

Известно, что в естественных языках некоторые буквосочетания встречаются чаще, чем другие. Буквосочетания из двух букв называют *биграммami*, а из трех — *триграммами*. Знание этих особенностей языка играет значительную роль в методах *криптоанализа* (вскрытия зашифрованных данных). Если количество биграмм в русском языке равно 992, то количество триграмм — 29760.

**Задача 5.** Требуется экспериментальным путем выявить наиболее часто используемые биграммы и триграммы в русском языке.

*Примечания.* 1. Для этого нужно иметь тексты из различных областей знания. 2. Решение путем прямого подсчета имеет «право на жизнь», но ваш уровень развития позволяет решить задачу и более достойными методами. ☺

Каждый из нас обладает определенным словарным запасом как в устной, так и в письменной речи (вариант «людоедки Элочки»<sup>1)</sup> или бездуховных манкуртов здесь не рассматривается. Образованный человек, читая стихи или роман и не зная автора, по особенностям языка с достаточно большой вероятностью может сказать, кем этот текст написан. Например, отличие поэзии В. Маяковского от творчества Е. Есенина очевидно для большинства.

*Конкретизация проблемы* (а точнее, проблема на долгие годы ☺)<sup>2)</sup>. Требуется разработать анализатор текстов русского языка. Другими словами, если на вход такой програм-

<sup>1)</sup> Героиня романа И. Ильфа и Е. Петрова «Двенадцать стульев», которая ограничивалась в общении только тридцатью словами.

<sup>2)</sup> Учеными, исследователями, да и просто умными людьми становятся не только в результате чтения книг и накопления в памяти фактографических данных. Скорее всего в этом случае как раз и не становятся! «Многознание уму не научает», — это говорил, кажется, еще Гераклит. Пусть несколько преувеличенно, но это так, — профессионалы «вырастают» в результате решения проблем, а иногда (и не очень редко) — в результате решения одной проблемы. Но она (проблема) должна обязательно вас «взять за живое», она должна быть *вашей!*

мы подается некий текст, то на выходе, например, должен быть указан автор этого текста.

*Примечания.* 1. В самом простом случае требуется различать тексты двух авторов. 2. При решении этой проблемы многое из того, что изучено с помощью этой книги, будет востребовано, но данного раздела *computer science* будет явно недостаточно. 3. Результат вашей работы, доведенный до «промышленного» варианта, будет востребован очень многими. 4. Автор не может дать развернутых рекомендаций по решению этой проблемы. ☺

**Проблема.** Требуется разработать программный синтезатор осмысленных текстов на заданную тему с точки зрения русского языка.

**Комментарий.** См. примечания к предыдущей конкретизации. Единственное дополнение: если вы научите компьютер по заданному сюжету «выдавать» детективы современного типа, то, вероятно, станете очень обеспеченным человеком. ☺

**Проблема.** Требуется разработать анализатор сообщений, передаваемых по электронной почте на определенный адрес.

*Конкретизация проблемы.* С электронной почтой в настоящее время не работает разве только очень ленивый. Однако многих раздражает обилие спама. Любое письмо имеет определенный формат и так или иначе представлено в виде битовой последовательности. Предположим, что в некотором хранилище (базе данных) есть «запрещенные» двоичные подпоследовательности: при их наличии в тексте сообщения последнее отбрасывается. Естественно, что должен быть обеспечен и инструмент для дополнения и изменения данных в хранилище.

**Комментарий.** На возражение о том, что такие программы уже есть и они называются таким-то «мудрым» английским словом, мы отвечать не будем, как и не будем использовать самих этих «мудрых» английских слов. Ваше решение может быть и лучше, и качественнее, — но самое главное — оно будет вашим! ☺

**Проблема.** Требуется разработать анализатор трафика электронной почты с целью выявления опасных сообщений.

*Конкретизация проблемы.* Известно, что электронную почту используют и не очень хорошие люди, например террористы. Предположим, что их сообщения можно охарактеризовать неким множеством двоичных слов, которые уже выявлены. Их наличие (в любом сочетании) в сообщении го-

ворит о том, что оно передается «нехорошим человеком» и является опасным.

**Комментарий.** См. комментарий к предыдущей конкретизации.

**Проблема.** Требуется написать программу для тематической классификации текстов.

*Конкретизация проблемы.* Дан массив текстов нескольких различных тематик (например, из области математики, физики, астрономии и т. д.). Требуется написать программу, которая на этом массиве обучается распознавать тематику текста. После процесса обучения программе предъявляется новый текст, которого не было в обучающем массиве. На выходе должна появиться тема нового текста или заключение о неизвестной теме.

**Комментарий.** Один из подходов к данной проблеме заключается в нахождении в обучающем массиве текстов всех терминов — слов или словосочетаний, значимых для определения тематики. Каждому термину присваивается свой *вес* — число, характеризующее значимость этого термина. Вес в простейшем случае совпадает с частотой вхождения термина в текст. Более сложный способ — присваивать больший вес терминам, которые часто встречаются в текстах определенной тематики и редко — в остальных документах, и наоборот, меньший вес давать общеупотребительным словам, которые часто встречаются во всех документах обучающего массива.

Таким образом, задача сводится к поиску в данном тексте всех словосочетаний длиной  $n$  ( $n = 1 \dots N$ ), вес которых превышает заданный порог. Когда все такие словосочетания будут определены, процесс обучения считается законченным, и на вход программы поступает новый текст, тематику которого нужно определить.

Заметим, что программа будет работать устойчивее, если обработку осуществлять после *морфологического разбора*, т. е. приведения всех слов к нормальной форме: *дома*, *домой*, *о доме*  $\equiv$  дом.

**Проблема.** Требуется составить программу для автоматического реферирования текста.

*Конкретизация проблемы.* На вход программы поступает текст длины  $N$ . Требуется на выходе получить текст длины  $n$  (*реферат*), адекватно отражающий содержание исходного текста; при этом  $n \ll N$ .

**Комментарий.** Существует несколько подходов к решению данной проблемы. В одном из них используется принцип отбора в реферат наиболее информативных предложений исходного текста, имеющих наибольший *вес* (см. задачу тематической классифика-



ции). Другой подход заключается в подстановке наиболее значимых слов и словосочетаний в заранее заданные фразы-клише, такие как «В статье рассматривается тема...», «В тексте указывается, что...».

Одной из важных характеристик подобной программы является возможность настройки длины реферата *n*.

**Проблема.** Требуется разработать программу машинного перевода с какого-либо иностранного языка на русский и обратно.

**Конкретизация проблемы.** Дан текст на иностранном языке. Программа должна (с использованием словаря) составить текст на русском языке, который является переводом иностранного текста.

**Комментарий.** Задача может решаться «в лоб», когда каждому иностранному (например, английскому) слову соответствует русское, а программа просто заменяет английские слова на соответствующие русские из словаря.

Следующим шагом может быть аналогичная замена не отдельных слов, а устойчивых словосочетаний, фразеологизмов. При этом, чтобы избежать слишком стремительного роста объема такого словаря, можно ограничиться определенной предметной областью (например, программированием).

Дальнейшие усовершенствования ведут нас в область синтаксического разбора предложения и выявления смысла текста (в область *семантики*). В ходе решения такой задачи возникает проблема *морфологического анализа* — определения и подстановки нужной формы слова (приставок, суффиксов, окончаний), например *выйти из дома* — *идти домой*. Можно подумать над собственным решением либо воспользоваться специальными программами морфологического анализа.





*Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"*

*Учебное электронное издание*

Серия: «Развитие интеллекта школьников»

**Окулов Станислав Михайлович**

## **АЛГОРИТМЫ ОБРАБОТКИ СТРОК**

Ведущий редактор *Д. Ю. Усенков*

Художник *Н. А. Новак*

Технический редактор *Е. В. Денюкова*

Корректор *Л. Н. Макарова*

Компьютерная верстка: *В. А. Носенко*

Подписано к использованию 23.09.19.

Формат 125×200 мм

Издательство «Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3

Телефон: (499) 157-5272

e-mail: [info@pilotLZ.ru](mailto:info@pilotLZ.ru), <http://www.pilotLZ.ru>

---

## РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

На материале задачи поиска подстроки в строке, решению которой посвящены работы многих профессионалов за последние 20–30 лет, показано, как построить занятия по информатике, чтобы побудить школьника к творчеству, развить у него вкус к решению исследовательских проблем.

Для школьников, преподавателей информатики, а также для студентов, выбравших информатику в качестве основной специальности. Книга может быть использована как в обычных школах при проведении факультативных занятий, так и в образовательных учреждениях с углубленным изучением информатики и математики.



### **ОКУЛОВ Станислав Михайлович**

Декан факультета информатики Вятского государственного гуманитарного университета, кандидат технических наук, доктор педагогических наук, профессор, с 2001 г. почетный работник высшего профессионального образования РФ. Автор 9 изобретений по элементам ассоциативных вычислительных структур и автор (соавтор) 15 книг по информатике для школьников и студентов.

Область интересов: развитие интеллектуальных способностей школьника при активном изучении информатики, методика преподавания информатики в школе и вузе.

С 1993 по 2003 год деятельность в вузе совмещал с работой учителя информатики. За это время его ученики отмечены 33 дипломами (1-й и 2-й степени) на российских олимпиадах школьников по информатике; трое из них представляли Россию на международных олимпиадах.