А. ШЕНЬ

ПРОГРАММИРОВАНИЕ
теоремы и задачи

Издание пятое, стереотипное

Москва
Издательство МЦНМО
2014

УДК 519.671
ББК 22.18
Ш47

Шень А.

Ш47 Программирование: теоремы и задачи. — 5-е изд., стереотип. — М.: МЦНМО, 2014. — 296 с.: ил.

ISBN 978-5-4439-0114-5

Книга содержит задачи по программированию различной трудности. Большинство задач приводятся с решениями. Цель книги — научить основным методам построения корректных и быстрых алгоритмов.

Для учителей информатики, старшеклассников, студентов младших курсов высших учебных заведений. Пособие может быть использовано на кружковых и факультативных занятиях в общеобразовательных учреждениях, в школах с углублённым изучением математики и информатики, а также в иных целях, не противоречащих законодательству РФ.

Предыдущее издание книги вышло в 2011 г.

ББК 22.18

ISBN 978-5-4439-0114-5

Несколько замечаний вместо предисловия

Книга написана по материалам занятий программированием со школьниками математических классов школы № 57 г. Москвы и студентами младших курсов (Московский государственный университет, Независимый московский университет, университет г. Uppsala, Швеция).

Книга написана в убеждении, что программирование имеет свой предмет, не сводящийся ни к конкретным языкам и системам, ни к методам построения быстрых алгоритмов.

Кто-то однажды сказал, что можно убедить в правильности алгоритма, но не в правильности программы. Одна из целей книги — попытаться продемонстрировать, что это не так.

В принципе, возможность практического исполнения программ не является непременным условием изучения программирования. Однако она является сильнейшим стимулом — без такого стимула вряд ли у кого хватит интереса и терпения.

Выбранный жанр книги по необходимости ограничивает её «программированием в малом», оставляя в стороне необходимую часть программистского образования — работу по модификации больших программ. Автор продолжает мечтать о наборе учебных программных систем эталонного качества, доступных для модификации школьниками.

Кажется, Хоар сказал, что эстетическая прелесть программы — это не архитектурное излишество, а то, что отличает в программировании успех от неудачи. Если, решая задачи из этой книги, читатель почувствует прелесть хорошо написанной программы, в которой «ни убавить, ни прибавить», и сомнения в правильности которой кажутся нелепыми, то автор будет считать свою цель достигнутой.

Характер глав различен: в одних предлагается набор мало связанных друг с другом задач с решениями, в других по существу излагается один-единственный алгоритм. Темы глав во многом пересекаются, и мы предпочли кое-какие повторения формальным ссылкам.

Уровень трудности задач и глав весьма различен. Мы старались включить как простые задачи, которые могут быть полезны для начинающих, так и трудные задачи, которые могут посадить в лужу сильного школьника. (Хоть и редко, но это бывает полезно.)

В качестве языка для записи программ был выбран паскаль. Он достаточно прост и естествен, имеет неплохие реализации (например, старые компиляторы Turbo Pascal фирмы Borland были выложены для бесплатного скачивания) и позволяет записать решения всех рассматриваемых задач. Возможно, Модула-2 или Оберон были бы более изящным выбором, но они менее доступны.

Практически все задачи и алгоритмы, разумеется, не являются новыми. (В некоторых редких случаях приведены ссылки на конкретную книгу или конкретного человека. См. также список книг для дальнейшего чтения.) Вместе с тем мы надеемся, что в некоторых случаях алгоритмы (и особенно доказательства) изложены более коротко и отчётливо.

Это не только и не столько учебник для школьника, сколько справочник и задачник для преподавателя, готовящегося к занятию.

 Адреса автора: `shen@mccme.ru`, `shen@landau.ac.ru`, `sasha.shen@gmail.com`, `alexander.shen@lif.univ-mrs.fr`.

При подготовке текста использовалась (свободно распространяемая) версия LaTeX*a, включающая стилевые файлы, составленные С. М. Львовским (см.* `ftp://ftp.mccme.ru/pub/tex/`*).*

Я рад случаю поблагодарить всех, с кем имел честь сотрудничать, преподавая программирование, особенно тех, кто был «по другую сторону баррикады», а также всех приславших мне замечания и исправления (специальная благодарность — Ю. В. Матиясевичу). Автор благодарит В. Шувалова за хлопоты по вёрстке, а также издательство МЦНМО за то, что оно ограничилось дизайном обложки (немного странным) и не вмешивалось в процесс подготовки оригинал-макета (включая текст на обложке). Благодарю также Институт проблем передачи информации РАН, Американское математическое общество (фонд помощи бывшему СССР), фонд Сороса, университет г. Бордо, фонд «Культурная инициатива», фонды STINT (Швеция), CNRS (Франция), Ecole Normale (Лион, Франция), LIF (Марсель, Франция), университет г. Уппсала (Швеция), Российский фонд фундаментальных исследований (гранты 02-01-22001 НЦНИа, 03-01-00475 и другие), а также Совет поддержки научных школ при Президенте РФ (грант НШ-358.2003.1) за поддержку. Вместе с тем содержание книги отражает точку зрения автора, за ошибки которого указанные организации и лица ответственности не несут (и наоборот).

Содержание

1. ПЕРЕМЕННЫЕ, ВЫРАЖЕНИЯ, ПРИСВАИВАНИЯ

1.1. Задачи без массивов

1.1.1. Даны две целые переменные a, b. Составить фрагмент программы, после исполнения которого значения переменных поменялись бы местами (новое значение a равно старому значению b и наоборот).

Решение. Введём дополнительную целую переменную t.

```
t := a;
a := b;
b := t;
```
□

Попытка обойтись без дополнительной переменной, написав

```
a := b;
b := a;
```

не приводит к цели (безвозвратно утрачивается начальное значение переменной a).

1.1.2. Решить предыдущую задачу, не используя дополнительных переменных (и предполагая, что значениями целых переменных могут быть произвольные целые числа).

Решение. Начальные значения a и b обозначим a0, b0.

```
a := a + b; {a = a0 +  b0, b = b0}
b := a - b; {a = a0 + b0, b = a0}
a := a - b; {a = b0, b = a0}
```
□

1.1.3. Дано целое число a и натуральное (целое неотрицательное) число n. Вычислить a^n. Другими словами, необходимо составить программу, при исполнении которой значения переменных a и n не меняются, а значение некоторой другой переменной (например, b) становится равным a^n. (При этом разрешается использовать и другие переменные.)

Решение. Введём целую переменную k, которая меняется от 0 до n, причём поддерживается такое свойство: $b = a^k$).

```
k := 0; b := 1;
{b = a в степени k}
while k <> n do begin
| k := k + 1;
| b := b * a;
end;
```
□

Другое решение той же задачи:

```
k := n; b := 1;
{a в степени n = b * (a в степени k)}
while k <> 0 do begin
| k := k - 1;
| b := b * a;
end;
```

1.1.4. Решить предыдущую задачу, если требуется, чтобы число действий (выполняемых операторов присваивания) было порядка $\log n$ (то есть не превосходило бы $C \log n$ для некоторой константы C; $\log n$ — это степень, в которую нужно возвести 2, чтобы получить n).

Решение. Внесём некоторые изменения во второе из предложенных решений предыдущей задачи:

```
k := n; b := 1; c:=a;
{a в степени n = b * (c в степени k)}
while k <> 0 do begin
| if k mod 2 = 0 then begin
| | k:= k div 2;
| | c:= c*c;
| end else begin
| | k := k - 1;
| | b := b * c;
| end;
end;
```

Каждый второй раз (не реже) будет выполняться первый вариант оператора выбора (если k нечётно, то после вычитания единицы становится чётным), так что за два цикла величина k уменьшается по крайней мере вдвое. □

1.1.5. Даны натуральные числа a, b. Вычислить произведение $a \cdot b$, используя в программе лишь операции +, -, =, <>.

Решение.

```
k := 0; c := 0;
{инвариант: c = a * k}
while k <> b do begin
| k := k + 1;
| c := c + a;
end;
{c = a * k и k = b, следовательно, c = a * b}
```
□

1.1.6. Даны натуральные числа a и b. Вычислить их сумму $a + b$. Использовать операторы присваивания лишь вида

$$\langle\text{переменная1}\rangle := \langle\text{переменная2}\rangle,$$
$$\langle\text{переменная}\rangle := \langle\text{число}\rangle,$$
$$\langle\text{переменная1}\rangle := \langle\text{переменная2}\rangle + 1.$$

Решение.

```
 ...
{инвариант: c = a + k}
 ...
```
□

1.1.7. Дано натуральное (целое неотрицательное) число a и целое положительное число d. Вычислить частное q и остаток r при делении a на d, не используя операций div и mod.

Решение. Согласно определению, $a = q \cdot d + r$, $0 \leqslant r < d$.

```
{a >= 0; d > 0}
r := a; q := 0;
{инвариант: a = q * d + r, 0 <= r}
while not (r < d) do begin
| {r >= d}
| r := r - d; {r >= 0}
| q := q + 1;
end;
```
□

1.1.8. Дано натуральное n, вычислить n! ($0! = 1$, $n! = n \cdot (n-1)!$). □

1.1.9. Последовательность Фибоначчи определяется так: $a_0 = 0$, $a_1 = 1$, $a_k = a_{k-1} + a_{k-2}$ при $k \geqslant 2$. Дано n, вычислить a_n. □

1.1.10. Та же задача, если требуется, чтобы число операций было пропорционально $\log n$. (Переменные должны быть целочисленными.)

[Указание. Пара соседних чисел Фибоначчи получается из предыдущей умножением на матрицу

$$\begin{Vmatrix} 1 & 1 \\ 1 & 0 \end{Vmatrix}$$

— так что задача сводится к возведению матрицы в степень n. Это можно сделать за $C \log n$ действий тем же способом, что и для чисел.] □

1.1.11. Дано натуральное n, вычислить

$$\frac{1}{0!} + \frac{1}{1!} + \ldots + \frac{1}{n!}.$$ □

1.1.12. То же, если требуется, чтобы количество операций (выполненных команд присваивания) было бы порядка n (не более Cn для некоторой константы C).

Решение. Инвариант: sum $= 1/1! + \ldots + 1/k!$, last $= 1/k!$ (важно не вычислять заново каждый раз k!). □

1.1.13. Даны два натуральных числа a и b, не равные нулю одновременно. Вычислить НОД(a,b) — наибольший общий делитель a и b.

Решение. Вариант 1.

```
if a < b then begin
| k := a;
end else begin
| k := b;
end;
{k = min (a,b)}
{инвариант: никакое  число, большее k, не является
  общим делителем (оно больше одного из чисел a,b)}
while not ((a mod k = 0) and (b mod k = 0)) do begin
| k := k - 1;
end;
{k - общий делитель, большие - нет}
```

Вариант 2 (алгоритм Евклида). Будем считать, что НОД(0,0)=0. Тогда НОД(a,b) = НОД(a-b,b) = НОД(a,b-a); НОД(a,0) = НОД(0,a) = a для всех a, b ⩾ 0.

```
 m := a; n := b;
{инвариант: НОД (a,b) = НОД (m,n); m,n >= 0 }
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n;
| end else begin
| | n := n - m;
| end;
end;
{m = 0 или n = 0}
if m = 0 then begin
| k := n;
end else begin {n = 0}
| k := m;
end;
```
□

1.1.14. Написать модифицированный вариант алгоритма Евклида, использующий соотношения НОД(a,b) = НОД(a mod b, b) при a ⩾ b, НОД(a,b) = НОД(a, b mod a) при b ⩾ a. □

1.1.15. Даны натуральные a и b, не равные 0 одновременно. Найти d = НОД(a,b) и такие целые x и y, что d = a · x + b · y.

Решение. Добавим в алгоритм Евклида переменные p, q, r, s и впишем в инвариант условия m = p*a+q*b; n = r*a+s*b.

```
m:=a; n:=b; p := 1; q := 0; r := 0; s := 1;
{инвариант: НОД (a,b) = НОД (m,n); m,n >= 0
            m = p*a + q*b; n = r*a + s*b.}
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n; p := p - r; q := q - s;
| end else begin
| | n := n - m; r := r - p; s := s - q;
| end;
end;
if m = 0 then begin
| k :=n; x := r; y := s;
end else begin
| k := m; x := p; y := q;
end;
```
□

1.1.16. Решить предыдущую задачу, используя в алгоритме Евклида деление с остатком. □

1.1.17. (Э. Дейкстра) Добавим в алгоритм Евклида дополнительные переменные u, v, z:

```
 m := a; n := b; u := b; v := a;
{инвариант: НОД (a,b) = НОД (m,n); m,n >= 0 }
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n; v := v + u;
| end else begin
| | n := n - m; u := u + v;
| end;
end;
if m = 0 then begin
| z:= v;
end else begin {n=0}
| z:= u;
end;
```

Доказать, что после исполнения алгоритма значение z равно удвоенному наименьшему общему кратному чисел a, b: $z = 2 \cdot НОК(a,b)$.

Решение. Заметим, что величина $m \cdot u + n \cdot v$ не меняется в ходе выполнения алгоритма. Остаётся воспользоваться тем, что вначале она равна 2ab и что $НОД(a,b) \cdot НОК(a,b) = ab$. □

1.1.18. Написать вариант алгоритма Евклида, использующий соотношения

$$НОД(2a, 2b) = 2 \cdot НОД(a, b),$$
$$НОД(2a, b) = НОД(a, b) \quad \text{при нечётном } b,$$

не включающий деления с остатком, а использующий лишь деление на 2 и проверку чётности. (Число действий должно быть порядка $\log k$ для исходных данных, не превосходящих k.)

Решение.

```
m:= a; n:=b; d:=1;
{НОД(a,b) = d * НОД(m,n)}
while not ((m=0) or (n=0)) do begin
| if (m mod 2 = 0) and (n mod 2 = 0) then begin
| | d:= d*2; m:= m div 2; n:= n div 2;
| end else if (m mod 2 = 0) and (n mod 2 = 1) then begin
```

```
    | | m:= m div 2;
    | end else if(m mod 2 = 1) and (n mod 2 = 0) then begin
    | | n:= n div 2;
    | end else if (m mod 2=1) and (n mod 2=1) and (m>=n) then begin
    | | m:= m-n;
    | end else if (m mod 2=1) and (n mod 2=1) and (m<=n) then begin
    | | n:= n-m;
    | end;
end;
{m=0 => ответ=d*n; n=0 => ответ=d*m}
```

Оценка числа действий: каждое второе действие делит хотя бы одно из чисел m и n пополам. □

1.1.19. Дополнить алгоритм предыдущей задачи поиском x и y, для которых $\mathtt{ax}+\mathtt{by}=\mathtt{НОД}(\mathtt{a},\mathtt{b})$.

Решение. (Идея сообщена Д. Звонкиным.) Прежде всего заметим, что одновременное деление a и b пополам не меняет искомых x и y. Поэтому можно считать, что с самого начала одно из чисел a и b нечётно. (Это свойство будет сохраняться и далее.)

Теперь попытаемся, как и раньше, хранить такие числа $\mathtt{p},\mathtt{q},\mathtt{r},\mathtt{s}$, что

$$\mathtt{m}=\mathtt{ap}+\mathtt{bq},$$
$$\mathtt{n}=\mathtt{ar}+\mathtt{bs}.$$

Проблема в том, что при делении, скажем, m на 2 надо разделить p и q на 2, и они перестанут быть целыми (а станут двоично-рациональными). Двоично-рациональное число естественно хранить в виде пары ⟨числитель, показатель степени двойки в знаменателе⟩. В итоге мы получаем d в виде комбинации a и b с двоично-рациональными коэффициентами. Иными словами, мы имеем

$$2^{\mathtt{i}}\mathtt{d}=\mathtt{ax}+\mathtt{by}$$

для некоторых целых $\mathtt{x},\mathtt{y}$ и натурального i. Что делать, если $\mathtt{i}>1$? Если x и y чётны, то на 2 можно сократить. Если это не так, положение можно исправить преобразованием

$$\mathtt{x}:=\mathtt{x}+\mathtt{b},$$
$$\mathtt{y}:=\mathtt{y}-\mathtt{a}$$

(оно не меняет $\mathtt{ax}+\mathtt{by}$). Убедимся в этом. Напомним, что мы считаем, что одно из чисел a и b нечётно. Пусть это будет a. Если при этом

y чётно, то и x должно быть чётным (иначе ax + by будет нечётным). А при нечётном y вычитание из него нечётного a делает y чётным. □

1.1.20. Составить программу, печатающую квадраты всех натуральных чисел от 0 до заданного натурального n.

Решение.

```
k:=0;
writeln (k*k);
{инвариант: k<=n, напечатаны все
  квадраты до k включительно}
while not (k=n) do begin
| k:=k+1;
| writeln (k*k);
end;
```
□

1.1.21. Та же задача, но разрешается использовать из арифметических операций лишь сложение и вычитание, причём общее число действий должно быть порядка n.

Решение. Введём переменную k_square (square — квадрат), связанную с k соотношением $k\_square = k^2$:

```
k := 0; k_square := 0;
writeln (k_square);
while not (k = n) do begin
| k := k + 1;
| {k_square = (k-1) * (k-1) = k*k - 2*k + 1}
| k_square := k_square + k + k - 1;
| writeln (k_square);
end;
```
□

Замечание. Можно обойтись без вычитания с помощью такой хитрости:

```
while not (k = n) do begin
| k_square := k_square + k;
| {k_square = k*k + k}
| k := k + 1;
| {k_square = (k-1)*(k-1)+(k-1)=k*k-k}
| k_square := k_square + k;
end;
```

1.1.22. Составить программу, печатающую разложение на простые множители заданного натурального числа $n > 0$ (другими словами, требуется печатать только простые числа и произведение напечатанных чисел должно быть равно n; если $n = 1$, печатать ничего не надо).

Решение. Вариант 1.

```
k := n;
{инвариант: произведение напечатанных чисел и k равно
 n, напечатаны только простые числа}
while not (k = 1) do begin
| l := 2;
| {инвариант: k не имеет делителей в интервале (1,l)}
| while k mod l <> 0 do begin
| | l := l + 1;
| end;
| {l - наименьший делитель k, больший 1, следовательно,
|  простой}
| writeln (l);
| k:=k div l;
end;
```

Вариант 2.

```
k := n; l := 2;
{произведение  k и напечатанных чисел равно n; напечатанные
 числа просты; k не имеет делителей, меньших l}
while not (k = 1) do begin
| if k mod l = 0  then begin
| | {k делится на l и не имеет делителей,
| |   меньших l, значит, l просто}
| | k := k div l;
| | writeln (l);
| end else begin
| | { k не делится на l }
| | l := l+1;
| end;
end;
```

□

1.1.23. Составить программу решения предыдущей задачи, использующую тот факт, что составное число имеет делитель, не превосходящий квадратного корня из этого числа.

Решение. Во втором варианте решения вместо `l:=l+1` можно написать

```
if l*l > k then begin
| l:=k;
end else begin
| l:=l+1;
end;
```

□

1.1.24. Проверить, является ли заданное натуральное число $\mathtt{n} > 1$ простым. □

1.1.25. (Для знакомых с основами алгебры) Дано целое гауссово число $\mathtt{n} + \mathtt{m}i$ (принадлежащее $\mathbb{Z}[i]$).

(а) Проверить, является ли оно простым (в $\mathbb{Z}[i]$).

(б) Напечатать его разложение на простые (в $\mathbb{Z}[i]$) множители. □

1.1.26. Разрешим применять команды `write(i)` лишь при $\mathtt{i} = 0, 1, 2, \ldots, 9$. Составить программу, печатающую десятичную запись заданного натурального числа $\mathtt{n} > 0$. (Случай $\mathtt{n} = 0$ явился бы некоторым исключением, так как обычно нули в начале числа не печатаются, а для $\mathtt{n} = 0$ — печатаются.)

Решение.

```
base:=1;
{base - степень 10, не превосходящая n}
while 10 * base <= n do begin
| base:= base * 10;
end;
{base - максимальная степень 10, не превосходящая n}
k:=n;
{инвариант: осталось напечатать k с тем же числом
 знаков, что в base; base = 100..00}
while base <> 1 do begin
| write(k div base);
| k:= k mod base;
| base:= base div 10;
end;
{base=1; осталось напечатать однозначное число k}
write(k);
```
□

Типичная ошибка при решении этой задачи: неправильно обрабатываются числа с нулями посередине. Приведённый инвариант допускает случай, когда $\mathtt{k} < \mathtt{base}$; в этом случае печатание `k` начинается со старших нулей.

1.1.27. То же самое, но надо напечатать десятичную запись в обратном порядке. (Для $\mathtt{n} = 173$ надо напечатать 371.)

Решение.

```
k:= n;
{инвариант: осталось напечатать k в обратном порядке}
```

```
while k <> 0 do begin
| write (k mod 10);
| k:= k div 10;
end;
```
□

1.1.28. Дано натуральное n. Подсчитать количество решений неравенства $x^2 + y^2 < n$ в натуральных (неотрицательных целых) числах, не используя действий с вещественными числами.

Решение.

```
k := 0; s := 0;
{инвариант: s = количество решений неравенства
  x*x + y*y < n с x < k}
while k*k < n do begin
| ...
| {t = число решений неравенства k*k + y*y < n
|  с y>=0 (при данном k) }
| k := k + 1;
| s := s + t;
end;
{k*k >= n, поэтому s = количество всех решений
  неравенства}
```

Здесь ... — пока ещё не написанный кусок программы, который будет таким:

```
l := 0; t := 0;
{инвариант: t = число решений
  неравенства k*k + y*y < n с 0<=y<l }
while k*k + l*l < n do begin
| l := l + 1;
| t := t + 1;
end;
{k*k + l*l >= n,  поэтому  t = число
  всех решений неравенства k*k + y*y < n}
```
□

1.1.29. Та же задача, но количество операций должно быть порядка $\sqrt{n}$. (В предыдущем решении, как можно подсчитать, порядка n операций.)

Решение. Нас интересуют точки решётки (с целыми координатами) в первом квадранте, попадающие внутрь круга радиуса $\sqrt{n}$. Интересующее нас множество (назовём его X) состоит из объединения

вертикальных столбцов убывающей высоты.

Идея решения состоит в том, чтобы «двигаться вдоль его границы», спускаясь по верхнему его краю, как по лестнице. Координаты движущейся точки обозначим `<k,l>`. Введём ещё одну переменную `s` и будем поддерживать истинность такого условия:

> `<k,l>` находится сразу над `k`-ым столбцом;
> `s` — число точек в предыдущих столбцах.

Формально:

- `l` — минимальное среди тех $\texttt{l} \geqslant 0$, для которых `<k,l>` не принадлежит `X`;
- `s` — число пар натуральных `x, y`, для которых $\texttt{x} < \texttt{k}$ и `<x,y>` принадлежит `X`.

Обозначим эти условия через (И).

```
k := 0; l := 0;
while <0,l> принадлежит X do begin
| l := l + 1;
end;
{k = 0, l - минимальное среди тех l >= 0,
 для которых <k,l> не принадлежит X}
s := 0;
{инвариант: И}
while not (l = 0) do begin
| s := s + l;
| {s - число точек в столбцах до k-го включительно}
| k := k + 1;
| {точка <k,l> лежит вне X, но, возможно, её надо сдвинуть
|    вниз, чтобы восстановить И}
| while (l <> 0) and (<k, l-1> не принадлежит X) do begin
| | l := l - 1;
| end;
end;
{И, l = 0, поэтому k-ый столбец и все следующие пусты, а
   s равно искомому числу}
```

Оценка числа действий очевидна: сначала мы движемся вверх не более чем на $\sqrt{n}$ шагов, а затем вниз и вправо — в каждую сторону не более чем на $\sqrt{n}$ шагов. □

1.1.30. Даны натуральные числа n и k, $n > 1$. Напечатать k десятичных знаков числа $1/n$. (При наличии двух десятичных разложений выбирается то из них, которое не содержит девятки в периоде.) Программа должна использовать только целые переменные.

Решение. Сдвинув в десятичной записи числа $1/n$ запятую на k мест вправо, получим число $10^k/n$. Нам надо напечатать его целую часть, то есть разделить 10^k на n нацело. Стандартный способ требует использования больших по величине чисел, которые могут выйти за границы диапазона представимых чисел. Поэтому мы сделаем иначе (следуя обычному методу «деления уголком») и будем хранить «остаток» r:

```
l := 0; r := 1;
{инв.: напечатано l разрядов 1/n, осталось напечатать
  k - l разрядов дроби r/n}
while l <> k do begin
  write ( (10 * r) div n);
  r := (10 * r) mod n;
  l := l + 1;
end;
```
□

1.1.31. Дано натуральное число $n > 1$. Определить длину периода десятичной записи дроби $1/n$.

Решение. Период дроби равен периоду в последовательности остатков (докажите это; в частности, надо доказать, что он не может быть меньше). Кроме того, в этой последовательности все периодически повторяющиеся члены различны, а предпериод имеет длину не более n. Поэтому достаточно найти $(n+1)$-ый член последовательности остатков и затем минимальное k, при котором $(n+1+k)$-ый член совпадает с $(n+1)$-ым.

```
l := 0; r := 1;
{инвариант: r/n = результат отбрасывания l знаков в 1/n}
while l <> n+1 do begin
  r := (10 * r) mod n;
  l := l + 1;
end;
c := r;
```

```
{c = (n+1)-ый член последовательности остатков}
r := (10 * r) mod n;
k := 1;
{r = (n+k+1)-ый член последовательности остатков}
while r <> c do begin
| r := (10 * r) mod n;
| k := k + 1;
end;
```
□

1.1.32. (Сообщил Ю. В. Матиясевич) Дана функция $\mathtt{f} : \{1 \ldots \mathtt{N}\} \to \{1 \ldots \mathtt{N}\}$ Найти период последовательности $1, \mathtt{f}(1), \mathtt{f}(\mathtt{f}(1)), \ldots$ Количество действий должно быть пропорционально суммарной длине предпериода и периода (эта сумма может быть существенно меньше N).

Решение. Если отбросить начальный кусок, последовательность периодична, причём все члены периода различны.

```
{Обозначение: f[n,1]=f(f(...f(1)...)) (n раз)}
k:=1; a:=f(1); b:=f(f(1));
{a=f[k,1]; b=f[2k,1]}
while a <> b do begin
| k:=k+1; a:=f(a); b:=f(f(b));
end;
{a=f[k,1]=f[2k,1]; f[k,1] входит в периодическую часть}
l:=1; b:=f(a);
{b=f[k+l,1]; f[k,1],...,f[k+l-1,1] различны}
while a <> b do begin
| l:=l+1; b:=f(b);
end;
{период равен l}
```
□

1.1.33. (Э. Дейкстра) Функция f с натуральными аргументами и значениями определена так: $\mathtt{f}(0) = 0$, $\mathtt{f}(1) = 1$, $\mathtt{f}(2\mathtt{n}) = \mathtt{f}(\mathtt{n})$, $\mathtt{f}(2\mathtt{n}+1) = \mathtt{f}(\mathtt{n}) + \mathtt{f}(\mathtt{n}+1)$. Составить программу вычисления $\mathtt{f}(\mathtt{n})$ по заданному n, требующую порядка $\log \mathtt{n}$ операций.

Решение.

```
k := n; a := 1; b := 0;
{инвариант: 0 <= k, f (n) = a * f(k) + b * f (k+1)}
while k <> 0 do begin
| if k mod 2 = 0  then begin
| | l := k div 2;
```

```
    | | {k=2l, f(k)=f(l), f(k+1) = f(2l+1) = f(l) + f(l+1),
    |  f (n) = a*f(k) + b*f(k+1) = (a+b)*f(l) + b*f(l+1)}
    | a := a + b; k := l;
  end else begin
    | l := k div 2;
    | {k = 2l + 1, f(k) = f(l) + f(l+1),
    |  f(k+1) = f(2l+2) = f(l+1),
    |  f(n) = a*f(k) + b*f(k+1) = a*f(l) + (a+b)*f(l+1)}
    | b := a + b; k := l;
  end;
end;
{k = 0, f(n) = a * f(0) + b * f(1) = b, что и требовалось}
```
□

1.1.34. То же, если $\mathtt{f}(0) = 13$, $\mathtt{f}(1) = 17$, $\mathtt{f}(2) = 20$, $\mathtt{f}(3) = 30$, $\mathtt{f}(2n) = 43\,\mathtt{f}(n) + 57\,\mathtt{f}(n+1)$, $\mathtt{f}(2n+1) = 91\,\mathtt{f}(n) + 179\,\mathtt{f}(n+1)$ при $n \geqslant 2$.

[Указание. Хранить коэффициенты в выражении $\mathtt{f}(n)$ через три соседних числа.] □

1.1.35. Даны натуральные числа a и b, причём $b > 0$. Найти частное и остаток при делении a на b, оперируя лишь с целыми числами и не используя операции div и mod, за исключением деления на 2 чётных чисел; число шагов не должно превосходить $C_1 \log(a/b) + C_2$ для некоторых констант C_1, C_2.

Решение.

```
b1 := b;
while b1 <= a do begin
  | b1 := b1 * 2;
end;
{b1 > a, b1 = b * (некоторая степень 2)}
q:=0; r:=a;
{инвариант: q, r - частное и остаток при делении a на b1,
 b1 = b * (некоторая степень 2)}
while b1 <> b do begin
  | b1 := b1 div 2 ; q := q * 2;
  | { a = b1 * q + r, 0 <= r, r < 2 * b1}
  | if r >= b1 then begin
  |   | r := r - b1;
  |   | q := q + 1;
  | end;
end;
{q, r - частное и остаток при делении a на b}
```
□

1.2. Массивы

В следующих задачах переменные x, y, z предполагаются описанными как `array[1..n] of integer` (где n — некоторое натуральное число, большее 0), если иное не оговорено явно.

1.2.1. Заполнить массив x нулями. (Это означает, что нужно составить фрагмент программы, после выполнения которого все значения x[1]..x[n] равнялись бы нулю, независимо от начального значения переменной x.)

Решение.

```
i := 0;
{инвариант: первые i значений x[1]..x[i] равны 0}
while i <> n do begin
| i := i + 1;
| {x[1]..x[i-1] = 0}
| x[i] := 0;
end;
```
□

1.2.2. Подсчитать количество нулей в массиве x. (Составить фрагмент программы, не меняющий значения x, после исполнения которого значение некоторой целой переменной k равнялось бы числу нулей среди компонент массива x.)

Решение.

```
...
{инвариант: k = число нулей среди x[1]...x[i] }
...
```
□

1.2.3. Не используя оператора присваивания для массивов, составить фрагмент программы, эквивалентный оператору x:=y.

Решение.

```
i := 0;
{инвариант: значение y не изменилось, x[l]=y[l] при l<=i}
while i <> n do begin
| i := i + 1;
| x[i] := y[i];
end;
```
□

1.2.4. Найти максимум из x[1]..x[n].

Решение.

```
i := 1; max := x[1];
{инвариант: max = максимум из x[1]..x[i]}
while i <> n do begin
| i := i + 1;
| {max = максимум из x[1]..x[i-1]}
| if x[i] > max then begin
| | max := x[i];
| end;
end;
```
□

1.2.5. Дан массив `x: array[1..n] of integer`, причём известно, что $x[1] \leqslant x[2] \leqslant \ldots \leqslant x[n]$. Найти количество различных чисел среди элементов этого массива.

Решение. Вариант 1.

```
i := 1; k := 1;
{инвариант: k - количество различных среди x[1]..x[i]}
while i <> n do begin
| i := i + 1;
| if x[i] <> x[i-1] then begin
| | k := k + 1;
| end;
end;
```

Вариант 2. Искомое число на 1 больше количества тех чисел `i` из `1..n-1`, для которых `x[i]` не равно `x[i+1]`.

```
k := 1;
for i := 1 to n-1 do begin
| if x[i]<> x[i+1] then begin
| | k := k + 1;
| end;
end;
```
□

1.2.6. Дан массив `x: array[1..n] of integer`. Найти количество различных чисел среди элементов этого массива. (Число действий должно быть порядка n^2.) □

1.2.7. Та же задача, если требуется, чтобы количество действий было порядка $n \log n$.

[Указание. Смотри главу 4 (Сортировка).] □

1.2.8. Та же задача, если известно, что все элементы массива — числа от 1 до k и число действий должно быть порядка n + k. □

1.2.9. (Сообщил А. Л. Брудно) Прямоугольное поле m × n разбито на mn квадратных клеток. Некоторые клетки покрашены в чёрный цвет. Известно, что все чёрные клетки могут быть разбиты на несколько непересекающихся и не имеющих общих вершин чёрных прямоугольников. Считая, что цвета клеток даны в виде массива типа

```
array [1..m] of array [1..n] of boolean;
```

подсчитать число чёрных прямоугольников, о которых шла речь. Число действий должно быть порядка mn.

Решение. Число прямоугольников равно числу их левых верхних углов. Является ли клетка верхним углом, можно узнать, посмотрев на её цвет, а также цвет верхнего и левого соседей. (Не забудьте, что их может не быть, если клетка с краю.) □

1.2.10. Дан массив x[1]..x[n] целых чисел. Не используя других массивов, переставить элементы массива в обратном порядке.

Решение. Элементы x[i] и x[n+1-i] нужно поменять местами для всех i, для которых $i < n + 1 - i$, то есть $2i < n + 1 \Leftrightarrow 2i \leqslant n \Leftrightarrow$ $\Leftrightarrow i \leqslant n \text{ div } 2$:

```
for i := 1 to n div 2 do begin
| ...поменять местами  x[i] и x[n+1-i];
end;
```
□

1.2.11. (Из книги Д. Гриса) Дан массив целых чисел x[1]..x[m+n], рассматриваемый как соединение двух его отрезков: начала x[1]..x[m] длины m и конца x[m+1]..x[m+n] длины n. Не используя дополнительных массивов, переставить начало и конец. (Число действий порядка m + n.)

Решение. Вариант 1. Перевернём (расположим в обратном порядке) отдельно начало и конец массива, а затем перевернём весь массив как единое целое.

Вариант 2. (А. Г. Кушниренко) Рассматривая массив записанным по кругу, видим, что требуемое действие — поворот круга. Как известно, поворот есть композиция двух осевых симметрий.

Вариант 3. Рассмотрим более общую задачу — обмен двух участков массива x[p+1]..x[q] и x[q+1]..x[r]. Предположим, что длина левого участка (назовём его A) не больше длины правого (назовём

его B). Выделим в B начало той же длины, что и A, назовём его B_1, а остаток B_2. (Так что $B = B_1 + B_2$, если обозначать плюсом приписывание массивов друг к другу.) Нам надо из $A + B_1 + B_2$ получить $B_1 + B_2 + A$. Меняя местами участки A и B_1 — они имеют одинаковую длину, и сделать это легко, — получаем $B_1 + A + B_2$, и осталось поменять местами A и B_2. Тем самым мы свели дело к перестановке двух отрезков меньшей длины. Итак, получаем такую схему программы:

```
p := 0; q := m; r := m + n;
{инвариант: осталось переставить x[p+1..q], x[q+1..r]}
while (p <> q) and (q <> r) do begin
| {оба участка непусты}
| if (q - p) <= (r - q) then begin
| | ..переставить x[p+1]..x[q] и x[q+1]..x[q+(q-p)]
| | pnew := q; qnew := q + (q - p);
| | p := pnew; q := qnew;
| end else begin
| | ..переставить x[q-(r-q)+1]..x[q] и x[q+1]..x[r]
| | qnew := q - (r - q); rnew := q;
| | q := qnew; r := rnew;
| end;
end;
```

Оценка времени работы: на очередном шаге оставшийся для обработки участок становится короче на длину A; число действий при этом также пропорционально длине A. □

1.2.12. Коэффициенты многочлена лежат в массиве `a: array[0..n] of integer` (`n` — натуральное число, степень многочлена). Вычислить значение этого многочлена в точке `x`, то есть $\mathtt{a[n]}\,\mathtt{x}^{\mathtt{n}} + \ldots + \mathtt{a[1]}\,\mathtt{x} + \mathtt{a[0]}$.

Решение. (Описываемый алгоритм называется схемой Горнера.)

```
k := 0; y := a[n];
{инвариант: 0 <= k <= n,
 y= a[n]*(x в степени k)+...+a[n-1]*(x в степени k-1)+...+
                  + a[n-k]*(x в степени 0)}
while k<>n do begin
| k := k + 1;
| y := y * x + a [n-k];
end;
```

□

1.2.13. (Для знакомых с основами анализа; сообщил А. Г. Кушниренко) Дополнить алгоритм вычисления значения многочлена в задан-

ной точке по схеме Горнера вычислением значения его производной в той же точке.

Решение. Добавление нового коэффициента соответствует переходу от многочлена $P(x)$ к многочлену $xP(x) + c$. Его производная в точке x равна $xP'(x) + P(x)$. (Это решение обладает забавным свойством: не надо знать заранее степень многочлена. Если требовать выполнения этого условия, да ещё просить вычислять только значение производной, не упоминая о самом многочлене, получается не такая уж простая задача.) □

Общее утверждение о сложности вычисления производных таково:

1.2.14. (В. Баур, Ф. Штрассен) Дана программа вычисления значения некоторого многочлена $P(x_1, \ldots, x_n)$, содержащая только команды присваивания. Их правые части — выражения, содержащие сложение, умножение, константы, переменные $x_1, \ldots, x_n$ и ранее встречавшиеся (в левой части) переменные. Доказать, что существует программа того же типа, вычисляющая все n производных $\partial P/\partial x_1, \ldots, \partial P/\partial x_n$, причём общее число арифметических операций не более чем в C раз превосходит число арифметических операций в исходной программе. Константа C не зависит от n.

[Указание. Можно считать, что каждая команда — сложение двух чисел, умножение двух чисел или умножение на константу. Использовать индукцию по числу команд, применяя индуктивное предположение к программе, получающейся отбрасыванием *первой* команды.] □

1.2.15. В массивах `a: array[0..k] of integer` и `b: array[0..l] of integer` хранятся коэффициенты двух многочленов степеней `k` и `l`. Поместить в массив `c: array[0..m] of integer` коэффициенты их произведения. (Числа `k, l, m` — натуральные, `m = k + l`; элемент массива с индексом `i` содержит коэффициент при степени `i`.)

Решение.

```
for i:=0 to m do begin
| c[i]:=0;
end;
for i:=0 to k do begin
| for j:=0 to l do begin
| | c[i+j] := c[i+j] + a[i]*b[j];
| end;
end;
```
□

1.2.16. Предложенный выше алгоритм перемножения многочленов требует порядка n^2 действий для перемножения двух многочленов степени n. Придумать более эффективный (для больши́х n) алгоритм, которому достаточно порядка $n^{\log 4/\log 3}$ действий.

[Указание. Представим себе, что надо перемножить два многочлена степени $2k$. Их можно представить в виде

$$A(x)\,x^k + B(x) \quad \text{и} \quad C(x)\,x^k + D(x).$$

Произведение их равно

$$A(x)C(x)\,x^{2k} + (A(x)D(x) + B(x)C(x))\,x^k + B(x)D(x).$$

Естественный способ вычисления AC, $AD + BC$, BD требует четырёх умножений многочленов степени k, однако их количество можно сократить до трёх с помощью такой хитрости: вычислить AC, BD и $(A + B)(C + D)$, а затем заметить, что $AD + BC = (A + B)(C + D) - AC - BD$.] □

1.2.17. Даны два возрастающих массива `x: array[1..k] of integer` и `y: array[1..l] of integer`. Найти количество общих элементов в этих массивах, то есть количество тех целых `t`, для которых `t = x[i] = y[j]` для некоторых `i` и `j`. (Число действий порядка `k + l`.)

Решение.

```
k1:=0; l1:=0; n:=0;
{инвариант: 0<=k1<=k; 0<=l1<=l;
 искомый ответ = n + количество общих
 элементов в x[k1+1]...x[k] и y[l1+1]...y[l]}
while (k1 <> k) and (l1 <> l) do begin
| if x[k1+1] < y[l1+1] then begin
| | k1 := k1 + 1;
| end else if x[k1+1] > y[l1+1] then begin
| | l1 := l1 + 1;
| end else begin {x[k1+1] = y[l1+1]}
| | k1 := k1 + 1;
| | l1 := l1 + 1;
| | n := n + 1;
| end;
end;
{k1 = k или l1 = l, поэтому одно из множеств, упомянутых
 в инварианте, пусто, а n равно искомому ответу}
```

□

Замечание. В третьей альтернативе достаточно было бы увеличивать одну из переменных `k1`, `l1`; вторая добавлена для симметрии.

1.2.18. Решить предыдущую задачу, если про массивы известно лишь, что `x[1]` $\leqslant \ldots \leqslant$ `x[k]` и `y[1]` $\leqslant \ldots \leqslant$ `y[l]` (возрастание заменено неубыванием).

Решение. Условие возрастания было использовано в третьей альтернативе выбора: сдвинув `k1` и `l1` на 1, мы тем самым уменьшали на 1 количество общих элементов в `x[k1+1]`...`x[k]` и `x[l1+1]`...`x[l]`. Теперь это придётся делать сложнее.

```
...
end else begin {x[k1+1] = y[l1+1]}
| t := x [k1+1];
| while (k1<k) and (x[k1+1]=t) do begin
| | k1 := k1 + 1;
| end;
| while (l1<l) and (x[l1+1]=t) do begin
| | l1 := l1 + 1;
| end;
| n := n + 1;
end;
```
□

Замечание. Эта программа имеет дефект: при проверке условия

```
(k1<k) and (x[k1+1]=t)
```

(или второго, аналогичного) при ложной первой скобке вторая окажется бессмысленной (индекс выйдет за границы массива) и возникнет ошибка. Некоторые версии паскаля, вычисляя `A and B`, сначала вычисляют `A` и при ложном `A` не вычисляют `B`. (Так ведёт себя, например, система Turbo Pascal версии 5.0 — но не 3.0.) Тогда описанная ошибка не возникнет.

Но если мы не хотим полагаться на такое свойство используемой нами реализации паскаля (не предусмотренное его автором Н. Виртом), то можно поступить так. Введём дополнительную переменную `b: boolean` и напишем:

```
if k1 < k  then b := (x[k1+1]=t)  else  b:=false;
{b = (k1<k) and (x[k1+1] = t)}
while  b  do  begin
| k1:=k1+1;
| if k1 < k then b := (x[k1+1]=t) else b:=false;
end;
```

Можно также сделать иначе:

```
end else begin {x[k1+1] = y[l1+1]}
| if k1 + 1 = k then begin
| | k1 := k1 + 1;
| | n := n + 1;
| end else if x[k1+1] = x [k1+2] then begin
| | k1 := k1 + 1;
| end else begin
| | k1 := k1 + 1;
| | n := n + 1;
| end;
end;
```

Так будет короче, хотя менее симметрично.

Наконец, можно увеличить размер массива в его описании, включив в него фиктивные элементы.

1.2.19. Даны два неубывающих массива `x: array[1..k] of integer` и `y: array[1..l] of integer`. Найти число различных элементов среди `x[1]`, ..., `x[k]`, `y[1]`, ..., `y[l]`. (Число действий порядка `k + l`.) □

1.2.20. Даны два массива `x[1]` $\leqslant \ldots \leqslant$ `x[k]` и `y[1]` $\leqslant \ldots \leqslant$ `y[l]`. «Соединить» их в массив `z[1]` $\leqslant \ldots \leqslant$ `z[m]` (`m` $=$ `k + l`; каждый элемент должен входить в массив `z` столько раз, сколько раз он входит в общей сложности в массивы `x` и `y`). Число действий порядка `m`.

Решение.

```
k1 := 0; l1 := 0;
{инвариант: ответ получится, если к z[1]..z[k1+l1] добавить
 справа соединение массивов x[k1+1]..x[k] и y[l1+1]..y[l]}
while (k1 <> k) or (l1 <> l) do begin
| if k1 = k then begin
| | {l1 < l}
| | l1 := l1 + 1;
| | z[k1+l1] := y[l1];
| end else if l1 = l then begin
| | {k1 < k}
| | k1 := k1 + 1;
| | z[k1+l1] := x[k1];
| end else if x[k1+1] <= y[l1+1] then begin
| | k1 := k1 + 1;
| | z[k1+l1] := x[k1];
| end else if x[k1+1] >= y[l1+1] then begin
```

```
  l1 := l1 + 1;
  z[k1+l1] := y[l1];
end else begin
  { такого не бывает }
end;
end;
{k1 = k, l1 = l, массивы соединены}
```
□

Этот процесс можно пояснить так. Пусть у нас есть две стопки карточек, отсортированных по алфавиту. Мы соединяем их в одну стопку, выбирая каждый раз ту из верхних карточек обеих стопок, которая идёт раньше в алфавитном порядке. Если в одной стопке карточки кончились, берём их из другой стопки.

1.2.21. Даны два массива $\mathtt{x[1]} \leqslant \ldots \leqslant \mathtt{x[k]}$ и $\mathtt{y[1]} \leqslant \ldots \leqslant \mathtt{y[l]}$. Найти их «пересечение», то есть массив $\mathtt{z[1]} \leqslant \ldots \leqslant \mathtt{z[m]}$, содержащий их общие элементы, причём кратность каждого элемента в массиве z равняется минимуму из его кратностей в массивах x и y. Число действий порядка k + l. □

1.2.22. Даны два массива $\mathtt{x[1]} \leqslant \ldots \leqslant \mathtt{x[k]}$ и $\mathtt{y[1]} \leqslant \ldots \leqslant \mathtt{y[l]}$ и число q. Найти сумму вида x[i] + y[j], наиболее близкую к числу q. (Число действий порядка k+l, дополнительная память — фиксированное число целых переменных, сами массивы менять не разрешается.)

[Указание. Надо найти минимальное расстояние между элементами $\mathtt{x[1]} \leqslant \ldots \leqslant \mathtt{x[k]}$ и $\mathtt{q} - \mathtt{y[l]} \leqslant \ldots \leqslant \mathtt{q} - \mathtt{y[1]}$, что нетрудно сделать в ходе их слияния в один (воображаемый) массив.] □

1.2.23. (Из книги Д. Гриса) Некоторое число содержится в каждом из трёх целочисленных неубывающих массивов $\mathtt{x[1]} \leqslant \ldots \leqslant \mathtt{x[p]}$, $\mathtt{y[1]} \leqslant \ldots \leqslant \mathtt{y[q]}$, $\mathtt{z[1]} \leqslant \ldots \leqslant \mathtt{z[r]}$. Найти одно из таких чисел. Число действий должно быть порядка p + q + r.

Решение.

```
p1:=1; q1=1; r1:=1;
{инвариант: x[p1]..x[p], y[q1]..y[q], z[r1]..z[r]
 содержат общий элемент}
while not ((x[p1]=y[q1]) and (y[q1]=z[r1])) do begin
  if x[p1]<y[q1] then begin
    p1:=p1+1;
  end else if y[q1]<z[r1] then begin
    q1:=q1+1;
  end else if z[r1]<x[p1] then begin
```

```
  | r1:=r1+1;
  end else begin
  | { так не бывает }
  end;
end;
{x[p1] = y[q1] = z[r1]}
writeln (x[p1]);
```
□

1.2.24. Та же задача, только заранее не известно, существует ли общий элемент в трёх неубывающих массивах и требуется это выяснить (и найти один из общих элементов, если они есть). □

1.2.25. Элементами массива `a[1..n]` являются неубывающие массивы `[1..m]` целых чисел:

$$\texttt{a: array [1..n] of array [1..m] of integer;}$$
$$\texttt{a[1][1]} \leqslant \ldots \leqslant \texttt{a[1][m]}, \ldots, \texttt{a[n][1]} \leqslant \ldots \leqslant \texttt{a[n][m]}.$$

Известно, что существует число, входящее во все массивы `a[i]` (существует такое `x`, что для всякого `i` из `1..n` найдётся `j` из `1..m`, для которого `a[i][j]` = `x`). Найти одно из таких чисел `x`.

Решение. Введём массив `b[1]` ... `b[n]`, отмечающий начало «остающейся части» массивов `a[1]`, ..., `a[n]`.

```
for k:=1 to n do begin
|  b[k]:=1;
end;
eq := true;
for k := 2 to n do begin
| eq := eq and (a[1][b[1]] = a[k][b[k]]);
end;
{инвариант: оставшиеся части пересекаются, т.е. существует
 такое x, что для всякого i из [1..n] найдётся j из [1..m],
 не меньшее b[i], для которого a[i][j] = x;  eq <=> первые
 элементы оставшихся частей равны}
while not eq do begin
| s := 1; k := 1;
| {a[s][b[s]] - минимальное среди a[1][b[1]]..a[k][b[k]]}
| while k <> n do begin
| | k := k + 1;
| | if a[k][b[k]] < a[s][b[s]] then begin
| | | s := k;
| | end;
```

```
  end;
  {a[s][b[s]] - минимальное среди a[1][b[1]]..a[n][b[n]]}
  b [s] := b [s] + 1;
  for k := 2 to n do begin
    eq := eq and (a[1][b[1]] = a[k][b[k]]);
  end;
end;
writeln (a[1][b[1]]);
```
□

1.2.26. Приведённое решение предыдущей задачи требует порядка $\mathtt{mn}^2$ действий. Придумать способ с числом действий порядка $\mathtt{mn}$.

[Указание. Придётся пожертвовать симметрией и выбрать одну из строк за основную. Двигаясь по основной строке, поддерживаем такое соотношение: во всех остальных строках отмечен максимальный элемент, не превосходящий текущего элемента основной строки.] □

1.2.27. (Двоичный поиск) Дана последовательность $\mathtt{x[1]} \leqslant \ldots \leqslant \mathtt{x[n]}$ целых чисел и число `a`. Выяснить, содержится ли `a` в этой последовательности, то есть существует ли `i` из `1..n`, для которого $\mathtt{x[i]} = \mathtt{a}$. (Количество действий порядка $\log \mathtt{n}$.)

Решение. (Предполагаем, что $\mathtt{n} > 0$.)

```
l := 1; r := n+1;
{r > l, если a есть вообще, то есть и среди x[l]..x[r-1]}
while r - l <> 1 do begin
  m := l + (r-l) div 2 ;
  {l < m < r }
  if x[m] <= a then begin
    l := m;
  end else begin {x[m] > a}
    r := m;
  end;
end;
```

(Обратите внимание, что и в случае $\mathtt{x[m]} = \mathtt{a}$ инвариант не нарушается.)

Каждый раз $\mathtt{r} - \mathtt{l}$ уменьшается примерно вдвое, откуда и вытекает требуемая оценка числа действий. □

Замечание.

$$\mathtt{l} + (\mathtt{r}-\mathtt{l})\,\mathtt{div}\,2 = (2\mathtt{l} + (\mathtt{r} - \mathtt{l}))\,\mathtt{div}\,2 = (\mathtt{r} + \mathtt{l})\,\mathtt{div}\,2.$$

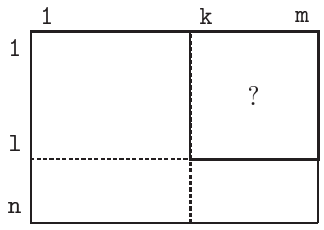
В этой задаче существенно, что массив упорядочен — поиск в неупорядоченном массиве требует времени, пропорционального длине массива. (Чтобы убедиться, что какого-то числа нет в массиве, надо просмотреть все его элементы.)

1.2.28. (Из книги Д. Гриса) Имеется массив `x: array[1..n] of array[1..m] of integer`, упорядоченный по строкам и по столбцам:

$$\texttt{x[i][j]} \leqslant \texttt{x[i][j+1]},$$
$$\texttt{x[i][j]} \leqslant \texttt{x[i+1][j]},$$

и число `a`. Требуется выяснить, встречается ли `a` среди `x[i][j]`.

Решение. Представляя себе массив `x` как матрицу (прямоугольник, заполненный числами), мы выберем прямоугольник, в котором только и может содержаться `a`, и будем его сужать. Прямоугольник этот будет содержать `x[i][j]` при $1 \leqslant \texttt{i} \leqslant \texttt{l}$ и $\texttt{k} \leqslant \texttt{j} \leqslant \texttt{m}$



(допускаются пустые прямоугольники при $\texttt{l} = 0$ и $\texttt{k} = \texttt{m} + 1$).

```
l:=n; k:=1;
{l>=0, k<=m+1, если a есть, то в описанном прямоугольнике}
while (l > 0) and (k < m+1) and (x[l][k] <> a) do begin
| if x[l][k] < a then begin
| | k := k + 1; {левый столбец не содержит a, удаляем его}
| end else begin {x[l][k] > a}
| | l := l - 1; {нижняя строка не содержит a, удаляем её}
| end;
end;
{x[l][k] = a или прямоугольник пуст }
answer:= (l > 0) and (k < m+1) ;
```

Замечание. Здесь та же ошибка: `x[l][k]` может оказаться неопределённым. (Её исправление предоставляется читателю.) □

1.2.29. (Московская олимпиада по программированию) Дан неубывающий массив положительных целых чисел a[1] ⩽ a[2] ⩽ ... ⩽ a[n]. Найти наименьшее целое положительное число, не представимое в виде суммы нескольких элементов этого массива (каждый элемент массива может быть использован не более одного раза). Число действий порядка n.

Решение. Пусть известно, что числа, представимые в виде суммы элементов a[1], ..., a[k], заполняют отрезок от 1 до некоторого N. Если a[k+1] > N+1, то N+1 и будет минимальным числом, не представимым в виде суммы элементов массива a[1] ... a[n]. Если же a[k+1] ⩽ N+1, то числа, представимые в виде суммы элементов a[1] ... a[k+1], заполняют отрезок от 1 до N+a[k+1].

```
k := 0; N := 0;
{инвариант: числа, представимые в виде суммы элементов
 массива a[1]..a[k], заполняют отрезок 1..N}
while (k <> n) and (a[k+1] <= N+1) do begin
  N := N + a[k+1];
  k := k + 1;
end;
{(k = n) или (a[k+1] > N+1); в обоих случаях ответ N+1}
writeln (N+1);
```

(Снова тот же дефект: в условии цикла при ложном первом условии второе не определено.) □

1.2.30. (Для знакомых с основами алгебры) В целочисленном массиве a[1] ... a[n] хранится перестановка чисел 1 ... n (каждое из чисел встречается по одному разу).

(а) Определить чётность перестановки. (И в (а), и в (б) количество действий порядка n.)

(б) Не используя других массивов, заменить перестановку на обратную (если до работы программы a[i] = j, то после должно быть a[j] = = i).

[Указание. (а) Чётность перестановки определяется количеством циклов. Чтобы отличать уже пройденные циклы, у их элементов можно, например, менять знак. (б) Обращение производим по циклам.] □

1.2.31. Дан массив a[1..n] и число b. Переставить числа в массиве таким образом, чтобы слева от некоторой границы стояли числа, меньшие или равные b, а справа от границы — бóльшие или равные b. Число действий порядка n.

Решение.

```
l:=0; r:=n;
{инвариант: a[1]..a[l]<=b; a[r+1]..a[n]>=b}
while l <> r do begin
| if a[l+1] <= b then begin
| | l:=l+1;
| end else if a[r] >=b then begin
| | r:=r-1;
| end else begin {a[l+1]>b; a[r]<b}
| | ..поменять a[l+1] и a[r]
| | l:=l+1; r:=r-1;
| end;
end;
```

□

1.2.32. Та же задача, но требуется, чтобы сначала шли элементы, меньшие b, затем равные b, а лишь затем бóльшие b.

Решение. Теперь потребуются три границы: до первой будут идти элементы, меньшие b, от первой до второй — равные b, затем неизвестно какие до третьей, а после третьей — бóльшие b. (Более симметричное решение использовало бы четыре границы, но вряд ли игра стоит свеч.) В качестве очередного рассматриваемого элемента берём элемент справа от средней границы.

```
l:=0; m:=0; r:=n;
{инвариант: a[1..l]<b; a[l+1..m]=b; a[r+1]..a[n]>b}
while m <> r do begin
| if a[m+1]=b then begin
| | m:=m+1;
| end else if a[m+1]>b then begin
| | ..обменять a[m+1] и a[r]
| | r:=r-1;
| end else begin {a[m+1]<b}
| | ..обменять a[m+1] и a[l+1]
| | l:=l+1; m:=m+1;
| end;
end;
```

□

1.2.33. (Вариант предыдущей задачи, названный в книге Дейкстры *задачей о голландском флаге.*) В массиве длины n стоят числа 0, 1 и 2. Переставить их в порядке возрастания, если единственной разрешённой операцией (помимо чтения) над массивом является перестановка двух элементов. Число действий порядка n. □

1.2.34. Дан массив a[1..n] и число $m \leqslant n$. Для каждого участка из m стоящих рядом членов (таких участков, очевидно, $n - m + 1$) вычислить его сумму. Общее число действий должно быть порядка n.

Решение. Переходя от участка к соседнему, мы добавляем один член, а другой вычитаем. □

1.2.35. Дана квадратная таблица a[1..n][1..n] и число $m \leqslant n$. Для каждого квадрата $m \times m$ в этой таблице вычислить сумму стоящих в нём чисел. Общее число действий порядка n^2.

Решение. Сначала для каждого горизонтального прямоугольника размером $m \times 1$ вычисляем сумму стоящих в нём чисел. (При сдвиге такого прямоугольника по горизонтали на 1 нужно добавить одно число и одно вычесть.) Затем, используя эти суммы, вычисляем суммы в квадратах. (При сдвиге квадрата по вертикали добавляется полоска, а другая полоска убавляется.) □

1.2.36. В массиве a[1] ... a[n] встречаются по одному разу все целые числа от 0 до n, кроме одного. Найти пропущенное число за время порядка n и с конечной дополнительной памятью.

[Указание. Сложить все числа в массиве.] □

1.3. Индуктивные функции (по А. Г. Кушниренко)

Пусть M — некоторое множество. Функция f, аргументами которой являются последовательности элементов множества M, а значениями — элементы некоторого множества N, называется *индуктивной*, если её значение на последовательности x[1] ... x[n] можно восстановить по её значению на последовательности x[1] ... x[n-1] и по x[n], то есть если существует функция $F\colon N \times M \to N$, для которой

$$f(\langle x[1], \ldots, x[n]\rangle) = F(\,f(\langle x[1], \ldots, x[n\text{-}1]\rangle), x[n]).$$

Например, функция sum (сумма всех членов последовательности) индуктивна, поскольку очередной член последовательности прибавляется к её сумме:

$$\mathrm{sum}(\langle x[1], \ldots, x[n]\rangle) = \mathrm{sum}(\langle x[1], \ldots, x[n\text{-}1]\rangle) + x[n].$$

Другой пример индуктивной функции — длина последовательности. В этом случае $F(n, m) = n + 1$.

Напротив, среднее арифметическое не является индуктивной функцией: если мы знаем среднее арифметическое некоторой последовательности, но не знаем её длины, то не можем предсказать, каким станет среднее арифметическое после дописывания некоторого (известного нам) числа.

Схема алгоритма вычисления индуктивной функции:

```
k := 0; f := f0;
{инвариант: f - значение функции на <x[1],...,x[k]>}
while  k<>n do begin
| k := k + 1;
| f := F (f, x[k]);
end;
```

Здесь `f0` — значение функции на пустой последовательности (последовательности длины `0`). Если функция `f` определена только на непустых последовательностях, то первая строка заменяется на

```
k:=1; f:=f(<x[1]>);
```

Если функция `f` не является индуктивной, полезно искать её *индуктивное расширение* — такую индуктивную функцию `g`, значения которой определяют значения `f` (это значит, что существует такая функция `t`, что

$$\texttt{f}(\langle \texttt{x[1]} \dots \texttt{x[n]} \rangle) = \texttt{t}(\texttt{g}(\langle \texttt{x[1]} \dots \texttt{x[n]} \rangle))$$

при всех $\langle \texttt{x[1]} \dots \texttt{x[n]} \rangle$). Можно доказать, что среди всех индуктивных расширений существует минимальное расширение `F` (минимальность означает, что для любого индуктивного расширения `g` значения `F` определяются значениями `g`).

1.3.1. Указать индуктивные расширения для следующих функций:

(а) среднее арифметическое последовательности вещественных чисел;

(б) число элементов последовательности целых чисел, равных её максимальному элементу;

(в) второй по величине элемент последовательности целых чисел (тот, который будет вторым, если переставить члены в неубывающем порядке);

(г) максимальное число идущих подряд одинаковых элементов;

(д) максимальная длина монотонного (неубывающего или невозрастающего) участка из идущих подряд элементов в последовательности целых чисел;

(е) число групп из единиц, разделённых нулями (в последовательности нулей и единиц).

Решение.
(а) ⟨сумма всех членов последовательности; длина⟩;
(б) ⟨число элементов, равных максимальному; значение максимального⟩;
(в) ⟨наибольший элемент последовательности; второй по величине элемент⟩;
(г) ⟨максимальное число идущих подряд одинаковых элементов; число идущих подряд одинаковых элементов в конце последовательности; последний элемент последовательности⟩;
(д) ⟨максимальная длина монотонного участка; максимальная длина неубывающего участка в конце последовательности; максимальная длина невозрастающего участка в конце последовательности; последний член последовательности⟩;
(е) ⟨число групп из единиц, последний член⟩. □

1.3.2. (Сообщил Д. В. Варсанофьев) Даны две последовательности целых чисел x[1] ... x[n] и y[1] ... y[k]. Выяснить, является ли вторая последовательность подпоследовательностью первой, то есть можно ли из первой вычеркнуть некоторые члены так, чтобы осталась вторая. Число действий порядка n + k.

Решение. Вариант 1. Будем сводить задачу к задаче меньшего размера.

```
n1:=n;
k1:=k;
{инвариант: искомый ответ <=> возможность из x[1]..x[n1]
   получить y[1]..y[k1] }
while (n1 > 0) and (k1 > 0) do begin
| if x[n1] = y[k1] then begin
| | n1 := n1 - 1;
| | k1 := k1 - 1;
| end else begin
| | n1 := n1 - 1;
| end;
end;
{n1 = 0 или k1 = 0; если k1 = 0, то ответ - да, если k1<>0
 (и n1 = 0), то ответ - нет}
answer := (k1 = 0);
```

Мы использовали то, что если x[n1] = y[k1] и y[1] ... y[k1] — под-

последовательность x[1]...x[n1], то y[1]...y[k1-1] — подпоследовательность x[1]...x[n1-1].

Вариант 2. Функция $\langle$x[1]...x[n1]$\rangle \mapsto$ [максимальное k1, для которого y[1]...y[k1] есть подпоследовательность x[1]...x[n1]] индуктивна. □

1.3.3. Даны две последовательности x[1]...x[n] и y[1]...y[k] целых чисел. Найти максимальную длину последовательности, являющейся подпоследовательностью обеих последовательностей. Количество операций порядка n · k.

Решение (сообщено М. Н. Вайнцвайгом, А. М. Дементманом). Обозначим через f(p, q) максимальную длину общей подпоследовательности последовательностей x[1]...x[p] и y[1]...y[q]. Тогда

$$\begin{aligned} &\texttt{x[p]} \neq \texttt{y[q]} \quad \Rightarrow \quad \texttt{f(p,q)} = \max\left(\texttt{f(p,q}-1\texttt{)}, \texttt{f(p}-1\texttt{,q)}\right);\\ &\texttt{x[p]} = \texttt{y[q]} \quad \Rightarrow \quad \texttt{f(p,q)} = \max\left(\texttt{f(p,q}-1\texttt{)}, \texttt{f(p}-1\texttt{,q)}, \texttt{f(p}-1\texttt{,q}-1\texttt{)}+1\right); \end{aligned}$$

(Поскольку $\texttt{f(p}-1\texttt{,q}-1\texttt{)}+1 \geqslant \texttt{f(p,q}-1\texttt{)}, \texttt{f(p}-1\texttt{,q)}$, во втором случае максимум трёх чисел можно заменить на третье из них.) Поэтому можно заполнять таблицу значений функции f, имеющую размер n · k. Можно обойтись и памятью порядка k (или n), если индуктивно (по p) вычислять $\langle$f(p,0),..., f(p,k)$\rangle$ (как функция от p этот набор индуктивен). □

1.3.4. (Из книги Д. Гриса) Дана последовательность целых чисел x[1],..., x[n]. Найти максимальную длину её возрастающей подпоследовательности (число действий порядка n log n).

Решение. Искомая функция не индуктивна, но имеет следующее индуктивное расширение: в него входят помимо максимальной длины возрастающей подпоследовательности (обозначим её k) также и числа u[1],..., u[k], где u[i] — минимальный из последних членов возрастающих подпоследовательностей длины i. Очевидно, $\texttt{u[1]} \leqslant \ldots \leqslant \texttt{u[k]}$. При добавлении нового члена в x значения u и k корректируются.

```
n1 := 1; k := 1; u[1] := x[1];
{инвариант: k и u соответствуют данному выше описанию}
while n1 <> n do begin
  n1 := n1 + 1;
  ...
  {i - наибольшее из тех чисел отрезка 1..k, для
    которых u[i] < x[n1]; если таких нет, то i=0 }
  if i = k then begin
```

```
    | | k := k + 1;
    | | u[k+1] := x[n1];
    | end else begin {i < k, u[i] < x[n1] <= u[i+1] }
    | | u[i+1] := x[n1];
    | end;
end;
```

Фрагмент ... использует идею двоичного поиска; в инварианте условно полагаем `u[0]` равным минус бесконечности, а `u[k+1]` — плюс бесконечности. Наша цель: `u[i]` $<$ `x[n1]` $\leqslant$ `u[i+1]`.

```
i:=0; j:=k+1;
{u[i] < x[n1] <= u[j], j > i}
while (j - i) <> 1 do begin
| s := i + (j-i) div 2;    {i < s < j}
| if x[n1] <= u[s] then begin
| | j := s;
| end else begin {u[s] < x[n1]}
| | i := s;
| end;
end;
{u[i] < x[n1] <= u[j], j-i = 1}
```

□

Замечание. Более простое (но не минимальное) индуктивное расширение получится, если для каждого `i` хранить максимальную длину возрастающей подпоследовательности, оканчивающейся на `x[i]`. Это расширение приводит к алгоритму с числом действий порядка n^2. Есть и другой изящный алгоритм с квадратичным временем работы (сообщил М. В. Вьюгин): найти максимальную общую подпоследовательность исходной последовательности и отсортированной последовательности с помощью предыдущей задачи.

1.3.5. Какие изменения нужно внести в решение предыдущей задачи, если надо искать максимальную *неубывающую* последовательность? □

2. ПОРОЖДЕНИЕ КОМБИНАТОРНЫХ ОБЪЕКТОВ

Здесь собраны задачи, в которых требуется получить один за другим все элементы некоторого множества.

2.1. Размещения с повторениями

2.1.1. Напечатать все последовательности длины k из чисел 1..n.

Решение. Будем печатать их в лексикографическом порядке (последовательность a предшествует последовательности b, если для некоторого s их начальные отрезки длины s равны, а (s+1)-ый член последовательности a меньше). Первой будет последовательность <1,1,...,1>, последней — последовательность <n,n,...,n>. Будем хранить последнюю напечатанную последовательность в массиве x[1]..x[k].

```
...x[1]...x[k] положить равными 1
...напечатать x
...last[1]...last[k] положить равным n
{напечатаны все до x включительно}
while x <> last do begin
  ...x := следующая за x последовательность
  ...напечатать x
end;
```

Опишем, как можно перейти от x к следующей последовательности. Согласно определению, у следующей последовательности первые s членов должны быть такими же, а (s+1)-ый — больше. Это возможно, если x[s+1] меньше n. Среди таких s нужно выбрать наибольшее (иначе полученная последовательность не будет непосредственно следующей).

Соответствующее x[s+1] нужно увеличить на 1. Итак, надо, двигаясь с конца последовательности, найти самый правый член, меньший n (он найдётся, т. к. по предположению x<>last), увеличить его на 1, а идущие за ним члены положить равными 1.

```
p:=k;
while not (x[p] < n) do begin
| p := p-1;
end;
{x[p] < n, x[p+1] =...= x[k] = n}
x[p] := x[p] + 1;
for i := p+1 to k do begin
| x[i]:=1;
end;
```
□

Замечание. Если членами последовательности считать числа не от 1 до n, а от 0 до n-1, то переход к следующему соответствует прибавлению единицы в n-ичной системе счисления.

2.1.2. В предложенном алгоритме используется сравнение двух массивов (x <> last). Устранить его, добавив булевскую переменную l и включив в инвариант соотношение

$$\texttt{l} \Leftrightarrow \text{последовательность } \texttt{x} \text{ — последняя.}$$ □

2.1.3. Напечатать все подмножества множества {1...k}.

Решение. Подмножества находятся во взаимно однозначном соответствии с последовательностями нулей и единиц длины k. □

2.1.4. Напечатать все последовательности положительных целых чисел длины k, у которых i-ый член не превосходит i. □

2.2. Перестановки

2.2.1. Напечатать все перестановки чисел 1..n (то есть последовательности длины n, в которые каждое из этих чисел входит по одному разу).

Решение. Перестановки будем хранить в массиве x[1]..x[n] и печатать в лексикографическом порядке. (Первой при этом будет перестановка ⟨1 2...n⟩, последней — ⟨n...2 1⟩. Для составления алгоритма перехода к следующей перестановке зададимся вопросом: в каком случае k-ый член перестановки можно увеличить, не меняя предыдущих?

Ответ: если он меньше какого-либо из следующих членов (т. е. членов с номерами больше k). Мы должны найти наибольшее k, при котором это так, т. е. такое k, что

$$x[k] < x[k+1] > \ldots > x[n]$$

После этого значение x[k] нужно увеличить минимальным возможным способом, т. е. найти среди x[k+1]..x[n] наименьшее число, большее его. Поменяв x[k] с ним, остаётся расположить числа с номерами k+1..n так, чтобы перестановка была наименьшей, т. е. в возрастающем порядке. Это облегчается тем, что они уже расположены в убывающем порядке.

Алгоритм перехода к следующей перестановке:

```
{<x[1]...x[n]> <> <n...2,1>}
k:=n-1;
{последовательность справа от k убывающая: x[k+1]>...>x[n]}
while x[k] > x[k+1] do begin
| k:=k-1;
end;
{x[k] < x[k+1] > ... >  x[n]}
t:=k+1;
{t <=n, все члены отрезка x[k+1] > ... > x[t] больше x[k]}
while (t < n) and (x[t+1] > x[k]) do begin
| t:=t+1;
end;
{x[k+1] > ... > x[t] > x[k] > x[t+1] > ... > x[n]}
... обменять x[k] и x[t]
{x[k+1] > ... > x[n]}
... переставить участок x[k+1] ... x[n] в обратном порядке
```
□

Замечание. Программа имеет знакомый дефект: если t=n, то x[t+1] не определено.

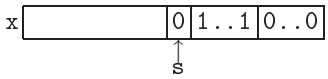
2.3. Подмножества

2.3.1. Для заданных n и k ($k \leqslant n$) перечислить все k-элементные подмножества множества {1..n}.

Решение. Будем представлять каждое подмножество последовательностью x[1]..x[n] нулей и единиц длины n, в которой ровно k единиц. (Другой способ представления разберём позже.) Такие последовательности упорядочим лексикографически (см. выше). Очевидный способ решения задачи — перебирать все последовательности как раньше,

а затем отбирать среди них те, у которых k единиц — мы отбросим, считая его неэкономичным (число последовательностей с k единицами может быть много меньше числа всех последовательностей). Будем искать такой алгоритм, чтобы получение очередной последовательности требовало не более C·n действий.

В каком случае s-ый член последовательности можно увеличить, не меняя предыдущие? Если x[s] меняется с 0 на 1, то для сохранения общего числа единиц нужно справа от x[s] заменить 1 на 0. Для этого надо, чтобы справа от x[s] единицы были. Если мы хотим перейти к *непосредственно* следующему, то x[s] должен быть *первым справа* нулём, за которым стоят единицы. Легко видеть, что x[s+1]=1 (иначе x[s] не первый). Таким образом надо искать наибольшее s, для которого x[s]=0, x[s+1]=1:



За x[s+1] могут идти ещё несколько единиц, а после них несколько нулей. Заменив x[s] на 1, надо выбрать идущие за ним члены так, чтобы последовательность была бы минимальна с точки зрения нашего порядка, т. е. чтобы сначала шли нули, а потом единицы. Вот что получается:

первая последовательность: 0..01..1 (n-k нулей, k единиц);

последняя последовательность: 1..10..0 (k единиц, n-k нулей);

алгоритм перехода к следующей за x[1]..x[n] последовательности (предполагаем, что она есть):

```
s := n - 1;
while not ((x[s]=0) and (x[s+1]=1)) do begin
| s := s - 1;
end;
{s - член, подлежащий изменению с 0 на 1}
num:=0;
for k := s to n do begin
| num := num + x[k];
end;
{num - число единиц на участке x[s]...x[n], число нулей
 равно (длина - число единиц), т.е. (n-s+1) - num}
x[s]:=1;
for k := s+1 to n-num+1 do begin
| x[k] := 0;
end;
```

```
{осталось поместить num-1 единиц в конце}
for k := n-num+2 to n do begin
| x[k]:=1;
end;
```
□

Другой способ представления подмножеств — это перечисление их элементов. Чтобы каждое подмножество имело ровно одно представление, договоримся перечислять элементы в возрастающем порядке. Приходим к такой задаче.

2.3.2. Перечислить все возрастающие последовательности длины k из чисел 1..n в лексикографическом порядке. (Пример: при n=5, k=2 получаем: 12 13 14 15 23 24 25 34 35 45.)

Решение. Минимальной будет последовательность ⟨1 2 . . . k⟩; максимальной — ⟨(n-k+1) . . . (n-1) n⟩. В каком случае s-ый член последовательности можно увеличить? Ответ: если он меньше n-k+s. После увеличения s-го элемента все следующие должны возрастать с шагом 1. Получаем такой алгоритм перехода к следующему:

```
s:=n;
while not (x[s] < n-k+s) do begin
| s:=s-1;
end;
{s - номер элемента, подлежащего увеличению};
x[s] := x[s]+1;
for i := s+1 to n do begin
| x[i] := x[i-1]+1;
end;
```
□

2.3.3. Пусть мы решили представлять k-элементные подмножества множества {1..n} убывающими последовательностями длины k, упорядоченными по-прежнему лексикографически. (Пример: 21 31 32 41 42 43 51 52 53 54.) Как выглядит тогда алгоритм перехода к следующей?

Ответ. Ищем наибольшее s, для которого x[s+1]+1 < x[s]. (Если такого s нет, полагаем s=0.) Увеличив x[s+1] на 1, кладём остальные минимально возможными (x[t]=k+1-t для t>s). □

2.3.4. Решить две предыдущие задачи, заменив лексикографический порядок на обратный (раньше идут те, которые больше в лексикографическом порядке). □

2.3.5. Перечислить все вложения (функции, переводящие разные элементы в разные) множества {1..k} в {1..n} (предполагается, что

$k \leqslant n$). Порождение очередного элемента должно требовать не более $C \cdot k$ действий.

[Указание. Эта задача может быть сведена к перечислению подмножеств и перестановок элементов каждого подмножества.] □

2.4. Разбиения

2.4.1. Перечислить все разбиения целого положительного числа n на целые положительные слагаемые (разбиения, отличающиеся лишь порядком слагаемых, считаются за одно). (Пример: n=4, разбиения 1+1+1+1, 2+1+1, 2+2, 3+1, 4.)

Решение. Договоримся, что (1) в разбиениях слагаемые идут в невозрастающем порядке, (2) сами разбиения мы перечисляем в лексикографическом порядке. Разбиение храним в начале массива x[1]..x[n], при этом количество входящих в него чисел обозначим k. В начале x[1]=...=x[n]=1, k=n, в конце x[1]=n, k=1.

В каком случае x[s] можно увеличить, не меняя предыдущих? Во-первых, должно быть x[s-1]>x[s] или s=1. Во-вторых, s должно быть не последним элементом (увеличение s надо компенсировать уменьшением следующих). Увеличив s, все следующие элементы надо взять минимально возможными.

```
s := k - 1;
while not ((s=1) or (x[s-1] > x[s])) do begin
| s := s-1;
end;
{s - подлежащее увеличению слагаемое}
x [s] := x[s] + 1;
sum := 0;
for i := s+1 to k do begin
| sum := sum + x[i];
end;
{sum - сумма членов, стоявших после x[s]}
for i := 1 to sum-1 do begin
| x [s+i] := 1;
end;
k := s+sum-1;
```
□

2.4.2. Представляя по-прежнему разбиения как невозрастающие последовательности, перечислить их в порядке, обратном лексикографическому (для n=4, например, должно быть 4, 3+1, 2+2, 2+1+1, 1+1+1+1).

[Указание. Уменьшать можно первый справа член, не равный 1; найдя его, уменьшим на 1, а следующие возьмём максимально возможными (равными ему, пока хватает суммы, а последний — сколько останется).] □

2.4.3. Представляя разбиения как неубывающие последовательности, перечислить их в лексикографическом порядке. Пример для n=4: 1+1+1+1, 1+1+2, 1+3, 2+2, 4.

[Указание. Последний член увеличить нельзя, а предпоследний — можно; если после увеличения на 1 предпоследнего члена за счёт последнего нарушится возрастание, то из двух членов надо сделать один, если нет, то последний член надо разбить на слагаемые, равные предыдущему, и остаток, не меньший его.] □

2.4.4. Представляя разбиения как неубывающие последовательности, перечислить их в порядке, обратном лексикографическому. Пример для n=4: 4, 2+2, 1+3, 1+1+2, 1+1+1+1.

[Указание. Чтобы элемент x[s] можно было уменьшить, необходимо, чтобы s=1 или x[s-1]<x[s]. Если x[s] не последний, то этого и достаточно. Если он последний, то нужно, чтобы $x[s-1] \leqslant \lfloor x[s]/2 \rfloor$ или s=1. (Здесь $\lfloor \alpha \rfloor$ обозначает целую часть α.)] □

2.5. Коды Грея и аналогичные задачи

Иногда бывает полезно перечислять объекты в таком порядке, чтобы каждый следующий минимально отличался от предыдущего. Рассмотрим несколько задач такого рода.

2.5.1. Перечислить все последовательности длины n из чисел 1..k в таком порядке, чтобы каждая следующая отличалась от предыдущей в единственной цифре, причём не более, чем на 1.

Решение. Рассмотрим прямоугольную доску ширины n и высоты k. На каждой вертикали будет стоять шашка. Таким образом, положения шашек соответствуют последовательностям из чисел 1..k длины n (s-ый член последовательности соответствует высоте шашки на s-ой вертикали). На каждой шашке нарисуем стрелочку, которая может быть направлена вверх или вниз. Вначале все шашки поставим на нижнюю горизонталь стрелочкой вверх. Далее двигаем шашки по такому правилу: найдя самую правую шашку, которую можно подвинуть в направлении (нарисованной на ней) стрелки, двигаем её на одну клетку

в этом направлении, а все стоящие правее неё шашки (они упёрлись в край) разворачиваем кругóм.

Ясно, что на каждом шаге только одна шашка сдвигается, т. е. один член последовательности меняется на 1. Докажем индукцией по `n`, что проходятся все последовательности из чисел `1..k`. Случай `n=1` очевиден. Пусть `n>1`. Все ходы поделим на те, где двигается последняя шашка, и те, где двигается не последняя. Во втором случае последняя шашка стоит у стены, и мы её поворачиваем, так что за каждым ходом второго типа следует `k-1` ходов первого типа, за время которых последняя шашка побывает во всех клетках. Если мы теперь забудем о последней шашке, то движения первых `n-1` по предположению индукции пробегают все последовательности длины `n-1` по одному разу; движения же последней шашки из каждой последовательности длины `n-1` делают `k` последовательностей длины `n`.

В программе, помимо последовательности `x[1]..x[n]`, будем хранить массив `d[1]..d[n]` из чисел `+1` и `-1` (`+1` соответствует стрелке вверх, `-1` — стрелке вниз).

Начальное состояние: `x[1]=...=x[n]=1`; `d[1]=...=d[n]=1`.

Приведём алгоритм перехода к следующей последовательности (одновременно выясняется, возможен ли переход — ответ становится значением булевской переменной `p`).

```
{если можно, сделать шаг и положить p := true, если нет,
 положить p := false }
i := n;
while (i > 1) and
| (((d[i]=1) and (x[i]=n)) or ((d[i]=-1) and (x[i]=1)))
|   do begin
| i:=i-1;
end;
if (d[i]=1 and x[i]=n) or (d[i]=-1 and x[i]=1) then begin
| p:=false;
end else begin
| p:=true;
| x[i] := x[i] + d[i];
| for j := i+1 to n do begin
| | d[j] := - d[j];
| end;
end;
```

□

Замечание. Для последовательностей нулей и единиц возможно другое решение, использующее двоичную систему. (Именно оно связывается обычно с названием «коды Грея».)

Запишем подряд все числа от 0 до $2^n - 1$ в двоичной системе. Например, для $n = 3$ напишем:

$$000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111$$

Затем каждое из чисел подвергнем преобразованию, заменив каждую цифру, кроме первой, на её сумму с предыдущей цифрой (по модулю 2). Иными словами, число $a_1, a_2, \ldots, a_n$ преобразуем в $a_1, a_1 + a_2, a_2 + a_3, \ldots$ $\ldots, a_{n-1} + a_n$ (сумма по модулю 2). Для $n = 3$ получим:

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100$$

Легко проверить, что описанное преобразование чисел обратимо (и тем самым даёт все последовательности по одному разу). Кроме того, двоичные записи соседних чисел отличаются заменой конца $011\ldots1$ на конец $100\ldots0$, что — после преобразования — приводит к изменению единственной цифры.

Применение кода Грея. Пусть есть вращающаяся ось, и мы хотим поставить датчик угла поворота этой оси. Насадим на ось барабан, выкрасим половину барабана в чёрный цвет, половину в белый и установим фотоэлемент. На его выходе будет в половине случаев 0, а в половине 1 (т. е. мы измеряем угол «с точностью до 180»).

Развёртка барабана:

← склеить бока

Сделав рядом другую дорожку из двух чёрных и белых частей и поставив второй фотоэлемент, получаем возможность измерить угол с точностью до 90°:

Сделав третью,

мы измерим угол с точностью до 45° и т. д. Эта идея имеет, однако, недостаток: в момент пересечения границ сразу несколько фотоэлементов меняют сигнал, и если эти изменения произойдут не совсем одновременно, на какое-то время показания фотоэлементов будут бессмысленными. Коды Грея позволяют избежать этой опасности. Сделаем так, чтобы на каждом шаге менялось показание лишь одного фотоэлемента (в том числе и на последнем, после целого оборота).

0 0 0 0 1 1 1 1
0 0 1 1 1 1 0 0
0 1 1 0 0 1 1 0

Написанная нами формула позволяет легко преобразовать данные от фотоэлементов в двоичный код угла поворота.

Заметим также, что геометрически существование кода Грея означает наличие «гамильтонова цикла» в n-мерном кубе (возможность обойти все вершины куба по разу, двигаясь по рёбрам, и вернуться в исходную вершину).

2.5.2. Напечатать все перестановки чисел `1..n` так, чтобы каждая следующая получалась из предыдущей перестановкой (транспозицией) двух соседних чисел. Например, при `n=3` допустим такой порядок:

`3.2 1` → `2 3.1` → `2.1 3` → `1 2.3` → `1.3 2` → `3 1 2`

(между переставляемыми числами вставлены точки).

Решение. Наряду с множеством перестановок рассмотрим множество последовательностей `y[1]..y[n]` целых неотрицательных чисел, для которых `y[1]` ⩽ `0`, ..., `y[n]` ⩽ `n-1`. В нём столько же элементов, сколько в множестве всех перестановок, и мы сейчас установим между ними взаимно однозначное соответствие. Именно, каждой перестановке поставим в соответствие последовательность `y[1]..y[n]`, где `y[i]` — количество чисел, меньших `i` и стоящих левее `i` в этой перестановке. Взаимная однозначность вытекает из такого замечания. Перестановка чисел `1..n` получается из перестановки чисел `1..n-1` добавлением числа `n`, которое можно вставить на любое из `n` мест. При этом к сопоставляемой с ней последовательности добавляется ещё один член, принимающий значения от `0` до `n-1`, а предыдущие члены не меняются. При этом оказывается, что изменение на единицу одного из членов последовательности `y` соответствует транспозиции двух соседних чисел,

если все следующие числа последовательности y принимают максимально или минимально возможные для них значения. Именно, увеличение y[i] на 1 соответствует транспозиции числа i с его правым соседом, а уменьшение — с левым.

Теперь вспомним решение задачи о перечислении всех последовательностей, на каждом шаге которого один член меняется на единицу. Заменив прямоугольную доску доской в форме лестницы (высота i-ой вертикали равна i) и двигая шашки по тем же правилам, мы перечислим все последовательности y, причём i-ый член будет меняться как раз только если все следующие шашки стоят у края. Надо ещё уметь параллельно с изменением y корректировать перестановку. Очевидный способ требует отыскания в ней числа i; это можно облегчить, если помимо самой перестановки хранить функцию

$$\texttt{i} \mapsto \text{позиция числа } \texttt{i} \text{ в перестановке},$$

т. е. обратное к перестановке отображение, и соответствующим образом её корректировать. Вот какая получается программа:

```
program test;
  const n=...;
  var
    x: array [1..n] of 1..n; {перестановка}
    inv_x: array [1..n] of 1..n; {обратная перестановка}
    y: array [1..n] of integer; {y[i] < i}
    d: array [1..n] of -1..1; {направления}
    b: boolean;

  procedure print_x;
    var i: integer;
  begin
    for i:=1 to n do begin
      write (x[i], ' ');
    end;
    writeln;
  end;

  procedure set_first;{первая: y[i]=0 при всех i}
    var i : integer;
  begin
    for i := 1 to n do begin
      x[i] := n + 1 - i;
      inv_x[i] := n + 1 - i;
      y[i]:=0;
```

```
    d[i]:=1;
  end;
end;

procedure move (var done : boolean);
  var i, j, pos1, pos2, val1, val2, tmp : integer;
begin
  i := n;
  while (i > 1) and (((d[i]=1) and (y[i]=i-1)) or
            ((d[i]=-1) and (y[i]=0))) do begin
    i := i-1;
  end;
  done := (i>1); {упрощение: первый член нельзя менять}
  if done then begin
    y[i] := y[i]+d[i];
    for j := i+1 to n do begin
      d[j] := -d[j];
    end;
    pos1 := inv_x[i];
    val1 := i;
    pos2 := pos1 + d[i];
    val2 := x[pos2];
    {pos1, pos2 - номера переставляемых элементов;
      val1, val2 - их значения; val2 < val1}
    tmp := x[pos1];
    x[pos1] := x[pos2];
    x[pos2] := tmp;
    tmp := inv_x[val1];
    inv_x[val1] := inv_x[val2];
    inv_x[val2] := tmp;
  end;
end;

begin
  set_first;
  print_x;
  b := true;
  {напечатаны все перестановки до текущей включительно;
    если b ложно, то текущая - последняя}
  while b do begin
    move (b);
    if b then print_x;
  end;
end.
```

□

2.6. Несколько замечаний

Посмотрим ещё раз на использованные нами приёмы. Вначале удавалось решить задачу по такой схеме: определяем порядок на подлежащих перечислению объектах и явно описываем процедуру перехода от данного объекта к следующему (в смысле этого порядка). В задаче о кодах Грея потребовалось хранить, помимо текущего объекта, и некоторую дополнительную информацию (направления стрелок). Наконец, в задаче о перечислении перестановок (на каждом шаге допустима одна транспозиция) мы применили такой приём: установили взаимно однозначное соответствие между перечисляемым множеством и другим, более просто устроенным. Таких соответствий в комбинаторике известно много. Мы приведём несколько задач, связанных с так называемыми «числами Каталана».

2.6.1. Перечислить все последовательности длины `2n`, составленные из `n` единиц и `n` минус единиц, у которых сумма любого начального отрезка неотрицательна, т. е. число минус единиц в нём не превосходит числа единиц. (Число таких последовательностей называют *числом Каталана*; формулу для чисел Каталана см. в следующем разделе.)

Решение. Изображая единицу вектором `(1,1)`, а минус единицу вектором `(1,-1)`, можно сказать, что мы ищем пути из точки `(0,0)` в точку `(n,0)`, не опускающиеся ниже оси абсцисс.

Будем перечислять последовательности в лексикографическом порядке, считая, что `-1` предшествует `1`. Первой последовательностью будет «пила»

```
1, -1, 1, -1, ...
```

а последней — «горка»

```
1, 1, 1,..., 1, -1, -1,..., -1.
```

Как перейти от последовательности к следующей? До некоторого места они должны совпадать, а затем надо заменить `-1` на `1`. Место замены должно быть расположено как можно правее. Но заменять `-1` на `1` можно только в том случае, если справа от неё есть единица (которую можно заменить на `-1`). После замены `-1` на `1` мы приходим к такой задаче: фиксирован начальный кусок последовательности, надо найти минимальное продолжение. Её решение: надо приписывать `-1`, если это не нарушит условия неотрицательности, а иначе приписывать `1`. Получаем

такую программу:

```
...
type array2n = array [1..2n] of integer;
...
procedure get_next (var a: array2n; var last: Boolean);
| {в a помещается следующая последовательность, если}
| {она есть (при этом last:=false), иначе last:=true}
| var k, i, sum: integer;
begin
| k:=2*n;
| {инвариант: в a[k+1..2n] только минус единицы}
| while a[k] = -1 do begin k:=k-1; end;
| {k - максимальное среди тех, для которых a[k]=1}
| while (k>0) and (a[k] = 1) do begin k:=k-1; end;
| {a[k] - самая правая -1, за которой есть 1;
|  если таких нет, то k=0}
| if k = 0 then begin
| | last := true;
| end else begin
| | last := false;
| | i:=0; sum:=0;
| | {sum = a[1]+...+a[i]}
| | while i<>k do begin
| | | i:=i+1; sum:= sum+a[i];
| | end;
| | {sum = a[1]+...+a[k], a[k]=-1}
| |  a[k]:= 1; sum:= sum+2;
| | {вплоть до a[k] всё изменено, sum=a[1]+...+a[k]}
| | while k <> 2*n do begin
| | | k:=k+1;
| | | if sum > 0 then begin
| | | | a[k]:=-1
| | | end else begin
| | | | a[k]:=1;
| | | end;
| | | sum:= sum+a[k];
| | end;
| | {k=2n, sum=a[1]+...a[2n]=0}
| end;
end;
```

□

2.6.2. Перечислить все расстановки скобок в произведении n сомножителей. Порядок сомножителей не меняется, скобки полностью опре-

деляют порядок действий. Например, для `n=4` есть 5 расстановок:

`((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd)).`

[Указание. Каждому порядку действий соответствует последовательность команд стекового калькулятора, описанного на с. 143.] □

2.6.3. На окружности задано `2n` точек, пронумерованных от 1 до `2n`. Перечислить все способы провести `n` непересекающихся хорд с вершинами в этих точках. □

2.6.4. Перечислить все способы разрезать `n`-угольник на треугольники, проведя `n-2` его диагонали. □

(Мы вернёмся к разрезанию многоугольника в разделе о динамическом программировании, с. 136.)

Ещё один класс задач на перечисление всех элементов заданного множества мы рассмотрим ниже, обсуждая метод поиска с возвратами (backtracking).

2.7. Подсчёт количеств

Иногда можно найти количество объектов с тем или иным свойством, не перечисляя их. Классический пример: C_n^k — число всех k-элементных подмножеств n-элементного множества — можно найти, заполняя таблицу по формулам

$$\begin{aligned} C_n^0 &= C_n^n = 1 && (n \geqslant 1) \\ C_n^k &= C_{n-1}^{k-1} + C_{n-1}^k && (n > 1, 0 < k < n) \end{aligned}$$

или по формуле

$$C_n^k = \frac{n!}{k! \cdot (n-k)!}.$$

(Первый способ эффективнее, если надо вычислить много значений C_n^k.)

Приведём другие примеры.

2.7.1. (Число разбиений; предлагалась на Всесоюзной олимпиаде по программированию 1988 года) Пусть $P(n)$ — число разбиений целого положительного n на целые положительные слагаемые (без учёта порядка, $1+2$ и $2+1$ — одно и то же разбиение). При $n=0$ положим $P(n)=1$ (единственное разбиение не содержит слагаемых). Построить алгоритм вычисления $P(n)$ для заданного n.

Решение. Можно доказать (это нетривиально) такую формулу для $P(n)$:

$$P(n)=P(n-1)+P(n-2)-P(n-5)-P(n-7)+P(n-12)+P(n-15)+\ldots$$

(знаки у пар членов чередуются, вычитаемые в одной паре равны $(3q^2-q)/2$ и $(3q^2+q)/2$; сумма конечна — мы считаем, что $P(k)=0$ при $k<0$).

Однако и без её использования можно придумать способ вычисления $P(n)$, который существенно эффективнее перебора и подсчёта всех разбиений.

Обозначим через $R(n,k)$ (для $n\geqslant 0$, $k\geqslant 0$) число разбиений n на целые положительные слагаемые, не превосходящие k. (При этом $R(0,k)$ считаем равным 1 для всех $k\geqslant 0$.) Очевидно, $P(n)=R(n,n)$. Все разбиения n на слагаемые, не превосходящие k, разобьём на группы в зависимости от максимального слагаемого (обозначим его i). Число $R(n,k)$ равно сумме (по всем i от 1 до k) количеств разбиений со слагаемыми не больше k и максимальным слагаемым, равным i. А разбиения n на слагаемые не более k с первым слагаемым, равным i, по существу представляют собой разбиения $n-i$ на слагаемые, не превосходящие i (при $i\leqslant k$). Так что

$$R(n,k)=\sum_{i=1}^{k}R(n-i,i) \qquad \text{при } k\leqslant n,$$
$$R(n,k)=R(n,n) \qquad \text{при } k\geqslant n,$$

что позволяет заполнять таблицу значений функции R. □

2.7.2. (Счастливые билеты; предлагалась на Всесоюзной олимпиаде по программированию 1989 года.) Последовательность из $2n$ цифр (каждая цифра от 0 до 9) называется счастливым билетом, если сумма первых n цифр равна сумме последних n цифр. Найти число счастливых последовательностей данной длины.

Решение. (Сообщено одним из участников олимпиады; к сожалению, не могу указать фамилию, так как работы проверялись зашифрованными.) Рассмотрим более общую задачу: найти число последовательностей, где разница между суммой первых n цифр и суммой последних n цифр равна k ($k=-9n,\ldots,9n$). Пусть $T(n,k)$ — число таких последовательностей.

Разобьём множество таких последовательностей на классы в зависимости от разницы между первой и последней цифрами. Если эта разница равна t, то разница между суммами групп из оставшихся $n-1$ цифр

равна $k-t$. Учитывая, что пар цифр с разностью t бывает $10-|t|$, получаем формулу

$$T(n,k)=\sum_{t=-9}^{9}(10-|t|)T(n-1,k-t).$$

(Некоторые слагаемые могут отсутствовать, так как $k-t$ может быть слишком велико.) □

В некоторых случаях ответ удаётся получить в виде явной формулы.

2.7.3. Доказать, что число Каталана (количество последовательностей длины $2n$ из n единиц и n минус единиц, в любом начальном отрезке которых не меньше единиц, чем минус единиц) равно $C_{2n}^n/(n+1)$.

[Указание. Число Каталана есть число ломаных, идущих из $(0,0)$ в $(2n,0)$ шагами $(1,1)$ и $(1,-1)$, не опускающихся в нижнюю полуплоскость, т. е. разность числа всех ломаных (которое есть C_{2n}^n) и числа ломаных, опускающихся в нижнюю полуплоскость. Последние можно описать также как ломаные, пересекающие прямую $y=-1$. Отразив их кусок справа от самой правой точки пересечения относительно указанной прямой, мы установим взаимно однозначное соответствие между ними и ломаными из $(0,0)$ в $(2n,-2)$. Остаётся проверить, что $C_{2n}^n-C_{2n}^{n+1}=$ $=C_{2n}^n/(n+1)$.] □

3. ОБХОД ДЕРЕВА. ПЕРЕБОР С ВОЗВРАТАМИ

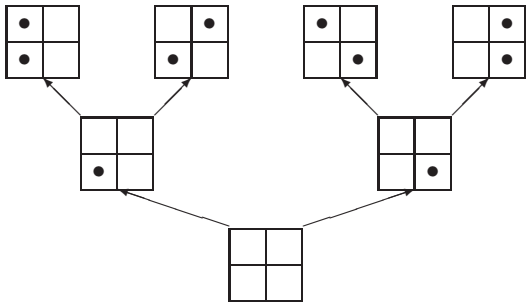
3.1. Ферзи, не бьющие друг друга: обход дерева позиций

В предыдущей главе мы рассматривали несколько задач одного и того же типа: «перечислить все элементы некоторого множества A». Схема решения была такова: на множестве A вводился порядок и описывалась процедура перехода от произвольного элемента множества A к следующему за ним (в этом порядке). Такую схему не всегда удаётся реализовать непосредственно, и в этой главе мы рассмотрим другой полезный приём перечисления всех элементов некоторого множества. Его называют «поиск с возвратами», «метод ветвей и границ», «backtracking». На наш взгляд, наиболее точное название этого метода — обход дерева.

3.1.1. Перечислить все способы расстановки n ферзей на шахматной доске $n \times n$, при которых они не бьют друг друга.

Решение. Очевидно, на каждой из n горизонталей должно стоять по ферзю. Будем называть k-позицией (для $k = 0, 1, \ldots, n$) произвольную расстановку k ферзей на k нижних горизонталях (ферзи могут бить друг друга). Нарисуем «дерево позиций»: его корнем будет единственная 0-позиция, а из каждой k-позиции выходит n стрелок вверх в $(k+1)$-позиции. Эти n позиций отличаются положением ферзя на $(k+1)$-ой горизонтали. Будем считать, что расположение их на рисунке соответствует положению этого ферзя: левее та позиция, в которой ферзь расположен левее.

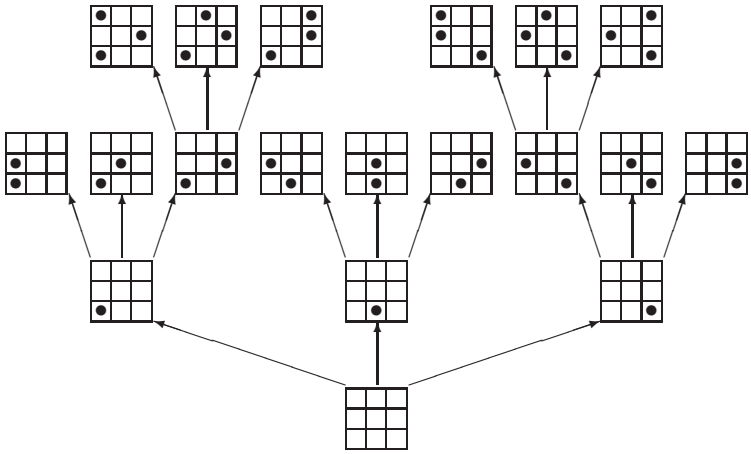
Среди позиций этого дерева нам надо отобрать те n-позиции, в которых ферзи не бьют друг друга. Программа будет «обходить дерево» и искать их. Чтобы не делать лишней работы, заметим вот что: если в какой-то k-позиции ферзи бьют друг друга, то ставить дальнейших



Дерево позиций для $n = 2$

ферзей смысла нет. Поэтому, обнаружив это, мы будем прекращать построение дерева в этом направлении.

Точнее, назовём k-позицию допустимой, если *после удаления верхнего ферзя* оставшиеся не бьют друг друга. Наша программа будет рассматривать только допустимые позиции.



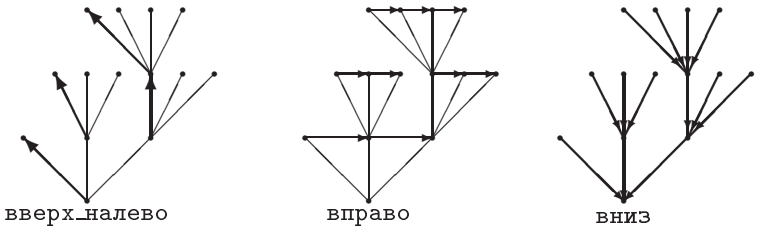
Дерево допустимых позиций для $n = 3$

Разобьём задачу на две части: (1) обход произвольного дерева и (2) реализацию дерева допустимых позиций.

Сформулируем задачу обхода произвольного дерева. Будем считать, что у нас имеется Робот, который в каждый момент находится в одной из вершин дерева (вершины изображены на рисунке кружочками). Он умеет выполнять команды:

- `вверх_налево` (идти по самой левой из выходящих вверх стрелок)
- `вправо` (перейти в соседнюю справа вершину)
- `вниз` (спуститься вниз на один уровень)

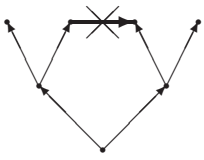
(На рисунках стрелками показано, какие перемещения соответствуют этим командам.)



Кроме того, в репертуар Робота входят проверки (соответствующие возможности выполнить каждую из команд):

- `есть_сверху`;
- `есть_справа`;
- `есть_снизу`;

(последняя проверка истинна всюду, кроме корня). Обратите внимание, что команда `вправо` позволяет перейти лишь к «родному брату», но не к «двоюродному».

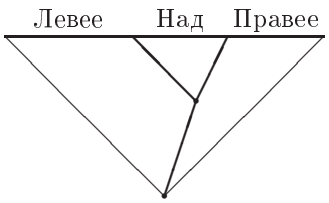


Так команда
`вправо`
не действует!

Будем считать, что у Робота есть команда `обработать` и что его задача — обработать все листья (вершины, из которых нет стрелок

вверх, то есть где условие `есть_сверху` ложно). Для нашей шахматной задачи команде `обработать` будет соответствовать проверка и печать позиции ферзей.

Доказательство правильности приводимой далее программы использует такие определения. Пусть фиксировано положение Робота в одной из вершин дерева. Тогда все листья дерева разбиваются на три категории: *над* Роботом, *левее* Робота и *правее* Робота. (Путь из корня в лист может проходить через вершину с Роботом, сворачивать влево, не доходя до неё и сворачивать вправо, не доходя до неё.) Через `(ОЛ)` обозначим условие «обработаны все листья левее Робота», а через `(ОЛН)` — условие «обработаны все листья левее и над Роботом».



Нам понадобится такая процедура:

```
procedure вверх_до_упора_и_обработать;
| {дано: (ОЛ), надо: (ОЛН)}
begin
| {инвариант: ОЛ}
| while есть_сверху do begin
| | вверх_налево;
| end
| {ОЛ, Робот в листе}
| обработать;
| {ОЛН}
end;
```

Основной алгоритм:

```
дано: Робот в корне, листья не обработаны
надо: Робот в корне, листья обработаны

{ОЛ}
вверх_до_упора_и_обработать;
{инвариант: ОЛН}
```

```
while есть_снизу do begin
| if есть_справа then begin {ОЛН, есть справа}
| | вправо;
| | {ОЛ}
| | вверх_до_упора_и_обработать;
| end else begin
| | {ОЛН, не есть_справа, есть_снизу}
| | вниз;
| end;
end;
{ОЛН, Робот в корне => все листья обработаны}
```

Осталось воспользоваться следующими свойствами команд Робота (в каждой строке в первой фигурной скобке записаны условия, в которых выполняется команда, во второй — утверждения о результате её выполнения):

(1) {ОЛ, не `есть_сверху`} `обработать` {ОЛН}
(2) {ОЛ, `есть_сверху`} `вверх_налево` {ОЛ}
(3) {`есть_справа`, ОЛН} `вправо` {ОЛ}
(4) {не `есть_справа`, `есть_снизу`, ОЛН} `вниз` {ОЛН}

3.1.2. Доказать, что приведённая программа завершает работу (на любом конечном дереве).

Решение. Процедура `вверх_до_упора_и_обработать` завершает работу (высота Робота не может увеличиваться бесконечно). Если программа работает бесконечно, то, поскольку листья не обрабатываются повторно, начиная с некоторого момента ни один лист не обрабатывается. А это возможно, только если Робот всё время спускается вниз. Противоречие. (Об оценке числа действий см. далее.) □

3.1.3. Доказать правильность следующей программы обхода дерева:

```
var state: (WL, WLU);
state := WL;
while есть_снизу or (state <> WLU) do begin
| if (state = WL) and есть_сверху then begin
| | вверх_налево;
| end else if (state = WL) and not есть_сверху then begin
| | обработать; state := WLU;
| end else if (state = WLU) and есть_справа then begin
| |  вправо; state := WL;
| end else begin {state = WLU, not есть_справа, есть_снизу}
| |  вниз;
| end;
end;
```

Решение. Инвариант цикла:

$$\texttt{state} = \texttt{WL} \Rightarrow \texttt{ОЛ}$$
$$\texttt{state} = \texttt{WLU} \Rightarrow \texttt{ОЛН}$$

Доказательство завершения работы: переход из состояния ОЛ в ОЛН возможен только при обработке вершины, поэтому если программа работает бесконечно, то с некоторого момента значение state не меняется, что невозможно. □

3.1.4. Написать программу обхода дерева, использующую процедуру перехода в следующий лист (с выходным параметром, сообщающим, удалось ли это сделать или лист оказался последним). □

3.1.5. Решить задачу об обходе дерева, если мы хотим, чтобы обрабатывались все вершины (не только листья).

Решение. Пусть x — некоторая вершина. Тогда любая вершина y относится к одной из четырёх категорий. Рассмотрим путь из корня в y. Он может:
(а) быть частью пути из корня в x (y *ниже* x);
(б) свернуть налево с пути в x (y *левее* x);
(в) пройти через x (y *над* x);
(г) свернуть направо с пути в x (y *правее* x);
В частности, сама вершина x относится к категории (в). Условия теперь будут такими:

(ОНЛ) обработаны все вершины ниже и левее;

(ОНЛН) обработаны все вершины ниже, левее и над.

Вот как будет выглядеть программа:

```
procedure вверх_до_упора_и_обработать;
| {дано: (ОНЛ), надо: (ОНЛН)}
begin
| {инвариант: ОНЛ}
| while есть_сверху do begin
| | обработать;
| | вверх_налево;
| end
| {ОНЛ, Робот в листе}
| обработать;
| {ОНЛН}
end;
```

Основной алгоритм:

```
дано: Робот в корне, ничего не обработано
надо: Робот в корне, все вершины обработаны

{ОНЛ}
вверх_до_упора_и_обработать;
{инвариант: ОНЛН}
while есть_снизу do begin
| if есть_справа then begin {ОНЛН, есть справа}
| | вправо;
| | {ОНЛ}
| | вверх_до_упора_и_обработать;
| end else begin
| | {ОЛН, не есть_справа, есть_снизу}
| | вниз;
| end;
end;
{ОНЛН, Робот в корне => все вершины обработаны}
```
□

3.1.6. Приведённая только что программа обрабатывает вершину до того, как обработан любой из её потомков. Как изменить программу, чтобы каждая вершина, не являющаяся листом, обрабатывалась дважды: один раз до, а другой раз после всех своих потомков? (Листья по-прежнему обрабатываются по разу.)

Решение. Под «обработано ниже и левее» будем понимать «ниже обработано по разу, слева обработано полностью (листья по разу, остальные по два)». Под «обработано ниже, левее и над» будем понимать «ниже обработано по разу, левее и над — полностью».

Программа будет такой:

```
procedure вверх_до_упора_и_обработать;
| {дано: (ОНЛ), надо: (ОНЛН)}
begin
| {инвариант: ОНЛ}
| while есть_сверху do begin
| | обработать;
| | вверх_налево;
| end
| {ОНЛ, Робот в листе}
| обработать;
| {ОНЛН}
end;
```

Основной алгоритм:

```
дано: Робот в корне, ничего не обработано
надо: Робот в корне, все вершины обработаны

{ОНЛ}
вверх_до_упора_и_обработать;
{инвариант: ОНЛН}
while есть_снизу do begin
| if есть_справа then begin {ОНЛН, есть справа}
| | вправо;
| | {ОНЛ}
| | вверх_до_упора_и_обработать;
| end else begin
| | {ОЛН, не есть_справа, есть_снизу}
| | вниз;
| | обработать;
| end;
end;
{ОНЛН, Робот в корне => все вершины обработаны полностью}   □
```

3.1.7. Доказать, что число операций в этой программе по порядку равно числу вершин дерева. (Как и в других программах, которые отличаются от этой лишь пропуском некоторых команд `обработать`.)

[Указание. Примерно каждое второе действие при исполнении этой программы — обработка вершины, а каждая вершина обрабатывается максимум дважды.] □

Вернёмся теперь к нашей задаче о ферзях (где из всех программ обработки дерева понадобится лишь первая, самая простая). Реализуем операции с деревом позиций. Позицию будем представлять с помощью переменной `k: 0..n` (число ферзей) и массива `c: array[1..n] of 1..n` (`c[i]` — координаты ферзя на `i`-ой горизонтали; при $i > k$ значение `c[i]` роли не играет). Предполагается, что все позиции допустимы (если убрать верхнего ферзя, остальные не бьют друг друга).

```
program queens;
| const n = ...;
| var
|   k: 0..n;
|   c: array [1..n] of 1..n;
|
```

```
procedure begin_work; {начать работу}
begin
  k := 0;
end;

function danger: boolean; {верхний ферзь под боем}
  var b: boolean; i: integer;
begin
  if k <= 1 then begin
    danger := false;
  end else begin
    b := false;
    i := 1;
    {b <=> верхний ферзь под боем ферзей с номерами < i}
    while i <> k do begin
      b := b or (c[i]=c[k]) {вертикаль}
          or (abs(c[i]-c[k]))=abs(i-k)); {диагональ}
      i := i+1;
    end;
    danger := b;
  end;
end;

function is_up: boolean; {есть_сверху}
begin
  is_up := (k < n) and not danger;
end;

function is_right: boolean; {есть_справа}
begin
  is_right := (k > 0) and (c[k] < n);
end;
{возможна ошибка: при k=0 не определено c[k]}

function is_down: boolean; {есть_снизу}
begin
  is_down := (k > 0);
end;

procedure up; {вверх_налево}
begin {k < n, not danger}
  k := k + 1;
  c [k] := 1;
end;
```

```
|
| procedure right; {вправо}
| begin {k > 0,  c[k] < n}
| | c [k] := c [k] + 1;
| end;
|
|
| procedure down; {вниз}
| begin {k > 0}
| | k := k - 1;
| end;
|
| procedure work; {обработать}
| | var i: integer;
| begin
| | if (k = n) and not danger then begin
| | | for i := 1 to n do begin
| | | | write ('<', i, ',' , c[i], '> ');
| | | end;
| | | writeln;
| | end;
| end;
|
|
| procedure UW; {вверх_до_упора_и_обработать}
| begin
| | while is_up do begin
| | | up;
| | end;
| | work;
| end;
|
begin
| begin_work;
| UW;
| while is_down do begin
| | if is_right then begin
| | | right;
| | | UW;
| | end else begin
| | | down;
| | end;
| end;
end.
```

□

3.1.8. Приведённая программа тратит довольно много времени на выполнение проверки `есть_сверху` (проверка, находится ли верхний ферзь под боем, требует числа действий порядка `n`). Изменить реализацию операций с деревом позиций так, чтобы все три проверки `есть_сверху/справа/снизу` и соответствующие команды требовали бы количества действий, ограниченного не зависящей от `n` константой.

Решение. Для каждой вертикали, каждой восходящей и каждой нисходящей диагонали будем хранить булевское значение — сведения о том, находится ли на этой линии ферзь (верхний ферзь не учитывается). (Заметим, что в силу допустимости позиции на каждой из линий может быть не более одного ферзя.) □

3.2. Обход дерева в других задачах

3.2.1. Использовать метод обхода дерева для решения следующей задачи: дан массив из `n` целых положительных чисел `a[1]` ... `a[n]` и число `s`; требуется узнать, может ли число `s` быть представлено как сумма некоторых из чисел массива `a`. (Каждое число можно использовать не более чем по одному разу.)

Решение. Будем задавать `k`-позицию последовательностью из `k` булевских значений, определяющих, входят ли в сумму числа `a[1]` ... `a[k]` или не входят. Позиция допустима, если её сумма не превосходит `s`. □

Замечание. По сравнению с полным перебором всех $2^{\mathtt{n}}$ подмножеств тут есть некоторый выигрыш. Можно также предварительно отсортировать массив `a` в убывающем порядке, а также считать недопустимыми те позиции, в которых сумма отброшенных членов больше, чем разность суммы всех членов и `s`. Последний приём называют «методом ветвей и границ». Но принципиального улучшения по сравнению с полным перебором тут не получается (эта задача, как говорят, NP-полна, подробности см. в книге Ахо, Хопкрофта и Ульмана «Построение и анализ вычислительных алгоритмов», Мир, 1979, а также в книге Гэри и Джонсона «Вычислительные машины и труднорешаемые задачи», Мир, 1982). Традиционное название этой задачи — «задача о рюкзаке» (рюкзак общей грузоподъёмностью `s` нужно упаковать под завязку, располагая предметами веса `a[1]` ... `a[n]`). См. также в главе 8 (Как обойтись без рекурсии) алгоритм её решения, полиномиальный по $\mathtt{n}+\mathtt{s}$ (использующий «динамическое программирование»).

3.2.2. Перечислить все последовательности из n нулей, единиц и двоек, в которых никакая группа цифр не повторяется два раза подряд (нет куска вида XX). □

3.2.3. Аналогичная задача для последовательностей нулей и единиц, в которых никакая группа цифр не повторяется три раза подряд (нет куска вида XXX). □

К этой же категории относятся задачи типа «можно ли сложить данную фигуру из пентамино» и им подобные. В них важно умелое сокращение перебора (вовремя распознать, что имеющееся расположение фигурок уже противоречит требованиям, и по этой ветви поиск не продолжать).

4. СОРТИРОВКА

4.1. Квадратичные алгоритмы

4.1.1. Пусть a[1], ..., a[n] — целые числа. Требуется построить массив b[1], ..., b[n], содержащий те же числа, для которого b[1] $\leqslant$ ... ... $\leqslant$ b[n].

Замечание. Среди чисел a[1] ... a[n] могут быть равные. Требуется, чтобы каждое целое число входило в b[1] ... b[n] столько же раз, сколько и в a[1] ... a[n].

Решение. Удобно считать, что числа a[1] ... a[n] и b[1] ... b[n] представляют собой начальное и конечное значения массива x. Требование «a и b содержат одни и те же числа» будет заведомо выполнено, если в процессе работы мы ограничимся перестановками элементов x.

```
k := 0;
{k наименьших элементов массива установлены на свои места}
while k <> n do begin
  s := k + 1; t := k + 1;
  {x[s] - наименьший среди x[k+1]...x[t] }
  while t<>n do begin
    t := t + 1;
    if x[t] < x[s] then begin
      s := t;
    end;
  end;
  {x[s] - наименьший среди x[k+1]..x[n] }
  ... переставить x[s] и x[k+1];
  k := k + 1;
end;
```

□

4.1.2. Дать другое решение задачи сортировки, использующее инвариант «первые k элементов упорядочены» (x[1] $\leqslant$ ... $\leqslant$ x[k]).

Решение.

```
k:=1;
{первые k элементов упорядочены}
while k <> n do begin
| t := k+1;
| {k+1-ый элемент продвигается к началу, пока не займёт
|   надлежащего места, t - его текущий номер}
| while (t > 1) and (x[t] < x[t-1]) do begin
| | ...поменять x[t-1] и x[t];
| | t := t - 1;
| end;
end;
```

□

Замечание. Дефект программы: при ложном выражении `(t>1)` проверка `x[t]` < `x[t-1]` требует несуществующего значения `x[0]`.

Оба предложенных решения требуют числа действий, пропорционального n^2. Существуют более эффективные алгоритмы.

4.2. Алгоритмы порядка $n \log n$

4.2.1. Предложить алгоритм сортировки за время $n \log n$ (число операций при сортировке n элементов не больше $Cn \log n$ для некоторого C и для всех n).

Мы предложим два решения.

Решение 1 (сортировка слиянием).

Пусть `k` — положительное целое число. Разобьём массив `x[1]` ... `x[n]` на отрезки длины `k`. (Первый — `x[1]` ... `x[k]`, затем `x[k+1]` ... `x[2k]` и так далее.) Последний отрезок будет неполным, если `n` не делится на `k`. Назовём массив `k`-упорядоченным, если каждый из этих отрезков в отдельности упорядочен. Любой массив 1-упорядочен. Если массив `k`-упорядочен и $n \leqslant k$, то он упорядочен.

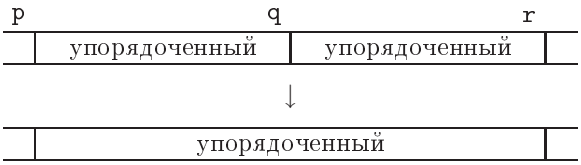
Мы опишем, как преобразовать `k`-упорядоченный массив в `2k`-упорядоченный (из тех же элементов). С помощью этого преобразования алгоритм записывается так:

```
k:=1;
{массив x является k-упорядоченным}
while k < n do begin
| ...преобразовать k-упорядоченный массив в 2k-упорядоченный;
| k := 2 * k;
end;
```

Требуемое преобразование состоит в том,что мы многократно «сливаем» два упорядоченных отрезка длины не больше k в один упорядоченный отрезок. Пусть процедура

```
слияние (p,q,r: integer)
```

при $p \leqslant q \leqslant r$ сливает отрезки x[p+1] ... x[q] и x[q+1] ... x[r] в упорядоченный отрезок x[p+1] ... x[r] (не затрагивая других частей массива x).



Тогда преобразование k-упорядоченного массива в 2k-упорядоченный осуществляется так:

```
t:=0;
{t кратно 2k или t = n, x[1]..x[t] является
 2k-упорядоченным; остаток массива x не изменился}
while t + k < n do begin
| p := t;
| q := t+k;
| r := min (t+2*k, n);
| {min(a,b) - минимум из a и b}
| слияние (p,q,r);
| t := r;
end;
```

Слияние требует вспомогательного массива для записи результатов слияния — обозначим его b. Через p0 и q0 обозначим номера последних элементов участков, подвергшихся слиянию, s0 — последний записанный в массив b элемент. На каждом шаге слияния производится одно из двух действий:

```
b[s0+1]:=x[p0+1];
p0:=p0+1;
s0:=s0+1;
```

или

```
b[s0+1]:=x[q0+1];
q0:=q0+1;
s0:=s0+1;
```

(Любители языка C написали бы в этом случае `b[++s0]=x[++p0]` и `b[++s0]=x[++q0]`.)

Первое действие (взятие элемента из первого отрезка) может производиться при одновременном выполнении двух условий:

(1) первый отрезок не кончился (`p0` $<$ `q`);

(2) второй отрезок кончился (`q0` $=$ `r`) или не кончился, но элемент в нём не меньше очередного элемента первого отрезка [(`q0` $<$ `r`) и(`x[p0+1]` $\leqslant$ `x[q0+1]`)].

Аналогично для второго действия. Итак, получаем

```
p0 := p; q0 := q; s0 := p;
while (p0 <> q) or (q0 <> r) do begin
| if (p0 < q) and ((q0 = r) or ((q0 < r) and
| |                 (x[p0+1] <= x[q0+1]))) then begin
| | b [s0+1] := x [p0+1];
| | p0 := p0+1;
| | s0 := s0+1;
| end else begin
| | {(q0 < r) and ((p0 = q) or ((p0<q) and
| |   (x[p0+1] >= x[q0+1])))}
| | b [s0+1] := x [q0+1];
| | q0 := q0 + 1;
| | s0 := s0 + 1;
| end;
end;
```

(Если оба отрезка не кончены и первые невыбранные элементы в них равны, то допустимы оба действия; в программе выбрано первое.)

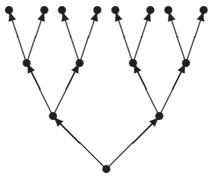
Остаётся лишь переписать результат слияния обратно в массив `x`. (Предупреждение. Если обратное копирование выполняется вне процедуры слияния, то не забудьте про последний отрезок.)

Программа имеет привычный дефект: обращение к несуществующим элементам массива при вычислении булевских выражений.

Решение 2 (сортировка деревом).

Нарисуем «полное двоичное дерево» — картинку, в которой снизу один кружок, из него выходят стрелки в два других, из каждого —

в два других и так далее:



Будем говорить, что стрелки ведут «от отцов к сыновьям»: у каждого кружка два сына и один отец (если кружок не в самом верху или низу). Предположим для простоты, что количество подлежащих сортировке чисел есть степень двойки, и они могут заполнить один из рядов целиком. Запишем их туда. Затем заполним часть дерева под ними по правилу:

число в кружке = минимум из чисел в кружка́х-сыновьях

Тем самым в корне дерева (нижнем кружке) будет записано минимальное число во всём массиве.

Изымем из сортируемого массива минимальный элемент. Для этого его надо вначале найти. Это можно сделать, идя от корня: от отца переходим к тому сыну, где записано то же число. Изъяв минимальный элемент, заменим его символом $+\infty$ и скорректируем более низкие ярусы (для этого надо снова пройти путь к корню). При этом считаем, что $\min(t, +\infty) = t$. Тогда в корне появится второй по величине элемент, мы изымаем его, заменяя бесконечностью и корректируя дерево. Так постепенно мы изымем все элементы в порядке возрастания, пока в корне не останется бесконечность.

При записи этого алгоритма полезно нумеровать кружки числами $1, 2, \ldots$ — при этом сыновьями кружка номер `n` являются кружки `2n` и `2n + 1`. Подробное изложение этого алгоритма мы опустим, поскольку мы изложим более эффективный вариант, не требующий дополнительной памяти, кроме конечного числа переменных (в дополнение к сортируемому массиву).

Мы будем записывать сортируемые числа во всех вершинах дерева, а не только на верхнем уровне. Пусть `x[1]` ... `x[n]` — массив, подлежащий сортировке. Вершинами дерева будут числа от `1` до `n`; о числе `x[i]` мы будем говорить как о числе, стоящем в вершине `i`. В процессе сортировки количество вершин дерева будет сокращаться. Число вершин текущего дерева будем хранить в пе-

ременной k. Таким образом, в процессе работы алгоритма массив x[1]...x[n] делится на две части: в x[1]...x[k] хранятся числа на дереве, а в x[k+1]...x[n] хранится уже отсортированная в порядке возрастания часть массива — элементы, уже занявшие своё законное место.

На каждом шаге алгоритм будет изымать максимальный элемент дерева и помещать его в отсортированную часть, на освободившееся в результате сокращения дерева место.

Договоримся о терминологии. Вершинами дерева считаются числа от 1 до текущего значения переменной k. У каждой вершины s могут быть *сыновья* 2s и 2s+1. Если оба этих числа больше k, то сыновей нет; такая вершина называется *листом*. Если 2s = k, то вершина s имеет ровно одного сына (2s).

Для каждого s из 1...k рассмотрим «поддерево» с корнем в s: оно содержит вершину s и всех её потомков (сыновей, внуков и так далее — до тех пор, пока мы не выйдем из отрезка 1...k). Вершину s будем называть *регулярной*, если стоящее в ней число — максимальный элемент s-поддерева; s-поддерево назовём *регулярным*, если все его вершины регулярны. (В частности, любой лист образует регулярное одноэлементное поддерево.)

Заметим, что истинность утверждения «s-поддерево регулярно» зависит не только от s, но от текущего значения k.

Схема алгоритма такова:

```
k:= n
... Сделать 1-поддерево регулярным;
{x[1],..,x[k] <= x[k+1] <=..<= x[n]; 1-поддерево регулярно,
 в частности, x[1] - максимальный элемент среди x[1]..x[k]}
while k <> 1 do begin
| ... обменять местами x[1] и x[k];
| k := k - 1;
| {x[1]..x[k-1] <= x[k] <=...<= x[n]; 1-поддерево
|   регулярно везде, кроме, возможно, самого корня }
| ... восстановить регулярность 1-поддерева всюду
end;
```

В качестве вспомогательной процедуры нам понадобится процедура восстановления регулярности s-поддерева в корне. Вот она:

```
{s-поддерево регулярно везде, кроме, возможно, корня}
t := s;
{s-поддерево регулярно везде, кроме, возможно, вершины t}
```

```
while ((2*t+1 <= k) and (x[2*t+1] > x[t])) or
      ((2*t <= k) and (x[2*t] > x[t])) do begin
  if (2*t+1 <= k) and (x[2*t+1] >= x[2*t]) then begin
    ... обменять x[t] и x[2*t+1];
    t := 2*t + 1;
  end else begin
    ... обменять x[t] и x[2*t];
    t := 2*t;
  end;
end;
```

Чтобы убедиться в правильности этой процедуры, посмотрим на неё повнимательнее. Пусть в s-поддереве все вершины, кроме разве что вершины t, регулярны. Рассмотрим сыновей вершины t. Они регулярны, и потому содержат наибольшие числа в своих поддеревьях. Таким образом, на роль наибольшего числа в t-поддереве могут претендовать число в самой вершине t и числа в её сыновьях. (В первом случае вершина t регулярна, и всё в порядке.) В этих терминах цикл можно записать так:

```
while наибольшее число не в t, а в одном из сыновей do begin
  if оно в правом сыне then begin
    поменять t с её правым сыном; t:= правый сын
  end else begin {наибольшее число - в левом сыне}
    поменять t с её левым сыном; t:= левый сын
  end
end
```

После обмена вершина t становится регулярной (в неё попадает максимальное число t-поддерева). Не принявший участия в обмене сын остаётся регулярным, а принявший участие может и не быть регулярным. В остальных вершинах s-поддерева не изменились ни числа, ни поддеревья их потомков (разве что два элемента поддерева переставились), так что регулярность не нарушилась.

Эта же процедура может использоваться для того, чтобы сделать 1-поддерево регулярным на начальной стадии сортировки:

```
k := n; u := n;
{все s-поддеревья с s>u регулярны }
while u<>0 do begin
  {u-поддерево регулярно везде, кроме разве что корня}
  ... восстановить регулярность u-поддерева в корне;
  u:=u-1;
end;
```

Теперь запишем процедуру сортировки на паскале (предполагая, что n — константа, x имеет тип arr = array [1..n] of integer).

```
procedure sort (var x: arr);
| var u, k: integer;
| procedure exchange(i, j: integer);
| | var tmp: integer;
| | begin
| | tmp  := x[i];
| | x[i] := x[j];
| | x[j] := tmp;
| end;
| procedure restore (s: integer);
| | var t: integer;
| | begin
| | t:=s;
| | while ((2*t+1 <= k) and (x[2*t+1] > x[t])) or
| | |     ((2*t <= k) and (x[2*t] > x[t])) do begin
| | | if (2*t+1 <= k) and (x[2*t+1] >= x[2*t]) then begin
| | | | exchange (t, 2*t+1);
| | | | t := 2*t+1;
| | | end else begin
| | | | exchange (t, 2*t);
| | | | t := 2*t;
| | | end;
| | end;
| end;
begin
| k:=n;
| u:=n;
| while u <> 0 do begin
| | restore (u);
| | u := u - 1;
| end;
| while k <> 1 do begin
| | exchange (1, k);
| | k := k - 1;
| | restore (1);
| end;
end;
```

□

Несколько замечаний.

Метод, использованный при сортировке деревом, бывает полезным в других случаях. (См. в главе 6 (Типы данных) об очереди с приоритетами.)

Сортировка слиянием хороша тем, что она на требует, чтобы весь сортируемый массив помещался в оперативной памяти. Можно сначала отсортировать такие куски, которые помещаются в памяти (например, с помощью дерева), а затем сливать полученные файлы.

Ещё один практически важный алгоритм сортировки (быстрая сортировка Хоара) таков: чтобы отсортировать массив, выберем случайный его элемент b, и разобьём массив на три части: меньшие b, равные b и большие b. (Эта задача приведена в главе 1.) Теперь осталось отсортировать первую и третью части: это делается тем же способом. Время работы этого алгоритма — случайная величина; можно доказать, что в среднем он работает не больше $Cn\log n$. На практике — он один из самых быстрых. (Мы ещё вернёмся к нему, приведя его рекурсивную и нерекурсивную реализации.)

Наконец, отметим, что сортировка за время порядка $Cn\log n$ может быть выполнена с помощью техники сбалансированных деревьев (см. главу 14), однако программы тут сложнее и константа C довольно велика.

4.3. Применения сортировки.

4.3.1. Найти количество различных чисел среди элементов данного массива. Число действий порядка $n\log n$. (Эта задача уже была в главе 1.)

Решение. Отсортировать числа, а затем посчитать количество различных, просматривая элементы массива по порядку. □

4.3.2. Дано `n` отрезков `[a[i],b[i]]` на прямой (`i` = 1...`n`). Найти максимальное `k`, для которого существует точка прямой, покрытая `k` отрезками («максимальное число слоёв»). Число действий — порядка `n` log `n`.

Решение. Упорядочим все левые и правые концы отрезков вместе (при этом левый конец считается меньше правого конца, расположенного в той же точке прямой). Далее двигаемся слева направо, считая число слоёв. Встреченный левый конец увеличивает число слоёв на 1, правый — уменьшает. Отметим, что примыкающие друг к другу отрезки обрабатываются правильно: сначала идёт левый конец (правого отрезка), а затем — правый (левого отрезка). □

4.3.3. Дано n точек на плоскости. Указать $(n-1)$-звенную несамопересекающуюся незамкнутую ломаную, проходящую через все эти

точки. (Соседним отрезкам ломаной разрешается лежать на одной прямой.) Число действий порядка $n \log n$.

Решение. Упорядочим точки по x-координате, а при равных x-координатах — по y-координате. В таком порядке и можно проводить ломаную. □

4.3.4. Та же задача, если ломаная должна быть замкнутой.

Решение. Возьмём самую левую точку (то есть точку с наименьшей x-координатой) и проведём из неё лучи во все остальные точки. Теперь упорядочим эти лучи снизу вверх, а точки на одном луче упорядочим по расстоянию от начала луча (это делается для всех лучей, кроме нижнего и верхнего). Ломаная выходит из выбранной (самой левой) точки по нижнему лучу, затем по всем остальным лучам (в описанном порядке) и возвращается по верхнему лучу. □

4.3.5. Дано n точек на плоскости. Построить их выпуклую оболочку — минимальную выпуклую фигуру, их содержащую. (Резиновое колечко, натянутое на вбитые в доску гвозди — их выпуклая оболочка.) Число операций не более $n \log n$.

[Указание. Упорядочим точки — годится любой из порядков, использованных в двух предыдущих задачах. Затем, рассматривая точки по очереди, будем строить выпуклую оболочку уже рассмотренных точек. (Для хранения выпуклой оболочки полезно использовать дек, см. главу 6. Впрочем, при упорядочении точек по углам это излишне.)] □

4.4. Нижние оценки для числа сравнений при сортировке

Пусть имеется n различных по весу камней и весы, которые позволяют за одно взвешивание определить, какой из двух выбранных нами камней тяжелее. (В программистских терминах: мы имеем доступ к функции `тяжелее(i,j:1..n):boolean`.) Надо упорядочить камни по весу, сделав как можно меньше взвешиваний (вызовов функции `тяжелее`).

Разумеется, число взвешиваний зависит не только от выбранного нами алгоритма, но и от того, как оказались расположены камни. Сложностью алгоритма назовём число взвешиваний при *наихудшем* расположении камней.

4.4.1. Доказать, что сложность произвольного алгоритма сортировки n камней не меньше $\log_2 n!$ (где $n! = 1 \cdot 2 \cdot \ldots \cdot n$).

Решение. Пусть имеется алгоритм сложности не более d. Для каждого из $n!$ возможных расположений камней запротоколируем результаты взвешиваний (обращений к функции `тяжелее`); их можно записать в виде последовательности из не более чем d нулей и единиц. Для единообразия дополним последовательность нулями, чтобы её длина стала равной d. Тем самым у нас имеется $n!$ последовательностей из d нулей и единиц. Все эти последовательности разные — иначе наш алгоритм дал бы одинаковые ответы для разных порядков (и один из ответов был бы неправильным). Получаем, что $2^d \geqslant n!$ — что и требовалось доказать. □

Другой способ объяснить то же самое — рассмотреть дерево вариантов, возникающее в ходе выполнения алгоритма, и сослаться на то, что дерево высоты d не может иметь более 2^d листьев.

Несложно заметить, что $\log_2 n! \geqslant cn \log n$ при подходящем $c > 0$, поскольку в сумме

$$\log n! = \log 1 + \log 2 + \log 3 + \ldots + \log n$$

вторая половина слагаемых не меньше $\log_2(n/2) = \log_2 n - 1$ каждое.

Тем самым любой алгоритм сортировки, использующий только сравнения элементов массива и их перестановки, требует не менее $cn \log n$ действий, так что наши алгоритмы близки к оптимальным. Однако алгоритм сортировки, использующий другие операции, может действовать и быстрее. Вот один из примеров.

4.4.2. Имеется массив целых чисел `a[1]` ... `a[n]`, причём все числа неотрицательны и не превосходят `m`. Отсортировать этот массив; число действий порядка `m + n`.

Решение. Для каждого числа от `0` до `m` подсчитываем, сколько раз оно встречается в массиве. После этого исходный массив можно стереть и заполнить заново в порядке возрастания, используя сведения о кратности каждого числа. □

Отметим, что этот алгоритм не переставляет числа в массиве, как большинство других, а «записывает их туда заново».

Есть также метод сортировки, в котором последовательно проводится ряд «частичных сортировок» по отдельным битам. Начнём с такой задачи.

4.4.3. В массиве `a[1]`...`a[n]` целых чисел переставить элементы так, чтобы чётные числа шли перед нечётными (не меняя взаимный порядок в каждой из групп).

Решение. Сначала спишем (во вспомогательный массив) все чётные, а потом — все нечётные. □

4.4.4. Имеется массив из n чисел от 0 до $2^k - 1$, каждое из которых мы будем рассматривать как k-битовое слово из нулей и единиц. Используя проверки «i-ый бит равен 0» и «i-ый бит равен 1» вместо сравнений, отсортировать все числа за время порядка nk.

Решение. Отсортируем числа по последнему биту (см. предыдущую задачу), затем по предпоследнему и так далее. В результате они будут отсортированы. В самом деле, индукцией по i легко доказать, что после i шагов любые два числа, отличающиеся только в i последних битах, идут в правильном порядке. (Вариант: после i шагов i-битовые концы чисел идут в правильном порядке.) □

Аналогичный алгоритм может быть применён для m-ичной системы счисления вместо двоичной. При этом полезна такая вспомогательная задача:

4.4.5. Даны n чисел и функция f, принимающая (на них) значения $1 \dots m$. Требуется переставить числа в таком порядке, чтобы значения функции f не убывали (сохраняя порядок для чисел с равными значениями f). Число действий порядка $m + n$.

[Указание. Завести m списков суммарной длины n (как это сделать, смотри в главе 6 о типах данных) и помещать в i-ый список числа, для которых значение функции f равно i. Вариант: посчитать для всех i, сколько имеется чисел x с $f(x) = i$, после чего легко определить, с какого места нужно начинать размещать числа x с $f(x) = i$.] □

4.4.6. Даны n целых чисел в диапазоне от 1 до n^2. Как отсортировать их, сделав порядка n действий? □

4.5. Родственные сортировке задачи

4.5.1. Какова минимально возможная сложность (число сравнений в наихудшем случае) алгоритма отыскания самого тяжёлого из n камней?

Решение. Очевидный алгоритм с инвариантом «найден самый тяжёлый камень среди первых i» требует $n-1$ сравнений. Алгоритма меньшей сложности нет. Это вытекает из следующего более сильного утверждения. □

4.5.2. Эксперт хочет убедить суд, что данный камень — самый тяжёлый среди n камней, сделав менее $n-1$ взвешиваний. Доказать, что это невозможно. (Веса камней неизвестны суду, но известны эксперту.)

Решение. Изобразим камни точками, а взвешивания — линиями между ними. Получим граф с n вершинами и менее чем $n-1$ рёбрами. Такой граф несвязен (добавление каждого следующего ребра уменьшает число связных компонент не более чем на 1). Поэтому суд ничего не знает относительно соотношения весов камней в различных связных компонентах и может допустить, что самый тяжёлый камень — в любой из них.

Более простое объяснение: будем следить за тем, сколько камней к данному моменту не «проиграли» (то есть не оказались легче других). Вначале их n; при каждом взвешивании проигрывает только один камень, а если есть двое не проигравших никому, любой из них может (с точки зрения суда) оказаться самым тяжёлым. □

Разница между этой задачей и предыдущей: в этой задаче мы доказываем, что $n-2$ взвешиваний не достаточно не только для нахождения самого тяжёлого, но даже для того, чтобы убедиться, что данный камень является таковым — если предположительный ответ известен. (В случае сортировки, зная предположительный ответ, мы можем убедиться в его правильности, сделав всего $n-1$ сравнений — каждый сравниваем со следующим по весу. Напомним, что сортировка требует в худшем случае значительно больше сравнений.)

4.5.3. Доказать, что можно найти самый лёгкий и самый тяжёлый из $2n$ камней (одновременно), сделав $3n-2$ взвешиваний.

Решение. Разобьём камни произвольным образом на n пар и сравним камни в каждой паре (n взвешиваний). Отложим отдельно «победителей» (более тяжёлых в своей паре) и «проигравших» (более лёгких). Ясно, что самый лёгкий камень надо искать среди проигравших ($n-1$ сравнений), а самый тяжёлый — среди победителей (ещё $n-1$ сравнений). □

4.5.4. Доказать, что не существует алгоритма, позволяющего гарантированно найти самый лёгкий и самый тяжёлый среди $2n$ камней (одновременно), сделав менее $3n-2$ взвешиваний.

Решение. Пусть такой алгоритм существует. Наблюдая за его применением к какой-то группе из $2n$ камней, мы будем следить за четырьмя параметрами. А именно, мы будем смотреть, сколько камней

(a) кому-то уже проиграли, а у кого-то уже выиграли;

(b) кому-то уже проиграли, но ещё ни у кого не выиграли;

(c) у кого-то уже выиграли, но ещё никому не проиграли;

(d) ни у кого не выиграли и никому не проиграли (то есть ни с кем не сравнивались).

(Напомним, что выигравшим в сравнении мы считаем более тяжёлый камень.) Камни типа (a), очевидно, не могут уже оказаться ни самыми лёгкими, ни самыми тяжёлыми, каковы бы ни были результаты дальнейших сравнений. Любой камень типа (b) имеет шанс оказаться самым лёгким (в самом деле, его можно произвольно облегчить, не меняя результатов уже выполненных сравнений), но уже не может быть самым тяжёлым; для камней типа (c) наоборот. Наконец, любой камень типа (d) может быть и самым лёгким, и самым тяжёлым.

Обозначим через a, b, c, d количества камней в соответствующих категориях и проследим, как меняются эти параметры при очередном сравнении (в зависимости от того, камни какого типа сравниваются и с каким результатом).

сравнение	a	b	c	d	$b+c+(3/2)d$
$a-a$	0	0	0	0	0
$a>b$	0	0	0	0	0
$a<b$	+1	−1	0	0	−1
$a<c$	0	0	0	0	0
$a>c$	+1	0	−1	0	−1
$a>d$	0	+1	0	−1	−1/2
$a<d$	0	0	+1	−1	−1/2
$b-b$	+1	−1	0	0	−1
$b<c$	0	0	0	0	0
$b>c$	+2	−1	−1	0	−2
$b<d$	0	0	+1	−1	−1/2
$b>d$	+1	0	0	−1	−3/2
$c-c$	+1	0	−1	0	−1
$c<d$	+1	0	0	−1	−3/2
$c>d$	0	+1	0	−1	−1/2
$d-d$	0	+1	+1	−2	−1

Последний столбец таблицы показывает, как меняется величина $s = b + c + (3/2)d$ (которую можно рассматривать в качестве меры «остав-

шейся работы»: камень, про который не известно ничего, с точки зрения этой меры в полтора раза сложнее камня, для которого есть односторонняя оценка). Изначально $s = 3n$, а в конце $s = 2$ (про все камни, кроме двух, известно, что они относятся к категории (а)). Из таблицы видно, что при любом взвешивании есть «неудачный исход», при котором s уменьшается не более чем на единицу. Такие исходы действительно возможны (не противоречат результатам предыдущих взвешиваний): при сравнении b-камня и c-камня может оказаться, что c-камень тяжелее (его вес не ограничен сверху предыдущими взвешиваниями), а при сравнении с участием d-камня результат может быть любым, поскольку про d-камень ничего не известно. (Кроме того, можно заметить, что если один из исходов взвешивания невозможен, то это взвешивание вообще излишне и его можно не делать.) А если исходы всех взвешиваний неудачны, то уменьшение s с $3n$ до 2 потребует как минимум $3n - 2$ взвешиваний, что и требовалось доказать. □

4.5.5. Дано n различных по весу камней. Найти самый тяжёлый и второй по весу камни, сделав не более $n + \lceil \log_2 n \rceil - 2$ взвешиваний ($\lceil \log_2 n \rceil$ — наименьшее целое k, при котором $2^k \geqslant n$).

Решение. Сначала найдём победителя (самый тяжёлый камень), а потом будем искать второй по весу. Ясно, что второго можно искать лишь среди тех, кто проиграл лично победителю (проигравшие кому-то ещё легче сразу двух камней). Если определять победителя в турнире по олимпийской системе (все делятся на пары, проигравшие выбывают, потом снова делятся на пары и так далее), то для 2^k участников понадобится k раундов, а для n участников — $\lceil \log_2 n \rceil$ раундов. В каждой игре турнира выбывает один участник, поэтому всего будет $n - 1$ игр для определения победителя и ещё $\lceil \log_2 n \rceil - 1$ в турнире за второе место среди проигравших победителю. □

4.5.6. Доказать, что никакой алгоритм нахождения самого тяжёлого и второго по весу среди n камней не может гарантированно сделать это менее чем за $n + \lceil \log_2 n \rceil - 2$ взвешиваний.

Решение. Пусть дан такой алгоритм. В каждый момент его исполнения рассмотрим число k_i камней-участников, проигравших не менее i игр-сравнений. (Косвенные проигрыши — если a проиграл b, а b проиграл c, — не учитываются.) Легко понять, что сумма k_i по всем i равна числу игр, так как после каждой игры одно из k_i увеличивается на единицу.

Поэтому достаточно показать, что каков бы ни был алгоритм, при неудачных для него результатах игр будет выполнено неравенство

$k_1 + k_2 \geqslant n + \lceil \log_2 n \rceil - 2$. Будем называть «лидерами» тех участников, которые ещё никому не проиграли. В начале их n, а в конце остаётся только один лидер (поскольку любой из лидеров может быть победителем). Поэтому $k_1 \geqslant n - 1$ (все игроки, кроме одного, кому-то проиграли). Объясним, как надо выбирать результаты матчей, чтобы добиться неравенства $k_2 \geqslant \lceil \log_2 n \rceil - 1$. Результат встречи двух не-лидеров может быть выбран любым. Если лидер встречается с не-лидером, то выигрывает лидер. При встрече двух лидеров выигрывает более опытный, то есть тот, кто выиграл к этому моменту больше игр (при равенстве — любой).

Чтобы доказать, что в этом случае выполнено искомое неравенство на k_2, введём отношения подчинения, считая при этом, что каждый игрок в любой момент игры подчинён ровно одному лидеру. В начале каждый сам себе лидер и подчинён только себе. При встрече лидера с не-лидером (или двух не-лидеров) подчинение не меняется; при встрече двух лидеров проигравший и все его подчинённые переподчиняются выигравшему.

Легко доказать по индукции, что если лидер выиграл k игр, то группа его подчинённых (включая его самого) содержит не более 2^k человек. Вначале $k = 0$ и в его группе только он сам. Если лидер выиграл k игр и побеждает лидера, выигравшего не более k игр, то в каждой из групп не более 2^k игроков, а в объединении не более 2^{k+1} игроков.

Следовательно, по окончании турнира лидер выиграл не менее $\lceil \log_2 n \rceil$ игр, поскольку в его группе все n игроков. Все побеждённые им, кроме второго по силе игрока, проиграли ещё кому-то (иначе почему мы уверены, что они не вторые по силе?). Отсюда и получается требуемая оценка на k_2. □

4.5.7. Доказать, что оценка предыдущей задачи остаётся в силе, если требуется найти лишь второй по весу камень, а самый тяжёлый искать не обязательно.

[Указание. Если по окончанию турнира определился второй по силе игрок, то он кому-то проиграл (откуда мы знаем иначе, что он не первый?), и тем самым известен и победитель.] □

4.5.8. Дано n различных по весу камней и число k (от 1 до n). Требуется найти k-ый по весу камень, сделав не более Cn взвешиваний, где C — некоторая константа, не зависящая от k и n.

Замечание. Сортировка позволяет сделать это за $Cn \log n$ взвешиваний. Указание к этой (трудной) задаче приведено в главе про рекурсию. □

Следующая задача имеет неожиданно простое решение.

4.5.9. Имеется n одинаковых на вид камней, некоторые из которых на самом деле различны по весу. Имеется прибор, позволяющий по двум камням определить, одинаковы они или различны (но не говорящий, какой тяжелее). Известно, что среди этих камней большинство (более $n/2$) одинаковых. Сделав не более n взвешиваний, найти хотя бы один камень из этого большинства. (Предостережение. Если два камня одинаковые, это не гарантирует их принадлежности к большинству.)

[Указание. Если найдены два различных камня, то их оба можно выбросить — хотя бы один из них плохой и большинство останется большинством.]

Решение. Программа просматривает камни по очереди, храня в переменной `i` число просмотренных камней. (Считаем камни пронумерованными от 1 до `n`.) Помимо этого программа хранит номер «текущего кандидата» `c` и его «кратность» `k`. Смысл этих названий объясняется инвариантом `(И)`:

> если к непросмотренным камням (с номерами `i+1` . . . `n`) добавили бы `k` копий `c`-го камня, то наиболее частым среди них был бы такой же камень, что и для исходного массива.

Получаем такую программу:

```
k:=0; i:=0;
{(И)}
while i<>n do begin
| if k=0 then begin
| | k:=1; c:=i+1; i:=i+1;
| end else if (i+1-ый камень одинаков с c-ым) then begin
| | i:=i+1; k:=k+1;
| |  {заменяем материальный камень идеальным}
| end else begin
| | i:=i+1; k:=k-1;
| | {выкидываем один материальный и один идеальный камень}
| end;
end;
искомым является c-ый камень
```

Замечание. Поскольку во всех трёх вариантах выбора стоит команда `i:=i+1`, её можно вынести наружу. □

Заметим также, что эта программа гарантирует отыскание наиболее частого камня, лишь если он составляет большинство.

Следующая задача не имеет на первый взгляд никакого отношения к сортировке.

4.5.10. Имеется квадратная таблица `a[1..n,1..n]`. Известно, что для некоторого `i` строка с номером `i` заполнена одними нулями, а столбец с номером `i` — одними единицами (за исключением их пересечения на диагонали, где стоит неизвестно что). Найти такое `i` (оно, очевидно, единственно). Число действий порядка `n`. (Заметим, что это существенно меньше числа элементов в таблице.)

[Указание. Рассмотрите `a[i][j]` как результат «сравнения» `i` с `j` и вспомните, что самый тяжёлый из `n` камней может быть найден за `n` сравнений. (Заметим, что таблица может не быть «транзитивной», но всё равно при «сравнении» двух элементов один из них отпадает.)] □

5. КОНЕЧНЫЕ АВТОМАТЫ И ОБРАБОТКА ТЕКСТОВ

5.1. Составные символы, комментарии и т.п.

5.1.1. В тексте возведение в степень обозначалось двумя идущими подряд звёздочками. Решено заменить это обозначение на `^` (так что, к примеру, `x**y` заменится на `x^y`). Как это проще всего сделать? Исходный текст читается символ за символом, получающийся текст требуется печатать символ за символом.

Решение. В каждый момент программа находится в одном из двух состояний: «основное» и «после» (звёздочки):

Состояние	Очередной входной символ	Новое состояние	Действие
основное	*	после	нет
основное	$x \neq$ *	основное	печатать x
после	*	основное	печатать ^
после	$x \neq$ *	основное	печатать *, x

Если в конце текста программа оказывается в состоянии «после», то следует напечатать звёздочку (и кончить работу). □

Замечание. Наша программа заменяет `***` на `^*` (но не на `*^`). В условии задачи мы не оговаривали деталей, как это часто делается — предполагается, что программа «должна действовать разумно». В данном случае, пожалуй, самый простой способ объяснить, как программа действует — это описать её состояния и действия в них.

5.1.2. Написать программу, удаляющую из текста все подслова вида `abc`. □

5.1.3. В паскале комментарии заключаются в фигурные скобки:

```
begin {начало цикла}
i:=i+1; {увеличиваем i на 1}
```

Написать программу, которая удаляла бы комментарии и вставляла бы вместо исключённого комментария пробел (чтобы `1{один}2` превратилось не в `12`, а в `1 2`).

Решение. Программа имеет два состояния: «основное» и «внутри» (комментария).

Состояние	Очередной входной символ	Новое состояние	Действие
основное	{	внутри	нет
основное	$x \neq$ {	основное	печатать x
внутри	}	основное	печатать пробел
внутри	$x \neq$ }	внутри	нет

□

Замечание. Эта программа не воспринимает вложенные комментарии: строка вроде

```
{{комментарий внутри} комментария}
```

превратится в

```
 комментария}
```

(в начале стоят два пробела). Обработка вложенных комментариев конечным автоматом невозможна (нужно «помнить число скобок» — а произвольное натуральное число не помещается в конечную память).

5.1.4. В паскалевских программах бывают также строки, заключённые в кавычки. Если фигурная скобка встречается внутри строки, то она не означает начала или конца комментария. В свою очередь, кавычка в комментарии не означает начала или конца строки. Как изменить программу, чтобы это учесть?

[Указание. Состояний будет три: основное, внутри комментария, внутри строки.] □

5.1.5. Ещё одна возможность многих реализаций паскаля — это комментарии вида

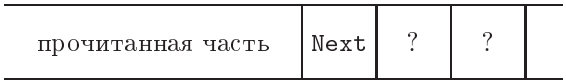
```
i:=i+1;     (*   here i is increased by 1  *)
```

при этом закрывающая скобка должна соответствовать открывающей (то есть {. . . `*)` не разрешается). Как удалять такие комментарии? □

5.2. Ввод чисел

Пусть десятичная запись числа подаётся на вход программы символ за символом. Мы хотим «прочесть» это число (поместить в переменную типа `real` его значение). Кроме того, надо сообщить об ошибке, если число записано неверно.

Более конкретно, представим себе такую ситуацию. Последовательность символов на входе делится на прочитанную и оставшуюся части. Мы можем пользоваться функцией `Next:char`, которая даёт первый символ оставшейся части, а также процедурой `Move`, которая забирает первый символ из оставшейся части, переводя его в категорию прочитанных.



Будем называть десятичной записью такую последовательность символов:

⟨0 или более пробелов⟩ ⟨1 или более цифр⟩,

а также такую:

⟨0 или более пробелов⟩ ⟨1 или более цифр⟩.⟨1 или более цифр⟩.

Заметим, что согласно этому определению

`1.` `.1` `1.␣1` `-1.1`

не являются десятичными записями. Сформулируем теперь задачу точно:

5.2.1. Прочесть из входной строки максимальную часть, которая может быть началом десятичной записи. Определить, является ли эта часть десятичной записью или нет.

Решение. Запишем программу на паскале (используя «перечислимый тип» для наглядности записи: переменная `state` может принимать одно из значений, указанных в скобках).

```
var state:
 (Accept, Error, Initial, IntPart, DecPoint, FracPart);

state := Initial;
```

```
while (state <> Accept) or (state <> Error) do begin
| if state = Initial then begin
| | if Next = ' ' then begin
| | | state := Initial; Move;
| | end else if Digit(Next) then begin
| | | state := IntPart; {после начала целой части}
| | | Move;
| | end else begin
| | | state := Error;
| | end;
| end else if state = IntPart then begin
| | if Digit (Next) then begin
| | | state := IntPart; Move;
| | end else if Next = '.' then begin
| | | state := DecPoint; {после десятичной точки}
| | | Move;
| | end else begin
| | | state := Accept;
| | end;
| end else if state = DecPoint then begin
| | if Digit (Next) then begin
| | | state := FracPart; Move;
| | end else begin
| | | state := Error; {должна быть хоть одна цифра}
| | end;
| end else if state = FracPart then begin
| | if Digit (Next) then begin
| | | state := FracPart; Move;
| | end else begin
| | | state := Accept;
| | end;
| end else if
| | {такого  быть не может}
| end;
end;
```

Заметьте, что присваивания `state:=Accept` и `state:=Error` не сопровождаются сдвигом (символ, который не может быть частью числа, не забирается). □

Приведённая программа не запоминает значение прочитанного числа.

5.2.2. Решить предыдущую задачу с дополнительным требованием: если прочитанный кусок является десятичной записью, то в переменную `val:real` следует поместить её значение.

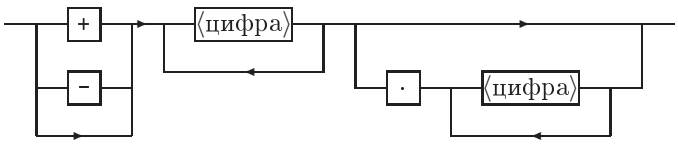
Решение. При чтении дробной части переменная `step` хранит множитель при следующей десятичной цифре.

```
state := Initial; val:= 0;
while (state <> Accept) or (state <> Error) do begin
| if state = Initial then begin
| | if Next = ' ' then begin
| | | state := Initial; Move;
| | end else if Digit(Next) then begin
| | | state := IntPart; {после начала целой части}
| | | val := DigitValue (Next); Move;
| | end else begin
| | | state := Error;
| | end;
| end else if state = IntPart then begin
| | if Digit (Next) then begin
| | | state := IntPart; val := 10*val + DigitVal(Next);
| | | Move;
| | end else if Next = '.' then begin
| | | state := DecPoint; {после десятичной точки}
| | | step := 0.1;
| | | Move;
| | end else begin
| | | state := Accept;
| | end;
| end else if state = DecPoint then begin
| | if Digit (Next) then begin
| | | state := FracPart;
| | | val := val + DigitVal(Next)*step; step := step/10;
| | | Move;
| | end else begin
| | | state := Error; {должна быть хоть одна цифра}
| | end;
| end else if state = FracPart then begin
| | if Digit (Next) then begin
| | | state := FracPart;
| | | val := val + DigitVal(Next)*step; step := step/10;
| | | Move;
| | end else begin
| | | state := Accept;
| | end;
| end else if
| | {такого  быть не может}
| end;
end;
```

□

5.2.3. Та же задача, если перед числом может стоять знак `-` или знак `+` (а может ничего не стоять).

Формат чисел в этой задаче обычно иллюстрируют такой картинкой:



□

5.2.4. Та же задача, если к тому же после числа может стоять показатель степени десяти, как в `254E-4` (= 0.0254) или в `0.123E+9` (= 123 000 000). Нарисовать соответствующую картинку. □

5.2.5. Что надо изменить в приведённой выше программе, чтобы разрешить пустые целую и дробную части (как в «`1.`», «`.1`» или даже «`.`» — последнее число считаем равным нулю)? □

Мы вернёмся к конечным автоматам в главе 10 (Сравнение с образцом).

6. ТИПЫ ДАННЫХ

6.1. Стеки

Пусть `T` — некоторый тип. Рассмотрим (отсутствующий в паскале) тип «стек элементов типа `T`». Его значениями являются последовательности значений типа `T`.

Операции:

- Сделать_пустым (`var s`: стек элементов типа `T`)
- Добавить (`t:T`; `var s`: стек элементов типа `T`)
- Взять (`var t:T`; `var s`: стек элементов типа `T`)
- Пуст (`s`: стек элементов типа `T`): `boolean`
- Вершина (`s`: стек элементов типа `T`): `T`

(Мы пользуемся обозначениями, напоминающими паскаль, хотя в паскале типа «стек» нет.) Процедура «Сделать_пустым» делает стек `s` пустым. Процедура «Добавить» добавляет `t` в конец последовательности `s`. Процедура «Взять» применима, если последовательность `s` непуста; она забирает из неё последний элемент, который становится значением переменной `t`. Выражение «Пуст(`s`)» истинно, если последовательность `s` пуста. Выражение «Вершина(`s`)» определено, если последовательность `s` непуста, и равно последнему элементу последовательности `s`.

Мы покажем, как моделировать стек в паскале и для чего он может быть нужен.

Моделирование ограниченного стека в массиве

Будем считать, что количество элементов в стеке не превосходит некоторого числа `n`. Тогда стек можно моделировать с помощью двух

переменных:

```
Содержание: array [1..n] of T;
Длина: integer;
```

считая, что в стеке находятся элементы

```
Содержание [1],...,Содержание [Длина].
```

- Чтобы сделать стек пустым, достаточно положить

```
Длина := 0
```

- Добавить элемент t:

```
{Длина < n}
Длина := Длина+1;
Содержание [Длина] :=t;
```

- Взять элемент в переменную t:

```
{Длина > 0}
t := Содержание [Длина];
Длина := Длина - 1;
```

- Стек пуст, если Длина = 0.
- Вершина стека равна Содержание [Длина].

Таким образом, вместо переменной типа стек в программе на паскале можно использовать две переменные Содержание и Длина. Можно также определить тип stack, записав

```
const N = ...
type
  stack = record
  | Содержание: array [1..N] of T;
  | Длина: integer;
  end;
```

(Мы позволяем себе использовать имена переменных из русских букв, хотя обычно паскаль этого не любит.) После этого могут быть — в соответствии с правилами паскаля — описаны процедуры работы со стеком. Например, можно написать

```
procedure Добавить (t: T; var s: stack);
begin
| {s.Длина < N}
| s.Длина := s.Длина + 1;
| s.Содержание [s.Длина] := t;
end;
```

Использование стека

Будем рассматривать последовательности открывающихся и закрывающихся круглых и квадратных скобок ( ) [ ]. Среди всех таких последовательностей выделим правильные — те, которые могут быть получены по таким правилам:

- пустая последовательность правильна.
- если A и B правильны, то и AB правильна.
- если A правильна, то [A] и (A) правильны.

Пример. Последовательности (), [[]], [()[]()][] правильны, а последовательности],)(, (], ([)] — нет.

6.1.1. Проверить правильность последовательности за время, не превосходящее константы, умноженной на её длину. Предполагается, что члены последовательности закодированы числами:

(	1
[	2
)	−1
]	−2

Решение. Пусть a[1]...a[n] — проверяемая последовательность. Разрешим хранить в стеке открывающиеся круглые и квадратные скобки (т. е. 1 и 2).

Вначале стек делаем пустым. Далее просматриваем члены последовательности слева направо. Встретив открывающуюся скобку (круглую или квадратную), помещаем её в стек. Встретив закрывающуюся, проверяем, что вершина стека — парная ей скобка; если это не так, то можно утверждать, что последовательность неправильна, если скобка парная, то заберём её (вершину) из стека. Последовательность правильна, если в конце стек оказывается пуст.

```
Сделать_пустым (s);
i := 0; Обнаружена_ошибка := false;
{прочитано i символов последовательности}
while (i < n) and not Обнаружена_ошибка do begin
| i := i + 1;
| if (a[i] = 1) or (a[i] = 2) then begin
| | Добавить (a[i], s);
| end else begin  {a[i] равно -1 или -2}
```

```
    if Пуст (s) then begin
      Обнаружена_ошибка := true;
    end else begin
      Взять (t, s);
      Обнаружена_ошибка := (t <> - a[i]);
    end;
  end;
end;
Правильно := (not Обнаружена_ошибка) and Пуст (s);
```

Убедимся в правильности программы.

(1) Если последовательность построена по правилам, то программа даст ответ «да». Это легко доказать индукцией по построению правильной последовательности. Надо проверить для пустой, для последовательности AB в предположении, что для A и B уже проверено, и, наконец, для последовательностей $[A]$ и (A) — в предположении, что для A уже проверено. Для пустой очевидно. Для AB действия программы происходят как для A и кончаются с пустым стеком; затем всё происходит как для B. Для $[A]$ сначала помещается в стек открывающая квадратная скобка и затем всё идёт как для A — с той разницей, что в глубине стека лежит лишняя скобка. По окончании A стек становится пустым — если не считать этой скобки — а затем и совсем пустым. Аналогично для (A).

(2) Покажем, что если программа завершает работу с ответом «да», то последовательность правильна. Рассуждаем индукцией по длине последовательности. Проследим за состоянием стека в процессе работы программы. Если он в некоторый промежуточный момент пуст, то последовательность разбивается на две части, для каждой из которых программа даёт ответ «да»; остаётся воспользоваться предположением индукции и определением правильности. Пусть стек всё время непуст. Это значит, что положенная в него на первом шаге скобка будет вынута лишь на последнем шаге. Тем самым, первый и последний символы последовательности — это парные скобки, и последовательность имеет вид (A) или $[A]$, а работа программы (кроме первого и последнего шагов) отличается от её работы на A лишь наличием лишней скобки на дне стека (раз её не вынимают, она никак не влияет на работу программы). Снова ссылаемся на предположение индукции и определение правильности. □

6.1.2. Как упростится программа, если известно, что в последовательности могут быть только круглые скобки?

Решение. В этом случае от стека остаётся лишь его длина, и мы фактически приходим к такому утверждению: последовательность круглых скобок правильна тогда и только тогда, когда в любом её начальном отрезке число закрывающихся скобок не превосходит числа открывающихся, а для всей последовательности эти числа равны. □

6.1.3. Реализовать с помощью одного массива два стека, суммарное количество элементов в которых ограничено длиной массива; все действия со стеками должны выполняться за время, ограниченное константой, не зависящей от длины стеков.

Решение. Стеки должны расти с концов массива навстречу друг другу: первый должен занимать места

$$\texttt{Содержание[1]} \ldots \texttt{Содержание[Длина1]},$$

а второй —

$$\texttt{Содержание[n]} \ldots \texttt{Содержание[n-Длина2+1]}$$

(вершины обоих стеков записаны последними). □

6.1.4. Реализовать k стеков с элементами типа T, общее количество элементов в которых не превосходит n, с использованием массивов суммарной длины $C(\texttt{n} + \texttt{k})$, затрачивая на каждое действие со стеками (кроме начальных действий, делающих все стеки пустыми) время не более некоторой константы C. (Как говорят, общая длина массивов должна быть $O(\texttt{m} + \texttt{n})$, а время на каждую операцию — $O(1)$.)

Решение. Применяемый метод называется «ссылочной реализацией». Он использует три массива:

```
Содержание: array [1..n] of T;
Следующий: array [1..n] of 0..n;
Вершина: array [1..k] of 0..n.
```

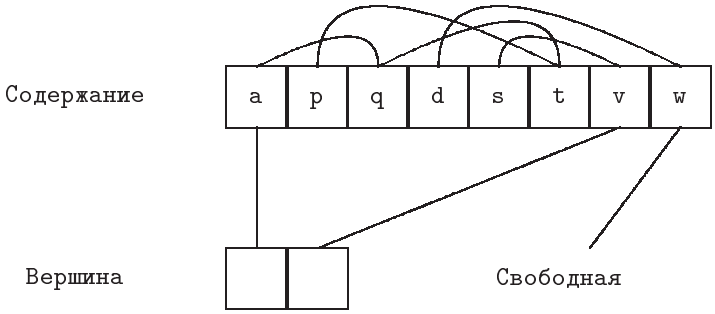
Удобно изображать массив Содержание как n ячеек с номерами 1...n, каждая из которых содержит элемент типа T. Массив Следующий изобразим в виде стрелок, проведя стрелку из i в j, если Следующий[i]=j. (Если Следующий[i]=0, стрелок из i не проводим.) Содержимое s-го стека ($\texttt{s} \in 1 \ldots \texttt{k}$) хранится так: вершина равна Содержание[Вершина[s]], остальные элементы s-го стека можно найти, идя по стрелкам — до тех пор, пока они не кончатся. При этом

$$(\texttt{s-ый стек пуст}) \Leftrightarrow \texttt{Вершина[s]=0}.$$

Стрелочные траектории, выходящие из

Вершина[1], . . . , Вершина[k]

(из тех, которые не равны 0) не должны пересекаться. Помимо них, нам понадобится ещё одна стрелочная траектория, содержащая все неиспользуемые в данный момент ячейки. Её начало мы будем хранить в переменной Свободная (равенство Свободная = 0 означает, что пустого места не осталось). Вот пример:



Содержание	a	p	q	d	s	t	v	w
Следующий	3	0	6	0	0	2	5	4
Вершина	1	7						

Свободная = 8

Стеки: 1-ый содержит p, t, q, a (a — вершина); 2-ой содержит s, v (v — вершина).

```
procedure Начать_работу; {Делает все стеки пустыми}
  var i: integer;
begin
  for i := 1 to k do begin
    Вершина [i]:=0;
  end;
  for i := 1 to n-1 do begin
    Следующий [i] := i+1;
  end;
  Следующий [n] := 0;
```

```
| Свободная:=1;
end;

function  Есть_место: boolean;
begin
| Есть_место := (Свободная <> 0);
end;

procedure Добавить (t: T; s: integer);
| {Добавить t к s-му стеку}
| var i: 1..n;
begin
| {Есть_место}
| i := Свободная;
| Свободная := Следующий [i];
| Следующий [i] := Вершина [s];
| Вершина [s] :=i;
| Содержание [i] := t;
end;

function Пуст (s: integer): boolean;
| {s-ый стек пуст}
begin
| Пуст := (Вершина [s] = 0);
end;

procedure Взять (var t: T; s: integer);
| {взять из s-го стека в t}
| var i: 1..n;
begin
| {not Пуст (s)}
| i := Вершина [s];
| t := Содержание [i];
| Вершина [s] := Следующий [i];
| Следующий [i] := Свободная;
| Свободная := i;
end;


function  Вершина_стека (s: integer): T;
| {вершина s-го стека}
begin
| Вершина_стека := Содержание[Вершина[s]];
end;
```

□

6.2. Очереди

Значениями типа «очередь элементов типа T», как и для стеков, являются последовательности значений типа T. Разница состоит в том, что берутся элементы не с конца, а с начала (а добавляются по-прежнему в конец).

Операции с очередями:

- Сделать_пустой (`var x:` очередь элементов типа `T`);
- Добавить (`t:T`, `var x:` очередь элементов типа `T`);
- Взять (`var t:T`, `var x:` очередь элементов типа `T`);
- Пуста (`x:` очередь элементов типа `T`): `boolean`;
- Очередной (`x:` очередь элементов типа `T`): `T`.

При выполнении команды «Добавить» указанный элемент добавляется в конец очереди. Команда «Взять» выполнима, лишь если очередь непуста, и забирает из неё первый (положенный туда раньше всех) элемент, помещая его в `t`. Значением функции «Очередной» (определённой для непустой очереди) является первый элемент очереди.

Английские названия стеков — Last In First Out (последним вошёл — первым вышел), а очередей — First In First Out (первым вошёл — первым вышел). Сокращения: LIFO, FIFO.

Реализация очередей в массиве

6.2.1. Реализовать операции с очередью ограниченной длины так, чтобы количество действий для каждой операции было ограничено константой, не зависящей от длины очереди.

Решение. Будем хранить элементы очереди в соседних элементах массива. Тогда очередь будет прирастать справа и убывать слева. Поскольку при этом она может дойти до края, свернём массив в окружность.

Введём массив

```
Содержание: array [0..n-1] of T
```

и переменные

```
Первый: 0..n-1,
Длина : 0..n.
```

При этом элементами очереди будут

`Содержание [Первый]`, `Содержание [Первый+1]`, . . . ,
`Содержание [Первый+Длина-1]`,

где сложение выполняется по модулю `n`. (Предупреждение. Если вместо этого ввести переменные `Первый` и `Последний`, значения которых — вычеты по модулю `n`, то пустая очередь может быть спутана с очередью из `n` элементов.)

Операции выполняются так.

Сделать пустой:

```
Длина := 0;
Первый := 0;
```

Добавить элемент:

```
{Длина < n}
Содержание [(Первый + Длина) mod n] := элемент;
Длина := Длина + 1;
```

Взять элемент:

```
{Длина > 0}
элемент := Содержание [Первый];
Первый := (Первый + 1) mod n;
Длина := Длина - 1;
```

Пуста:

```
Длина = 0
```

Очередной:

```
Содержание [Первый]
```

□

6.2.2. (Сообщил А. Г. Кушниренко) Придумать способ моделирования очереди с помощью двух стеков (и фиксированного числа переменных типа `T`). При этом отработка `n` операций с очередью (начатых, когда очередь была пуста) должна требовать порядка `n` действий.

Решение. Инвариант: *стеки, составленные концами, образуют очередь*. (Перечисляя элементы одного стека вглубь и затем элементы второго наружу, мы перечисляем все элементы очереди от первого до последнего.) Ясно, что добавление сводится к добавлению к одному из

стеков, а проверка пустоты — к проверке пустоты обоих стеков. Если мы хотим взять элемент, есть два случая. Если стек, где находится начало очереди, не пуст, то берём из него элемент. Если он пуст, то предварительно переписываем в него все элементы второго стека, меняя порядок (это происходит само собой при перекладывании из стека в стек) и сводим дело к первому случаю. Хотя число действий на этом шаге и не ограничено константой, но требование задачи выполнено, так как каждый элемент очереди может участвовать в этом процессе не более одного раза. □

6.2.3. Деком называют структуру, сочетающую очередь и стек: класть и забирать элементы можно с обоих концов. Как реализовать дек ограниченного размера на базе массива так, чтобы каждая операция требовала ограниченного числа действий? □

6.2.4. (Сообщил А. Г. Кушниренко.) Имеется дек элементов типа T и конечное число переменных типа T и целого типа. В начальном состоянии в деке некоторое число элементов. Составить программу, после исполнения которой в деке остались бы те же самые элементы, а их число было бы в одной из целых переменных.

[Указание. (1) Элементы дека можно циклически переставлять, забирая с одного конца и помещая в другой. После этого, сделав столько же шагов в обратном направлении, можно вернуть всё на место. (2) Как понять, прошли мы полный круг или не прошли? Если бы какой-то элемент заведомо отсутствовал в деке, то можно было бы его подсунуть и ждать вторичного появления. Но таких элементов нет. Вместо этого можно для данного n выполнить циклический сдвиг на n дважды, подсунув разные элементы, и посмотреть, появятся ли разные элементы через n шагов.] □

Применение очередей

6.2.5. Напечатать в порядке возрастания первые n натуральных чисел, в разложение которых на простые множители входят только числа 2, 3, 5.

Решение. Введём три очереди x2, x3, x5, в которых будем хранить элементы, которые в 2 (3, 5) раз больше напечатанных, но ещё не напечатаны. Определим процедуру

```
procedure напечатать_и_добавить (t: integer);
begin
```

```
  writeln (t);
  Добавить (2*t, x2);
  Добавить (3*t, x3);
  Добавить (5*t, x5);
end;
```

Вот схема программы:

```
...сделать x2, x3, x5 пустыми
напечатать_и_добавить (1);
k := 1; { k - число напечатанных }
{инвариант: напечатано в порядке возрастания k минимальных
членов нужного множества; в очередях элементы, вдвое,
втрое и впятеро большие напечатанных, но не напечатанные,
расположенные в возрастающем порядке}
while k <> n do begin
  x := min (очередной(x2), очередной(x3), очередной(x5));
  напечатать_и_добавить (x);
  k := k+1;
  ...взять x из тех очередей, где он был очередным;
end;
```

Пусть инвариант выполняется. Рассмотрим наименьший из ненапечатанных элементов множества; пусть это x. Тогда он делится нацело на одно из чисел 2, 3, 5, и частное также принадлежит множеству. Значит, оно напечатано. Значит, x находится в одной из очередей и, следовательно, является в ней первым (меньшие напечатаны, а элементы очередей не напечатаны). Напечатав x, мы должны его изъять и добавить его кратные.

Длины очередей не превосходят числа напечатанных элементов. □

Следующая задача связана с графами (к которым мы вернёмся в главе 9).

Пусть задано конечное множество, элементы которого называют *вершинами*, а также некоторое множество упорядоченных пар вершин, называемых *рёбрами*. В этом случае говорят, что задан *ориентированный граф*. Пару $\langle p, q\rangle$ называют ребром с *началом* p и *концом* q; говорят также, что оно *выходит* из вершины p и *входит* в вершину q. Обычно вершины графа изображают точками, а рёбра — стрелками, ведущими из начала в конец. (В соответствии с определением из данной вершины в данную ведёт не более одного ребра; возможны рёбра, у которых начало совпадает с концом.)

6.2.6. Известно, что ориентированный граф связен, т. е. из любой вершины можно пройти в любую по рёбрам. Кроме того, из каждой

вершины выходит столько же рёбер, сколько входит. Доказать, что существует замкнутый цикл, проходящий по каждому ребру ровно один раз. Составить алгоритм отыскания такого цикла.

Решение. Змеёй будем называть непустую очередь из вершин, в которой любые две вершины соединены ребром графа (началом является та вершина, которая ближе к началу очереди). Стоящая в начале очереди вершина будет хвостом змеи, последняя — головой. На рисунке змея изобразится в виде цепи рёбер графа, стрелки ведут от хвоста к голове. Добавление вершины в очередь соответствует росту змеи с головы, взятие вершины — отрезанию кончика хвоста.

Вначале змея состоит из единственной вершины. Далее мы следуем такому правилу:

```
while змея включает не все рёбра do begin
  if из головы выходит не входящее в змею ребро then begin
    удлинить змею этим ребром
  end else begin
    {голова змеи в той же вершине, что и хвост}
    отрезать конец хвоста и добавить его к голове
    {"змея откусывает конец хвоста"}
  end;
end;
```

Докажем, что мы достигнем цели.

(1) Идя по змее от хвоста к голове, мы входим в каждую вершину столько же раз, сколько выходим. Так как в любую вершину входит столько же рёбер, сколько выходит, то невозможность выйти означает, что голова змеи в той же точке, что и хвост.

(2) Змея не укорачивается, поэтому либо она охватит все рёбра, либо, начиная с некоторого момента, будет иметь постоянную длину. Во втором случае змея будет бесконечно «скользить по себе». Это возможно, только если из всех вершин змеи не выходит неиспользованных рёбер. В этом случае из связности следует, что змея проходит по всем рёбрам.

Замечание по реализации на паскале. Вершинами графа будем считать числа `1`...`n`. Для каждой вершины `i` будем хранить число `Out[i]` выходящих из неё рёбер, а также номера `Num[i][1]`,...,`Num[i][Out[i]]` тех вершин, куда эти рёбра ведут. В процессе построения змеи будем выбирать первое свободное ребро. Тогда достаточно хранить для каждой вершины *число* выходящих из неё использованных рёбер — это будут рёбра, идущие в начале списка. □

6.2.7. Доказать, что для всякого n существует последовательность нулей и единиц длины 2^n со следующим свойством: если «свернуть её в кольцо» и рассмотреть все фрагменты длины n (их число равно 2^n), то мы получим все возможные последовательности нулей и единиц длины n. Построить алгоритм отыскания такой последовательности, требующий не более C^n действий для некоторой константы C.

[Указание. Рассмотрим граф, вершинами которого являются последовательности нулей и единиц длины $n-1$. Будем считать, что из вершины x ведёт ребро в вершину y, если x может быть началом, а y — концом некоторой последовательности длины n. Тогда из каждой вершины входит и выходит два ребра. Цикл, проходящий по всем рёбрам, и даст требуемую последовательность.] □

6.2.8. Реализовать k очередей с ограниченной суммарной длиной n, используя память $O(n+k)$ [= не более $C(n+k)$ для некоторой константы C], причём каждая операция (кроме начальной, делающей все очереди пустыми) должна требовать ограниченного константой числа действий.

Решение. Действуем аналогично ссылочной реализации стеков: мы помним (для каждой очереди) первого, каждый участник очереди помнит следующего за ним (для последнего считается, что за ним стоит фиктивный элемент с номером 0). Кроме того, мы должны для каждой очереди знать последнего (если он есть) — иначе не удастся добавлять. Как и для стеков, отдельно есть цепь свободных ячеек. Заметим, что для пустой очереди информация о последнем элементе теряет смысл — но она и не используется при добавлении.

```
Содержание: array [1..n] of T;
Следующий: array [1..n] of 0..n;
Первый: array [1..k] of 0..n;
Последний: array [1..k] of 0..n;
Свободная : 0..n;

procedure Сделать_пустым;
| var i: integer;
begin
| for i := 1 to n-1 do begin
| | Следующий [i] := i + 1;
| end;
| Следующий [n] := 0;
| Свободная := 1;
| for i := 1 to k do begin
```

```
  | Первый [i]:=0;
  end;
end;

function Есть_место : boolean;
begin
  Есть_место := Свободная <> 0;
end;

function Пуста (номер_очереди: integer): boolean;
begin
  Пуста := Первый [номер_очереди] = 0;
end;

procedure Взять (var t: T; номер_очереди: integer);
  var перв: integer;
begin
  {not Пуста (номер_очереди)}
  перв := Первый [номер_очереди];
  t := Содержание [перв]
  Первый [номер_очереди] := Следующий [перв];
  Следующий [перв] := Свободная;
  Свободная := перв;
end;

procedure Добавить (t: T; номер_очереди: integer);
  var нов, посл: 1..n;
begin
  {Есть_место }
  нов := Свободная; Свободная := Следующий [Свободная];
  {из списка свободного места изъят номер нов}
  if Пуста (номер_очереди) then begin
    Первый [номер_очереди] := нов;
    Последний [номер_очереди] := нов;
    Следующий [нов] := 0;
    Содержание [нов] := t;
  end else begin
    посл := Последний [номер_очереди];
    {Следующий [посл] = 0 }
    Следующий [посл] := нов;
    Следующий [нов] := 0;
    Содержание [нов] := t
    Последний [номер_очереди] := нов;
  end;
end;
```

```
function Очередной (номер_очереди: integer): T;
begin
| Очередной := Содержание [Первый [номер_очереди]];
end;
```
□

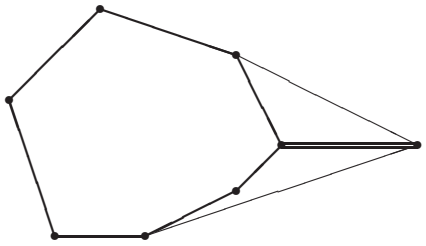
6.2.9. Та же задача для деков вместо очередей.

[Указание. Дек — структура симметричная, поэтому надо хранить ссылки в обе стороны (вперёд и назад). При этом удобно к каждому деку добавить фиктивный элемент, замкнув его в кольцо, и точно такое же кольцо образовать из свободных позиций.] □

В следующей задаче дек используется для хранения вершин выпуклого многоугольника.

6.2.10. На плоскости задано n точек, пронумерованных слева направо (а при равных абсциссах — снизу вверх). Составить программу, которая строит многоугольник, являющийся их выпуклой оболочкой, за $O(n)$ [= не более чем Cn] действий.

Решение. Будем присоединять точки к выпуклой оболочке одна за другой. Легко показать, что последняя присоединённая точка будет одной из вершин выпуклой оболочки. Эту вершину мы будем называть выделенной. Очередная присоединяемая точка видна из выделенной (почему?). Дополним наш многоугольник, выпустив из выделенной вершины «иглу», ведущую в присоединяемую точку. Получится вырожденный многоугольник, и остаётся ликвидировать в нём «впуклости».



Будем хранить вершины многоугольника в деке в порядке обхода его периметра по часовой стрелке. При этом выделенная вершина является началом и концом (головой и хвостом) дека. Присоединение «иглы» теперь состоит в добавлении присоединяемой вершины в голову и в

хвост дека. Устранение впуклостей несколько более сложно. Назовём *подхвостом* и *подподхвостом* элементы дека, стоящие за его хвостом. Устранение впуклости у хвоста делается так:

```
while по дороге из хвоста в подподхвост  мы поворачиваем
|          у подхвоста влево ("впуклость") do begin
| выкинуть подхвост из дека
end
```

Таким же способом устраняется впуклость у головы дека.

Замечание. Действия с подхвостом и подподхвостом не входят в определение дека, однако сводятся к небольшому числу манипуляций с деком (надо забрать три элемента с хвоста, сделать что надо и вернуть).

Ещё одно замечание. Есть два вырожденных случая: если мы вообще не поворачиваем у подхвоста (т. е. три соседние вершины лежат на одной прямой) и если мы поворачиваем на 180° (так бывает, если наш многоугольник есть двуугольник). В первом случае подхвост стоит удалить (чтобы в выпуклой оболочке не было лишних вершин), а во втором случае — обязательно оставить. □

6.3. Множества

Пусть `T` — некоторый тип. Существует много способов хранить (конечные) множества элементов типа `T`; выбор между ними определяется типом `T` и набором требуемых операций.

Подмножества множества $\{1 \dots n\}$

6.3.1. Используя память $O(n)$ [= пропорциональную n], хранить подмножества множества $\{1 \dots n\}$.

Операции	Число действий
Сделать пустым	Cn
Проверить принадлежность	C
Добавить	C
Удалить	C
Минимальный элемент	Cn
Проверка пустоты	Cn

Решение. Храним множество как `array [1..n] of Boolean`. □

6.3.2. То же, но проверка пустоты должна выполняться за время C.

Решение. Храним дополнительно количество элементов. □

6.3.3. То же при следующих ограничениях на число действий:

Операции	Число действий
Сделать пустым	Cn
Проверить принадлежность	C
Добавить	C
Удалить	Cn
Минимальный элемент	C
Проверка пустоты	C

Решение. Дополнительно храним минимальный элемент множества. □

6.3.4. То же при следующих ограничениях на число действий:

Операции	Число действий
Сделать пустым	Cn
Проверить принадлежность	C
Добавить	Cn
Удалить	C
Минимальный элемент	C
Проверка пустоты	C

Решение. Храним минимальный, а для каждого — следующий и предыдущий по величине. □

Множества целых чисел

В следующих задачах величина элементов множества не ограничена, но их количество не превосходит n.

6.3.5. Память Cn.

Операции	Число действий
Сделать пустым	C
Число элементов	C
Проверить принадлежность	Cn
Добавить новый (заведомо отсутствующий)	C
Удалить	Cn
Минимальный элемент	Cn
Взять какой-то элемент	C

Решение. Множество представляем с помощью переменных

```
a:array [1..n] of integer, k: 0..n;
```

множество содержит k элементов a[1],... a[k]; все они различны. По существу мы храним элементы множества в стеке (без повторений). С тем же успехом можно было бы воспользоваться очередью вместо стека. □

6.3.6. Память Cn.

Операции	Число действий
Сделать пустым	C
Проверить пустоту	C
Проверить принадлежность	$C \log n$
Добавить	Cn
Удалить	Cn
Минимальный элемент	C

Решение. См. решение предыдущей задачи с дополнительным условием a[1] < ... < a[k]. При проверке принадлежности используем двоичный поиск. □

В следующей задаче полезно комбинировать разные способы.

6.3.7. Используя описанные способы представления множеств, найти все вершины ориентированного графа, доступные из данной по рёбрам. (Вершины считаем числами 1...n.) Время не больше $C \cdot$ (общее число рёбер, выходящих из доступных вершин).

Решение. (Другое решение смотри в главе о рекурсии, задача 7.4.6) Пусть num[i] — число рёбер, выходящих из i, а out[i][1],... ..., out[i][num[i]] — вершины, куда ведут рёбра из вершины i.

```
procedure Доступные (i: integer);
|   {напечатать все вершины, доступные из i, включая i}
| var  X: подмножество 1..n;
|      P: подмножество 1..n;
|      q, v, w: 1..n;
|      k: integer;
begin
| ...сделать X, P пустыми;
| writeln (i);
| ...добавить i к X, P;
```

```
| {(1) P = множество напечатанных вершин; P содержит i;
|  (2) напечатаны только доступные из i вершины;
|  (3) X - подмножество P;
|  (4) все напечатанные вершины, из которых выходит
|      ребро в ненапечатанную вершину, принадлежат X}
|  while X непусто do begin
|  | ...взять какой-нибудь элемент X в v;
|  | for k := 1 to num [v] do begin
|  | | w := out [v][k];
|  | | if w не принадлежит P then begin
|  | | | writeln (w);
|  | | | добавить w в P;
|  | | | добавить w в X;
|  | | end;
|  | end;
|  end;
| end;
```

Свойство (1) не нарушается, так как печать происходит одновременно с добавлением в P. Свойство (2): раз v было в X, то v доступно, поэтому w доступно. Свойство (3) очевидно. Свойство (4): мы удалили из X элемент v, но все вершины, куда из v идут рёбра, перед этим напечатаны. □

6.3.8. Показать, что можно использовать и другой инвариант: P — напечатанные вершины; $X \subset P$; осталось напечатать вершины, доступные из X по ненапечатанным вершинам. □

Оценка времени работы. Заметим, что изъятые из X элементы больше туда не добавляются, так как они в момент изъятия (и, следовательно, всегда позже) принадлежат P, а добавляются только элементы не из P. Поэтому тело цикла `while` для каждой доступной вершины выполняется не более, чем по разу, при этом тело цикла `for` выполняется столько раз, сколько из вершины выходит рёбер.

Для X надо использовать представление со стеком или очередью (см. выше), для P — булевский массив. □

6.3.9. Решить предыдущую задачу, если требуется, чтобы доступные вершины печатались в таком порядке: сначала заданная вершина, потом её соседи, потом соседи соседей (ещё не напечатанные) и т. д.

[Указание. Так получится, если использовать очередь для хранения X в приведённом выше решении: докажите индукцией по k, что существует момент, в который напечатаны все вершины на расстоянии не больше k, а в очереди находятся все вершины, удалённые ровно на k.] □

Более сложные способы представления множеств будут разобраны в главах 13 (хеширование) и 14 (деревья).

6.4. Разные задачи

6.4.1. Реализовать структуру данных, которая имеет все те же операции, что массив длины `n`, а именно

- начать работу;
- положить в `i`-ю ячейку число `x`;
- узнать, что лежит в `i`-ой ячейке;

а также операцию

- указать номер минимального элемента

(точнее, одного из минимальных элементов). Количество действий для всех операций должно быть не более $C \log \mathtt{n}$, не считая операции «начать работу» (которая требует не более $C\mathtt{n}$ действий).

Решение. Используется приём, изложенный в разделе о сортировке деревом. Именно, надстроим над элементами массива как над листьями двоичное дерево, в каждой вершине которого храним минимум элементов соответствующего поддерева. Корректировка этой информации, а также прослеживание пути из корня к минимальному элементу требуют логарифмического числа действий. □

6.4.2. Приоритетная очередь — это очередь, в которой важно не то, кто встал последним (порядок помещения в неё не играет роли), а кто главнее. Более точно, при помещении в очередь указывается приоритет помещаемого объекта (будем считать приоритеты целыми числами), а при взятии из очереди выбирается элемент с наибольшим приоритетом (или один из таких элементов). Реализовать приоритетную очередь так, чтобы помещение и взятие элемента требовали логарифмического числа действий (от размера очереди).

Решение. Следуя алгоритму сортировки деревом (в его окончательном варианте), будем размещать элементы очереди в массиве `x[1..k]`, поддерживая такое свойство: `x[i]` старше (имеет больший приоритет) своих сыновей `x[2i]` и `x[2i+1]`, если таковые существуют — и, следовательно, всякий элемент старше своих потомков. (Сведения о приоритетах также хранятся в массиве, так что мы имеем дело с массивом пар ⟨элемент, приоритет⟩.) Удаление элемента с сохранением этого

свойства описано в алгоритме сортировки. Надо ещё уметь восстанавливать свойство после добавления элемента в конец. Это делается так:

```
t:= номер добавленного элемента
{инвариант: в дереве любой предок приоритетнее потомка,
    если этот потомок - не t}
while t - не корень и t старше своего отца do begin
| поменять t с его отцом
end;
```

Если очередь образуют граждане, стоящие в вершинах дерева, т. е. за каждым стоит двое, а перед каждым (кроме первого) — один, то смысл этого алгоритма ясен: встав в конец, приоритетный гражданин начинает пробираться к началу, вытесняя впереди стоящих — пока не встретит более приоритетного. □

Замечание. Приоритетную очередь естественно использовать при моделировании протекающих во времени процессов. При этом элементы очереди — это ожидаемые события, а их приоритет определяется временем, когда они произойдут.

7. РЕКУРСИЯ

7.1. Примеры рекурсивных программ

При анализе рекурсивной программы возникает, как обычно, два вопроса:

(а) почему программа заканчивает работу?

(б) почему она работает правильно, если заканчивает работу?

Для (б) достаточно проверить, что (содержащая рекурсивный вызов) программа работает правильно, предположив, что вызываемая ею одноимённая программа работает правильно. В самом деле, в этом случае в цепочке рекурсивно вызываемых программ все программы работают правильно (убеждаемся в этом, идя от конца цепочки к началу).

Чтобы доказать (а), обычно проверяют, что с каждым рекурсивным вызовом значение какого-то параметра уменьшается, и это не может продолжаться бесконечно.

7.1.1. Написать рекурсивную процедуру вычисления факториала целого положительного числа n (т. е. произведения $1 \cdot 2 \cdots n$, обозначаемого $n!$).

Решение. Используем равенства $1! = 1$, $n! = (n-1)! \cdot n$.

```
procedure factorial (n: integer; var fact: integer);
| {положить fact равным факториалу числа n}
begin
| if n=1 then begin
| | fact:=1;
| end else begin {n>1}
| | factorial (n-1, fact);
| | {fact = (n-1)!}
| | fact:= fact*n;
| end;
end;
```

□

С использованием процедур-функций можно написать так:

```
function factorial (n: integer): integer;
begin
| if n=1 then begin
| | factorial:=1;
| end else begin {n>1}
| | factorial:=  factorial (n-1)*n;
| end;
end;
```

Обратите внимание на некоторую двойственность использования имени `factorial` внутри описания функции: оно обозначает как переменную, так и вызываемую рекурсивно функцию. К счастью, в нашем случае они различаются по скобкам после имени, но если бы функция была без параметров, то дело было бы плохо. (Стандартная, но трудно находимая ошибка возникает, если автор программы на паскале полагает, что он использует значение переменной, а компилятор в этом месте видит рекурсивный вызов.)

7.1.2. Обычно факториал определяют и для нуля, считая, что $0! = 1$. Изменить программы соответственно. □

7.1.3. Написать рекурсивную программу возведения в целую неотрицательную степень. □

7.1.4. То же, если требуется, чтобы глубина рекурсии не превосходила $C \log n$, где n — показатель степени.

Решение.

```
function power (a,n: integer): integer;
begin
| if n = 0 then begin
| | power:= 1;
| end else if n mod 2 = 0 then begin
| | power:= power(a*a, n div 2);
| end else begin
| | power:= power(a, n-1)*a;
| end;
end;
```

□

7.1.5. Что будет, если изменить программу, приведённую в решении предыдущей задачи, заменив строку

```
power:= power(a*a, n div 2)
```

на

```
power:= power(a, n div 2)* power(a, n div 2)?
```

Решение. Программа останется правильной. Однако она станет работать медленнее. Дело в том, что теперь вызов может породить два вызова (хотя и одинаковых) вместо одного — и число вызовов быстро растёт с глубиной рекурсии. Программа по-прежнему имеет логарифмическую глубину рекурсии, но число шагов работы становится линейным вместо логарифмического.

Этот недостаток можно устранить, написав

```
t:= power(a, n div 2);
power:= t*t;
```

или воспользовавшись функцией возведения в квадрат (`sqr`). □

7.1.6. Используя команды `write(x)` лишь при $x = 0 \ldots 9$, написать рекурсивную программу печати десятичной записи целого положительного числа n.

Решение. Здесь использование рекурсии облегчает жизнь (проблема была в том, что цифры легче получать с конца, а печатать надо с начала).

```
procedure print (n:integer); {n>0}
begin
  if n<10 then begin
    write (n);
  end else begin
    print (n div 10);
    write (n mod 10);
  end;
end;
```

□

7.1.7. Игра «Ханойские башни» состоит в следующем. Есть три стержня. На первый из них надета пирамидка из N колец (большие кольца снизу, меньшие сверху). Требуется переместить кольца на другой стержень. Разрешается перекладывать кольца со стержня на стержень, но класть большее кольцо поверх меньшего нельзя. Составить программу, указывающую требуемые действия.

Решение. Напишем рекурсивную процедуру перемещения i верхних колец с m-го стержня на n-ый (остальные кольца предполагаются большими по размеру и лежат на стержнях без движения).

```
procedure move(i,m,n: integer);
| var s: integer;
begin
| if i = 1 then begin
| | writeln ('сделать ход ', m, '->', n);
| end else begin
| | s:=6-m-n; {s - третий стержень: сумма номеров равна 6}
| | move (i-1, m, s);
| | writeln ('сделать ход ', m, '->', n);
| | move (i-1, s, n);
| end;
end;
```

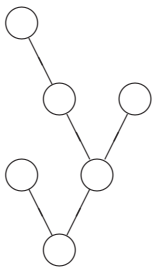
(Сначала переносится пирамидка из i-1 колец на третью палочку. После этого i-ое кольцо освобождается, и его можно перенести куда следует. Остаётся положить на него пирамидку.) □

7.1.8. Написать рекурсивную программу суммирования массива a: array [1..n] of integer.

[Указание. Рекурсивно определяемая функция должна иметь дополнительный параметр — число складываемых элементов.] □

7.2. Рекурсивная обработка деревьев

Двоичным деревом называется картинка вроде такой:



Нижняя вершина называется *корнем*. Из каждой вершины могут идти две линии: влево вверх и вправо вверх. Вершины, куда они ведут,

называются *левым и правым сыновьями* исходной вершины. Вершина может иметь двух сыновей, а может иметь только одного сына (левого или правого). Она может и вовсе не иметь сыновей, и в этом случае называется *листом*.

Пусть x — какая-то вершина двоичного дерева. Она сама вместе с сыновьями, внуками, правнуками и т. д. образует поддерево с корнем в x — *поддерево потомков* x.

В следующих задачах мы предполагаем, что вершины дерева пронумерованы целыми положительными числами, причём номера всех вершин различны. Мы считаем, что номер корня хранится в переменной `root`. Мы считаем, что имеются два массива

```
l,r: array [1..N] of integer
```

и левый и правый сын вершины с номером `i` имеют соответственно номера `l[i]` и `r[i]`. Если вершина с номером `i` не имеет левого (или правого) сына, то `l[i]` (соответственно `r[i]`) равно `0`. (По традиции при записи программ мы используем вместо нуля константу `nil`, равную нулю.)

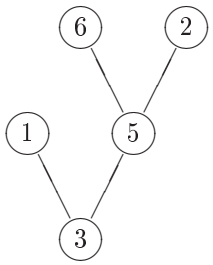
Здесь `N` — достаточно большое натуральное число (номера всех вершин не превосходят `N`). Отметим, что номер вершины никак не связан с её положением в дереве и что не все числа от `1` до `N` обязаны быть номерами вершин (и, следовательно, часть данных в массивах `l` и `r` — это мусор).

7.2.1. Пусть `N` = 7, `root` = 3, массивы `l` и `r` таковы:

i	1	2	3	4	5	6	7
l[i]	0	0	1	0	6	0	7
r[i]	0	0	5	3	2	0	7

Нарисовать соответствующее дерево.

Ответ.



□

7.2.2. Написать программу подсчёта числа вершин в дереве.

Решение. Рассмотрим функцию `n(x)`, равную числу вершин в поддереве с корнем в вершине номер `x`. Считаем, что `n(nil)` = 0 (полагая соответствующее поддерево пустым), и не заботимся о значениях `nil(s)` для чисел `s`, не являющихся номерами вершин. Рекурсивная программа для `n` такова:

```
function n(x:integer):integer;
begin
| if x = nil then begin
| | n:= 0;
| end else begin
| | n:= n(l[x]) + n(r[x]) + 1;
| end;
end;
```

(Число вершин в поддереве над вершиной `x` равно сумме чисел вершин над её сыновьями плюс она сама.) Глубина рекурсии конечна, так как с каждым шагом высота соответствующего поддерева уменьшается. □

7.2.3. Написать программу подсчёта числа листьев в дереве.

Ответ.

```
function n (x:integer):integer;
begin
| if x = nil then begin
| | n:= 0;
| end else if (l[x]=nil) and (r[x]=nil) then begin {лист}
| | n:= 1;
| end else begin
| | n:= n(l[x]) + n(r[x]);
| end;
end;
```

□

7.2.4. Написать программу подсчёта высоты дерева (корень имеет высоту 0, его сыновья — высоту 1, внуки — 2 и т. п.; высота дерева — это максимум высот его вершин).

[Указание. Рекурсивно определяется функция `f(x)` = высота поддерева с корнем в `x`.] □

7.2.5. Написать программу, которая по заданному `n` считает число всех вершин высоты `n` (в заданном дереве). □

Вместо подсчёта количества вершин того или иного рода можно просить напечатать список этих вершин (в том или ином порядке).

7.2.6. Написать программу, которая печатает (по одному разу) все вершины дерева.

Решение. Процедура `print_subtree(x)` печатает все вершины поддерева с корнем в `x` по одному разу; главная программа содержит вызов `print_subtree(root)`.

```
procedure print_subtree (x:integer);
begin
| if x = nil then begin
| | {ничего не делать}
| end else begin
| | writeln (x);
| | print_subtree (l[x]);
| | print_subtree (r[x]);
| end;
end;
```

Данная программа печатает сначала корень поддерева, затем поддерево над левым сыном, а затем над правым. Три строки в `else`-части могут быть переставлены 6 способами, и каждый из этих способов даёт свой порядок печати вершин. □

7.3. Порождение комбинаторных объектов, перебор

Рекурсивные программы являются удобным способом порождения комбинаторных объектов заданного вида. Мы решим заново несколько задач соответствующей главы.

7.3.1. Написать программу, которая печатает по одному разу все последовательности длины `n`, составленные из чисел `1`...`k` (их количество равно k^n).

Решение. Программа будет оперировать с массивом `a[1]`...`a[n]` и числом `t`. Рекурсивная процедура `generate` печатает все последовательности, начинающиеся на `a[1]`...`a[t]`; после её окончания `t` и `a[1]`...`a[t]` имеют то же значение, что и в начале:

```
procedure generate;
| var i,j : integer;
```

```
begin
| if t = n then begin
| | for i:=1 to n do begin
| | | write(a[i]);
| | end;
| | writeln;
| end else begin {t < n}
| | for j:=1 to k do begin
| | | t:=t+1;
| | | a[t]:=j;
| | | generate;
| | | t:=t-1;
| | end;
| end;
end;
```

Основная программа теперь состоит из двух операторов:

```
t:=0; generate;
```

□

Замечание. Команды `t:=t+1` и `t:=t-1` для экономии можно вынести из цикла `for`.

7.3.2. Написать программу, которая печатала бы все перестановки чисел `1`...`n` по одному разу.

Решение. Программа оперирует с массивом `a[1]`...`a[n]`, в котором хранится перестановка чисел `1`...`n`. Рекурсивная процедура `generate` в такой ситуации печатает все перестановки, которые на первых `t` позициях совпадают с перестановкой `a`; по выходе из неё переменные `t` и `a` имеют те же значения, что и до входа. Основная программа такова:

```
for i:=1 to n do begin a[i]:=i; end;
t:=0;
generate;
```

Вот описание процедуры:

```
procedure generate;
| var i,j : integer;
begin
| if t = n then begin
| | for i:=1 to n do begin
| | | write(a[i]);
| | end;
| | writeln;
```

```
  end else begin {t < n}
    for j:=t+1 to n do begin
      поменять местами a[t+1] и a[j]
      t:=t+1;
      generate;
      t:=t-1;
      поменять местами a[t+1] и a[j]
    end;
  end;
end;
```

□

7.3.3. Напечатать (по одному разу) все последовательности из n нулей и единиц, содержащие ровно k единиц. □

7.3.4. Напечатать все возрастающие последовательности длины k, элементами которых являются натуральные числа от 1 до n. (Предполагается, что $k \leqslant n$, иначе таких последовательностей не существует.)

Решение. Программа оперирует с массивом a[1] ... a[k] и целой переменной t. Предполагая, что a[1] ... a[t] — возрастающая последовательность натуральных чисел из отрезка 1 ... n, рекурсивно определённая процедура generate печатает все её возрастающие продолжения длины k. (При этом t и a[1] ... a[t] в конце такие же, как в начале.)

```
procedure generate;
  var i: integer;
begin
  if t = k then begin
    печатать a[1]..a[k]
  end else begin
    t:=t+1;
    for i:=a[t-1]+1 to t-k+n do begin
      a[t]:=i;
      generate;
    end;
    t:=t-1;
  end;
end;
```

Замечание. Цикл for мог бы иметь верхней границей n (вместо $t - k + n$). Наш вариант экономит часть работы, учитывая тот факт, что предпоследний (k-1-ый) член не может превосходить n-1, k-2-ой член не может превосходить n-2 и т. п.

Основная программа теперь выглядит так:

```
t:=1;
for j:=1 to 1-k+n do begin
| a[1]:=j;
| generate;
end;
```

Можно было бы добавить к массиву a слева фиктивный элемент a[0] = = 0, положить t = 0 и ограничиться единственным вызовом процедуры generate. □

7.3.5. Перечислить все представления положительного целого числа n в виде суммы последовательности невозрастающих целых положительных слагаемых.

Решение. Программа оперирует с массивом a[1..n] (максимальное число слагаемых равно n) и с целой переменной t. Предполагая, что a[1] ... a[t] — невозрастающая последовательность целых чисел, сумма которых не превосходит n, процедура generate печатает все представления требуемого вида, продолжающие эту последовательность. Для экономии вычислений сумма a[1] + ... + a[t] хранится в специальной переменной s.

```
procedure generate;
| var i: integer;
begin
| if s = n then begin
| | печатать последовательность a[1]..a[t]
| end else begin
| | for i:=1 to min(a[t], n-s) do begin
| | | t:=t+1;
| | | a[t]:=i;
| | | s:=s+i;
| | | generate;
| | | s:=s-i;
| | | t:=t-1;
| | end;
| end;
end;
```

Основная программа при этом может быть такой:

```
t:=1;
for j:=1 to n do begin
| a[1]:=j
```

```
  s:=j;
  generate;
end;
```

Замечание. Можно немного сэкономить, вынеся операции увеличения и уменьшения `t` из цикла, а также не возвращая `s` каждый раз к исходному значению (увеличивая его на 1 и возвращая к исходному значению в конце). Кроме того, добавив фиктивный элемент `a[0] = n`, можно упростить основную программу:

```
t:=0; s:=0; a[0]:=n; generate;
```
□

7.3.6. Написать рекурсивную программу обхода дерева (используя те же команды и проверки, что и в главе 3 (Обход дерева).

Решение. Процедура `обработать_над` обрабатывает все листья над текущей вершиной и заканчивает работу в той же вершине, что и начала. Вот её рекурсивное описание:

```
procedure обработать_над;
begin
  if есть_сверху then begin
    вверх_налево;
    обработать_над;
    while есть_справа do begin
      вправо;
      обработать_над;
    end;
    вниз;
  end else begin
    обработать;
  end;
end;
```
□

7.4. Другие применения рекурсии

Топологическая сортировка. Представим себе n чиновников, каждый из которых выдаёт справки определённого вида. Мы хотим получить все эти справки, соблюдая установленные ограничения: у каждого чиновника есть список справок, которые нужно собрать перед обращением к нему. Дело безнадёжно, если схема зависимостей имеет цикл (справку A нельзя получить без B, B без C,..., Y без Z и Z без A).

Предполагая, что такого цикла нет, требуется составить план, указывающий один из возможных порядков получения справок.

Изображая чиновников точками, а зависимости — стрелками, приходим к такой формулировке. Имеется n точек, пронумерованных от 1 до n. Из каждой точки ведёт несколько (возможно, 0) стрелок в другие точки. (Такая картинка называется *ориентированным графом*.) Циклов нет. Требуется расположить вершины графа (точки) в таком порядке, чтобы конец любой стрелки предшествовал её началу. Эта задача называется *топологической сортировкой*.

7.4.1. Доказать, что это всегда возможно.

Решение. Из условия отсутствия циклов вытекает, что есть вершина, из которой вообще не выходит стрелок (иначе можно двигаться по стрелкам, пока не зациклимся). Её будем считать первой. Выкидывая эту вершину и все соседние стрелки, мы сводим задачу к графу с меньшим числом вершин и продолжаем рассуждение по индукции. □

7.4.2. Предположим, что ориентированный граф без циклов хранится в такой форме: для каждого `i` от `1` до `n` в `num[i]` хранится число выходящих из `i` стрелок, в `adr[i][1]`, ..., `adr[i][num[i]]` — номера вершин, куда эти стрелки ведут. Составить (рекурсивный) алгоритм, который производит топологическую сортировку не более чем за $C \cdot (\mathtt{n} + \mathtt{m})$ действий, где `m` — число рёбер графа (стрелок).

Замечание. Непосредственная реализация приведённого выше доказательства существования не даёт требуемой оценки; её приходится немного подправить.

Решение. Наша программа будет печатать номера вершин. В массиве

```
printed: array[1..n] of boolean
```

мы будем хранить сведения о том, какие вершины напечатаны (и корректировать их одновременно с печатью вершины). Будем говорить, что напечатанная последовательность вершин корректна, если никакая вершина не напечатана дважды и для любого номера `i`, входящего в эту последовательность, все вершины, в которые ведут стрелки из `i`, напечатаны, и притом до `i`.

```
procedure add (i: 1..n);
| {дано: напечатанное корректно;}
| {надо: напечатанное корректно и включает вершину i}
```

```
begin
| if printed [i] then begin {вершина i уже напечатана}
| | {ничего делать не надо}
| end else begin
| | {напечатанное корректно}
| | for j:=1 to num[i] do begin
| | | add(adr[i][j]);
| | end;
| | {напечатанное корректно, все вершины, в которые из
| |  i ведут стрелки, уже напечатаны - так что можно
| |  печатать i, не нарушая корректности}
| | if not printed[i] then begin
| | | writeln(i); printed [i]:= TRUE;
| | end;
| end;
end;
```

Основная программа:

```
for i:=1 to n do begin
| printed[i]:= FALSE;
end;
for i:=1 to n do begin
| add(i)
end;
```

К оценке времени работы мы вскоре вернёмся.

7.4.3. В приведённой программе можно выбросить проверку, заменив

```
if not printed[i] then begin
| writeln(i); printed [i]:= TRUE;
end;
```

на

```
writeln(i); printed [i]:= TRUE;
```

Почему? Как изменится спецификация процедуры?

Решение. Спецификацию можно выбрать такой:

```
дано: напечатанное корректно
надо: напечатанное корректно и включает вершину i;
      все вновь напечатанные вершины доступны из i.
```

□

7.4.4. Где использован тот факт, что граф не имеет циклов?

Решение. Мы опустили доказательство конечности глубины рекурсии. Для каждой вершины рассмотрим её «глубину» — максимальную длину пути по стрелкам, из неё выходящего. Условие отсутствия циклов гарантирует, что эта величина конечна. Из вершины нулевой глубины стрелок не выходит. Глубина конца стрелки по крайней мере на 1 меньше, чем глубина начала. При работе процедуры `add(i)` все рекурсивные вызовы `add(j)` относятся к вершинам меньшей глубины. □

Вернёмся к оценке времени работы. Сколько вызовов `add(i)` возможно для какого-то фиксированного `i`? Прежде всего ясно, что первый из них печатает `i`, остальные сведутся к проверке того, что `i` уже напечатано. Ясно также, что вызовы `add(i)` индуцируются «печатающими» (первыми) вызовами `add(j)` для тех `j`, из которых в `i` ведёт ребро. Следовательно, число вызовов `add(i)` равно числу входящих в `i` рёбер (стрелок). При этом все вызовы, кроме первого, требуют $O(1)$ операций, а первый требует времени, пропорционального числу исходящих из `i` стрелок. (Не считая времени, уходящего на выполнение `add(j)` для концов `j` выходящих рёбер.) Отсюда видно, что общее время пропорционально числу рёбер (плюс число вершин). □

Связная компонента графа. *Неориентированный граф* — набор точек (вершин), некоторые из которых соединены линиями (рёбрами). Неориентированный граф можно считать частным случаем ориентированного графа, в котором для каждой стрелки есть обратная.

Связной компонентой вершины `i` называется множество всех тех вершин, в которые можно попасть из `i`, идя по рёбрам графа. (Поскольку граф неориентированный, отношение «`j` принадлежит связной компоненте `i`» является отношением эквивалентности.)

7.4.5. Дан неориентированный граф (для каждой вершины указано число соседей и массив номеров соседей, как в задаче о топологической сортировке). Составить алгоритм, который по заданному `i` печатает все вершины связной компоненты `i` по одному разу (и только их). Число действий не должно превосходить $C \cdot$ (общее число вершин и рёбер в связной компоненте).

Решение. Программа в процессе работы будет «закрашивать» некоторые вершины графа. Незакрашенной частью графа будем называть то, что останется, если выбросить все закрашенные вершины и ведущие в них рёбра. Процедура `add(i)` закрашивает связную компоненту `i` *в незакрашенной части графа* (и не делает ничего, если вершина `i` уже закрашена).

```
procedure  add (i:1..n);
begin
| if вершина i закрашена then begin
| | ничего делать не надо
| end else begin
| | закрасить i (напечатать и пометить как закрашенную)
| | для всех j, соседних с i
| | | add(j);
| | end;
| end;
end;
```

Докажем, что эта процедура действует правильно (в предположении, что рекурсивные вызовы работают правильно). В самом деле, ничего, кроме связной компоненты незакрашенного графа, она закрасить не может. Проверим, что вся она будет закрашена. Пусть `k` — вершина, доступная из вершины `i` по пути $\texttt{i} \to \texttt{j} \to \ldots \to \texttt{k}$, проходящему только по незакрашенным вершинам. Будем рассматривать только пути, не возвращающиеся снова в `i`. Из всех таких путей выберем путь с наименьшим `j` (в порядке просмотра соседей в процедуре). Тогда при рассмотрении предыдущих соседей ни одна из вершин пути $\texttt{j} \to \ldots \to \texttt{k}$ не будет закрашена (иначе `j` не было бы минимальным) и потому `k` окажется в связной компоненте незакрашенного графа к моменту вызова `add(j)`. Что и требовалось.

Чтобы установить конечность глубины рекурсии, заметим, что на каждом уровне рекурсии число незакрашенных вершин уменьшается хотя бы на 1.

Оценим число действий. Каждая вершина закрашивается не более одного раза — при первым вызове `add(i)` с данным `i`. Все последующие вызовы происходят при закрашивании соседей — количество таких вызовов не больше числа соседей — и сводятся к проверке того, что вершина `i` уже закрашена. Первый же вызов состоит в просмотре всех соседей и рекурсивных вызовах `add(j)` для всех них. Таким образом, общее число действий, связанных с вершиной `i`, не превосходит константы, умноженной на число её соседей. Отсюда и вытекает требуемая оценка. □

7.4.6. Решить ту же задачу для ориентированного графа (напечатать все вершины, доступные из данной по стрелкам; граф может содержать циклы).

Ответ. Годится по существу та же программа (строку «для всех соседей» надо заменить на «для всех вершин, куда ведут стрелки»). □

Следующий вариант задачи о связной компоненте имеет скорее теоретическое значение (и называется *теоремой Сэвича*).

7.4.7. Ориентированный граф имеет 2^n вершин (двоичные слова длины n) и задан в виде функции `есть_ребро`, которая по двум вершинам x и y сообщает, есть ли в графе ребро из x в y. Составить алгоритм, который для данной пары вершин u и v определяет, есть ли путь (по рёбрам) из u в v, используя память, ограниченную многочленом от n. (Время при этом может быть — и будет — очень большим.)

[Указание. Использовать рекурсивную процедуру, выясняющую, существует ли путь из x в y длины не более 2^k (и вызывающую себя с уменьшенным на единицу значением k).] □

Быстрая сортировка Хоара. В заключение приведём рекурсивный алгоритм сортировки массива, который на практике является одним из самых быстрых. Пусть дан массив `a[1]`...`a[n]`. Рекурсивная процедура `sort(l,r:integer)` сортирует участок массива с индексами из полуинтервала $(l, r]$, то есть `a[l+1]`...`a[r]`, не затрагивая остального массива.

```
procedure sort (l,r: integer);
begin
  if l = r then begin
    ничего делать не надо - участок пуст
  end else begin
    выбрать случайное число s в полуинтервале (l,r]
    b := a[s]
    переставить элементы сортируемого участка так, чтобы
      сначала шли элементы, меньшие b - участок (l,ll]
      затем элементы, равные b        - участок (ll,rr]
      затем элементы, большие b       - участок (rr,r]
    sort (l,ll);
    sort (rr,r);
  end;
end;
```

Разделение элементов сортируемого участка на три категории (меньшие, равные, больше) рассматривалась в главе 1, с. 36 (это можно сделать за время, пропорциональное длине участка). Конечность глубины рекурсии гарантируется тем, что длина сортируемого участка на каждом уровне рекурсии уменьшается хотя бы на 1.

7.4.8. (Для знакомых с основами теории вероятностей). Доказать, что математическое ожидание числа операций при работе этого ал-

горитма не превосходит $Cn\log n$, причём константа C не зависит от сортируемого массива.

[Указание. Пусть $T(n)$ — максимум математического ожидания числа операций для всех входов длины n. Из текста процедуры вытекает такое неравенство:

$$T(n) \leqslant Cn + \frac{1}{n}\sum_{k+l=n-1}\big(T(k)+T(l)\big)$$

Первый член соответствует распределению элементов на меньшие, равные и большие. Второй член — это среднее математическое ожидание для всех вариантов случайного выбора. (Строго говоря, поскольку среди элементов могут быть равные, в правой части вместо $T(k)$ и $T(l)$ должны стоять максимумы $T(x)$ по всем x, не превосходящим k или l, но это не мешает дальнейшим рассуждениям.) Далее индукцией по n нужно доказывать оценку $T(n) \leqslant C'n\ln n$. При этом для вычисления среднего значения $x\ln x$ по всем $x = 1,\ldots,n-1$ нужно вычислять $\int_1^n x\ln x\,dx$ по частям как $\int \ln x\,d(x^2)$. При достаточно большом C' член Cn в правой части перевешивается за счёт интеграла $\int x^2\,d\ln x$, и индуктивный шаг проходит.] □

7.4.9. Имеется массив из n различных целых чисел и число k. Требуется найти k-ое по величине число в этом массиве, сделав не более Cn действий, где C — некоторая константа, не зависящая от k и n.

Замечание. Сортировка позволяет очевидным образом сделать это за $Cn\log n$ действий. Очевидный способ: найти наименьший элемент, затем найти второй, затем третий, ..., k-ый требует порядка kn действий, то есть не годится (константа при n зависит от k).

[Указание. Изящный (хотя практически и бесполезный — константы слишком велики) способ сделать это таков:

А. Разобьём наш массив на $n/5$ групп, в каждой из которых по 5 элементов. Каждую группу упорядочим.

Б. Рассмотрим средние элементы всех групп и перепишем их в массив из $n/5$ элементов. С помощью рекурсивного вызова найдём средний по величине элемент этого массива.

В. Сравним этот элемент со всеми элементами исходного массива: они разделятся на большие его и меньшие его (и один равный ему). Подсчитав количество тех и других, мы узнаем, в какой из этих частей должен находится искомый (k-ый) элемент и каков он там по порядку.

Г. Применим рекурсивно наш алгоритм к выбранной части.

Пусть $T(n)$ — максимально возможное число действий, если этот способ применять к массивам из не более чем n элементов (k может быть каким угодно). Имеем оценку:

$$T(n) \leqslant Cn + T(n/5) + T(\text{примерно } 0{,}7n).$$

Последнее слагаемое объясняется так: при разбиении на части каждая часть содержит не менее $0{,}3n$ элементов. В самом деле, если x — средний из средних, то примерно половина всех средних меньше x. А если в пятёрке средний элемент меньше x, то ещё два заведомо меньше x. Тем самым по крайней мере $3/5$ от половины элементов меньше x.

Теперь по индукции можно доказать оценку $T(n) \leqslant Cn$ (решающую роль при этом играет то обстоятельство, что $1/5 + 0{,}7 < 1$).] □

8. КАК ОБОЙТИСЬ БЕЗ РЕКУРСИИ

Для универсальных языков программирования (каковым является паскаль) рекурсия не даёт ничего нового: для всякой рекурсивной программы можно написать эквивалентную программу без рекурсии. Мы не будем доказывать этого, а продемонстрируем некоторые приёмы, позволяющие избавиться от рекурсии в конкретных ситуациях.

Зачем это нужно? Ответ прагматика мог бы быть таким: во многих компьютерах (в том числе, к сожалению, и в современных, использующих так называемые RISC-процессоры), рекурсивные программы в несколько раз медленнее соответствующих нерекурсивных программ. Ещё один возможный ответ: в некоторых языках программирования рекурсивные программы запрещены. А главное, при удалении рекурсии возникают изящные и поучительные конструкции.

8.1. Таблица значений (динамическое программирование)

8.1.1. Следующая рекурсивная процедура вычисляет числа сочетаний (биномиальные коэффициенты). Написать эквивалентную нерекурсивную программу.

```
function C(n,k: integer):integer;
| {n >= 0; 0 <=  k <=n}
begin
| if (k = 0) or (k = n) then begin
| | C:=1;
| end else begin {0<k<n}
| | C:= C(n-1,k-1)+C(n-1,k)
| end;
end;
```

Замечание. C_n^k — число k-элементных подмножеств n-элементного множества. Соотношение $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ получится, если мы фиксируем некоторый элемент n-элементного множества и отдельно подсчитаем k-элементные подмножества, включающие и не включающие этот элемент. Таблица значений C_n^k

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
. & . & . & . & . & . & . & . & .
\end{array}
$$

называется треугольником Паскаля (того самого). В нём каждый элемент, кроме крайних единиц, равен сумме двух стоящих над ним.

Решение. Можно воспользоваться формулой

$$C_n^k = \frac{n!}{k!\,(n-k)!}$$

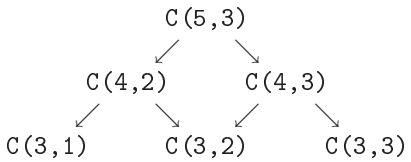
Мы, однако, не будем этого делать, так как хотим продемонстрировать более общие приёмы устранения рекурсии. Вместо этого составим таблицу значений функции `C(n,k)` $= C_n^k$, заполняя её для $n = 0, 1, 2, \ldots$, пока не дойдём до интересующего нас элемента. □

8.1.2. Что можно сказать о времени работы рекурсивной и нерекурсивной версий в предыдущей задаче? Тот же вопрос о памяти.

Решение. Таблица занимает место порядка n^2, его можно сократить до n, если заметить, что для вычисления следующей строки треугольника Паскаля нужна только предыдущая. Время работы остаётся порядка n^2. Рекурсивная программа требует существенно большего времени: вызов `C(n,k)` сводится к двум вызовам для `C(n-1,..)` , те — к четырём вызовам для `C(n-2,..)` и так далее. Таким образом, время оказывается экспоненциальным (порядка 2^n). Используемая рекурсивной версией память пропорциональна n — умножаем глубину рекурсии (n) на количество памяти, используемое одним экземпляром процедуры (константа). □

Кардинальный выигрыш во времени при переходе от рекурсивной версии к нерекурсивной связан с тем, что в рекурсивном варианте одни и те же вычисления происходят много раз. Например, вызов `C(5,3)`

в конечном счёте порождает два вызова `C(3,2)`:



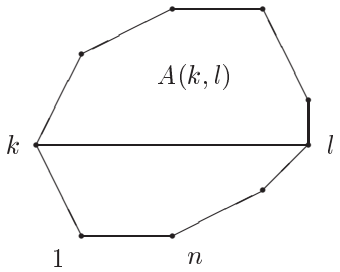
Заполняя таблицу, мы каждую клетку заполняем только однажды — отсюда и экономия. Этот приём называется *динамическим программированием*, и применим в тех случаях, когда объём хранимой в таблице информации оказывается не слишком большим.

8.1.3. Порассуждать на ту же тему на примере рекурсивной и (простейшей) нерекурсивной программ для вычисления чисел Фибоначчи, заданных соотношением

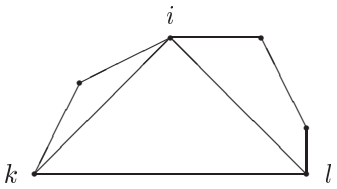
$$\Phi_1 = \Phi_2 = 1; \quad \Phi_n = \Phi_{n-1} + \Phi_{n-2} \ (n > 2). \qquad \square$$

8.1.4. Дан выпуклый n-угольник (заданный координатами своих вершин в порядке обхода). Его разрезают на треугольники диагоналями, для чего необходимо $n-2$ диагонали (это можно доказать индукцией по n). Стоимостью разрезания назовём сумму длин всех использованных диагоналей. Найти минимальную стоимость разрезания. Число действий должно быть ограничено некоторым многочленом от n. (Перебор не подходит, так как число вариантов не ограничено многочленом.)

Решение. Будем считать, что вершины пронумерованы от 1 до n и идут по часовой стрелке. Пусть k, l — номера вершин, причём $l > k$. Через $A(k,l)$ обозначим многоугольник, отрезаемый от нашего хордой $k-l$. (Эта хорда разрезает многоугольник на два, один из которых включает сторону $1-n$; через $A(k,l)$ мы обозначаем другой.) Исходный многоугольник естественно обозначить $A(1,n)$. При $l = k+1$ получается «двуугольник» с совпадающими сторонами.



Через $a(k,l)$ обозначим стоимость разрезания многоугольника $A(k,l)$ диагоналями на треугольники. Напишем рекуррентную формулу для $a(k,l)$. При $l = k+1$ получается двуугольник, и мы полагаем $a(k,l) = 0$. При $l = k+2$ получается треугольник, и в этом случае также $a(k,l) = 0$. Пусть $l > k+2$.



Хорда $k-l$ является стороной многоугольника $A(k,l)$ и, следовательно, стороной одного из треугольников, на которые он разрезан. Противоположной вершиной i этого треугольника может быть любая из вершин $k+1, \ldots, l-1$, и минимальная стоимость разрезания может быть вычислена как

$$\min\{(\text{длина хорды } k-i) + (\text{длина хорды } i-l) + a(k,i) + a(i,l)\}$$

по всем $i = k+1, \ldots, l-1$. При этом надо учесть, что при $q = p+1$ хорда $p-q$ — не хорда, а сторона, и её длину надо считать равной 0 (по стороне разрез не проводится).

Составив таблицу для $a(k,l)$ и заполняя её в порядке возрастания числа вершин (равного $l-k+1$), мы получаем программу, использующую память порядка n^2 и время порядка n^3 (однократное применение рекуррентной формулы требует выбора минимума из не более чем n чисел). □

8.1.5. Матрицей размера $m \times n$ называется прямоугольная таблица из m строк и n столбцов, заполненная числами. Матрицу размера $m \times n$ можно умножить на матрицу размера $n \times k$ (ширина левого сомножителя должна равняться высоте правого), и получается матрица размером $m \times k$. Ценой такого умножения будем считать произведение mnk (таково число умножений, которые нужно выполнить при стандартном способе умножения — но сейчас это нам не важно). Умножение матриц ассоциативно, поэтому произведение s матриц можно вычислять в разном порядке. Для каждого порядка подсчитаем суммарную цену всех матричных умножений. Найти минимальную цену вычисления произведения, если известны размеры всех матриц. Число действий должно быть ограничено многочленом от числа матриц.

Пример. Матрицы размером 2×3, 3×4, 4×5 можно перемножать двумя способами. В первом цена равна $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 24 + 40 = 64$, во втором цена равна $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 90$.

Решение. Представим себе, что первая матрица написана на отрезке $[0, 1]$, вторая — на отрезке $[1, 2]$, ..., s-ая — на отрезке $[s-1, s]$. Матрицы на отрезках $[i-1, i]$ и $[i, i+1]$ имеют общий размер, позволяющий их перемножить. Обозначим его через $d[i]$. Таким образом, исходным данным в задаче является массив $d[0] \ldots d[s]$.

Через $a(i, j)$ обозначим минимальную цену вычисления произведения матриц на участке $[i, j]$ (при $0 \leqslant i < j \leqslant s$). Искомая величина равна $a(0, s)$. Величины $a(i, i+1)$ равны нулю (матрица одна и перемножать ничего не надо). Рекуррентная формула будет такой:

$$a(i, j) = \min\{a(i, k) + a(k, j) + d[i]d[k]d[j]\}$$

где минимум берётся по всем возможных местам последнего умножения, то есть по всем $k = i+1, \ldots, j-1$. В самом деле, произведение матриц на отрезке $[i, k]$ есть матрица размера $d[i]d[k]$, произведение матриц на отрезке $[k, j]$ имеет размер $d[k]d[j]$, и цена вычисления их произведения равна $d[i]d[k]d[j]$. □

Замечание. Две последние задачи похожи. Это сходство станет яснее, если написать матрицы-множители на сторонах $1-2$, $2-3$, ..., $(s-1)-s$ многоугольника, а на каждой хорде $i-j$ написать произведение всех матриц, стягиваемых этой хордой.

8.1.6. Железная дорога с односторонним движением имеет n станций. Известны цены билетов от i-ой станции до j-ой (при $i < j$ — в обратную сторону проезда нет). Найти минимальную стоимость проезда от начала до конца (с учётом возможной экономии за счёт пересадок). □

8.1.7. Задано конечное множество с бинарной операцией (вообще говоря, не коммутативной и даже не ассоциативной). Имеется n элементов $a_1, \ldots, a_n$ этого множества и ещё один элемент x. Проверить, можно ли так расставить скобки в произведении $a_1 \times \ldots \times a_n$, чтобы в результате получился x. Число операций должно не превосходить Cn^3 для некоторой константы C (зависящей от числа элементов в выбранном конечном множестве).

Решение. Заполняем таблицу, в которой для каждого участка $a_i \ldots a_j$ нашего произведения хранится список всех возможных его значений (при разной расстановке скобок). □

По существу этот же приём применяется в полиномиальном алгоритме проверки принадлежности слова произвольному контекстно-свободному языку (см. главу 15).

Следующая задача (задача о рюкзаке) уже упоминалась в главе 3.

8.1.8. Имеется n положительных целых чисел $x_1, \ldots, x_n$ и число N. Выяснить, можно ли получить N, складывая некоторые из чисел $x_1, \ldots$ $\ldots, x_n$. Число действий должно быть порядка Nn.

[Указание. После i шагов хранится множество тех чисел на отрезке $0 \ldots N$, которые представимы в виде суммы некоторых из $x_1 \ldots x_i$.] □

Замечание. Мы видели, что замена рекурсивной программы на заполнение таблицы значений иногда позволяет уменьшить число действий. Примерно того же эффекта можно добиться иначе: оставить программу рекурсивной, но в ходе вычислений запоминать уже вычисленные значения, а перед очередным вычислением проверять, нет ли уже готового значения.

8.2. Стек отложенных заданий

Другой приём устранения рекурсии продемонстрируем на примере задачи о ханойских башнях.

8.2.1. Написать нерекурсивную программу для нахождения последовательности перемещений колец в задаче о ханойских башнях.

Решение. Вспомним рекурсивную программу, перекладывающую `i` верхних колец с `m` на `n`:

```
procedure move(i,m,n: integer);
| var s: integer;
begin
| if i = 1 then begin
| | writeln ('сделать ход ', m, '->', n);
| end else begin
| | s:=6-m-n; {s - третий стержень: сумма номеров равна 6}
| | move (i-1, m, s);
| | writeln ('сделать ход ', m, '->', n);
| | move (i-1, s, n);
| end;
end;
```

Видно, что задача «переложить `i` верхних дисков с `m`-го стержня на `n`-ый» сводится к трём задачам того же типа: двум задачам с `i-1` дис-

ками и к одной задаче с единственным диском. Занимаясь этими задачами, важно не позабыть, что́ ещё осталось сделать.

Для этой цели заведём *стек отложенных заданий*, элементами которого будут тройки $\langle \mathtt{i}, \mathtt{m}, \mathtt{n} \rangle$. Каждая такая тройка интерпретируется как заказ «переложить i верхних дисков с m-го стержня на n-ый». Заказы упорядочены в соответствии с требуемым порядком их выполнения: самый срочный — вершина стека. Получаем такую программу:

```
procedure move(i,m,n: integer);
begin
| сделать стек заказов пустым
| положить в стек тройку <i,m,n>
| {инвариант: осталось выполнить заказы в стеке}
| while стек непуст do begin
| | удалить верхний элемент, переложив его в <j,p,q>
| | if j = 1 then begin
| | | writeln ('сделать ход', p, '->', q);
| | end else begin
| | | s:=6-p-q;
| | |      {s - третий стержень: сумма номеров равна 6}
| | | положить в стек тройки <j-1,s,q>, <1,p,q>, <j-1,p,s>
| | end;
| end;
end;
```

(Заметим, что первой в стек кладётся тройка, которую надо выполнять последней.) Стек троек может быть реализован как три отдельных стека. (Кроме того, в паскале есть специальный тип, называемый «запись» (record), который может быть применён.) □

8.2.2. (Сообщил А. К. Звонкин со ссылкой на Анджея Лисовского.) Для задачи о ханойских башнях есть и другие нерекурсивные алгоритмы. Вот один из них: простаивающим стержнем (не тем, с которого переносят, и не тем, на который переносят) должны быть все стержни по очереди. Другое правило: поочерёдно перемещать наименьшее кольцо и не наименьшее кольцо, причём наименьшее — по кругу. □

8.2.3. Использовать замену рекурсии стеком отложенных заданий в рекурсивной программе печати десятичной записи целого числа.

Решение. Цифры добываются с конца и закладываются в стек, а затем печатаются в обратном порядке. □

8.2.4. Написать нерекурсивную программу, печатающую все вершины двоичного дерева.

Решение. В этом случае стек отложенных заданий будет содержать заказы двух сортов: «напечатать данную вершину» и «напечатать все вершины поддерева с данным корнем» (при этом `nil` считается корнем пустого дерева). Таким образом, элемент стека есть пара: ⟨тип заказа, номер вершины⟩.

Вынимая элемент из стека, мы либо сразу исполняем его (если это заказ первого типа), либо помещаем в стек три порождённых им заказа — в одном из шести возможных порядков. □

8.2.5. Что изменится, если требуется не печатать вершины двоичного дерева, а подсчитать их количество?

Решение. Печатание вершины следует заменить прибавлением единицы к счётчику. Другими словами, инвариант таков: (общее число вершин) = (счётчик) + (сумма чисел вершин в поддеревьях, корни которых лежат в стеке). □

8.2.6. Для некоторых из шести возможных порядков возможны упрощения, делающие ненужным хранение в стеке элементов двух видов. Указать некоторые из них.

Решение. Если требуемый порядок таков:

корень, левое поддерево, правое поддерево,

то заказ на печатание корня можно не закладывать в стек, а выполнять сразу.

Несколько более сложная конструкция применима для порядка

левое поддерево, корень, правое поддерево.

В этом случае все заказы в стеке, кроме самого первого (напечатать поддерево) делятся на пары:

напечатать вершину `x`, напечатать «правое поддерево» `x`

(= поддерево с корнем в правом сыне `x`). Объединив эти пары в заказы специального вида и введя переменную для отдельного хранения первого заказа, мы обойдёмся стеком однотипных заказов.

То же самое, разумеется, верно, если поменять местами левое и правое — получается ещё два порядка. □

Замечание. Другую программу печати всех вершин дерева можно построить на основе программы обхода дерева, разобранной в главе 3. Там используется команда «вниз». Поскольку теперешнее представление дерева с помощью массивов `l` и `r` не позволяет найти предка заданной вершины, придётся хранить список всех вершин на пути от корня к текущей вершине. Смотри также главу 9.

8.2.7. Написать нерекурсивный вариант программы быстрой сортировки (см. с. 131). Как обойтись стеком, глубина которого ограничена $C \log n$, где n — число сортируемых элементов?

Решение. В стек кладутся пары $\langle i, j \rangle$, интерпретируемые как отложенные задания на сортировку соответствующих участков массива. Все эти заказы не пересекаются, поэтому размер стека не может превысить n. Чтобы ограничиться стеком логарифмической глубины, будем придерживаться такого правила: глубже в стек помещать больший из возникающих двух заказов. Пусть $f(n)$ — максимальная глубина стека, которая может встретиться при сортировке массива из не более чем n элементов таким способом. Оценим $f(n)$ сверху таким способом: после разбиения массива на два участка мы сначала сортируем более короткий (храня в стеке более длинный про запас), при этом глубина стека не больше $f(n/2) + 1$, затем сортируем более длинный, так что

$$f(n) \leqslant \max(f(n/2) + 1, f(n-1)),$$

откуда очевидной индукцией получаем $f(n) = O(\log n)$. □

8.3. Более сложные случаи рекурсии

Пусть функция f с натуральными аргументами и значениями определена рекурсивно условиями

$$\begin{aligned} f(0) &= a, \\ f(x) &= h(x, f(l(x))) \quad (x > 0) \end{aligned}$$

где a — некоторое число, а h и l — известные функции. Другими словами, значение функции f в точке x выражается через значение f в точке $l(x)$. При этом предполагается, что для любого x в последовательности

$$x, l(x), l(l(x)), \ldots$$

рано или поздно встретится 0.

Если дополнительно известно, что $l(x) < x$ для всех x, то вычисление f не представляет труда: вычисляем последовательно $f(0), f(1), f(2), \ldots$

8.3.1. Написать нерекурсивную программу вычисления f для общего случая.

Решение. Для вычисления $f(x)$ вычисляем последовательность

$$l(x), l(l(x)), l(l(l(x))), \ldots$$

до появления нуля и запоминаем её, а затем вычисляем значения f в точках этой последовательности, идя справа налево. □

Ещё более сложный случай из следующей задачи вряд ли встретится на практике (а если и встретится, то проще рекурсию не устранять, а оставить). Но тем не менее: пусть функция f с натуральными аргументами и значениями определяется соотношениями

$$f(0) = a,$$
$$f(x) = h(x, f(l(x)), f(r(x))) \quad (x > 0),$$

где a — некоторое число, а l, r и h — известные функции. Предполагается, что если взять произвольное число и начать применять к нему функции l и r в произвольном порядке, то рано или поздно получится 0.

8.3.2. Написать нерекурсивную программу вычисления f.

Решение. Можно было бы сначала построить дерево, у которого в корне находится x, а в сыновьях вершины i стоят $l(i)$ и $r(i)$ — если только i не равно нулю. Затем вычислять значения функции, идя от листьев к корню. Однако есть и другой способ.

Обратной польской записью (или *постфиксной записью*) выражения называют запись, где знак функции стоит после всех её аргументов, а скобки не используются. Вот несколько примеров:

$f(2)$	2	f					
$f(g(2))$	2	g	f				
$s(2, t(7))$	2	7	t	s			
$s(2, u(2, s(5, 3))$	2	2	5	3	s	u	s

Постфиксная запись выражения позволяет удобно вычислять его с помощью *стекового калькулятора*. Этот калькулятор имеет стек, который мы будем представлять себе расположенным горизонтально (числа вынимаются и кладутся справа), и клавиши — числовые и функциональные. При нажатии на клавишу с числом это число кладётся в стек. При нажатии на функциональную клавишу соответствующая функция применяется к нескольким аргументам у вершины стека. Например, если в стеке были числа

2 3 4 5 6

и нажата функциональная клавиша s, соответствующая функции от двух аргументов, то в стеке окажутся числа

2 3 4 $s(5, 6)$.

Перейдём теперь к нашей задаче. В процессе вычисления значения функции f мы будем работать со стеком чисел, а также с последовательностью чисел и символов `f`, `l`, `r`, `h`, которую мы будем интерпретировать как последовательность нажатий клавиш на стековом калькуляторе. Инвариант такой:

> если стек чисел представляет собой текущее состояние стекового калькулятора, то после нажатия всех клавиш последовательности в стеке останется единственное число, и оно будет искомым ответом.

Пусть нам требуется вычислить значение $f(x)$. Тогда вначале мы помещаем в стек число x, а последовательность содержит единственный символ `f`. (При этом инвариант соблюдается.) Далее с последовательностью и стеком выполняются такие преобразования:

старый стек	старая последовательность	новый стек	новая последовательность
X	$x\ P$	$X\ x$	P
$X\ x$	`l` P	$X\ l(x)$	P
$X\ x$	`r` P	$X\ r(x)$	P
$X\ x\ y\ z$	`h` P	$X\ h(x,y,z)$	P
$X\ 0$	`f` P	$X\ a$	P
$X\ x$	`f` P	X	$x\ x$ `l f` x `r f h` P

Здесь x, y, z — числа, X — последовательность чисел, P — последовательность чисел и символов `f`, `l`, `r`, `h`. В последней строке предполагается, что $x \neq 0$. Эта строка соответствует равенству

$$f(x) = h(x, f(l(x)), f(r(x))).$$

Преобразования выполняются, пока последовательность не станет пуста. В этот момент в стеке окажется единственное число, которое и будет ответом. □

Замечание. Последовательность по существу представляет собой стек отложенных заданий (вершина которого находится слева).

9. РАЗНЫЕ АЛГОРИТМЫ НА ГРАФАХ

9.1. Кратчайшие пути

В этом разделе рассматриваются различные варианты одной задач. Пусть имеется `n` городов, пронумерованных числами от `1` до `n`. Для каждой пары городов с номерами `i`, `j` в таблице `a[i][j]` хранится целое число — цена прямого авиабилета из города `i` в город `j`. Считается, что рейсы существуют между любыми городами, `a[i][i]` = `0` при всех `i`, `a[i][j]` может отличаться от `a[j][i]`. Наименьшей стоимостью проезда из `i` в `j` считается минимально возможная сумма цен билетов для маршрутов (в том числе с пересадками), ведущих из `i` в `j`. (Она не превосходит `a[i][j]`, но может быть меньше.)

В предлагаемых ниже задачах требуется найти наименьшую стоимость проезда для некоторых пар городов при тех или иных ограничениях на массив `a` и на время работы алгоритма.

9.1.1. Предположим, что не существует замкнутых маршрутов, для которых сумма цен отрицательна. Доказать, что в этом случае маршрут с наименьшей стоимостью существует.

Решение. Маршрут длиной больше `n` всегда содержит цикл, поэтому минимум можно искать среди маршрутов длиной не более `n`, а их конечное число. □

Во всех следующих задачах предполагается, что это условие (отсутствие циклов с отрицательной суммой) выполнено.

9.1.2. Найти наименьшую стоимость проезда из 1-го города во все остальные за время $O(\mathtt{n}^3)$.

Решение. Обозначим через `МинСт(1,s,k)` наименьшую стоимость проезда из `1` в `s` менее чем с `k` пересадками. Тогда выполняется та-

кое соотношение:

$$\texttt{МинСт}(1,\texttt{s},\texttt{k+1}) = \min\Big(\texttt{МинСт}(1,\texttt{s},\texttt{k}),\ \min_{\texttt{i=1..n}} \texttt{МинСт}(1,\texttt{i},\texttt{k}) + \texttt{a[i][s]})\Big)$$

Как отмечалось выше, искомым ответом является `МинСт(1,i,n)` для всех $\texttt{i} = 1 \ldots \texttt{n}$.

```
k:= 1;
for i := 1 to n do begin x[i] := a[1][i]; end;
{инвариант: x[i] = МинСт(1,i,k)}
while k <> n do begin
| for s := 1 to n do begin
| | y[s] := x[s];
| | for i := 1 to n do begin
| | | if y[s] > x[i]+a[i][s] then begin
| | | | y[s] := x[i]+a[i][s];
| | | end;
| | end
| | {y[s] = МинСт(1,s,k+1)}
| end;
| for i := 1 to n do begin x[s] := y[s]; end;
| k := k + 1;
end;
```
□

Приведённый алгоритм называют алгоритмом динамического программирования, или алгоритмом Форда–Беллмана.

9.1.3. Доказать, что программа останется правильной, если не заводить массива `y`, а производить изменения в самом массиве `x` (заменив в программе все вхождения буквы `y` на `x` и затем удалить ставшие лишними строки).

Решение. Инвариант будет таков:

$$\texttt{МинСт}(1,\texttt{i},\texttt{n}) \leqslant \texttt{x[i]} \leqslant \texttt{МинСт}(1,\texttt{i},\texttt{k}).$$
□

Этот алгоритм может быть улучшен в двух отношениях: можно за то же время $O(\texttt{n}^3)$ найти наименьшую стоимость проезда $\texttt{i} \to \texttt{j}$ для *всех* пар `i`, `j` (а не только при $\texttt{i} = 1$), а можно сократить время работы до $O(\texttt{n}^2)$. Правда, в последнем случае нам потребуется, чтобы все цены `a[i][j]` были неотрицательны.

9.1.4. Найти наименьшую стоимость проезда $\texttt{i} \to \texttt{j}$ для всех `i`, `j` за время $O(\texttt{n}^3)$.

Решение. Для $\mathtt{k} = 0 \ldots \mathtt{n}$ через `A(i,j,k)` обозначим наименьшую стоимость маршрута из `i` в `j`, если в качестве пересадочных разрешено использовать только пункты с номерами не больше `k`. Тогда

$$\mathtt{A(i,j,0)} = \mathtt{a[i][j]},$$
$$\mathtt{A(i,j,k+1)} = \min\bigl(\mathtt{A(i,j,k)}, \mathtt{A(i,k+1,k)} + \mathtt{A(k+1,j,k)}\bigr)$$

(два варианта соответствуют неиспользованию и использованию пункта `k+1` в качестве пересадочного; отметим, что в нём незачем бывать более одного раза). □

Этот алгоритм называют алгоритмом Флойда.

9.1.5. Как проверить за $O(\mathtt{n}^3)$ действий, имеет ли граф с `n` вершинами циклы с отрицательной суммой?

[Указание. Можно применять алгоритм Флойда, причём разрешать $\mathtt{i} = \mathtt{j}$ в `A(i,j,k)`, пока не появится первый отрицательный цикл.] □

9.1.6. Имеется n валют и таблица обменных курсов (сколько флоринов дают за талер и т.п.). Коммерсант хочет неограниченно обогатиться, обменивая свой начальный капитал туда-сюда по этим курсам. Как проверить, возможно ли это?

[Указание. После логарифмирования деньги уподобляются расстояниям.] □

9.1.7. Известно, что все цены неотрицательны. Найти наименьшую стоимость проезда $1 \to \mathtt{i}$ для всех $\mathtt{i} = 1 \ldots \mathtt{n}$ за время $O(\mathtt{n}^2)$.

Решение. В процессе работы алгоритма некоторые города будут *выделенными* (в начале — только город `1`, в конце — все). При этом:

- для каждого выделенного города `i` хранится наименьшая стоимость пути $1 \to \mathtt{i}$; при этом известно, что минимум достигается на пути, проходящем только через выделенные города;
- для каждого невыделенного города `i` хранится наименьшая стоимость пути $1 \to \mathtt{i}$, в котором в качестве промежуточных используются только выделенные города.

Множество выделенных городов расширяется на основании следующего замечания: если среди всех невыделенных городов взять тот, для которого хранимое число минимально, то это число является истинной наименьшей стоимостью. В самом деле, пусть есть более короткий

путь. Рассмотрим первый невыделенный город на этом пути — уже до него путь длиннее! (Здесь существенна неотрицательность цен.)

Добавив выбранный город к выделенным, мы должны скорректировать информацию, хранимую для невыделенных городов. При этом достаточно учесть лишь пути, в которых новый город является последним пунктом пересадки, а это легко сделать, так как минимальную стоимость проезда в новый город мы уже знаем.

При самом бесхитростном способе хранения множества выделенных городов (в булевском векторе) добавление одного города к числу выделенных требует времени $O(\mathtt{n})$. □

Этот алгоритм называют алгоритмом Дейкстры.

9.1.8. Имеется n городов, соединённых дорогами (с односторонним движением). Для любых городов i, j известен максимальный вес груза, который можно везти из i в j (грузоподъёмность дороги). Найти за время $O(n^2)$ для всех городов максимальный вес груза, который в них можно привезти из столицы.

[Указание. Действовать аналогично алгоритму Дейкстры, заменив сумму на максимум.] □

Отыскание кратчайшего пути имеет естественную интерпретацию в терминах матриц. Пусть A — матрица цен одной авиакомпании, а B — матрица цен другой. Пусть мы хотим лететь с одной пересадкой, причём сначала самолётом компании A, а затем — компании B. Сколько нам придётся заплатить, чтобы попасть из города `i` в город `j`?

9.1.9. Доказать, что эта матрица вычисляется по обычной формуле для произведения матриц, только вместо суммы надо брать минимум, а вместо умножения — сумму. □

9.1.10. Доказать, что таким образом определённое произведение матриц ассоциативно. □

9.1.11. Доказать, что задача о кратчайших путях эквивалентна вычислению A^∞ для матрицы цен A: в последовательности $A, A^2, A^3, \ldots$ все элементы, начиная с некоторого, равны искомой матрице стоимостей кратчайших путей. (Если нет отрицательных циклов!) □

9.1.12. Начиная с какого элемента можно гарантировать равенство в предыдущей задаче? □

Обычное (не модифицированное) умножение матриц тоже может оказаться полезным, только матрицы должны быть другие. Пусть есть

не все рейсы (как раньше), а только некоторые, `a[i][j]` равно `1`, если рейс есть, и `0`, если рейса нет. Возведём матрицу `a` (обычным образом) в степень `k` и посмотрим на её (`i-j`)-ый элемент.

9.1.13. Чему он равен?

Ответ. Числу различных способов попасть из `i` в `j` за `k` рейсов (с `k-1` пересадками). □

При описании кратчайших путей случай, когда есть не все рейсы, можно свести к исходному, введя фиктивные рейсы с бесконечно большой (или достаточно большой) стоимостью. Тем не менее возникает такой вопрос. Число реальных рейсов может быть существенно меньше n^2, поэтому интересны алгоритмы, которые работают эффективно в такой ситуации. Исходные данные естественно представлять тогда в такой форме: для каждого города известно число выходящих из него рейсов, их пункты назначения и цены.

9.1.14. Доказать, что алгоритм Дейкстры можно модифицировать так, чтобы для n городов и m рейсов (всего) он требовал не более $C(n+m)\log n$ операций.

[Указание. Что надо сделать на каждом шаге? Выбрать невыделенный город с минимальной стоимостью и скорректировать цены для всех городов, в которые из него есть маршруты. Если бы кто-то сообщал нам, для какого города стоимость минимальна, то хватило бы $C(n+m)$ действий. А поддержание сведений о том, какой элемент в массиве минимален (см. задачу на с. 114) обходится ещё в множитель $\log n$.] □

9.2. Связные компоненты, поиск в глубину и ширину

Наиболее простой случай задачи о кратчайших путях — если все цены равны 0 или $+\infty$. Другими словами, мы интересуемся возможностью попасть из i в j, но за ценой не постоим. В других терминах: мы имеем ориентированный граф (картинку из точек, некоторые из которых соединены стрелками) и нас интересуют вершины, доступные из данной.

Для этого случая задачи о кратчайших путях приведённые в предыдущем разделе алгоритмы — не наилучшие. В самом деле, более быстрая рекурсивная программа решения этой задачи приведена в главе 7, а нерекурсивная — в главе 6. Сейчас нас интересует такая задача: не

просто перечислить все вершины, доступные из данной, но перечислить их в определённом порядке. Два популярных случая — поиск в ширину и в глубину.

Поиск в ширину.
Надо перечислить все вершины ориентированного графа, доступные из данной, в порядке увеличения длины пути от неё. (Тем самым мы решим задачу о кратчайших путях, когда цены рёбер равны 1 или $+\infty$.)

9.2.1. Придумать алгоритм решения этой задачи с числом действий не более $C \cdot$ (число рёбер, выходящих из интересующих нас вершин).

Решение. Эта задача рассматривалась в главе 6, с. 113. Здесь мы приведём подробное решение. Пусть `num[i]` — количество рёбер, выходящих из `i`, `out[i][1]`, ..., `out[i][num[i]]` — вершины, куда ведут рёбра. Вот программа, приведённая ранее:

```
procedure Доступные (i: integer);
|   {напечатать все вершины, доступные из i, включая i}
  var  X: подмножество 1..n;
       P: подмножество 1..n;
       q, v, w: 1..n;
       k: integer;
begin
  ...сделать X, P пустыми;
  writeln (i);
  ...добавить i к X, P;
  {(1) P = множество напечатанных вершин; P содержит i;
   (2) напечатаны только доступные из i вершины;
   (3) X - подмножество P;
   (4) все напечатанные вершины, из которых выходит
       ребро в ненапечатанную вершину, принадлежат X}
  while X непусто do begin
    ...взять какой-нибудь элемент X в v;
    for k := 1 to num [v] do begin
      w := out [v][k];
      if w не принадлежит P then begin
        writeln (w);
        добавить w в P;
        добавить w в X;
      end;
    end;
  end;
end;
```

Тогда нам было безразлично, какой именно элемент множества `X` выбирается. Если мы будем считать `X` очередью (первым пришёл — первым ушёл), то эта программа напечатает все вершины, доступные из `i`, в порядке возрастания их расстояния от `i` (числа рёбер на кратчайшем пути из `i`). Докажем это.

Обозначим через $V(k)$ множество всех вершин, расстояние которых от `i` (в описанном смысле) равно k. Имеет место такое соотношение:

$$V(k+1) = (\text{концы рёбер с началами в } V(k)) \setminus (V(0) \cup \ldots \cup V(k))$$

Докажем, что для любого $k = 0, 1, 2 \ldots$ в ходе работы программы будет такой момент (после очередной итерации цикла `while`), когда

> в очереди стоят все элементы $V(k)$ и только они;
> напечатаны все элементы $V(0), \ldots, V(k)$.

(Для $k = 0$ — это состояние перед циклом.) Рассуждая по индукции, предположим, что в очереди скопились все элементы $V(k)$. Они будут просматриваться в цикле, пока не кончатся (поскольку новые элементы добавляются в конец, они не перемешаются со старыми). Концы ведущих из них рёбер, если они уже не напечатаны, печатаются и ставятся в очередь — то есть всё как в записанном выше соотношении для $V(k+1)$. Так что когда все старые элементы кончатся, в очереди будут стоять все элементы $V(k+1)$. □

Поиск в глубину.

Рассматривая поиск в глубину, удобно представлять себе ориентированный граф как образ дерева. Более точно, пусть есть ориентированный граф, одна из вершин которого выделена. Будем предполагать, что все вершины доступны из выделенной по ориентированным путям. Построим дерево, которое можно было бы назвать «универсальным накрытием» нашего графа. Его корнем будет выделенная вершина графа. Из корня выходят те же стрелки, что и в графе — их концы будут сыновьями корня. Из них в дереве выходят те же стрелки, что и в графе и так далее. Разница между графом и деревом в том, что пути в графе, ведущие в одну и ту же вершину, в дереве «расклеены». В других терминах: вершина дерева — это путь в графе, выходящий из корня. Её сыновья — это пути, продолженные на одно ребро. Заметим, что дерево бесконечно, если в графе есть ориентированные циклы.

Имеется естественное отображение дерева в граф (вершин в вершины). При этом каждая вершина графа имеет столько прообразов, сколько путей в неё ведёт. Поэтому обход дерева (посещение его вершин

в том или ином порядке) одновременно является и обходом графа — только каждая вершина посещается многократно.

Будем предполагать, что для каждой вершины графа выходящие из неё рёбра упорядочены (например, пронумерованы). Тем самым для каждой вершины дерева выходящие из неё рёбра также упорядочены. Будем обходить дерево так: сначала корень, а потом поддеревья (в порядке ведущих в них рёбер). Такой обход дерева рассматривался нами в главе 7. Ему соответствует обход графа. Если выкинуть из этого обхода повторные посещения уже посещённых вершин, то получится то, что называется «поиск в глубину».

Другими словами, на путях, выходящих из выделенной вершины, введём порядок: путь предшествует своему продолжению; если два пути расходятся в некоторой вершине, то меньшим считается тот, который выходит из неё по меньшему ребру. Вершины теперь упорядочиваются в соответствии с минимальными путями, в них ведущими. Обход вершин графа в указанном порядке называется *поиском в глубину*.

9.2.2. Написать программу поиска в глубину.

[Указание. Возьмём программу обхода дерева (корень → левое поддерево → правое поддерево) из главы 7 или из главы 8 и используем её применительно к обстоятельствам. Главное изменение: не надо посещать вершины повторно. Так что если мы попали в уже посещённую вершину, то можно с ней ничего не делать. (Если путь не минимален среди ведущих в данную вершину, то и все его продолжения не минимальны — их просматривать не надо).] □

Замечание. Напомним, что в главе 8 упоминались две возможности устранения рекурсии в программе обхода дерева (с. 141). Оба варианта можно использовать для поиска в глубину.

Поиск в глубину лежит в основе многих алгоритмов на графах, порой в несколько модифицированном виде.

9.2.3. Неориентированный граф называется *двудольным*, если его вершины можно раскрасить в два цвета так, что концы любого ребра — разного цвета. Составить алгоритм проверки, является ли заданный граф двудольным, в котором число действий не превосходит $C \cdot$ (число рёбер + число вершин).

[Указание. (а) Каждую связную компоненту можно раскрашивать отдельно. (б) Выбрав цвет одной вершины и обходя её связную компоненту, мы определяем единственно возможный цвет остальных.] □

Замечание. В этой задаче безразлично, производить поиск в ширину или в глубину.

9.2.4. Составить нерекурсивный алгоритм топологической сортировки ориентированного графа без циклов. (Рекурсивный алгоритм смотри на с. 127.)

Решение. Предположим, что граф имеет вершины с номерами `1`...`n`, для каждой вершины `i` известно число `num[i]` выходящих из неё рёбер и номера вершин `dest[i][1]`, ..., `dest[i][num[i]]`, в которые эти рёбра ведут. Будем условно считать, что рёбра перечислены «слева направо»: левее то ребро, у которого номер меньше. Нам надо напечатать все вершины в таком порядке, чтобы конец любого ребра был напечатан перед его началом. Мы предполагаем, что в графе нет ориентированных циклов — иначе такое невозможно.

Для начала добавим к графу вершину `0`, из которой рёбра ведут в вершины `1`, ..., `n`. Если её удастся напечатать с соблюдением правил, то тем самым все вершины будут напечатаны.

Алгоритм хранит путь, выходящий из нулевой вершины и идущий по рёбрам графа. Переменная `l` отводится для длины этого пути. Путь образован вершинами `vert[1]`...`vert[l]` и рёбрами, имеющими номера `edge[1]`...`edge[l]`. Номер `edge[s]` относится к нумерации рёбер, выходящих из вершины `vert[s]`. Тем самым для всех `s` должны выполняться неравенство

$$\texttt{edge[s]} \leqslant \texttt{num[vert[s]]}$$

и равенство

$$\texttt{vert[s+1]} = \texttt{dest[vert[s]][edge[s]]}.$$

Заметим, что конец последнего ребра нашего пути (то есть вершина `dest[vert[l]][edge[l]]`, не включается в массив `vert`. Кроме того, для последнего ребра мы делаем исключение, разрешая ему указывать «в пустоту», т. е. разрешаем `edge[l]` равняться `num[vert[l]]+1`.

В процессе работы алгоритм будет печатать номера вершин, при этом соблюдая требование «вершина напечатана только после тех вершин, в которые из неё ведут рёбра». Кроме того, будет выполняться такое требование (И):

> вершины пути, кроме последней (`vert[1]`...`vert[l]`) не напечатаны, но свернув с пути налево, мы немедленно упираемся в напечатанную вершину.

Вот что получается:

```
l:=1; vert[1]:=0; edge[1]:=1;
while not( (l=1) and (edge[1]=n+1)) do begin
| if edge[l]=num[vert[l]]+1 then begin
| | {путь кончается в пустоте, поэтому все вершины,
| |     следующие за vert[l], напечатаны - можно
| |     печатать vert[l]}
| | writeln (vert[l]);
| | l:=l-1; edge[l]:=edge[l]+1;
| end else begin
| |  {edge[l] <= num[vert[l]], путь кончается в
| |     вершине}
| |  lastvert:= dest[vert[l]][edge[l]]; {последняя}
| |  if lastvert напечатана then begin
| |  | edge[l]:=edge[l]+1;
| |  end else begin
| |  | l:=l+1; vert[l]:=lastvert; edge[l]:=1;
| |  end;
| end;
end;
{путь сразу же ведёт в пустоту, поэтому все вершины
 левее, то есть 1..n, напечатаны}
```

9.2.5. Доказать, что если в графе нет циклов, то этот алгоритм заканчивает работу.

Решение. Пусть это не так. Каждая вершина может печататься только один раз, так что с некоторого момента вершины не печатаются. В графе без циклов длина пути ограничена (вершина не может входить в путь дважды), поэтому подождав ещё, мы можем дождаться момента, после которого путь не удлиняется. После этого может разве что увеличиваться `edge[l]` — но и это не беспредельно. □

Тем самым мы построили искомый нерекурсивный алгоритм топологической сортировки графа. □

9.2.6. Доказать, что время работы этого алгоритма не превосходит O(число вершин + число рёбер). □

9.2.7. Как модифицировать алгоритм так, чтобы он отыскивал один из циклов, если таковые имеются, и производил топологическую сортировку, если циклов нет? □

10. СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

10.1. Простейший пример

10.1.1. Имеется последовательность символов `x[1]` ... `x[n]`. Определить, имеются ли в ней идущие друг за другом символы `abcd`. (Другими словами, требуется выяснить, есть ли в слове `x[1]` ... `x[n]` подслово `abcd`.)

Решение. Имеется примерно `n` (если быть точным, `n-3`) позиций, на которых может находиться искомое подслово в исходном слове. Для каждой из позиций можно проверить, действительно ли там оно находится, сравнив четыре символа. Однако есть более эффективный способ. Читая слово `x[1]` ... `x[n]` слева направо, мы ожидаем появления буквы `a`. Как только она появилась, мы ищем за ней букву `b`, затем `c`, и, наконец, `d`. Если наши ожидания оправдываются, то слово `abcd` обнаружено. Если же какая-то из нужных букв не появляется, мы оказываемся у разбитого корыта и начинаем всё сначала. □

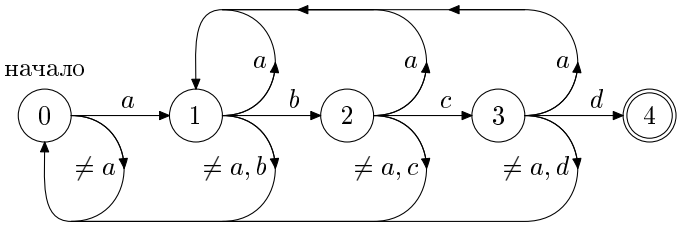
Этот простой алгоритм можно описать в разных терминах. Используя терминологию так называемых *конечных автоматов*, можно сказать, что при чтении слова `x` слева направо мы в каждый момент находимся в одном из следующих состояний: «начальное» (0), «сразу после `a`» (1), «сразу после `ab`» (2), «сразу после `abc`» (3) и «сразу после `abcd`» (4). Читая очередную букву, мы переходим в следующее состояние по правилу, указанному в таблице (см. следующую страницу). Как только мы попадём в состояние 4, работа заканчивается.

Наглядно выполнение алгоритма можно представить себе так: фишка двигается из кружка в кружок по стрелкам; стрелка выбирается так, чтобы надпись на ней соответствовала очередной букве входного слова. Чтобы этот процесс был успешным, нужно, чтобы для каждой буквы

Текущее состояние	Очередная буква	Новое состояние
0	a	1
0	кроме a	0
1	b	2
1	a	1
1	кроме a,b	0
2	c	3
2	a	1
2	кроме a,c	0
3	d	4
3	a	1
3	кроме a,d	0

Правила перехода для конечного автомата

была ровно одна подходящая стрелка из любого кружка.



Соответствующая программа очевидна (мы указываем новое состояние, даже если оно совпадает со старым; эти строки можно опустить):

```
i:=1; state:=0;
{i - первая непрочитанная буква, state - состояние}
while (i <> n+1) and (state <> 4) do begin
  if state = 0 then begin
    if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end else if state = 1 then begin
    if x[i] = b then begin
      state:= 2;
    end else if x[i] = a then begin
      state:= 1;
```

```
    | | end else begin
    | |   state:= 0;
    | | end;
    | end else if state = 2 then begin
    | |   if x[i] = c then begin
    | |     state:= 3;
    | |   end else if x[i] = a then begin
    | |     state:= 1;
    | |   end else begin
    | |     state:= 0;
    | |   end;
    | end else if state = 3 then begin
    | |   if x[i] = d then begin
    | |     state:= 4;
    | |   end else if x[i] = a then begin
    | |     state:= 1;
    | |   end else begin
    | |     state:= 0;
    | |   end;
    | end;
end;
answer := (state = 4);
```

Иными словами, мы в каждый момент храним информацию о том, какое максимальное начало нашего образца `abcd` является концом прочитанной части. (Его длина и есть то «состояние», о котором шла речь.)

Терминология, нами используемая, такова. *Слово* — это любая последовательность символов из некоторого фиксированного конечного множества. Это множество называется *алфавитом*, его элементы — *буквами*. Если отбросить несколько букв с конца слова, останется другое слово, называемое *началом* первого. Любое слово также считается своим началом. *Конец* слова — то, что останется, если отбросить несколько первых букв. Любое слово считается своим концом. *Подслово* — то, что останется, если отбросить буквы и с начала, и с конца. (Другими словами, подслова — это концы начал, или, что то же, начала концов.)

В терминах индуктивных функций (см. раздел 1.3) ситуацию можно описать так: рассмотрим функцию на словах, которая принимает два значения «истина» и «ложь» и истинна на словах, имеющих `abcd` своим подсловом. Эта функция не является индуктивной, но имеет индуктивное расширение

$x \mapsto$ длина максимального начала слова `abcd`, являющегося концом x.

10.2. Повторения в образце — источник проблем

10.2.1. Можно ли в предыдущих рассуждениях заменить слово `abcd` на произвольное слово?

Решение. Нет, и проблемы связаны с тем, что в образце могут быть повторяющиеся буквы. Пусть, например, мы ищем вхождения слова `ababc`. Вот появилась буква `a`, за ней идёт `b`, за ней идёт `a`, затем снова `b`. В этот момент мы с нетерпением ждём буквы `c`. Однако — к нашему разочарованию — вместо неё появляется другая буква, и наш образец `ababc` не обнаружен. Однако нас может ожидать утешительный приз: если вместо `c` появилась буква `a`, то не всё потеряно: за ней могут последовать буквы `b` и `c`, и образец-таки будет найден. □

Вот картинка, поясняющая сказанное:

```
x y z a b a b a b c ...  ←  входное слово
      a b a b c         ←  мы ждали образца здесь
        a b a b c       ←  а он оказался здесь
```

Таким образом, к моменту

```
x y z a b a b |         ←  входное слово
      a b a b | c       ←  мы ждали образца здесь
        a b   | a b c   ←  а он оказался здесь
```

есть два возможных положения образца, каждое из которых подлежит проверке. Тем не менее по-прежнему возможен конечный автомат, читающий входное слово буква за буквой и переходящий из состояния в состояние в зависимости от прочитанных букв.

10.2.2. Указать состояния соответствующего автомата и таблицу перехода (новое состояние в зависимости от старого и читаемой буквы).

Решение. По-прежнему состояния будут соответствовать наибольшему началу образца, являющемуся концом прочитанной части слова. Их будет шесть: 0, 1 (`a`), 2 (`ab`), 3 (`aba`), 4 (`abab`), 5 (`ababc`). Таблица перехода такая.

Текущее состояние	Очередная буква	Новое состояние
0	`a`	1 (`a`)
0	кроме `a`	0
1 (`a`)	`b`	2 (`ab`)
1 (`a`)	`a`	1 (`a`)
1 (`a`)	кроме `a`,`b`	0
2 (`ab`)	`a`	3 (`aba`)
2 (`ab`)	кроме `a`	0
3 (`aba`)	`b`	4 (`abab`)
3 (`aba`)	`a`	1 (`a`)
3 (`aba`)	кроме `a`,`b`	0
4 (`abab`)	`c`	5 (`ababc`)
4 (`abab`)	`a`	3 (`aba`)
4 (`abab`)	кроме `a`,`c`	0

Для проверки посмотрим, к примеру, на вторую снизу строку. Если прочитанная часть кончалась на `abab`, а затем появилась буква `a`, то теперь прочитанная часть кончается на `ababa`. Наибольшее начало образца (`ababc`), являющееся её концом — это `aba`. □

Философский вопрос: мы говорили, что трудность состоит в том, что есть несколько возможных положений образца, каждое из которых может оказаться истинным. Им соответствуют несколько начал образца, являющихся концами входного слова. Но конечный автомат помнит лишь самое длинное из них. Как же остальные?

Философский ответ. Дело в том, что самое длинное из них определяет все остальные — это его концы, одновременно являющиеся его началами.

Не составляет труда для любого конкретного образца написать программу, осуществляющую поиск этого образца описанным способом. Однако хотелось бы написать программу, которая ищет произвольный образец в произвольном слове. Это можно делать в два этапа: сначала по образцу строится таблица переходов конечного автомата, а затем читается входное слово и состояние преобразуется в соответствии с этой таблицей. Подобный метод часто используется для более сложных задач поиска (см. далее), но для поиска подслова существует более простой и эффективный алгоритм, называемый алгоритмом Кнута–Морриса–Пратта. (Ранее сходные идеи были предложены Ю. В. Матиясевичем.) Но прежде нам понадобятся некоторые вспомогательные утверждения.

10.3. Вспомогательные утверждения

Для произвольного слова X рассмотрим все его начала, одновременно являющиеся его концами, и выберем из них самое длинное. (Не считая, конечно, самого слова X.) Будем обозначать его $l(X)$.

Примеры: $l(\texttt{aba}) = \texttt{a}$, $l(\texttt{abab}) = \texttt{ab}$, $l(\texttt{ababa}) = \texttt{aba}$, $l(\texttt{abc}) =$ пустое слово.

10.3.1. Доказать, что все слова $l(X)$, $l(l(X))$, $l(l(l(X)))$ и т. д. являются началами слова X.

Решение. Каждое из них (согласно определению) является началом предыдущего. □

По той же причине все они являются концами слова X.

10.3.2. Доказать, что последовательность предыдущей задачи обрывается (на пустом слове).

Решение. Каждое слово короче предыдущего. □

10.3.3. Доказать, что любое слово, одновременно являющееся началом и концом слова X (кроме самого X) входит в последовательность $l(X), l(l(X)), \ldots$

Решение. Пусть слово Y есть одновременно начало и конец X. Слово $l(X)$ — самое длинное из таких слов, так что Y не длиннее $l(X)$. Оба эти слова являются началами X, поэтому более короткое из них является началом более длинного: Y есть начало $l(X)$. Аналогично, Y есть конец $l(X)$. Рассуждая по индукции, можно предполагать, что утверждение задачи верно для всех слов короче X, в частности, для слова $l(X)$. Так что слово Y, являющееся концом и началом $l(X)$, либо равно $l(X)$, либо входит в последовательность $l(l(X)), l(l(l(X))), \ldots$, что и требовалось доказать. □

10.4. Алгоритм Кнута – Морриса – Пратта

Алгоритм Кнута – Морриса – Пратта (КМП) получает на вход слово

$$\texttt{X} = \texttt{x[1]x[2]}\ldots\texttt{x[n]}$$

и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел $\texttt{l[1]}\ldots\texttt{l[n]}$, где

$$\texttt{l[i]} = \text{длина слова } l(\texttt{x[1]}\ldots\texttt{x[i]})$$

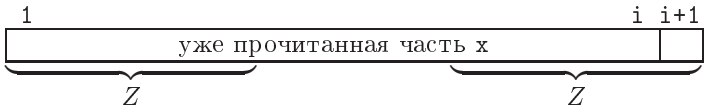
(функция l определена в предыдущем пункте). Словами: `l[i]` есть длина наибольшего начала слова `x[1]`...`x[i]`, одновременно являющегося его концом.

10.4.1. Какое отношение всё это имеет к поиску подслова? Другими словами, как использовать алгоритм КМП для определения того, является ли слово `A` подсловом слова `B`?

Решение. Применим алгоритм КМП к слову `A#B`, где `#` — специальная буква, не встречающаяся ни в `A`, ни в `B`. Слово `A` является подсловом слова `B` тогда и только тогда, когда среди чисел в массиве `l` будет число, равное длине слова `A`. □

10.4.2. Описать алгоритм заполнения таблицы `l[1]`...`l[n]`.

Решение. Предположим, что первые `i` значений `l[1]`...`l[i]` уже найдены. Мы читаем очередную букву слова (т. е. `x[i+1]`) и должны вычислить `l[i+1]`.



Другими словами, нас интересуют начала Z слова `x[1]`...`x[i+1]`, одновременно являющиеся его концами — из них нам надо выбрать самое длинное. Откуда берутся эти начала? Каждое из них (не считая пустого) получается из некоторого слова Z' приписыванием буквы `x[i+1]`. Слово Z' является началом и концом слова `x[1]`...`x[i]`. Однако не любое слово, являющееся началом и концом слова `x[1]`...`x[i]`, годится — надо, чтобы за ним следовала буква `x[i+1]`.

Получаем такой рецепт отыскания слова Z. Рассмотрим все начала слова `x[1]`...`x[i]`, являющиеся одновременно его концами. Из них выберем подходящие — те, за которыми идёт буква `x[i+1]`. Из подходящих выберем самое длинное. Приписав в его конец `x[i+1]`, получим искомое слово Z.

Теперь пора воспользоваться сделанными нами приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l из предыдущего раздела. Вот что получается:

```
i:=1; l[1]:= 0;
{таблица l[1]..l[i] заполнена правильно}
while i <> n do begin
| len := l[i]
```

```
    {len - длина начала слова x[1]..x[i], которое является
       его концом; все более длинные начала оказались
       неподходящими}
    while (x[len+1] <> x[i+1]) and (len > 0) do begin
    | {начало не подходит, применяем к нему функцию l}
    | len := l[len];
    end;
    {нашли подходящее или убедились в отсутствии}
    if x[len+1] = x[i+1] do begin
    | {x[1]..x[len] - самое длинное подходящее начало}
    | l[i+1] := len+1;
    end else begin
    | {подходящих нет}
    | l[i+1] := 0;
    end;
    i := i+1;
end;
```

□

10.4.3. Доказать, что число действий в приведённом только что алгоритме не превосходит $C\mathtt{n}$ для некоторой константы C.

Решение. Это не вполне очевидно: обработка каждой очередной буквы может потребовать многих итераций во внутреннем цикле. Однако каждая такая итерация уменьшает `len` по крайней мере на 1, и в этом случае `l[i+1]` окажется заметно меньше `l[i]`. С другой стороны, при увеличении `i` на единицу величина `l[i]` может возрасти не более чем на 1, так что часто и сильно убывать она не может — иначе убывание не будет скомпенсировано возрастанием.

Более точно, можно записать неравенство

$$\mathtt{l[i+1]} \leqslant \mathtt{l[i]} - (\text{число итераций на } \mathtt{i}\text{-м шаге}) + 1$$

или

$$(\text{число итераций на } \mathtt{i}\text{-м шаге}) \leqslant \mathtt{l[i]} - \mathtt{l[i+1]} + 1.$$

Остаётся сложить эти неравенства по всем `i` и получить оценку сверху для общего числа итераций. □

10.4.4. Будем использовать этот алгоритм, чтобы выяснить, является ли слово `X` длины `n` подсловом слова `Y` длины `m`. (Как это делать с помощью специального разделителя `#`, описано выше.) При этом число действий будет не более $C(\mathtt{n}+\mathtt{m})$, и используемая память тоже. Придумать, как обойтись памятью не более $C\mathtt{n}$ (что может быть существенно меньше, если искомый образец короткий, а слово, в котором его ищут — длинное).

Решение. Применяем алгоритм КМП к слову `A#B`. При этом вычисление значений `l[1]`, . . . , `l[n]` проводим для слова `X` длины `n` и запоминаем эти значения. Дальше мы помним только значение `l[i]` для текущего `i` — кроме него и кроме таблицы `l[1]` . . . `l[n]`, нам для вычислений ничего не нужно. □

На практике слова `X` и `Y` могут не находиться подряд, поэтому просмотр слова `X` и затем слова `Y` удобно оформить в виде разных циклов. Это избавляет также от хлопот с разделителем.

10.4.5. Написать соответствующий алгоритм (проверяющий, является ли слово `X` = `x[1]` . . . `x[n]` подсловом слова `Y` = `y[1]` . . . `y[m]`).

Решение. Сначала вычисляем таблицу `l[1]` . . . `l[n]` как раньше. Затем пишем такую программу:

```
j:=0; len:=0;
{len - длина максимального начала слова X, одновременно
       являющегося концом слова y[1]..y[j]}
while (len <> n) and (j <> m) do begin
| while (x[len+1] <> y[j+1]) and (len > 0) do begin
| | {начало не подходит, применяем к нему функцию l}
| | len := l[len];
| end;
| {нашли подходящее или убедились в отсутствии}
| if x[len+1] = y[j+1] do begin
| | {x[1]..x[len] - самое длинное подходящее начало}
| | len := len+1;
| end else begin
| | {подходящих нет}
| | len := 0;
| end;
| j := j+1;
end;
{если len=n, слово X встретилось; иначе мы дошли до конца
   слова Y, так и не встретив X}
```

□

10.5. Алгоритм Бойера – Мура

Этот алгоритм делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный образец. Как так может быть? Идея проста. Пусть, например, мы ищем образец `abcd`. Посмотрим на четвёртую букву слова: если, к примеру, это буква `e`, то нет никакой необхо-

димости читать первые три буквы. (В самом деле, в образце буквы e нет, поэтому он может начаться не раньше пятой буквы.)

Мы приведём самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть x[1] ... x[n] — образец, который надо искать. Для каждого символа s найдём самое правое его вхождение в слово X, то есть наибольшее k, при котором x[k] = s. Эти сведения будем хранить в массиве pos[s]; если символ s вовсе не встречается, то нам будет удобно положить pos[s] = 0 (мы увидим дальше, почему).

10.5.1. Как заполнить массив pos?

Решение.

```
положить все pos[s] равными 0
for i:=1 to n do begin
  pos[x[i]]:=i;
end;
```
□

В процессе поиска мы будем хранить в переменной last номер буквы в слове, против которой стоит последняя буква образца. Вначале last = n (длина образца), затем last постепенно увеличивается.

```
last:=n;
{все предыдущие положения образца уже проверены}
while last <= m do begin {слово не кончилось}
| if x[n] <> y[last] then begin {последние буквы разные}
| | last := last + (n - pos[y[last]]);
| | {n - pos[y[last]]  - это минимальный сдвиг образца,
| |    при котором напротив y[last] встанет такая же
| |    буква в образце. Если такой буквы нет вообще,
| |    то сдвигаем на всю длину образца}
| end else begin
| | если нынешнее положение подходит, т.е. если
| | x[1]..x[n] = y[last-n+1]..y[last],
| | то сообщить о совпадении;
| | last := last+1;
| end;
end;
```

Знатоки рекомендуют проверку совпадения проводить справа налево, т. е. начиная с последней буквы образца (в которой совпадение заведомо есть). Можно также немного сэкономить, произведя вычитание заранее и храня не pos[s], а n-pos[s], т. е. число букв в образце справа от последнего вхождения буквы s.

Возможны разные модификации этого алгоритма. Например, можно строку `last:=last+1` заменить на `last:=last+(n-u)`, где `u` — координата второго справа вхождения буквы `x[n]` в образец.

10.5.2. Как проще всего учесть это в программе?

Решение. При построении таблицы `pos` написать

```
for i:=1 to n-1 do...
```

(далее как раньше), а в основной программе вместо `last:=last+1` написать

```
last:= last+n-pos[y[last]];
```
□

Приведённый нами упрощённый вариант алгоритма Бойера – Мура в некоторых случаях требует существенно больше `n` действий (число действий порядка `mn`), проигрывая алгоритму Кнута – Морриса – Пратта.

10.5.3. Привести пример ситуации, в которой образец не входит в слово, но алгоритму требуется порядка `mn` действий, чтобы это установить.

Решение. Пусть образец имеет вид `baaa`...`aa`, а само слово состоит только из букв `a`. Тогда на каждом шаге несоответствие выясняется лишь в последний момент. □

Настоящий (не упрощённый) алгоритм Бойера – Мура гарантирует, что число действий не превосходит $C(\mathtt{m}+\mathtt{n})$ в худшем случае. Он использует идеи, близкие к идеям алгоритма Кнута – Морриса – Пратта. Представим себе, что мы сравнивали образец со входным словом, идя справа налево. При этом некоторый кусок Z (являющийся концом образца) совпал, а затем обнаружилось различие: перед Z в образце стоит не то, что во входном слове. Что можно сказать в этот момент о входном слове? В нём обнаружен фрагмент, равный Z, а перед ним стоит не та буква, что в образце. Эта информация может позволить сдвинуть образец на несколько позиций вправо без риска пропустить его вхождение. Эти сдвиги следует вычислить заранее для каждого конца Z нашего образца. Как говорят знатоки, всё это (вычисление таблицы сдвигов и её использование) можно уложить в $C(\mathtt{m}+\mathtt{n})$ действий.

10.6. Алгоритм Рабина

Этот алгоритм основан на простой идее. Представим себе, что в слове длины m мы ищем образец длины n. Вырежем окошечко размера n

и будем двигать его по входному слову. Нас интересует, не совпадает ли слово в окошечке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определённую на словах длины n. Если значения этой функции на слове в окошечке и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

Что мы выигрываем при таком подходе? Казалось бы, ничего — ведь чтобы вычислить значение функции на слове в окошечке, всё равно нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, и вот за счёт чего. При сдвиге окошечка слово не меняется полностью, а лишь добавляется буква в конце и убирается в начале. Хорошо бы, чтобы по этим данным можно было рассчитать, как меняется функция.

10.6.1. Привести пример удобной для вычисления функции.

Решение. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошечка нужно добавить новое число и вычесть пропавшее.) □

Для каждой функции существуют слова, к которым она применима плохо. Зато другая функция в этом случае может работать хорошо. Возникает идея: надо запасти много функций и в начале работы алгоритма выбирать из них случайную. (Тогда враг, желающий подгадить нашему алгоритму, не будет знать, с какой именно функцией ему бороться.)

10.6.2. Привести пример семейства удобных функций.

Решение. Выберем некоторое число p (желательно простое, смотри далее) и некоторый вычет x по модулю p. Каждое слово длины n будем рассматривать как последовательность целых чисел (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени $n-1$ и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя функция). Сдвиг окошка на 1 соответствует вычитанию старшего члена (x^{n-1} следует вычислить заранее), умножению на x и добавлению свободного члена. □

Следующее соображение говорит в пользу того, что совпадения не слишком вероятны. Пусть число p фиксировано и к тому же простое, а X и Y — два различных слова длины n. Тогда им соответствуют различные многочлены (мы предполагаем, что коды всех букв различны —

это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, то есть их разность обращается в 0. Разность есть многочлен степени $n-1$ и имеет не более $n-1$ корней. Таким образом, если n много меньше p, то случайному x мало шансов попасть в неудачную точку.

10.7. Более сложные образцы и автоматы

Мы можем искать не конкретное слово, а подслова заданного вида. Например, можно искать слова вида `a?b`, где вместо `?` может стоять любая буква (иными словами, нас интересует буква `b` на расстоянии 2 после буквы `a`).

10.7.1. Указать конечный автомат, проверяющий, есть ли во входном слове фрагмент вида `a?b`.

Решение. Читая слово, следует помнить, есть ли буква `a` на последнем месте и на предпоследнем — пока не встретим искомый фрагмент. Автомат имеет состояния 00, 01, 10, 11, их смысл таков:

00 на предпоследнем и последнем местах нет `a`
01 на предпоследнем нет, на последнем есть
10 не предпоследнем есть, на последнем нет
11 есть и там, и там

Таблица переходов автомата:

Текущее состояние	Очередная буква	Новое состояние
00	`a`	01
00	не `a`	00
01	`a`	11
01	не `a`	10
10	`a`	01
10	`b`	найдено
10	не `a` и не `b`	00
11	`a`	11
11	`b`	найдено
11	не `a` и не `b`	10

□

Другой стандартный знак в образце — это звёздочка (`*`), на место которой может быть подставлено любое слово. Например, образец

`ab*cd` означает, что мы ищем подслово `ab`, за которым следует что угодно, а затем (на любом расстоянии) идёт `cd`.

10.7.2. Указать конечный автомат, проверяющий, есть ли во входном слове образец `ab*cd` (в описанном только что смысле).

Решение.

Текущее состояние	Очередная буква	Новое состояние
начальное	`a`	`a`
начальное	не `a`	начальное
`a`	`b`	`ab`
`a`	`a`	`a`
`a`	не `a` и не `b`	начальное
`ab`	`c`	`abc`
`ab`	не `c`	`ab`
`abc`	`d`	найдено
`abc`	`c`	`abc`
`abc`	не `c` и не `d`	`ab`

□

Ещё один вид поиска — это поиск любого из слов некоторого списка.

10.7.3. Дан список слов $X_1, \ldots, X_k$ и слово Y. Определить, входит ли хотя бы одно из слов X_i в слово Y (как подслово). Количество действий не должно превосходить константы, умноженной на суммарную длину всех слов (из списка и того, в котором происходит поиск).

Решение. Очевидный способ состоит в том, чтобы каждое слово из списка проверять отдельно (с помощью одного из рассмотренных алгоритмов). Однако при этом мы не укладываемся в заданное число действий (из-за умножения k на длину слова Y).

Посмотрим на дело с другой стороны. Каждому образцу из списка соответствует конечный автомат с некоторым множеством состояний. Эти автоматы можно объединить в один, множеством состояний которого будет произведение множеств состояний всех тех автоматов. Это — очень большое множество. Однако на самом деле большинство его элементов недоступны (не могут появиться при чтении входного слова) и за счёт этого получается экономия. Примерно эту идею (но в изменённом виде) мы и будем использовать.

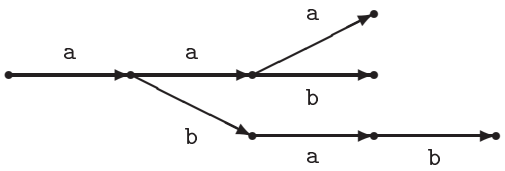
Вспомним алгоритм Кнута–Морриса–Пратта. В нём, читая входное слово, мы хранили наибольшее начало образца, являющееся концом прочитанной части. Теперь нам следует хранить для каждого из

образцов наибольшее его начало, являющееся концом прочитанной части. Решающим оказывается такое замечание: достаточно хранить самое длинное из них — все остальные по нему восстанавливаются (как наибольшие начала образцов, являющиеся его концами).

Склеим все образцы в дерево, объединив их совпадающие начальные участки. Например, набору образцов

$$\{\texttt{aaa}, \texttt{aab}, \texttt{abab}\}$$

соответствует дерево



Формально говоря, вершинами дерева являются все начала всех образцов, а сыновья вершины получаются приписыванием буквы.

Читая входное слово, мы двигаемся по этому дереву: текущая вершина — это наибольшая (самая правая) из вершин, являющихся концом прочитанной части (= наибольший конец прочитанной части, являющийся началом одного из образцов).

Определим функцию l, аргументами и значениями которой являются вершины дерева. Именно, $l(P)$ = наибольшая вершина дерева, являющаяся концом P. (Напомним, вершины дерева — это слова.) Нам понадобится такое утверждение:

10.7.4. Пусть P — вершина дерева. Доказать, что множество всех вершин, являющихся концами P, равно $\{l(P), l(l(P)), \ldots\}$

Решение. См. доказательство аналогичного утверждения для алгоритма Кнута–Морриса–Пратта. □

Теперь ясно, что нужно делать, находясь в вершине P и читая букву z входного слова. Надо просматривать последовательно вершины $P, l(P), l(l(P)), \ldots$, пока не обнаружится такая, из которой выходит стрелка с буквой z. Та вершина, в которую эта стрелка ведёт, и будет нашим следующим положением.

Остаётся понять, как для каждой вершины дерева вычислить указатель на значение функции l в этой вершине. Это делается как раньше, при этом значения l для более коротких слов используются при вычислении очередного значения функции l. Это означает, что вершины дерева следует просматривать в порядке возрастания их длины. Нетрудно понять, что всё это можно уложить в требуемое число действий (хотя

константа зависит от числа букв в алфавите). Относящиеся к этому подробности см. в главе 9. □

Можно поинтересоваться, какие свойства слов распознаются с помощью конечных автоматов. Оказывается, что существует просто описываемый класс образцов, задающий все такие свойства — класс *регулярных выражений*.

Определение. Пусть фиксирован конечный алфавит Γ, не содержащий символов Λ, ε, (,), $*$ и | (они будут использоваться для построения регулярных выражений и не должны перемешиваться с буквами). Регулярные выражения строятся по таким правилам:

(а) буква алфавита Γ — регулярное выражение;

(б) символы Λ, ε — регулярные выражения;

(в) если $A, B, C, \ldots, E$ — регулярные выражения, то $(ABC\ldots E)$ — регулярное выражение;

(г) если $A, B, C, \ldots, E$ — регулярные выражения, то $(A|B|C|\ldots|E)$ — регулярное выражение;

(д) если A — регулярное выражение, то $A*$ — регулярное выражение.

Каждое регулярное выражение задаёт множество слов в алфавите Γ по таким правилам:

(а) букве соответствует одноэлементное множество, состоящее из однобуквенного слова, состоящего из этой буквы;

(б) символу ε соответствует пустое множество, а символу Λ — одноэлементное множество, единственным элементом которого является пустое слово;

(в) регулярному выражению $(ABC\ldots E)$ соответствует множество всех слов, которые можно получить, если к слову из A приписать слово из B, затем из C,..., затем из E (*конкатенация* множеств);

(г) регулярному выражению $(A|B|C|\ldots|E)$ соответствует объединение множеств, соответствующих выражениям $A, B, C, \ldots, E$;

(д) регулярному выражению $A*$ соответствует *итерация* множества, соответствующего выражению A, то есть множество всех слов, которые можно так разрезать на куски, что каждый кусок принадлежит множеству, соответствующему выражению A. (В частности, пустое слово всегда содержится в $A*$.)

Множества, соответствующие регулярным выражениям, называются *регулярными*. Вот несколько примеров:

Выражение	Множество
`(a|b)*`	все слова из букв `a` и `b`
`(aa)*`	слова из чётного числа букв `a`
`(Λ|a|b|aa|ab|ba|bb)`	все слова длины не более 2 из букв `a`, `b`

10.7.5. Написать регулярное выражение, которому соответствует множество всех слов из букв `a` и `b`, в которых число букв `a` чётно.

Решение. Выражение `b*` задаёт все слова без буквы `a`, а выражение `(b*ab*ab*)` — все слова ровно с двумя буквами `a`. Остаётся объединить эти множества, а потом применить итерацию:

```
( (b*ab*ab*) | b* )*
```

Другой вариант ответа:

```
(b*ab*a)*b*
```

□

10.7.6. Написать регулярное выражение, которое задаёт множество всех слов из букв `a`, `b`, `c`, в которых слово `bac` является подсловом.

Решение. `((a|b|c)*bac(a|b|c)*)` □

10.7.7. Написать регулярное выражение, которое задаёт множество всех слов из букв `a`, `b`, `c`, в которых слово `bac` *не* является подсловом.

[Указание. Эта задача сложнее предыдущей; видимо, самый простой способ её решить — перейти к конечным автоматам и вернуться обратно (см. ниже задачу 10.7.14).] □

Теперь задачу о поиске образца в слове можно переформулировать так: проверить, принадлежит ли слово множеству, соответствующему данному регулярному выражению.

10.7.8. Какие выражения соответствуют образцам `a?b` и `ab*cd`, рассмотренным ранее? (В образце символ `*` используется не в том смысле, что в регулярных выражениях!) Предполагается, что алфавит содержит буквы `a`, `b`, `c`, `d`, `e`.

Решение.

```
((a|b|c|d|e)*a(a|b|c|d|e)b(a|b|c|d|e)*)
((a|b|c|d|e)*ab(a|b|c|d|e)*cd(a|b|c|d|e)*)
```

□

10.7.9. Доказать, что для всякого регулярного выражения можно построить конечный автомат, который распознаёт соответствующее этому выражению множество слов.

Решение. Нам потребуется новое понятие — понятие *источника*, или *недетерминированного конечного автомата*. Представим себе ориентированный граф — картинку из нескольких точек (вершин) и некоторых стрелок, соединяющих эти точки (рёбер). Пусть на некоторых рёбрах написаны буквы (не обязательно на всех). Пусть также среди вершин выбраны две — начальная Н и конечная К. Такая картинка называется *источником*.

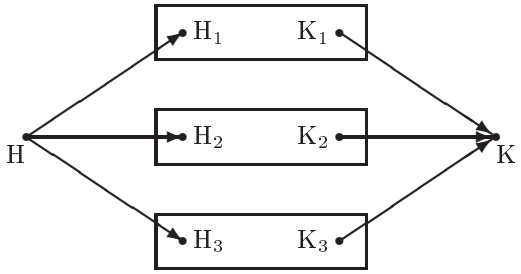
Будем двигаться различными способами из Н в К, читая буквы по дороге (на тех стрелках, где они есть). Каждому пути из Н в К, таким образом, соответствует некоторое слово. А источнику в целом соответствует множество слов — тех слов, которые можно прочесть на путях из Н в К.

Замечание. Если нарисовать состояния конечного автомата в виде точек, а переходы при чтении букв изобразить в виде стрелок, то станет ясно, что конечный автомат — это частный случай источника. (С дополнительными требованиями: (а) на всех стрелках, за исключением ведущих в К, есть буквы; (б) для любой точки на выходящих из неё стрелках каждая буква встречается ровно один раз.)

Мы будем строить конечный автомат по регулярному выражению в два приёма. Сначала мы построим источник, которому соответствует то же самое множество слов. Затем для произвольного источника построим автомат, который проверяет, принадлежит ли слово соответствующему множеству.

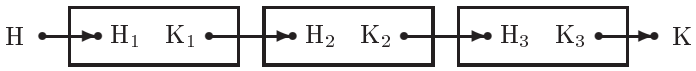
10.7.10. По регулярному выражению построить источник, задающий то же множество.

Решение. Индукция по построению регулярного выражения. Буквам соответствуют графы из одной стрелки. Объединение реализуется так:

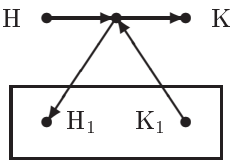


Нарисована картинка для объединения трёх множеств, прямоугольники — это источники, им соответствующие; указаны начальные и конечные вершины. На новых стрелках (их 6) букв не написано.

Конкатенации соответствует картинка



Наконец, итерации соответствует картинка



□

10.7.11. Дан источник. Построить конечный автомат, проверяющий, принадлежит ли входное слово соответствующему множеству (то есть можно ли прочесть это слово, идя из Н в К).

Решение. Состояниями автомата будут множества вершин источника. Именно, прочтя некоторое начало X входного слова, мы будем помнить множество всех вершин источника, в которые можно пройти из начальной, прочитав на пути слово X. □

Тем самым задача 10.7.9 решена. □

Оказывается, что регулярные выражения, автоматы и источники распознают одни и те же множества. Чтобы убедиться в этом, нам осталось решить такую задачу:

10.7.12. Дан источник. Построить регулярное выражение, задающее то же множество, что и этот источник.

Решение. Пусть источник имеет вершины $1, \ldots, k$. Будем считать, что 1 — это начало, а k — конец. Через $D_{i,j,s}$ обозначим множество всех слов, которые можно прочесть на пути из i в j, если в качестве промежуточных пунктов разрешается использовать только вершины $1, \ldots, s$. Согласно определению, источнику соответствует множество $D_{1,k,k}$.

Индукцией по s будем доказывать регулярность всех множеств $D_{i,j,s}$ при всех i и j. При $s = 0$ это очевидно (промежуточные вершины запрещены, поэтому каждое из множеств состоит только из букв).

Из чего состоит множество $D_{i,j,s+1}$? Отметим на пути моменты, в которых он заходит в $(s+1)$-ую вершину. При этом путь разбивается на части, каждая из которых уже не заходит в неё. Поэтому легко сообразить, что

$$D_{i,j,s+1} = D_{i,j,s} \,|\, (D_{i,s+1,s} \;\; D_{s+1,s+1,s}* \;\; D_{s+1,j,s})$$

(вольность записи: мы используем для операций над множествами обозначения как для регулярных выражений). Остаётся воспользоваться предположением индукции. □

10.7.13. Где ещё используется то же самое рассуждение?

Ответ. В алгоритме Флойда вычисления цены кратчайшего пути, см. главу 9 (Разные алгоритмы на графах). □

10.7.14. Доказать, что класс множеств, задаваемых регулярными выражениями, не изменился бы, если бы мы разрешили использовать не только объединение, но и отрицание (а следовательно, и пересечение — оно выражается через объединение и отрицание).

Решение. Для автоматов переход к отрицанию очевиден. □

Замечание. На практике важную роль играет число состояний автомата. Оказывается, что тут всё не так просто, и переход от источника к автомату требует экспоненциального роста числа состояний. Подробное рассмотрение связанных с этим теоретических и практических вопросов — дело особое (см. книгу Ахо, Ульмана и Сети о компиляторах).

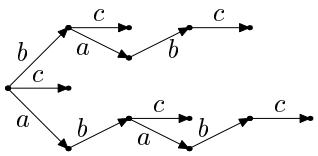
10.8. Суффиксные деревья

До сих про наши программы сначала получали образец, который надо искать, а потом текст, в котором надо искать. В следующих задачах всё наоборот.

10.8.1. Программа получает на вход слово Y длины m и может его обрабатывать (пока без ограничений на время и память). Затем она получает слово X длины n и должна сообщить, является ли оно подсловом слова Y. При этом число операций при обработке слова X должно быть порядка n (не превосходить cn, где константа c может зависеть от размера алфавита). Как написать такую программу?

Решение. Пока не накладывается никаких ограничений на время и память при обработке Y, это не представляет труда. Именно, надо склеить все подслова слова Y в дерево, объединив слова с общими нача-

лами (как мы это делали, распознавая вхождения нескольких образцов). Например, для $Y = ababc$ получится такое дерево подслов (на ребре написана буква, которая добавляется при движении по этому ребру; вершины находятся во взаимно однозначном соответствии с подсловами слова Y):



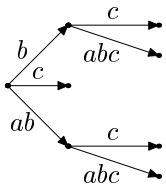
Пусть такое дерево построено. После этого, читая слово X слева направо, мы прослеживаем X в дереве, начав с корня; слово X будет подсловом слова Y, если при этом мы не выйдем за пределы дерева. □

Заметим, что аналогичная конструкция годится для любого множества слов U, а не только для множества всех подслов данного слова: после того как соответствующее дерево построено, мы можем про любое слово X определить его принадлежность к U за время, пропорциональное длине X. (Надо только дополнительно хранить в вершине дерева информацию, принадлежит ли соответствующее ей слово множеству U или лишь является началом другого слова, принадлежащего U.)

10.8.2. Решить предыдущую задачу с дополнительным ограничением: объём используемой памяти пропорционален длине слова Y.

Решение. Прежний способ не годится: число вершин дерева равно числу подслов слова Y, а у слова длины m число подслов может быть порядка m^2, а не m. Однако мы можем «сжать» наше дерево, оставив вершинами лишь точки ветвления (где больше одного сына). Тогда на рёбрах дерева надо написать уже не буквы, а куски слова Y.

Вот что получится при сжатии нашего примера:



Будем считать (здесь и далее), что последняя буква слова Y больше в нём не встречается. (Этого всегда можно достичь, дописав дополнительный фиктивный символ.) Тогда листья сжатого дерева соответствуют концам слова Y, а внутренние вершины (точки ветвления) —

таким подсловам s слова Y, которые встречаются в Y несколько раз, и притом с разными буквами после s.

У каждой внутренней вершины (не листа) сжатого дерева есть не менее двух сыновей. В деревьях с такими свойствами число внутренних вершин не превосходит числа листьев. (В самом деле, при движении слева направо в каждой точке ветвления добавляется новый путь к листу.) Поскольку листьев m, всего вершин не более $2m$, и мы уложимся в линейную по m память, если будем экономно хранить пометки на рёбрах. Каждая такая пометка является подсловом слова Y, и потому достаточно указывать координату её начала и конца в Y. Это не помешает впоследствии прослеживать произвольное слово X в этом дереве буква за буквой, просто в некоторые моменты мы будем находиться внутри рёбер (и должны помнить, внутри какого ребра и в какой позиции мы находимся). При появлении новой буквы слова X её нужно сравнить с соответствующей буквой пометки этого ребра (что можно сделать за $O(1)$ действий, так как координату этой буквы мы знаем.) □

Построенное нами сжатое дерево называют *сжатым суффиксным деревом* слова Y (концы слова называют «суффиксами»).

10.8.3. Показать, что построение сжатого суффиксного дерева можно выполнить за время $O(m^2)$ с использованием $O(m)$ памяти.

Решение. Будем добавлять в суффиксное дерево суффиксы по очереди. Добавление очередного суффикса делается так же, как и проверка принадлежности: мы читаем его буква за буквой и прокладываем путь в дереве. В некоторый момент добавляемый суффикс выйдет за пределы дерева (напомним, что мы считаем, что последний символ слова уникален).

Если это произойдёт посередине ребра, то ребро придётся в этом месте разрезать. Ребро превратится в два, его пометка разрежется на две, появится новая вершина (точка ветвления) и её новый сын-лист. Если точка ветвления совпадёт с уже имевшейся в дереве, то у неё появится новый сын-лист. В любом случае после обнаружения места ветвления требуется $O(1)$ операций для перестройки дерева (в частности, разрезание пометки на две выполняется легко, так как пометки хранятся в виде координат начала и конца в слове Y). □

Гораздо более сложной задачей является построение сжатого суффиксного дерева за линейное время (вместо квадратичного, как в предыдущей задаче). Чтобы изложить алгоритм МакКрейта, который решает эту задачу, нам понадобятся некоторые приготовления.

Для начала опишем более подробно структуру дерева, которое мы используем, и операции с ним.

Мы рассматриваем деревья с корнем, на рёбрах которых написаны слова (пометки); все пометки являются подсловами некоторого заранее фиксированного слова Y. При этом выполнены такие свойства:

- каждая внутренняя вершина имеет хотя бы двух сыновей;
- пометки на рёбрах, выходящих из данной вершины, начинаются на разные буквы.

Каждой вершине v такого дерева соответствует слово, которое записано на пути от корня r к вершине v. Будем обозначать это слово $s(v)$. Обозначим пометку на ребре, ведущем к v, через $l(v)$, а отца вершины v — через $f(v)$. Тогда $s(r) = \Lambda$ (пустое слово), а

$$s(v) = s(f(v)) + l(v)$$

для любой вершины $v \neq r$ (знак «+» обозначает соединение строк).

Помимо вершин дерева, мы будем рассматривать *позиции* в нём, которые могут быть расположены в вершинах, а также «внутри рёбер» (разделяя пометку этого ребра на две части). Формально говоря, позиция представляет собой пару (v, k), где v — вершина (отличная от корня), а k — целое число в промежутке $[0, |l(v)|)$, указывающее, на сколько букв надо вернуться от v к корню. Здесь $|l(v)|$ — длина пометки $l(v)$; значение $k = l(v)$ соответствовало бы предыдущей вершине и потому не допускается. К числу позиций мы добавляем также пару $(r, 0)$, соответствующую корню дерева. Каждой позиции $p = (v, k)$ соответствует слово $s(p)$, которое получается удалением k последних символов из $s(v)$.

Пусть p — произвольная позиция в дереве, а w — слово. Пройти вдоль w, начиная с p, означает найти другую позицию q, для которой $s(q) = s(p) + w$. Если такая позиция есть, то (при описанном способе хранения пометок, когда указываются координаты их начала и конца внутри Y) её можно найти за время, пропорциональное длине слова w. Если такой позиции нет, то в какой-то момент мы «свернём с пути»; в этот момент можно пополнить дерево, сделав отсутствующую в дереве часть слова w пометкой на пути к новому листу. Надо только, чтобы эта пометка была подсловом слова Y (при нашем способе хранения пометок); это будет гарантировано, если прослеживаемое слово w является подсловом слова Y.

Заметим, что при этом может образоваться новая вершина (если развилка оказалась внутри ребра), а может и не образоваться (если развилка оказалась в вершине). Число действий при такой модификации

пропорционально длине пройденной части слова (длина непройденной не важна).

Оказывается, что навигацию в дереве можно ускорить, если заранее известно, что она будет успешной.

10.8.4. Пусть для данной позиции p и слова w заранее известно, что в дереве есть позиция q, для которой $s(q) = s(p) + w$. Показать, что позицию q можно найти за время, пропорциональное числу рёбер дерева на пути от p к q. (Это число может быть значительно меньше длины слова w, если пометки на рёбрах длинные.)

Решение. В самом деле, при навигации нужно ориентироваться лишь в вершинах (выбирать исходящее ребро в зависимости от очередной буквы); в остальных местах путь однозначный и потому можно сдвигаться сразу к концу ребра. □

Подведём итоги. Рассмотренный способ хранения деревьев позволяет (для фиксированного слова Y)

- создать дерево из одного корня $[O(1)]$;
- найти отца любой вершины (кроме корня) $[O(1)]$;
- узнать пометку любой вершины (кроме корня), то есть пометку ведущего к ней ребра $[O(1)]$;
- пройти из любой позиции p вдоль любого слова w, если заранее известно, что мы не выйдем из дерева; результатом является позиция q в дереве, для которой $s(q)=s(p)+w$ $[O(\text{число рёбер на пути})]$;
- добавить слово w (некоторое подслово слова Y), начав с позиции p; если при этом в дереве нет позиции q, для которой $s(q) = s(p) + w$, то дерево меняется и такая позиция q создаётся (она будет листом) $[O(\text{число букв в } w, \text{ не вошедших в } l(q)]$;
- наконец, для любого слова X можно выяснить, найдётся ли в дереве позиция q, для которой $s(q) = X$ $[O(|X|)]$.

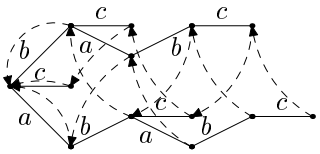
В квадратных скобках указано число действий при выполнении соответствующих операций.

Ещё мы будем хранить в вершинах дерева «суффиксные ссылки» (в каждой вершине будет не более одной ссылки на другую вершину), но сначала надо объяснить, что это такое.

Начнём с полного (не сжатого) суффиксного дерева для слова Y. Каждой его вершине (кроме корня) отвечает некоторое непустое подслово слова Y. Если мы отрежем у этого подслова последнюю букву,

то в дереве спустимся на один шаг к корню. Но что будет, если мы отрежем первую букву? Снова получится подслово, но оно уже будет совсем в другом месте дерева.

Вот как выглядят эти переходы в нашем примере (отрезание первой буквы соответствует пунктирной стрелке):



Эти стрелки мы будем называть *суффиксными ссылками*, поскольку они соответствуют переходу от слова к его суффиксу на единицу меньшей длины. Они определены для всех вершин, кроме корня.

Формально можно сказать так. Пусть w' означает слово w без первой буквы (w' определено для любого непустого слова w). Тогда суффиксная ссылка ведёт из вершины p в вершину q, если $s(q) = s(p)'$ (напомним, что $s(u)$ — слово, соответствующее вершине u).

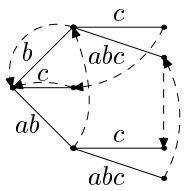
10.8.5. Как связаны суффиксные ссылки двух соседних вершин (отца и сына)?

Ответ. Они указывают на соседние вершины, и буква на соединяющем их ребре та же самая. □

10.8.6. Доказать, что при переходе к сжатому суффиксному дереву ссылки по-прежнему идут из вершины в вершину (а не внутрь рёбер).

Решение. В самом деле, по нашему предположению последняя буква слова больше в нём не встречается, поэтому из листа ссылка ведёт в лист. А если вершина (отличная от корня) является точкой ветвления, то соответствующее ей слово s встречается с различными буквами после него. Другими словами, для некоторых букв a и b слова sa и sb являются подсловами слова Y. Отрезав от них первую букву, получим слова $s'a$ и $s'b$, которые также являются подсловами слова Y, поэтому и s' является точкой ветвления. □

Вот что получится для нашего примера:



Теперь мы уже готовы к изложению алгоритма МакКрейта. Сжатое суффиксное дерево строим постепенно, добавляя к нему суффиксы по мере уменьшения их длины. Обозначим через Y_i суффикс, начинающийся с i-ой буквы слова Y. (Таким образом, $Y_1 = Y$, а Y_m состоит из одной буквы.) После i шагов построения наше дерево будет хранить $Y_1, \ldots, Y_i$.

10.8.7. Показать, что суффиксные ссылки в таком дереве определены корректно (ведут в другую вершину того же дерева) для всех вершин, кроме, возможно, последнего добавленного листа (соответствующего слову Y_i) и его отца.

Решение. В самом деле, суффиксная ссылка из листа Y_j ведёт в лист Y_{j+1}, и потому ей есть куда вести. Рассмотрим теперь внутреннюю вершину v, не являющуюся отцом последнего листа. Пусть в ней разветвляются два пути в листья Y_j и Y_k. Без ограничения общности можно считать, что $j, k < i$ (если один из путей ведёт в Y_i, то его можно заменить другим, ведь вершина v по предположению не последняя развилка на этом пути). Отрезав от этих путей первый символ, получим пути в листья Y_{j+1} и Y_{k+1}; эти пути присутствуют в дереве (поскольку $j+1$ и $k+1$ не превосходят i), а точка их развилки будет концом суффиксной ссылки вершины v. □

Суффиксные ссылки для листьев нам не понадобятся, и вычислять мы их не будем, а для всех остальных вершин дерева мы их будем вычислять и хранить. Более точно, после i шагов алгоритма

- в дереве хранятся слова $Y_1, \ldots Y_i$ (и все их начала);
- адрес листа, соответствующего последнему добавленному суффиксу (Y_i) хранится в переменной *last*;
- для всех внутренних вершин дерева, кроме, быть может, отца вершины *last*, хранится правильная суффиксная ссылка.

Надо понять, как поддерживать это при добавлении очередного суффикса. Можно, не мудрствуя лукаво, добавлять Y_{i+1} буква за буквой, начиная с корня дерева. (Именно так мы раньше и делали, и это требовало квадратичного времени.)

Какие тут возможны оптимизации? Первая связана с тем, что мы можем двигаться по дереву быстрее, если знаем, что заведомо из него не выйдем. Вторая связана с использованием суффиксных ссылок.

Оба варианта предполагают, что отец u листа *last* не совпадает с корнем. (Если совпадает, нам придётся добавлять Y_{i+1} от корня.)

Пусть *tail* — пометка листа *last*, а $head = s(u)$; другими словами, слово *head* соответствует вершине u. Тогда

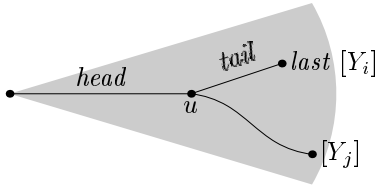
$$Y_i = head + tail.$$

Отрезая первую букву, получаем

$$Y_{i+1} = head' + tail.$$

Заметим, что $head'$ заведомо не выходит за пределы дерева. В самом деле, u было точкой ветвления, поэтому помимо листа Y_i через точку u проходил и лист Y_j с $j < i$. Тогда Y_j начинается на *head*, а Y_{j+1} начинается на $head'$ и уже есть в дереве.

Поэтому мы можем сначала проследить $head'$ (найти позицию v, для которой $s(v) = head'$), а потом уже добавить *tail*, начиная с v.



Первый способ оптимизации: $head'$ заведомо есть в дереве.

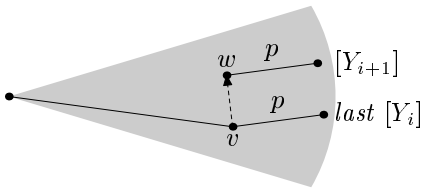
Эта оптимизация никак не использует суффиксных ссылок. Второй способ оптимизации их использует и позволяет (в том случае, когда применим) обойтись без прослеживания Y_{i+1} от корня (как ускоренного, так и обычного). Пусть на пути к листу *last*, представляющему суффикс Y_i, имеется вершина v, у которой суффиксная ссылка указывает на вершину w, так что $s(w) = s(v)'$. Пусть p — слово на пути от v к *last*. Тогда

$$Y_i = s(v) + p, \quad Y_{i+1} = Y_i' = s(v)' + p = s(w) + p,$$

и для добавления Y_{i+1} в дерево достаточно добавить слово p, начиная с вершины w.

Второй способ оптимизации сочетается с первым: отрезок слова p от v до отца листа *last* можно проходить с уверенностью, что мы не выйдем за пределы дерева.

Итак, мы можем описать действия, выполняемые при добавлении очередного суффикса Y_{i+1} в дерево, следующим образом.



Второй способ оптимизации: пользуемся суффиксной ссылкой вершины на пути к *last*.

Пусть u — отец листа *last*, соответствующего последнему уже добавленному суффиксу Y_i.

Случай 1: u есть корень дерева. Тогда ни одна из оптимизаций не применима, и мы добавляем Y_{i+1}, начиная от корня.

Случай 2: u не есть корень дерева, но отец u есть корень дерева (лист *last* находится на высоте 2). Тогда $Y_i = head + tail$, где *head* и *tail* — пометки вершин u и *last*. Мы применяем первую оптимизацию и прослеживаем $head'$ с гарантией до некоторой позиции z, а потом добавляем *tail* от z.

Случай 3: u не есть корень дерева и его отец v также не есть корень дерева. Тогда для v имеется суффиксная ссылка на некоторую вершину w, и $s(w) = s(v)'$. Пусть *pretail* — пометка вершины u, а *tail* — пометка листа *last*, при этом $Y_i = s(v) + pretail + tail$ и потому $Y_{i+1} = Y_i' = s(w) + pretail + tail$. Остаётся проследить *pretail* от вершины w с гарантией, получив некоторую позицию z, а потом добавить *tail* от вершины z.

Остаётся ещё понять, как поддерживать структуру суффиксных ссылок (адрес нового листа у нас получается при добавлении сам собой, так что с ним проблем нет) и оценить число действий при выполнении этих процедур последовательно для всех суффиксов.

Начнём с суффиксных ссылок. По правилам они должны быть у всех внутренних вершин, кроме отца только что добавленного листа. Поэтому нам надо заботиться об отце листа *last*, соответствующего Y_i (этот лист перестал быть «только что добавленным»; напротив, единственная новая вершина как раз является отцом только что добавленного листа и в ней суффиксная ссылка не нужна). Это актуально в случаях 2 и 3, но в этих случаях по ходу дела была найдена нужная вершина z, куда и будет направлена суффиксная ссылка из u. Строго говоря, z могла быть не вершиной, а позицией, но тогда после добавления она станет вершиной (отцом только что добавленного листа) — ведь в новом де-

реве u уже не является отцом последнего листа, и потому суффиксная ссылка из u, как было доказано, должна вести в вершину. (Другими словами, в случаях 2 и 3, если позиция z была внутри ребра, то в ней ребро разрезается.)

Всё сказанное можно условно записать в виде такого алгоритма добавления суффикса Y_{i+1}:

```
{ дерево содержит суффиксы Y_1, ..., Y_i
  s(last) = Y_i
  имеются корректные суффиксные ссылки для всех
  внутренних вершин, кроме отца листа last }
u := отец листа last;
tail := пометка листа last;
{ Y_i = s(u) + tail }
if u = корень дерева then begin
| { Y_{i+1} = tail' }
| добавить tail', начиная с корня,
|       полученный лист поместить в last
end else begin
| v := отец вершины u;
| pretail := пометка вершины u;
| { Y_i = s(v) + pretail + tail }
| if v = корень дерева then begin
| | { Y_{i+1} = pretail' + tail }
| | проследить pretail' из корня в z
| end else begin
| | w := суффиксная ссылка вершины v;
| | { s(w) = s(v)', Y_{i+1} = s(w) + pretail + tail }
| | проследить pretail из w в z
| end;
| { осталось добавить tail из z и ссылку из u в z }
| if позиция z является вершиной then begin
| | поместить в u ссылку на z;
| | добавить tail, начиная с z,
| |       полученный лист поместить в last;
| end else begin
| | добавить tail, начиная с z,
| |       полученный лист поместить в last;
| | поместить в u ссылку на отца листа last;
| end
end;
```

Осталось оценить число действий, которые выполняются при последовательном добавлении суффиксов $Y_1, \ldots, Y_m$. При добавлении каждого следующего суффикса выполняется ограниченное количество действий, если не считать действий при «прослеживании» и «добавлении». Нам надо установить, что общее число действий есть $O(m)$; для этого достаточно отдельно доказать, что суммарное число действий при всех прослеживаниях есть $O(m)$ и суммарное число действий при всех добавлениях есть $O(m)$. (Заметим, что некоторые прослеживания или добавления могут быть долгими — но это компенсируется другими.)

Прослеживания. Длительность прослеживания пропорциональна числу k задействованных в нём рёбер, но при этом высота последнего добавленного листа (число рёбер на пути к нему) увеличивается на $k - O(1)$ (по сравнению с предыдущим добавленным листом). Чтобы убедиться в этом, достаточно заметить, что в третьем случае высота вершины $s(w)$ может быть меньше высоты вершины $s(v)$ разве что на единицу, поскольку суффиксные ссылки из всех вершин на пути к v (не считая корня, где нет суффиксной ссылки) ведут в вершины на пути к w. Поскольку высота любого листа ограничена числом m, заключаем, что общая длительность всех прослеживаний есть $O(m)$.

Добавления. Рассуждаем аналогично, но следим не за высотой последнего листа, а за длиной его пометки. При добавлении слова *tail* (или *tail′*) число действий пропорционально числу просмотренных букв, но каждая просмотренная буква (кроме, быть может, одной) уменьшает длину пометки хотя бы на единицу: в пометке остаются лишь непросмотренные буквы (не считая первой). Поэтому на все добавления уходит в общей сложности $O(m)$ действий.

Тем самым мы доказали, что описанный алгоритм строит сжатое суффиксное дерево слова Y длины m за $O(m)$ действий. После этого для любого слова X длины n можно за $O(n)$ действий выяснить, является ли X подсловом слова Y.

10.8.8. Как модифицировать алгоритм построения суффиксного дерева, чтобы не только узнавать, является ли данное слово X подсловом слова Y, но и (если является) указывать место, где оно встречается (одно из таких мест, если их несколько)? Время построения должно оставаться $O(|Y|)$, время поиска подслова — $O(|X|)$.

Решение. Каждая вершина сжатого суффиксного дерева соответствует некоторому подслову слова Y; в момент, когда эта вершина была добавлена в дерево, известно, в каком месте есть такое подслово, и можно записать в вершине, где соответствующее подслово кончается. □

10.8.9. Как модифицировать этот алгоритм, чтобы для каждого подслова можно было бы указывать его первое (самое левое) вхождение?

[Указание. При возникновении новой вершины на ребре нужно брать её первое вхождение (информация о котором есть на конце ребра), а не второе, только что обнаруженное.] □

10.8.10. Как модифицировать этот алгоритм, чтобы для каждого подслова можно было бы указывать его последнее (самое правое) вхождение?

[Указание. Если при каждом проходе корректировать информацию вдоль пути, это будет долго; быстрее построить дерево, затем вновь его обойти и для каждой вершины вычислить момент последнего появления соответствующего подслова.] □

10.8.11. Как использовать сжатое суффиксное дерево, чтобы для данного слова Y за время $O(|Y|)$ найти самое длинное подслово, которое входит в Y более одного раза?

Решение. Такое подслово является внутренней вершиной сжатого суффиксного дерева, поэтому достаточно из всех его вершин взять ту, которой соответствует самое длинное слово. Для этого достаточно обойти все его вершины (длину можно вычислять по мере обхода, складывая длины пометок на рёбрах). □

На практике можно использовать также и другой способ нахождения самого длинного подслова, входящего дважды, — так называемый *массив суффиксов*. А именно, будем рассматривать число i как «код» конца слова, начинающего с i-ой буквы. Введём на кодах порядок, соответствующий лексикографическому (словарному) порядку на словах: код i предшествует коду j, если конец слова, начинающийся с i, в лексикографическом порядке идёт раньше конца слова, начинающегося с j. После этого отсортируем коды в соответствии с этим порядком, получив некоторую перестановку массива $1, 2, 3, \ldots, m$ (где m — длина исходного слова Y). Если какое-то слово X входит в слово Y дважды, то оно является началом двух концов слова Y. При этом эти концы можно выбрать соседними в лексикографическом порядке, поскольку все промежуточные слова тоже начинаются на X. Значит, достаточно для всех соседних концов посмотреть, сколько начальных букв у них совпадает, и взять максимум.

Этот способ требует меньше памяти (нам нужен лишь один массив из целых чисел той же длины, что исходное слово), но может требовать большого времени: во-первых, сортировка сама по себе требует

порядка $m \log m$ сравнений, во-вторых, каждое сравнение может длиться долго, если совпадающий кусок большой. Но в случаях, когда длинных совпадающих кусков мало, такой алгоритм работает неплохо.

10.8.12. Применить один из таких алгоритмов к любимой книге и объяснить результат.

[Указание. Длинные повторяющиеся куски могут быть художественным приёмом (как в известном стишке про дом, который построил Джек) или следствием забывчивости автора. Для современных авторов возможно также неумеренное использование функций вырезания и вставки (заливки текста в мышь и выливания из мыши, если использовать графический интерфейс) в текстовом редакторе.] □

11. АНАЛИЗ ИГР

11.1. Примеры игр

11.1.1. Двое играют в такую игру: на столе лежит 20 спичек; играющие по очереди могут взять от 1 до 4 спичек; кто не может сделать хода (спичек не осталось) — проигрывает. Кто выигрывает при правильной игре?

Решение. Второй выигрывает, если будет дополнять ход первого до 5 спичек (если первый берёт одну, второй должен взять четыре и так далее). Тогда после четырёх раундов спичек не останется и первый проиграет. □

11.1.2. Кто выиграет — первый или второй — если спичек не 20, а 23?

Решение. Первый: если он возьмёт три спички, то станет вторым в уже разобранной игре и потому сможет выиграть. □

Аналогично получается ответ и для произвольного числа спичек (N): если N кратно пяти, то выигрывает второй, а если нет, то первый.

11.1.3. Изменим условия игры: пусть взявший последнюю спичку проигрывает. Кто теперь выигрывает при правильной игре? □

11.1.4. Пусть теперь игрокам разрешено брать 1, 2 или 4 спички, а кто не может сделать ход, проигрывает. Кто выигрывает при правильной игре, если вначале было 20 спичек?

Решение. Здесь уже не так просто сразу указать выигрышную стратегию для первого или второго. Начнём с небольшого числа спичек, изобразив разрешённые ходы в виде стрелок (рис. 11.1): Игрок, оказавшийся в позиции 0, проигрывает (таковы правила), поэтому соответствующий кружок пометим буквой П. Игрок, оказавшийся в позициях 1, 2 или 4, выигрывает, поскольку он может забрать все спички и перевести противника по стрелке в позицию 0. Поэтому мы пометим

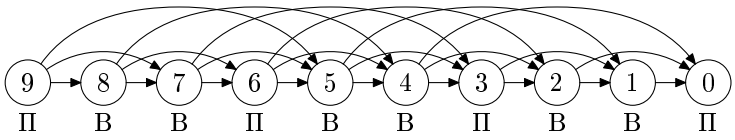


Рис. 11.1. Игра со спичками.

эти позиции буквой В. Теперь ясно, что позиция 3 является проигрышной: из неё можно пойти только в 1 и 2, и тогда противник (как мы уже знаем) выиграет. Пометим её буквой П. Далее замечаем, что позиции 4, 5 и 7 будут выигрышными (поскольку из них можно попасть в проигрышную для противника позицию 3; заметим, что из позиции 4 можно выиграть и быстрее, пойдя в 0). Теперь видно, что позиция 6 проигрышная (все стрелки из неё ведут в выигрышные для противника позиции), 8 — выигрышная, 9 — проигрышная и так далее с периодом 3.

Таким образом, если число спичек делится на 3, то позиция проигрышная, если нет — то выигрышная. Поэтому в игре с 20 спичками первый игрок выигрывает. □

11.1.5. Как он для этого должен играть?

Решение. Ставить противника в проигрышную позицию, то есть следить, чтобы после его хода число спичек было кратно трём (в частности, в начале игры взять 2 спички, чтобы осталось 18). □

11.1.6. На столе лежат две кучки спичек: в одной m, в другой n. За один ход разрешается взять любое (ненулевое) число спичек, но только из одной кучки (можно взять все спички в ней); кто не может сделать ход, проигрывает. Кто выигрывает при правильной игре?

Ответ: при $m = n$ выигрывает второй, при $m \neq n$ — первый. □

11.1.7. На шахматной доске стоит ладья, которую игроки по очереди двигают, при этом разрешено сдвигать её влево и вниз (оставлять на месте нельзя); кто не может сделать ход, проигрывает. Кто выигрывает при правильной игре?

[Указание. Как эта игра связана с предыдущей?] □

11.1.8. (Игра «ним») Имеется k кучек из $n_1, \ldots, n_k$ спичек; за один ход можно взять любое (ненулевое) число спичек, но только из одной кучи (можно взять все спички в ней); кто не может сделать ход, проигрывает. Кто выигрывает при правильной игре?

Решение. Запишем числа $n_1, \ldots, n_k$ в двоичной системе счисления друг под другом, как если бы мы собирались их складывать. Если в каждом разряде при этом оказалось чётное число единиц, то выигрывает второй, в остальных случаях — первый. В самом деле, если во всех разрядах чётное число единиц, то после уменьшения одного из чисел какой-то из его разрядов изменится и в этом разряде получится нечётное число единиц. (Это соответствует тому, что из проигрышной позиции любой ход ведёт в выигрышную.) Если же в некоторых («плохих») разрядах нечётное число единиц, возьмём старший плохой разряд и то из чисел, которое содержит в этом разряде единицу. Тогда, изменив в этом числе все плохие разряды, получим меньшее число, которое поставит противника в проигрышную позицию. (См. правила для выигрышных и проигрышных позиций в следующем разделе.) □

11.1.9. В ряд лежат N ящиков, в каждом из них по монете. За один ход игрок может взять любую монету или любые две монеты из соседних ящиков; кто не может сделать ход, проигрывает. Кто выигрывает при правильной игре?

Решение. Первый: он должен взять одну или две монеты в центре, а потом симметрично повторять ходы второго. □

11.2. Цена игры

Анализируя игры в предыдущем разделе, мы использовали следующие (очевидные) правила:

1. Если из некоторой позиции p можно пойти (по стрелкам) в некоторую проигрышную (для попавшего в неё игрока) позицию, то позиция p является выигрышной (для попавшего в неё).

2. Если из некоторой позиции p можно пойти только в выигрышные позиции, то позиция p является проигрышной.

11.2.1. Доказать, что если число позиций в игре конечно, нет циклов (нельзя вернуться в однажды пройденную позицию) и про все заключительные позиции (где нельзя сделать хода) известно, кто выигрывает, то правила 1 и 2 однозначно разбивают все позиции на выигрышные и проигрышные.

Решение. Будем применять эти правила, пока это возможно. Ясно, что никакая позиция не будет объявлена одновременно выигрышной и проигрышной (для попавшего в неё). Надо лишь доказать, что не останется «сомнительных» позиций (не отнесённых ни к выигрышным,

ни к проигрышным). Заметим, что из каждой сомнительной позиции ведёт стрелка хотя бы в одну сомнительную позицию. (В самом деле, если все стрелки ведут в несомненные позиции, то либо все они выигрышные, либо есть хоть одна проигрышная, и можно было бы воспользоваться одним из двух правил.) Значит, идя по стрелкам в сомнительные позиции, мы рано или поздно получим цикл, что противоречит предположению. □

11.2.2. Сформулировать и доказать аналогичное утверждение для игр, допускающих ничьи. □

Игры с ничейным исходом являются частными случаями *конечных игр с полной информацией и нулевой суммой*, рассматриваемых в теории игр. Чтобы задать такую игру, необходимо:

1) указать конечное множество, элементы которого называются *позициями*;

2) для каждой позиции указать, является ли она *заключительной* (игра закончена) или нет;

3) для каждой заключительной позиции указать результат игры (число); это число понимается как сумма денег, которую один игрок платит другому;

4) для каждой незаключительной позиции указать, кто из игроков должен делать ход в этой позиции и какие разрешены ходы (в какие позиции этот игрок может перейти);

5) указать начальную позицию игры.

При этом требуется, чтобы не было циклов (нельзя было вернуться в уже пройденную позицию после нескольких ходов).

Позиции игры удобно рассматривать как вершины графа и изображать точками; возможные ходы при этом становятся рёбрами графа и изображаются стрелками. Игру можно рассматривать как передвижение фишки, обозначающей текущую позицию игры, по этому графу. В каждой вершине написано, кто должен делать ход (если игра не кончилась) или кто и сколько выиграл (если игра кончилась). Одна из вершин указана как начальная позиция.

Будем называть игроков Макс и Мин и считать, что результат игры определяет, сколько Мин платит Максу. (Мотивировка: Макс хочет, чтобы это число было максимальным, а Мин — минимальным — а лучше всего отрицательным, поскольку тогда он получает деньги!) Кто

из игроков делает первый ход, определяется начальной позицией. Заметим, что мы теперь не предполагаем, что игроки ходят по очереди: один и тот же игрок может делать несколько ходов подряд.

(Тем самым, например, в игре со спичками каждый кружок на рисунке 11.1 теперь превращается в две позиции: с ходом Макса и с ходом Мина.)

Игра, в которой один из игроков выигрывает, а другой проигрывает, соответствует значениям ± 1 в заключительных вершинах ($+1$ означает выигрыш Макса, -1 означает выигрыш Мина). Игры с ничейными исходами получатся, если приписать число 0 ничейным позициям.

Определим теперь понятие *стратегии*. Стратегия для Макса (или Мина) определяет, как он должен ходить в каждой из позиций (где ход за ним); формально это функция s, определённая на множестве позиций, где ход за ним. Значениями этой функции являются позиции, при этом ходы должны быть допустимыми, то есть из p в $s(p)$ должна вести стрелка.

Стратегии такого типа называют в теории игр *позиционными*, подчёркивая, что выбор хода зависит лишь от текущей позиции, но не от истории игры (как мы в эту позицию попали). (Другие стратегии нам не понадобятся, так что слово «позиционная» мы будем опускать.)

Если фиксировать стратегии для Макса и Мина, то исход игры предопределён: эти стратегии однозначно определяют последовательность позиций («партию») и результат игры.

11.2.3. Доказать, что для любой игры G можно найти число c и стратегии M и m для Макса и Мина, при которых:

(1) Макс, пользуясь стратегией M, гарантирует себе выигрыш не менее c, как бы ни играл Мин;

(2) Мин, пользуясь стратегией m, гарантирует себе проигрыш не более c, как бы ни играл Макс.

Число c называют *ценой* игры G. Заметим, что цена игры определяется однозначно: из условий (1) и (2) следует, что у Макса нет стратегии, гарантирующей ему выигрыш больше c (поскольку она не может это сделать против стратегии m), а у Мина нет стратегии, гарантирующей ему проигрыш меньше c.

Для игр с двумя исходами утверждение задачи (называемое *теоремой Цермело*) означает, что ровно у одного из игроков имеется выигрышная стратегия. Если разрешить и ничьи, то либо у одного из игроков есть выигрышная стратегия, либо у обоих есть стратегия, гарантирующая ничью.

Решение. Пусть p — произвольная позиция игры G. Рассмотрим игру G_p, которая отличается от G лишь начальной позицией, и эта начальная позиция есть p. (Если p — заключительная вершина, то игра G_p тривиальна: игра кончается, не начавшись, и игрокам сообщается результат игры.) Как мы сейчас увидим, цену игры G_p (как функцию от p) можно определить рекурсивно, начиная с заключительных позиций.

Более точно, рассмотрим следующее рекурсивное определение некоторой функции c, определённой на вершинах графа:

- $c(p)$ равно выигрышу Макса (=проигрышу Мина) в позиции p, если позиция p является заключительной;
- $c(p) = \max\{c(p')\}$, если в вершине p ходит Макс; максимум берётся по всем вершинам p', в которые Макс может пойти из p по правилам игры;
- $c(p) = \min\{c(p')\}$, если в вершине p ходит Мин; минимум берётся по всем вершинам p', в которые Мин может пойти из p по правилам игры.

Лемма. Это определение корректно: существует и единственна функция c (аргументы — вершины графа, значения — числа), удовлетворяющая указанным требованиям.

Доказательство леммы. Назовём *рангом* вершины максимальное число ходов, которое можно сделать из этой вершины. Поскольку по предположению в игре нет циклов, то ранг любой вершины не больше числа вершин. Докажем индукцией по k, что существует и единственна функция c, определённая на вершинах ранга не больше k и удовлетворяющая рекурсивному определению. Для $k = 0$ это очевидно. Шаг индукции использует такое (очевидное) замечание: если из вершины p можно сделать ход в вершину p', то ранг вершины p' меньше ранга вершины p. Поэтому рекурсивное определение однозначно задаёт значения на вершинах ранга k, если известны значения на вершинах меньших рангов. Лемма доказана.

Осталось доказать, что значение $c(p)$ является ценой игры G_p. Рассмотрим следующую (позиционную) стратегию для Макса: из вершины p ходить в ту вершину p', для которой значение $c(p')$ максимально (и равно $c(p)$). Если Макс следует этой стратегии, то независимо от ходов Мина значение $c(q)$ для текущей вершины q не убывает в ходе игры (при ходах Мина оно убывать вообще не может, при ходах Макса оно не убывает по построению стратегии). Тем самым в вершине p

Максу гарантирован выигрыш не меньше $c(p)$. Аналогичным образом, если Мин ходит в ту вершину p', где достигается минимум $c(p')$ (равный $c(p)$), то значение $c(q)$ не возрастает в ходе игры и потому Мин проигрывает не более $c(p)$.

Теорема Цермело доказана. □

11.2.4. Игра в крестики-нолики состоит в следующем: на большом квадратном поле два игрока по очереди ставят крестики и нолики в ещё не занятые клетки (начинают крестики). Выигрывает тот, кто первым поставит пять своих знаков подряд (по вертикали, горизонтали или диагонали). Если всё поле заполнено, а такого не случилось, партия считается ничейной. Доказать, что у крестиков есть стратегия, гарантирующая им ничью или выигрыш.

Решение. Согласно теореме Цермело, в противном случае у ноликов есть стратегия, гарантирующая им выигрыш. Покажем, что крестики могут использовать по существу ту же стратегию, забыв о своём первом ходе. А именно, представим себе, что крестики делают произвольный первый ход (карандашом), а затем отвечают (чернилами) на ходы ноликов по выигрышной стратегии для ноликов (считая ходы ноликов крестиками и забыв о своём первом ходе).

Может ли при этом первый ход помешать? Может, если стратегия указывает как раз на ту клетку, где уже стоит карандашный крестик. В этом случае надо карандашный крестик обвести чернилами, а карандашом сделать ход в любую свободную клетку. Если свободных клеток нет, то позиция соответствует (с точностью до замены крестиков на нолики) заключительной позиции в выигрышной партии для ноликов, и потому является выигрышной.

Кроме того, игра может кончиться раньше времени, если карандашный крестик образует выигрышный ряд с чернильными — но это нам только лучше.

Таким образом, мы доказали, что если у ноликов есть выигрышная стратегия, то и у крестиков есть выигрышная стратегия — и если дать этим стратегиям играть друг против друга, получится противоречие. □

11.2.5. Доказать, что цена любой игры равна выигрышу в одной из заключительных вершин. □

11.2.6. Показать, что теорема Цермело вытекает из своего частного случая игр с двумя исходами (выигрыш первого и второго).

[Указание. Для каждого c будем считать выигрыш меньше c проигрышем, а больше c — выигрышем.] □

11.2.7. Пусть дана некоторая игра G. Выберем одну из заключительных вершин и будем менять выигрыш в этой вершине: положив его равным c, получим игру $G[c]$. Рассмотрим цену этой игры как функцию от c. Что это может быть за функция?

Ответ. цена игры $G[c]$ равна ближайшей к c точке некоторого отрезка $[a, b]$ (зависящего от игры G). □

Вот ещё один пример игры, где теорема Цермело позволяет доказать существование выигрышной стратегии для первого игрока. Эта игра названа в книгах М. Гарднера («Математические досуги», М.: Мир, 1972; «Математические головоломки и развлечения», М.: Мир, 1971) игрой Гейла или «бридж-ит». Рассмотрим прямоугольную сеть из пунктирных линий высоты n и ширины $n + 1$ (рис. 11.2); вершины

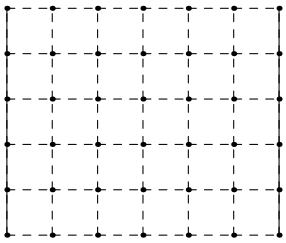


Рис. 11.2. Игра Гейла.

сети соединены пунктирными отрезками длины 1. Первый игрок каждым своим ходом обводит (сплошной линией) один из отрезков. Его задача — соединить сплошными линиями левую и правую стороны прямоугольника. Задача второго игрока — ему помешать; каждым своим ходом он стирает один из отрезков (лишая первого возможности впоследствии его обвести). Игра заканчивается, когда все отрезки обведены или стёрты; первый выиграл, если при этом левая и правая стороны прямоугольника соединены.

11.2.8. Используя теорему Цермело, доказать, что первый игрок имеет выигрышную стратегию.

[Указание. Игру можно представить в более симметричном виде, если добавить сетку для второго игрока (рис. 11.3) и считать, что второй хочет соединить верхнюю и нижнюю стороны своей сетки, а линиям первого и второго игроков запрещено пересекаться (тем самым проведя свою линию, второй игрок как бы стирает пересекающую её линию

первого). Если игра закончилась (в каждой возможной точке пересечения проведена вертикальная или горизонтальная линия), то ровно один из игроков выиграл: можно пройти по линиям или слева направо, или сверху вниз, но не одновременно. Аккуратное доказательство этого интуитивно ясного топологического факта, впрочем, не так просто.] □

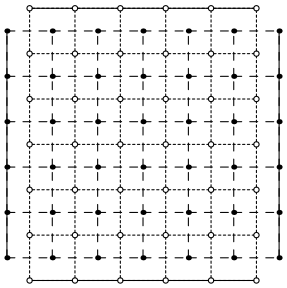


Рис. 11.3. Игра Гейла, симметричный вариант.

Как пишет Гарднер, Клод Шеннон (создатель теории информации) придумал для этой игры любопытную «физическую» стратегию, которая легко обобщается на любую сеть линий. Представим себе, что все стороны всех клеток сети (для первого игрока) представляют собой резисторы одинакового сопротивления, кроме левой и правой сторон прямоугольника, которые сделаны из провода нулевого сопротивления. Первый игрок своим ходом закорачивает эти сопротивления, а второй игрок разрывает их (делает бесконечными). Стратегия первого игрока состоит в том, что надо подключить напряжение между левой и правой сторонами прямоугольника, и закорачивать (обводить) то сопротивление, через которое идёт максимальный ток (или, что то же самое, на котором падает наибольшая разность потенциалов). Если таких сопротивлений оказалось несколько, можно закорачивать любое из них.

Из книг Гарднера не ясно, является ли эта стратегия выигрышной. Зато там приведена явная выигрышная стратегия (со ссылкой на О. Гросса). Чтобы объяснить её, будем считать, что целью первого игрока является не дать второму соединить верх и низ. (Мы уже упоминали, что эта цель равносильна исходной.) Начальный ход первого игрока показан на рис. 11.4; этот ход запрещает одно из рёбер второго игрока. Разделим остальные рёбра второго игрока на пары соседних, как показано на том же рисунке. Первый игрок препятствует второму провести оба ребра какой-либо пары: если второй провёл одно из

рёбер пары, первый не даёт провести второе ребро этой пары (проведя пересекающее его своё ребро). Следующая задача показывает, что

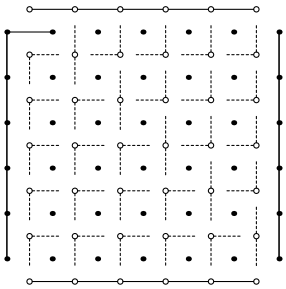


Рис. 11.4. Игра Гейла: выигрышная стратегия

эта стратегия является выигрышной (первый игрок не даёт второму соединить верх и низ и потому соединяет левую и правую стороны).

11.2.9. Доказать, что любой путь по линиям пунктирной сетки, соединяющий верх и низ рисунка 11.4, обязательно покрывает два ребра одной пары.

Решение. Для ясности оставим на рисунке только пунктирные линии и соответствующие вершины (рис. 11.5). Отметим серую область, как показано на рисунке; тем самым рёбра делятся на серые и белые. Предположим, что имеется путь снизу вверх, который не покрывает ни одной пары рёбер. Можно считать, что этот путь не проходит дваж-

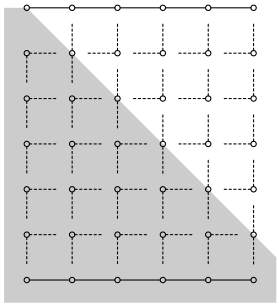


Рис. 11.5. Игра Гейла: анализ выигрышной стратегии.

ды через одну вершину (выбросим циклы). Каждый шаг на этом пути

может относиться к одной из восьми категорий: четыре направления (север, восток, юг и запад) комбинируются с двумя цветами (серым и белым). Как видно из рисунка, путь должен начинаться с серого шага на север, а заканчиваться белым шагом на север.

Покажем, что это невозможно в силу наших ограничений (нельзя использовать два ребра одной пары и нельзя дважды проходить через одну вершину). Что, к примеру, может следовать за серым шагом на север? Ещё один серый шаг на север, серый шаг на запад или белый шаг на восток. За серым шагом на запад может следовать серый шаг на запад или серый шаг на север. Разбирая поочерёдно все варианты, легко убедиться, что путь, начавшись серым шагом на север, никогда не выйдет (если не нарушит правил) за пределы множества

{серый шаг на север, серый шаг на запад,
белый шаг на восток, белый шаг на юг}.

Поэтому белый шаг на север (который должен быть последним в пути) невозможен, и мы доказали, что верх и низ рисунка нельзя соединить путём, не проходящим по двум рёбрам одной пары. □

11.2.10. Двое играют на бесконечной клетчатой бумаге, по очереди обводя красным и синим стороны клеток (за один ход можно обвести одну сторону любой клетки, если она ещё не обведена). Доказать, что второй может воспрепятствовать первому построить замкнутый путь из линий своего цвета.

[Указание. Он может помешать первому, например, повернуть с запада на север, разбив все стороны клеток на пары и не давая покрыть оба члена пары.] □

11.2.11. (Для знакомых с теорией вероятностей) На поле для игры Гейла (рис. 11.2) каждая из пунктирных линий обведена с вероятностью 1/2 независимо от других. Доказать, что путь от левой до правой стороны (по обведённым линиям) существует с вероятностью 1/2. □

11.3. Вычисление цены: полный обход

Как видно из доказательства теоремы Цермело, для нахождения оптимальной стратегии достаточно уметь вычислять цены всех вершин. В этом разделе мы рассмотрим случай, когда позиции игры образуют дерево (ведущие вверх рёбра дерева соответствуют возможным

в данной позиции ходам) и покажем, как применить программу обхода дерева (глава 3).

Напомним, что мы рассматривали Робота, который в каждый момент находится в одной из вершин дерева и умеет выполнять команды вверх_налево, вправо и вниз. Робот начинает работу в корне дерева (роль которого теперь играет начальная позиция игры). Раньше Робот умел ещё обрабатывать вершины; теперь мы предполагаем, что он может определить тип текущей вершины (один из трёх: max, min и final, что соответствует вершинам Макса, Мина и заключительным) и может определить стоимость текущей вершины, если она является заключительной.

11.3.1. Написать программу, которая управляет Роботом и вычисляет цену игры.

Решение. Напишем рекурсивную процедуру, которая, начав с некоторой вершины, обходит поддерево этой вершины, возвращает Робота на место и сообщает цену вершины (где она начала и кончила):

```
procedure find_cost (var c: integer)
| var x: integer;
begin
| if тип = final then begin
| | c:= стоимость;
| end else if тип = max then begin
| | вверх_налево;
| | find_cost (c);
| | {c = максимум цен текущей вершины и братьев слева}
| | while есть_справа do begin
| | | вправо;
| | | find_cost (x);
| | | c := max (c,x);
| | end;
| | {c=цена вершины под текущей}
| | вниз;
| end else begin {тип = мин}
| | ...аналогично с заменой max(c,x) на min(c,x)
| end;
end;
```

Мы пользуемся тем, что у вершин типа max и min есть хотя бы один сын (вершины без сыновей должны быть заключительными, и мы предполагаем, что они отнесены к типу final). □

11.3.2. Написать нерекурсивную программу для вычисления цены игры (заданной деревом, по которому ходит Робот).

Решение. Как обычно, рекурсию можно устранить, используя стек. В данном случае каждый элемент стека будет хранить информацию об одном из предков текущей вершины (чем дальше, тем глубже — на дне стека будет информация о корне). Когда мы находимся в корне, стек пуст, при движении вверх по дереву он удлиняется, при движении вниз — укорачивается.

Каждый элемент стека представляет собой пару; первый элемент — тип соответствующей вершины (`min`/`max`), а второй элемент — минимум/максимум значений всех её сыновей левее текущего. В программе из главы 3 существенную роль играли два утверждения: ОЛ означало, что обработаны все вершины левее текущей (те, путь в которые отклоняется налево от пути в текущую); ОЛН означало, что обработаны все вершины левее и над текущей (это бывало, когда мы проходили вершину второй раз).

Помимо стека (который всегда будет хранить данные, указанные выше) программа использует ещё переменную `c`. В ситуации ОЛ эта переменная не используется, а в ситуации ОЛН она хранит цену текущей вершины. Покажем, как можно поддерживать это, описав действия с переменной и стеком для каждого варианта движения Робота (ср. с. 63):

- {ОЛ, не `есть_сверху`} `обработать` {ОЛН}:
  в переменную `c` записываем цену текущего листа;
- {ОЛ, `есть_сверху`} `вверх_налево` {ОЛ}:
  перед тем, как идти вверх, добавляем в стек тип текущей вершины (`max`/`min`) и значение $-\infty/+\infty$ соответственно, имея в виду, что максимум пустого множества равен $-\infty$, а минимум равен $+\infty$;
- {`есть_справа`, ОЛН} `вправо` {ОЛ}:
  обновляем значение в вершине стека, беря максимум или минимум (в зависимости от типа вершины стека) со значением переменной `c`;
- {не `есть_справа`, `есть_снизу`, ОЛН} `вниз` {ОЛН}:
  в переменную `c` помещаем максимум/минимум (в зависимости от типа вершины стека) её прежнего значения и значения на вершине стека (оно забирается из стека, и стек укорачивается).

Легко видеть, что при этом утверждения о содержании стека и значении переменной `c` не нарушаются, и по окончанию работы программы стек будет пуст, а значение переменной `c` будет равно цене игры. □

11.4. Альфа-бета-процедура

Мы видели, как можно вычислить цену игры, обойдя все вершины её дерева. Однако иногда можно сэкономить и часть дерева не посещать. Пусть, например, игра имеет два исхода (выигрыш и проигрыш) и мы обнаружили (после просмотра части дерева), что некоторый ход является для нас выигрышным. Тогда нет смысла рассматривать остальные ходы. Более общо, если мы нашли ход, гарантирующий нам максимальный выигрыш (допускаемый правилами игры), то нет смысла искать дальше.

Подобная оптимизация возможна не только в тех случаях, когда мы знаем максимально возможный выигрыш. Пусть, например, дерево игры имеет такой вид, как на рис. 11.6, причём $a \geqslant b$ и мы обходим вершины дерева слева направо. Тогда после просмотра вершины a мы

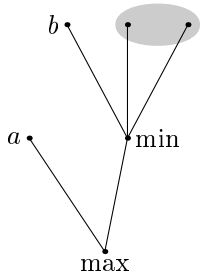


Рис. 11.6. Оптимизация возможна при $a \geqslant b$.

знаем, что цена корневой вершины не меньше a. Перейдя к min-вершине и просмотрев её первого сына b, мы определяем, что цена min-вершины не больше b и (при $b \leqslant a$) она не может повлиять на цену корня. Поэтому следующие вершины (серая область на рисунке) и их поддеревья нам просматривать не нужно.

Применённую в обоих случаях оптимизацию можно описать так. Приступая к оценке некоторой вершины, мы знаем некоторый промежуток $[m, M]$, в пределах которого нас интересует цена этой вершины — либо потому, что она заведомо не может выйти за пределы промежутка (как в первом случае, когда лучше выигрыша ничего не бывает), либо потому, что это нам ничего не даёт (как во втором случае, когда все цены меньше a для нас неотличимы от a).

Более формально, введём обозначение $x_{[a,b]}$, где x — число, а $[a, b]$ —

промежуток:

$$x_{[a,b]} = \begin{cases} a, & \text{если } x \leqslant a; \\ x, & \text{если } a \leqslant x \leqslant b; \\ b, & \text{если } b \leqslant x. \end{cases}$$

Другими словами, $x_{[a,b]}$ — ближайшая к x точка промежутка $[a, b]$, которую можно назвать «приведённым к $[a, b]$ значением x». Теперь можно сказать, что после просмотра вершины a на рисунке 11.6 нас интересует приведённая к $[a, +\infty]$ цена min-вершины (все значения, меньшие a, безразличны), а после просмотра вершины b эта приведённая цена уже известна (равна a). Аналогичным образом цена игры с двумя исходами ± 1 равна её приведённой к отрезку $[-1, +1]$ цене, и после обнаружения выигрышного хода становится ясным, что эта цена равна $+1$.

Используя это соображение, напишем оптимизированный алгоритм, в котором рекурсивно определяется приведённая к промежутку $[a, b]$ цена игры в текущей вершине:

```
procedure find_reduced_cost (a,b: integer; var c: integer)
| var x: integer;
begin
| if тип = final then begin
| | c:= стоимость, приведённая к [a,b]
| end else if тип = max then begin
| | вверх_налево;
| | find_reduced_cost (a,b,c);
| | {c = максимум цены вершины и братьев слева,
| |                     приведённый к [a,b]
| | while есть_справа and (c<b) do begin
| | | вправо;
| | | find_reduced_cost (c,b,x);
| | | c := x;
| | end;
| | {c=цена вершины под текущей, приведённая к [a,b]}
| | вниз;
| end else begin {тип = мин}
| | ...симметрично
| end;
end;
```

Естественный вопрос: насколько такого рода оптимизация помогает уменьшить перебор? Мы рассмотрим простейший пример. Пусть игра имеет фиксированную длину, из каждой позиции возможны два хода, игроки ходят по очереди, каждый делает n ходов и цены листьев

равны 0 или 1. Дерево такой игры — полное двоичное дерево, min- и max-уровни чередуются, в листьях написаны нули и единицы, и нужно вычислить значение в корне. (Если считать, что 1 = истина, 0 = ложь, то максимум и минимум соответствуют операциям OR (ИЛИ) и AND (И), поэтому иногда говорят об AND-OR-дереве.)

Сколько листьев нужно посетить, чтобы вычислить значение в корне? Напомним, что всего листьев 2^{2n} для дерева с $2n$ уровнями (каждый из игроков делает n ходов).

11.4.1. Доказать, что для любых значений в листьях описанный нами оптимизированный алгоритм просматривает не менее 2^n листьев.

Решение. На уровне 2 находятся четыре вершины. В ходе работы алгоритм должен узнать цену игры хотя бы в двух из них. В самом деле, пусть нижняя вершина есть min-вершина. Если в ней нуль, то в одном из её сыновей тоже нуль. А раз это max-вершина, то для установления этого факта нужно знать цену обоих сыновей (равную нулю). Второй случай: в корне единица. Тогда в обеих его сыновьях должна быть единица, и чтобы быть в этом уверенным, нужно в каждом из них посмотреть как минимум одного сына.

Аналогично ради каждого значения на уровне 2 нужны два значения на уровне 4 и так далее — в конце концов на уровне $2n$ нужно знать 2^n значений. □

Для наглядности мы говорили о конкретном алгоритме, описанном выше. Но справедлив и более общий факт: любой набор значений в листьях, который однозначно определяет значение в корне, содержит не менее 2^n значений.

11.4.2. Провести аккуратное доказательство этого утверждения.

[Указание. По существу уже всё доказано, надо только это оформить.] □

Только что полученная оценка относилась к самому благоприятному случаю. Утверждение следующей задачи, напротив, говорит о наихудшем случае.

11.4.3. Пусть у нас спрашивают значения в листьях AND-OR-дерева в заранее неизвестном нам порядке, и мы можем называть эти значения по собственному усмотрению. Доказать, что мы можем действовать так, чтобы до последнего момента (пока есть хоть одно не названное значение) цена корня оставалось бы неизвестной (то есть могла быть и нулём, и единицей в зависимости от ещё не названных значений).

Эта задача показывает, что любой алгоритм отыскания цены корня в наиболее неблагоприятном случае вынужден обходить все листья (в частности, наш оптимизированный алгоритм никакого выигрыша не даёт).

Решение. Будем доказывать это индукцией по высоте дерева. Пусть корень является AND-вершиной. Тогда будем оттягивать (по предположению индукции) определение значений в детях корня, а когда дальше оттягивать будет нельзя (последний лист поддерева становится известным), сделаем так, чтобы это поддерево было истинным. Тем самым значение в корне совпадает со значением в другом поддереве и может оставаться неопределённым до последнего момента. □

Более интересной является оценка среднего числа опрошенных листьев. Будем считать, что алгоритм `find_reduced_cost` применяется к некоторому фиксированному AND-OR-дереву (с фиксированными значениями в листьях), но для каждой вершины порядок просмотра двух её детей выбирается случайно. Тогда общее число просмотренных листьев становится случайной величиной.

11.4.4. Доказать, что математическое ожидание этой случайной величины (среднее по всем порядкам просмотров) для любого AND-OR-дерева высоты $2n$ (с 4^n вершинами) не превосходит 3^n.

Решение. Рассмотрим сначала случай $n = 1$, то есть дерево глубины 2. Пусть его корень является AND-вершиной. Если в корне находится 0, то на первом уровне $0,0$ или $0,1$. В первом случае нам достаточно просмотреть две вершины (найдя первый нуль, мы не ищем второй). Во втором случае с вероятностью $1/2$ нам хватит двух, а с вероятностью $1/2$ понадобится три или четыре. Если же в AND-корне находится 1, то в обеих OR-вершинах первого уровня находится единица, и на каждую из них нужно в среднем не больше $3/2$ просмотров листьев (с вероятностью не менее $1/2$ мы сразу попадаем в лист с единицей и второй лист не смотрим).

Дальнейшее рассуждение легко происходит по индукции. Пусть среднее значение числа запрашиваемых листьев для любого дерева глубины $2k$ не превосходит 3^k. Рассмотрим дерево глубины $2k + 2$ с фиксированными значениями в листьях. Для каждого выбора порядка на первых двух уровнях известно, какие из четырёх вершин высоты 2 будут рассмотрены. По предположению среднее число использованных листьев при рассмотрении каждой вершины высоты 2 (усреднение по всем порядкам обхода) не больше 3^k. Дополнительно усредняем по порядкам на двух первых уровнях и замечаем, что в среднем рассматривается не больше трёх вершин высоты 2. □

11.4.5. Получить более точную оценку для числа просмотренных листьев в предыдущей задаче. [Указание. Используйте разные оценки в зависимости от значения в корне; это позволит заменить $\sqrt{3}$ в оценке на меньшее число $(1+\sqrt{33})/4$.] □

11.5. Ретроспективный анализ

Существенно ли описанное в прошлом разделе улучшение алгоритма (переход от полного перебора к α-β-процедуре)? С одной стороны, да: в нашем примере переход от 4^n к 3^n даёт выигрыш в $(4/3)^n$ раз, а $(4/3)^n$ экспоненциально растёт с ростом n. С другой стороны, экспонента остаётся экспонентой, даже если её показатель уменьшается с 4 до 3, поэтому надежды полностью проанализировать даже не очень сложную и долгую игру таким способом почти нет.

Поэтому на практике обычно выбирают некоторую *оценку позиции* — легко вычислимую функцию, которая по мнению практиков как-то отражает преимущество того или иного игрока (скажем, материальный перевес в шахматах). Затем вместо настоящей игры рассматривают ограниченную игру, в которой делается сравнительно небольшое число k ходов, а затем результатом игры считается оценка полученной позиции, и в этой игре выполняют перебор (применяя α-β-оптимизацию). Конечно, это ничего не гарантирует в настоящей игре, но что поделаешь.

Бывают, однако, и ситуации, когда удаётся определить цену данной позиции точно. Это удаётся сделать для шахматных эндшпилей с небольшим числом фигур — например, можно рассчитать, за какое минимальное число ходов можно поставить мат королём, слоном и конём против одинокого короля в заданных начальных условиях. Заметим, что при этом число ходов может измеряться десятками, а каждый ход имеет десятки вариантов, поэтому о полном переборе (или даже о несколько сокращённом) не может идти и речи.

11.5.1. Придумать другой подход, использующий ограниченность общего числа возможных позиций (скажем, для четырёх упомянутых фигур на шахматной доске это $64^4 = 2^{24} = 16$ «мегапозиций»; с учётом очерёдности хода будет 32 мегапозиции; массив такого размера помещается в память современных компьютеров без труда).

Решение. Заведём массив, отведя ячейку для каждой позиции. Просмотрим его один раз и отметим все матовые позиции (записав туда число 0 в знак того, что позиция проиграна и больше ходить нельзя).

Затем просмотрим массив ещё раз и пометим как выигрышные все позиции, из которых можно пойти в матовые (напишем там 1 в знак того, что можно выиграть за 1 ход). Затем отметим все (ещё не отмеченные) позиции, из которых все ходы ведут в позиции с меткой 1, написав там -2; они проигрываются в два хода. Затем — позиции, из которых есть ход в позиции с меткой -2 (написав там 3), после этого — позиции, из которых все ходы ведут в 1- или 3-позиции (написав там -4) и т. п. Так будем делать до тех пор, пока будут появляться новые пометки. Как только это кончится, для каждой позиции будет известно, можно ли в ней выиграть и сколько ходов для этого нужно. □

Фактически эта процедура повторяет доказательство теоремы Цермело (но дополнительно мы получаем информацию о том, сколько ходов до выигрыша или проигрыша при наилучшей игре).

11.5.2. Могут ли при этом остаться неотмеченные позиции и чему они соответствуют?

Ответ. Это позиции, в которых оба игрока могут гарантировать сколь угодно длинную игру без проигрыша. Впрочем, правило троекратного повторения позиции в шахматах в этом случае позволяет считать партию ничейной. (На самом деле учёт этого правила существенно осложняет ситуацию, поскольку теперь в понятие позиции надо включать информацию о том, какие конфигурации фигур на доске уже встречались.) □

А. Л. Брудно заметил, что есть ситуация, в которой такой «ретроспективный» анализ требует совсем небольших ресурсов и может быть реализован на очень небольшой памяти, хотя для человека соответствующая задача не проста: пусть белые имеют короля на поле c3, которого им запрещено двигать, и ферзя (на каком-то другом поле) и хотят поставить мат одинокому чёрному королю. Ограничение (неподвижность короля), затрудняющее жизнь человеку-шахматисту, облегчает анализ (уменьшая количество позиций почти что в 64 раза за счёт того, что не надо рассматривать разные положения короля!)

Использование таблицы описанного типа можно считать применением метода динамического программирования (мы не вычисляем цену игры для одной и той же позиции снова и снова, обходя дерево, а заполняем таблицу цен систематически).

12. ОПТИМАЛЬНОЕ КОДИРОВАНИЕ

12.1. Коды

Имея 2^n символов, мы можем кодировать каждый из них n битами, поскольку существует 2^n комбинаций из n битов. Например, можно закодировать $4 = 2^2$ символа А, Г, Т, Ц (используемые при записи геномов) двухбитовыми комбинациями 00, 01, 10 и 11. Другой пример: последовательностями из 8 битов (байтами) можно закодировать 256 символов (и этого хватает на латинские и русские буквы, знаки препинания и др.).

Более формально: пусть нам дан *алфавит*, то есть конечное множество, элементы которого называются *символами* или *буквами* этого алфавита. *Кодом* для алфавита A называется функция (таблица) α, которая для каждого символа a из A указывает двоичное слово $\alpha(a)$, называемое *кодовым словом*, или просто *кодом* этого символа. (Двоичное слово — конечная последовательность нулей и единиц.) Не требуется, чтобы коды всех символов имели равные длины.

Мы допускаем, чтобы разные символы имели одинаковые коды. Согласно нашему определению, разрешается все буквы алфавита закодировать словом 0 (и даже пустым словом) — но, конечно, такой код будет бесполезен. Хороший код должен позволять декодирование (восстановление последовательности символов по её коду).

Формально это определяется так. Пусть фиксирован алфавит A и код α для этого алфавита. Для каждого слова P в алфавите A (то есть для любой конечной последовательности букв алфавита A) рассмотрим двоичное слово $\alpha(P)$, которое получается, если записать подряд коды всех букв из P (без каких-либо разделителей). Код α называется *однозначным*, если коды различных слов различны: $\alpha(P) \neq \alpha(P')$ при $P \neq P'$.

12.1.1. Рассмотрим трёхбуквенный алфавит $\{a, b, c\}$ и код $\alpha(a) = 0$, $\alpha(b) = 01$ и $\alpha(c) = 00$. Будет ли этот код однозначным?

Решение. Нет, поскольку слова aa и c кодируются одинаково. □

12.1.2. Для того же алфавита рассмотрим код $\alpha(a) = 0$, $\alpha(b) = 10$ и $\alpha(c) = 11$. Будет ли этот код однозначным?

Решение. Будет. Чтобы доказать это, достаточно объяснить, как можно восстановить слово P по его коду $\alpha(P)$. Если $\alpha(P)$ начинается с нуля, то ясно, что слово P начинается с a. Если $\alpha(P)$ начинается с единицы, то слово P начинается с b или с c — чтобы узнать, с чего именно, достаточно посмотреть на второй бит слова $\alpha(P)$. Восстановив первую букву слова P, мы забываем о ней и о её коде, и продолжаем всё сначала. □

Верно и более общее утверждение. Назовём код *префиксным*, если коды букв не являются началами друг друга (слово $\alpha(p)$ не является началом слова $\alpha(q)$, если буквы p и q различны).

12.1.3. Доказать, что любой префиксный код является однозначным.

Решение. Декодирование можно вести слева направо. Первая буква восстанавливается однозначно: если для двух букв p и q слова $\alpha(p)$ и $\alpha(q)$ являются началами кода, то одно из слов $\alpha(p)$ и $\alpha(q)$ является началом другого, что невозможно для префиксного кода. И так далее. □

12.1.4. Привести пример однозначного кода, не являющегося префиксным.

[Указание. Пусть $\alpha(a) = 0$, $\alpha(b) = 01$, $\alpha(c) = 11$. Этот код является «суффиксным», но не префиксным.] □

12.1.5. Найти таблицу для азбуки Морзе. Объяснить, почему её можно использовать на практике, хотя она не является ни префиксным, ни даже однозначным кодом. □

12.2. Неравенство Крафта – Макмиллана

Зачем вообще нужны коды с разной длиной кодовых слов? Дело в том, что на практике разные символы алфавита встречаются с разной частотой, и выгодно закодировать частые символы короткими словами. (Это соображение, кстати, учитывалось при составлении азбуки Морзе.)

Пусть для каждой буквы a алфавита A фиксирована её *частота* $p(a)$ — положительное число, причём суммы частот всех букв равны

единице. Тогда для любого кода α можно определить *среднюю длину* этого кода как сумму

$$E = \sum p(a)|\alpha(a)|$$

по всем буквам $a \in A$, где $|\alpha(a)|$ — длина кодового слова $\alpha(a)$ буквы a. (Смысл этого определения: если в слове длины N буква a встречается с частотой $p(a)$, то таких букв будет $Np(a)$ и на их кодирование уйдёт $Np(a)|\alpha(a)|$ битов; общая длина кода будет $\sum Np(a)|\alpha(a)|$ и в среднем на кодирование каждой буквы уйдёт E битов.)

Теперь возникает задача: для данных частот построить однозначный код минимальной средней длины. Теоретически это можно сделать перебором (если в коде есть хотя бы одно очень длинное кодовое слово, то его средняя длина велика, поэтому такие коды можно не рассматривать; остаётся конечное число вариантов). Но можно обойтись и без перебора, и в этом разделе мы научимся это делать.

Для начала поймём, что мешает нам выбирать кодовые слова короткими. Оказывается, что есть ровно одно препятствие: длины $n_1, \ldots, n_k$ кодовых слов должны удовлетворять неравенству

$$2^{-n_1} + 2^{-n_2} + \ldots + 2^{-n_k} \leqslant 1,$$

называемому в теории кодирования *неравенством Крафта–Макмиллана*.

12.2.1. Проверить, что оно выполнено для рассмотренных выше примеров однозначных кодов. □

12.2.2. Доказать, что для всякого префиксного кода выполняется неравенство Крафта–Макмиллана.

Решение. Отрезок $[0, 1]$ можно разбить на две половины. Назовём левую I_0, а правую I_1. Каждую из них разобьём пополам: отрезок I_0 разделится на левую половину I_{00} и правую I_{01}, аналогично I_1 делится на I_{10} и I_{11}. И так далее: любому двоичному слову x соответствует отрезок I_x. Длина этого отрезка есть $2^{-|x|}$, где $|x|$ — длина слова x. Если слово x является началом слова y, то отрезок I_x содержит отрезок I_y; если ни одно из слов x и y не является началом другого, то отрезки I_x и I_y не перекрываются (на том знаке, где x и y впервые расходятся, I_x и I_y попадают в разные половины).

Рассмотрим теперь отрезки, соответствующие словам префиксного кода. Они не перекрываются. А значит, сумма их длин не больше единицы, что и даёт неравенство Крафта–Макмиллана. □

12.2.3. Пусть даны k целых положительных чисел $n_1, \ldots, n_k$, удовлетворяющие неравенству Крафта–Макмиллана. Доказать, что можно построить префиксный код для k-буквенного алфавита с длинами кодовых слов $n_1, \ldots, n_k$.

Решение. И здесь полезно использовать соответствие между словами и отрезками и представлять себе дело так: у нас есть единичный отрезок $[0, 1]$, и мы выделяем его части пользователям по требованию. Если пользователь приходит с числом n_i, то это значит, что ему надо выдать в пользование один из отрезков длиной 2^{-n_i}, соответствующих кодовым словам длины n_i. (Тем самым годятся не любые отрезки такой длины, а лишь «правильно расположенные».) Код должен быть префиксным, это значит, что отрезки разных пользователей не должны перекрываться. Нам дано, что суммарная длина всех требований не больше единицы. Как их удовлетворить? Можно отводить место слева направо, при этом рассматривать требования в порядке убывания длин (тогда более короткие отрезки будут правильно расположены после предыдущих более длинных). □

12.2.4. Показать, что выделять кодовые слова (место на отрезке) можно и в порядке поступления требований (как иногда говорят, в «режиме on-line»): пользователь приходит с числом n_i и уходит с правильно расположенным отрезком длины 2^{-n_i}, причём если выполнено неравенство Крафта–Макмиллана, то никто не уйдёт обиженным (всем хватит места, и перераспределять его не придётся).

[Указание. Нужно поддерживать свободное пространство как объединение правильно расположенных отрезков попарно различных длин, выделяя каждому пользователю кусок из кратчайшего подходящего отрезка и доразбивая остаток.] □

12.2.5. Показать, что неравенство Крафта–Макмиллана выполняется не только для любого префиксного кода, но и вообще для любого однозначного кода. (Именно это доказал Макмиллан; Крафт доказал неравенство для префиксных кодов.)

Решение. Есть разные способы решить эту задачу; мы приведём простое и красивое, хотя и несколько загадочное, решение. Пусть имеется однозначный код с k кодовыми словами $P_1, \ldots, P_k$. Нам надо доказать, что их длины $n_i = |P_i|$ удовлетворяют неравенству Крафта–Макмиллана. Представим себе, что вместо нулей и единиц используются символы a и b (какая разница, из чего составлять коды?). Запишем

формально сумму всех кодовых слов как алгебраическое выражение

$$P_1 + P_2 + \ldots + P_k$$

(многочлен от a и b, в котором одночлены записаны как произведения переменных a и b, без возведения в степень). Теперь (ещё более странное на первый взгляд действие) возведём это выражение в степень N (произвольное натуральное число) и раскроем скобки, сохраняя порядок переменных (не собирая вместе одинаковые переменные) в одночленах:

$$(P_1 + P_2 + \ldots + P_k)^N = \text{сумма одночленов.}$$

Например, для кода со словами $0, 10, 11$ (которые теперь записываются как a, ba, bb) и для $N = 2$ получаем

$$(a + ba + bb)^2 = (a + ba + bb)(a + ba + bb) = \\ = aa + aba + abb + baa + baba + babb + bba + bbba + bbbb.$$

В этом примере все одночлены в правой части различны (если не переставлять переменные), и это не случайно: так будет для любого однозначного кода. В самом деле, по определению однозначности никакое слово не может быть получено двумя способами при соединении кодовых слов.

Теперь подставим $a = b = 1/2$ в наше равенство (если оно верно для букв, то оно верно и для любых их числовых значений). Слева получится

$$(2^{-n_1} + 2^{-n_2} + \ldots + 2^{-n_k})^N$$

(в скобке как раз выражение из неравенства Крафта–Макмиллана). Правую часть мы оценим сверху, сгруппировав слова по длинам: имеется не более 2^l слагаемых длины l, каждое из которых равно 2^{-l}, и потому слагаемые данной длины в сумме не превосходят единицы, а правая часть не превосходит максимальной длины слагаемых, то есть $N \max n_i$. Итак, получаем, что

$$(2^{-n_1} + 2^{-n_2} + \ldots + 2^{-n_k})^N < N \max n_i,$$

и это верно при любом N. Если основание степени в левой части больше единицы, то при больши́х N это неравенство нарушится (показательная функция растёт быстрее линейной). Поэтому для однозначного кода выполняется неравенство Крафта–Макмиллана. □

12.3. Код Хаффмена

Теперь задача о коде минимальной средней длины приобретает такую форму: для данных положительных $p_1, \ldots, p_k$, равных в сумме единице, найти целые положительные $n_1, \ldots, n_k$, для которых выполнено неравенство Крафта–Макмиллана, а сумма

$$\sum_{i=1}^{k} p_i n_i$$

является минимально возможной (среди наборов $n_1, \ldots, n_k$, удовлетворяющих неравенству). Задача 12.2.5 показывает показывает, что средняя длина однозначного кода не меньше этого минимума, а задача 12.2.3 говорит, что этот минимум достигается, причём даже для префиксного кода. Как же найти числа $n_1, \ldots, n_k$, доставляющие этот минимум?

12.3.1. Доказать, что для двух букв оптимальный код состоит из двух слов длины 1, независимо от частот букв. □

Чтобы решить задачу в общем случае, начнём с нескольких простых замечаний.

12.3.2. Пусть частоты расположены в убывающем порядке: $p_1 > > p_2 > \ldots > p_k$. Доказать, что тогда длины слов оптимального кода идут в неубывающем порядке: $n_1 \leqslant n_2 \leqslant \ldots \leqslant n_k$.

Решение. Если бы более редкая буква имела бы более короткое кодовое слово, то, обменяв кодовые слова, мы сократили бы среднюю длину кода. □

12.3.3. Останется ли утверждение предыдущей задачи в силе, если частоты расположены в невозрастающем порядке (возможны равные)?

Решение. Нет: если, скажем, имеются три буквы с частотой $1/3$, то оптимальный код будет иметь длины слов $1, 2, 2$ (если бы два кодовых слова имели длину 1, то на третье уже не осталось бы места), и они могут идти в любом порядке. □

Заметим, однако, что при поиске оптимального кода (для невозрастающих частот) мы вправе ограничиваться лишь кодами, в которых длины кодовых слов неубывают (поскольку кодовые слова для букв одинаковых частот можно переставлять без изменения средней длины кода).

12.3.4. Пусть частоты расположены в невозрастающем порядке ($p_1 \geqslant p_2 \geqslant \ldots \geqslant p_k$), а длины слов в оптимальном коде расположены в неубывающем порядке $n_1 \leqslant n_2 \leqslant \ldots \leqslant n_k$. Доказать, что $n_{k-1} = n_k$ (при $k \geqslant 2$).

Решение. Предположим, что это не так, и что есть единственное самое длинное кодовое слово длины n_k. Тогда неравенство Крафта–Макмиллана не может обращаться в равенство, поскольку все слагаемые, кроме наименьшего (последнего), кратны *удвоенному* последнему слагаемому. Значит, в этом неравенстве есть запас, причём не меньший последнего слагаемого. А тогда можно уменьшить n_k на единицу, не нарушая неравенства, что противоречит предположению об оптимальности исходного кода. □

Эта задача показывает, что при поиске оптимального кода можно рассматривать лишь коды, в которых две самые редкие буквы имеют коды одинаковой длины.

12.3.5. Как свести задачу отыскания длин кодовых слов оптимального кода для k частот

$$p_1 \geqslant p_2 \geqslant \ldots \geqslant p_{k-2} \geqslant p_{k-1} \geqslant p_k$$

к задаче поиска длин оптимального кода для $k-1$ частот

$$p_1, p_2, \ldots, p_{k-2}, p_{k-1} + p_k$$

(частоты двух самых редких букв объединены)?

Решение. Мы уже знаем, что можно рассматривать лишь коды с $n_{k-1} = n_k$. Неравенство Крафта–Макмиллана тогда запишется как

$$2^{-n_1} + 2^{-n_2} + \ldots + 2^{-n_{k-2}} + 2^{-n_{k-1}} + 2^{-n_k} = \\ = 2^{-n_1} + 2^{-n_2} + \ldots + 2^{-n_{k-2}} + 2^{-n} \leqslant 1,$$

если положить $n_{k-1} = n_k = n + 1$. Таким образом, числа $n_1, \ldots, n_{k-2}, n$ должны удовлетворять неравенству Крафта–Макмиллана для $k-1$ букв. Средняя длины этих двух кодов будут связаны:

$$p_1 n_1 + \ldots + p_{k-2} n_{k-2} + p_{k-1} n_{k-1} + p_k n_k = \\ = p_1 n_1 + \ldots + p_{k-2} n_{k-2} + (p_{k-1} + p_k) n + [p_{k-1} + p_k].$$

Последнее слагаемое (квадратная скобка) не зависит от выбираемого кода, поэтому минимизировать надо остальное, то есть как раз среднюю длину кода с длинами слов $n_1, \ldots, n_{k-2}, n$ для частот $p_1, p_2, \ldots$ $\ldots, p_{k-2}, p_{k-1} + p_k$. После этого надо положить $n_{k-1} = n_k = n + 1$, и это даст оптимальный код для исходной задачи. □

Используя эту задачу, несложно составить рекурсивную программу для отыскания длин кодовых слов. С каждым вызовом число букв будет уменьшаться, пока мы не сведём задачу к случаю двух букв, когда оптимальный код состоит из слов 0 и 1. Затем можно найти и сами кодовые слова (согласно задаче 12.2.3). Но проще объединить эти действия и сразу искать кодовые слова: ведь замена числа n на два числа $n+1$ соответствует замене кодового слова P на два слова $P0$ и $P1$ на единицу большей длины (и эта последняя замена сохраняет префиксность кода).

Код, построенный таким методом, называется *кодом Хаффмена*. Мы доказали, что он имеет минимальную среднюю длину среди всех кодов (для данных частот букв). В следующей задаче оценивается число операций, необходимых для построения кода Хаффмена.

12.3.6. Показать, что можно обработать частоты $p_1, \ldots, p_k$, сделав $O(k \log k)$ операций, после чего i-ое кодовое слово можно указать за время, пропорциональное его длине.

[Указание. Заметим, что оценка времени довольно сильная: только на сортировку чисел p_i уже уходит $O(k \log k)$ действий. Поэтому, применяя предыдущую задачу, нужно использовать результаты сортировки k чисел при сортировке меньшего количества чисел. Это можно сделать с помощью очереди с приоритетами, вынимая два минимальных числа и добавляя их сумму за $O(\log k)$ действий. Это позволяет определить, какие две буквы надо соединять в одну на каждом шаге. Параллельно с соединением букв можно строить дерево кодов, проводя рёбра (помеченные 0 и 1) от соединённой буквы к каждой из её половинок. При этом требуется $O(1)$ действий на каждом шаге. После завершения построение прослеживать код любой буквы можно символ за символом.] □

12.4. Код Шеннона – Фано

Мы видели, как можно построить оптимальный код (имеющий минимальную среднюю длину) для данного набора частот. Однако эта конструкция не даёт никакой оценки для средней длины оптимального кода (как функции от частот p_i). Следующие задачи указывает такую оценку (с абсолютной погрешностью не более 1).

12.4.1. Показать, что для любых положительных частот $p_1, \ldots, p_k$ (в сумме равных единице) существует код средней длиной не более

$H(p_1, \ldots, p_k) + 1$, где функция H (называемая *энтропией Шеннона*) определяется формулой

$$H(p_1, \ldots, p_n) = p_1(-\log_2 p_1) + \ldots + p_k(-\log_2 p_k)$$

Решение. Если частоты p_i представляют собой целые (отрицательные) степени двойки, то это утверждение почти очевидно. Положим $n_i = -\log p_i$ (здесь и далее все логарифмы двоичные). Тогда $2^{-n_i} = p_i$ и потому для чисел n_i выполнено неравенство Крафта–Макмиллана. По задаче 12.2.3 можно построить префиксный код с длинами кодовых слов $n_1, \ldots, n_k$, и средняя длина этого кода будет равна $H(p_1, \ldots, p_k)$ (и даже единицу добавлять не надо).

Эта единица пригодится, если $\log p_i$ не целые. В этом случае надо взять наименьшее n_i, при котором $2^{-n_i} \leqslant p_i$. Для таких n_i выполняется неравенство Крафта–Макмиллана, и они больше $-\log p_i$ не более чем на единицу (потому и после усреднения ухудшение будет не более чем на единицу). □

Построенный на основе этой задачи код называется *кодом Шеннона–Фано*. Это построение легко извлекается из решения задачи 12.2.3: рассматривая числа $n_i = -\lfloor \log p_i \rfloor$ (наименьшие целые числа, для которых $2^{-n_i} \leqslant p_i$) в порядке убывания, мы отводим для каждого из них кодовое слово и соответствующий участок отрезка $[0, 1]$ слева направо.

При этом мы проигрываем в длине кода (по сравнению с оптимальным кодом) не более единицы: как мы сейчас увидим, средняя длина любого (в том числе и оптимального) кода не меньше $H(p_1, \ldots, p_k)$.

12.4.2. (Для знакомых с математическим анализом) Доказать, что (при данных положительных частотах, в сумме дающих единицу) средняя длина любого (однозначного) кода не меньше $H(p_1, \ldots, p_k)$.

Решение. Имея в виду неравенство Крафта–Макмиллана, мы должны доказать такой факт: если

$$2^{-n_1} + \ldots 2^{-n_k} \leqslant 1,$$

то

$$p_1 n_1 + \ldots + p_k n_k \geqslant H(p_1, \ldots, p_k).$$

Это верно для любых n_i, не обязательно целых. Удобно перейти от n_i к величинам $q_i = 2^{-n_i}$; интересующее нас утверждение тогда гласит, что если $p_1, \ldots, p_k$ и $q_1, \ldots, q_k$ — два набора положительных чисел,

и сумма чисел в каждом равна единице, то

$$p_1(-\log q_1) + \ldots + p_k(-\log q_k) \geqslant p_1(-\log p_1) + \ldots + p_k(-\log p_k).$$

Другими словами, выражение

$$p_1(-\log q_1) + \ldots + p_k(-\log q_k)$$

(рассматриваемое при фиксированных p_i как функция на множестве всех положительных $q_1, \ldots, q_k$, в сумме равных единице) достигает минимума при $q_i = p_i$. Область определения этой функции есть внутренность симплекса (треугольника при $n = 3$, тетраэдра при $n = 4$ и т. д.) и при приближении к границе одно из q_i становится малым, а его минус логарифм уходит в бесконечность. Значит, минимум функции достигается внутри области. В точке минимума градиент $(-p_1/q_1, \ldots, -p_n/q_n)$ должен быть перпендикулярен плоскости, на которой функция определена (иначе сдвиг вдоль этой плоскости уменьшал бы функцию), то есть все p_i/q_i равны. Поскольку $\sum p_i = \sum q_i = 1$, то это означает, что $p_i = q_i$.

Другое объяснение: функция log выпукла вверх, поэтому для любых неотрицательных коэффициентов α_i, в сумме равных единице, и для любых точек x_i из области определения логарифма выполняется неравенство

$$\log\left(\sum \alpha_i x_i\right) \geqslant \sum \alpha_i \log x_i.$$

Остаётся положить $\alpha_i = p_i$, $x_i = q_i/p_i$; в левой части будет логарифм единицы, то есть нуль, а $\sum p_i \log(q_i/p_i)$ есть как раз разность между левой и правой частями доказываемого неравенства. □

Велика ли экономия от использования кодов, описанных в этом разделе? Это, конечно, зависит от частот букв: если они все одинаковые, то никакой экономии не будет. Легко заметить, что в русском языке разные буквы имеют разную частоту. Если, скажем, в текстах (TEX-файлах) этой книжки (на момент эксперимента) оставить только 33 строчные русские буквы от «а» до «я», а все остальные символы не учитывать, то самой частой буквой будет буква «о» (частота 0,105), а самой редкой — твёрдый знак (частота 0,00019). Значение энтропии Шеннона при этом будет равно 4,454 (сравните с 5 битами, необходимыми для кодирования 32 букв). Выигрыш не так велик. Он будет больше, если учитывать также и другие символы (прописные буквы, знаки препинания и др.), которые встречаются в тексте гораздо реже. Наконец, можно кодировать не буквы, а двухбуквенные комбинации или

ещё что-нибудь. Именно так поступают популярные программы сжатия информации (типа zip), которые позволяют сократить многие тексты в полтора-два раза (а некоторые другие файлы данных — и в большее число раз).

12.4.3. Компания М. утверждает, что её новая программа суперсжатия файлов позволяет сжать любой файл длиной больше 100 000 байтов по крайней мере на 10% без потери информации (можно восстановить исходный файл по его сжатому варианту). Доказать, что она врёт. □

13. ПРЕДСТАВЛЕНИЕ МНОЖЕСТВ. ХЕШИРОВАНИЕ

13.1. Хеширование с открытой адресацией

В главе 6 было указано несколько представлений для множеств, элементами которых являются целые числа произвольной величины. Однако в любом из них хотя бы одна из операций проверки принадлежности, добавления и удаления элемента требовала количества действий, пропорционального числу элементов множества. На практике это бывает слишком много. Существуют способы, позволяющие получить для всех трёх упомянутых операций оценку $C \log n$. Один из таких способов мы рассмотрим в следующей главе. В этой главе мы разберём способ, которые хотя и приводит к Cn действиям в худшем случае, но зато «в среднем» требует значительно меньшего их числа. (Мы не будем уточнять слов «в среднем», хотя это и можно сделать.) Этот способ называется *хешированием*.

Пусть нам необходимо представлять множества элементов типа `T`, причём число элементов в них заведомо меньше `n`. Выберем некоторую функцию `h`, определённую на значениях типа `T` и принимающую значения `0...n-1`. Было бы хорошо, чтобы эта функция принимала на элементах будущего множества по возможности более разнообразные значения. (Худший случай — это когда её значения на всех элементах хранимого множества одинаковы.) Эту функцию будем называть *хеш-функцией*, или, как ещё говорят, *функцией расстановки*.

Введём два массива

```
val:  array [0..n-1] of T;
used: array [0..n-1] of boolean;
```

(мы позволяем себе писать `n-1` в качестве границы в определении типа,

хотя в паскале это не разрешается). В этих массивах будут храниться элементы множества: оно равно множеству всех `val[i]` для тех `i`, для которых `used[i]`, причём все эти `val[i]` различны. По возможности мы будем хранить элемент `t` на месте `h(t)`, считая это место «исконным» для элемента `t`. Однако может случиться так, что новый элемент, который мы хотим добавить, претендует на уже занятое место (для которого `used` истинно). В этом случае мы отыщем ближайшее справа свободное место и запишем элемент туда. («Справа» значит «в сторону увеличения индексов»; дойдя до края, мы перескакиваем в начало.) По предположению, число элементов всегда меньше `n`, так что пустые места заведомо будут.

Формально говоря, в любой момент должно соблюдаться такое требование: для любого элемента множества участок справа от его исконного места до его фактического места полностью заполнен.

Благодаря этому проверка принадлежности заданного элемента `t` осуществляется легко: встав на `h(t)`, двигаемся направо, пока не дойдём до пустого места или до элемента `t`. В первом случае элемент `t` отсутствует в множестве, во втором — присутствует. Если элемент отсутствует, то его можно добавить на найденное пустое место. Если присутствует, то можно его удалить (положив `used = false`).

13.1.1. В предыдущем абзаце есть ошибка. Найти её и исправить.

Решение. Дело в том, что при удалении требуемое свойство «отсутствия пустот» может нарушиться. Поэтому будем делать так. Создав дыру, будем двигаться направо, пока не натолкнёмся на элемент, стоящий не на исконном месте, или на ещё одно пустое место. Во втором случае на этом можно успокоиться. В первом случае посмотрим, не нужно ли найденный элемент поставить на место дыры. Если нет, то продолжаем поиск, если да, то затыкаем им старую дыру. При этом образуется новая дыра, с которой делаем всё то же самое. □

13.1.2. Написать программы проверки принадлежности, добавления и удаления.

Решение.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
```

```
| принадлежит := used [i] and (val [i] = t);
end;

procedure добавить (t: T);
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if not used [i] then begin
| | used [i] := true;
| | val [i] := t;
| end;
end;

procedure исключить (t: T);
| var i, gap: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if used [i] and (val [i] = t) then begin
| | used [i] := false;
| | gap := i;
| | i := (i + 1) mod n;
| | {gap - дыра, которая может закрыться одним из i,i+1,...}
| | while used [i] do begin
| | | if i = h (val[i]) then begin
| | | | {на своём месте, ничего не делать}
| | | end else if dist(h(val[i]),i) < dist(gap,i) then begin
| | | | {gap...h(val[i])...i, ничего не делать}
| | | end else begin
| | | | used [gap] := true;
| | | | val [gap] := val [i];
| | | | used [i] := false;
| | | | gap := i;
| | | end;
| | | i := (i + 1) mod n;
| | end;
| end;
end;
```

Здесь `dist(a,b)` — измеренное по часовой стрелке (слева направо)

расстояние от `a` до `b`, то есть

$$\texttt{dist(a,b)} = \texttt{(b-a+n) mod n}.$$

(Мы прибавили `n`, так как функция `mod` правильно работает при положительном делимом.) □

13.1.3. Существует много вариантов хеширования. Один из них таков: обнаружив, что исконное место (обозначим его i) занято, будем искать свободное не среди $i+1, i+2, \ldots$, а среди $r(i), r(r(i)), r(r(r(i))), \ldots$, где r — некоторое отображение $\{0, \ldots, n-1\}$ в себя. Какие при этом будут трудности?

Ответ. (1) Не гарантируется, что если пустые места есть, то мы их найдём. (2) При удалении неясно, как заполнять дыры. (На практике во многих случаях удаление не нужно, так что такой способ также применяется. Считается, что удачный подбор функции r может предотвратить образование «скоплений» занятых ячеек.) □

13.1.4. Пусть для хранения множества всех правильных русских слов в программе проверки орфографии используется хеширование. Что нужно добавить, чтобы к тому же уметь находить английский перевод любого правильного слова?

Решение. Помимо массива `val`, элементы которого являются русскими словами, нужен параллельный массив их английских переводов. □

13.2. Хеширование со списками

На хеш-функцию с k значениями можно смотреть как на способ свести вопрос о хранении одного большого множества к вопросу о хранении нескольких меньших. Именно, если у нас есть хеш-функция с k значениями, то любое множество разбивается на k подмножеств (возможно, пустых), соответствующих возможным значениям хеш-функции. Вопрос о проверке принадлежности, добавлении или удалении для большого множества сводится к такому же вопросу для одного из меньших (чтобы узнать, для какого, надо посмотреть на значение хеш-функции).

Эти меньшие множества удобно хранить с помощью ссылок; их суммарный размер равен числу элементов хешируемого множества. Следующая задача предлагает реализовать этот план.

13.2.1. Пусть хеш-функция принимает значения `1...k`. Для каждого значения хеш-функции рассмотрим список всех элементов множе-

ства с данным значением хеш-функции. Будем хранить эти k списков с помощью переменных

```
Содержание: array [1..n] of T;
Следующий: array [1..n] of 1..n;
ПервСвоб: 1..n;
Вершина: array [1..k] of 1..n;
```

так же, как мы это делали для k стеков ограниченной суммарной длины. Написать соответствующие программы. (Удаление по сравнению с открытой адресацией упрощается.)

Решение. Перед началом работы надо положить Вершина[i] = 0 для всех i = 1 . . . k, и связать все места в список свободного пространства, положив ПервСвоб = 1 и Следующий[i] = i+1 для i = 1 . . . n-1, а также Следующий[n] = 0.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := Вершина[h(t)];
| {осталось искать в списке, начиная с i}
| while (i <> 0) and (Содержание[i] <> t) do begin
| | i := Следующий[i];
| end; {(i=0) or (Содержание [i] = t)}
| принадлежит := (i<>0) and (Содержание[i]=t);
end;

procedure добавить (t: T);
| var i: integer;
begin
| if not принадлежит(t) then begin
| | i := ПервСвоб;
| | {ПервСвоб <> 0 - считаем, что не переполняется}
| | ПервСвоб := Следующий[ПервСвоб]
| | Содержание[i]:=t;
| | Следующий[i]:=Вершина[h(t)];
| | Вершина[h(t)]:=i;
| end;
end;

procedure исключить (t: T);
| var i, pred: integer;
begin
| i := Вершина[h(t)]; pred := 0;
```

```
  {осталось искать в списке, начиная с i;  pred -
     предыдущий, если он есть, и 0, если нет}
  while (i <> 0) and (Содержание[i] <> t) do begin
  | pred := i; i := Следующий[i];
  end; {(i=0) or (Содержание [i] = t)}
  if i <> 0 then begin
  | {Содержание[i]=t, элемент есть, надо удалить}
  | if pred = 0 then begin
  | | {элемент оказался первым в списке}
  | | Вершина[h(t)] := Следующий[i];
  | end else begin
  | | Следующий[pred] := Следующий[i]
  | end;
  | {осталось вернуть i в список свободных}
  | Следующий[i] := ПервСвоб;
  | ПервСвоб:=i;
  end;
end;
```

□

13.2.2. (Для знакомых с теорией вероятностей.) Пусть хеш-функция с k значениями используется для хранения множества, в котором в данный момент n элементов. Доказать, что математическое ожидание числа действий в предыдущей задаче не превосходит $C(1 + n/k)$, если добавляемый (удаляемый, искомый) элемент t выбран случайно, причём все значения $h(t)$ имеют равные вероятности (равные $1/k$).

Решение. Если $l(i)$ — длина списка, соответствующего хеш-значению i, то число операций не превосходит $C(1 + l(h(t)))$; усредняя, получаем искомый ответ, так как $\sum_i l(i) = n$. □

Эта оценка основана на предположении о равных вероятностях. Однако в конкретной ситуации всё может быть совсем не так, и значения хеш-функции могут «скучиваться»: для каждой конкретной хеш-функции есть «неудачные» ситуации, когда число действий оказывается большим. Приём, называемый *универсальным хешированием*, позволяет обойти эту проблему. Идея состоит в том, что берётся семейство хеш-функций, причём любая ситуация оказывается неудачной лишь для небольшой части этого семейства.

Пусть H — семейство функций, каждая из которых отображает множество T в множество из k элементов (например, $0 \dots k-1$). Говорят, что H — *универсальное семейство хеш-функций*, если для любых двух различных значений s и t из множества T вероятность события $h(s) = h(t)$ для случайной функции h из семейства H равна $1/k$. (Дру-

гими словами, те функции из H, для которых $h(s) = h(t)$, составляют $1/k$-ую часть всех функций в H.)

Замечание. Более сильное требование к семейству H могло бы состоять в том, чтобы для любых двух различных элементов s и t множества T значения $h(s)$ и $h(t)$ случайной функции h являются независимыми случайными величинами, равномерно распределёнными на $0 \ldots k-1$.

13.2.3. Пусть $t_1, \ldots, t_n$ — произвольная последовательность различных элементов множества T. Рассмотрим количество действий, происходящих при помещении элементов $t_1, \ldots, t_n$ в множество, хешируемое с помощью функции h из универсального семейства H. Доказать, что среднее количество действий (усреднение — по всем h из H) не превосходит $Cn(1 + n/k)$.

Решение. Обозначим через m_i количество элементов последовательности, для которых хеш-функция равна i. (Числа $m_0, \ldots, m_{k-1}$ зависят, конечно, от выбора хеш-функции.) Количество действий, которое мы хотим оценить, с точностью до постоянного множителя равно $m_0^2 + m_1^2 + \ldots + m_{k-1}^2$. (Если s чисел попадают в одну хеш-ячейку, то для этого требуется примерно $1 + 2 + \ldots + s$ действий.) Эту же сумму квадратов можно записать как число пар $\langle p, q \rangle$, для которых $h(t_p) = h(t_q)$. Последнее равенство, если его рассматривать как событие при фиксированных p и q, имеет вероятность $1/k$ при $p \neq q$, поэтому среднее значение соответствующего члена суммы равно $1/k$, а для всей суммы получаем оценку порядка n^2/k, а точнее $n + n^2/k$, если учесть члены с $p = q$. □

Эта задача показывает, что на каждый добавляемый элемент приходится в среднем $C(1 + n/k)$ операций. В этой оценке дробь n/k имеет смысл «коэффициента заполнения» хеш-таблицы.

13.2.4. Доказать аналогичное утверждение для произвольной последовательности операций добавления, поиска и удаления (а не только для добавления, как в предыдущей задаче).

[Указание. Будем представлять себе, что в ходе поиска, добавления и удаления элемент проталкивается по списку своих коллег с тем же хеш-значением, пока не найдёт своего двойника или не дойдёт до конца списка. Будем называть i-j-столкновением столкновение t_i с t_j. (Оно либо произойдёт, либо нет — в зависимости от h.) Общее число действий примерно равно числу всех происшедших столкновений плюс число элементов. При $t_i \neq t_j$ вероятность i-j-столкновения не превосходит $1/k$. Осталось проследить за столкновениями между равными

элементами. Фиксируем некоторое значение x из множества T и посмотрим на связанные с ним операции. Они идут по циклу: добавление — проверки — удаление — добавление — проверки — удаление — ... Столкновения происходят между добавляемым элементом и следующими за ним проверками (до удаления включительно), поэтому общее их число не превосходит числа элементов, равных x.] □

Теперь приведём примеры универсальных семейств. Очевидно, для любых конечных множеств A и B семейство всех функций, отображающих A в B, является универсальным. Однако этот пример с практической точки зрения бесполезен: для запоминания случайной функции из этого семейства нужен массив, число элементов в котором равно числу элементов в множестве A. (А если мы можем себе позволить такой массив, то никакого хеширования не требуется!)

Более практичные примеры универсальных семейств могут быть построены с помощью несложных алгебраических конструкций. Через $\mathbb{Z}_p$ мы обозначаем множество вычетов по простому модулю p, т. е. $\{0, 1, \ldots, p-1\}$; арифметические операции в этом множестве выполняются по модулю p. Универсальное семейство образуют все линейные функционалы на $\mathbb{Z}_p^n$ со значениями в $\mathbb{Z}_p$. Более подробно, пусть $a_1, \ldots, a_n$ — произвольные элементы $\mathbb{Z}_p$; рассмотрим отображение

$$h : \langle x_1 \ldots x_n \rangle \mapsto a_1 x_1 + \ldots + a_n x_n.$$

Мы получаем семейство из p^n отображений $\mathbb{Z}_p^n \to \mathbb{Z}_p$ параметризованное наборами $\langle a_1 \ldots a_n \rangle$.

13.2.5. Доказать, что это семейство является универсальным.

[Указание. Пусть x и y — различные точки пространства $\mathbb{Z}_p^n$. Какова вероятность того, что случайный функционал принимает на них одинаковые значения? Другими словами, какова вероятность того, что он равен нулю на их разности $x - y$? Ответ даётся таким утверждением: пусть u — ненулевой вектор; тогда все значения случайного функционала на нём равновероятны.] □

В следующей задаче множество $\mathbb{B} = \{0, 1\}$ рассматривается как множество вычетов по модулю 2.

13.2.6. Семейство всех линейных отображений из $\mathbb{B}^n$ в $\mathbb{B}^m$ является универсальным. □

Родственные хешированию идеи неожиданно оказываются полезными в следующей ситуации (рассказал Д. Варсанофьев). Пусть мы хо-

тим написать программу, которая обнаруживала (большинство) опечаток в тексте, но не хотим хранить список всех правильных словоформ. Предлагается поступить так: выбрать некоторое N и набор функций $f_1, \ldots, f_k$, отображающих русские слова в $1, \ldots, N$. В массиве из N битов положим все биты равными нулю, кроме тех, которые являются значением какой-то функции набора на какой-то правильной словоформе. Теперь приближённый тест на правильность словоформы таков: проверить, что значения всех функций набора на этой словоформе попадают на места, занятые единицами. (Этот тест может не заметить некоторых ошибок, но все правильные словоформы будут одобрены.)

14. ПРЕДСТАВЛЕНИЕ МНОЖЕСТВ. ДЕРЕВЬЯ. СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

14.1. Представление множеств с помощью деревьев

Полное двоичное дерево. T-деревья

Нарисуем точку. Из неё проведём две стрелки (влево вверх и вправо вверх) в две другие точки. Из каждой из этих точек проведём по две стрелки и так далее. Полученную картинку (в n-ом слое будет 2^{n-1} точек) называют полным двоичным *деревом*. Нижнюю точку называют *корнем*. У каждой вершины есть два *сына* (две вершины, в которые идут стрелки) — *левый* и *правый*. У всякой вершины, кроме корня, есть единственный *отец*.

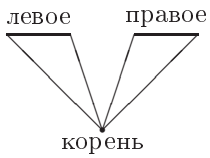
Пусть выбрано некоторое конечное множество вершин полного двоичного дерева, содержащее вместе с каждой вершиной и всех её предков. Пусть на каждой вершине этого множества написано значение фиксированного типа T (то есть задано отображение множества вершин в множество значений типа T). То, что получится, будем называть *T-деревом*. Множество всех T-деревьев обозначим $\mathrm{Tree}(T)$.

Рекурсивное определение. Всякое непустое T-дерево разбивается на три части: корень (несущий пометку из T), левое и правое поддеревья (которые могут быть пустыми). Это разбиение устанавливает взаимно однозначное соответствие между множеством непустых T-деревьев и произведением $T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T)$. Обозначив через empty пустое дерево, можно написать

$$\mathrm{Tree}(T) = \{\mathrm{empty}\} + T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T).$$

Поддеревья. Высота

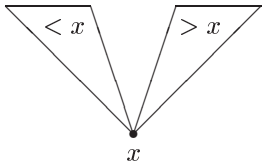
Фиксируем некоторое T-дерево. Для каждой его вершины x определено её *левое поддерево* (левый сын вершины x и все его потомки), *правое поддерево* (правый сын вершины x и все его потомки) и *поддерево с корнем в x* (вершина x и все её потомки).



Левое и правое поддеревья вершины x могут быть пустыми, а поддерево с корнем в x всегда непусто (содержит по крайней мере x). *Высотой* поддерева будем считать максимальную длину цепи $y_1 \dots y_n$ его вершин, в которой y_{i+1} — сын y_i для всех i. (Высота дерева из одного корня равна единице, высота пустого дерева — нулю.)

Упорядоченные T-деревья

Пусть на множестве значений типа T фиксирован порядок. Назовём T-дерево *упорядоченным*, если выполнено такое свойство: для любой вершины x все пометки в её левом поддереве меньше пометки в x, а все пометки в её правом поддереве больше пометки в x.



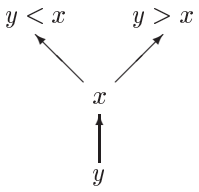
14.1.1. Доказать, что в упорядоченном дереве все пометки различны.

[Указание. Индукция по высоте дерева.] □

Представление множеств с помощью деревьев

Каждое дерево будем считать представлением множества всех пометок на его вершинах. При этом одно и то же множество может иметь различные представления.

Если дерево упорядочено, то каждый элемент может легко «найти своё место» в дереве: придя в какую-то вершину и сравнив себя с тем, кто там находится, элемент решает, идти ему налево или направо.



Начав с корня и двигаясь по этому правилу, он либо обнаружит, что такой элемент уже есть, либо найдёт место, в котором он должен быть.

Всюду далее мы предполагаем, что на значениях типа T задан порядок, и рассматриваем только упорядоченные деревья.

Хранение деревьев в программе

Можно было бы сопоставить вершины полного двоичного дерева с числами $1, 2, 3, \ldots$ (считая, что левый сын n есть $2n$, правый сын n есть $2n + 1$) и хранить пометки в массиве `val [1...]`. Однако этот способ неэкономен, поскольку тратится место на хранение пустых вакансий в полном двоичном дереве.

Более экономен такой способ. Введём три массива

```
val: array [1..n] of T;
left, right: array [1..n] of 0..n;
```

(`n` — максимальное возможное число вершин дерева) и переменную `root:0..n`. Каждая вершина хранимого T-дерева будет иметь номер — число от `1` до `n`. Разные вершины будут иметь разные номера. Пометка в вершине с номером `x` равна `val[x]`. Корень имеет номер `root`. Если вершина с номером `i` имеет сыновей, то их номера равны `left[i]` и `right[i]`. Отсутствующим сыновьям соответствует число `0`. Аналогичным образом значение `root=0` соответствует пустому дереву.

Для хранения дерева используется лишь часть массива; для тех `i`, которые свободны (не являются номерами вершин), значения `val[i]` безразличны. Нам будет удобно, чтобы все свободные числа были «связаны в список»: первое хранится в специальной переменной `free:0..n`, а следующее за `i` свободное число хранится в `left[i]`, так что свободны числа

```
free, left[free], left[left[free]],...
```

Для последнего свободного числа i значение left[i] равно 0. Равенство free=0 означает, что свободных чисел больше нет.

Замечание. Мы использовали для связывания свободных вершин массив left, но, конечно, с тем же успехом можно было использовать массив right.

Вместо значения 0 (обозначающего отсутствие вершины) можно было бы воспользоваться любым другим числом вне 1..n. Чтобы подчеркнуть это, будем вместо 0 использовать константу null=0.

14.1.2. Составить программу, определяющую, содержится ли элемент t:T в упорядоченном дереве (хранимом так, как только что описано).

Решение.

```
if root = null then begin
| ..не принадлежит
end else begin
| x := root;
| {инвариант: остаётся проверить наличие t в непустом
|   поддереве с корнем x}
| while ((t < val [x]) and (left [x] <> null)) or
| |     ((t > val [x]) and (right [x] <> null)) do begin
| | if t < val [x] then begin {left [x] <> null}
| | | x := left [x];
| | end else begin {t > val [x], right [x] <> null}
| | | x := right [x];
| | end;
| end;
| {либо t = val [x], либо t отсутствует в дереве}
| ..ответ = (t = val [x])
end;
```

□

14.1.3. Упростить решение, используя следующий трюк. Расширим область определения массива val, добавив ячейку с номером null и положим val[null]=t.

Решение.

```
val [null] := t;
x := root;
while t <> val [x] do begin
| if t < val [x] then begin
| | x := left [x];
| end else begin
```

```
  | x := right [x];
  end;
end;
..ответ: (x <> null).
```
□

14.1.4. Составить программу добавления элемента t в множество, представленное упорядоченным деревом (если элемент t уже есть, ничего делать не надо).

Решение. Определим процедуру get_free (var i:integer), дающую свободное (не являющееся номером) число i и соответствующим образом корректирующую список свободных чисел.

```
procedure get_free (var i: integer);
begin
  {free <> null}
  i := free;
  free := left [free];
end;
```

С её использованием программа приобретает такой вид:

```
if root = null then begin
  get_free (root);
  left [root] := null; right [root] := null;
  val [root] := t;
end else begin
  x := root;
  {инвариант: осталось добавить t к непустому поддереву с
   корнем в x}
  while ((t < val [x]) and (left [x] <> null)) or
        ((t > val [x]) and (right [x] <> null)) do begin
    if t < val [x] then begin
      x := left [x];
    end else begin {t > val [x]}
      x := right [x];
    end;
  end;
  if t <> val [x] then begin {t нет в дереве}
    get_free (i);
    left [i] := null; right [i] := null;
    val [i] := t;
    if t < val [x] then begin
      left [x] := i;
    end else begin {t > val [x]}
```

```
|  |  | right [x] := i;
|  | end;
| end;
end;
```
□

14.1.5. Составить программу удаления элемента t из множества, представленного упорядоченным деревом (если его там нет, ничего делать не надо).

Решение.

```
if root = null then begin
| {дерево пусто, ничего делать не надо}
end else begin
| x := root;
| {осталось удалить t из поддерева с корнем в x; поскольку
|  это может потребовать изменений в отце x, введём
|  переменные  father: 1..n и direction: (l, r);
|  поддерживаем такой инвариант: если x не корень, то father
|  - его отец, а direction равно l или r в зависимости от
|  того, левым или правым сыном является x}
| while ((t < val [x]) and (left [x] <> null)) or
|       ((t > val [x]) and (right [x] <> null)) do begin
|  if t < val [x] then begin
|  |  father := x; direction := l;
|  |  x := left [x];
|  end else begin {t > val [x]}
|  |  father := x; direction := r;
|  |  x := right [x];
|  end;
| end;
| {t = val [x] или t нет в дереве}
| if t = val [x] then begin
| | ..удаление вершины x с отцом father и
| |   направлением direction
| end;
end;
```

Удаление вершины использует процедуру

```
procedure make_free (i: integer);
begin
| left [i] := free;
| free := i;
end;
```

Она включает число i в список свободных. При удалении различаются 4 случая в зависимости от наличия или отсутствия сыновей у удаляемой вершины.

```
if (left [x] = null) and (right [x] = null) then begin
| {x - лист, т.е. не имеет сыновей}
| make_free (x);
| if x = root then begin
| | root := null;
| end else if direction = l then begin
| | left [father] := null;
| end else begin {direction = r}
| | right [father] := null;
| end;
end else if (left[x]=null) and (right[x] <> null) then begin
| {x удаляется, а right [x] занимает место x}
| make_free (x);
| if x = root then begin
| | root := right [x];
| end else if direction = l then begin
| | left [father] := right [x];
| end else begin {direction = r}
| | right [father] := right [x];
| end;
end else if (left[x] <> null) and (right[x]=null) then begin
| ..симметрично
end else begin {left [x] <> null, right [x] <> null}
| ..удалить вершину с двумя сыновьями
end;
```

Удаление вершины с двумя сыновьями нельзя сделать просто так, но её можно предварительно поменять с вершиной, пометка на которой является непосредственно следующим (в порядке возрастания) элементом за пометкой на x.

```
y := right [x];
father := x; direction := r;
{теперь father и direction относятся к вершине y}
while left [y] <> null do begin
| father := y; direction := l;
| y := left [y];
end;
{val [y] - минимальная из пометок, больших val [x],
 y не имеет левого сына}
```

```
val [x] := val [y];
..удалить вершину y (как удалять вершину, у которой нет
  левого сына, мы уже знаем)
```
□

14.1.6. Упростить программу удаления, заметив, что некоторые случаи (например, первые два из четырёх) можно объединить. □

14.1.7. Использовать упорядоченные деревья для представления функций, область определения которых — конечные множества значений типа `T`, а значения имеют некоторый тип `U`. Операции: вычисление значения на данном аргументе, изменение значения на данном аргументе, доопределение функции на данном аргументе, исключение элемента из области определения функции.

Решение. Делаем как раньше, добавив ещё один массив

```
func_val: array [1..n] of U;
```

если `val[x] = t`, `func_val[x] = u`, то значение хранимой функции на `t` равно `u`. □

14.1.8. Предположим, что необходимо уметь также отыскивать `k`-ый элемент множества (в порядке возрастания), причём количество действий должно быть не более $C \cdot$ (высота дерева). Какую дополнительную информацию надо хранить в вершинах дерева?

Решение. В каждой вершине будем хранить число всех её потомков. Добавление и исключение вершины требует коррекции лишь на пути от корня к этой вершине. В процессе поиска `k`-ой вершины поддерживается такой инвариант: искомая вершина является `s`-ой вершиной поддерева с корнем в `x` (здесь `s` и `x` — переменные). □

Оценка количества действий

Для каждой из операций (проверки, добавления и исключения) количество действий не превосходит $C \cdot$ (высота дерева). Для «ровно подстриженного» дерева (когда все листья на одной высоте) высота по порядку величины равна логарифму числа вершин. Однако для кривобокого дерева всё может быть гораздо хуже: в наихудшем случае все вершины образуют цепь и высота равна числу вершин. Так случится, если элементы множества добавляются в возрастающем или убывающем порядке. Можно доказать, однако, что при добавлении элементов «в случайном порядке» средняя высота дерева будет не больше $C \log$(число вершин). Если этой оценки «в среднем» мало, необходимы дополнительные действия по поддержанию «сбалансированности» дерева. Об этом смотри в следующем пункте.

14.2. Сбалансированные деревья

Дерево называется *сбалансированным* (или АВЛ-деревом в честь изобретателей этого метода Г. М. Адельсона-Вельского и Е. М. Ландиса), если для любой его вершины высоты левого и правого поддеревьев этой вершины отличаются не более чем на 1. (В частности, когда одного из сыновей нет, другой — если он есть — обязан быть листом.)

14.2.1. Найти минимальное и максимальное возможное количество вершин в сбалансированном дереве высоты n.

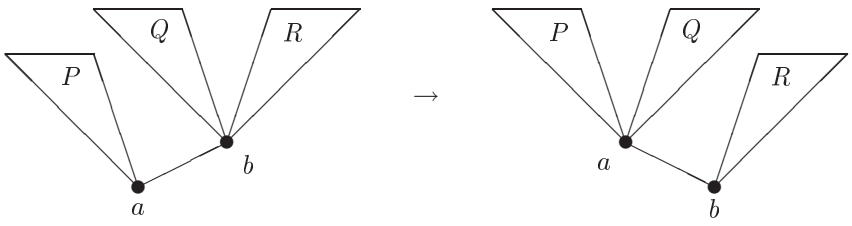
Решение. Максимальное число вершин равно $2^n - 1$. Если m_n — минимальное число вершин, то, как легко видеть, $m_{n+2} = 1 + m_n + m_{n+1}$, откуда $m_n = \Phi_{n+2} - 1$ (Φ_n — n-ое число Фибоначчи, $\Phi_1 = 1$, $\Phi_2 = 1$, $\Phi_{n+2} = \Phi_n + \Phi_{n+1}$). □

14.2.2. Доказать, что сбалансированное дерево с n вершинами имеет высоту не больше $C \log n$ для некоторой константы C, не зависящей от n.

Решение. Индукцией по n легко доказать, что $\Phi_{n+2} \geqslant a^n$, где a — больший корень квадратного уравнения $a^2 = 1 + a$, то есть $a = (\sqrt{5} + 1)/2$. Остаётся воспользоваться предыдущей задачей. □

Вращения

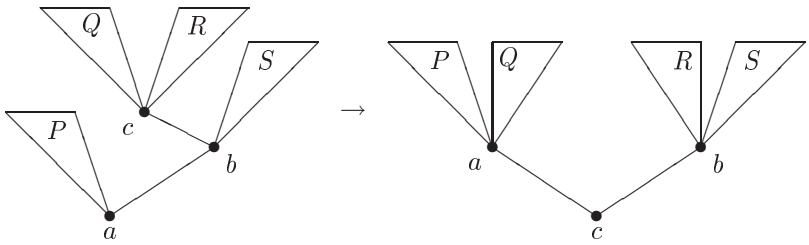
Мы хотим восстанавливать сбалансированность дерева после включения и удаления элементов. Для этого необходимы какие-то преобразования дерева, не меняющие множества пометок на его вершинах и не нарушающие упорядоченности, но способствующие лучшей сбалансированности. Опишем несколько таких преобразований.



Пусть вершина a имеет правого сына b. Обозначим через P левое поддерево вершины a, через Q и R — левое и правое поддеревья вершины b. Упорядоченность дерева требует, чтобы $P < a < Q < b < R$ (точнее сле-

довало бы сказать «любая пометка на P меньше пометки на a», «пометка на a меньше любой пометки на Q» и т. д., но мы позволим себе этого не делать). Точно того же требует упорядоченность дерева с корнем b, его левым сыном a, в котором P и Q — левое и правое поддеревья a, R — правое поддерево b. Поэтому первое дерево можно преобразовать во второе, не нарушая упорядоченности. Такое преобразование назовём *малым правым вращением* (правым — поскольку существует симметричное, левое, малым — поскольку есть и большое, которое мы сейчас опишем).

Пусть b — правый сын a, c — левый сын b, P — левое поддерево a, Q и R — левое и правое поддеревья c, S — правое поддерево b. Тогда $P < a < Q < c < R < b < S$.

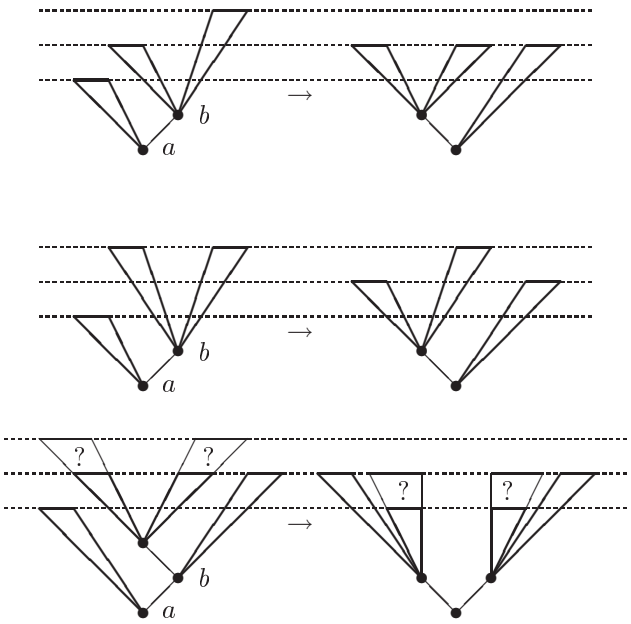


Такой же порядок соответствует дереву с корнем c, имеющим левого сына a и правого сына b, для которого P и Q — поддеревья вершины a, а R и S — поддеревья вершины b. Соответствующее преобразование будем называть *большим правым вращением*. (Аналогично определяется симметричное ему *большое левое вращение*.)

14.2.3. Дано дерево, сбалансированное всюду, кроме корня, в котором разница высот равна 2 (т. е. левое и правое поддеревья корня сбалансированы и их высоты отличаются на 2). Доказать, что оно может быть превращено в сбалансированное одним из четырёх описанных преобразований, причём высота его останется прежней или уменьшится на 1.

Решение. Пусть более низким является, например, левое поддерево, и его высота равна k. Тогда высота правого поддерева равна $k + 2$. Обозначим корень через a, а его правого сына (он обязательно есть) через b. Рассмотрим левое и правое поддеревья вершины b. Одно из них обязательно имеет высоту $k + 1$, а другое может иметь высоту k или $k + 1$ (меньше k быть не может, так как поддеревья сбалансированы).

Если высота левого поддерева равна $k+1$, а правого — k, то потребуется большое правое вращение; в остальных случаях помогает малое. Вот как выглядят три случая балансировки дерева:



□

14.2.4. В сбалансированное дерево добавили или из него удалили лист. Доказать, что можно восстановить сбалансированность с помощью нескольких вращений, причём их число не больше высоты дерева.

Решение. Будем доказывать более общий факт:

Лемма. Если в сбалансированном дереве X одно из его поддеревьев Y заменили на сбалансированное дерево Z, причём высота Z отличается от высоты Y не более чем на 1, то полученное такой «прививкой» дерево можно превратить в сбалансированное вращениями (причём количество вращений не превосходит высоты, на которой делается прививка).

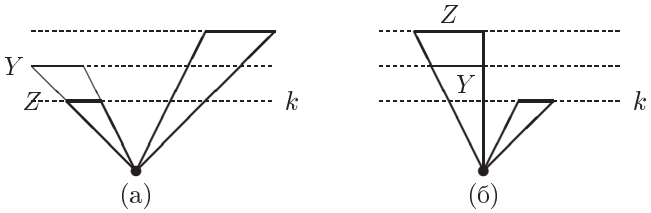
Частным случаем прививки является замена пустого поддерева на лист или наоборот, так что достаточно доказать эту лемму.

Доказательство леммы. Индукция по высоте, на которой делается прививка. Если она происходит в корне (заменяется всё дерево цели-

ком), то всё очевидно («привой» сбалансирован по условию). Пусть заменяется некоторое поддерево, например, левое поддерево некоторой вершины x. Возможны два случая.

1) После прививки сбалансированность в вершине x не нарушилась (хотя, возможно, нарушилась сбалансированность в предках x: высота поддерева с корнем в x могла измениться). Тогда можно сослаться на предположение индукции, считая, что мы прививали целиком поддерево с корнем в x.

2) Сбалансированность в x нарушилась. При этом разница высот равна 2 (больше она быть не может, так как высота Z отличается от высоты Y не более чем на 1). Разберём два варианта.



а) Выше правое (не заменявшееся) поддерево вершины x. Пусть высота левого (т. е. Z) равна k, правого — $k+2$. Высота старого левого поддерева вершины x (т. е. Y) была равна $k+1$. Поддерево с корнем x имело в исходном дереве высоту $k+3$, и эта высота не изменилась после прививки.

По предыдущей задаче вращение преобразует поддерево с корнем в x в сбалансированное поддерево высоты $k+2$ или $k+3$. То есть высота поддерева с корнем x — в сравнении с его прежней высотой — не изменилась или уменьшилась на 1, и мы можем воспользоваться предположением индукции.

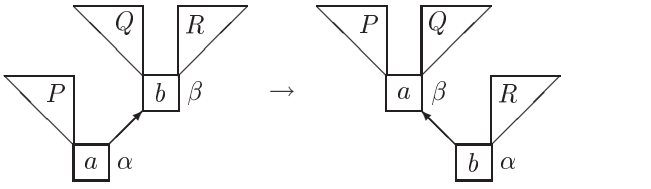
б) Выше левое поддерево вершины x. Пусть высота левого (т. е. Z) равна $k+2$, правого — k. Высота старого левого поддерева (т. е. Y) была равна $k+1$. Поддерево с корнем x в исходном дереве X имело высоту $k+2$, после прививки она стала равна $k+3$. После подходящего вращения (см. предыдущую задачу) поддерево с корнем в x станет сбалансированным, его высота будет равна $k+2$ или $k+3$, так что изменение высоты по сравнению с высотой поддерева с корнем x в дереве X не превосходит 1 и можно сослаться на предположение индукции. □

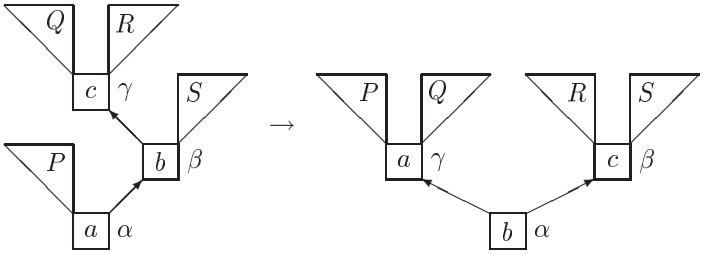
14.2.5. Составить программы добавления и удаления элементов, сохраняющие сбалансированность. Число действий не должно превосходить $C \cdot$ (высота дерева). Разрешается хранить в вершинах дерева дополнительную информацию, необходимую при балансировке.

Решение. Будем хранить для каждой вершины разницу между высотой её правого и левого поддеревьев:

`diff [i]` = (высота правого поддерева вершины `i`) −
− (высота левого поддерева вершины `i`).

Нам потребуются четыре процедуры, соответствующие большим и малым правым и левым вращениями. Но вначале два замечания. (1) Нам нужно, чтобы при вращении поддерева номер его корня не менялся. (В противном случае потребовалось бы корректировать информацию в отце корня, что нежелательно.) Этого можно достичь, так как номера вершин дерева можно выбирать независимо от их значений. (На картинках номер указан сбоку от вершины, а значение — внутри.)





(2) После преобразований мы должны также изменить соответственно значения в массиве `diff`. Для этого достаточно знать высоты деревьев $P, Q, \ldots$ с точностью до константы, поэтому можно предполагать, что одна из высот равна нулю.

Вот процедуры вращений:

```
procedure SR (a:integer); {малое правое вращение с корнем a}
  var b: 1..n; val_a,val_b: T; h_P,h_Q,h_R: integer;
begin
  b := right [a]; {b <> null}
  val_a := val [a]; val_b := val [b];
  h_Q := 0; h_R := diff[b]; h_P := (max(h_Q,h_R)+1)-diff[a];
  val [a] := val_b; val [b] := val_a;
  right [a] := right [b] {поддерево R}
  right [b] := left [b] {поддерево Q}
  left [b] := left [a] {поддерево P}
  left [a] := b;
  diff [b] := h_Q - h_P;
  diff [a] := h_R - (max (h_P, h_Q) + 1);
end;

procedure BR(a:integer);{большое правое вращение с корнем a}
  var b,c: 1..n; val_a,val_b,val_c: T;
      h_P,h_Q,h_R,h_S: integer;
begin
  b := right [a]; c := left [b]; {b,c <> null}
  val_a := val [a]; val_b := val [b]; val_c := val [c];
  h_Q := 0; h_R := diff[c]; h_S := (max(h_Q,h_R)+1)+diff[b];
  h_P := 1 + max (h_S, h_S-diff[b]) - diff [a];
  val [a] := val_c; val [c] := val_a;
  left [b] := right [c] {поддерево R}
  right [c] := left [c] {поддерево Q}
  left [c] := left [a] {поддерево P}
  left [a] := c;
  diff [b] := h_S - h_R;
  diff [c] := h_Q - h_P;
  diff [a] := max (h_S, h_R) - max (h_P, h_Q);
end;
```

Левые вращения (большое и малое) записываются симметрично.

Процедуры добавления и удаления элементов пишутся как раньше, но только добавление и удаление должно сопровождаться коррекцией массива `diff` и восстановлением сбалансированности.

При этом используется процедура с такими свойствами:

> **дано:** левое и правое поддеревья вершины с номером `a` сбалансированы, в самой вершине разница высот не больше 2, в поддереве с корнем `a` массив `diff` заполнен правильно;

надо: поддерево с корнем a сбалансировано и массив diff соответственно изменён, d — изменение его высоты (равно 0 или -1); в остальной части всё осталось как было — в частности, значения diff.

```
procedure balance (a: integer; var d: integer);
begin {-2 <= diff[a] <= 2}
  if diff [a] = 2 then begin
    b := right [a];
    if diff [b] = -1 then begin
      BR (a); d := -1;
    end else if diff [b] = 0 then begin
      SR (a); d := 0;
    end else begin {diff [b] = 1}
      SR (a); d := - 1;
    end;
  end else if diff [a] = -2 then begin
    b := left [a];
    if diff [b] = 1 then begin
      BL (a); d := -1;
    end else if diff [b] = 0 then begin
      SL (a); d := 0;
    end else begin {diff [b] = -1}
      SL (a); d := - 1;
    end;
  end else begin {-2 < diff [a] < 2, ничего делать не надо}
    d := 0;
  end;
end;
```

Восстановление сбалансированности требует движения от листьев к корню, поэтому будем хранить в стеке путь от корня к рассматриваемой в данный момент вершине. Элементами стека будут пары ⟨вершина, направление движения из неё⟩, т. е. значения типа

```
record
  vert: 1..n; {вершина}
  direction : (l, r); {l - левое, r - правое}
end;
```

Программа добавления элемента t теперь выглядит так:

```
if root = null then begin
  get_free (root);
  left[root] := null; right[root] := null; diff[root] := 0;
```

```
  val[root] := t;
end else begin
  x := root;
  ..сделать стек пустым
  {инвариант: осталось добавить t к непустому поддереву с
   корнем в x; стек содержит путь к x}
  while ((t < val [x]) and (left [x] <> null)) or
        ((t > val [x]) and (right [x] <> null)) do begin
    if t < val [x] then begin
      ..добавить в стек пару <x, l>
      x := left [x];
    end else begin {t > val [x]}
      ..добавить в стек пару <x, r>
      x := right [x];
    end;
  end;
  if t <> val [x] then begin {t нет в дереве}
    get_free (i); val [i] := t;
    left [i] := null; right [i] := null; diff [i] := 0;
    if t < val [x] then begin
      ..добавить в стек пару <x, l>
      left [x] := i;
    end else begin {t > val [x]}
      ..добавить в стек пару <x, r>
      right [x] := i;
    end;
    d := 1;
    {инвариант: стек содержит путь к изменившемуся
     поддереву, высота  которого увеличилась по сравнению
     с высотой в исходном дереве на d (=0 или 1); это
     поддерево  сбалансировано; значения diff для его
     вершин правильны; в остальном дереве  всё  осталось
     как  было - в частности, значения diff}
    while (d <> 0) and ..стек непуст do begin {d = 1}
      ..взять из стека пару в <v, direct>
      if direct = l then begin
        if diff [v] = 1 then begin
          c := 0;
        end else begin
          c := 1;
        end;
        diff [v] := diff [v] - 1;
      end else begin
        {direct = r}
```

```
    if diff [v] = -1 then begin
      c := 0;
    end else begin
      c := 1;
    end;
    diff [v] := diff [v] + 1;
  end;
  {c = изменение высоты поддерева с корнем в v по
   сравнению с исходным деревом; массив diff
   содержит правильные значения для этого поддерева;
   возможно нарушение сбалансированности в v}
  balance (v, d1); d := c + d1;
 end;
end;
```

Легко проверить, что значение d может быть равно только 0 или 1 (но не -1): если c=0, то diff[v]=0 и балансировка не производится.

Программа удаления строится аналогично. Её основной фрагмент таков:

```
{инвариант: стек содержит путь к изменившемуся поддереву,
 высота которого изменилась по сравнению с высотой в
 исходном дереве на d (=0 или -1); это поддерево
 сбалансировано; значения diff для его вершин правильны;
 в остальном дереве всё осталось как было -
 в частности, значения diff}
while (d <> 0) and ..стек непуст do begin
 {d = -1}
 ..взять из стека пару в <v, direct>
 if direct = l then begin
  if diff [v] = -1 then begin
    c := -1;
  end else begin
    c := 0;
  end;
  diff [v] := diff [v] + 1;
 end else begin {direct = r}
  if diff [v] = 1 then begin
    c := -1;
  end else begin
    c := 0;
  end;
  diff [v] := diff [v] - 1;
 end;
```

```
    {c = изменение высоты поддерева с корнем в v по
     сравнению с исходным деревом; массив diff содержит
     правильные значения для этого поддерева;
     возможно нарушение сбалансированности в v}
    balance (v, d1);
    d := c + d1;
  end;
```

Легко проверить, что значение `d` может быть равно только `0` или `-1` (но не `-2`): если `c=-1`, то `diff[v]=0` и балансировка не производится.

Отметим также, что наличие стека делает излишними переменные `father` и `direction` (их роль теперь играет вершина стека). □

14.2.6. Доказать, что при добавлении элемента

(а) второй из трёх случаев балансировки (см. рисунок на с. 236) невозможен;

(б) полная балансировка требует не более одного вращения (после чего всё дерево становится сбалансированным), в то время как при удалении элемента может понадобиться много вращений. □

Замечание. Мы старались записать программы добавления и удаления так, чтобы они были как можно более похожими друг на друга. Используя специфику каждой из них, можно многое упростить.

Существуют и другие способы представления множеств, гарантирующие число действий порядка $\log n$ на каждую операцию. Опишем один из них (называемый *Б-деревьями*).

До сих пор каждая вершина содержала один элемент хранимого множества. Этот элемент служил границей между левым и правым поддеревом. Будем теперь хранить в вершине $k \geqslant 1$ элементов множества (число k может меняться от вершины к вершине, а также при добавлении и удалении новых элементов, см. далее). Эти k элементов служат разделителями для $k+1$ поддеревьев. Пусть фиксировано некоторое число $t \geqslant 1$. Будем рассматривать деревья, обладающие такими свойствами:

1) Каждая вершина содержит от t до $2t$ элементов (за исключением корня, который может содержать любое число элементов от 0 до $2t$).

2) Вершина с k элементами либо имеет $k+1$ сыновей, либо не имеет сыновей вообще (является *листом*).

3) Все листья находятся на одной и той же высоте.

Добавление элемента происходит так. Если лист, в который он попадает, неполон (т. е. содержит менее $2t$ элементов), то нет проблем. Если он полон, то $2t+1$ элемент (все элементы листа и новый элемент) разбиваем на два листа по t элементов и разделяющий их серединный элемент. Этот серединный элемент надо добавить в вершину предыдущего уровня. Это возможно, если в ней менее $2t$ элементов. Если и она полна, то её разбивают на две, выделяют серединный элемент и т. д. Если в конце концов мы захотим добавить элемент в корень, а он окажется полным, то корень расщепляется на две вершины, а высота дерева увеличивается на 1.

Удаление элемента, находящегося не в листе, сводится к удалению непосредственно следующего за ним, который находится в листе. Поэтому достаточно научиться удалять элемент из листа. Если лист при этом становится слишком маленьким, то его можно пополнить за счёт соседнего листа — если только и он не имеет минимально возможный размер t. Если же оба листа имеют размер t, то на них вместе $2t$ элементов, вместе с разделителем — $2t+1$. После удаления одного элемента остаётся $2t$ элементов — как раз на один лист. Если при этом вершина предыдущего уровня становится меньше нормы, процесс повторяется и т. д.

14.2.7. Реализовать описанную схему хранения множеств, убедившись, что она также позволяет обойтись $C \log n$ действий для операций включения, исключения и проверки принадлежности. □

14.2.8. Можно определять сбалансированность дерева иначе: требовать, чтобы для каждой вершины её левое и правое поддеревья имели не слишком сильно отличающиеся количества вершин. (Преимущество такого определения состоит в том, что при вращениях не нарушается сбалансированность в вершинах, находящихся ниже точки вращения.) Реализовать на основе этой идеи способ хранения множеств, гарантирующий оценку в $C \log n$ действий для включения, удаления и проверки принадлежности.

[Указание. Он также использует большие и малые вращения. Подробности см. в книге Рейнгольда, Нивергельта и Део «Комбинаторные алгоритмы».] □

15. КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

15.1. Общий алгоритм разбора

Чтобы определить то, что называют *контекстно-свободной грамматикой* (КС-грамматикой), надо:

- указать конечное множество A, называемое *алфавитом*; его элементы называют *символами*; конечные последовательности символов называют *словами* (в данном алфавите);
- разделить все символы алфавита A на две группы: *терминальные* («окончательные») и *нетерминальные* («промежуточные»);
- выбрать среди нетерминальных символов один, называемый *начальным*;
- указать конечное число *правил* грамматики, каждое из которых должно иметь вид $K \to X$, где K — некоторый нетерминальный символ, а X — слово (в него могут входить и терминальные, и нетерминальные символы).

Пусть фиксирована КС-грамматика (мы часто будем опускать префикс «КС-», так как других грамматик у нас не будет). *Выводом* в этой грамматике называется последовательность слов $X_0, X_1, \ldots, X_n$, в которой X_0 состоит из одного символа, и этот символ — начальный, а X_{i+1} получается из X_i заменой некоторого нетерминального символа K на слово X по одному из правил грамматики. Слово, составленное из терминальных символов, называется *выводимым*, если существует вывод, который им кончается. Множество всех выводимых слов (из терминальных символов) называется *языком, порождаемым данной грамматикой*.

В этой и следующей главе нас будет интересовать такой вопрос: дана КС-грамматика; построить алгоритм, который по любому слову проверяет, выводимо ли оно в этой грамматике.

Пример 1. Алфавит:

```
( ) [ ] E
```

(четыре терминальных символа и один нетерминальный символ `E`). Начальный символ: `E`. Правила:

$$\texttt{E} \to \texttt{(E)}$$
$$\texttt{E} \to \texttt{[E]}$$
$$\texttt{E} \to \texttt{EE}$$
$$\texttt{E} \to$$

(в последнем правиле справа стоит пустое слово).

Примеры выводимых слов:

```
                (пустое слово)
  ()
  ([])
 ()[([])]
[()[]()[]]
```

Примеры невыводимых слов:

```
(
)(
(]
([)]
```

Эта грамматика встречалась в разделе 6.1 (где выводимость в ней проверялась с помощью стека).

Пример 2. Другая грамматика, порождающая тот же язык:

Алфавит: `( ) [ ] T E`

Правила:

$$\texttt{E} \to$$
$$\texttt{E} \to \texttt{TE}$$
$$\texttt{T} \to \texttt{(E)}$$
$$\texttt{T} \to \texttt{[E]}$$

Начальным символом во всех приводимых далее примерах будем считать символ, стоящий в левой части первого правила (в данном случае это символ `E`), не оговаривая этого особо.

Для каждого нетерминального символа можно рассмотреть множество всех слов из терминальных символов, которые из него выводятся (аналогично тому, как это сделано для начального символа в определении выводимости в грамматике). Каждое правило грамматики можно рассматривать как свойство этих множеств. Покажем это на примере только что приведённой грамматики. Пусть T и E — множества слов (из скобок), выводимых из нетерминалов `T` и `E` соответственно. Тогда правилам грамматики соответствуют такие свойства:

`E` →	E содержит пустое слово
`E` → `TE`	если слово A принадлежит T, а слово B принадлежит E, то слово AB принадлежит E
`T` → `[E]`	если A принадлежит E, то слово `[`A`]` принадлежит T
`T` → `(E)`	если A принадлежит E, то слово `(`A`)` принадлежит T

Сформулированные свойства множеств E, T не определяют эти множества однозначно (например, они остаются верными, если в качестве E и T взять множество всех слов). Однако можно доказать, что множества, задаваемые грамматикой, являются минимальными среди удовлетворяющих этим условиям.

15.1.1. Сформулировать точно и доказать это утверждение для произвольной контекстно-свободной грамматики. □

15.1.2. Построить грамматику, в которой выводимы слова
(а) `00..0011..11` (число нулей равно числу единиц);
(б) `00..0011..11` (число нулей вдвое больше числа единиц);
(в) `00..0011..11` (число нулей больше числа единиц);
(и только они). □

15.1.3. Доказать, что не существует КС-грамматики, в которой были бы выводимы слова вида `00..0011..1122..22`, в которых числа нулей, единиц и двоек равны, и только они.

[Указание. Доказать следующую лемму о произвольной КС-грамматике: для любого достаточно длинного слова F, выводимого в этой грамматике, существует такое его представление в виде $ABCDE$, что

любое слово вида $AB\ldots BCD\ldots DE$, где B и D повторены одинаковое число раз, также выводимо в этой грамматике. (Это можно установить, найдя нетерминальный символ, оказывающийся своим собственным «наследником» в процессе вывода.)] □

Нетерминальный символ можно рассматривать как «родовое имя» для выводимых из него слов. В следующем примере для наглядности в качестве нетерминальных символов использованы фрагменты русских слов, заключённые в угловые скобки. (С точки зрения грамматики каждый такой фрагмент — один символ!)

Пример 3. Алфавит:

терминалы: `+ * ( ) x`
нетерминалы: ⟨выр⟩ ⟨оствыр⟩ ⟨слаг⟩ ⟨остслаг⟩ ⟨множ⟩

Правила:

⟨выр⟩ → ⟨слаг⟩ ⟨оствыр⟩
⟨оствыр⟩ → `+` ⟨выр⟩
⟨оствыр⟩ →
⟨слаг⟩ → ⟨множ⟩ ⟨остслаг⟩
⟨остслаг⟩ → `*` ⟨слаг⟩
⟨остслаг⟩ →
⟨множ⟩ → `x`
⟨множ⟩ → `(` ⟨выр⟩ `)`

Согласно этой грамматике, выражение ⟨выр⟩ — это последовательность слагаемых ⟨слаг⟩, разделённых плюсами, слагаемое — это последовательность множителей ⟨множ⟩, разделённых звёздочками (знаками умножения), а множитель — это либо буква `x`, либо выражение в скобках.

15.1.4. Привести пример другой грамматики, задающей тот же язык.

Ответ. Вот один из вариантов:

⟨выр⟩ → ⟨выр⟩ `+` ⟨выр⟩
⟨выр⟩ → ⟨выр⟩ `*` ⟨выр⟩
⟨выр⟩ → `x`
⟨выр⟩ → `(` ⟨выр⟩ `)` □

Эта грамматика хоть и проще, но в некоторых отношениях хуже, о чём мы ещё будем говорить.

15.1.5. Дана произвольная КС-грамматика. Построить алгоритм проверки принадлежности задаваемому ей языку, работающий полиномиальное время (т. е. число действий не превосходит полинома от длины проверяемого слова; полином может зависеть от грамматики).

Решение. Заметим, что требование полиномиальности исключает возможность решения, основанном на переборе всех возможных выводов. Тем не менее полиномиальный алгоритм существует. Поскольку практического значения он не имеет (используемые на практике КС-грамматики обладают дополнительными свойствами, позволяющими строить более эффективные алгоритмы), мы изложим лишь общую схему решения.

(1) Пусть в грамматике есть нетерминалы $K_1, \ldots, K_n$. Построим новую грамматику с нетерминалами $K'_1, \ldots, K'_n$ так, чтобы выполнялось такое свойство: из K'_i выводятся (в новой грамматике) те же слова, что из K_i в старой, за исключением пустого слова, которое не выводится.

Чтобы выполнить такое преобразование грамматики, надо выяснить, из каких нетерминалов исходной грамматики выводится пустое слово, а затем каждое правило заменить на совокупность правил, получающихся, если в правой части опустить какие-либо из нетерминалов, из которых выводится пустое слово, а у остальных поставить штрихи. Например, если в исходной грамматике было правило

$$\mathtt{K} \to \mathtt{L\ M\ N},$$

причём из L и N выводится пустое слово, а из M нет, то это правило надо заменить на правила

$$\mathtt{K}' \to \mathtt{L}'\mathtt{M}'\mathtt{N}'$$
$$\mathtt{K}' \to \mathtt{M}'\mathtt{N}'$$
$$\mathtt{K}' \to \mathtt{L}'\mathtt{M}'$$
$$\mathtt{K}' \to \mathtt{M}'$$

(2) Итак, мы свели дело к грамматике, где ни из одного нетерминала не выводится пустое слово. Теперь устраним «циклы» вида

$$\mathtt{K} \to \mathtt{L}$$
$$\mathtt{L} \to \mathtt{M}$$
$$\mathtt{M} \to \mathtt{N}$$
$$\mathtt{N} \to \mathtt{K}$$

(в правой части каждого правила один символ, и эти символы образуют цикл произвольной длины): это легко сделать, отождествив все входящие в цикл нетерминалы.

(3) Теперь проверка принадлежности какого-либо слова языку, порождённому грамматикой, может выполняться так: для каждого подслова проверяемого слова и для каждого нетерминала выясняем, порождается ли это подслово этим нетерминалом. При этом подслова проверяются в порядке возрастания длин, а нетерминалы — в таком порядке, чтобы при наличии правила $K \to L$ нетерминал L проверялся раньше нетерминала K. (Это возможно в силу отсутствия циклов.) Поясним этот процесс на примере.

Пусть в грамматике есть правила

$$\texttt{K} \to \texttt{L}$$
$$\texttt{K} \to \texttt{M N L}$$

и других правил, содержащих K в левой части, нет. Мы хотим узнать, выводится ли данное слово A из нетерминала K. Это будет так в одном из случаев:

- если A выводится из L;
- если A можно разбить на непустые слова B, C, D, для которых B выводится из M, C выводится из N, а D выводится из L.

Вся эта информация уже есть (слова B, C, D короче A, а L рассмотрен до K).

Легко видеть, что число действий этого алгоритма полиномиально. Степень полинома зависит от числа нетерминалов в правых частях правил и может быть понижена, если грамматику преобразовать к форме, в которой правая часть каждого правила не более 2 нетерминалов (это легко сделать, вводя новые нетерминалы: например, правило $\texttt{K} \to \texttt{LMK}$ можно заменить на $\texttt{K} \to \texttt{LN}$ и $\texttt{N} \to \texttt{MK}$, где N — новый нетерминал). □

15.1.6. Рассмотрим грамматику с единственным нетерминалом K, нетерминалами 0, 1, 2, 3 и правилами

$$\texttt{K} \to \texttt{0}$$
$$\texttt{K} \to \texttt{1 K}$$
$$\texttt{K} \to \texttt{2 K K}$$
$$\texttt{K} \to \texttt{3 K K K}$$

Как проверить выводимость слова в этой грамматике, читая слово слева направо? (Число действий при прочтении одной буквы должно быть ограничено.)

Решение. Хранится целая переменная `n`, инвариант: слово выводимо ⇔ непрочитанная часть представляет собой конкатенацию (соединение) `n` выводимых слов. □

15.1.7. Тот же вопрос для грамматики

```
K → 0
K → K 1
K → K K 2
K → K K K 3
```

□

15.2. Метод рекурсивного спуска

В отличие от алгоритма предыдущего раздела (представляющего чисто теоретический интерес), алгоритмы на основе рекурсивного спуска часто используются на практике. Этот метод применим, однако, далеко не ко всем грамматикам. Мы обсудим необходимые ограничения позднее.

Идея метода рекурсивного спуска такова. Для каждого нетерминала `K` мы строим процедуру `ReadK`, которая — в применении к любому входному слову x — делает две вещи:

- находит наибольшее начало z слова x, которое может быть началом выводимого из `K` слова;
- сообщает, является ли найденное слово z выводимым из `K`.

Прежде чем описывать этот метод более подробно, договоримся о том, как процедуры получают сведения о входном слове и как сообщают о результатах своей работы. Мы предполагаем, что буквы входного слова поступают к ним по одной, т. е. имеется граница, отделяющая «прочитанную» часть от «непрочитанной». Будем считать, что есть функция (без параметров)

```
Next: Symbol
```

дающая первый непрочитанный символ. Её значениями могут быть терминальные символы, а также специальный символ `EOI` (End Of Input — конец входа), означающий, что всё слово уже прочитано. Вызов этой функции, разумеется, не сдвигает границы между прочитанной и непрочитанной частью — для этого есть процедура `Move`, которая сдвигает границу на один символ. (Она применима, если `Next<>EOI`.) Пусть, наконец, имеется булевская переменная `b`.

Теперь мы можем сформулировать наши требования к процедуре ReadK. Они состоят в следующем:

- ReadK прочитывает из оставшейся части слова максимальное начало A, являющееся началом некоторого слова, выводимого из K;
- значение b становится истинным или ложным в зависимости от того, является ли A выводимым из K или лишь невыводимым началом выводимого (из K) слова.

Для удобства введём такую терминологию: выводимое из K слово будем называть K-словом, а любое начало любого выводимого из K слова — K-началом. Два сформулированных требования вместе будем выражать словами «ReadK корректна для K».

Начнём с примера. Пусть правило

K → L M

является единственным правилом грамматики, содержащим K в левой части, пусть L, M — нетерминалы и ReadL, ReadM — корректные (для них) процедуры.

Рассмотрим такую процедуру:

```
procedure ReadK;
begin
| ReadL;
| if b then begin
| | ReadM;
| end;
end;
```

15.2.1. Привести пример, когда эта процедура будет некорректной для K.

Ответ. Пусть из L выводится любое слово вида 00..00, а из M выводится лишь слово 01. Тогда из K выводится слово 00001, но процедура ReadK этого не заметит. □

Укажем достаточные условия корректности процедуры ReadK. Для этого нам понадобятся некоторые обозначения. Пусть фиксированы КС-грамматика и некоторый нетерминал N этой грамматики. Рассмотрим N-слово A, которое имеет собственное начало B, также являю-

щееся N-словом (если такие есть). Для любой пары таких слов A и B рассмотрим терминальный символ, идущий в A непосредственно за B. Множество всех таких терминалов обозначим Посл(N). (Если никакое N-слово не является собственным началом другого N-слова, то множество Посл(N) пусто.)

15.2.2. Указать (а) Посл(E) для примера 1 (с. 246); (б) Посл(E) и Посл(T) для примера 2 (с. 246); (в) Посл(⟨слаг⟩) и Посл(⟨множ⟩) для примера 3 (с. 248);

Ответ. (а) Посл(E) = {[, (}. (б) Посл(E) = {[, (}; Посл(T) пусто (никакое T-слово не является началом другого). (в) Посл(⟨слаг⟩) = {*}; Посл(⟨множ⟩) пусто. □

Кроме того, для каждого нетерминала N обозначим через Нач(N) множество всех терминалов, являющихся первыми буквами непустых N-слов. Это обозначение — вместе с предыдущим — позволит дать достаточное условие корректности процедуры ReadK в описанной выше ситуации.

15.2.3. Доказать, что если Посл(L) не пересекается с Нач(M) и множество всех M-слов непусто, то ReadK корректна.

Решение. Рассмотрим два случая.

(1) Пусть после ReadL значение переменной b ложно. В этом случае ReadL читает со входа максимальное L-начало A, не являющееся L-словом. Оно является K-началом (здесь важно, что множество M-слов непусто.). Будет ли оно максимальным K-началом среди начал входа? Если нет, то A является началом слова BC, где B есть L-слово, C есть M-начало и BC — более длинное начало входа, чем A. Если B длиннее A, то A — не максимальное начало входа, являющееся L-началом, что противоречит корректности ReadL. Если $B = A$, то A было бы L-словом, а это не так. Значит, B короче A, C непусто и первый символ слова C следует в A за последним символом слова B, т. е. Посл(L) пересекается с Нач(M). Противоречие. Итак, A максимально. Из сказанного следует также, что A не является K-словом. Корректность процедуры ReadK в этом случае проверена.

(2) Пусть после ReadL значение переменной b истинно. Тогда прочитанное процедурой ReadK начало входа имеет вид AB, где A есть L-слово, а B есть M-начало. Тем самым AB есть K-начало. Проверим его максимальность. Пусть C есть большее K-начало. Тогда либо C есть L-начало (что невозможно, так как A было максимальным L-началом), либо $C = A'B'$, где A' — L-слово, B' — M-начало. Если A' короче A, то B' непусто и начинается с символа, принадлежащего и Нач(M), и Посл(L),

что невозможно. Если A' длиннее A, то A — не максимальное L-начало. Итак, $A' = A$. Но в этом случае B' есть продолжение B, что противоречит корректности ReadM. Таким образом, AB — максимальное K-начало. Остаётся проверить правильность выдаваемого процедурой ReadK значения переменной b. Если оно истинно, то это очевидно. Если оно ложно, то B не есть M-слово, и надо проверить, что AB — не K-слово. В самом деле, если бы выполнялось $AB = A'B'$, где A' — L-слово, B' — M-слово, то A' не может быть длиннее A (ReadL читает максимальное слово), A' не может быть равно A (тогда B' равно B и не является M-словом) и A' не может быть короче A (тогда первый символ B' принадлежит и Нач(M), и Посл(L)). Задача решена. □

Перейдём теперь к другому частному случаю. Пусть в КС-грамматике есть правила

$$\mathtt{K} \to \mathtt{L}$$
$$\mathtt{K} \to \mathtt{M}$$
$$\mathtt{K} \to \mathtt{N}$$

и других правил с левой частью K нет.

15.2.4. Считая, что ReadL, ReadM и ReadN корректны (для L, M и N) и что множества Нач(L), Нач(M) и Нач(N) не пересекаются, написать процедуру, корректную для K.

Решение. Схема процедуры такова:

```
procedure ReadK;
begin
| if (Next принадлежит Нач(L)) then begin
| | ReadL;
| end else if (Next принадлежит Нач(M)) then begin
| | ReadM;
| end else if (Next принадлежит Нач(N)) then begin
| | ReadN;
| end else begin
| | b := true или false в зависимости от того,
| |      выводимо ли пустое слово из K или нет
| end;
end;
```

Докажем, что ReadK корректно реализует K. Если Next не принадлежит ни одному из множеств Нач(L), Нач(M), Нач(N), то пустое слово является

наибольшим началом входа, являющимся K-началом. Если `Next` принадлежит одному (и, следовательно, только одному) из этих множеств, то максимальное начало входа, являющееся K-началом, непусто и читается соответствующей процедурой. □

15.2.5. Используя сказанное, составить процедуру распознавания выражений для грамматики (пример 3, с. 248):

⟨выр⟩ → ⟨слаг⟩ ⟨оствыр⟩
⟨оствыр⟩ → `+` ⟨выр⟩
⟨оствыр⟩ →
⟨слаг⟩ → ⟨множ⟩ ⟨остслаг⟩
⟨остслаг⟩ → `*` ⟨слаг⟩
⟨остслаг⟩ →
⟨множ⟩ → `x`
⟨множ⟩ → `(` ⟨выр⟩ `)`

Решение. Эта грамматика не полностью подпадает под рассмотренные частные случаи: в правых частях есть комбинации терминалов и нетерминалов

`+` ⟨выр⟩

и группы из трёх символов

`(` ⟨выр⟩ `)`

В грамматике есть также несколько правил с одной левой частью и с правыми частями разного рода, например

⟨оствыр⟩ → `+` ⟨выр⟩
⟨оствыр⟩ →

Эти ограничения не являются принципиальными. Так, правило типа `K` → `L M N` можно было бы заменить на два правила `K` → `L Q` и `Q` → `M N`, терминальные символы в правой части — на нетерминалы (с единственным правилом замены на соответствующие терминалы). Несколько правил с одной левой частью и разнородными правыми также можно

свести к уже разобранному случаю: например,

$$\begin{aligned} \mathtt{K} &\to \mathtt{L}\ \ \mathtt{M}\ \ \mathtt{N} \\ \mathtt{K} &\to \mathtt{P}\ \ \mathtt{Q} \\ \mathtt{K} &\to \end{aligned}$$

можно заменить на правила

$$\begin{aligned} \mathtt{K} &\to \mathtt{K}_1 \\ \mathtt{K} &\to \mathtt{K}_2 \\ \mathtt{K} &\to \mathtt{K}_3 \\ \mathtt{K}_1 &\to \mathtt{L}\ \ \mathtt{M}\ \ \mathtt{N} \\ \mathtt{K}_2 &\to \mathtt{P}\ \ \mathtt{Q} \\ \mathtt{K}_3 &\to \end{aligned}$$

Но мы не будем этого делать — а сразу же запишем то, что получится, если подставить описания процедур для новых терминальных символов в места их использования. Например, для правила

$$\mathtt{K} \to \mathtt{L}\ \ \mathtt{M}\ \ \mathtt{N}$$

это даёт процедуру

```
procedure ReadK;
begin
| ReadL;
| if b then begin
| | ReadM;
| end;
| if b then begin
| | ReadN;
| end;
end;
```

Для её корректности надо, чтобы Посл(L) не пересекалось с Нач(MN) (которое равно Нач(M), если из M не выводится пустое слово, и равно объединению Нач(M) и Нач(N), если выводится), а также чтобы Посл(M) не пересекалось с Нач(N).

Аналогичным образом правила

$$\begin{aligned} \mathtt{K} &\to \mathtt{L}\ \ \mathtt{M}\ \ \mathtt{N} \\ \mathtt{K} &\to \mathtt{P}\ \ \mathtt{Q} \\ \mathtt{K} &\to \end{aligned}$$

приводят к процедуре

```
procedure ReadK;
begin
| if (Next принадлежит Нач(LMN)) then begin
| | ReadL;
| | if b then begin ReadM; end;
| | if b then begin ReadN; end;
| end else if (Next принадлежит Нач(PQ)) then begin
| | ReadP;
| | if b then begin ReadQ; end;
| end else begin
| | b := true;
| end;
end;
```

корректность которой требует, чтобы Нач(LMN) не пересекалось с Нач(PQ).

Читая приведённую далее программу, полезно иметь в виду соответствие между русскими и английскими словами:

ВЫРажение	EXPRession
ОСТаток ВЫРажения	REST of EXPRession
СЛАГаемое	ADDitive term
ОСТаток СЛАГаемого	REST of ADDitive term
МНОЖитель	FACTor

```
procedure ReadSymb (c: Symbol);
| b := (Next = c);
| if b then begin
| | Move;
| end;
end;

procedure ReadExpr;
| ReadAdd;
| if b then begin ReadRestExpr; end;
end;

procedure ReadRestExpr;
| if Next = '+' then begin
| | ReadSymb ('+');
| | if b then begin ReadExpr; end;
| end else begin
```

```
  b := true;
 end;
end;

procedure ReadAdd;
 ReadFact;
 if b then begin ReadRestAdd; end;
end;

procedure ReadRestAdd;
 if Next = '*' then begin
  ReadSymb ('*');
  if b then begin ReadAdd; end;
 end else begin
  b := true;
 end;
end;

procedure ReadFact;
 if Next = 'x' then begin
  ReadSymb ('x');
 end else if Next = '(' then begin
  ReadSymb ('(');
  if b then begin ReadExpr; end;
  if b then begin ReadSymb (')'); end;
 end else begin
  b := false;
 end;
end;
```

Осталось обсудить проблемы, связанные с взаимной рекурсивностью этих процедур (одна использует другую и наоборот). В паскале это допускается, только требуется дать предварительное описание процедур («forward»). Как всегда для рекурсивных процедур, помимо доказательства того, что каждая процедура работает правильно в предположении, что используемые в ней вызовы процедур работают правильно, надо доказать отдельно, что работа завершается. (Это не очевидно: если в грамматике есть правило $\mathtt{K} \to \mathtt{KK}$, то из K ничего не выводится, Посл(K) и Нач(K) пусты, но написанная по нашим канонам процедура

```
procedure ReadK;
begin
 ReadK;
 if b then begin
```

```
|  | ReadK;
|  end;
end;
```

не заканчивает работы.)

В данном случае процедуры `ReadRestExpr`, `ReadRestAdd`, `ReadFact` либо завершаются, либо уменьшают длину непрочитанной части входа. Поскольку любой цикл вызовов включает одну из них, то зацикливание невозможно. □

15.2.6. Пусть в грамматике имеются два правила с нетерминалом K в левой части, имеющих вид

$$\mathtt{K} \to \mathtt{L}\ \mathtt{K}$$
$$\mathtt{K} \to$$

по которым K-слово представляет собой конечную последовательность L-слов, причём множества Посл(L) и Нач(K) (в данном случае равное Нач(L)) не пересекаются. Используя корректную для L процедуру `ReadL`, написать корректную для K процедуру `ReadK`, не используя рекурсии.

Решение. По нашим правилам следовало бы написать

```
procedure ReadK;
begin
|  if (Next принадлежит Нач(L)) then begin
|  |  ReadL;
|  |  if b then begin ReadK; end;
|  end else begin
|  |  b := true;
|  end;
end;
```

завершение работы гарантируется тем, что перед рекурсивным вызовом длина непрочитанной части уменьшается.

Эта рекурсивная процедура эквивалентна нерекурсивной:

```
procedure ReadK;
begin
|  b := true;
|  while b and (Next принадлежит Нач(L)) do begin
|  |  ReadL;
|  end;
end;
```

Формально можно проверить эту эквивалентность так. Завершаемость в обоих случаях ясна. Достаточно проверить поэтому, что тело рекурсивной процедуры эквивалентно нерекурсивной в предположении, что её рекурсивный вызов эквивалентен вызову нерекурсивной процедуры. Подставим:

```
if (Next принадлежит Нач(L)) then begin
| ReadL;
| if b then begin
| | b := true;
| | while b and (Next принадлежит Нач(L)) do begin
| | | ReadL;
| | end;
| end;
end else begin
| b := true;
end;
```

Первую команду `b:=true` можно выкинуть (в этом месте и так `b` истинно). Вторую команду можно перенести в начало:

```
b := true;
if (Next принадлежит Нач(L) then begin
| ReadL;
| if b then begin
| | while b and (Next принадлежит Нач(L)) do begin
| | | ReadL;
| | end;
| end;
end;
```

Теперь внутренний `if` можно выкинуть (если `b` ложно, цикл `while` всё равно не выполняется) и добавить в условие внешнего `if` условие `b` (которое всё равно истинно).

```
b := true;
if b and (Next принадлежит Нач(L)) then begin
| ReadL;
| while b and (Next принадлежит Нач(L)) do begin
| | ReadL;
| end;
end;
```

что эквивалентно приведённой выше нерекурсивной процедуре (из которой вынесена первая итерация цикла). □

15.2.7. Доказать корректность приведённой выше нерекурсивной программы непосредственно, без ссылок на рекурсивную.

Решение. Рассмотрим наибольшее начало входа, являющееся K-началом. Оно представляется в виде конкатенации (последовательного приписывания) нескольких непустых L-слов и, возможно, одного непустого L-начала, не являющегося L-словом. Инвариант цикла: прочитано несколько из них; b ⇔ (последнее прочитанное является L-словом).

Сохранение инварианта: если осталось последнее слово, это очевидно; если осталось несколько, то за первым L-словом (из числа оставшихся) идёт символ из Нач(L), и потому это слово — максимальное начало входа, являющееся L-началом. □

На практике при записи грамматики используют сокращения. Если правила для какого-то нетерминала K имеют вид

$$\mathtt{K} \to \mathtt{L}\ \mathtt{K}$$
$$\mathtt{K} \to$$

(т. е. K-слова — это последовательности L-слов), то этих правил не пишут, а вместо K пишут L в фигурных скобках. Несколько правил с одной левой частью и разными правыми записывают как одно правило, разделяя альтернативные правые части вертикальной чертой.

Например, рассмотренная выше грамматика для ⟨выр⟩ может быть записана так:

$$\langle\text{выр}\rangle \to \langle\text{слаг}\rangle\ \{\ \mathtt{+}\ \langle\text{слаг}\rangle\ \}$$
$$\langle\text{слаг}\rangle \to \langle\text{множ}\rangle\ \{\ \mathtt{*}\ \langle\text{множ}\rangle\ \}$$
$$\langle\text{множ}\rangle \to \mathtt{x} \mid (\ \langle\text{выр}\rangle\ )$$

15.2.8. Написать процедуру, корректную для ⟨выр⟩, следуя этой грамматике и используя цикл вместо рекурсии, где можно.

Решение.

```
procedure ReadSymb (c: Symbol);
| b := (Next = c);
| if b then begin Move; end;
end;

procedure ReadExpr;
begin
| ReadAdd;
```

```
  while b and (Next = '+') do begin
  | Move; ReadAdd;
  end;
end;

procedure ReadAdd;
begin
  ReadFact;
  while b and (Next = '*') do begin
  | Move; ReadFact;
  end;
end;

procedure ReadFact;
begin
  if Next = 'x' do begin
  | Move; b := true;
  end else if Next = '(' then begin
  | Move; ReadExpr;
  | if b then begin ReadSymb (')'); end;
  end else begin
  | b := false;
  end;
end;
```

□

15.2.9. В последней процедуре команду `b:=true` можно опустить. Почему?

Решение. Можно предполагать, что все процедуры вызываются при `b=true`. □

15.3. Алгоритм разбора для LL(1)-грамматик

В этом разделе мы рассмотрим ещё один метод проверки выводимости в КС-грамматике, называемый по традиции LL(1)-разбором. Вот его идея в одной фразе: можно считать, что в процессе вывода мы всегда заменяем самый левый нетерминал и нужно лишь выбрать одно из правил; если нам повезёт с грамматикой, то выбрать правило можно, глядя на первый символ выводимого из этого нетерминала слова. Говоря более формально, дадим такое

Определение. *Левым выводом* (слова в грамматике) называется вывод, в котором на каждом шаге замене подвергается самый левый из нетерминалов.

15.3.1. Для каждого выводимого слова (из терминалов) существует его левый вывод.

Решение. Различные нетерминалы заменяются независимо; если в процессе вывода появилось слово $\ldots K \ldots L \ldots$, где K, L — нетерминалы, то замены K и L можно производить в любом порядке. Поэтому можно перестроить вывод так, чтобы стоящий левее нетерминал заменялся раньше. (Формально говоря, надо доказывать индукцией по длине вывода такой факт: если из некоторого нетерминала K выводится некоторое слово A, то существует левый вывод A из K.) □

15.3.2. В грамматике с 4 правилами

(1) `E` →
(2) `E` → `TE`
(3) `T` → `(E)`
(4) `T` → `[E]`

найти левый вывод слова $A =$ `[()([])]` и доказать, что он единствен.

Решение. На первом шаге можно применить только правило (2):

`E` → `TE`

Что будет дальше с `T`? Так как слово A начинается на `[`, то может примениться только правило (4):

`E` → `TE` → `[E]E`

Первое `E` должно замениться на `TE` (иначе вторым символом была бы скобка `]`):

`E` → `TE` → `[E]E` → `[TE]E`

и `T` должно заменяться по (3):

`E` → `TE` → `[E]E` → `[TE]E` → `[(E)E]E`

Далее первое `E` должно замениться на пустое слово (иначе третьей буквой слова будет `(` или `[` — только на эти символы может начинаться слово, выводимое из `T`):

`E` → `TE` → `[E]E` → `[TE]E` → `[(E)E]E` → `[()E]E`

и далее

$\ldots$ → `[()TE]E` → `[()(E)E]E` → `[()(TE)E]E` → `[()([E]E)E]E` →
→ `[()([]E)E]E` → `[()([])E]E` → `[()([])]E` → `[()([])]` □

Что требуется от грамматики, чтобы такой метод поиска левого вывода был применим? Пусть, например, на очередном шаге самым левым нетерминалом оказался нетерминал K, т. е. мы имеем слово вида AKU, где A — слово из терминалов, а U — слово из терминалов и нетерминалов. Пусть в грамматике есть правила

$$K \to LMN$$
$$K \to PQ$$
$$K \to R$$

Нам надо выбрать одно из них. Мы будем пытаться сделать этот выбор, глядя на первый символ той части входного слова, которая выводится из KU.

Рассмотрим множество Нач(LMN) тех терминалов, с которых начинаются непустые слова, выводимые из LMN. (Это множество равно Нач(L), объединённому с Нач(M), если из L выводится пустое слово, а также с Нач(N), если из L и из M выводится пустое слово.) Чтобы описанный метод был применим, надо, чтобы Нач(LMN), Нач(PQ) и Нач(R) не пересекались. Но этого мало. Ведь может быть так, например, что из LMN будет выведено пустое слово, а из слова U будет выведено слово, начинающееся на букву из Нач(PQ). Следующие определения учитывают эту проблему.

Напомним, что определение выводимости в КС-грамматике было дано только для слова из терминалов. Оно очевидным образом обобщается на случай слов из терминалов и нетерминалов. Можно также говорить о выводимости одного слова (содержащего терминалы и нетерминалы) из другого. (Если говорится о выводимости слова без указания того, откуда оно выводится, то всегда подразумевается выводимость в грамматике, т. е. выводимость из начального нетерминала.)

Для каждого слова X из терминалов и нетерминалов через Нач(X) обозначаем множество всех терминалов, с которых начинаются непустые слова из терминалов, выводимые из X. (В случае, если из любого нетерминала выводится хоть одно слово из терминалов, не играет роли, рассматриваем ли мы при определении Нач(X) слова только из терминалов или любые слова. Мы будем предполагать далее, что это условие выполнено.)

Для каждого нетерминала K через Послед(K) обозначим множество терминалов, которые встречаются в выводимых (в грамматике) словах сразу же за K. (Не смешивать с Посл(K) предыдущего раздела!) Кроме того, в Послед(K) включается символ `EOI`, если существует выводимое слово, оканчивающееся на K.

Для каждого правила

$$K \to V$$

(где K — нетерминал, V — слово, содержащее терминалы и нетерминалы) определим множество *направляющих терминалов*, обозначаемое Напр($K \to V$). По определению оно равно Нач(V), к которому добавлено Послед(K), если из V выводится пустое слово.

Определение. Грамматика называется LL(1)-*грамматикой*, если для любых правил $K \to V$ и $K \to W$ с одинаковыми левыми частями множества Напр($K \to V$) и Напр($K \to W$) не пересекаются.

15.3.3. Является ли грамматика

```
K → K #
K →
```

(выводимыми словами являются последовательности диезов) LL(1)-грамматикой?

Решение. Нет: символ `#` принадлежит множествам направляющих символов для обоих правил (для второго — поскольку `#` принадлежит Послед(K)). □

15.3.4. Написать LL(1)-грамматику для того же языка.

Решение.

```
K → # K
K →
```

□

Как говорят, «леворекурсивное» правило заменено на «праворекурсивное».

Следующая задача показывает, что для LL(1)-грамматики существует не более одного возможного продолжения левого вывода.

15.3.5. Пусть дано выводимое в LL(1)-грамматике слово X, в котором выделен самый левый нетерминал K: $X = AKS$, где A — слово из терминалов, S — слово из терминалов и нетерминалов. Пусть существуют два различных правила грамматики с нетерминалом K в левой части, и мы применили их к выделенному в X нетерминалу K, затем продолжили вывод и в конце концов получили два слова из терминалов, начинающихся на A. Доказать, что в этих словах за началом A идут разные буквы. (Здесь к числу букв мы относим `EOI`.)

Решение. Эти буквы принадлежат направляющим множествам различных правил. □

15.3.6. Доказать, что если слово выводимо в LL(1)-грамматике, то его левый вывод единствен.

Решение. Предыдущая задача показывает, что на каждом шаге левый вывод продолжается однозначно. □

15.3.7. Грамматика называется *леворекурсивной*, если из некоторого нетерминала K выводится слово, начинающееся с K, но не совпадающее с ним. Доказать, что леворекурсивная грамматика, в которой из каждого нетерминала выводится хотя бы одно непустое слово из терминалов и для каждого нетерминала существует вывод (начинающийся с начального нетерминала), в котором он встречается, не является LL(1)-грамматикой.

Решение. Пусть из K выводится KU, где K — нетерминал, а U — непустое слово. Можно считать, что это левый вывод (другие нетерминалы можно не заменять). Рассмотрим вывод

$$K \rightsquigarrow KU \rightsquigarrow KUU \rightsquigarrow \dots$$

(знак $\rightsquigarrow$ обозначает несколько шагов вывода) и левый вывод $K \rightsquigarrow A$, где A — непустое слово из терминалов. На каком-то шаге второй вывод отклоняется от первого, а между тем по обоим путям может быть получено слово, начинающееся на A (в первом случае это возможно, так как сохраняется нетерминал K, который может впоследствии быть заменён на A). Это противоречит возможности однозначного определения правила, применяемого на очередном шаге поиска левого вывода. (Однозначность выполняется для выводов из начального нетерминала, и надо воспользоваться тем, что K по предположению встречается в таком выводе.) □

Таким образом, к леворекурсивным грамматикам (кроме тривиальных случаев) LL(1)-метод неприменим. Их приходится преобразовывать в эквивалентные LL(1)-грамматики — или пользоваться другими методами распознавания.

15.3.8. Используя сказанное, построить алгоритм проверки выводимости слова из терминалов в LL(1)-грамматике.

Решение. Мы следуем описанному выше методу поиска левого вывода, храня лишь часть слова, находящуюся правее уже прочитанной

части входного слова. Другими словами, мы храним слово S из терминалов и нетерминалов, обладающее такими свойствами (прочитанную часть входа обозначаем через A):

1) слово AS выводимо в грамматике;

2) любой левый вывод входного слова проходит через стадию AS

Эти свойства вместе будем обозначать «(И)».

Вначале A пусто, а S состоит из единственного символа — начального нетерминала.

Если в некоторый момент S начинается на терминал t и t = Next, то можно выполнить команду Move и удалить символ t, являющийся начальным в S, поскольку при этом AS не меняется.

Если S начинается на терминал t и t ≠ Next, то входное слово невыводимо — ибо по условию любой его вывод должен проходить через AS. (Это же справедливо и в случае Next = EOI.)

Если S пусто, то из условия (И) следует, что входное слово выводимо тогда и только тогда, когда Next = EOI.

Остаётся случай, когда S начинается с некоторого нетерминала K. По доказанному выше все левые выводы из S слов, начинающихся на символ Next, начинаются с применения к S одного и того же правила — того, для которого Next принадлежит направляющему множеству. Если таких правил нет, то входное слово невыводимо. Если такое правило есть, то нужно применить его к первому символу слова S — при этом свойство (И) не нарушится. Приходим к такому алгоритму:

```
s := пустое слово;
error := false;
{error => входное слово невыводимо;}
{not error => (И)}
while (not error) and not ((Next=EOI) and (S пусто))
|   do begin
|  if (S начинается на терминал, равный Next) then begin
|  | Move; удалить из S первый символ;
|  end else if (S начинается на терминал, не равный Next)
|  |           then begin
|  | error := true;
|  end else if (S пусто) and (Next <> EOI) then begin
|  | error := true;
|  end else if (S начинается на нетерминал и Next входит в
|  |    направляющее множество одного из правил для этого
|  |    нетерминала) then begin
|  | применить это правило
```

```
    end else if (S начинается на нетерминал и Next не входит
      в направляющее множество ни одного из правил для этого
      нетерминала) then begin
      error := true;
    end else begin
      {так не бывает}
    end;
  end;
  {входное слово выводимо <=> not error}
```

Алгоритм заканчивает работу, поскольку при появлении терминала в начале слова S происходит чтение со входа или остановка, а бесконечный цикл сменяющих друг друга нетерминалов в начале S означал бы, что грамматика леворекурсивна. (А мы можем предполагать, согласно предыдущей задаче, что это не так: нетерминалы, не встречающиеся в выводах, а также нетерминалы, из которых не выводится непустого слова, несложно удалить из грамматики.) □

Замечания.

- Приведённый алгоритм использует S как стек (все действия производятся с левого конца).
- Действия двух последних вариантов внутри цикла не приводят к чтению очередного символа со входа, поэтому их можно заранее предвычислить для каждого нетерминала и каждого символа Next. После этого на каждом шаге цикла будет читаться очередной символ входа.
- При практической реализации удобно составить таблицу, в которой записаны варианты действий в зависимости от входного символа и первого символа S, и небольшую программу, выполняющую действия в соответствии с этой таблицей.

15.3.9. При проверке того, относится ли данная грамматика к типу LL(1), необходимо вычислить Послед(T) и Нач(T) для всех нетерминалов T. Как это сделать?

Решение. Пусть, например, в грамматике есть правило $K \to L\,M\,N$. Тогда

$$\begin{aligned}
&\text{Нач}\,(L) \subset \text{Нач}\,(K), && \\
&\text{Нач}\,(M) \subset \text{Нач}\,(K), && \text{если из } L \text{ выводимо пустое слово,} \\
&\text{Нач}\,(N) \subset \text{Нач}\,(K), && \text{если из } L \text{ и } M \text{ выводимо пустое слово,}
\end{aligned}$$

$$
\begin{aligned}
&\text{Послед}\,(K) \subset \text{Послед}\,(N),\\
&\text{Послед}\,(K) \subset \text{Послед}\,(M), && \text{если из } N \text{ выводимо пустое слово,}\\
&\text{Послед}\,(K) \subset \text{Послед}\,(L), && \text{если из } M \text{ и } N \text{ выводимо пустое слово,}\\
&\text{Нач}\,(N) \subset \text{Послед}\,(M),\\
&\text{Нач}\,(M) \subset \text{Послед}\,(L),\\
&\text{Нач}\,(N) \subset \text{Послед}\,(L), && \text{если из } M \text{ выводимо пустое слово.}
\end{aligned}
$$

Подобные правила позволяют шаг за шагом порождать множества Нач(T), а затем и Послед(T), для всех терминалов и нетерминалов T. При этом началом служит

$$\texttt{EOI} \in \text{Послед}\,(K)$$

для начального нетерминала K и

$$z \in \text{Нач}\,(z)$$

для любого терминала z. Порождение заканчивается, когда применение правил перестаёт давать новые элементы множеств Нач(T) и Послед(T). □

16. СИНТАКСИЧЕСКИЙ РАЗБОР СЛЕВА НАПРАВО (LR)

Сейчас мы рассмотрим ещё один метод синтаксического разбора, называемый LR(1)-разбором, а также некоторые упрощённые его варианты.

16.1. LR-процессы

Два отличия LR(1)-разбора от LL(1)-разбора: во-первых, строится не левый вывод, а правый, во-вторых, он строится не с начала, а с конца. (Вывод в КС-грамматике называется *правым*, если на каждом шаге замене подвергается самый правый нетерминал.)

16.1.1. Доказать, что если слово, состоящее из терминалов, выводимо, то оно имеет правый вывод. □

Нам будет удобно смотреть на правый вывод «задом наперёд». Определим понятие *LR-процесса над словом* A. В этом процессе, помимо A, будет участвовать и другое слово S, которое может содержать как терминалы, так и нетерминалы. Вначале слово S пусто. В ходе LR-процесса разрешены два вида действий:

(1) можно перенести первый символ слова A (его называют *очередным* символом и обозначают `Next`) в конец слова S, удалив его из A (это действие называют *сдвигом*);

(2) если правая часть одного из правил грамматики оказалась концом слова S, то разрешается заменить её на нетерминал, стоящий в левой части этого правила; при этом слово A не меняется. (Это действие называют *свёрткой*, или *приведением*.)

Отметим, что LR-процесс не является детерминированным: в одной и той же ситуации могут быть разрешены разные действия.

Говорят, что LR-процесс на слове A *успешно завершается*, если слово A становится пустым, а в слове S остаётся единственный нетерминал — начальный нетерминал грамматики.

16.1.2. Доказать, что для любого слова A (из терминалов) успешно завершающийся LR-процесс существует тогда и только тогда, когда слово A выводимо в грамматике. В ходе доказательства установить взаимно однозначное соответствие между правыми выводами и успешно завершающимися LR-процессами.

Решение. При сдвиге слово SA не меняется, при свёртке слово SA подвергается преобразованию, обратному шагу вывода. Этот вывод будет правым, так как сворачивается конец S, а в A все символы — терминальные. Таким образом, каждому LR-процессу соответствует правый вывод. Обратное соответствие: пусть дан правый вывод. Представим себе, что за последним нетерминалом в слове стоит перегородка. Применив к этому нетерминалу правило грамматики, мы должны сдвинуть перегородку влево (если правая часть правила кончается на терминал). Разбивая этот сдвиг на отдельные шаги, получим процесс, в точности обратный LR-процессу. □

Поскольку в ходе LR-процесса все изменения в слове S происходят с правого конца, слово S называют *стеком* LR-процесса.

Задача построения правого вывода для данного слова сводится, таким образом, к правильному выбору очередного шага LR-процесса. Нам нужно решить, будем ли мы делать сдвиг или свёртку, и если свёртку, то по какому правилу — ведь подходящих правил может быть несколько. В LR(1)-алгоритме это решение принимается на основе S и первого символа слова A; если используется только S, то говорят о LR(0)-алгоритме. (Точные определения смотри ниже.)

Пусть фиксирована грамматика, в которой из любого нетерминала можно вывести какое-либо слово из терминалов. (Это ограничение мы будет всегда предполагать выполненным.)

Пусть $\mathtt{K} \to \mathtt{U}$ — одно из правил грамматики (K — нетерминал, U — слово из терминалов и нетерминалов). Определим множество слов (из терминалов и нетерминалов), называемое *левым контекстом* правила $\mathtt{K} \to \mathtt{U}$. (Обозначение: ЛевКонт($\mathtt{K} \to \mathtt{U}$).) По определению в него входят все слова, которые являются содержимым стека непосредственно перед свёрткой U в K в ходе некоторого успешно завершающегося LR-процесса.

16.1.3. Переформулировать это определение на языке правых выводов.

Решение. Рассмотрим все правые выводы вида

$$\langle\text{начальный нетерминал}\rangle \rightsquigarrow \mathtt{XKA} \to \mathtt{XUA},$$

где A — слово из терминалов, X — слово из терминалов и нетерминалов. Все возникающие при этом слова XU и образуют левый контекст правила K → U. Чтобы убедиться в этом, следует вспомнить, что мы предполагаем, что из любого нетерминала можно вывести какое-то слово из терминалов, так что правый вывод слова XUA может быть продолжен до правого вывода какого-то слова из терминалов. □

16.1.4. Все слова из ЛевКонт(K → U) кончаются, очевидно, на U. Доказать, что если у всех них этот конец U отбросить, то полученное множество слов не зависит от того, какое из правил для нетерминала K выбрано. (Это множество обозначается Лев(K).)

Решение. Из предыдущей задачи ясно, что Лев(K) — это всё, что может появиться в правых выводах левее самого правого нетерминала K. □

16.1.5. Доказать, что в предыдущей фразе можно отбросить слова «самого правого»: Лев(K) — это всё то, что может появляться в правых выводах левее любого вхождения нетерминала K.

Решение. Продолжив построение правого вывода, все нетерминалы справа от K можно заменить на терминалы (а слева от K при этом ничего не изменится). □

16.1.6. Построить грамматику, содержащую для каждого нетерминала K исходной грамматики нетерминал ⟨ЛевK⟩, причём следующее свойство должно выполняться для любого нетерминала K исходной грамматики: в новой грамматике из ⟨ЛевK⟩ выводимы все элементы Лев(K) и только они. (При этом терминалы и нетерминалы исходной грамматики являются терминалами новой.)

Решение. Пусть P — начальный нетерминал грамматики. Тогда в новой грамматике будет правило

$$\langle\text{ЛевP}\rangle \to \qquad \text{(пустое слово)}$$

Для каждого правила исходной грамматики, например, правила

$$\mathtt{K} \to \mathtt{L\,t\,M\,N} \qquad (\mathtt{L},\ \mathtt{M},\ \mathtt{N} \text{ — нетерминалы, } \mathtt{t} \text{ — терминал}),$$

в новую грамматику мы добавим правила

$$\langle\text{ЛевL}\rangle \to \langle\text{ЛевK}\rangle$$
$$\langle\text{ЛевM}\rangle \to \langle\text{ЛевK}\rangle\,\texttt{L t}$$
$$\langle\text{ЛевN}\rangle \to \langle\text{ЛевK}\rangle\,\texttt{L t M}$$

и аналогично поступим с другими правилами. Смысл новых правил таков: пустое слово может появиться слева от P; если слово X может появиться слева от K, то X может появиться слева от L, XLt может появиться слева от M, XLtM — слева от N. Индукцией по длине правого вывода легко проверить, что всё, что может появиться слева от какого-то нетерминала, появляется в соответствии с этими правилами. □

16.1.7. Почему в предыдущей задаче важно, что мы рассматриваем только правые выводы?

Ответ. В противном случае следовало бы учитывать преобразования, происходящие внутри слова, стоящего слева от K. □

16.1.8. Для данной грамматики построить алгоритм, который по любому слову выясняет, каким из множеств Лев(K) оно принадлежит.

(Замечание для знатоков. Существование такого алгоритма — и даже конечного автомата, то есть индуктивного расширения с конечным числом значений, см. раздел 1.3, — вытекает из предыдущей задачи, так как построенная в ней грамматика имеет специальный вид: в правых частях всего один нетерминал, причём он стоит у левого края. Тем не менее мы приведём явное построение.)

Решение. Будем называть *ситуацией* данной грамматики одно из её правил, в правой части которого отмечена одна из позиций (до первой буквы, между первой и второй буквой, ... , после последней буквы). Например, правило

$$\texttt{K} \to \texttt{L t M N}$$

(K, L, M, N — нетерминалы, t — терминал) порождает пять ситуаций

$$\texttt{K} \to \texttt{\_L t M N} \quad \texttt{K} \to \texttt{L \_t M N} \quad \texttt{K} \to \texttt{L t \_M N} \quad \texttt{K} \to \texttt{L t M \_N} \quad \texttt{K} \to \texttt{L t M N \_}$$

(позиция указывается знаком подчёркивания).

Будем говорить, что слово S *согласовано* с ситуацией $\texttt{K} \to \texttt{U \_V}$, если S кончается на U, то есть $\texttt{S} = \texttt{TU}$ при некотором T, и, кроме того, T принадлежит Лев(K). (Смысл этого определения примерно таков: в стеке S

подготовлена часть U для будущей свёртки UV в K.) В этих терминах ЛевКонт(K → X) — это множество всех слов, согласованных с ситуацией K → X_, а Лев(K) — это множество всех слов, согласованных с ситуацией K → _X (где K → X — любое правило для нетерминала K).

Эквивалентное определение в терминах LR-процесса: S согласовано с ситуацией K → U_V, если существует успешный LR-процесс, в котором события развиваются так:

- в ходе процесса в стеке появляется слово S, и оно оканчивается на U;
- некоторое время S не затрагивается, а справа от него появляется V;
- UV сворачивается в K;
- процесс продолжается и успешно завершается.

16.1.9. Доказать эквивалентность этих определений.

[Указание. Если S = TU и T принадлежит Лев(K), то можно получить в стеке сначала T, потом U, потом V, потом свернуть UV в K и затем успешно завершить процесс. (Мы используем несколько раз тот факт, что из любого нетерминала что-то да выводится: благодаря этому мы можем добавить в стек любое слово.)] □

Наша цель — построение алгоритма, распознающего принадлежность произвольного слова к Лев(K). Рассмотрим функцию, сопоставляющую с каждым словом S (из терминалов и нетерминалов) множество всех согласованных с ним ситуаций. Это множество назовём *состоянием, соответствующим слову S*. Будем обозначать его Сост(S). Достаточно показать, что функция Сост(S) индуктивна, то есть что значение Сост(SJ), где J — терминал или нетерминал, может быть вычислено, если известно Сост(S) и символ J. (Мы видели ранее, как принадлежность к Лев(K) выражается в терминах этой функции.) Значение Сост(SJ) вычисляется по таким правилам:

(1) Если слово S согласовано с ситуацией K → U_V, причём слово V начинается на букву J, то есть V = JW, то SJ согласовано с ситуацией K → UJ_W.

Это правило полностью определяет все ситуации с непустой левой половиной (то есть не начинающиеся с подчёркивания), согласованные с SJ. Осталось определить, для каких нетерминалов K слово SJ принадлежит Лев(K). Это делается по двум правилам:

> (2) Если ситуация L → U_V согласована с SJ (согласно правилу (1)), а V начинается на нетерминал K, то SJ принадлежит Лев(K).
>
> (3) Если SJ входит в Лев(L) для некоторого L, причём L → → V — правило грамматики и V начинается на нетерминал K, то SJ принадлежит Лев(K).

Заметим, что правило (3) можно рассматривать как аналог правила (2): в указанных в (3) предположениях ситуация L → _V согласована с SJ, а V начинается на нетерминал K.

Корректность этих правил в общем-то очевидна, если хорошенько подумать. Единственное, что требует некоторых пояснений — это то, почему с помощью правил (2) и (3) обнаружатся *все* терминалы K, для которых SJ принадлежит Лев(K). Попытаемся это объяснить. Рассмотрим правый вывод, в котором SJ стоит слева от K. Откуда мог взяться в нём нетерминал K? Если правило, которое его породило, породило также и конец слова SJ, то принадлежность SJ к Лев(K) будет обнаружена по правилу (2). Если же K было первой буквой слова, порождённого каким-то другим нетерминалом L, то — благодаря правилу (3) — достаточно установить принадлежность SJ к Лев(L). Осталось применить те же рассуждения к L и так далее.

В терминах LR-процесса то же самое можно сказать так. Сначала нетерминал K может участвовать в нескольких свёртках, не затрагивающих SJ (они соответствуют применению правила (3)), но затем он обязан подвергнуться свёртке, затрагивающей SJ (что соответствует применению правила (2)).

Осталось выяснить, какие ситуации согласованы с пустым словом, то есть для каких нетерминалов K пустое слово принадлежит Лев(K). Это определяется по следующим правилам:

> (1) начальный нетерминал таков;
>
> (2) если K таков и K → V — правило грамматики, причём слово V начинается с нетерминала L, то и L таков. □

16.1.10. Проделать описанный анализ для грамматики

$$\begin{aligned}
&\mathtt{E} \to \mathtt{E+T}\\
&\mathtt{E} \to \mathtt{T}\\
&\mathtt{T} \to \mathtt{T*F}\\
&\mathtt{T} \to \mathtt{F}\\
&\mathtt{F} \to \mathtt{x}\\
&\mathtt{F} \to \mathtt{(E)}
\end{aligned}$$

(задающей тот же язык, что и грамматика примера 3, с. 248).

Решение. Множества Сост(S) для различных S приведены в таблице на с. 277. Знак равенства означает, что множества ситуаций, являющиеся значениями функции Сост(S) на словах, стоящих слева и справа от знака равенства, одинаковы.

Правило определения Сост(SJ), если известны Сост(S) и J (здесь S — слово из терминалов и нетерминалов, J — терминал или нетерминал), таково:

> надо найти Сост(S) в правой колонке, взять соответствующее ему слово T в левой колонке, приписать к нему J и взять множество, стоящее напротив слова TJ (если слово TJ в таблице отсутствует, то Сост(SJ) пусто). □

16.2. LR(0)-грамматики

Напомним, что наша основная цель — это поиск вывода заданного слова, или, другими словами, поиск успешного LR-процесса над ним. Во всех рассматриваемых нами грамматиках успешный LR-процесс (над данным словом) единствен. Искать этот единственный успешный процесс мы будем постепенно: в каждый момент мы смотрим, какой шаг возможен следующим. Для этого на грамматику надо наложить дополнительные требования, и сейчас мы рассмотрим простейший случай так называемых LR(0)-грамматик. Мы уже знаем:

> (1) В успешном LR-процессе возможна свёртка по правилу K → U при содержимом стека S тогда и только тогда, когда S принадлежит ЛевКонт(K → U) или, другими словами, когда слово S согласовано с ситуацией K → U_.

Слово S	Сост(S)
пустое	E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E	E → E_+T
T	E → T_ T → T_*F
F	T → F_
x	F → x_
(	F → (_E) E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E+	E → E+_T T → _T*F T → _F F → _x F → _(E)
T*	T → T*_F F → _x F → _(E)
(E	F → (E_) E → E_+T
(T	= T
(F	= F
(x	= x
((	= (
E+T	E → E+T_ T → T_*F
E+F	= F
E+x	= x
E+(	= (
T*F	T → T*F_
T*x	= x
T*(	= (
(E)	F → (E)_
(E+	= E+
E+T*	= T*

К задаче 16.1.10

Аналогичное утверждение про сдвиг гласит:

> (2) В успешном LR-процессе при содержимом стека S возможен сдвиг с очередным символом a тогда и только тогда, когда S согласовано с некоторой ситуацией K → U_aV.

16.2.1. Доказать это.

[Указание. Пусть произошёл сдвиг и к стеку S добавилась буква a. Рассмотрите первую свёртку, затрагивающую эту букву.] □

Рассмотрим некоторую грамматику и произвольное слово S из терминалов и нетерминалов. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал, то говорят, что *для слова S возможен сдвиг*. Если в Сост(S) есть ситуация, в которой справа от подчёркивания ничего нет, то говорят, что *для слова S возможна свёртка* (по соответствующему правилу). Говорят, что для слова S возникает *конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для слова S возникает *конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка.

Грамматика называется LR(0)-*грамматикой*, если в ней нет конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одного слова S.

16.2.2. Является ли приведённая выше грамматика LR(0)-грамматикой?

Решение. Нет, не является. Для слов T и E+T имеются конфликты типа сдвиг/свёртка. □

16.2.3. Являются ли LR(0)-грамматиками такие:

(а)
T → 0
T → T1
T → TT2
T → TTT3

(б)
T → 0
T → 1T
T → 2TT
T → 3TTT

Решение. Являются, см. таблицы на с. 279–280 (конфликтов нет). □

Эта задача показывает, что LR(0)-грамматики могут быть как леворекурсивными, так и праворекурсивными.

16.2.4. Пусть дана LR(0)-грамматика. Доказать, что у любого слова существует не более одного правого вывода. Построить алгоритм проверки выводимости в LR(0)-грамматике.

Слово S	Сост(S)
пустое	T → _0 T → _T1 T → _TT2 T → _TTT3
0	T → 0_
T	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
T1	T → T1_
TT	T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT2	T → TT2_
TTT	T → TTT_3 T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT0	= 0
TTT3	T → TTT3_
TTT2	= TT2
TTTT	= TTT
TTT0	= 0

(а)

Слово S	Сост(S)
пустое	T → _0 T → _1T T → _2TT T → _3TTT
0	T → 0_
1	T → 1_T
	T → _0 T → _1T T → _2TT T → _3TTT
2	T → 2_TT
	T → _0 T → _1T T → _2TT T → _3TTT
3	T → 3_TTT
	T → _0 T → _1T T → _2TT T → _3TTT

(б), начало

К задаче 16.2.3.

Слово S	Сост(S)
1T	T → 1T_
10	= 0
11	= 1
12	= 2
13	= 3
2T	T → 2T_T
	T → _0 T → _1T T → _2TT T → _3TTT
20	= 0
21	= 1
22	= 2
23	= 3
3T	T → 3T_TT
	T → _0 T → _1T T → _2TT T → _3TTT
30	= 0
31	= 1
32	= 2
33	= 3
2TT	T → 2TT_
2T0	= 0
2T1	= 1
2T2	= 2
2T3	= 3
3TT	T → 3TT_T
	T → _0 T → _1T T → _2TT T → _3TTT
3T0	= 0
3T1	= 1
3T2	= 2
3T3	= 3
3TTT	T → 3TTT_
3TT0	= 0
3TT1	= 1
3TT2	= 2
3TT3	= 3

(б), окончание

Решение. Пусть дано произвольное слово. Будем строить LR-процесс над ним по шагам. Пусть текущее состояние стека LR-процесса равно S. Нам надо решить, делать сдвиг или свёртку (и если свёртку, то по какому правилу). Согласно определению LR(0)-грамматики, в нашем состоянии S возможен либо только сдвиг, либо только свёртка (причём лишь по одному правилу). Таким образом, поиск возможных продолжений LR-процесса происходит детерминированно (на каждом шаге можно определить, какое действие только и возможно). □

16.2.5. Что произойдёт, если анализируемое слово не имеет вывода в данной грамматике?

Ответ. Либо на некотором шаге не будет возможен ни сдвиг, ни свёртка, либо все возможные сдвиги будет иметь неподходящий очередной символ. □

Замечания. 1. При реализации этого алгоритма нет необходимости каждый раз заново вычислять множество Сост(S) для текущего значения S. Эти множества можно также хранить в стеке (в каждый момент хранятся множества Сост(T) для всех начал T текущего слова S).

2. На самом деле само слово S можно не хранить — достаточно хранить множества ситуаций Сост(T) для всех его начал T (включая само S).

В алгоритме проверки выводимости в LR(0)-грамматике мы используем не всю информацию, которую могли бы. В этом алгоритме для каждого состояния известно заранее, что в нём возможен только сдвиг или только свёртка (причём в последнем случае известно, по какому правилу). Более изощрённый алгоритм мог бы принимать решение о выборе между сдвигом и свёрткой, посмотрев на очередной символ (Next). Глядя на состояние, можно сказать, при каких значениях Next возможен сдвиг (это те терминалы, которые в ситуациях этого состояния стоят непосредственно за подчёркиванием). Сложнее воспользоваться информацией о символе Next для решения вопроса о том, возможна ли свёртка. Для этого есть упрощённый метод (грамматики, к которым он применим, называют SLR(1)-грамматиками [сокращение от Simple LR(1)]) и полный метод (более сложный, но использующий всю возможную информацию; грамматики, к которым он применим, называют LR(1)-грамматиками). Есть и промежуточный класс грамматик, называемый LALR(1).

16.3. SLR(1)-грамматики

Напомним, что для любого нетерминала K мы определяли (с. 264) множество Послед(K) тех терминалов, которые могут стоять непосредственно за K в выводимом (из начального нетерминала) слове; в это множество добавляют также символ EOI, если нетерминал K может стоять в конце выводимого слова.

16.3.1. Доказать, что если в данный момент LR-процесса последний символ стека S равен K, причём процесс этот может в дальнейшем успешно завершиться, то Next принадлежит Послед(K).

Решение. Этот факт является непосредственным следствием определения (вспомним соответствие между правыми выводами и LR-процессами). □

Рассмотрим некоторую грамматику, произвольное слово S из терминалов и нетерминалов и терминал x. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал x, то говорят, что *для пары* $\langle$S, x$\rangle$ *возможен сдвиг*. Если в Сост(S) есть ситуация K → U_, причём x принадлежит Послед(K), то говорят, что *для пары* $\langle$S, x$\rangle$ SLR(1)-*возможна свёртка* (по правилу K → U). Говорят, что для пары $\langle$S, x$\rangle$ возникает SLR(1)-*конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для пары $\langle$S, x$\rangle$ возникает SLR(1)-*конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка.

Грамматика называется SLR(1)-*грамматикой*, если в ней нет SLR(1)-конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одной пары $\langle$S, x$\rangle$.

16.3.2. Пусть дана SLR(1)-грамматика. Доказать, что у любого слова существует не более одного правого вывода. Построить алгоритм проверки выводимости в SLR(1)-грамматике.

Решение. Аналогично случаю LR(0)-грамматик, только при выборе между сдвигом и свёрткой учитывается очередной символ (Next). □

16.3.3. Проверить, является ли приведённая выше на с. 276 грамматика (с нетерминалами E, T и F) SLR(1)-грамматикой.

Решение. Да, является, так как оба конфликта, мешающие ей быть LR(0)-грамматикой, разрешаются с учётом очередного символа: и для слова T, и для слова E+T сдвиг возможен только при Next = *, а символ * не принадлежит ни Послед(E) = {EOI, +,)}, ни Послед(T) = {EOI, +, *,)}, и поэтому при Next = * свёртка невозможна. □

16.4. LR(1)-грамматики, LALR(1)-грамматики

Описанный выше SLR(1)-подход используют не всю возможную информацию при выяснении того, возможна ли свёртка. Именно, он отдельно проверяет, возможна ли свёртка при данном состоянии стека `S` и отдельно — возможна ли свёртка по данному правилу при данном символе `Next`. Между тем эти проверки не являются независимыми: обе могут дать положительный ответ, но тем не менее свёртка при стеке `S` и очередном символе `Next` невозможна. В LR(1)-подходе этот недостаток устраняется.

LR(1)-подход состоит вот в чём: все наши определения и утверждения модифицируются так, чтобы учесть, какой символ стоит справа от разворачиваемого нетерминала (другими словами, чему равен `Next` при свёртке).

Пусть $\mathtt{K} \to \mathtt{U}$ — одно из правил грамматики, а `t` — некоторый терминал или спецсимвол `EOI` (который мы домысливаем в конце входного слова). Определим множество ЛевКонт$(\mathtt{K} \to \mathtt{U}, \mathtt{t})$ как множество всех слов, которые являются содержимым стека непосредственно перед свёрткой `U` в `K` в ходе успешного LR-процесса, при условии $\mathtt{Next} = \mathtt{t}$ (в момент свёртки).

Если отбросить у всех слов из ЛевКонт$(\mathtt{K} \to \mathtt{U})$ их конец `U`, то получится множество всех слов, которые могут появиться в правых выводах перед нетерминалом `K`, за которым стоит символ `t`. Это множество (не зависящее от того, какое из правил $\mathtt{K} \to \mathtt{U}$ для нетерминала `K` выбрано) мы будем обозначать Лев$(\mathtt{K}, \mathtt{t})$.

16.4.1. Написать грамматику для порождения множеств Лев$(\mathtt{K}, \mathtt{t})$.

Решение. Её нетерминалами будут символы $\langle \mathtt{ЛевK\ t} \rangle$ для каждого нетерминала `K` и для каждого терминала `t` (а также для $\mathtt{t} = \mathtt{EOI}$). Её правила таковы. Пусть `P` — начальный нетерминал исходной грамматики. Тогда в новой грамматике будет правило

$$\langle \mathtt{ЛевP\ EOI} \rangle \to \qquad \text{(пустое слово)}.$$

Каждое правило исходной грамматики порождает несколько правил новой. Например, для правила

$$\mathtt{K} \to \mathtt{L\,u\,M\,N}$$

(`L`, `M`, `N` — нетерминалы, `u` — терминал) в новую грамматику мы добавим правила

$$\langle \mathtt{ЛевL\ u} \rangle \to \langle \mathtt{ЛевK\ x} \rangle$$

(для всех терминалов x);

$$\langle \text{ЛевM s} \rangle \to \langle \text{ЛевK y} \rangle \, \text{L u}$$

(для всех s, которые могут начинать слова, выводимые из N, и для всех y, а также для всех пар s = y, если из N выводимо пустое слово);

$$\langle \text{ЛевN s} \rangle \to \langle \text{ЛевK s} \rangle \, \text{L u M}$$

(для всех терминалов s). □

16.4.2. Как меняется определение ситуации?

Решение. Ситуацией называется пара

$$[\text{ситуация в старом смысле}, \text{терминал или EOI}]$$ □

16.4.3. Как изменится определение согласованности?

Решение. Слово S из терминалов и нетерминалов согласовано с ситуацией [K → U_V, t] (здесь t — терминал или EOI), если S кончается на U, то есть S = TU, и, кроме того, T принадлежит Лев(K, t). □

16.4.4. Каковы правила для индуктивного вычисления множества Сост(S) ситуаций, согласованных с данным словом S?

Ответ.

(1) Если слово S согласовано с ситуацией [K → U_V, t], причём слово V начинается на букву J, то есть V = JW, то слово SJ согласовано с ситуацией [K → UJ_W, t].

Это правило полностью определяет все ситуации с непустой левой половиной (то есть не начинающиеся с подчёркивания), согласованные с SJ. Осталось определить, для каких нетерминалов K и терминалов t слово SJ принадлежит Лев(K, t). Это делается по двум правилам:

(2) Если ситуация [L → U_V, t] согласована с SJ (согласно правилу (1)), а V начинается на нетерминал K, то SJ принадлежит Лев(K, s) для всех терминалов s, которые могут начинать слова, выводимые из слова V \ K (слово V без первой буквы K), а также для s = t, если из V \ K выводится пустое слово.

(3) Если SJ входит в Лев(L, t) для некоторых L и t, причём L → V — правило грамматики и V начинается на нетерминал K, то SJ принадлежит Лев(K, s) для всех терминалов s, которые могут начинать слова, выводимые из V \ K, а также для s = t, если из V \ K выводится пустое слово. □

16.4.5. Дать определения LR(1)-конфликтов сдвиг/свёртка и свёртка/свёртка по аналогии с данными выше.

Решение. Пусть дана некоторая грамматика. Пусть S — произвольное слово из терминалов и нетерминалов. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал t, то говорят, что для пары $\langle$S, t$\rangle$ возможен сдвиг. (Это определение не изменилось по сравнению с SLR(1)-случаем — вторые компоненты пар из Сост(S) не учитываются.)

Если в Сост(S) есть ситуация, в которой справа от подчёркивания ничего нет, а вторым членом пары является терминал t, то говорят, что для пары $\langle$S, t$\rangle$ LR(1)-*возможна свёртка* (по соответствующему правилу). Говорят, что для пары $\langle$S, t$\rangle$ возникает LR(1)-*конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для пары $\langle$S, t$\rangle$ возникает LR(1)-*конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка. □

Грамматика называется LR(1)-*грамматикой*, если в ней нет LR(1)-конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одной пары $\langle$S, t$\rangle$.

16.4.6. Построить алгоритм проверки выводимости слова в LR(1)-грамматике.

Решение. Как и раньше, на каждом шаге LR-процесса можно однозначно определить, какой шаг только и может быть следующим. □

Полезно (в частности, для LALR(1)-разбора, смотри ниже) понять, как связаны понятия LR(0) и LR(1)-согласованности.

16.4.7. Сформулировать и доказать соответствующее утверждение.

Ответ. Пусть фиксирована некоторая грамматика. Слово S из терминалов и нетерминалов является LR(0)-согласованным с ситуацией K → U_V тогда и только тогда, когда оно LR(1)-согласовано с парой [K → U_V, t] для некоторого терминала t (или для t = EOI). То же самое другими словами: Лев(K) есть объединение Лев(K, t) по всем t. В последней форме это совсем ясно. □

Замечание. Таким образом, функция Сост(S) в LR(1)-смысле является расширением функции Сост(S) в LR(0)-смысле: $\text{Сост}_{\text{LR}(0)}$(S) получается из $\text{Сост}_{\text{LR}(1)}$(S), если во всех парах выбросить вторые члены.

Теперь мы можем дать определение LALR(1)-грамматики. Пусть фиксирована некоторая грамматика, S — слово из нетерминалов и терминалов, t — некоторый терминал (или EOI). Будем говорить, что для

пары $\langle \mathtt{S}, \mathtt{t} \rangle$ LALR(1)-*возможна свёртка по некоторому правилу*, если существует другое слово $\mathtt{S}_1$ с $\mathrm{Сост}_{\mathrm{LR}(0)}(\mathtt{S}_0) = \mathrm{Сост}_{\mathrm{LR}(0)}(\mathtt{S}_1)$, причём для пары $\langle \mathtt{S}_1, \mathtt{t} \rangle$ LR(1)-возможна свёртка по рассматриваемому правилу. Далее определяются конфликты (естественным образом), и грамматика называется LALR(1)-*грамматикой*, если конфликтов нет.

16.4.8. Доказать, что всякая SLR(1)-грамматика является LALR(1)-грамматикой, а всякая LALR(1)-грамматика является LR(1)-грамматикой.

[Указание. Это — простое следствие определений.] □

16.4.9. Построить алгоритм проверки выводимости в LALR(1)-грамматике, который хранит в стеке меньше информации, чем соответствующий LR(1)-алгоритм.

[Указание. Достаточно хранить в стеке множества $\mathrm{Сост}_{\mathrm{LR}(0)}(\mathtt{S})$, поскольку согласно определению LALR(1)-возможность свёртки ими определяется. (Так что сам алгоритм ничем не отличается от SLR(1)-случая, кроме таблицы возможных свёрток.)] □

16.4.10. Привести пример LALR(1)-грамматики, которая не является SLR(1)-грамматикой. □

16.4.11. Привести пример LR(1)-грамматики, которая не является LALR(1)-грамматикой. □

16.5. Общие замечания о разных методах разбора

Применение этих методов на практике имеет свои хитрости и тонкости, которых мы не касались. (Например, таблицы следует хранить по возможности экономно.) Часто оказывается также, что для некоторого входного языка наиболее естественная грамматика не является LL(1)-грамматикой, но является LR(1)-грамматикой, а также может быть заменена на LL(1)-грамматику без изменения языка. Какой из этих вариантов выбрать, не всегда ясно. Дилетантский совет: если Вы сами проектируете входной язык, то не следует выпендриваться и употреблять одни и те же символы для разных целей — и тогда обычно несложно написать LL(1)-грамматику или рекурсивный анализатор. Если же входной язык задан заранее с помощью LR(1)-грамматики, не являющейся LL(1)-грамматикой, то лучше её не трогать, а разбирать как есть. При этом могут оказаться полезные средства автоматического

порождения анализаторов, наиболее известными из которых являются yacc (UNIX) и bison (GNU).

Большое количество полезной и хорошо изложенной информации о теории и практике синтаксического разбора имеется в книге Ахо, Сети и Ульмана (см. список книг для чтения).

Книги для чтения

А. Ахо, *Р. Сети*, *Дж. Ульман*. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001.

А. Ахо, *Дж. Хопкрофт*, *Дж. Ульман*. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

Н. Вирт. Систематическое программирование. Введение. М.: Мир, 1977.

Н. Вирт. Алгоритмы + структуры данных = программы. М.: Мир, 1985.

Д. Гасфилд. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, 2003.

Д. Грис. Наука программирования. М.: Мир, 1984.

М. Гэри, *Д. Джонсон*. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.

Э. Дейкстра. Дисциплина программирования. М.: Мир, 1978.

Т. Кормен, *Ч. Лейзерсон*, *Р. Ривест*. Алгоритмы: построение и анализ. М.: МЦНМО, 2000.

А. Г. Кушниренко, *Г. В. Лебедев*. Программирование для математиков. М.: Наука, 1988.

В. Липский. Комбинаторика для программистов. М.: Мир, 1988.

Э. Рейнгольд, *Ю. Нивергельт*, *Н. Део*. Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980.

Дж. Хопкрофт, *Р. Мотвани*, *Дж. Ульман*. Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.

Предметный указатель

Указатель имён

Александр Шень

ПРОГРАММИРОВАНИЕ: ТЕОРЕМЫ И ЗАДАЧИ

Оригинал-макет: В. Шувалов
Дизайн обложки: У. Сопова

Подписано в печать 22.10.2013 г. Формат $60 \times 90\ ^1/_{16}$. Бумага офсетная.
Печать офсетная. Печ. л. 18,5. Тираж 2000 экз. Заказ №

Издательство Московского центра
непрерывного математического образования.
119002, Москва, Большой Власьевский пер., 11. Тел. (499) 241-74-83.

Отпечатано в ППП «Типография „Наука“».
119099, Москва, Шубинский пер., 6.

Книги издательства МЦНМО можно приобрести в магазине
«Математическая книга», Большой Власьевский пер., д. 11.
Тел. (499) 241-72-85. E-mail: biblio@mccme.ru