ПРОГРАММИРОВАНИЕ: ТЕОРЕМЫ И ЗАДАЧИ

А. Шень

Несколько замечаний вместо предисловия.

Книга написана по материалам занятий программированием со школьниками математических классов школы № 57 г. Москвы

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 ее «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 достачно прост и естествен, имеет неплохие реализации (например,

Turbo Pascal 3.0 и 5.0 фирмы Borland) и позволяет записать решения всех рассматриваемых задач. Возможно, Модула-2 или Оберон были бы более изящным выбором, но пока что они менее доступны.

Неудачный опыт писания «популярных» учебников по программированию учит: никакого сюсюканья — писать надо так, чтобы потом самим было не стыдно прочесть.

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.

Об «авторских правах»: право формулировать задачу и объяснять её решение является неотчуждаемым естественным правом всякого, кто на это способен. В соответствии с этим текст (в ASCII и TEX-версиях) является свободно распространяемым. Адрес автора:

shen@ium.ac.msk.su, shen@landau.ac.ru
125384, Москва, Беговая, 17, кв. 14
(7-095)-945-82-16

Благодарности. Я рад случаю поблагодарить всех, с кем имел честь сотрудничать, преподавая программирование, особенно тех, кто был «по другую сторону баррикады». Благодарю также Американское математическое общество (фонд помощи бывшему СССР), Международный научный фонд, университет г. Бордо и фонд «Культурная инициатива» за полученные во время подготовки этой книги деньги.

Оглавление

1. Переменные, выражения, присваивания 6
1.1. Задачи без массивов 6
1.2. Массивы 22
1.3. Индуктивные функции (по А.Г. Кушниренко). 38

2. Порождение комбинаторных объектов 43
2.1. Размещения с повторениями 43
2.2. Перестановки 44
2.3. Подмножества 45
2.4. Разбиения 48
2.5. Коды Грея и аналогичные задачи 50
2.6. Несколько замечаний 56
2.7. Подсчет количеств 59

3. Обход дерева. Перебор с возвратами 62
3.1. Ферзи, не бьющие друг друга: обход дерева позиций . . 62
3.2. Обход дерева в других задачах 73

4. Сортировка 75
4.1. Квадратичные алгоритмы 75
4.2. Алгоритмы порядка $n \log n$ 76
4.3. Применения сортировки. 84
4.4. Нижние оценки для числа сравнений при сортировке . . 85
4.5. Родственные сортировке задачи 87

5. Конечные автоматы и обработка текстов 90
5.1. Составные символы, комментарии и т.п. 90
5.2. Ввод чисел 92

6. Типы данных 97
6.1. Стеки 97
6.2. Очереди 105
6.3. Множества 115
6.4. Разные задачи 119

7. Рекурсия 121
7.1. Примеры рекурсивных программ 121
7.2. Рекурсивная обработка деревьев 125
7.3. Порождение комбинаторных объектов, перебор 128
7.4. Другие применения рекурсии 133

8. Как обойтись без рекурсии 141
8.1. Таблица значений (динамическое программирование) . . 141
8.2. Стек отложенных заданий 147
8.3. Более сложные случаи рекурсии 150

9. Разные алгоритмы на графах 154
9.1. Кратчайшие пути 154
9.2. Связные компоненты, поиск в глубину и ширину 158

10. Сопоставление с образцом 165
10.1. Простейший пример 165
10.2. Повторения в образце — источник проблем 168
10.3. Вспомогательные утверждения 170
10.4. Алгоритм Кнута–Морриса–Пратта 171
10.5. Алгоритм Бойера–Мура 174
10.6. Алгоритм Рабина 177
10.7. Более сложные образцы и автоматы 178

11. Представление множеств. Хеширование 188
11.1. Хеширование с открытой адресацией 188
11.2. Хеширование со списками 191

12. Множества. Деревья. Сбалансированные деревья 198
12.1. Представление множеств с помощью деревьев 198
12.2. Сбалансированные деревья 207

13. Контекстно-свободные грамматики 220
13.1. Общий алгоритм разбора 220
13.2. Метод рекурсивного спуска 226
13.3. Алгоритм разбора для LL(1)-грамматик 239

14. Синтаксический разбор слева направо (LR) 248
14.1. LR-процессы 248
14.2. LR(0)-грамматики 254
14.3. SLR(1)-грамматики 260
14.4. LR(1)-грамматики, LALR(1)-грамматики 261
14.5. Общие замечания о разных методах разбора 265

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. Введем 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. Введем целую переменную k, которая меняется от 0 до n, причем 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. Внесем 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 нечетно, то после вычитания единицы становится

четным), 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

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

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: $a_0 = 1$, $a_1 = 1$, $a_k = a_{k-1} + a_{k-2}$ 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: $\mathtt{sum} = 1/1! + \ldots + 1/k!$, $\mathtt{last} = 1/k!$ (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 = max (a,b)}
{инвариант: никакое  число, большее k, не является
  общим делителем}
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 $a, b \geqslant 0$.

```
 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 $a \geqslant b$, НОД(a,b) = НОД(a, b mod a) при $b \geqslant a$.

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

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. Остается 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

$$\begin{aligned} НОД(2a,2b) &= 2 \cdot НОД(a,b) \\ НОД(2a,b) &= НОД(a,b) \quad \text{при нечетном } b, \end{aligned}$$

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 $ax + by = НОД(a, b)$.

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 нечетно. (Это свойство будет сохраняться и далее.)

Теперь попытаемся, как и раньше, хранить такие числа p, q, r, s, что

$$\begin{aligned} m &= ap + bq \\ n &= ar + bs \end{aligned}$$

Проблема в том, что при делении, скажем, m на 2 надо разделить p и q на 2, и они перестанут быть целыми (a 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

$$2^i d = ax + by$$

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

$$\begin{aligned} x &:= x + b \\ y &:= y - a \end{aligned}$$

(оно не меняет $ax + by$). 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 нёчетного 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, причем общее число действий должно быть порядка n.

Решение. Введем переменную k_square (square — квадрат), связанную с k соотношением $\mathtt{k_square} = \mathtt{k}^2$:

```
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;
```

1.1.22. Составить программу, печатающую разложение на простые множители заданного натурального числа $\mathtt{n} > 0$ (другими словами, требуется печатать только простые числа и произведение напечатанных чисел должно быть равно n; если $\mathtt{n} = 1$, 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; напеча-
 танные 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 $n > 1$ простым.

1.1.25. (Для знакомых с основами алгебры) Дано целое гауссово число $n + mi$ (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 $i = 0,1,2,\ldots,9$. Составить программу, печатающую десятичную запись заданного натурального числа $n > 0$. (Случай $n = 0$ явился бы некоторым исключением, так как обычно нули в начале числа не печатаются, а для $n = 0$ — 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. Приведенный инвариант допускает случай, когда k < 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 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}
```

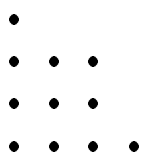
Здесь ... — пока еще не написанный кусок программы, который будет таким:

```
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 решетки (с целыми координатами) в первом квадранте, попадающие внутрь круга радиуса $\sqrt{n}$. Интересующее нас множество (назовем 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>. Введем еще 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 $l \geqslant 0$, для которых <k,l> не принадлежит X;

- s — число пар натуральных x, y, для которых x < 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, ее 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 все 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. (Э. Дейкстра). Функция f с натуральными аргументами и значениями определена так: f(0) = 0, f(1) = 1, f(2n) = f(n), f(2n + 1) = f(n) + f(n + 1). Составить программу вычисления f(n) по заданному n, требующую порядка log 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}
```

1.1.33. То же, если f(0) = 13, f(1) = 17, f(2n) = 43 f(n)+57 f(n+1), f(2n + 1) = 91 f(n) + 179 f(n + 1) при n ⩾ 1.

[Указание. Хранить коэффициенты в выражении f(n) через три соседних числа.]

1.1.34. Даны натуральные числа a и b, причем 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 четных 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[1]=y[1] при 1<=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 `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. (Сообщил А.Л. Брудно.) Прямоугольное поле `m` $\times$ `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.)

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

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

[Указание. Смотри главу 4 (Сортировка)]

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

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

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

```
for i := 1 to n div 2 do begin
| ...обменять 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) Перевернем (расположим в обратном порядке) отдельно начало и конец массива, а затем перевернем 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 (назовем его A) не больше длины правого (назовем его B). Выделим в B начало той же длины, что и A, назовем 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 $a[n]\,x^n + \ldots + a[1]\,x + a[0]$.

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 заданной 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 еще просить вычислять только значение производной, не упоминая о самом многочлене, получается не такая уж простая задача.)

1.2.14. 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;
```

1.2.15. Предложенный выше алгоритм перемножения многочленов требует порядка n^2 действий для перемножения двух многочленов степени

n. Придумать более эффективный (для больших 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 четырех умножений многочленов степени k, однако их количество можно сократить до трех с помощью такой хитрости: вычислить AC, BD и $(A+B)(C+D)$, а затем заметить, что $AD+BC = (A+B)(C+D) - AC - BD$.]

1.2.16. 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.

1.2.17. Решить предыдущую задачу, если известно лишь, что `x[1]` $\leqslant$ ... $\leqslant$ `x[k]` и `y[1]` $\leqslant$ ... $\leqslant$ `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 придется 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;
end;
```

Замечание. Эта программа имеет дефект: при проверке условия

```
(l1<l) and (x[l1+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 ведет себя, например, система Turbo Pascal, 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. Введем 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.

1.2.18. 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 $k + l$.)

1.2.19. 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 идет раньше в алфавитном порядке. Если в одной стопке карточки кончились, берём их из другой стопки.

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

1.2.21. Даны два массива `x[1]` $\leqslant \ldots \leqslant$ `x[k]` и `y[1]` $\leqslant \ldots \leqslant$ `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 x[1] $\leqslant \ldots \leqslant$ x[k] и q $-$ y[l] $\leqslant \ldots \leqslant$ q $-$ y[1], что нетрудно сделать в ходе их слияния в один (воображаемый) массив.]

1.2.22. (из книги Д. Гриса) Некоторое число содержится в каждом из трех целочисленных неубывающих массивов x[1] $\leqslant \ldots \leqslant$ x[p], y[1] $\leqslant \ldots \leqslant$ y[q], z[1] $\leqslant \ldots \leqslant$ 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]);
```

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

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

```
a: array [1..n] of array [1..m] of integer;
```

a[1][1] $\leqslant \ldots \leqslant$ a[1][m], ..., a[n][1] $\leqslant \ldots \leqslant$ 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. Введем 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] найдется 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]]);
```

1.2.25. Приведенное решение предыдущей задачи требует порядка mn^2 действий. Придумать способ с числом действий порядка mn.

[Указание. Придется 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.]

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

Решение. (Предполагаем, что `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 `x[m]` $=$ `a` инвариант не нарушается.)

Каждый раз `r` $-$ `l` уменьшается примерно вдвое, откуда и вытекает требуемая оценка числа действий.

Замечание.

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

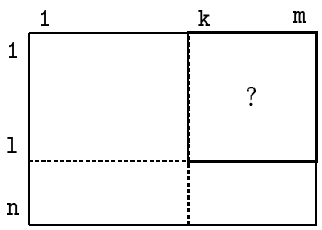
1.2.27. (Из книги Д. Гриса) Дан массив `x: array[1..n] of array[1..m] of integer`, упорядоченный по строкам и по столбцам:

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

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

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

Решение. Представляя себе массив a 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 $1 \leqslant i \leqslant l$ и $k \leqslant j \leqslant m$.



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

```
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 ее}
| 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.)

1.2.28. (Московская олимпиада по программированию) Дан неубывающий массив положительных целых чисел $a[1] \leqslant a[2] \leqslant \ldots \leqslant a[n]$. 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 $a[1],\ldots,a[k]$, 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] $\leqslant$ 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.)

1.2.29. (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.]

1.2.30. 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.

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;
```

1.2.31. 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 берем 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;
```

1.2.32. (Вариант предыдущей задачи, названный в книге Дейкстры *задачей о голландском флаге*.) В массиве стоят числа 0, 1 и 2. Переставить их в порядке возрастания, если единственной разрешенной операцией (помимо чтения) над массивом является перестановка двух элементов.

1.2.33. 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.

1.2.34. 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 нем чисел. Общее число действий порядка n^2.

Решение. Сначала для каждого горизонтального прямоугольника размером $m \times 1$ вычисляем сумму стоящих в нем 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.)

1.2.35. 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 $F : N \times M \to N$, для которой

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

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

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

при всех $\langle \texttt{x[1]} \ldots \texttt{x[n]} \rangle$). 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 ее 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, разделенных 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] — подпоследовательность 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. Диментманом). Обозначим через f(p, q) максимальную длину общей подпоследовательности последовательностей x[1]...x[p] и y[1]...y[q]. Тогда

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

(Поскольку $\texttt{f(p-1,q-1)}+1 \geqslant \texttt{f(p,q-1)}, \texttt{f(p-1,q)}$, во втором случае максимум трех 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 ее 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 кото-
```

```
    |   рых 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.

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 найдется, 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, 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 $\langle \texttt{1 2...n} \rangle$, последней — $\langle \texttt{n...2 1} \rangle$. 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] > ... > 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, остается 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.

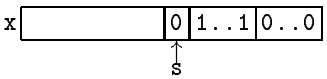
2.3. Подмножества

2.3.1. 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 разберем 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 еще 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, кладем 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·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 возьмем 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 счет 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, причем 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 горизонтали). 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 кругом.

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 ее 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 ее сумму с предыдущей цифрой (по модулю 2).

Иными словами, число $a_1, a_2, \ldots, a_n$ преобразуем в $a_1, a_1 + a_2, a_2 + a_3, \ldots, a_{n-1} + a_n$ (сумма по модулю 2). Для $n = 3$ получим:

000 001 011 010 110 111 101 100

Легко проверить, что описанное преобразование чисел обратимо (и тем самым дает все последовательности по одному разу). Кроме того, двоичные записи соседних чисел отличаются заменой конца 011...1 на конец 100...0, 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:

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.

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:

$$3.2\ 1 \to 2\ 3.1 \to 2.1\ 3 \to 1\ 2.3 \to 1.3\ 2 \to 3\ 1\ 2$$

(между переставляемыми числами вставлены точки).

Решение. Наряду с множеством перестановок рассмотрим множество последовательностей `y[1]..y[n]` целых неотрицательных чисел, для которых $\texttt{y[1]} \leqslant 0, \ldots, \texttt{y[n]} \leqslant \texttt{n-1}$. В нем 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 еще 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, причем i-ый член будет меняться как раз только если все следующие шашки стоят у края. Надо еще 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, исоответствующим 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 приведем 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 от-

резка положительна (т.е. число минус единиц в нём не превосходит числа единиц).

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. Ее 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] все 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 определяют 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. 152.]

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. 143.)

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

2.7. Подсчет 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

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

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 учета 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 $(3q^2 + q)/2$).

Однако и без её использования можно придумать способ вычисления $P(n)$, который существенно эффективнее перебора и подсчета 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, разобьем 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

$$\begin{aligned} R(n,k) &= \sum_{i=1}^{k} R(n-i,i) \quad \text{при } k \leqslant n; \\ R(n,k) &= R(n,n) \quad \text{при } k \geqslant n, \end{aligned}$$

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.

Разобьем 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.)

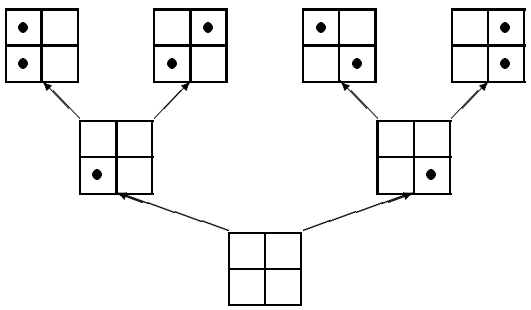
3. Обход дерева. Перебор с возвратами

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 удается реализовать непосредственно, и в этой главе мы рассмотрим другой полезный прием 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.

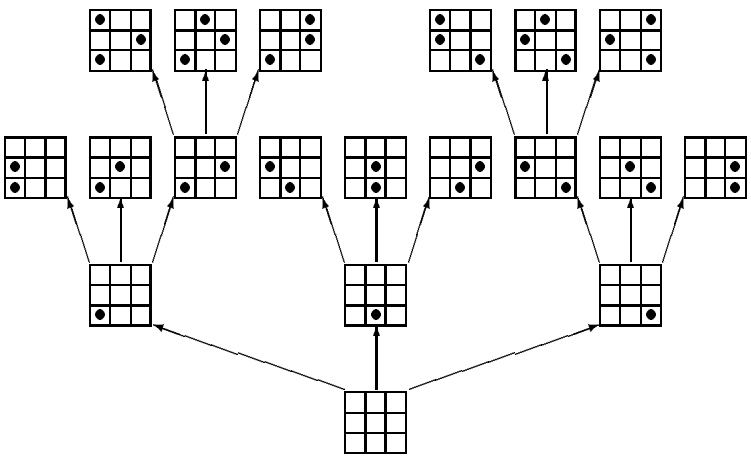
3.1.1. Перечислить все способы расстановки n ферзей на шахматной доске $n \times n$, при которых они не бьют друг друга.

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

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



Дерево допустимых позиций для $n = 3$

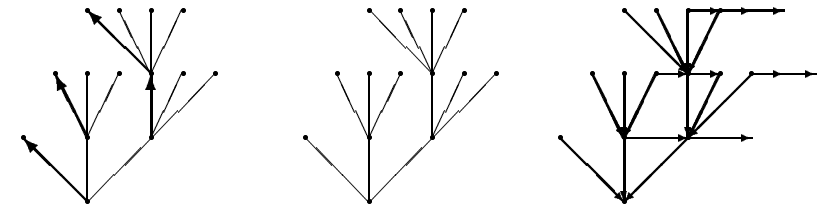
Разобьем задачу на две части: (1) обход произвольного дерева и (2) реализацию дерева допустимых позиций.

Сформулируем задачу обхода произвольного дерева. Будемсчитать, 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.)

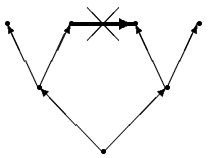


- от этого не видны стрелки

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».

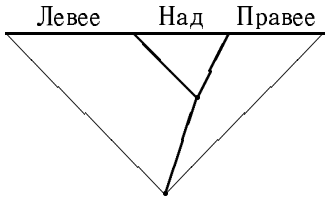


Так команда
`вправо`
не действует!

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 нее и сворачивать вправо, не доходя до нее.) 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 ее выполнения):

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

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

Решение. Процедура `вверх_налево` 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;
| 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:

$$\begin{aligned} \texttt{state} = \texttt{WL} &\Rightarrow \texttt{ОЛ} \\ \texttt{state} = \texttt{WLU} &\Rightarrow \texttt{ОЛН} \end{aligned}$$

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. 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}
```

3.1.5. Приведенная 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}
```

3.1.6. 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.
```

3.1.7. 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 ее сумма не превосходит `s`.

Замечание. По сравнению с полным перебором всех 2^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»). Традиционное название этой задачи — «задача о рюкзаке» (рюкзак общей грузоподъемностью `s` нужно упаковать под завязку, располагая предметами веса `a[1]...a[n]`). См. также в главе 8 алгоритм её решения, полиномиальный по `n + 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] $\leqslant \ldots \leqslant$ 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] $\leqslant \ldots \leqslant$ 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 займет
|   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, число действий которого было бы порядка $n \log n$, то есть не превосходило бы $Cn \log n$ для некоторого C и для всех n.

Мы предложим два решения.

Решение 1 (сортировка слиянием).

Пусть `k` — положительное целое число. Разобьем 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`. Назовем 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:

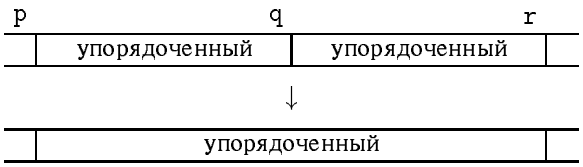
```
k:=1;
{массив x является k-упорядоченным}
```

```
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;
```

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

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

(2) второй отрезок кончился (q0 $=$ r) или не кончился, но элемент в нем 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;
```

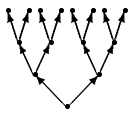
(Если оба отрезка не кончены и первые невыбранные элементы в них равны, то допустимы оба действия; в программе выбрано первое.)

Остаётся лишь переписать результат слияния обратно в массив `x`.

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 кружках-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,...` — 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 дополнениеи 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 переменной 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 свое 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 назовем *регулярным*, если все его вершины регулярны. (В частности, любой лист образует регулярное одноэлементное поддерево.)

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 регу-
    лярно 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 ее правым сыном; t:= правый сын
  end else begin {наибольшее число - в левом сыне}
    поменять t с ее 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 регуярность 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 (о типах 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.

Еще один практически важный алгоритм сортировки (быстрая сортировка Хоара) таков: чтобы отсортировать массив, выберем случайный его элемент b, и разобъем массив на три части: меньшие b, равные b и боль-

шие 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 еще вернемся 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 12), 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, расположеннного в той же точке прямой). Далее двигаемся слева направо, считая число слоёв. Встреченный левый конец увеличивает число слоев на 1, правый — уменьшает. Отметим, что примыкающие друг к другу отрезки обрабатываются правильно: сначала идет 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, а точки на одном луче поместим в порядке увеличения расстояния от начала луча.

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 назовем число взвешиваний при *наихудшем* расположении камней.

4.4.1. Доказать, что сложность любого алгоритма сортировки n камней не меньше $\log_2 n!$. (Здесь $n! = 1 \cdot 2 \cdots n$.)

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.

Это рассуждение показывает, что 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 быстрее. Вот один из примеров.

4.4.2. Имеется массив целых чисел `a[1]...a[n]`, причем 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 применен 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 $1 \ldots m$. 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 $f(x) = i$.]

4.5. Родственные сортировке задачи

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

Решение. Очевидный алгоритм с инвариантом «найден самый лёгкий 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 легкий 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 компонент не более чем на 1). Поэтому суд ничего не знает относительно соотношения весов камней в двух связных компонентах и может допустить, что самый легкий камень — в любой из них.

Разница между этой задачей и предыдущей в том, что $n-1$ взвешиваний не достаточно не только для нахождения самого лёгкого, но даже для того, чтобы убедиться, что данный камень является самым лёгким — 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.)

4.5.3. 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.

4.5.4. 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.

4.5.5. 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 тяжелый 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 исключенного 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, заключенные 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. Еще одна возможность многих реализаций паскаля — это комментарии вида

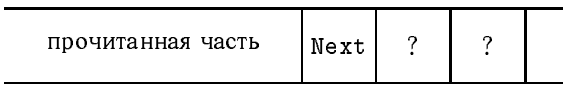
```
i:=i+1;      (*   here i is increased by 1  *)
```

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

5.2. Ввод чисел

Пусть десятичная запись числа подается 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 функцией `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 ее значение.

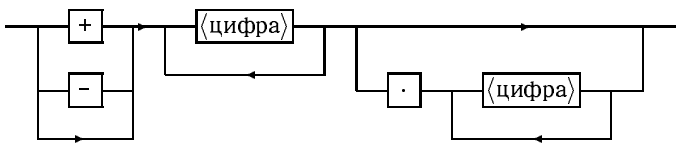
Решение. При чтении дробной части используется переменная `step` — множитель при следующей десятичной цифре.

```
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). Нарисуйте 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 вернемся 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, наполняющими 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:

```
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. Рассмотрим стек, элементами которого являются 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 заберем 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 идет 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 последо-