```
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 $\langle 1\,2\ldots k\rangle$; максимальной — $\langle (n-k+1)\ldots(n-1)\,n\rangle$. 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⩽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 вертикали). На каждой шашке нарисуем стрелочку, которая может быть направлена вверх или вниз. Вначале все шашки поставим на нижнюю горизонталь стрелочкой вверх. Далее двигаем шашки по такому правилу: найдя самую правую шашку, которую можно подвинуть в направлении (нарисованной на ней) стрелки, двигаем её на одну клетку в этом направлении, а все стоящие правее неё шашки (они упёрлись в край) разворачиваем 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:

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

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:

0 1

$\leftarrow$ склеить бока

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

0 0 1 1
0 1 0 1

Сделав третью,

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

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 y[1] $\leqslant$ 0, ..., y[n] $\leqslant$ 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

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

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 отрезка неотрицательна, т.е. число минус единиц в нём не превосходит числа единиц. (Число таких последовательностей называют *числом Каталана*; формулу для чисел Каталана см. в следующем разделе.)

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

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

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

а последней — «горка»

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

Как перейти от последовательности к следующей? До некоторого места они должны совпадать, а затем надо заменить -1 на 1. Место замены должно быть расположено как можно правее. Но заменять -1 на 1 можно только в том случае, если справа от неё есть единица (которую можно заменить на -1). После замены -1 на 1 мы приходим к такой задаче: фиксирован начальный кусок последовательности, надо найти минимальное продолжение. 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. 157.]

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. 148.)

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

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

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.)

В некоторых случаях ответ удаётся получить в виде явной формулы.

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

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

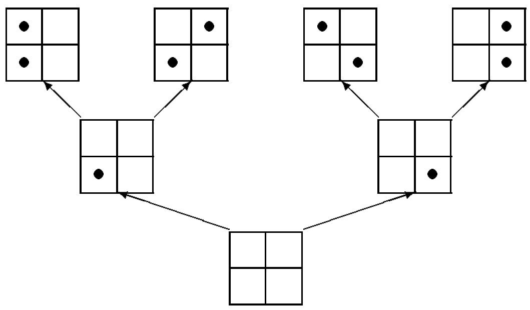
3. Обход дерева. Перебор с возвратами

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

В предыдущей главе мы рассматривали несколько задач одного и того же типа: «перечислить все элементы некоторого множества A». Схема решения была такова: на множестве A вводился порядок и описывалась процедура перехода от произвольного элемента множества A к следующему за ним (в этом порядке). Такую схему не всегда 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.

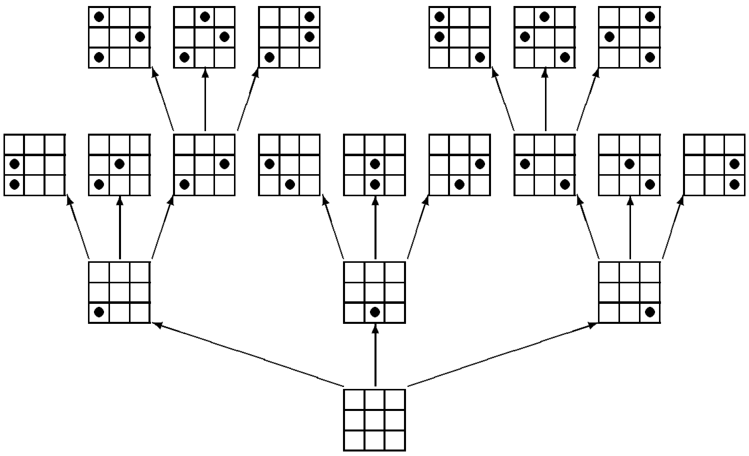
3.1.1. Перечислить все способы расстановки n ферзей на шахматной доске $n \times n$, при которых они не бьют друг друга.

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 $n = 3$

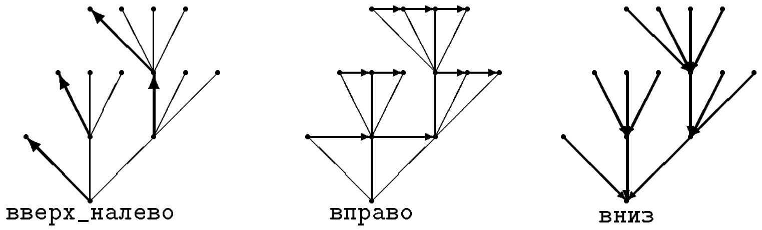
Разобьем задачу на две части: (1) обход произвольного дерева и (2) реализацию дерева допустимых позиций.

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)

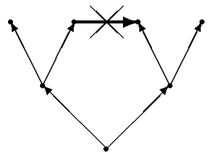
(На рисунках стрелками показано, какие перемещения соответствуют этим командам.)



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

- есть_сверху;
- есть_справа;
- есть_снизу;

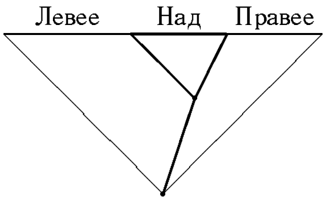
(последняя проверка истинна всюду, кроме корня). Обратите внимание, что команда вправо позволяет перейти лишь к «родному брату», но не к «двоюродному».



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`, `есть_снизу`, ОЛН} `вниз` {ОЛН}

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

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

3.1.3. Доказать правильность следующей программы обхода дерева:

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

Решение. Инвариант цикла:

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

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», Мир, 1979, а также в книге Гэри и Джонсона «Вычислительные машины и труднорешаемые задачи», Мир, 1982). Традиционное название этой задачи — «задача о рюкзаке» (рюкзак общей 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:

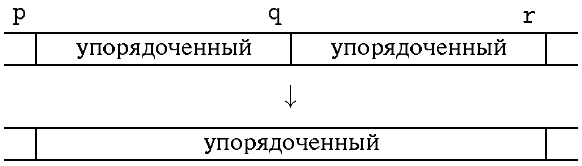
```
k:=1;
{массив x является k-упорядоченным}
```

```
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;
```

(Любители языка C оценят сокращения `b[++s0]=x[++p0]` и `b[++s0]=x[++q0]`.) Первое действие (взятие элемента из первого отрезка) может производиться при одновременном выполнении двух условий:

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

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

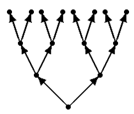
(Если оба отрезка не кончены и первые невыбранные элементы в них равны, то допустимы оба действия; в программе выбрано первое.)

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

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

Решение 2 (сортировка деревом).

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



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

$$\text{число в кружке} = \text{минимум из чисел в кружках-сыновьях}$$

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

Мы будем записывать сортируемые числа во всех вершинах дерева, а не только на верхнем уровне. Пусть `x[1]...x[n]` — массив, подлежащий сортировке. Вершинами дерева будут числа от 1 до n; о числе `x[i]` мы будем говорить как о числе, стоящем в вершине i. В процессе сортировки количество вершин дерева будет сокращаться. Число вершин текущего дерева будем хранить в 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.) Заметим, что истинность утверждения «s-поддерево регулярно» зависит не только от s, но от текущего значения k.

Схема алгоритма такова:

```
k:= n
... Сделать 1-поддерево регулярным;
```

```
{x[1],..,x[k] <= x[k+1] <=..<= x[n]; 1-поддерево регулярно,
 в частности, x[1] - максимальный элемент среди x[1]..x[k]}
while k <> 1 do begin
| ... обменять местами x[1] и x[k];
| k := k - 1;
| {x[1]..x[k-1] <= x[k] <=...<= x[n]; 1-поддерево 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 регулярность не нарушилась.

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

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

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

```
procedure sort (var x: arr);
  var u, k: integer;
  procedure exchange(i, j: integer);
    var tmp: integer;
    begin
    tmp  := x[i];
    x[i] := x[j];
    x[j] := tmp;
  end;
  procedure restore (s: integer);
```

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

Несколько замечаний.

Метод, использованный при сортировке деревом, бывает полезным в других случаях. (См. в главе 6 (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 разобьём массив на три части: меньшие b, равные b и большие b. (Эта задача приведена в главе 1.) Теперь осталось отсортировать первую и третью части: это делается тем же способом. Время работы этого алгоритма — случайная величина; можно доказать, что в среднем он работает не больше $Cn \log n$. На практике — он один из самых быстрых. (Мы 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, расположенного 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 снизу вверх, а точки на одном луче упорядочим по расстоянию от начала луча (это делается для всех лучей, кроме нижнего и верхнего). Ломаная выходит из выбранной (самой левой) точки по нижнему лучу, затем по всем остальным лучам (в описанном порядке) и возвращается по верхнему лучу.

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

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

4.4. Нижние оценки для числа сравнений при сортировке

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

Разумеется, число взвешиваний зависит не только от выбранного нами алгоритма, но и от того, как оказались расположены камни. Сложностью алгоритма 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 $\mathtt{m} + \mathtt{n}$.

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 этой задаче мы доказываем, что $n-2$ 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. Доказать, что можно найти самый легкий и самый тяжёлый из $2n+1$ камней (одновременно), сделав $3n$ взвешиваний.

[Указание. Для начала разобьём камни на пары (n пар, один камень остаётся непарным) и сравним камни внутри пар.]

4.5.4. 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.

4.5.5. 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.

4.5.6. 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 ≠ *	основное	печатать x
после	*	основное	печатать ^
после	x ≠ *	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 процедурой `Move`, которая забирает первый символ из оставшейся части, переводя его в категорию прочитанных.

прочитанная часть	`Next`	?	?	

Будем называть десятичной записью такую последовательность символов:

$$\langle \text{0 или более пробелов} \rangle\ \langle \text{1 или более цифр} \rangle,$$

а также такую:

$$\langle \text{0 или более пробелов} \rangle\ \langle \text{1 или более цифр} \rangle\texttt{.}\langle \text{1 или более цифр} \rangle.$$

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.

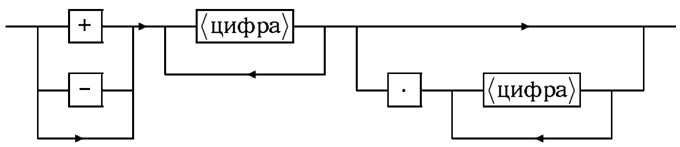
Решение. При чтении дробной части используется переменная `step` — множитель при следующей десятичной цифре.

```
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` (=