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-

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

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

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

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

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

```
Содержание[1]...Содержание[Длина1],
```

а второй —

```
Содержание[n]...Содержание[n-Длина2+1]
```

(вершины обоих стеков записаны последними).

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

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

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

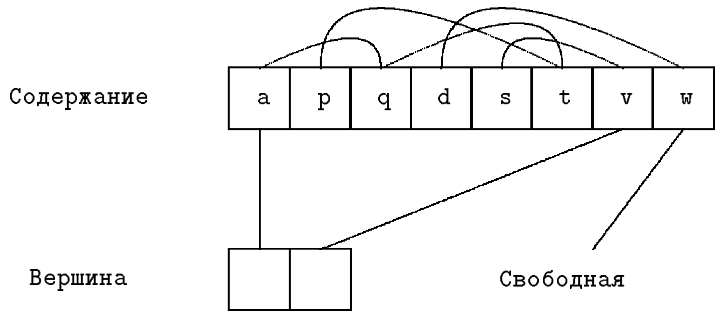
Массив `Содержание` будем изображать как `n` ячеек с номерами `1...n`, каждая из которых содержит элемент типа `T`. Массив `Следующий` изобразим в виде стрелок, проведя стрелку из `i` в `j`, если `Следующий[i]=j`. (Если `Следующий[i]=0`, стрелок из `i` не проводим.) Содержимое `s`-го стека ($s \in 1\ldots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

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

Стрелочные траектории, выходящие из

`Вершина[1]`, ..., `Вершина[k]`

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



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

Свободная = 8

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

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

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

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

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

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

function  Вершина_стека (s: integer): T;
| {вершина s-го стека}
begin
```

```
| Вершина_стека := Содержание[Вершина[s]];
end;
```

6.2. Очереди

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

Операции с очередями:

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

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

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

Реализация очередей в массиве

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

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

Введем массив

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

и переменные

```
Первый: 0..n-1,
Длина : 0..n.
```

При этом элементами очереди будут

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

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

Операции выполняются так:

Сделать пустой:

```
Длина := 0;
Первый := 0;
```

Добавить элемент:

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

Взять элемент:

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

Пуста:

```
Длина = 0
```

Очередной:

```
Содержание [Первый]
```

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

Решение. Инвариант: *стеки, составленные концами, образуют очередь.* (Перечисляя элементы одного стека вглубь и затем элементы второго наружу, мы перечисляем все элементы очереди от первого до последнего.) Ясно, что добавление сводится к добавлению к одному из стеков, а проверка пустоты — к проверке пустоты обоих стеков. Если мы хотим взять элемент, есть два случая. Если стек, где находится начало очереди, не пуст, то берём из него элемент. Если он пуст, то предварительно переписываем в него все элементы второго стека, меняя порядок (это происходит само собой при перекладывании из стека в стек) и сводим дело к первому случаю. Хотя число действий на этом шаге и не ограничено константой, но требование задачи выполнено, так как каждый элемент очереди может участвовать в этом процессе не более одного раза.

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

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

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

Применение очередей

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

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

```
procedure напечатать_и_добавить (t: integer);
begin
| writeln (t);
| Добавить (2*t, x2);
| Добавить (3*t, x3);
| Добавить (5*t, x5);
end;
```

Вот схема программы:

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

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

Длины очередей не превосходят числа напечатанных элементов.

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

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

6.2.6. Известно, что ориентированный граф связен, т. е. из любой вершины можно пройти в любую по рёбрам. Кроме того, из каждой вершины выходит столько же рёбер, сколько входит. Доказать, что существует замкнутый цикл, проходящий по каждому ребру ровно один раз. Составить алгоритм отыскания такого цикла.

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

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

```
while змея включает не все ребра do begin
| if из головы выходит не входящее в змею ребро then begin
```

```
| | удлинить змею этим ребром
| end else begin
| | {голова змеи в той же вершине, что и хвост}
| | отрезать конец хвоста и добавить его к голове
| | {"змея откусывает конец хвоста"}
| end;
end;
```

Докажем, что мы достигнем цели.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
| | Следующий [нов] := 0;
| | Содержание [нов] := t
| | Последний [номер_очереди] := нов;
| end;
end;

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

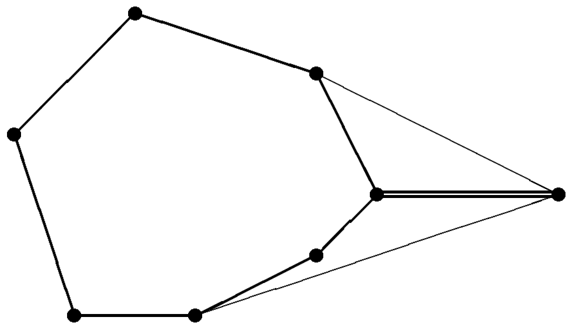
6.2.9. Та же задача для деков вместо очередей.

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

В следующей задаче дек используется для хранения вершин выпуклого многоугольника.

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

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



Будем хранить вершины многоугольника в деке в порядке обхода его периметра по часовой стрелке. При этом выделенная вершина является началом и концом (головой и хвостом) дека. Присоединение «иглы» теперь состоит в добавлении присоединяемой вершины в голову и в хвост дека. Устранение впуклостей несколько более сложно. Назовем *подхвостом* и *подподхвостом* элементы дека, стоящие за его хвостом. Устранение впуклости у хвоста делается так:

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

Таким же способом устраняется впуклость у головы дека.

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

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

6.3. Множества

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

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

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

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

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

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

Решение. Храним дополнительно количество элементов.

6.3.3. То же при следующих ограничениях на число действий:

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

Решение. Дополнительно храним минимальный элемент множества.

6.3.4. То же при следующих ограничениях на число действий:

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

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

Множества целых чисел

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

6.3.5. Память Cn.

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

Решение. Множество представляем с помощью переменных

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

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

6.3.6. Память Cn.

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

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

В следующей задаче полезно комбинировать разные способы.

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

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

```
procedure Доступные (i: integer);
|   {напечатать все вершины, доступные из i, включая i}
| var  X: подмножество 1..n;
|      P: подмножество 1..n;
|      q, v, w: 1..n;
|      k: integer;
begin
| ...сделать X, P пустыми;
| writeln (i);
| ...добавить i к X, P;
| {(1) P = множество напечатанных вершин; P содержит i;
|  (2) напечатаны только доступные из i вершины;
|  (3) X - подмножество P;
```

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

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

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

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

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

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

Более сложные способы представления множеств будут разобраны в главах 11 (хеширование) и 12 (деревья).

6.4. Разные задачи

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

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

а также операцию

- указать номер минимального элемента

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

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

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

Решение. Следуя алгоритму сортировки деревом (в его окончательном варианте), будем размещать элементы очереди в массиве `x[1..k]`, поддерживая такое свойство: `x[i]` старше (имеет больший приоритет) своих сыновей `x[2i]` и `x[2i+1]`, если таковые существуют — и, следовательно, всякий элемент старше своих потомков. (Сведения о приоритетах также хранятся в массиве, так что мы имеем дело с массивом пар $\langle$элемент, приоритет$\rangle$.) Удаление элемента с сохранением этого свойства описано в алгоритме сортировки. Надо еще уметь восстанавливать свойство после добавления элемента в конец. Это делается так:

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

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

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

7. Рекурсия

7.1. Примеры рекурсивных программ

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

(а) почему программа заканчивает работу?

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

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

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

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

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

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

С использованием процедур-функций можно написать так:

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

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

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

7.1.3. Напишите рекурсивную программу возведения в целую неотрицательную степень.

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

Решение.

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

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

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

на

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

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

Этот недостаток можно устранить, написав

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

или воспользовавшись функцией возведения в квадрат (sqr).

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

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

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

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

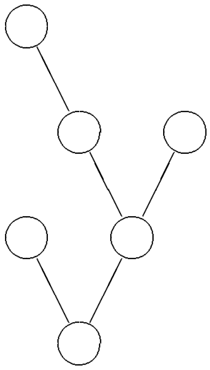
Решение. Напишем рекурсивную процедуру перемещения `i` верхних колец с `m`-го стержня на `n`-ый (остальные кольца предполагаются большими по размеру и лежат на стержнях без движения).

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

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

7.2. Рекурсивная обработка деревьев

Двоичным деревом называется картинка вроде такой:



Нижняя вершина называется *корнем*. Из каждой вершины могут идти две линии: влево вверх и вправо вверх. Вершины, куда они ведут, называются *левым и правым сыновьями* исходной вершины. Вершина может иметь двух сыновей, а может иметь только одного сына (левого или правого). Она может и вовсе не иметь сыновей, и в этом случае называется *листом*.

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

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

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

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

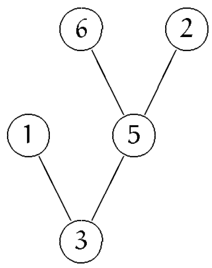
Здесь `N` — достаточно большое натуральное число (номера всех вершин не превосходят `N`). Отметим, что номер вершины никак не связан с ее положением в дереве и что не все числа от `1` до `N` обязаны быть номерами вершин (и, следовательно, часть данных в массивах `l` и `r` — это мусор).

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

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

Нарисовать соответствующее дерево.

Ответ:



7.2.2. Написать программу подсчёта числа вершин в дереве.

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

```
function n(x:integer):integer;
begin
| if x = nil then begin
| | n:= 0;
| end else begin
```

```
| | n:= n(l[x]) + n(r[x]) + 1;
| end;
end;
```

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

7.2.3. Написать программу подсчёта числа листьев в дереве.

Ответ.

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

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

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

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

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

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

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

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

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

7.3. Порождение комбинаторных объектов, перебор

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

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

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

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

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

Основная программа теперь состоит из двух операторов:

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

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

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

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

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

Вот описание процедуры:

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

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

7.3.3. Напечатать все возрастающие последовательности длины k, элементами которых являются натуральные числа от 1 до n. (Предполагается, что $k \leqslant n$, иначе таких последовательностей не существует.)

Решение. Программа оперирует с массивом a[1]...a[k] и целой переменной t. Предполагая, что a[1]...a[t] — возрастающая последовательность натуральных чисел из отрезка 1...n, рекурсивно определённая процедура generate печатает все её возрастающие продолжения длины k. (При этом t и a[1]...a[t] в конце такие же, как в начале.)

```
procedure generate;
| var i: integer;
begin
| if t = k then begin
| | печатать a[1]..a[k]
| end else begin
| | t:=t+1;
| | for i:=a[t-1]+1 to t-k+n do begin
| | | a[t]:=i;
| | | generate;
```

```
    end;
    t:=t-1;
  end;
end;
```

Замечание. Цикл for мог бы иметь верхней границей n (вместо $t - k + n$). Наш вариант экономит часть работы, учитывая тот факт, что предпоследний (k-1-ый) член не может превосходить n-1, k-2-ой член не может превосходить n-2 и т. п.

Основная программа теперь выглядит так:

```
t:=1;
for j:=1 to 1-k+n do begin
  a[1]:=j;
  generate;
end;
```

Можно было бы добавить к массиву a слева фиктивный элемент a[0] = 0, положить t = 0 и ограничиться единственным вызовом процедуры generate.

7.3.4. Перечислить все представления положительного целого числа n в виде суммы последовательности невозрастающих целых положительных слагаемых.

Решение. Программа оперирует с массивом a[1..n] (максимальное число слагаемых равно n) и с целой переменной t. Предполагая, что a[1]...a[t] — невозрастающая последовательность целых чисел, сумма которых не превосходит n, процедура generate печатает все представления требуемого вида, продолжающие эту последовательность. Для экономии вычислений сумма a[1] + ... + a[t] хранится в специальной переменной s.

```
procedure generate;
  var i: integer;
begin
  if s = n then begin
    печатать последовательность a[1]..a[t]
```

```
  end else begin
    for i:=1 to min(a[t], n-s) do begin
      t:=t+1;
      a[t]:=i;
      s:=s+i;
      generate;
      s:=s-i;
      t:=t-1;
    end;
  end;
end;
```

Основная программа при этом может быть такой:

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

Замечание. Можно немного сэкономить, вынеся операции увеличения и уменьшения t из цикла, а также не возвращая s каждый раз к исходному значению (увеличивая его на 1 и возвращая к исходному значению в конце). Кроме того, добавив фиктивный элемент a[0] $=$ n, можно упростить основную программу:

```
t:=0; s:=0; a[0]:=n; generate;
```

7.3.5. Написать рекурсивную программу обхода дерева (используя те же команды и проверки, что и в главе 3 (обход дерева).

Решение. Процедура обработать_над обрабатывает все листья над текущей вершиной и заканчивает работу в той же вершине, что и начала. Вот ее рекурсивное описание:

```
procedure обработать_над;
begin
```

```
  | if есть_сверху then begin
  | | вверх_налево;
  | | обработать_над;
  | | while есть_справа do begin
  | | | вправо;
  | | | обработать_над;
  | | end;
  | | вниз;
  | end else begin
  | | обработать;
  | end;
end;
```

7.4. Другие применения рекурсии

Топологическая сортировка. Представим себе n чиновников, каждый из которых выдаёт справки определенного вида. Мы хотим получить все эти справки, соблюдая установленные ограничения: у каждого чиновника есть список справок, которые нужно собрать перед обращением к нему. Дело безнадёжно, если схема зависимостей имеет цикл (справку A нельзя получить без B, B без $C, \ldots$, Y без Z и Z без A). Предполагая, что такого цикла нет, требуется составить план, указывающий один из возможных порядков получения справок.

Изображая чиновников точками, а зависимости — стрелками, приходим к такой формулировке. Имеется n точек, пронумерованных от 1 до n. Из каждой точки ведет несколько (возможно, 0) стрелок в другие точки. (Такая картинка называется *ориентированным графом.*) Циклов нет. Требуется расположить вершины графа (точки) в таком порядке, чтобы конец любой стрелки предшествовал её началу. Эта задача называется *топологической сортировкой.*

7.4.1. Доказать, что это всегда возможно.

Решение. Из условия отсутствия циклов вытекает, что есть вершина, из которой вообще не выходит стрелок (иначе можно двигаться по стрелкам, пока не зациклимся). Ее будем считать первой. Выкидывая все стрелки, в нее ведущие, мы сводим задачу к графу с меньшим числом вершин и продолжаем рассуждение по индукции.

7.4.2. Предположим, что ориентированный граф без циклов хранится в такой форме: для каждого i от 1 до n в num[i] хранится число выходящих из i стрелок, в adr[i][1], ..., adr[i][num[i]] — номера вершин, куда эти стрелки ведут. Составить (рекурсивный) алгоритм, который производит топологическую сортировку не более чем за $C \cdot (n + m)$ действий, где m — число ребер графа (стрелок).

Замечание. Непосредственная реализация приведенного выше доказательства существования не дает требуемой оценки; её приходится немного подправить.

Решение. Наша программа будет печатать номера вершин. В массиве

```
printed: array[1..n] of boolean
```

мы будем хранить сведения о том, какие вершины напечатаны (и корректировать их одновременно с печатью вершины). Будем говорить, что напечатанная последовательность вершин корректна, если никакая вершина не напечатана дважды и для любого номера i, входящего в эту последовательность, все вершины, в которые ведут стрелки из i, напечатаны, и притом до i.

```
procedure add (i: 1..n);
| {дано: напечатанное корректно;}
| {надо: напечатанное корректно и включает вершину i}
begin
| if printed [i] then begin {вершина i уже напечатана}
| | {ничего делать не надо}
| end else begin
| | {напечатанное корректно}
| | for j:=1 to num[i] do begin
| | | add(adr[i][j]);
| | end;
| | {напечатанное корректно, все вершины, в которые из
| |  i ведут стрелки, уже напечатаны - так что можно
| |  печатать i, не нарушая корректности}
```

```
|  |  if not printed[i] then begin
|  |  | writeln(i); printed [i]:= TRUE;
|  |  end;
|  end;
end;
```

Основная программа:

```
for i:=1 to n do begin
| printed[i]:= FALSE;
end;
for i:=1 to n do begin
| add(i)
end;
```

К оценке времени работы мы вскоре вернёмся.

7.4.3. В приведённой программе можно выбросить проверку, заменив

```
if not printed[i] then begin
| writeln(i); printed [i]:= TRUE;
end;
```

на

```
writeln(i); printed [i]:= TRUE;
```

Почему? Как изменится спецификация процедуры?

Решение. Спецификацию можно выбрать такой:

```
дано: напечатанное корректно
надо: напечатанное корректно и включает вершину i;
      все вновь напечатанные вершины доступны из i.
```

7.4.4. Где использован тот факт, что граф не имеет циклов?

Решение. Мы опустили доказательство конечности глубины рекурсии. Для каждой вершины рассмотрим её «глубину» — максимальную длину пути по стрелкам, из неё выходящего. Условие отсутствия циклов гарантирует, что эта величина конечна. Из вершины нулевой глубины стрелок не выходит. Глубина конца стрелки по крайней мере на 1 меньше, чем глубина начала. При работе процедуры `add(i)` все рекурсивные вызовы `add(j)` относятся к вершинам меньшей глубины.

Вернёмся к оценке времени работы. Сколько вызовов `add(i)` возможно для какого-то фиксированного `i`? Прежде всего ясно, что первый из них напечатано `i`, остальные сведутся к проверке того, что `i` уже напечатано. Ясно также, что вызовы `add(i)` индуцируются «печатающими» (первыми) в ызовами `add(j)` для тех `j`, из которых в `i` ведёт ребро. Следовательно, число вызовов `add(i)` равно числу входящих в `i` рёбер (стрелок). При этом все вызовы, кроме первого, требуют O(1) операций, а первый требует времени, пропорционального числу исходящих из `i` стрелок. (Не считая времени, уходящего на выполнение `add(j)` для концов `j` выходящих рёбер.) Отсюда видно, что общее время пропорционально числу рёбер (плюс число вершин).

Связная компонента графа. *Неориентированный граф* — набор точек (вершин), некоторые из которых соединены линиями (рёбрами). Неориентированный граф можно считать частным случаем ориентированного графа, в котором для каждой стрелки есть обратная.

Связной компонентой вершины `i` называется множество всех тех вершин, в которые можно попасть из `i`, идя по рёбрам графа. (Поскольку граф неориентированный, отношение «`j` принадлежит связной компоненте `i`» является отношением эквивалентности.)

7.4.5. Дан неориентированный граф (для каждой вершины указано число соседей и массив номеров соседей, как в задаче о топологической сортировке). Составить алгоритм, который по заданному `i` печатает все вершины связной компоненты `i` по одному разу (и только их). Число действий не должно превосходить $C \cdot$ (общее число вершин и рёбер в связной компоненте).

Решение. Программа в процессе работы будет «закрашивать» некоторые вершины графа. Незакрашенной частью графа будем называть то, что останется, если выбросить все закрашенные вершины и ведущие в них рёбра. Процедура `add(i)` закрашивает связную компоненту `i` *в незакрашенной части графа* (и не делает ничего, если вершина `i` уже закрашена).

```
procedure  add (i:1..n);
begin
| if вершина i закрашена then begin
| | ничего делать не надо
| end else begin
| | закрасить i (напечатать и пометить как закрашенную)
| | для всех j, соседних с i
| | | add(j);
| | end;
| end;
end;
```

Докажем, что эта процедура действует правильно (в предположении, что рекурсивные вызовы работают правильно). В самом деле, ничего, кроме связной компоненты незакрашенного графа, она закрасить не может. Проверим, что вся она будет закрашена. Пусть k — вершина, доступная из вершины i по пути i $\to$ j $\to \ldots \to$ k, проходящему только по незакрашенным вершинам. Будем рассматривать только пути, не возвращающиеся снова в i. Из всех таких путей выберем путь с наименьшим j (в порядке просмотра соседей в процедуре). Тогда при рассмотрении предыдущих соседей ни одна из вершин пути j $\to \ldots \to$ k не будет закрашена (иначе j не было бы минимальным) и потому k окажется в связной компоненте незакрашенного графа к моменту вызова add(j). Что и требовалось.

Чтобы установить конечность глубины рекурсии, заметим, что на каждом уровне рекурсии число незакрашенных вершин уменьшается хотя бы на 1.

Оценим число действий. Каждая вершина закрашивается не более одного раза — при первым вызове add(i) с данным i. Все последующие вызовы происходят при закрашивании соседей — количество таких вызовов не больше числа соседей — и сводятся к проверке того, что вершина i уже закрашена. Первый же вызов состоит в просмотре всех соседей и рекурсивных вызовах add(j) для всех них. Таким образом, общее число действий, связанных с вершиной i, не превосходит константы, умноженной на число ее соседей. Отсюда и вытекает требуемая оценка.

7.4.6. Решить ту же задачу для ориентированного графа (напечатать все вершины, доступные из данной по стрелкам; граф может содержать циклы).

Ответ. Годится по существу та же программа (строку «для всех соседей» надо заменить на «для всех вершин, куда ведут стрелки»).

Быстрая сортировка Хоара. В заключение приведем рекурсивный алгоритм сортировки массива, который на практике является одним из самых быстрых. Пусть дан массив `a[1]...a[n]`. Рекурсивная процедура `sort(l,r:integer)` сортирует участок массива с индексами из полуинтервала $(\mathrm{l}, \mathrm{r}]$, то есть `a[l+1]...a[r]`, не затрагивая остального массива.

```
procedure sort (l,r: integer);
begin
| if (l = r) then begin
| | ничего делать не надо - участок пуст
| end else begin
| | выбрать случайное число s в полуинтервале (l,r]
| | b := a[s]
| | переставить элементы сортируемого участка так, чтобы
| |   сначала шли элементы, меньшие b - участок (l,ll]
| |   затем элементы, равные b        - участок (ll,rr]
| |   затем элементы, большие b       - участок (rr,r]
| | sort (l,ll);
| | sort (rr,r);
| end;
end;
```

Разделение элементов сортируемого участка на три категории (меньшие, равные, больше) рассматривалась в главе 1, с. 40 (это можно сделать за время, пропорциональное длине участка). Конечность глубины рекурсии гарантируется тем, что длина сортируемого участка на каждом уровне рекурсии уменьшается хотя бы на 1.

7.4.7. (Для знакомых с основами теории вероятностей). Доказать, что математическое ожидание числа операций при работе этого алгоритма не превосходит $Cn\log n$, причем константа C не зависит от сортируемого массива.

[Указание. Пусть $T(n)$ — максимум математического ожидания числа операций для всех входов длины n. Из текста процедуры вытекает такое неравенство:

$$T(n) \leqslant Cn + \frac{1}{n} \sum_{k+l=n-1} \bigl(T(k) + T(l)\bigr)$$

Первый член соответствует распределению элементов на меньшие, равные и большие. Второй член — это среднее математическое ожидание для всех вариантов случайного выбора. (Строго говоря, поскольку среди элементов могут быть равные, в правой части вместо $T(k)$ и $T(l)$ должны стоять максимумы $T(x)$ по всем x, не превосходящим k или l, но это не мешает дальнейшим рассуждениям.) Далее индукцией по n нужно доказывать оценку $T(n) \leqslant C'n \ln n$. При этом для вычисления среднего значения $x \ln x$ по всем $x = 1, \ldots, n-1$ нужно вычислять $\int_1^n x \ln x\, dx$ по частям как $\int \ln x\, d(x^2)$. При достаточно большом C' член Cn в правой части перевешивается за счет интеграла $\int x^2\, d\ln x$, и индуктивный шаг проходит.]

7.4.8. Имеется массив из n различных целых чисел и число k. Требуется найти k-ое по величине число в этом массиве, сделав не более Cn действий, где C — некоторая константа, не зависящая от k и n.

Замечание. Сортировка позволяет очевидным образом сделать это за $Cn \log n$ действий. Очевидный способ: найти наименьший элемент, затем найти второй, затем третий,..., k-ый требует порядка kn действий, то есть не годится (константа при n зависит от k).

[Указание. Изящный (хотя практически и бесполезный — константы слишком велики) способ сделать это таков:

А. Разобьем наш массив на $n/5$ групп, в каждой из которых по 5 элементов. Каждую группу упорядочим.

Б. Рассмотрим средние элементы всех групп и перепишем их в массив из $n/5$ элементов. С помощью рекурсивного вызова найдём средний по величине элемент этого массива.

В. Сравним этот элемент со всеми элементами исходного массива: они разделятся на большие его и меньшие его (и один равный ему). Подсчитав количество тех и других, мы узнаем, в какой из этих частей должен находится искомый (k-ый) элемент и каков он там по порядку.

Г. Применим рекурсивно наш алгоритм к выбранной части.

Пусть $T(n)$ — максимально возможное число действий, если этот способ применять к массивам из не более чем n элементов (k может быть каким угодно). Имеем оценку:

$$T(n) \leqslant Cn + T(n/5) + T(\text{примерно } 0{,}7n).$$

Последнее слагаемое объясняется так: при разбиении на части каждая часть содержит не менее $0{,}3n$ элементов. В самом деле, если x — средний из средних, то примерно половина всех средних меньше x. А если в пятёрке средний элемент меньше x, то ещё два заведомо меньше x. Тем самым по крайней мере $3/5$ от половины элементов меньше x.

Теперь по индукции можно доказать оценку $T(n) \leqslant Cn$ (решающую роль при этом играет то обстоятельство, что $1/5 + 0{,}7 < 1$).]

8. Как обойтись без рекурсии

Для универсальных языков программирования (каковым является паскаль) рекурсия не дает ничего нового: для всякой рекурсивной программы можно написать эквивалентную программу без рекурсии. Мы не будем доказывать этого, а продемонстрируем некоторые приёмы, позволяющие избавиться от рекурсии в конкретных ситуациях.

Зачем это нужно? Ответ прагматика мог бы быть таким: во многих компьютерах (в том числе, к сожалению, и в современных, использующих так называемые RISC-процессоры), рекурсивные программы в несколько раз медленнее соответствующих нерекурсивных программ. Ещё один возможный ответ: в некоторых языках программирования рекурсивные программы запрещены. А главное, при удалении рекурсии возникают изящные и поучительные конструкции.

8.1. Таблица значений (динамическое программирование)

8.1.1. Следующая рекурсивная процедура вычисляет числа сочетаний (биномиальные коэффициенты). Написать эквивалентную нерекурсивную программу.

```
function C(n,k: integer):integer;
| {n >= 0; 0 <=  k <=n}
begin
| if (k = 0) or (k = n) then begin
| | C:=1;
| end else begin {0<k<n}
| | C:= C(n-1,k-1)+C(n-1,k)
| end;
end;
```

Замечание. C_n^k — число k-элементных подмножеств n-элементного множества. Соотношение $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ получится, если мы фиксируем некоторый элемент n-элементного множества и отдельно подсчитаем k-элементные подмножества, включающие и не включающие этот элемент.

Таблица значений C_n^k

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

называется треугольником Паскаля (того самого). В нём каждый элемент, кроме крайних единиц, равен сумме двух стоящих над ним.

Решение. Можно воспользоваться формулой

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

Мы, однако, не будем этого делать, так как хотим продемонстрировать более общие приёмы устранения рекурсии. Вместо этого составим таблицу значений функции `C(n,k)` $= C_n^k$, заполняя её для $n = 0, 1, 2, \ldots$, пока не дойдём до интересующего нас элемента.

8.1.2. Что можно сказать о времени работы рекурсивной и нерекурсивной версий в предыдущей задаче? Тот же вопрос о памяти.

Решение. Таблица занимает место порядка n^2, его можно сократить до n, если заметить, что для вычисления следующей строки треугольника Паскаля нужна только предыдущая. Время работы остаётся порядка n^2. Рекурсивная программа требует существенно большего времени: вызов `C(n,k)` сводится к двум вызовам для `C(n-1,..)` , те — к четырём вызовам для `C(n-2,..)` и так далее. Таким образом, время оказывается экспоненциальным (порядка 2^n). Используемая рекурсивной версией память пропорциональна n — умножаем глубину рекурсии (n) на количество памяти, используемое одним экземпляром процедуры (константа).

Кардинальный выигрыш во времени при переходе от рекурсивной версии к нерекурсивной связан с тем, что в рекурсивном варианте одни и те же вычисления происходят много раз. Например, вызов `C(5,3)` в конечном счёте порождает два вызова C(3,2):

$$\begin{tikzcd}[column sep=tiny]
 & & \texttt{C(5,3)} \arrow[dl] \arrow[dr] & & \\
 & \texttt{C(4,2)} \arrow[dl] \arrow[dr] & & \texttt{C(4,3)} \arrow[dl] \arrow[dr] & \\
\texttt{C(3,1)} & & \texttt{C(3,2)} & & \texttt{C(3,3)}
\end{tikzcd}$$

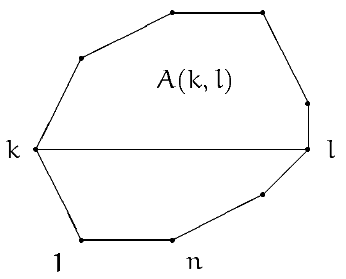
Заполняя таблицу, мы каждую клетку заполняем только однажды — отсюда и экономия. Этот прием называется *динамическим программированием*, и применим в тех случаях, когда объём хранимой в таблице информации оказывается не слишком большим.

8.1.3. Порассуждать на ту же тему на примере рекурсивной и (простейшей) нерекурсивной программ для вычисления чисел Фибоначчи, заданных соотношением

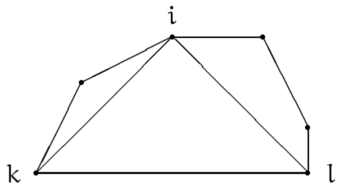
$$\Phi_1 = \Phi_2 = 1; \quad \Phi_n = \Phi_{n-1} + \Phi_{n-2} \ (n > 2).$$

8.1.4. Дан выпуклый n-угольник (заданный координатами своих вершин в порядке обхода). Его разрезают на треугольники диагоналями, для чего необходимо $n-2$ диагонали (докажите индукцией по n). Стоимостью разрезания назовём сумму длин всех использованных диагоналей. Найти минимальную стоимость разрезания. Число действий должно быть ограничено некоторым многочленом от n. (Перебор не подходит, так как число вариантов не ограничено многочленом.)

Решение. Будем считать, что вершины пронумерованы от 1 до n и идут по часовой стрелке. Пусть k, l — номера вершин, причём $l > k$. Через $A(k,l)$ обозначим многоугольник, отрезаемый от нашего хордой k–l. (Эта хорда разрезает многоугольник на два, один из которых включает сторону 1–n; через $A(k,l)$ мы обозначаем другой.) Исходный многоугольник естественно обозначить $A(1,n)$. При $l = k+1$ получается «двуугольник» с совпадающими сторонами.



Через $a(k,l)$ обозначим стоимость разрезания многоугольника $A(k,l)$ диагоналями на треугольники. Напишем рекуррентную формулу для $a(k,l)$. При $l = k+1$ получается двуугольник, и мы полагаем $a(k,l) = 0$. При $l = k+2$ получается треугольник, и в этом случае также $a(k,l) = 0$. Пусть $l > k+2$.



Хорда k–l является стороной многоугольника $A(k,l)$ и, следовательно, стороной одного из треугольников, на которые он разрезан. Противоположной вершиной i этого треугольника может быть любая из вершин $k+1,\ldots,l-1$, и минимальная стоимость разрезания может быть вычислена как

$$\min\{(\text{длина хорды } k\text{–}i) + (\text{длина хорды } i\text{–}l) + a(k,i) + a(i,l)\}$$

по всем $i = k+1,\ldots,l-1$. При этом надо учесть, что при $q = p+1$ хорда p–q — не хорда, а сторона, и ее длину надо считать равной 0 (по стороне разрез не проводится).

Составив таблицу для $a(k, l)$ и заполняя её в порядке возрастания числа вершин (равного $l-k+1$), мы получаем программу, использующую память порядка n^2 и время порядка n^3 (однократное применение рекуррентной формулы требует выбора минимума из не более чем n чисел).

8.1.5. Матрицей размера $m \times n$ называется прямоугольная таблица из m строк и n столбцов, заполненная числами. Матрицу размера $m \times n$ можно умножить на матрицу размера $n \times k$ (ширина левого сомножителя должна равняться высоте правого), и получается матрица размером $m \times k$. Ценой такого умножения будем считать произведение mnk (таково число умножений, которые нужно выполнить при стандартном способе умножения — но сейчас это нам не важно). Умножение матриц ассоциативно, поэтому произведение s матриц можно вычислять в разном порядке. Для каждого порядка подсчитаем суммарную цену всех матричных умножений. Найти минимальную цену вычисления произведения, если известны размеры всех матриц. Число действий должно быть ограничено многочленом от числа матриц.

Пример. Матрицы размером 2×3, 3×4, 4×5 можно перемножать двумя способами. В первом цена равна $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 24 + 40 = 64$, во втором цена равна $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 90$.

Решение. Представим себе, что первая матрица написана на отрезке $[0, 1]$, вторая — на отрезке $[1, 2]$,..., s-ая — на отрезке $[s-1, s]$. Матрицы на отрезках $[i-1, i]$ и $[i, i+1]$ имеют общий размер, позволяющий их перемножить. Обозначим его через $d[i]$. Таким образом, исходным данным в задаче является массив $d[0] \ldots d[s]$.

Через $a(i, j)$ обозначим минимальную цену вычисления произведения матриц на участке $[i, j]$ (при $0 \leqslant i < j \leqslant s$). Искомая величина равна $a(0, s)$. Величины $a(i, i+1)$ равны нулю (матрица одна и перемножать ничего не надо). Рекуррентная формула будет такой:

$$a(i, j) = \min\{a(i, k) + a(k, j) + d[i]d[k]d[j]\}$$

где минимум берется по всем возможным местам последнего умножения, то есть по всем $k = i+1, \ldots, j-1$. В самом деле, произведение матриц на отрезке $[i, k]$ есть матрица размера $d[i]d[k]$, произведение матриц на отрезке $[k, j]$ имеет размер $d[k]d[j]$, и цена вычисления их произведения равна $d[i]d[k]d[j]$.