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

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

а второй —

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

(вершины обоих стеков записаны последними).

6.1.4. Реализовать `k` стеков с элементами типа `T`, общее количество элементов в которых не превосходит `n`, с использованием массивов суммарной длины $C(\texttt{n}+\texttt{k})$, затрачивая на каждое действие со стеками (кроме

начальных действий, делающих все стеки пустыми) время не более некоторой константы C. (Как говорят, общая длина массивов должна быть $O(m+n)$, 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.
```

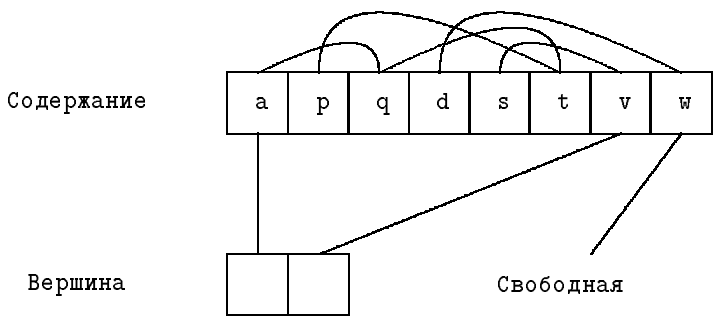
Массив `Содержание` будем изображать как `n` ячеек с номерами `1...n`, каждая из которых содержит элемент типа `T`. Массив `Следующий` изобразим в виде стрелок, проведя стрелку из `i` в `j`, если `Следующий[i]=j`. (Если `Следующий[i]=0`, стрелок из `i` не проводим.) Содержимое `s`-го стека (`s` $\in$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

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

Стрелочные траектории, выходящие из

$$\texttt{Вершина[1]}, \ldots, \texttt{Вершина[k]}$$

(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;
```

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 вошел — первым вышел), а очередей — First In First Out (первым вошел — первым вышел).

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;
```

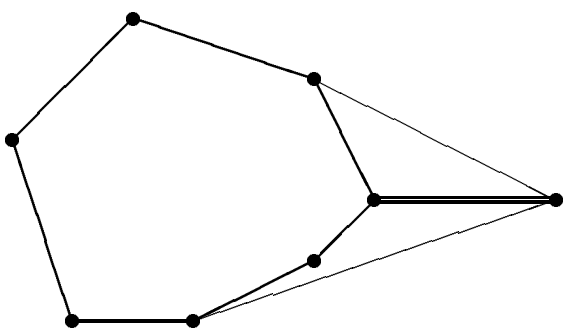
6.2.9. Та же задача для деков вместо очередей.

[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 $\{1 \dots n\}$

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

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

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

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

Решение. Храним дополнительно количество элементов.

6.3.3. То же при следующих ограничениях на число действий:

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

Решение. Дополнительно храним минимальный элемент множества.

6.3.4. То же при следующих ограничениях на число действий:

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

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

Множества целых чисел

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

6.3.5. Память Cn.

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

Решение. Множество представляем с помощью переменных

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

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

6.3.6. Память Cn.

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

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

В следующей задаче полезно комбинировать разные способы.

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

Решение. (Другое решение смотри в главе о рекурсии.) 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 $C \log n$, не считая операции «начать работу» (которая требует не более Cn действий).

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

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

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

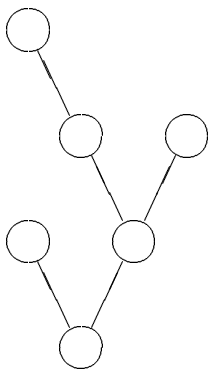
Решение. Напишем рекурсивную процедуру перемещения i верхних колец с m-го стержня на n-ый (остальные кольца предполагаются большими по размеру и лежат на стержнях без движения).

```
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.)

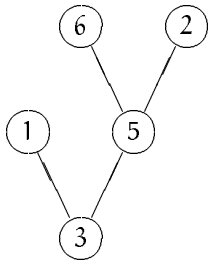
Здесь N — достаточно большое натуральное число (номера всех вершин не превосходят N). Отметим, что номер вершины никак не связан с ее положением в дереве и что не все числа от 1 до N обязаны быть номерами вершин (и, следовательно, часть данных в массивах l и r — это мусор).

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 s 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;
| 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 про 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,..., 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 последостельность, 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 закрасит `i`, остальные сведутся к проверке того, что `i` уже закрашено. Ясно также, что вызовы `add(i)` индуцируются «закрашивающими» вызовами `add(j)` для тех `j`, из которых в `i` ведёт ребро. Следовательно, число вызовов `add(i)` равно числу входящих в `i` рёбер (стрелок). При этом все вызовы, кроме первого, требуют $O(1)$ операций, а первый требует времени, пропорционального числу исходящих из `i` стрелок. (Не считая времени, уходящего на выполнение `add(j)` для концов `j` выходящих рёбер.) Отсюда видно, что общее время пропорционально числу рёбер (плюс число вершин).

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

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

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

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

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

Докажем, что эта процедура действует правильно (в предположении, что рекурсивные вызовы работают правильно). В самом деле, ничего, кроме связной компоненты незакрашенного графа, она закрасить не может. Проверим, что вся она будет закрашена. Пусть `k` — вершина, доступная из вершины `i` по пути `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 `(l,r]`, то есть `a[l+1]...a[r]`, не затрагивая остального массива.

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

Разделение элементов сортируемого участка на три категории (меньшие, равные, больше) рассматривалась в главе 1, с. 37 (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:

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

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.

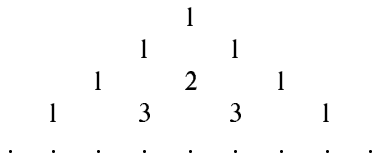
8.1. Таблица значений (динамическое программирование)

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



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

Решение. Можно воспользоваться формулой

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

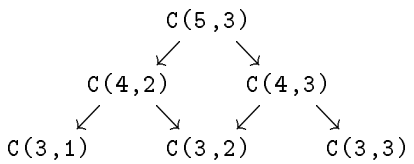
Мы, однако, не будем этого делать, так как хотим продемонстрировать более общие приёмы устранения рекурсии. Вместо этого составим таблицу значений функции `C(n,k)` $= C_n^k$, заполняя её для $n = 0, 1, 2, \ldots$, пока не дойдём до интересующего нас элемента.

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

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

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

счёте порождает два вызова `C(3,2)`:



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

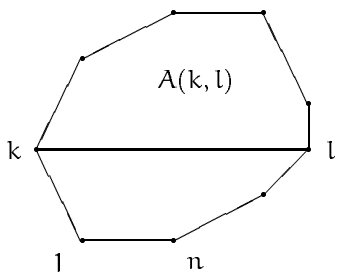
8.1.3. Порассуждать на ту же тему на примере рекурсивной и (простейшей) нерекурсивной программ для вычисления чисел Фибоначчи, заданных соотношением

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

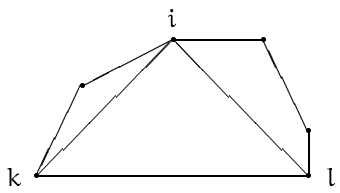
8.1.4. Дан выпуклый n-угольник (заданный координатами своих вершин в порядке обхода). Его разрезают на треугольники диагоналями, для чего необходимо $n-2$ диагонали (докажите индукцией по n). Стоимостью разрезания назовём сумму длин всех использованных диагоналей. Найти минимальную стоимость разрезания. 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 возможных местам последнего умножения, то есть по всем $k = i+1, \ldots, j-1$. В самом деле, произведение матриц на отрезке $[i,k]$ есть матрица размера $d[i]d[k]$, произведение матриц на отрезке $[k,j]$ имеет размер $d[k]d[j]$, и цена вычисления их произведения равна $d[i]d[k]d[j]$.

Замечание. Две последние задачи похожи. Это сходство станет яснее, если написать матрицы-множители на сторонах 1–2, 2–3,...,$(s-1)$–s многоугольника, а на каждой хорде i–j написать произведение всех матриц, стягиваемых этой хордой.

8.1.6. Железная дорога с односторонним движением имеет n станций. Известны цены билетов от i-ой станции до j-ой (при $i < j$ — в обратную сторону проезда нет). Найти минимальную стоимость проезда от начала до конца (с учетом возможной экономии за счет пересадок).

Мы видели, что замена рекурсивной программы на заполнение таблицы значений иногда позволяет уменьшить число действий. Примерно того же эффекта можно добиться иначе: оставить программу рекурсивной, но в ходе вычислений запоминать уже вычисленные значения, а перед очередным вычислением проверять, нет ли уже готового значения.

8.1.7. Задано конечное множество с бинарной операцией (вообще говоря, не коммутативной и даже не ассоциативной). Имеется n элементов $a_1, \ldots, a_n$ этого множества и еще один элемент x. Проверить, можно ли так расставить скобки в произведении $a_1 \times \ldots \times a_n$, чтобы в результате получился x. Число операций должно не превосходить Cn^3 для некоторой константы C (зависящей от числа элементов в выбранном конечном множестве).

Решение. Заполняем таблицу, в которой для каждого участка $a_i \cdots a_j$ нашего произведения хранится список всех возможных его значений (при разной расстановке скобок).

По существу этот же приём применяется в полиномиальном алгоритме проверки принадлежности слова произвольному контекстно-свободному языку (см. главу 13).

Следующая задача (задача о рюкзаке) уже упоминалась в главе 3.

8.1.8. Имеется n положительных целых чисел $x_1 \ldots x_n$ и число N. Выяснить, можно ли получить N, складывая некоторые из чисел $x_1 \ldots x_n$. Число действий должно быть порядка Nn.

[Указание. После i шагов хранится множество тех чисел на отрезке $0 \ldots N$, которые представимы в виде суммы некоторых из $x_1 \ldots x_i$.]

8.2. Стек отложенных заданий

Другой прием устранения рекурсии продемонстрируем на примере задачи о ханойских башнях.

8.2.1. Написать нерекурсивную программу для нахождения последовательности перемещений колец в задаче о ханойских башнях.

Решение. Вспомним рекурсивную программу, перекладывающую `i` верхних колец с `m` на `n`:

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

Видно, что задача «переложить `i` верхних дисков с `m`-го стержня на `n`-ый» сводится к трём задачам того же типа: двум задачам с `i-1` дисками и к одной задаче с единственным диском. Занимаясь этими задачами, важно не позабыть, что́ ещё осталось сделать.

Для этой цели заведём *стек отложенных заданий*, элементами которого будут тройки $\langle \mathtt{i}, \mathtt{m}, \mathtt{n} \rangle$. Каждая такая тройка интерпретируется как заказ «переложить `i` верхних дисков с `m`-го стержня на `n`-ый». Заказы упорядочены в соответствии с требуемым порядком их выполнения: самый срочный — вершина стека. Получам такую программу:

```
procedure move(i,m,n: integer);
begin
| сделать стек заказов пустым
| положить в стек тройку <i,m,n>
```

```
  {инвариант: осталось выполнить заказы в стеке}
  while стек непуст do begin
    удалить верхний элемент, переложив его в <j,p,q>
    if j = 1 then begin
      writeln ('сделать ход', p, '->', q);
    end else begin
      s:=6-p-q;
           {s - третий стержень: сумма номеров равна 6}
      положить в стек тройки <j-1,s,q>, <1,p,q>, <j-1,p,s>
    end;
  end;
end;
```

(Заметим, что первой в стек кладётся тройка, которую надо выполнять последней.) Стек троек может быть реализован как три отдельных стека. (Кроме того, в паскале есть специальный тип, называемый «запись» (`record`), который может быть применён.)

8.2.2. (Сообщил А.К.Звонкин со ссылкой на Анджея Лисовского.) Для задачи о ханойских башнях есть и другие нерекусивные алгоритмы. Вот один из них: простаивающим стержнем (не тем, с которого переносят, и не тем, на который переносят) должны быть все стержни по очереди. Другое правило: поочерёдно перемещать наименьшее кольцо и не наименьшее кольцо, причем наименьшее — по кругу.

8.2.3. Использовать замену рекурсии стеком отложенных заданий в рекурсивной программе печати десятичной записи целого числа.

Решение. Цифры добываются с конца и закладываются в стек, а затем печатаются в обратном порядке.

8.2.4. Написать нерекурсивную программу, печатающую все вершины двоичного дерева.

Решение. В этом случае стек отложенных заданий будет содержать заказы двух сортов: «напечатать данную вершину» и «напечатать все вершины поддерева с данным корнем» (при этом `nil` считается корнем пустого дерева). Таким образом, элемент стека есть пара: ⟨тип заказа, номер вершины⟩.

Вынимая элемент из стека, мы либо сразу исполняем его (если это заказ первого типа), либо помещаем в стек три порождённых им заказа — в одном из шести возможных порядков.

8.2.5. Что изменится, если требуется не печатать вершины двоичного дерева, а подсчитать их количество?

Решение. Печатание вершины следует заменить прибавлением единицы к счётчику. Другими словами, инвариант таков: (общее число вершин) = (счётчик) + (сумма чисел вершин в поддеревьях, корни которых лежат в стеке).

8.2.6. Для некоторых из шести возможных порядков возможны упрощения, делающие ненужным хранение в стеке элементов двух видов. Указать некоторые из них.

Решение. Если требуемый порядок таков:

корень, левое поддерево, правое поддерево,

то заказ на печатание корня можно не закладывать в стек, а выполнять сразу.

Несколько более сложная конструкция применима для порядка

левое поддерево, корень, правое поддерево.

В этом случае все заказы в стеке, кроме самого первого (напечатать поддерево) делятся на пары:

напечатать вершину `x`, напечатать «правое поддерево» `x`

(= поддерево с корнем в правом сыне `x`). Объединив эти пары в заказы специального вида и введя переменную для отдельного хранения первого заказа, мы обойдемся стеком однотипных заказов.

То же самое, разумеется, верно, если поменять местами левое и правое — получается ещё два порядка.

Замечание. Другую программу печати всех вершин дерева можно построить на основе программы обхода дерева, разобранной в главе 3. Там используется команда «вниз». Поскольку теперешнее представление дерева с помощью массивов `l` и `r` не позволяет найти предка заданной вершины,

придётся хранить список всех вершин на пути от корня к текущей вершине. Смотри также главу 9.

8.2.7. Написать нерекурсивный вариант программы быстрой сортировки. Как обойтись стеком, глубина которого ограничена $C \log n$, где n — число сортируемых элементов?

Решение. В стек кладутся пары $\langle i, j \rangle$, интерпретируемые как отложенные задания на сортировку соответствующих участков массива. Все эти заказы не пересекаются, поэтому размер стека не может превысить n. Чтобы ограничиться стеком логарифмической глубины, будем придерживаться такого правила: глубже в стек помещать больший из возникающих двух заказов. Пусть $f(n)$ – максимальная глубина стека, которая может встретиться при сортировке массива из не более чем n элементов таким способом. Оценим $f(n)$ сверху таким способом: после разбиения массива на два участка мы сначала сортируем более короткий (храня в стеке более длинный про запас), при этом глубина стека не больше $f(n/2) + 1$, затем сортируем более длинный, так что

$$f(n) \leqslant \max(f(n/2) + 1, f(n-1)),$$

откуда очевидной индукцией получаем $f(n) = O(\log n)$.

8.3. Более сложные случаи рекурсии

Пусть функция f с натуральными аргументами и значениями определена рекурсивно условиями

$$\begin{aligned} f(0) &= a, \\ f(x) &= h(x, f(l(x))) \quad (x > 0) \end{aligned}$$

где a — некоторое число, а h и l — известные функции. Другими словами, значение функции f в точке x выражается через значение f в точке $l(x)$. При этом предполагается, что для любого x в последовательности

$$x, l(x), l(l(x)), \ldots$$

рано или поздно встретится 0.

Если дополнительно известно, что $l(x) < x$ для всех x, то вычисление f не представляет труда: вычисляем последовательно $f(0), f(1), f(2), \ldots$

8.3.1. Написать нерекурсивную программу вычисления f для общего случая.

Решение. Для вычисления $f(x)$ вычисляем последовательность

$$l(x), l(l(x)), l(l(l(x))), \ldots$$

до появления нуля и запоминаем её, а затем вычисляем значения f в точках этой последовательности, идя справа налево.

Ещё более сложный случай из следующей задачи вряд ли встретится на практике (а если и встретится, то проще рекурсию не устранять, а оставить). Но тем не менее: пусть функция f с натуральными аргументами и значениями определяется соотношениями

$$f(0) = a,$$
$$f(x) = h(x, f(l(x)), f(r(x))) \quad (x > 0),$$

где a — некоторое число, а l, r и h — известные функции. Предполагается, что если взять произвольное число и начать применять к нему функции l и r в произвольном порядке, то рано или поздно получится 0.

8.3.2. Написать нерекурсивную программу вычисления f.

Решение. Можно было бы сначала построить дерево, у которого в корне находится x, а в сыновьях вершины i стоят $l(i)$ и $r(i)$ — если только i не равно нулю. Затем вычислять значения функции, идя от листьев к корню. Однако есть и другой способ.

Обратной польской записью (или *постфиксной записью*) выражения называют запись, где знак функции стоит после всех ее аргументов, а скобки не используются. Вот несколько примеров:

$f(2)$	2	f					
$f(g(2))$	2	g	f				
$s(2, t(7))$	2	7	t	s			
$s(2, u(2, s(5, 3))$	2	2	5	3	s	u	s

Постфиксная запись выражения позволяет удобно вычислять его с помощью *стекового калькулятора*. Этот калькулятор имеет стек, который мы будем представлять себе расположенным горизонтально (числа вынимаются и кладутся справа), и клавиши — числовые и функциональные. При нажатии на клавишу с числом это число кладётся в стек. При нажатии на функциональную клавишу соответствующая функция применяется к нескольким аргументам у вершины стека. Например, если в стеке были числа

$$2\ 3\ 4\ 5\ 6$$

и нажата функциональная клавиша s, соответствующая функции от двух аргументов, то в стеке окажутся числа

$$2\ 3\ 4\ s(5,6)$$

Перейдём теперь к нашей задаче. В процессе вычисления значения функции f мы будем работать со стеком чисел, а также с последовательностью чисел и символов `f`, `l`, `r`, `h`, которую мы будем интерпретировать как последовательность нажатий клавиш на стековом калькуляторе. Инвариант такой:

> если стек чисел представляет собой текущее состояние стекового калькулятора, то после нажатия всех клавиш последовательности в стеке останется единственное число, и оно будет искомым ответом.

Пусть нам требуется вычислить значение $f(x)$. Тогда вначале мы помещаем в стек число x, а последовательность содержит единственный символ `f`. (При этом инвариант соблюдается.) Далее с последовательностью и стеком выполняются такие преобразования:

старый стек	старая последовательность	новый стек	новая последовательность
X	$x\ P$	$X\ x$	P
$X\ x$	`l` P	$X\ l(x)$	P
$X\ x$	`r` P	$X\ r(x)$	P
$X\ x\ y\ z$	`h` P	$X\ h(x,y,z)$	P
$X\ 0$	`f` P	$X\ a$	P
$X\ x$	`f` P	X	x `x l f x r f h` P

Здесь x, y, z — числа, X — последовательность чисел, P — последовательность чисел и символов `f`, `l`, `r`, `h`. В последней строке предполагается, что $x \neq 0$. Эта строка соответствует равенству

$$f(x) = h(x, f(l(x)), f(r(x))).$$

Преобразования выполняются, пока последовательность не станет пуста. В этот момент в стеке окажется единственное число, которое и будет ответом.

Замечание. Последовательность по существу представляет собой стек отложенных заданий (вершина которого находится слева).

9. Разные алгоритмы на графах

9.1. Кратчайшие пути

В этом разделе рассматриваются различные варианты одной задач. Пусть имеется `n` городов, пронумерованных числами от `1` до `n`. Для каждой пары городов с номерами `i`, `j` в таблице `a[i][j]` хранится целое число — цена прямого авиабилета из города `i` в город `j`. Считается, что рейсы существуют между любыми городами, `a[i][i]` = `0` при всех `i`, `a[i][j]` может отличаться от `a[j][i]`. Наименьшей стоимостью проезда из `i` в `j` считается минимально возможная сумма цен билетов для маршрутов (в том числе с пересадками), ведущих из `i` в `j`. (Она не превосходит `a[i][j]`, но может быть меньше.)

В предлагаемых ниже задачах требуется найти наименьшую стоимость проезда для некоторых пар городов при тех или иных ограничениях на массив `a` и на время работы алгоритма.

9.1.1. Предположим, что не существует замкнутых маршрутов, для которых сумма цен отрицательна. Доказать, что в этом случае маршрут с наименьшей стоимостью существует.

Решение. Маршрут длиной больше `n` всегда содержит цикл, поэтому минимум можно искать среди маршрутов длиной не более `n`, а их конечное число.

Во всех следующих задачах предполагается, что это условие (отсутствие циклов с отрицательной суммой) выполнено.

9.1.2. Найти наименьшую стоимость проезда из 1-го города во все остальные за время $O(n^3)$.

Решение. Обозначим через `МинСт(1,s,k)` наименьшую стоимость проезда из `1` в `s` менее чем с `k` пересадками. Тогда выполняется такое соотношение:

$$\texttt{МинСт(1,s,k+1)} = \min\Big(\texttt{МинСт(1,s,k)},\ \min_{\texttt{i=1..n}} \texttt{МинСт(1,i,k)+a[i][s]}\Big)$$

Как отмечалось выше, искомым ответом является `МинСт(1,i,n)` для всех `i` = `1...n`.

```
k:= 1;
for i := 1 to n do begin x[i] := a[1][i]; end;
{инвариант: x[i] := МинСт(1,i,k)}
while k <> n do begin
| for s := 1 to n do begin
| | y[s] := x[s];
| | for i := 1 to n do begin
| | | if y[s] > x[i]+a[i][s] then begin
| | | | y[s] := x[i]+a[i][s];
| | | end;
| | end
| | {y[s] = МинСт(1,s,k+1)}
| end;
| for i := 1 to n do begin x[s] := y[s]; end;
| k := k + 1;
end;
```

Приведенный алгоритм называют алгоритмом динамического программирования, или алгоритмом Форда–Беллмана.

9.1.3. Доказать, что программа останется правильной, если не заводить массива y, а производить изменения в самом массиве x (заменив в программе все вхождения буквы y на x и затем удалить ставшие лишними строки).

Решение. Инвариант будет таков:

$$\texttt{МинСт(1,i,n)} \leqslant \texttt{x[i]} \leqslant \texttt{МинСт(1,i,k)}.$$

Этот алгоритм может быть улучшен в двух отношениях: можно за то же время $O(n^3)$ найти наименьшую стоимость проезда $i \to j$ для *всех* пар i,j (а не только при $i = 1$), а можно сократить время работы до $O(n^2)$. Правда, в последнем случае нам потребуется, чтобы все цены a[i][j] были неотрицательны.

9.1.4. Найти наименьшую стоимость проезда $i \to j$ для всех i,j за время $O(n^3)$.

Решение. Для $\mathtt{k} = \mathtt{0}\ldots\mathtt{n}$ через A(i,j,k) обозначим наименьшую стоимость маршрута из i в j, если в качестве пересадочных разрешено использовать только пункты с номерами не больше k. Тогда

$$\mathtt{A(i,j,0)} = \mathtt{a[i][j]},$$

$$\mathtt{A(i,j,k+1)} = \min\Big(\mathtt{A(i,j,k)}, \mathtt{A(i,k+1,k)} + \mathtt{A(k+1,j,k)}\Big)$$

(два варианта соответствуют неиспользованию и использованию пункта k+1 в качестве пересадочного; отметим, что в нем незачем бывать более одного раза).

Этот алгоритм называют алгоритмом Флойда.

9.1.5. Известно, что все цены неотрицательны. Найти наименьшую стоимость проезда $1 \to i$ для всех $i = 1\ldots n$ за время $O(n^2)$.

Решение. В процессе работы алгоритма некоторые города будут выделенными (в начале — только город 1, в конце — все). При этом:

- для каждого выделенного города i хранится наименьшая стоимость пути $1 \to i$; при этом известно, что минимум достигается на пути, проходящем только через выделенные города;
- для каждого невыделенного города i хранится наименьшая стоимость пути $1 \to i$, в котором в качестве промежуточных используются только выделенные города.

Множество выделенных городов расширяется на основании следующего замечания: если среди всех невыделенных городов взять тот, для которого хранимое число минимально, то это число является истинной наименьшей стоимостью. В самом деле, пусть есть более короткий путь. Рассмотрим первый невыделенный город на этом пути — уже до него путь длиннее! (Здесь существенна неотрицательность цен.)

Добавив выбранный город к выделенным, мы должны скорректировать информацию, хранимую для невыделенных городов. При этом достаточно учесть лишь пути, в которых новый город является последним пунктом

пересадки, а это легко сделать, так как минимальную стоимость проезда в новый город мы уже знаем.

При самом бесхитростном способе хранения множества выделенных городов (в булевском векторе) добавление одного города к числу выделенных требует времени O(n).

Этот алгоритм называют алгоритмом Дейкстры.

Отыскании кратчайшего пути имеет естественную интерпретацию в терминах матриц. Пусть A — матрица цен одной авиакомпании, а B — матрица цен другой. Пусть мы хотим лететь с одной пересадкой, причём сначала самолётом компании A, а затем — компании B. Сколько нам придется заплатить, чтобы попасть из города `i` в город `j`?

9.1.6. Доказать, что эта матрица вычисляется по обычной формуле для произведения матриц, только вместо суммы надо брать минимум, а вместо умножения — сумму.

9.1.7. Доказать, что таким образом определенное произведение матриц ассоциативно.

9.1.8. Доказать, что задача о кратчайших путях эквивалентна вычислению A^∞ для матрицы цен A: в последовательности $A, A^2, A^3, \ldots$ все элементы, начиная с некоторого, равны искомой матрице стоимостей кратчайших путей. (Если нет отрицательных циклов!)

9.1.9. Начиная с какого элемента можно гарантировать равенство в предыдущей задаче?

Обычное (не модифицированное) умножение матриц тоже может оказаться полезным, только матрицы должны быть другие. Пусть есть не все рейсы (как раньше), а только некоторые, `a[i,j]` равно `1`, если рейс есть, и `0`, если рейса нет. Возведем матрицу `a` (обычным образом) в степень `k` и посмотрим на её `i`-`j`-ый элемент.

9.1.10. Чему он равен?

Ответ. Числу различных способов попасть из `i` в `j` за `k` рейсов (с `k-1` пересадками).

Случай, когда есть не все рейсы, можно свести к исходному, введя фиктивные рейсы с бесконечно большой (или достаточно большой) стоимостью. Тем не менее возникает такой вопрос. Число реальных рейсов может быть существенно меньше n^2, поэтому интересны алгоритмы, которые работают эффективно в такой ситуации. Исходные данные естественно представлять тогда в такой форме: для каждого города известно число выходящих из него рейсов, их пункты назначения и цены.

9.1.11. Доказать, что алгоритм Дейкстры можно модифицировать так, чтобы для n городов и m рейсов (всего) он требовал не более $C(n + m) \log n$ операций.

[Указание. Что надо сделать на каждом шаге? Выбрать невыделенный город с минимальной стоимостью и скорректировать цены для всех городов, в которые из него есть маршруты. Если бы кто-то сообщал нам, для какого города стоимость минимальна, то хватило бы $C(n + m)$ действий. А поддержание сведений о том, какой элемент в массиве минимален (см. задачу на с. 119) обходится ещё в множитель $\log n$.]

9.2. Связные компоненты, поиск в глубину и ширину

Наиболее простой случай задачи о кратчайших путях — если все цены равны 0 или $+\infty$. Другими словами, мы интересуемся возможностью попасть из i в j, но за ценой не постоим. В других терминах: мы имеем ориентированный граф (картинку из точек, некоторые из которых соединены стрелками) и нас интересуют вершины, доступные из данной.

Для этого случая задачи о кратчайших путях приведенные в предыдущем разделе алгоритмы — не наилучшие. В самом деле, более быстрая рекурсивная программа решения этой задачи приведена в главе 7, а нерекурсивная — в главе 6. Сейчас нас интересует такая задача: не просто перечислить все вершины, доступные из данной, но перечислить их в определенном порядке. Два популярных случая — поиск в ширину и в глубину.

Поиск в ширину.

Надо перечислить все вершины ориентированного графа, доступные из данной, в порядке увеличения длины пути от неё. (Тем самым мы решим задачу о кратчайших путях, кода цены рёбер равны 1 или $+\infty$.)

9.2.1. Придумать алгоритм решения этой задачи с числом действий не более $C \cdot$ (число рёбер, выходящих из интересующих нас вершин).

Решение. Эта задача рассматривалась в главе 6, с. 118. Здесь мы приведём подробное решение. Пусть `num[i]` — количество рёбер, выходящих из `i`, `out[i][1],...,out[i][num[i]]` — вершины, куда ведут рёбра. Вот программа, приведённая ранее:

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

Тогда нам было безразлично, какой именно элемент множества `X` выби-

рается. Если мы будем считать `X` очередью (первым пришёл — первым ушёл), то эта программа напечатает все вершины, доступные из `i`, в порядке возрастания их расстояния от `i` (числа рёбер на кратчайшем пути из `i`). Докажем это.

Обозначим через $V(k)$ множество всех вершин, расстояние которых от `i` (в описанном смысле) равно k. Имеет место такое соотношение:

$$V(k+1) = (\text{концы ребер с началами в } V(k)) \setminus V(0) \setminus V(1) \setminus \ldots \setminus V(k)$$

Докажем, что для любого $k = 0, 1, 2\ldots$ в ходе работы программы будет такой момент (после очередной итерации цикла `while`), когда

> в очереди стоят все элементы $V(k)$ и только они;
> напечатаны все элементы $V(1), \ldots, V(k)$

(Для $k = 0$ — это состояние перед циклом.) Рассуждая по индукции, предположим, что в очереди скопились все элементы $V(k)$. Они будут просматриваться в цикле, пока не кончатся (поскольку новые элементы добавляются в конец, они не перемешаются со старыми). Концы ведущих из них рёбер, если они уже не напечатаны, печатаются и ставятся в очередь — то есть всё как в записанном выше соотношении для $V(k+1)$. Так что когда все старые элементы кончатся, в очереди будут стоять все элементы $V(k+1)$.

Поиск в глубину.

Рассматривая поиск в глубину, удобно представлять себе ориентированный граф как образ дерева. Более точно, пусть есть ориентированный граф, одна из вершин которого выделена. Будем предполагать, что все вершины доступны из выделенной по ориентированным путям. Построим дерево, которое можно было бы назвать «универсальным накрытием» нашего графа. Его корнем будет выделенная вершина графа. Из корня выходят те же стрелки, что и в графе — их концы будут сыновьями корня. Из них в дереве выходят те же стрелки, что и в графе и так далее. Разница между графом и деревом в том, что пути в графе, ведущие в одну и ту же вершину, в дереве «расклеены». В других терминах: вершина дерева — это путь в графе, выходящий из корня. Её сыновья — это пути, продолженные на одно ребро. Заметим, что дерево бесконечно, если в графе есть ориентированные циклы.

Имеется естественное отображение дерева в граф (вершин в вершины). При этом каждая вершина графа имеет столько прообразов, сколько путей в неё ведет. Поэтому обход дерева (посещение его вершин в том или ином порядке) одновременно является и обходом графа — только каждая вершина посещается многократно.

Будем предполагать, что для каждой вершины графа выходящие из неё рёбра упорядочены (например, пронумерованы). Тем самым для каждой вершины дерева выходящие из неё рёбра также упорядочены. Будем обходить дерево так: сначала корень, а потом поддеревья (в порядке ведущих в них рёбер). Такой обход дерева рассматривался нами в главе 7. Ему соответствует обход графа. Если выкинуть из этого обхода повторные посещения уже посещённых вершин, то получится то, что называется «поиск в глубину».

Другими словами, на путях, выходящих из выделенной вершины, введём порядок: путь предшествует своему продолжению; если два пути расходятся в некоторой вершине, то меньшим считается тот, который выходит из нее по меньшему ребру. Вершины теперь упорядочиваются в соответствии с минимальными путями, в них ведущими. Обход вершин графа в указанном порядке называется *поиском в глубину*.

9.2.2. Написать программу поиска в глубину.

[Указание. Возьмём программу обхода дерева (корень $\to$ левое поддерево $\to$ правое поддерево) из главы 7 или из главы 8 и используем её применительно к обстоятельствам. Главное изменение: не надо посещать вершины повторно. Так что если мы попали в уже посещённую вершину, то можно с ней ничего не делать. (Если путь не минимален среди ведущих в данную вершину, то и все его продолжения не минимальны — их просматривать не надо).]

Замечание. Напомним, что в главе 8 упоминались две возможности устранения рекурсии в программе обхода дерева (с.150). Оба варианта можно использовать для поиска в глубину.

Поиск в глубину лежит в основе многих алгоритмов на графах, порой в несколько модифицированном виде.

9.2.3. Неориентированный граф называется *двудольным*, если его вершины можно раскрасить в два цвета так, что концы любого ребра — раз-

ного цвета. Составить алгоритм проверки, является ли заданный граф двудольным, в котором число действий не превосходит $C\cdot$(число рёбер + число вершин).

[Указание. (а) Каждую связную компоненту можно раскрашивать отдельно. (б) Выбрав цвет одной вершины и обходя ее связную компоненту, мы определяем единственно возможный цвет остальных.]

Замечание. В этой задаче безразлично, производить поиск в ширину или в глубину.

9.2.4. Составить нерекурсивный алгоритм топологической сортировки ориентированного графа без циклов. (Рекурсивный алгоритм смотри на с. 134.)

Решение. Предположим, что граф имеет вершины с номерами `1...n`, для каждой вершины `i` известно число `num[i]` выходящих из неё рёбер и номера вершин `dest[i][1],...,dest[i][num[i]]`, в которые эти рёбра ведут. Будем условно считать, что рёбра перечислены «слева направо»: левее то ребро, у которого номер меньше. Нам надо напечатать все вершины в таком порядке, чтобы конец любого ребра был напечатан перед его началом. Мы предполагаем, что в графе нет ориентированных циклов — иначе такое невозможно.

Для начала добавим к графу вершину `0`, из которой ребра ведут в вершины `1,...,n`. Если её удастся напечатать с соблюдением правил, то тем самым все вершины будут напечатаны.

Алгоритм хранит путь, выходящий из нулевой вершины и идущий по рёбрам графа. Переменная `l` отводится для длины этого пути. Путь образован вершинами `vert[1]...vert[l]` и рёбрами, имеющими номера `edge[1]...edge[l]`. Номер `edge[s]` относится к нумерации рёбер, выходящих из вершины `vert[s]`. Тем самым для всех `s` должны выполняться неравенство

$$\texttt{edge[s]} \leqslant \texttt{num[vert[s]]}$$

и равенство

$$\texttt{vert[s+1]} = \texttt{dest[vert[s]][edge[s]]}.$$

Заметим, что конец последнего ребра нашего пути (то есть вершина `dest[vert[l]][edge[l]]`, не включается в массив `vert`. Кроме того,

для последнего ребра мы делаем исключение, разрешая ему указывать «в пустоту», т. е. разрешаем `edge[l]` равняться `num[vert[l]]+1`.

В процессе работы алгоритм будет печатать номера вершин, при этом соблюдая требование «вершина напечатана только после тех вершин, в которые из неё ведут рёбра». Кроме того, будет выполняться такое требование (И):

> вершины пути, кроме последней (`vert[1]...vert[l]`) не напечатаны, но свернув с пути налево, мы немедленно упираемся в напечатанную вершину

Вот что получается:

```
l:=1; vert[1]:=0; edge[1]:=1;
while not( (l=1) and (edge[1]=n+1)) do begin
| if edge[l]=num[vert[l]]+1 then begin
| | {путь кончается в пустоте, поэтому все вершины,
| |     следующие за vert[l], напечатаны - можно
| |     печатать vert[l]}
| | writeln (vert[l]);
| | l:=l-1; edge[l]:=egde[l]+1;
| end else begin
| |  {edge[l] <= num[vert[l]], путь кончается в
| |     вершине}
| |  lastvert:= dest[vert[l]][edge[l]]; {последняя}
| |  if lastvert напечатана then begin
| |  | edge[l]:=edge[l]+1;
| |  end else begin
| |  | l:=l+1; vert[l]:=lastvert; edge[l]:=1;
| |  end;
| end;
end;
{путь сразу же ведет в пустоту, поэтому все вершины
 левее, то есть 1..n, напечатаны}
```

9.2.5. Доказать, что если в графе нет циклов, то этот алгоритм заканчивает работу.

Решение. Пусть это не так. Каждая вершина может печататься только один раз, так что с некоторого момента вершины не печатаются. В графе без циклов длина пути ограничена (вершина не может входить в путь дважды), поэтому подождав ещё, мы можем дождаться момента, после которого путь не удлиняется. После этого может разве что увеличиваться `edge[1]` — но и это не беспредельно.

9.2.6. Доказать, что время работы этого алгоритма не превосходит O(число вершин + число рёбер).

9.2.7. Как модифицировать алгоритм так, чтобы он отыскивал один из циклов, если таковые имеются, и производил топологическую сортировку, если циклов нет?

10. Сопоставление с образцом

10.1. Простейший пример

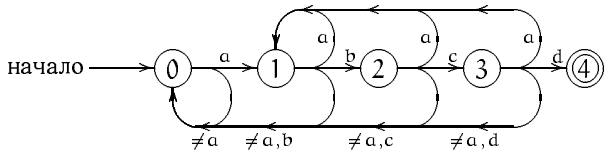
10.1.1. Имеется последовательность символов `x[1]...x[n]`. Определить, имеются ли в ней идущие друг за другом символы `abcd`. (Другими словами, требуется выяснить, есть ли в слове `x[1]...x[n]` подслово `abcd`.)

Решение. Имеется примерно `n` (если быть точным, `n-3`) позиций, на которых может находиться искомое подслово в исходном слове. Для каждой из позиций можно проверить, действительно ли там оно находится, сравнив четыре символа. Однако есть более эффективный способ. Читая слово `x[1]...x[n]` слева направо, мы ожидаем появления буквы `a`. Как только она появилась, мы ищем за ней букву `b`, затем `c`, и, наконец, `d`. Если наши ожидания оправдываются, то слово `abcd` обнаружено. Если же какая-то из нужных букв не появляется, мы оказываемся у разбитого корыта и начинаем всё сначала.

Этот простой алгоритм можно описать в разных терминах. Используя терминологию так называемых *конечных автоматов*, можно сказать, что при чтении слова `x` слева направо мы в каждый момент находимся в одном из следующих состояний: «начальное» (0), «сразу после `a`»(1), «сразу после `ab`» (2), «сразу после `abc`» (3) и «сразу после `abcd`» (4). Читая очередную букву, мы переходим в следующее состояние по правилу

Текущее состояние	Очередная буква	Новое состояние
0	a	1
0	кроме a	0
1	b	2
1	a	1
1	кроме a,b	0
2	c	3
2	a	1
2	кроме a,c	0
3	d	4
3	a	1
3	кроме a,d	0

Как только мы попадем в состояние 4, работа заканчивается.



Соответствующая программа очевидна:

```
i:=1; state:=0;
{i - первая непрочитанная буква, state - состояние}
while (i <> n+1) and (state <> 4) do begin
| if state = 0 then begin
| | if x[i] = a then begin
| | | state:= 1;
| | end else begin
| | | state:= 0;
| | end;
| end else if state = 1 then begin
| | if x[i] = b then begin
| | | state:= 2;
```

```
    | | end else if x[i] = a then begin
    | |   state:= 1;
    | | end else begin
    | |   state:= 0;
    | | end;
    | end else if state = 2 then begin
    | | if x[i] = c then begin
    | |   state:= 3;
    | | end else if x[i] = a then begin
    | |   state:= 1;
    | | end else begin
    | |   state:= 0;
    | | end;
    | end else if state = 3 then begin
    | | if x[i] = d then begin
    | |   state:= 4;
    | | end else if x[i] = a then begin
    | |   state:= 1;
    | | end else begin
    | |   state:= 0;
    | | end;
    | end;
  end;
  answer := (state = 4);
```

Иными словами, мы в каждый момент храним информацию о том, какое максимальное начало нашего образца `abcd` является концом прочитанной части. (Его длина и есть то «состояние», о котором шла речь.)

Терминология, нами используемая, такова. *Слово* — это любая последовательность символов из некоторого фиксированного конечного множества. Это множество называется *алфавитом*, его элементы — *буквами*. Если отбросить несколько букв с конца слова, останется другое слово, называемое *началом* первого. Любое слово также считается своим началом. *Конец* слова — то, что останется, если отбросить несколько первых букв. Любое слово считается своим концом. *Подслово* — то, что останется, если отбросить буквы и с начала, и с конца. (Другими словами, подслова — это

концы начал, или, что то же, начала концов.)

В терминах индуктивных функций (см. раздел 1.3) ситуацию можно описать так: рассмотрим функцию на словах, которая принимает два значения «истина» и «ложь» и истинна на словах, имеющих abcd своим подсловом. Эта функция не является индуктивной, но имеет индуктивное расширение

x $\mapsto$ длина максимального начала слова abcd, являющегося концом x

10.2. Повторения в образце — источник проблем

10.2.1. Можно ли в предыдущих рассуждениях заменить слово abcd на произвольное слово?

Решение. Нет, и проблемы связаны с тем, что в образце могут быть повторяющиеся буквы. Пусть, например, мы ищем вхождения слова ababc. Вот появилась буква a, за ней идет b, за ней идет a, затем снова b. В этот момент мы с нетерпением ждем буквы c. Однако — к нашему разочарованию — вместо нее появляется другая буква, и наш образец ababc не обнаружен. Однако нас может ожидать утешительный приз: если вместо c появилась буква a, то не всё потеряно: за ней могут последовать буквы b и c, и образец-таки будет найден.

Вот картинка, поясняющая сказанное:

```
x y z a b a b a b c ...  ←  входное слово
      a b a b c         ←  мы ждали образца здесь
        a b a b c       ←  а он оказался здесь
```

Таким образом, к моменту

```
x y z a b a b |         ←  входное слово
      a b a b | c       ←  мы ждали образца здесь
        a b   | a b c   ←  а он оказался здесь
```

есть два возможных положения образца, каждое из которых подлежит проверке. Тем не менее по-прежнему возможен конечный автомат, читающий входное слово буква за буквой и переходящий из состояния в состояние в зависимости от прочитанных букв.

10.2.2. Указать состояния соответствующего автомата и таблицу перехода (новое состояние в зависимости от старого и читаемой буквы).

Решение. По-прежнему состояния будут соответствовать наибольшему началу образца, являющемуся концом прочитанной части слова. Их будет шесть: 0, 1 (a), 2 (ab), 3 (aba), 4 (abab), 5 (ababc). Таблица перехода:

Текущее состояние	Очередная буква	Новое состояние
0	a	1 (a)
0	кроме a	0
1 (a)	b	2 (ab)
1 (a)	a	1 (a)
1 (a)	кроме a,b	0
2 (ab)	a	3 (aba)
2 (ab)	кроме a	0
3 (aba)	b	4 (abab)
3 (aba)	a	1 (a)
3 (aba)	кроме a,b	0
4 (abab)	c	5 (ababc)
4 (abab)	a	3 (aba)
4 (abab)	кроме a,c	0

Для проверки посмотрим, к примеру, на вторую снизу строку. Если прочитанная часть кончалась на abab, а затем появилась буква a, то теперь прочитанная часть кончается на ababa. Наибольшее начало образца (ababc), являющееся её концом — это aba.

Философский вопрос: мы говорили, что трудность состоит в том, что есть несколько возможных положений образца, каждое из которых может оказаться истинным. Им соответствуют несколько начал образца, являющихся концами входного слова. Но конечный автомат помнит лишь самое длинное из них. Как же остальные?

Философский ответ. Дело в том, что самое длинное из них определяет все остальные — это его концы, одновременно являющиеся его началами.

Не составляет труда для любого конкретного образца написать программу, осуществляющую поиск этого образца описанным способом. Однако хотелось бы написать программу, которая ищет произвольный образец

в произвольном слове. Это можно делать в два этапа: сначала по образцу строится таблица переходов конечного автомата, а затем читается входное слово и состояние преобразуется в соответствии с этой таблицей. Подобный метод часто используется для более сложных задач поиска (см. далее), но для поиска подслова существует более простой и эффективный алгоритм, называемый алгоритмом Кнута–Морриса–Пратта. Но прежде нам понадобятся некоторые вспомогательные утверждения.

10.3. Вспомогательные утверждения

Для произвольного слова X рассмотрим все его начала, одновременно являющиеся его концами, и выберем из них самое длинное. (Не считая, конечно, самого слова X.) Будем обозначать его l(X).

Примеры: l(aba) = a, l(abab) = ab, l(ababa) = aba, l(abc) = пустое слово.

10.3.1. Доказать, что все слова l(X), l(l(X)), l(l(l(X))) и т.д. являются началами слова X.

Решение. Каждое из них (согласно определению) является началом предыдущего.

По той же причине все они являются концами слова X.

10.3.2. Доказать, что последовательность предыдущей задачи обрывается (на пустом слове).

Решение. Каждое слово короче предыдущего.

10.3.3. Доказать, что любое слово, одновременно являющееся началом и концом слова X (кроме самого X) входит в последовательность $l(X), l(l(X)), \ldots$

Решение. Пусть слово Y есть одновременно начало и конец X. Слово l(X) — самое длинное из таких слов, так что Y не длиннее l(X). Оба эти слова являются началами X, поэтому более короткое из них является началом более длинного: Y есть начало l(X). Аналогично, Y есть конец l(X). Рассуждая по индукции, можно предполагать, что утверждение задачи верно для всех слов короче X, в частности, для слова l(X). Так что слово Y, являющееся концом и началом l(X), либо равно l(X), либо входит в последовательность $l(l(X)), l(l(l(X))), \ldots$, что и требовалось доказать.

10.4. Алгоритм Кнута–Морриса–Пратта

Алгоритм Кнута–Морриса–Пратта (КМП) получает на вход слово

$$\mathtt{X} = \mathtt{x[1]x[2]}\ldots\mathtt{x[n]}$$

и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел `l[1]`...`l[n]`, где

$$\mathtt{l[i]} = \text{длина слова } \mathtt{l(x[1]}\ldots\mathtt{x[i])}$$

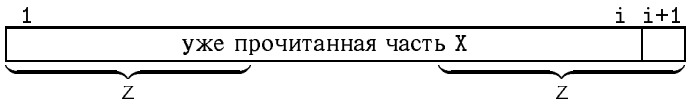
(функция `l` определена в предыдущем пункте). Словами: `l[i]` есть длина наибольшего начала слова `x[1]`...`x[i]`, одновременно являющегося его концом.

10.4.1. Какое отношение всё это имеет к поиску подслова? Другими словами, как использовать алгоритм КМП для определения того, является ли слово `A` подсловом слова `B`?

Решение. Применим алгоритм КМП к слову `A#B`, где `#` — специальная буква, не встречающаяся ни в `A`, ни в `B`. Слово `A` является подсловом слова `B` тогда и только тогда, когда среди чисел в массиве `l` будет число, равное длине слова `A`.

10.4.2. Описать алгоритм заполнения таблицы `l[1]`...`l[n]`.

Решение. Предположим, что первые `i` значений `l[1]`...`l[i]` уже найдены. Мы читаем очередную букву слова (т.е. `x[i+1]`) и должны вычислить `l[i+1]`.



Другими словами, нас интересуют начала Z слова `x[1]`...`x[i+1]`, одновременно являющиеся его концами — из них нам надо выбрать самое длинное. Откуда берутся эти начала? Каждое из них (не считая пустого) получается из некоторого слова Z' приписыванием буквы `x[i+1]`. Слово

Z' является началом и концом слова x[1]...x[i]. Однако не любое слово, являющееся началом и концом слова x[1]...x[i], годится — надо, чтобы за ним следовала буква x[i+1].

Получаем такой рецепт отыскания слова Z. Рассмотрим все начала слова x[1]...x[i], являющиеся одновременно его концами. Из них выберем подходящие — те, за которыми идет буква x[i+1]. Из подходящих выберем самое длинное. Приписав в его конец x[i+1], получим искомое слово Z.

Теперь пора воспользоваться сделанными нами приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l из предыдущего раздела. Вот что получается:

```
i:=1; l[1]:= 0;
{таблица l[1]..l[i] заполнена правильно}
while i <> n do begin
| len := l[i]
| {len - длина начала слова x[1]..x[i], которое является
|    его концом; все более длинные начала оказались
|    неподходящими}
| while (x[len+1] <> x[i+1]) and (len > 0) do begin
| | {начало не подходит, применяем к нему функцию l}
| | len := l[len];
| end;
| {нашли подходящее или убедились в отсутствии}
| if x[len+1] = x[i+1] do begin
| | {x[1]..x[len] - самое длинное подходящее начало}
| | l[i+1] := len+1;
| end else begin
| | {подходящих нет}
| | l[i+1] := 0;
| end;
| i := i+1;
end;
```

10.4.3. Доказать, что число действий в приведённом только что алго-

ритме не превосходит Cn для некоторой константы C.

Решение. Это не вполне очевидно: обработка каждой очередной буквы может потребовать многих итераций во внутреннем цикле. Однако каждая такая итерация уменьшает `len` по крайней мере на 1, и в этом случае `l[i+1]` окажется заметно меньше `l[i]`. С другой стороны, при увеличении `i` на единицу величина `l[i]` может возрасти не более чем на 1, так что часто и сильно убывать она не может — иначе убывание не будет скомпенсировано возрастанием.

Более точно, можно записать неравенство

$$\texttt{l[i+1]} \leqslant \texttt{l[i]} - (\text{число итераций на } \texttt{i}\text{-м шаге}) + 1$$

или

$$(\text{число итераций на } \texttt{i}\text{-м шаге}) \leqslant \texttt{l[i]} - \texttt{l[i+1]} + 1$$

Остаётся сложить эти неравенства по всем `i` и получить оценку сверху для общего числа итераций.

10.4.4. Будем использовать этот алгоритм, чтобы выяснить, является ли слово `X` длины `n` подсловом слова `Y` длины `m`. (Как это делать с помощью специального разделителя `#`, описано выше.) При этом число действий будет не более $C(\texttt{n} + \texttt{m})$, и используемая память тоже. Придумать, как обойтись памятью не более Cn (что может быть существенно меньше, если искомый образец короткий, а слово, в котором его ищут — длинное).

Решение. Применяем алгоритм КМП к слову `A#B`. При этом вычисление значений `l[1]`, ..., `l[n]` проводим для слова `X` длины `n` и запоминаем эти значения. Дальше мы помним только значение `l[i]` для текущего `i` — кроме него и кроме таблицы `l[1]...l[n]`, нам для вычислений ничего не нужно.

На практике слова `X` и `Y` могут не находиться подряд, поэтому просмотр слова `X` и затем слова `Y` удобно оформить в виде разных циклов. Это избавляет также от хлопот с разделителем.

10.4.5. Написать соответствующий алгоритм (проверяющий, является ли слово $\texttt{X} = \texttt{x[1]}\ldots\texttt{x[n]}$ подсловом слова $\texttt{Y} = \texttt{y[1]}\ldots\texttt{y[m]}$).

Решение. Сначала вычисляем таблицу `l[1]...l[n]` как раньше. Затем пишем такую программу:

```
j:=0; len:=0;
{len - длина максимального начала слова X, одновременно
       являющегося концом слова y[1]..j[j]}
while (len <> n) and (j <> m) do begin
| while (x[len+1] <> y[j+1]) and (len > 0) do begin
| | {начало не подходит, применяем к нему функцию l}
| | len := l[len];
| end;
| {нашли подходящее или убедились в отсутствии}
| if x[len+1] = y[j+1] do begin
| | {x[1]..x[len] - самое длинное подходящее начало}
| | len := len+1;
| end else begin
| | {подходящих нет}
| | len := 0;
| end;
| j := j+1;
end;
{если len=n, слово X встретилось; иначе мы дошли до конца
   слова Y, так и не встретив X}
```

10.5. Алгоритм Бойера–Мура

Этот алгоритм делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный образец. Как так может быть? Идея проста. Пусть, например, мы ищем образец `abcd`. Посмотрим на четвертую букву слова: если, к примеру, это буква `e`, то нет никакой необходимости читать первые три буквы. (В самом деле, в образце буквы `e` нет, поэтому он может начаться не раньше пятой буквы.)

Мы приведем самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть `x[1]...x[n]` — образец, который надо искать. Для каждого символа `s` найдем самое правое его вхождение в слово `X`, то есть наибольшее `k`, при котором `x[k]` = `s`. Эти сведения будем хранить в массиве `pos[s]`; если символ `s` вовсе не встре-

чается, то нам будет удобно положить `pos[s] = 0` (мы увидим дальше, почему).

10.5.1. Как заполнить массив `pos`?

Решение.

```
положить все pos[s] равными 0
for i:=1 to n do begin
  pos[x[i]]:=i;
end;
```

В процессе поиска мы будем хранить в переменной `last` номер буквы в слове, против которой стоит последняя буква образца. Вначале `last = n` (длина образца), затем `last` постепенно увеличивается.

```
last:=n;
{все предыдущие положения образца уже проверены}
while last <= m do begin {слово не кончилось}
| if x[m] <> y[last] then begin {последние буквы разные}
| | last := last + (n - pos[y[last]]);
| | {n - pos[y[last]]  - это минимальный сдвиг образца,
| |    при котором напротив y[last] встанет такая же
| |    буква в образце. Если такой буквы нет вообще,
| |    то сдвигаем на всю длину образца}
| end else begin
| | если нынешнее положение подходит, т.е. если
| | x[1]..x[n] = y[last-n+1]..y[last],
| | то сообщить о совпадении;
| | last := last+1;
| end;
end;
```

Знатоки рекомендуют проверку совпадения проводить справа налево, т.е. начиная с последней буквы образца (в которой совпадение заведомо есть). Можно также немного сэкономить, произведя вычитание заранее и храня не `pos[s]`, а `n-pos[s]`, т.е. число букв в образце справа от последнего вхождения буквы `s`.

Возможны разные модификации этого алгоритма. Например, можно строку `last:=last+1` заменить на `last:=last+(n-u)`, где `u` — координата второго справа вхождения буквы `x[n]` в образец.

10.5.2. Как проще всего учесть это в программе?

Решение. При построении таблицы `pos` написать

```
for i:=1 to n-1 do...
```

(далее как раньше), а в основной программе вместо `last:=last+1` написать

```
last:= last+n-pos[y[last]];
```

Приведенная нами упрощённый вариант алгоритма Бойера–Мура в некоторых случаях требует существенно больше `n` действий (число действий порядка `mn`), проигрывая алгоритму Кнута–Морриса–Пратта.

10.5.3. Привести пример ситуации, в которой образец не входит в слово, но авлгоритму требуется порядка `mn` действий, чтобы это установить.

Решение. Пусть образец имеет вид `baaa...aa`, а само слово состоит только из букв `a`. Тогда на каждом шаге несоответствие выясняется лишь в последний момент.

Настоящий (не упрощённый) алгоритм Бойера–Мура гарантирует, что число действий не првосходит $C(m+n)$ в худшем случае. Он использует идеи, близкие к идеям алгоритма Кнута–Морриса–Пратта. Представим себе, что мы сравнивали образец со входным словом, идя справа налево. При этом некоторый кусок Z (являющийся концом образца) совпал, а затем обнаружилось различие: перед Z в образце стоит не то, что во входном слове. Что можно сказать в этот момент о входном слове? В нем обнаружен фрагмент, равный Z, а перед ним стоит не та буква, что в образце. Эта информация может позволить сдвинуть образец на несколько позиций вправо без риска пропустить его вхождение. Эти сдвиги следует вычислить заранее для каждого конца Z нашего образца. Как говорят знатоки, всё это (вычисление таблицы сдвигов и её использование) можно уложить в $C(m+n)$ действий.

10.6. Алгоритм Рабина

Этот алгоритм основан на простой идее. Представим себе, что в слове длины m мы ищем образец длины n. Вырежем окошечко размера n и будем двигать его по входному слову. Нас интересует, не совпадает ли слово в окошечке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определённую на словах длины n. Если значения этой функции на слове в окошечке и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

Что мы выигрываем при таком подходе? Казалось бы, ничего — ведь чтобы вычислить значение функции на слове в окошечке, всё равно нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, и вот за счёт чего. При сдвиге окошечка слово не меняется полностью, а лишь добавляется буква в конце и убирается в начале. Хорошо бы, чтобы по этим данным можно было рассчитать, как меняется функция.

10.6.1. Привести пример удобной для вычисления функции.

Решение. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошечка нужно добавить новое число и вычесть пропавшее.)

Для каждой функции существуют слова, к которым она применима плохо. Зато другая функция в этом случае может работать хорошо. Возникает идея: надо запасти много функций и в начале работы алгоритма выбирать из них случайную. (Тогда враг, желающий подгадить нашему алгоритму, не будет знать, с какой именно функцией ему бороться.)

10.6.2. Привести пример семейства удобных функций.

Решение. Выберем некоторое число p (желательно простое, смотри далее) и некоторый вычет x по модулю p. Каждое слово длины n будем рассматривать как последовательность целых чисел (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени $n-1$ и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя функция). Сдвиг окошка на 1 соответствует

вычитанию старшего члена (x^{n-1} следует вычислить заранее), умножению на x и добавлению свободного члена.

Следующее соображение говорит в пользу того, что совпадения не слишком вероятны. Пусть число p фиксировано и к тому же простое, а X и Y — два различных слова длины n. Тогда им соответствуют различные многочлены (мы предполагаем, что коды всех букв различны — это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, то есть их разность обращается в 0. Разность есть многочлен степени $n-1$ и имеет не более $n-1$ корней. Таким образом, если n много меньше p, то случайному x мало шансов попасть в неудачную точку.

10.7. Более сложные образцы и автоматы

Мы можем искать не конкретное слово, а подслова заданного вида. Например, можно искать слова вида `a?b`, где вместо `?` может стоять любая буква (иными словами, нас интересует буква `b` на расстоянии 2 после буквы `a`).

10.7.1. Указать конечный автомат, проверяющий, есть ли во входном слове фрагмент вида `a?b`.

Решение. Читая слово, следует помнить, есть ли буква `a` на последнем месте и на предпоследнем — пока не встретим искомый фрагмент. Автомат имеет состояния 00, 01, 10, 11, их смысл таков:

00 на предпоследнем и последнем местах нет `a`
01 на предпоследнем нет, на последнем есть
10 не предпоследнем есть, на последнем нет
11 есть и там, и там

Таблица переходов автомата:

Текущее состояние	Очередная буква	Новое состояние
00	a	01
00	не a	01
01	a	11
01	не a	10
10	a	01
10	b	найдено
10	не a и не b	00
11	a	11
11	b	найдено
11	не a и не b	10

Другой стандартный знак в образце — это звездочка (*), на место которой может быть подставлено любое слово. Например, образец `ab*cd` означает, что мы ищем подслово `ab`, за которым следует что угодно, а затем (на любом расстоянии) идёт `cd`.

10.7.2. Указать конечный автомат, проверяющий, есть ли во входном слове образец `ab*cd` (в описанном только что смысле).

Решение.

Текущее состояние	Очередная буква	Новое состояние
начальное	a	a
начальное	не a	начальное
a	b	ab
a	a	a
a	не a и не b	начальное
ab	c	abc
ab	не c	ab
abc	d	найдено
abc	c	abc
abc	не c и не d	ab

Еще один вид поиска — это поиск любого из слов некоторого списка.

10.7.3. Дан список слов $X_1, \ldots, X_k$ и слово Y. Определить, входит ли хотя бы одно из слов X_i в слово Y (как подслово). Количество действий не должно превосходить константы, умноженной на суммарную длину всех слов (из списка и того, в котором происходит поиск).

Решение. Очевидный способ состоит в том, чтобы каждое слово из списка проверять отдельно (с помощью одного из рассмотренных алгоритмов). Однако при этом мы не укладываемся в заданное число действий (из-за умножения k на длину слова Y).

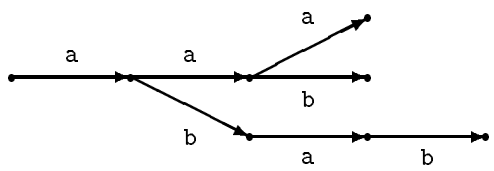
Посмотрим на дело с другой стороны. Каждому образцу из списка соответствует конечный автомат с некоторым множеством состояний. Эти автоматы можно объединить в один, множеством состояний которого будет произведение множеств состояний всех тех автоматов. Это — очень большое множество. Однако на самом деле большинство его элементов недоступны (не могут появиться при чтении входного слова) и за счет этого получается экономия. Примерно эту идею (но в изменённом виде) мы и будем использовать.

Вспомним алгоритм Кнута–Морриса–Пратта. В нём, читая входное слово, мы хранили наибольшее начало образца, являющееся концом прочитанной части. Теперь нам следует хранить для каждого из образцов наибольшее его начало, являющееся концом прочитанной части. Решающим оказывается такое замечание: достаточно хранить самое длинное из них — все остальные по нему восстанавливаются (как наибольшие начала образцов, являющиеся его концами).

Склеим все образцы в дерево, объединив их совпадающие начальные участки. Например, набору образцов

$$\{\mathtt{aaa}, \mathtt{aab}, \mathtt{abab}\}$$

соответствует дерево



Формально говоря, вершинами дерева являются все начала всех образцов, а сыновья вершины получаются приписыванием буквы.

Читая входное слово, мы двигаемся по этому дереву: текущая вершина — это наибольшая (самая правая) из вершин, являющихся концом прочитанной части (= наибольший конец прочитанной части, являющийся началом одного из образцов).

Определим функцию l, аргументами и значениями которой являются вершины дерева. Именно, $l(P)$ = наибольшая вершина дерева, являющаяся концом P. (Напомним, вершины дерева — это слова.) Нам понадобится такое утверждение:

10.7.4. Пусть P — вершина дерева. Докажите, что множество всех вершин, являющихся концами P, равно $\{l(P), l(l(P)), \ldots\}$

Решение. См. доказательство аналогичного утверждения для алгоритма Кнута–Морриса–Пратта.

Теперь ясно, что нужно делать, находясь в вершине P и читая букву z входного слова. Надо просматривать последовательно вершины P, $l(P)$, $l(l(P)),\ldots$, пока не обнаружится такая, из которой выходит стрелка с буквой z. Та вершина, в которую эта стрелка ведёт, и будет нашим следующим положением.

Остается понять, как для каждой вершины дерева вычислить указатель на значение функции l в этой вершине. Это делается как раньше, при этом значения l для более коротких слов используются при вычислении очередного значения функции l. Это означает, что вершины дерева следует просматривать в порядке возрастания их длины. Нетрудно понять, что всё

это можно уложить в требуемое число действий (хотя константа зависит от числа букв в алфавите). Относящиеся к этому подробности см. в главе 9.

Можно поинтересоваться, какие свойства слов распознаются с помощью конечных автоматов. Оказывается, что существует просто описываемый класс образцов, задающий все такие свойства — класс *регулярных выражений*.

Определение. Пусть фиксирован конечный алфавит Γ, не содержащий символов Λ, ε, (,), $*$ и | (они будут использоваться для построения регулярных выражений и не должны перемешиваться с буквами). Регулярные выражения строятся по таким правилам:

(а) буква алфавита Γ — регулярное выражение;

(б) символы Λ, ε — регулярные выражения;

(в) если $A, B, C, \ldots, E$ — регулярные выражения, то $(ABC\ldots E)$ — регулярное выражение;

(г) если $A, B, C, \ldots, E$ — регулярные выражения, то $(A|B|C|\ldots|E)$ — регулярное выражение;

(д) если A — регулярное выражение, то $A*$ — регулярное выражение.

Каждое регулярное выражение задает множество слов в алфавите Γ по таким правилам:

(а) букве соответствует одноэлементное множество, состоящее из однобуквенного слова, состоящего из этой буквы;

(б) символу ε соответствует пустое множество, а символу Λ — одноэлементное множество, единственным элементом которого является пустое слово;

(в) регулярному выражению $(ABC\ldots E)$ соответствует множество всех слов, которые можно получить, если к слову из A приписать слово из B, затем из C,..., затем из E (*конкатенация* множеств);

(г) регулярному выражению $(A|B|C|\ldots|E)$ соответствует объединение множеств, соответствующих выражениям $A, B, C, \ldots, E$;

(д) регулярному выражению $A*$ соответствует *итерация* множества, соответствующего выражению A, то есть множество всех слов, которые можно так разрезать на куски, что каждый кусок принадлежит множеству, соответствующему выражению A. (В частности, пустое слово всегда содержится в $A*$.)

Множества, соответствующие регулярным выражениям, называются *регулярными*. Вот несколько примеров:

Выражение	Множество
`(a|b)*`	все слова из букв `a` и `b`
`(aa)*`	слова из чётного числа букв `a`
`(Λ|a|b|aa|ab|ba|bb)`	любое слово длины не более 2 из букв `a,b`

10.7.5. Написать регулярное выражение, которому соответствует множество всех слов из букв `a` и `b`, в которых число букв `a` чётно.

Решение. Выражение `b*` задает все слова без буквы `a`, а выражение `(b* a b* a b*)` — все слова ровно с двумя буквами `a`. Остается объединить эти множества, а потом применить итерацию:

```
( (b* a b* a b*) | b* )*
```

10.7.6. Написать регулярное выражение, которое задаёт множество всех слов из букв `a, b, c`, в которых слово `bac` является подсловом.

Решение. `((a|b|c)* bac (a|b|c)*)`

Теперь задачу о поиске образца в слове можно переформулировать так: проверить, принадлежит ли слово множеству, соответствующему данному регулярному выражению.

10.7.7. Какие выражения соответствуют образцам `a?b` и `ab*cd`, рассмотренным ранее? (В образце символ `*` используется не в том смысле, что в регулярных выражениях!) Предполается, что алфавит содержит буквы `a, b, c, d, e`.

Решение.

```
((a|b|c|d|e)*a (a|b|c|d|e) b (a|b|c|d|e)*)
```

```
((a|b|c|d|e)*ab(a|b|c|d|e)*cd(a|b|c|d|e)*)
```

10.7.8. Доказать, что для всякого регулярного выражения можно построить конечный автомат, который распознает соответствующее этому выражению множество слов.

Решение. Нам потребуется новое понятие — понятие *источника*, или *недетерминированного конечного автомата*. Представим себе ориентированный граф — картинку из нескольких точек (вершин) и некоторых стрелок, соединяющих эти точки (рёбер). Пусть на некоторых рёбрах написаны буквы (не обязательно на всех). Пусть также среди вершин выбраны две — начальная H и конечная K. Такая картинка называется *источником*.

Будем двигаться различными способами из H в K, читая буквы по дороге (на тех стрелках, где они есть). Каждому пути из H в K, таким образом, соответствует некоторое слово. А источнику в целом соответствует множество слов — тех слов, которые можно прочесть на путях из H в K.

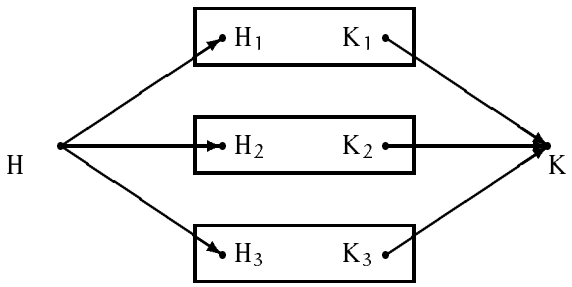
Замечание. Если нарисовать состояния конечного автомата в виде точек, а переходы при чтении букв изобразить в виде стрелок, то станет ясно, что конечный автомат — это частный случай источника. (С дополнительными требованиями: (а) на всех стрелках есть буквы, и (б) для любой точки на выходящих из неё стрелках каждая буква встречается ровно один раз.)

Мы будем строить конечный автомат по регулярному выражению в два приёма. Сначала мы построим источник, которому соответствует то же самое множество слов. Затем для произвольного источника построим автомат, который проверяет, принадлежит ли слово соответствующему множеству.

10.7.9. По регулярному выражению построить источник, задающий то же множество.

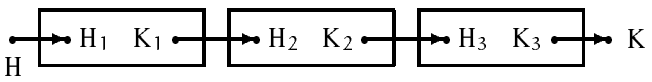
Решение. Индукция по построению регулярного выражения. Буквам

соответствуют графы из одной стрелки. Объединение реализуется так:

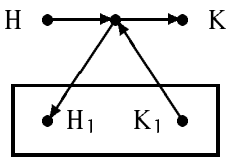


Нарисована картинка для объединения трёх множеств, прямоугольники — это источники, им соответствующие; указаны начальные и конечные вершины. На новых стрелках (их 6) букв не написано.

Конкатенации соответствует картинка



Наконец, итерации соответствует картинка



10.7.10. Дан источник. Построить конечный автомат, проверяющий, принадлежит ли входное слово соответствующему множеству (то есть можно ли прочесть это слово, идя из H в K).

Решение. Состояниями автомата будут множества вершин источника. Именно, прочтя некоторое начало X входного слова, мы будем помнить множество всех вершин источника, в которые можно пройти из начальной, прочитав на пути слово X.

Оказывается, что регулярные выражения, автоматы и источники распознают одни и те же множества. Чтобы убедиться в этом, нам осталось решить такую задачу:

10.7.11. Дан источник. Построить регулярное выражение, задающее то же множество, что и этот источник.

Решение. (Сообщено участниками просеминара по логике.) Пусть источник имеет вершины $1, \ldots, k$. Будем считать, что 1 — это начало, а k — конец. Через $D_{i,j,s}$ обозначим множество всех слов, которые можно прочесть на пути из i в j, если в качестве промежуточных пунктов разрешается использовать только вершины $1, \ldots, s$. Согласно определению, источнику соответствует множество $D_{1,k,k}$.

Индукцией по s будем доказывать регулярность всех множеств $D_{i,j,s}$ при всех i и j. При $s = 0$ это очевидно (промежуточные вершины запрещены, поэтому каждое из множеств состоит только из букв).

Из чего состоит множество $D_{i,j,s+1}$? Отметим на пути моменты, в которых он заходит в $(s+1)$-ую вершину. При этом путь разбивается на части, каждая из которых уже не заходит в неё. Поэтому легко сообразить, что

$$D_{i,j,s+1} = D_{i,j,s} \mid (D_{i,s+1,s} \quad D_{s+1,s+1,s}* \quad D_{s+1,j,s})$$

(вольность записи: мы используем для операций над множествами обозначения как для регулярных выражений). Остаётся воспользоваться предположением индукции.

10.7.12. Где ещё используется то же самое рассуждение?

Ответ. В алгоритме Флойда вычисления цены кратчайшего пути, см. главу 9.

10.7.13. Доказать, что класс множеств, задаваемых регулярными выражениями, не изменился бы, если бы мы разрешили использовать не только объединение, но и отрицание (а следовательно, и пересечение — оно выражается через объединение и отрицание).

Решение. Для автоматов переход к отрицанию очевиден.

Замечание. На практике важную роль играет число состояний автомата. Оказывается, что тут всё не так просто, и переход от источника к автомату требует экспоненциального роста числа состояний. Подробное рассмотрение связанных с этим теоретических и практических вопросов — дело особое (см. книгу Ахо, Ульмана и Сети о компиляторах).

11. Представление множеств. Хеширование

11.1. Хеширование с открытой адресацией

В главе 6 было указано несколько представлений для множеств, элементами которых являются целые числа произвольной величины. Однако в любом из них хотя бы одна из операций проверки принадлежности, добавления и удаления элемента требовала количества действий, пропорционального числу элементов множества. На практике это бывает слишком много. Существуют способы, позволяющие получить для всех трёх упомянутых операций оценку $C \log n$. Один из таких способов мы рассмотрим в следующей главе. В этой главе мы разберем способ, которые хотя и приводит к Cn действиям в худшем случае, но зато «в среднем» требует значительно меньшего их числа. (Мы не будем уточнять слов «в среднем», хотя это и можно сделать.) Этот способ называется *хешированием*.

Пусть нам необходимо представлять множества элементов типа `T`, причем число элементов в них заведомо меньше `n`. Выберем некоторую функцию `h`, определенную на значениях типа `T` и принимающую значения `0...n-1`. Было бы хорошо, чтобы эта функция принимала на элементах будущего множества по возможности более разнообразные значения. (Худший случай — это когда её значения на всех элементах хранимого множества одинаковы.) Эту функцию будем называть *хеш-функцией*, или, как ещё говорят, *функцией расстановки*.

Введём два массива

```
val:  array [0..n-1] of T;
used: array [0..n-1] of boolean;
```

(мы позволяем себе писать `n-1` в качестве границы в определении типа, хотя в паскале это не разрешается). В этих массивах будут храниться элементы множества: оно равно множеству всех `val[i]` для тех `i`, для которых `used[i]`, причём все эти `val[i]` различны. По возможности мы будем хранить элемент `t` на месте `h(t)`, считая это место «исконным» для элемента `t`. Однако может случиться так, что новый элемент, который мы хотим добавить, претендует на уже занятое место (для которого `used` истинно). В этом случае мы отыщем ближайшее справа свободное место и запишем элемент туда. («Справа» значит «в сторону увеличения индексов»;

дойдя до края, мы перескакиваем в начало.) По предположению, число элементов всегда меньше `n`, так что пустые места заведомо будут.

Формально говоря, в любой момент должно соблюдаться такое требование: для любого элемента множества участок справа от его исконного места до его фактического места полностью заполнен.

Благодаря этому проверка принадлежности заданного элемента `t` осуществляется легко: встав на `h(t)`, двигаемся направо, пока не дойдём до пустого места или до элемента `t`. В первом случае элемент `t` отсутствует в множестве, во втором — присутствует. Если элемент отсутствует, то его можно добавить на найденное пустое место. Если присутствует, то можно его удалить (положив `used` = `false`).

11.1.1. В предыдущем абзаце есть ошибка. Найдите её и исправьте.

Решение. Дело в том, что при удалении требуемое свойство «отсутствия пустот» может нарушиться. Поэтому будем делать так. Создав дыру, будем двигаться направо, пока не натолкнёмся на элемент, стоящий не на исконном месте, или на ещё одно пустое место. Во втором случае на этом можно успокоиться. В первом случае посмотрим, не нужно ли найденный элемент поставить на место дыры. Если нет, то продолжаем поиск, если да, то затыкаем им старую дыру. При этом образуется новая дыра, с которой делаем всё то же самое.

11.1.2. Написать программы проверки принадлежности, добавления и удаления.

Решение.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| принадлежит := used [i] and (val [i] = t);
end;
```

```
procedure добавить (t: T);
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if not used [i] then begin
| | used [i] := true;
| | val [i] := t;
| end;
end;

procedure исключить (t: T);
| var i, gap: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if used [i] and (val [i] = t) then begin
| | used [i] := false;
| | gap := i;
| | i := (i + 1) mod n;
| | {gap - дыра, которая может закрыться одним из i,i+1,...}
| | while used [i] do begin
| | | if i = h (val[i]) then begin {на своем месте}
| | | | i := (i + 1) mod n;
| | | end else if dist(h(val[i]),i) < dist(gap,i) then begin
| | | | {gap...h(val[i])...i}
| | | | i := (i + 1) mod n;
| | | end else begin
| | | | used [gap] := true;
| | | | val [gap] := val [i];
| | | | used [i] := false;
```

```
        gap := i;
        i := (i + 1) mod n;
      end;
    end;
  end;
end;
```

Здесь `dist(a,b)` — измеренное по часовой стрелке (слева направо) расстояние от `a` до `b`, то есть

$$\texttt{dist(a,b)} = \texttt{(b-a+n) mod n}.$$

(Мы прибавили `n`, так как функция `mod` правильно работает при положительном делимом.)

11.1.3. Существует много вариантов хеширования. Один из них таков: обнаружив, что исконное место (обозначим его i) занято, будем искать свободное не среди $i+1, i+2, \ldots$, а среди $r(i), r(r(i)), r(r(r(i))), \ldots$, где r — некоторое отображение $\{0, \ldots, n-1\}$ в себя. Какие при этом будут трудности?

Ответ. (1) Не гарантируется, что если пустые места есть, то мы их найдем. (2) При удалении неясно, как заполнять дыры. (На практике во многих случаях удаление не нужно, так что такой способ также применяется. Считается, что удачный подбор функции r может предотвратить образование «скоплений» занятых ячеек.)

11.1.4. Пусть для хранения множества всех правильных русских слов в программе проверки орфографии используется хеширование. Что нужно добавить, чтобы к тому же уметь находить английский перевод любого правильного слова?

Решение. Помимо массива `val`, элементы которого являются русскими словами, нужен параллельный массив их английских переводов.

11.2. Хеширование со списками

На хеш-функцию с k значениями можно смотреть как на способ свести вопрос о хранении одного большого множества к вопросу о хранении нескольких меньших. Именно, если у нас есть хеш-функция с k значениями,

то любое множество разбивается на k подмножеств (возможно, пустых), соответствующих возможным значениям хэш-функции. Вопрос о проверке принадлежности, добавлении или удалении для большого множества сводится к такому же вопросу для одного из меньших (чтобы узнать, для какого, надо посмотреть на значение хеш-функции).

Эти меньшие множества удобно хранить с помощью ссылок; их суммарный размер равен числу элементов хешируемого множества. Следующая задача предлагает реализовать этот план.

11.2.1. Пусть хеш-функция принимает значения `1...k`. Для каждого значения хеш-функции рассмотрим список всех элементов множества с данным значением хеш-функции. Будем хранить эти `k` списков с помощью переменных

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

так же, как мы это делали для `k` стеков ограниченной суммарной длины. Напишите соответствующие программы. (Удаление по сравнению с открытой адресацией упрощвется.)

Решение. Перед началом работы надо положить `Вершина[i] = 0` для всех `i = 1...k`, и связать все места в список свободного пространства, положив `ПервСвоб = 1` и `Следующий[i] = i+1` для `i = 1...n-1`, а также `Следующий[n] = 0`.

```
function принадлежит (t: T): boolean;
| var i: integer;
begin
| i := Вершина[h(t)];
| {осталось искать в списке, начиная с i}
| while (i <> 0) and (Содержание[i] <> t) do begin
| | i := Следующий[i];
| end; {(i=0) or (Содержание [i] = t)}
| принадлежит := (i<>0) and (Содержание[i]=t);
end;
```

```
procedure добавить (t: T);
| var i: integer;
begin
| if not принадлежит(t) then begin
| | i := ПервСвоб;
| | {ПервСвоб <> 0 - считаем, что не переполняется}
| | ПервСвоб := Следующий[ПервСвоб]
| | Содержание[i]:=t;
| | Следующий[i]:=Вершина[h(t)];
| | Вершина[h(t)]:=i;
| end;
end;

procedure исключить (t: T);
| var i, pred: integer;
begin
| i := Вершина[h(t)]; pred := 0;
| {осталось искать в списке, начиная с i;  pred -
|    предыдущий, если он есть, и 0, если нет}
| while (i <> 0) and (Содержание[i] <> t) do begin
| | pred := i; i := Следующий[i];
| end; {(i=0) or (Содержание [i] = t)}
| if i <> 0 then begin
| | {Содержание[i]=t, элемент есть, надо удалить}
| | if pred = 0 then begin
| | | {элемент оказался первым в списке}
| | | Вершина[h(t)] := Следующий[i];
| | end else begin
| | | Следующий[pred] := Следующий[i]
| | end;
| | {осталось вернуть i в список свободных}
| | Следующий[i] := ПервСвоб;
| | ПервСвоб:=i;
| end;
```

```
end;
```

11.2.2. (Для знакомых с теорией вероятностей.) Пусть хеш-функция с k значениями используется для хранения множества, в котором в данный момент n элементов. Доказать, что математическое ожидание числа действий в предыдущей задаче не превосходит $C(1 + n/k)$, если добавляемый (удаляемый, искомый) элемент t выбран случайно, причём все значения $h(t)$ имеют равные вероятности (равные $1/k$).

Решение. Если $l(i)$ — длина списка, соответствующего хеш-значению i, то число операций не превосходит $C(1 + l(h(t)))$; усредняя, получаем искомый ответ, так как $\sum_i l(i) = n$.

Эта оценка основана на предположении о равных вероятностях. Однако в конкретной ситуации всё может быть совсем не так, и значения хеш-функции могут «скучиваться»: для каждой конкретной хеш-функции есть «неудачные» ситуации, когда число действий оказывается большим. Приём, называемый *универсальным хешированием*, позволяет обойти эту проблему. Идея состоит в том, что берётся семейство хеш-функций, причем любая ситуация оказывается неудачной лишь для небольшой части этого семества.

Пусть H — семейство функций, каждая из которых отображает множество T в множество из k элементов (например, $0 \ldots k-1$). Говорят, что H — *универсальное семейство хеш-функций*, если для любых двух различных значений s и t из множества T вероятность события $h(s) = h(t)$ для случайной функции h из семейства H равна $1/k$. (Другими словами, те функции из H, для которых $h(s) = h(t)$, составляют $1/k$-ую часть всех функций в H.)

Замечание. Более сильное требование к семейству H могло бы состоять в том, чтобы для любых двух различных элементов s и t множества T значения $h(s)$ и $h(t)$ случайной функции h являются независимыми случайными величинами, равномерно распределёнными на $0 \ldots k-1$.

11.2.3. Пусть $t_1, \ldots, t_n$ — произвольная последовательность различных элементов множества T. Рассмотрим количество действий, происходящих при помещении элементов $t_1, \ldots, t_n$ в множество, хешируемое с помощью функции h из универсального семейства H. Доказать, что сред-

нее количество действий (усреднение — по всем h из H) не превосходит $Cn(1 + n/k)$.

Решение. Обозначим через m_i количество элементов последовательности, для которых хеш-функция равна i. (Числа $m_0, \ldots, m_{k-1}$ зависят, конечно, от выбора хеш-функции.) Количество действий, которое мы хотим оценить, с точностью до постоянного множителя равно $m_0^2 + m_1^2 + \ldots + m_{k-1}^2$. (Если s чисел попадают в одну хеш-ячейку, то для этого требуется примерно $1 + 2 + \ldots + s$ действий.) Эту же сумму квадратов можно записать как число пар $\langle p, q\rangle$, для которых $h(t_p) = h(t_q)$. Последнее равенство, если его рассматривать как событие при фиксированных p и q, имеет вероятность $1/k$ при $p \neq q$, поэтому среднее значение соответствующего члена суммы равно $1/k$, а для всей суммы получаем оценку порядка n^2/k, а точнее $n + n^2/k$, если учесть члены с $p = q$.

Эта задача показывает, что на каждый добавляемый элемент приходится в среднем $C(1 + n/k)$ операций. В этой оценке дробь n/k имеет смысл «коэффициента заполнения» хеш-таблицы.

11.2.4. Доказать аналогичное утверждение для произвольной последовательности операций добавления, поиска и удаления (а не только для добавления, как в предыдущей задаче).

[Указание. Будем представлять себе, что в ходе поиска, добавления и удаления элемент проталкивается по списку своих коллег с тем же хеш-значением, пока не найдёт своего двойника или не дойдёт до конца списка. Будем называть i-j-столкновением столкновение t_i с t_j. (Оно либо произойдёт, либо нет — в зависимости от h.) Общее число действий примерно равно числу всех происшедших столкновений плюс число элементов. При $t_i \neq t_j$ вероятность i-j-столкновения не превосходит $1/k$. Осталось проследить за столкновениями между равными элементами. Фиксируем некоторое значение x из множества T и посмотрим на связанные с ним операции. Они идут по циклу: добавление — проверки — удаление — добавление — проверки — удаление — ... Столкновения происходят между добавляемым элементом и следующими за ним проверками (до удаления включительно), поэтому общее их число не превосходит числа элементов, равных x.]

Теперь приведем примеры универсальных семейств. Очевидно, для лю-

бых конечных множеств A и B семейство всех функций, отображающих A в B, является универсальным. Однако этот пример с практической точки зрения бесполезен: для запоминания случайной функции из этого семейства нужен массив, число элементов в котором равно числу элементов в множестве A. (А если мы можем себе позволить такой массив, то никакого хеширования не требуется!)

Более практичные примеры универсальных семейств могут быть построены с помощью несложных алгебраических конструкций. Через $\mathbb{Z}_p$ мы обозначаем множество вычетов по простому модулю p, т.е. $\{0, 1, \ldots, p-1\}$; арифметические операции в этом множестве выполняются по модулю p. Универсальное семейство образуют все линейные функционалы на $\mathbb{Z}_p^n$ со значениями в $\mathbb{Z}_p$. Более подробно, пусть $a_1, \ldots, a_n$ — произвольные элементы $\mathbb{Z}_p$; рассмотрим отображение

$$h : \langle x_1 \ldots x_n \rangle \mapsto a_1 x_1 + \ldots + a_n x_n$$

Мы получаем семейство из p^n отображений $\mathbb{Z}_p^n \to \mathbb{Z}_p$ параметризованное наборами $\langle a_1 \ldots a_n \rangle$.

11.2.5. Доказать, что это семейство является универсальным.

[Указание. Пусть x и y — различные точки пространства $\mathbb{Z}_p^n$. Какова вероятность того, что случайный функционал принимает на них одинаковые значения? Другими словами, какова вероятность того, что он равен нулю на их разности $x - y$? Ответ дается таким утверждением: пусть u — ненулевой вектор; тогда все значения случайного функционала на нём равновероятны.]

В следующей задаче множество $\mathbb{B} = \{0, 1\}$ рассматривается как множество вычетов по модулю 2.

11.2.6. Семейство всех линейных отображений из $\mathbb{B}^n$ в $\mathbb{B}^m$ является универсальным.

Родственные хешированию идеи неожиданно оказываются полезными в следующей ситуации (рассказал Д. Варсанофьев). Пусть мы хотим написать программу, которая обнаруживала (большинство) опечаток в тексте, но не хотим хранить список всех правильных словоформ. Предлагается

поступить так: выбрать некоторое N и набор функций $f_1, \ldots, f_k$, отображающих русские слова в $1 \ldots N$. В массиве из N битов положим все биты равными нулю, кроме тех, которые являются значением какой-то функции набора на какой-то правильной словоформе. Теперь приближённый тест на правильность словоформы таков: проверить, что значения всех функций набора на этой словоформе попадают на места, занятые единицами. (Этот тест может не заметить некоторых ошибок, но все правильные словоформы будут одобрены.)

12. Представление множеств. Деревья. Сбалансированные деревья

12.1. Представление множеств с помощью деревьев

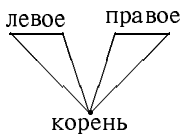
Полное двоичное дерево. T-деревья

Нарисуем точку. Из нее проведём две стрелки (влево вверх и вправо вверх) в две другие точки. Из каждой из этих точек проведём по две стрелки и так далее. Полученную картинку (в n-ом слое будет 2^{n-1} точек) называют полным двоичным *деревом*. Нижнюю точку называют *корнем*. У каждой вершины есть два *сына* (две вершины, в которые идут стрелки) — *левый* и *правый*. У всякой вершины, кроме корня, есть единственный *отец*.

Пусть выбрано некоторое конечное множество вершин полного двоичного дерева, содержащее вместе с каждой вершиной и всех ее предков. Пусть на каждой вершине этого множества написано значение фиксированного типа T (то есть задано отображение множества вершин в множество значений типа T). То, что получится, будем называть T-деревом. Множество всех T-деревьев обозначим Tree(T).

Рекурсивное определение. Всякое непустое T-дерево разбивается на три части: корень (несущий пометку из T), левое и правое поддеревья (которые могут быть пустыми). Это разбиение устанавливает взаимно однозначное соответствие между множеством непустых T-деревьев и произведением $T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T)$. Обозначив через empty пустое дерево, можно написать

$$\mathrm{Tree}(T) = \{\mathrm{empty}\} + T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T).$$



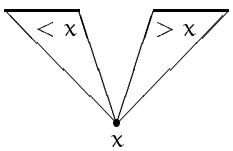
Поддеревья. Высота

Фиксируем некоторое T-дерево. Для каждой его вершины x определено ее *левое поддерево* (левый сын вершины x и все его потомки), *правое*

поддерево (правый сын вершины x и все его потомки) и *поддерево с корнем в* x (вершина x и все её потомки). Левое и правое поддеревья вершины x могут быть пустыми, а поддерево с корнем в x всегда непусто (содержит по крайней мере x). *Высотой* поддерева будем считать максимальную длину цепи $y_1 \ldots y_n$ его вершин, в которой y_{i+1} — сын y_i для всех i. (Высота пустого дерева равна нулю, высота дерева из одного корня — единице.)

Упорядоченные T-деревья

Пусть на множестве значений типа T фиксирован порядок. Назовем T-дерево *упорядоченным*, если выполнено такое свойство: для любой вершины x все пометки в ее левом поддереве меньше пометки в x, а все пометки в ее правом поддереве больше пометки в x.



12.1.1. Доказать, что в упорядоченном дереве все пометки различны.

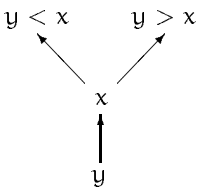
[Указание. Индукция по высоте дерева.]

Представление множеств с помощью деревьев

Каждое дерево будем считать представлением множества всех пометок на его вершинах. При этом одно и то же множество может иметь различные представления.

Благодаря упорядоченности каждый элемент может легко «найти своё место» в дереве: придя в какую-то вершину и сравнив себя с тем, кто там

находится, элемент решает, идти ему налево или направо.



Начав с корня и двигаясь по этому правилу, он либо обнаружит, что такой элемент уже есть, либо найдет место, в котором он должен быть.

Всюду далее мы предполагаем, что на значениях типа T задан порядок, и рассматриваем только упорядоченные деревья.

Хранение деревьев в программе

Можно было бы сопоставить вершины полного двоичного дерева с числами $1, 2, 3, \ldots$ (считая, что левый сын n есть $2n$, правый сын n есть $2n+1$) и хранить пометки в массиве `val [1...]`. Однако этот способ неэкономен, поскольку тратится место на хранение пустых вакансий в полном двоичном дереве.

Более экономен такой способ. Введем три массива

```
val: array [1..n] of T;
left, right: array [1..n] of 0..n;
```

(`n` — максимальное возможное число вершин дерева) и переменную `root:0..n`. Каждая вершина хранимого T-дерева будет иметь номер — число от `1` до `n`. Разные вершины будут иметь разные номера. Пометка в вершине с номером `x` равна `val[x]`. Корень имеет номер `root`. Если вершина с номером `i` имеет сыновей, то их номера равны `left[i]` и `right[i]`. Отсутствующим сыновьям соответствует число `0`. Аналогичным образом значение `root=0` оответствует пустому дереву.

Для хранения дерева используется лишь часть массива; для тех `i`, которые свободны (не являются номерами вершин), значения `val[i]` безразличны. Нам будет удобно, чтобы все свободные числа были «связаны в список»: первое хранится в специальное переменной `free:0..n`, а следующее за `i` свободное число хранится в `left[i]`, так что свободны числа