Ministerul Educației al Republicii Moldova

Анатол Гремальски

ИНФОРМАТИКА

Учебник для 11 класса

Știința, 2014

CZU 004(075.3)
Г 70

Elaborat conform curriculumului disciplinar în vigoare și aprobat prin Ordinul ministrului educației al Republicii Moldova (nr. 267 din 11 aprilie 2014). Editat din sursele financiare ale *Fondului Special pentru Manuale*.

Comisia de evaluare: *Gheorghe Curbet*, profesor școlar, grad didactic superior, Liceul Teoretic „Mihai Eminescu", Bălți; *Arcadie Malearovici*, șef direcție, Centrul Tehnologiilor Informaționale și Comunicaționale în Educație, MET; *Varvara Vanovscaia*, profesor școlar, grad didactic superior, Liceul Teoretic „Vasile Alecsandri", Chișinău

Recenzenți: *Gheorghe Căpățână*, doctor-inginer, conferențiar universitar, Universitatea de Stat din Moldova; *Alexei Colîbneac*, maestru în arte, profesor universitar, Academia de Muzică, Teatru și Arte Plastice, Chișinău; *Tatiana Cartaleanu*, doctor în filologie, conferențiar universitar, Universitatea Pedagogică de Stat „Ion Creangă", Chișinău; *Mihai Șleahtițchi*, doctor în psihologie și în pedagogie, conferențiar universitar, Universitatea Liberă Internațională din Moldova; *Valeriu Cabac*, doctor în științe fizico-matematice, conferențiar universitar, Universitatea de Stat „Alecu Russo", Bălți

Traducere din limba română: *Veronica Musteață (capit. 1, 2, 3); Arcadie Malearovici (capit. 4, 5, 6, 7)*
Responsabil de ediție: *Valentina Rîbalchina*
Redactor: *Tatiana Bolgar*
Redactor tehnic: *Nina Duduciuc*
Machetare computerizată: *Anatol Andrițchi*
Copertă: *Vitalie Ichim*

Întreprinderea Editorial-Poligrafică *Știința*,
str. Academiei, nr. 3; MD-2028, Chișinău, Republica Moldova;
tel.: (+373 22) 73-96-16; fax: (+373 22) 73-96-27;
e-mail: prini@stiinta.asm.md; prini_stiinta@yahoo.com
www.stiinta.asm.md

DIFUZARE:
Republica Moldova: ÎM Societatea de Distribuție a Cărții *PRO-NOI*
str. Alba-Iulia, 75; MD-2051, Chișinău;
tel.: (+373 22) 51-68-17, 71-96-74; fax: (+373 22) 58-02-68;
e-mail: info@pronoi.md; www.pronoi.md

Toate drepturile asupra acestei ediții aparțin Întreprinderii Editorial-Poligrafice *Știința*.

Descrierea CIP a Camerei Naționale a Cărții

Гремальски, Анатол
Информатика: Учеб. для 11 класса/Анатол Гремальски; trad. din lb. rom.: Veronica Musteață, Arcadie Malearovici; Min. Educației al Rep. Moldova. – Ch.: Î.E.P. *Știința*, 2014 (Tipografia „BALACRON"). – 192 p.

ISBN 978-9975-67-932-9
004(075.3)

ISBN 978-9975-67-932-9

© *Anatol Gremalschi*. 2008, 2014
© Traducere: *Veronica Musteață*, *Arcadie Malearovici*.
© Î.E.P. *Știința*. 2008, 2014

СОДЕРЖАНИЕ

Оглавление	Гуманитарный	Реальный	Факультативный	Страница
Введение				4
Глава 1. **ФУНКЦИИ И ПРОЦЕДУРЫ**				5
1.1. Подпрограммы	•	•		5
1.2. Функции	•	•		6
1.3. Процедуры	•	•		10
1.4. Области видимости	•	•		15
1.5. Связь через глобальные переменные	•	•		18
1.6. Побочные эффекты		•		21
1.7. Рекурсия		•		24
1.8. Синтаксис описаний и вызовов подпрограмм	•	•		27
Глава 2. **ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ**				31
2.1. Динамические переменные. Ссылочный тип		•		31
2.2. Структуры данных		•		35
2.3. Односвязные списки		•		36
2.4. Обработка односвязных списков		•		41
2.5. Стеки		•		48
2.6. Очереди		•		52
2.7. Двоичные деревья		•		56
2.8. Обход двоичных деревьев		•		63
2.9. Деревья *m*-го порядка			•	68
2.10. Тип данных `pointer`			•	73
Глава 3. **МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ**				81
3.1. Модульное программирование			•	81
3.2. Проверка и отладка программ			•	88
3.3. Элементы структурного программирования			•	91
Глава 4. **АНАЛИЗ АЛГОРИТМОВ**				94
4.1. Сложность алгоритмов		•		94
4.2. Оценка объема памяти		•		96
4.3. Измерение времени выполнения алгоритма		•		101
4.4. Оценка времени выполнения алгоритма		•		105
4.5. Временная сложность алгоритмов		•		110
Глава 5. **МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ**				114
5.1. Итерация или рекурсия?		•		114
5.2. Метод полного перебора		•		119
5.3. Метод Greedy – “жадный” алгоритм		•		123
5.4. Метод перебора с возвратом		•		128
5.5. Метод “разделяй и властвуй”			•	135
5.6. Метод динамического программирования			•	142
5.7. Метод *ветвей и границ*			•	146
5.8. Применения метода *ветвей и границ*			•	149
5.9. Точные и эвристические алгоритмы			•	161
Глава 6. **АЛГОРИТМЫ РЕШЕНИЯ НЕКОТОРЫХ МАТЕМАТИЧЕСКИХ ЗАДАЧ**				172
6.1. Операции над множествами			•	172
6.2. Комбинаторный анализ			•	178
Глава 7. **ЗАДАЧИ НА ПОВТОРЕНИЕ**	•	•		185

Введение

Данный учебник, написанный в соответствии с Куррикулумом по информатике, содержит знания, необходимые учащимся для формирования информационной культуры и развития алгоритмического мышления. Он знакомит учащихся с функциями и процедурами языка ПАСКАЛЬ, динамическими структурами данных и методами разработки программных продуктов, представляет к изучению наиболее распространенные методы программирования: полный перебор, метод Greedy, перебор с возвратом, метод "разделяй и властвуй", метод динамического программирования, метод ветвей и границ, эвристические алгоритмы, методы оценки объема памяти и времени, дает понятие о рекурсивных и итеративных алгоритмах.

В начале учебника вводятся синтаксис и семантика изучаемых конструкций языка ПАСКАЛЬ, за которыми следуют примеры их применения и рекомендации по написанию программ, предназначенных для непосредственного выполнения на компьютере. При изучении методов программирования изложены базовые понятия и математические основы соответствующих подходов, после чего приведены способы организации данных, описания алгоритмов, рекомендации по разработке и отладке программ на языке ПАСКАЛЬ. Особое внимание уделено взаимозависимости между способами практического внедрения изучаемых методов программирования, производительностью компьютера и сложностью решаемых на нем задач. Реализация наиболее широко известных методов программирования иллюстрируется на примере задач, которые часто встречаются в современной жизни и изучаются в рамках других предметов лицейского цикла.

Темы учебника тесно связаны со знаниями из других областей общественной жизни, науки и техники. Они ориентированы на решение задач, требующих больших объемов вычислений, что подчеркивает существенную роль математического мышления в появлении и развитии информатики. Примеры, упражнения и индивидуальные задания, включенные в учебник, помогут лучше понять роль и место компьютера, его влияние на развитие математики, физики, химии, социально-гуманитарных наук. Для большинства тем учебника были разработаны обучающие программы, что позволяет индивидуализировать процесс обучения, организовать практические занятия и развивать творческие способности каждого ученика.

Материал учебника ориентирован на развитие следующих компетенций: структурный анализ задачи; деление сложных задач на более простые и сведение их к уже ранее решенным; оценка сложности алгоритмов, предназначенных для решения предложенных задач; использование формальных методов для разработки алгоритмов, написания и отладки соответствующих программ. Очевидно, что перечисленными навыками должны обладать не только будущие специалисты по информатике, но и каждый культурно развитый человек, который будет жить и работать в обществе, основанном на широком применении информационных технологий.

Автор

Глава 1

ФУНКЦИИ И ПРОЦЕДУРЫ

1.1. Подпрограммы

Любая сложная задача может быть решена путем ее разбиения на ряд подзадач. Для решения каждой подзадачи записывается соответствующая последовательность операторов, называемая **подпрограммой**. Таким образом, задача решается с помощью основной программы, в которой для решения подзадач вызываются подпрограммы. Когда в основной программе встречается обращение к подпрограмме, управление передается первому оператору вызванной подпрограммы (*рис. 1.1*). После завершения выполнения операторов подпрограммы управление передается оператору основной программы, следующему за оператором вызова подпрограммы.

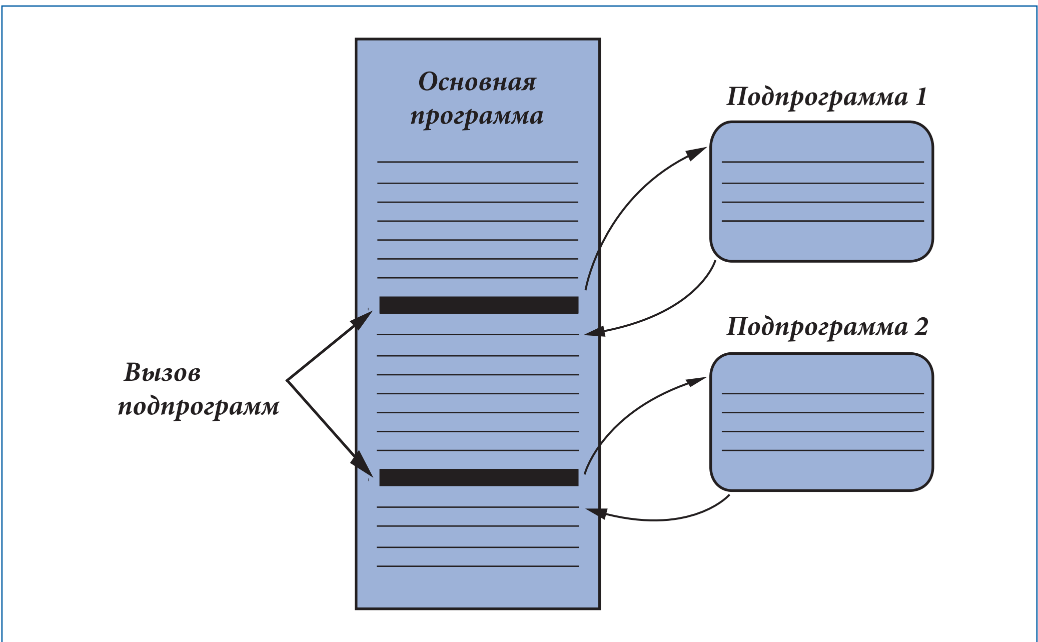


Рис. 1.1. Взаимодействие между программой и подпрограммой

В языке ПАСКАЛЬ существуют два вида подпрограмм: функция и процедура.

<*Подпрограммы*> ::= {< *Функция* >; |< *Процедура* >;}

Функция – это подпрограмма, которая вычисляет и возвращает некоторое значение. Язык ПАСКАЛЬ содержит ряд стандартных функций, известных любой программе: `sin`, `cos`, `eof` и т.д. Помимо этого программист может создавать собственные функции, к которым можно обращаться так же, как и к стандартным функциям. Таким образом, концепция функции расширяет понятие **выражения** в языке ПАСКАЛЬ.

Процедура – это подпрограмма, которая осуществляет обработку данных, переданных в момент обращения. В языке ПАСКАЛЬ существуют стандартные процедуры: `read`, `readln`, `write`, `writeln` и т. д. Помимо этого программист может создавать собственные процедуры, к которым можно обращаться так же, как и к стандартным процедурам. Таким образом, концепция процедуры расширяет понятие **оператора** в языке ПАСКАЛЬ.

Подпрограммы объявляются в разделе описаний программы. Обращение к функциям и процедурам производится в разделе операторов.

Подпрограмма может обращаться и к самой себе, такое обращение называется **рекурсивным**.

Вопросы и упражнения

❶ Объясните термины *основная программа* и *подпрограмма*.

❷ Как взаимодействуют программа и подпрограмма?

❸ В чем разница между процедурами и функциями?

❹ Как происходит обращение к функции? В каких операторах языка может появиться обращение к функции?

❺ Как происходит обращение к процедуре?

❻ Назовите тип аргумента и тип результата стандартных функций `abs`, `chr`, `eof`, `eoln`, `exp`, `ord`, `sin`, `sqr`, `sqrt`, `pred`, `succ`, `trunc`.

❼ Назовите тип фактических параметров процедур `read` и `write`.

❽ Как обрабатываются данные с помощью процедур `read` и `write`?

1.2. Функции

На языке ПАСКАЛЬ **функция описывается** следующим образом:

```
function f(x1, x2, ..., xn) : tr;
D;
  begin
    ...
    f := e;
    ...
  end;
```

Первая строка – это заголовок функции, состоящий из:

f – имя функции;

$(x_1, x_2, \ldots; x_n)$ – произвольный список формальных параметров, являющихся аргументами функции;

t_r – тип результата; может быть простым или ссылочным.

За заголовком следует **тело функции**, состоящее из произвольных локальных описаний D и составного оператора **begin** … **end**.

Локальные описания состоят из следующих разделов (некоторые из которых могут отсутствовать): **label, const, type, var, function, procedure.**

Имя функции f должно появиться хотя бы один раз в левой части некоторого оператора присваивания: $f := e$. Последнее значение, присвоенное функции f, возвращается в основную программу.

Как правило, формальный параметр из списка $(x_1, x_2, \ldots, x_n)$ имеет вид:

$$v_1, v_2, \ldots, v_k : t_p$$

где $v_1, v_2, \ldots, v_k$ – идентификаторы, а t_p – имя типа.

Обращение к функции f осуществляется следующим образом:

$$f(a_1, a_2, \ldots, a_n)$$

где $(a_1, a_2, \ldots, a_n)$ – **список фактических параметров**. Обычно, фактическими параметрами являются выражения, значения которых передаются функции. Связь между фактическим и формальным параметрами определяется их позицией в рассматриваемых списках. Фактический параметр должен быть **совместимым** с точки зрения присваивания с типом формального параметра.

Пример:

```
Program P97;
  {Описание и использование функции Putere}
  type Natural=0..MaxInt;
  var a : real;
      b : Natural;
      c : real;
      s : integer;
      t : integer;
      v : real;

  function Putere(x : real; n : Natural) : real;
   {вычисление x в степени n }
  var p : real;
      i : integer;
  begin
    p:=1;
    for i:=1 to n do p:=p*x;
    Putere:=p;
  end; { Putere }

  begin
    a:=3.0;
```

```
  b:=2;
  c:=Putere(a, b);
  writeln(a:10:5, b:4, c:10:5);          s:=2;
  t:=4;
  v:=Putere(s, t);
  writeln(s:5, t:4, v:10:5);
  readln;
end.
```

Функция `Putere` имеет два формальных параметра: `x` типа `real` и `n` типа `Natural`. Функция возвращает значение типа `real`. В теле функции описаны локальные переменные `p` и `i`.

При обращении к функции `Putere (a, b)` значения `3.0` и `2` фактических параметров `a, b` передаются формальным параметрам – соответственно `x` и `n`. Отметим, что тип параметра `a` совпадает с типом параметра `x`, а тип `b` – с типом `n`.

В случае обращения к функции `Putere(s, t)` тип фактических параметров `s, t` не совпадает с типом формальных параметров – соответственно `x` и `n`. Однако такое обращение корректно, так как соответствующие типы совместимы с точки зрения присваивания.

Вопросы и упражнения

❶ Даны следующие описания:

```
function Factorial(n : integer) : integer;
var p, i : integer;
begin
  p:=1;
  for i:=1 to n do p:=p * i;
  Factorial:=p;
end;
```

Назовите тип формального параметра и тип результата, возвращаемого функцией. Укажите переменные, описанные в теле функции. Напишите программу, которая выводит на экран значения *n*! для *n* = 2, 3 и 7.

❷ В какой части основной программы находятся описания функций?

❸ Прокомментируйте следующую программу:

```
Program P98;
 { Ошибка }
function Factorial(n : 0..7) : integer;
var p, i : integer;
begin
 p:=1;
for i:=1 to n do p:=p*i;
Factorial:=p;
end; { Factorial }
```

```
begin
  writeln(Factorial(4));
  readln;
end.
```

❹ Дан заголовок:

```
function F(x : real; y : integer; z : char) : boolean;
```

Какие из следующих вызовов функций корректны?

a) `F(3.18, 4, 'a')`

b) `F(4, 4, '4')`

c) `F(4, 4, 4)`

d) `F(4, 3.18, 'a')`

e) `F(3.18, 4, 4)`

f) `F('3.18', 4, '4')`

g) `F(15, 21, '3')`

h) `F(15,21,3)`

❺ Напишите функцию, которая возвращает:

а) сумму действительных чисел *a, b, c, d;*
б) среднее арифметическое целых чисел *i, j, k, m;*
в) минимальное из чисел *a, b, c, d;*
г) количество гласных в строке символов;
д) количество согласных в строке символов;
е) корень уравнения $ax + b = 0$;
ж) наименьший делитель целого числа $n>0$, отличный от 1;
з) наибольший общий делитель натуральных чисел *a* и *b*;
и) общее наименьшее кратное натуральных чисел *a* и *b*;
к) последнюю цифру в десятичном представлении целого числа $n > 0$;
л) сумму цифр в десятичном представлении целого числа $n > 0$;
м) наибольшую цифру в десятичном представлении целого числа $n > 0$;
н) количество появлений заданного символа в строке символов.

❻ Даны следующие описания:

```
const nmax=100;
type Vector=array [1..nmax] of real;
```

Напишите функцию, которая возвращает:

а) сумму элементов некоторого массива типа `Vector`;
б) среднее арифметическое элементов данного массива;
в) максимальный элемент;
г) минимальный элемент.

❼ Даны следующие типы данных:

```
type Punct=record
             x, y : real
           end;
```

```
      Segment=record
               A, B : Punct
              end;
      Triunghi=record
                A, B, C : Punct
               end;
      Dreptunghi=record
                  A, B, C, D : Punct
                 end;
      Cerc=record
             Centru : Punct;
             Raza : real
           end;
```

Напишите функцию, которая возвращает:

а) длину отрезка;

б) длину окружности;

в) площадь круга;

г) площадь треугольника;

д) площадь прямоугольника.

❽ Переменная A вводится с помощью описания:

```
var A : set of char;
```

Напишите функцию, которая возвращает количество символов в множестве A.

❾ Напишите функцию, которая вычисляет в секундах разность между двумя моментами времени, заданными в часах, минутах и секундах.

❿ Треугольник задан координатами своих вершин. Напишите функции, которые для двух заданных треугольников проверяют:

а) одинаковы ли площади данных треугольников;

б) являются ли треугольники подобными;

в) находится ли первый треугольник внутри второго.

1.3. Процедуры

Процедура описывается следующим образом:

```
procedure p(x1, x2, ..., xn);
 D;
begin
...
end;
```

где

p – имя процедуры;

$(x_1, x_2, \ldots; x_n)$ – произвольный список формальных параметров.

Тело процедуры включает:

D – произвольные локальные описания, сгруппированные таким же образом, как и в случае функций;

begin ... **end** – составной оператор, в котором имя процедуры не появляется в операциях присваивания.

Процедура может возвращать несколько результатов, но не через свое имя, а через специально предназначенные для этого переменные (следующие за словом **var**) из списка формальных параметров.

Параметры из списка, вводимые через описания вида

$v_1, v_2, ..., v_k : t_p$

называются **параметрами-значениями**. Они служат для передачи значений из основной программы в процедуру.

Формальные параметры, введенные в список через описания вида

var $v_1, v_2, ..., v_k : t_p$

называются **параметрами-переменными** и служат для возвращения результатов из процедуры в основную программу.

Запуск процедуры выполняется путем ее вызова:

$p(a_1, a_2, ..., a_n)$

где $(a_1, a_2, \ldots, a_n)$ – список **фактических параметров**. В отличие от функции, обращение к процедуре является оператором. Операторы вызова процедур вставляются в основную программу там, где требуется выполнить соответствующую обработку данных.

Для **параметра-значения** в качестве фактического параметра можно использовать любое выражение соответствующего типа, в частности константу или переменную. Изменения параметров-значений не передаются во внешнее окружение подпрограммы.

Для **параметра-переменной** в качестве фактического параметра можно использовать только переменную. Все изменения указанных параметров будут переданы в программу, которая вызвала соответствующую процедуру.

Пример:

```
Program P99;
 {Описание и использование процедуры Lac }
var   a, b, c, t, q : real;

procedure Lac(r : real; var l, s : real);
 {Длина окружности и площадь круга }
 {r - радиус; l - длина; s - площадь }
const Pi=3.14159;
begin
  l:=2*Pi*r;
  s:=Pi*sqr(r);
end; {Lac }
```

```
begin
  a:=1.0;
  Lac(a, b, c);
  writeln(a:10:5, b:10:5, c:10:5);
  Lac(3.0, t, q);
  writeln(3.0:10:5, t:10:5, q:10:5);

  readln;
end.
```

Процедура `Lac` имеет три формальных параметра: `r, l, s`. Параметр `r` является параметром-значением, а `l` и `s` – параметрами-переменными.

При выполнении процедуры `Lac(a, b, c)` значение `1.0` передается формальному параметру `r`, а адреса переменных `b` и `c` передаются формальным параметрам `l` и `s`. Таким образом, последовательность операторов

```
a:=1.0;
Lac(a, b, c)
```

эквивалентна последовательности

```
b:=2*Pi*1.0;
c:=Pi*sqr(1.0).
```

Аналогично оператор

```
Lac(3.0,t,q)
```

эквивалентен последовательности

```
t:=2*Pi*3.0;
q:=Pi*sqr(3.0).
```

Вопросы и упражнения

❶ В чем разница между *параметром-значением* и *параметром- переменной*?

❷ Даны описания:

```
var k, m, n : integer;
    a, b, c : real;
procedure P(i : integer; var j : integer;
            x : real; var y : real);
begin
  {...}
end.
```

Какие из следующих операторов корректны?

a) `P(k,m,a,b)` *b)* `P(3,m,a,b)`

c) `P(k,3,a,b)`

d) `P(m,m,a,b)`

e) `P(m,k,6.1,b)`

f) `P(n,m,6,b)`

g) `P(m,k,6.1,b)`

h) `P(a,m,b,c)`

i) `P(i,i,i,i)`

j) `P(a,a,a,a)`

Аргументируйте ваш ответ.

❸ Прокомментируйте следующую программу:

```
Program P100;
 {Ошибка }
var  a : real;
     b : integer;
procedure P(x : real; var y : integer);
begin
 {... }
end; { P }
begin
 P(a, b);
 P(0.1, a);
 P(1, b);
 P(a, 1);
end.
```

❹ Что будет выведено на экран следующей программой?

```
Program P101;
   {Параметр-значение и параметр-переменная }
var a, b : integer;

procedure P(x : integer; var y : integer);
begin
 x:=x+1;
 y:=y+1;
 writeln('x=', x, ' y=', y);
end; {P }

begin
 a:=0;
 b:=0;
 P(a, b);
 writeln('a=', a, ' b=', b);
 readln;
end.
```

Аргументируйте ваш ответ.

❺ Напишите процедуру, которая:

а) находит корни уравнения $ax^2 + bx + c = 0$;

б) удаляет из строки символ, указанный при вызове процедуры;

в) обрамляет строку символами “#”;

г) упорядочивает элементы массива `array [1..100] of real` в порядке возрастания;

д) упорядочивает элементы файла `file of integer` в порядке убывания;

е) находит и заносит в массив простые числа, меньшие заданного натурального числа *n*.

❻ Рассматриваются следующие типы данных:

```
type Data = record
              Ziua : 1..31;
              Luna : 1..12;
              Anul : integer;
            end;
     Persoana = record
                  NumePrenume : string;
                  DataNasterii : Data;
                end;
     ListaPersoane = array [1..50] of Persoana;
```

Напишите процедуру, которая получает из главной программы список лиц и возвращает:

а) фамилии и имена тех, кто родился в день *z* месяца;

б) фамилии и имена тех, кто родился в месяц *l* года;

в) фамилии и имена тех, кто родился в год *a*;

г) фамилии и имена тех, чья дата рождения *z.l.a*;

д) фамилию и имя самого старшего человека;

е) фамилию и имя самого младшего человека;

ж) возраст каждого человека в годах, месяцах и днях;

з) список лиц старше *v* лет;

и) список лиц в алфавитном порядке;

к) список лиц, упорядоченный согласно дате рождения;

л) список лиц одного возраста (рожденных в одном и том же году).

❼ Напишите процедуру, которая:

а) создает резервную копию любого текстового файла;

б) удаляет из текстового файла пустые строки;

в) находит количество строк в текстовом файле;

г) выполняет слияние двух текстовых файлов в один;

д) выполняет слияние *n* ($n > 2$) текстовых файлов в один.

❽ Назовем *большими* натуральные числа, содержащие более 20 значимых цифр. Введите тип данных для обработки больших натуральных чисел и напишите процедуры для их сложения и вычитания.

1.4. Области видимости

Тело любой программы или подпрограммы называется **блоком**. Поскольку подпрограммы включены в основную программу и, в свою очередь, могут содержать другие подпрограммы, блоки могут быть **вложенными** (включенными один в другой). Такое вложение блоков называется **блочной структурой** программы.

В таких структурах каждому блоку i соответствует некоторый уровень вложенности. Основной программе соответствует уровень вложенности 0, блоку, определенному в основной программе – уровень вложенности 1. Блоку, определенному на уровне n, соответствует уровень вложенности n+1.

В качестве примера на *рис. 1.2* представлена блочная структура программы `P105`.

Как правило, любой блок на языке ПАСКАЛЬ содержит описания меток, переменных, функций, параметров и т.д. При описании вводится имя, которое может быть меткой или идентификатором. Описание, находящееся во внутреннем блоке, может переопределить имя, описанное за его пределами. Таким образом, одно и то же имя в различных частях программы может обозначать различные объекты.

Под **областью видимости** некоторого описания понимается текст программы, в котором введенные имена обозначают объект, определенный данным описанием. Область видимости начинается сразу же после завершения описания и заканчивается вместе с текстом соответствующего блока. Поскольку блоки могут быть вложенными, область видимости не является непрерывной зоной в тексте программы. Область видимости описания из некоторого вложенного блока перекрывает область видимости описания, в котором участвует это же имя, находящееся во внешнем блоке.

Например, в программе `P105` областью видимости описания **`var`** `a: real` является текст, заключенный между пунктами `{1}-{7}`. Область видимости описания **`var`** `c: real` состоит из двух фрагментов текста, заключенных между `{2}, {3}` и `{5}, {6}`. Областью видимости описания **`var`** `c: char` является текст, заключенный между `{4}` и `{5}`.

Для нахождения объекта, обозначенного некоторым именем, необходимо знать области видимости.

Например, в программе `P105` идентификатор `c` из оператора:

```
c:=chr(d)
```

обозначает переменную типа `char`. Тот же самый идентификатор из оператора:

```
c:=b+1
```

обозначает переменную типа `real`.

Напомним, что описание имени функции/процедуры завершается в конце заголовка. Таким образом, область видимости такого описания включает и тело соответствующей функции/процедуры. Это делает возможным **рекурсивный**

вызов: в тело функции/процедуры можно включать вызов ее самой, так как соответствующее имя находится в области видимости. Естественно, описание любого формального параметра является видимым только в теле соответствующей подпрограммы.

```
Program P105;                                {уровень 0}
 { Блочная структура программы }
var a : real;
{1}

   procedure P(b : real);                    {уровень 1}
   var c : real;
   {2}

      procedure Q(d : integer);              {уровень 2}
      {3}
      var c : char;
      {4}
      begin
        c:=chr(d);
        writeln('В процедуре Q  c=', c);
      end; {5}

   begin
     writeln('b=', b);
     c:=b+1;
     writeln('В процедуре P  c=', c);
     Q(35);
   end; {6}


   function F(x : real) : real;               {уровень 1}
   begin
     f:=x/2;
   end;

begin
   a:=F(5);
   writeln('a=', a);
   P(a);
   readln;
end {7}.
```

Рис. 1.2. Блочная структура программы на языке ПАСКАЛЬ

Например, областью видимости описания **procedure** `Q` является текст, находящийся между пунктами `{3}` и `{6}`. Областью видимости описания `d:integer` является текст, находящийся между пунктами `{3}` и `{5}`.

Вопросы и упражнения

❶ Как определяется область видимости некоторого описания?

❷ Определите области видимости описаний `b:real` и `x:real` из программы `P105` (*рис. 1.2*).

❸ Определите блочную структуру программы, представленной ниже. Для каждого описания установите область видимости и определите объекты, которые обозначают идентификаторы *c* и *x* при каждом своем появлении.

```
Program P106;
  {Переопределение констант }
 const c=1;

 function F1(x : integer) : integer;
 begin
  F1:=x+c;
 end; { F1 }

 function F2(c : real) : real;
 const x=2.0;
 begin
  F2:=x+c;
 end; { F2 }

 function F3(x : char) : char;
 const c=3;
 begin
  F3:=chr(ord(x)+c);
 end; { F3 }

 begin
  writeln('F1=', F1(1));
  writeln('F2=', F2(1));
  writeln('F3=', F3('1'));
  readln;
 end.
```

Что выведет на экран данная программа?

❹ Определите области видимости идентификаторов `P` и `F` из программы `P105` (*рис. 1.2*).

❺ Прокомментируйте следующую программу:

```
Program P107;
  { Ошибка }
 var a : real;

 procedure P(x : real);
 var a : integer;
```

```
begin
 a:=3.14;
 writeln(x+a);
end; { P }

begin
 a:=3.14;
 P(a);
end.
```

❻ Как определить, какой объект представляет имя, которое появляется в определенном блоке программы на языке ПАСКАЛЬ?

1.5. Связь через глобальные переменные

Вызов подпрограммы подразумевает передачу подлежащих обработке данных вызываемой функции или процедуре. После выполнения последнего оператора подпрограммы полученные результаты необходимо возвратить на место вызова. Мы уже знаем, что подлежащие обработке данные и полученные результаты могут передаваться через параметры. Формальные параметры указываются в заголовке функции или процедуры, а фактические – в той части программы, где производится вызов.

В дополнение к способу передачи данных через параметры, язык ПАСКАЛЬ допускает передачу данных между программой и вызываемыми подпрограммами через глобальные переменные.

Любая переменная является локальной по отношению к подпрограмме в которой она объявлена. Переменная является **глобальной по отношению к подпрограмме,** если она объявляется в основной программе или во внешней подпрограмме, без повторного объявления в рассматриваемой подпрограмме. Так как глобальные переменные известны как в подпрограмме, так и за ее пределами, они могут использоваться для передачи данных, подлежащих обработке, и возвращения результатов.

Пример:

```
Program P108;
  {Связь через глобальные переменные }
 var a,              {переменная глобальная в P }
     b : real;       {переменная глобальная в P, F }

 procedure P;
 var c : integer;   {переменная глобальная в P }
 begin
  c:=2;
  b:=a*c;
 end; { P }
```

```
function F : real;
var a : 1..5;     {переменная локальная в F }
begin
 a:=3;
 F:=a+b;
end; { F }

begin
 a:=1;
 P;
 writeln(b);     {на экран выводится 2.0000000000E+00 }
 writeln(F);     {на экран выводится 5.0000000000E+00 }
 readln;
end.
```

Данные, подлежащие обработке, передаются процедуре `P` через глобальную переменную `a`. Результат, полученный процедурой, возвращается на место вызова через глобальную переменную `b`. Значение аргумента функции `F` передается через глобальную переменную `b`. Отметим, что переменная `a` является локальной для `F` и не может использоваться для передачи данных в эту функцию.

Как правило, связь через глобальные переменные используется в тех случаях, когда несколько подпрограмм обрабатывают одни и те же данные. Например, функции, аргументы которых относятся к типу *массив*; процедуры, которые обрабатывают массивы или файлы с данными о работниках, учениках и т.д.

Вопросы и упражнения

❶ Объясните термины *переменная глобальная по отношению к некоторой подпрограмме* и *переменная локальная в некоторой подпрограмме*.

❷ Назовите глобальные и локальные переменные, описанные в программе `P105` (*рис. 1.2*).

❸ Может ли локальная переменная в то же время являться и глобальной по отношению к какой-либо подпрограмме?

❹ Назовите локальные и глобальные переменные, описанные в следующей программе. Что выводит на экран данная программа?

```
Program P109;
  {Связь через глобальные переменные  }
 var a : integer;

 procedure P;
 var b, c, d : integer;

procedure Q;
  begin
   c:=b+1;
  end; { Q }
```

```
procedure R;
 begin
  d:=c+1;
 end; { R }
begin
 b:=a;
 Q;
 R;
 a:=d;
end; { P }

begin
 a:=1;
 P;
 writeln(a);
 readln;
end.
```

❺ Даны описания:

```
Type  Ora=0..23;
      Grade=-40..+40;
      Temperatura=array [Ora] of Grade;
```

Компоненты переменных типа `Temperatura` представляют собой значения температуры, измеряемой каждый час в течение 24 часов. Напишите процедуру, которая:

а) находит максимальную и минимальную температуры;

б) находит час (часы), когда была зарегистрирована максимальная температура;

в) записывает в текстовый файл час, когда была зарегистрирована минимальная температура.

Связь с соответствующими процедурами осуществляется через глобальные переменные.

❻ Даны произвольные текстовые файлы. Напишите функцию, которая:

а) возвращает количество строк в файле;

б) находит количество гласных в тексте;

в) находит количество слов в тексте (слова представляют собой последовательности символов, разделенных пробелами или символами конца строки);

г) возвращает среднюю длину строк текста;

д) возвращает среднюю длину слов текста;

е) возвращает число знаков пунктуации в тексте.

Связь с соответствующими процедурами осуществляется через глобальные переменные.

1.6. Побочные эффекты

Любая функция возвращает единственный результат – значение функции. Как правило, значения аргументов передаются функции через параметры-значения, а результат возвращается на место вызова через имя функции. Помимо этого, язык ПАСКАЛЬ допускает передачу аргументов через глобальные переменные и параметры-переменные.

Под **побочным эффектом** понимается присваивание (внутри функции) некоторого значения глобальной переменной или формальному параметру-переменной. Побочные эффекты могут непредвиденным образом влиять на ход программы, усложняя тем самым процесс ее отладки.

Ниже представлены примеры программ, в которых используются функции с побочными эффектами.

```
Program P110;
 {Побочный эффект - присваивание глобальной переменной}
 var a : integer; {глобальная переменная}

 function F(x : integer) : integer;
  begin
    F:=a*x;
    a:=a+1;   {присваивание, влекущее за собой побочный эффект}
  end; { F }
begin
    a:=1;
    writeln(F(1));  { на экран выводится 1 }
    writeln(F(1));  { на экран выводится 2 }
    writeln(F(1));  { на экран выводится 3 }
    readln;
 end.
```

В программе `P110` функция `F` возвращает значение выражения `a*x`. Наряду с этим операция присваивания `a:=a+1` изменяет значение глобальной переменной `a`. Таким образом, для одного и того же значения `1` аргумента `x` функция возвращает различные результаты, что противоречит обычному определению функции.

```
Program P111;
 {Побочный эффект - присваивание формальному параметру}
 var a : integer;

 function F(var x : integer) : integer;
      begin
          F:=2*x;
          x:=x+1;    { присваивание, влекущее за собой
                         побочный эффект }
      end; { F }
```

```
begin
 a:=2;
 writeln(F(a));     { на экран выводится 4 }
 writeln(F(a));     { на экран выводится 6 }
 writeln(F(a));     { на экран выводится 8 }
 readln;
end.
```

В программе `P111` функция `F` возвращает значение выражения `2*x`. Так как `x` – формальный параметр-переменная, то операция присваивания `x:=x+1` изменяет значение фактического параметра, указываемого при вызове, т.е. значение переменной `a` из основной программы. Тот факт, что идентичные по тексту обращения `F(a)`, `F(a)` и `F(a)` возвращают различные результаты, может привести к осложнениям в процессе отладки.

Что касается процедур, то здесь операции присваивания над глобальными переменными приводят к таким же побочным эффектам, которые рассматривались в случае функций. Так как стандартным способом возврата результатов процедуры является возврат через формальные параметры-переменные, операции присваивания над такими параметрами не считаются побочными эффектами.

Побочные эффекты вносят отклонения в стандартный процесс связи программа–подпрограмма, при котором используемые переменные указываются явно в качестве формальных параметров в описании и фактических параметров при вызове. Последствия побочных эффектов могут распространяться на области видимости глобальных описаний и взаимодействовать с подобными эффектами, возникающими при выполнении других процедур и функций. В таких условиях использование глобальных переменных становится рискованным. Поэтому для составления сложных программ рекомендуется:

1. Связь функций со средой вызова осуществлять посредством передачи данных в функцию через формальные параметры- значения, возврат единственного результата – через ее имя.
2. Связь процедур со средой вызова осуществлять посредством передачи данных в процедуру через формальные параметры-значения или параметры-переменные, а возврат результатов – через формальные параметры-переменные.
3. Глобальные переменные можно использовать для передачи данных в подпрограммы, однако внутри подпрограмм их значения не должны меняться.

Вопросы и упражнения

❶ В чем причина побочных эффектов? К каким последствиям могут привести указанные эффекты?

❷ Что выводят на экран следующие программы?

```
Program P112;
  {Побочные эффекты}
 var a, b : integer;
```

```
  function F(x : integer) : integer;
    begin
    F:=a*x;
    b:=b+1;
    end; { F }

  function G(x : integer) : integer;
    begin
      G:=b+x;
      a:=a+1;
    end; { G }
begin
   a:=1; b:=1;
   writeln(F(1));
   writeln(G(1));
   writeln(F(1));
   writeln(G(1));
   readln;
end.
```

```
Program P113;
 {Побочные эффекты}
 var a : integer;
     b : real;

  function F(var x : integer) : integer;
    begin
      F:=x;
      x:=x+1;
    end; { F }

  procedure P(x,y:integer; var z:real);
    begin
      z:=x/y;
    end; { P }

begin
   a:=1;
   P(F(a), a, b);
   writeln(a, '    ', b);
   readln;
end.
```

```
Program P114;
  {Побочные эффекты}
 var a, b : real;
```

```
  procedure P(var x, y : real);
  {Обмен значениями между переменными x и y }
    begin
      a:=x;
      x:=y;
      y:=a;
    end; { P }

begin
  a:=1; b:=2;
  P(a, b);
  writeln(a, b);
  a:=3; b:=4;
  P(a, b);
  writeln(a, b);
  readln;
end.
```

❸ Как можно избежать побочных эффектов?

1.7. Рекурсия

Рекурсией называется прямое или косвенное обращение подпрограммы к самой себе. Подпрограмма, которая вызывает саму себя, называется ***рекурсивной***.

Предположим, что объявлен следующий тип

```
type Natural = 0..MaxInt;
```

Функция *факториал*

$$f(n) = \begin{cases} 1, & \text{если } n = 0; \\ n \cdot f(n-1), & \text{если } n > 0 \end{cases}$$

может быть представлена на языке ПАСКАЛЬ, исходя из ее определения, в следующем виде:

```
function F(n : Natural) : Natural;
begin
  if n=0 then F:=1
  else F:=n*F(n-1)
end; {F}
```

Вызов `F(7)` влечет за собой ряд обращений к функции `F` с фактическими параметрами `6, 5, …, 2, 1, 0`:

```
F(7) -> F(6) -> F(5) -> ... -> F(1) -> F(0).
```

При вызове `F(0)` вычисляется непосредственное значение функции, что останавливает процесс повторений; после этого следует процесс возврата и вычисление значений функции `F` для аргументов `1, 2, …, 6, 7`, последнее вычисленное значение `F(7)` возвращается на место первого вызова функции.

Значениями функции

$$fib(n) = \begin{cases} 0, & \text{если } n = 0; \\ 1, & \text{если } n = 1; \\ fib(n-1) + fib(n-2), & \text{если } n > 1 \end{cases}$$

являются числа Фибоначчи. Следуя определению, получаем:

```
function Fib(n:Natural):Natural;
begin
  if n=0 then Fib:=0
  else if n=1 then Fib:=1
              else Fib:=Fib(n-1)+Fib(n-2)
end; {Fib}
```

Каждый вызов функции `Fib` для `n > 1` влечет за собой два вызова `Fib(n-1)`, `Fib(n-2)` и т. д., например:

```
Fib(4) ->
    Fib(3), Fib(2) ->
        Fib(2), Fib(1), Fib(1), Fib(0) ->
            Fib(1), Fib(0).
```

Из данных примеров видно, что рекурсия используется для того, чтобы запрограммировать повторяющиеся вычисления. Повторение осуществляется с помощью подпрограммы, внутри которой есть вызов самой себя: когда процесс выполнения программы достигает соответствующего места, вызывается новое выполнение данной процедуры.

Очевидно, любая рекурсивная подпрограмма должна содержать условие остановки процесса повторения. Например, в случае *факториала* процесс повторений останавливается, когда `n` принимает значение 0; в случае функции `Fib` процесс останавливается, когда `n` принимает значения 0 или 1.

При любом вызове подпрограммы в память компьютера заносится следующая информация:

– текущие значения параметров, передаваемых через значение;

– местонахождение (адреса) параметров-переменных;

– адрес возврата, т. е. адрес оператора, который следует за оператором вызова.

Таким образом, при рекурсивном вызове занимаемая область памяти увеличивается очень быстро, что ведет к риску переполнения памяти компьютера. Этого можно избежать путем замены рекурсии на итерацию (операторы **`for`**, **`while`**, **`repeat`**). В качестве примера представим нерекурсивную форму функции для вычисления *факториала*:

```
function F(n: Natural): Natural;
var i, p : Natural;
begin
   p:=1;
   for i:=1 to n do p:=p*i;
   F:=p;
end; {F}
```

Рекурсию необходимо использовать в случаях, когда написание нерекурсивных алгоритмов является очень сложным: перевод программ на языке ПАСКАЛЬ на язык компьютера, машинная графика, распознавание образов и т. д.

Вопросы и упражнения

❶ Как выполняется рекурсивная подпрограмма? Какая информация заносится в память компьютера при выполнении рекурсивного вызова?

❷ В чем разница между рекурсией и итерацией?

❸ Напишите нерекурсивную функцию Фибоначчи.

❹ Напишите рекурсивную подпрограмму, которая:
а) вычисляет сумму $S(n) = 1 + 3 + 5 + ... + (2n - 1)$;
б) вычисляет произведение $P(n) = 1 \times 4 \times 7 \times ... \times (3n - 2)$;
в) переставляет символы строки в обратном порядке;
г) вычисляет произведение $P(n) = 2 \times 4 \times 6 \times ... \times 2n$.

❺ Напишите программу, которая вводит с клавиатуры натуральные числа *m, n* и выводит на экран значения функции Акерманна:

$$a(m,n) = \begin{cases} n+1, & \text{если } m = 0; \\ a(m-1,1), & \text{если } n = 0; \\ a(m-1, a(m, n-1)), & \text{если } m > 0 \text{ и } n > 0. \end{cases}$$

Вычислите *a(0, 0)*, *a(1, 2)*, *a(2, 1)* и *a(2, 2)*. Попробуйте вычислить *a(4, 4)* и *a(10, 10)*. Объясните сообщения, выводимые на экран.

❻ Дано описание

```
type Vector=array [1..20] of integer;
```

Напишите рекурсивную подпрограмму, которая:
а) выводит на экран компоненты вектора;
б) вычисляет сумму соответствующих компонент;
в) переставляет компоненты вектора в обратном порядке;
г) вычисляет сумму положительных компонент вектора;
д) проверяет, есть ли среди компонент вектора хотя бы одна отрицательная;
е) вычисляет произведение отрицательных компонент;
ж) проверяет, есть ли среди компонент вектора хотя бы одна, равная заданному числу.

❼ Перепишите следующую функцию в нерекурсивном виде:

```
function S(n:Natural):Natural;
 begin
      if n=0 then S:=0
      else S:=n+S(n-1)
 end; {S}
```

❽ Напишите рекурсивную функцию, которая возвращает значение `true`, если строка символов `s` соответствует следующему определению

<Число> ::= *<Цифра>* | *<Цифра>* *<Число>*

Указание. Рекурсивная форма такой функции непосредственно вытекает из металингвистической формулы. Нерекурсивный вариант

```
function N(s : string) : boolean;
var    i : integer;
       p : boolean
begin
       p:=(s<>'');
       for i=1 to length(s) do
       p:=p and (s[i] in ['0'..'9']);
       N:=p;
end;
```

следует из определения

<Число> ::= *<Цифра>* {*<Цифра>*}

❾ Даны следующие металингвистические формулы:

<Число> ::= *<Цифра>* {*<Цифра>*}
<Знак> ::= + | –
<Выражение> ::= *<Число>* | *<Выражение>* *<Знак>* *<Выражение>*

Напишите рекурсивную функцию, которая возвращает значение `true`, если строка символов `s` соответствует определению лексической единицы *<Выражение>*.

1.8. Синтаксис описаний и вызовов подпрограмм

В общем случае описание функций осуществляется с помощью следующих металингвистических формул:

<Функция> ::= *<Заголовок функции>*; *<Блок>*
| *<Заголовок функции>*; *<Директива>*
| **function** *<Идентификатор>*; *<Блок>*

<Заголовок функции> ::=
function *<Идентификатор>* [*<Список формальных параметров>*] : *<Идентификатор>*

Синтаксические диаграммы представлены на *рис. 1.3*.

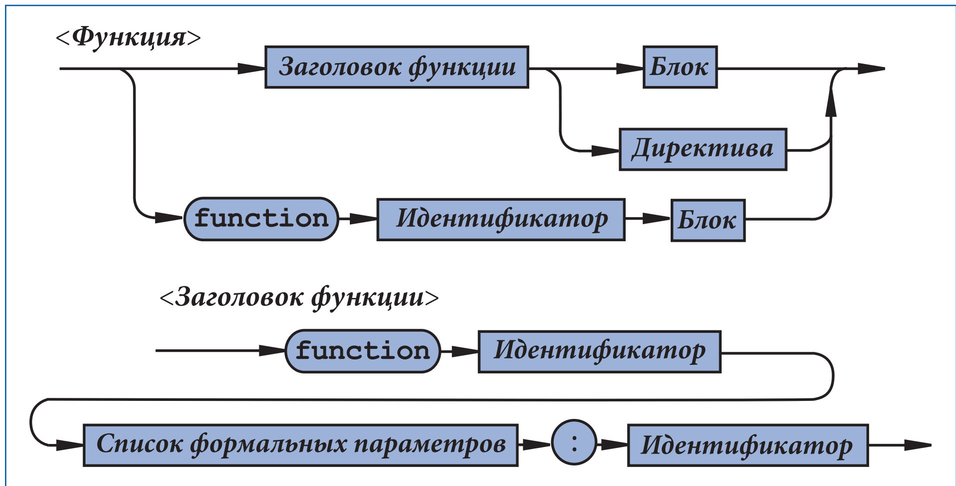


Рис. 1.3. Синтаксис описания функции

Процедуры описываются с помощью следующих формул:

<Процедура>::=*<Заголовок процедуры>*`;`*<Блок>*
| *<Заголовок процедуры>*`;`*<Директива>*
| **`procedure`** *<Идентификатор>*`;`*<Блок>*

<Заголовок процедуры> := **`procedure`** *<Идентификатор>* [*<Список формальных параметров>*]

Синтаксические диаграммы представлены на *рис. 1.4*.

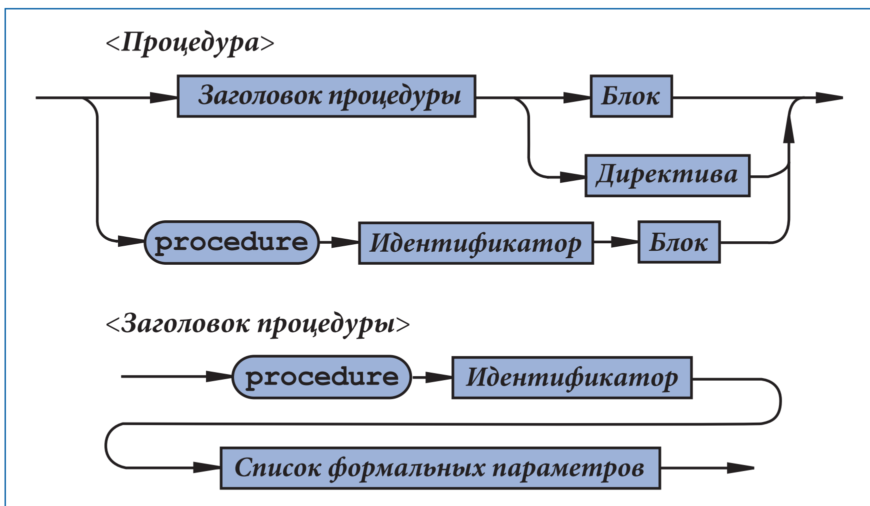


Рис. 1.4. Синтаксис описания процедуры

Список формальных параметров имеет синтаксис:

<Список формальных параметров> ::=
(*<Формальный параметр>* {`;` *<Формальный параметр>*})

<Формальный параметр> ::=
[**var**] *<Идентификатор>* {, *<Идентификатор>*} **:** *<Идентификатор>*
| *<Заголовок функции>*
| *<Заголовок процедуры>*

Синтаксическая диаграмма представлена на *рис. 1.5*.

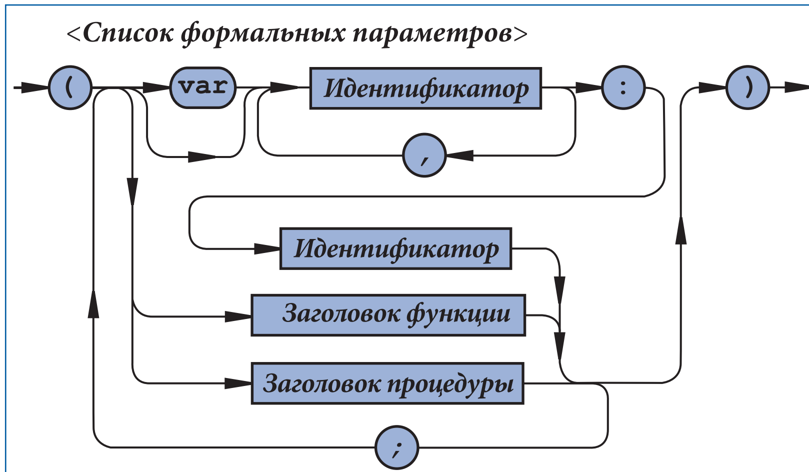


Рис. 1.5. Синтаксическая диаграмма *<Список формальных параметров>*

Отметим, что при отсутствии ключевого слова **var** идентификаторы из списка указывают **параметры-значения**. Слово **var** указывает **параметры-переменные**. Заголовок некоторой функции (процедуры) указывает на **параметр-функцию** (**процедуру**). В Turbo PASCAL такие параметры описываются явно, относятся к **процедурному типу** и имеют вид параметров-значений. Язык ПАСКАЛЬ расширяет обычный смысл понятия функции, допуская возврат значений не только через имя функции, но и через параметры-переменные.

Оператор **вызова функции** имеет вид:

<Вызов функции> ::= *<Имя функции>* [*<Список фактических параметров>*]

а оператор **вызова процедуры**:

<Вызов процедуры> ::= *<Имя процедуры>* [*<Список фактических параметров>*]

Фактические параметры описываются с помощью формул:

<Список фактических параметров> ::=
(*<Фактический параметр>* {,*<Фактический параметр>*})

<Фактический параметр> ::=*<Выражение>* | *<Переменная>*
| *<Имя функции>* | *<Имя процедуры>*

Синтаксическая диаграмма представлена на *рис. 1.6*.

Связь между фактическим и формальным параметром определяется позицией, которую они занимают в этих двух списках.

В случае **параметра-значения** в качестве фактического параметра может использоваться любое выражение, в частности константа или переменная.

Соответствующее выражение должно быть совместимым с точки зрения присваивания с типом формального параметра. Изменения параметров-значений не передаются за пределы подпрограммы.

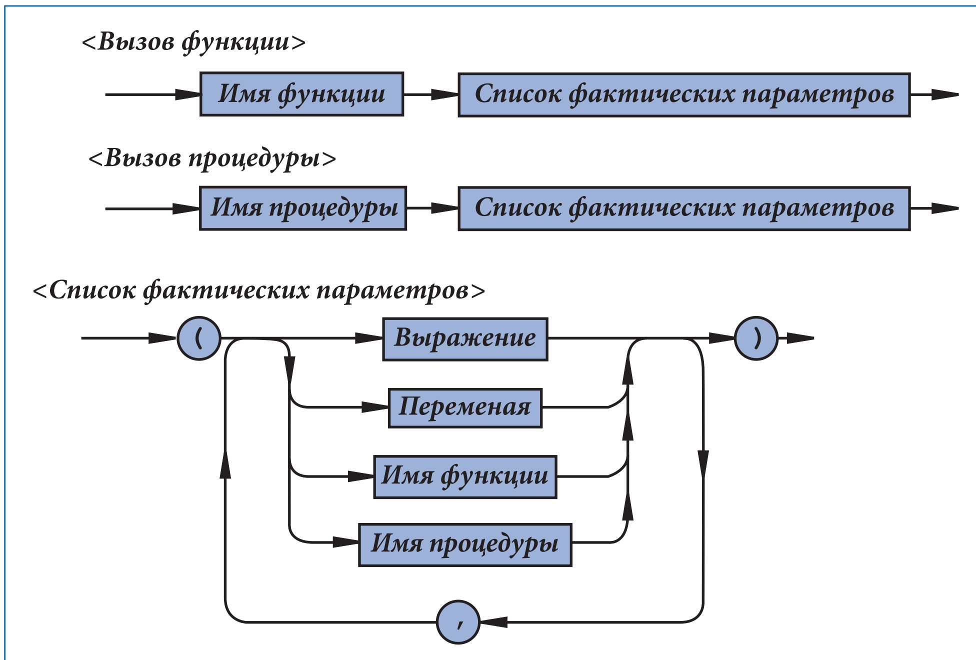


Рис. 1.6. Синтаксис вызова функций и процедур

В случае **параметра-переменной** в качестве фактических параметров могут использоваться только переменные. Изменения параметров-переменных передаются за пределы подпрограммы.

В случае **параметра-функции (процедуры)** в качестве фактического параметра может использоваться любое имя функции (процедуры), заголовок которой имеет вид, указанный в списке формальных параметров.

Вопросы и упражнения

❶ Когда используется описание вида

function *<Идентификатор>*; *<Блок>* ?

❷ Укажите на синтаксических диаграммах *рис. 1.3* и *1.5* пути, которые соответствуют описаниям функций из программы `P106`, параграф 1.4.

❸ В чем разница между параметром-значением и параметром-переменной?

❹ Укажите на синтаксических диаграммах *рис. 1.4* и 1.5 пути, которые соответствуют описаниям процедур из программы `P101`, параграф 1.3.

❺ Укажите на синтаксических диаграммах *рис. 1.3 – 1.6* пути, которые соответствуют описанию и вызову подпрограмм из программы `P105`, параграф 1.4.

Глава 2

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

2.1. Динамические переменные. Ссылочный тип

Переменные, описанные в разделе **var** любой программы или подпрограммы, называются **статическими переменными**. Число статических переменных устанавливается в момент написания программы и не может изменяться в процессе ее выполнения. Однако очень часто встречаются задачи, в которых необходимое число переменных заранее неизвестно.

Предположим, что необходимо обработать данные о людях, которые стоят в очереди в кассу за билетами. Длина очереди заранее неизвестна. Каждый раз, когда появляется новый человек, необходимо создать переменную соответствующего типа. После того, как человек уходит, соответствующая переменная становится излишней.

Переменные, которые создаются и уничтожаются в процессе выполнения программы, называются **динамическими переменными**.

Доступ к динамическим переменным осуществляется через переменные *ссылочного типа*. Как правило, ссылочный тип определяется описанием вида:

type T_r = ^ T_b;

где T_r – имя ссылочного типа, а T_b – базовый тип. Знак “^” читается как “адрес”. Очевидно, могут использоваться и анонимные ссылочные типы. Синтаксическая диаграмма <*Ссылочный тип* > представлена на *рис. 2.1*.

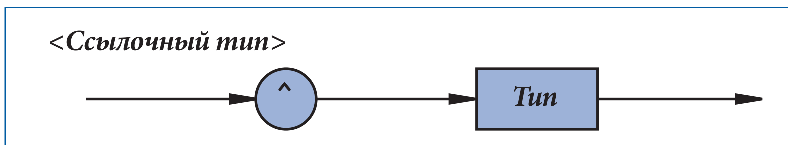


Рис. 2.1. Синтаксическая диаграмма <*Ссылочный тип*>

Множество значений ссылочного типа состоит из адресов. Каждый адрес указывает на динамическую переменную, которая принадлежит базовому типу. К указанному множеству адресов добавляется специальное значение **nil** (нуль), которое не указывает никакую переменную.

Пример:

```
type AdresaInteger=^integer;
     AdresaChar=^char;
```

```
var  i : AdresaInteger;
     r : ^real;
     c : AdresaChar;
```

Текущее значение переменной i будет указывать на динамическую переменную типа integer. Аналогично переменные ссылочного типа r и c указывают на переменные типа real и соответственно char. Отметим, что типы данных AdresaInteger, AdresaChar и анонимный тип ^real являются различными *ссылочными* типами.

К значениям ссылочного типа можно применять операции = и <>. Значения *ссылочного* типа не могут быть введены с клавиатуры и выведены на экран.

Динамическая переменная создается с помощью стандартной функции new (новая). Вызов данной процедуры имеет вид

```
new(p)
```

где p – переменная ссылочного типа.

Процедура выделяет область памяти для создаваемой переменной и возвращает адрес соответствующей области через переменную p. В дальнейшем доступ к динамической переменной можно получить **указанием адреса**: за именем переменной *ссылочного* типа p необходимо поставить знак "^". Если в качестве адреса использовать некоторую переменную *ссылочного* типа с содержанием **nil**, то это приведет к ошибке выполнения.

Пример:

new(i); i^:=1 – создание динамической переменной типа integer; созданной переменной присваивается значение 1;

new(r); r^:=2.0 – создание динамической переменной типа real; созданной переменной присваивается значение 2.0;

new(c); c^:='*' – создание динамической переменной типа char; созданной переменной присваивается значение '*'.

Динамическая переменная p^, созданная при вызове new(p), отличается от всех переменных, созданных ранее. Следовательно, выполнение операторов

```
new(p); new(p); ...; new(p)
```

приведет к созданию последовательности динамических переменных v_1, v_2, ..., v_n. При этом переменная p указывает на последнюю из созданных динамических переменных – на v_n. Так как значения переменных *ссылочного* типа являются адресами конкретных областей внутренней памяти компьютера, данные переменные называются **указателями адреса,** или просто **указателями.**

Уничтожение динамической переменной и освобождение соответствующей области памяти осуществляются с помощью стандартной процедуры dispose (освобождение). Вызов данной процедуры имеет вид:

```
dispose(p)
```

где p – переменная *ссылочного* типа.

Примеры:

```
dispose(i); dispose(r); dispose(c)
```

После выполнения процедуры `dispose(p)` значение переменной `p` *ссылочного* типа становится неопределенным.

К динамическим переменным можно применять все операции, допустимые в базовом типе.

Пример:

```
Program P117;
 {Операции над динамическими переменными}
 type AdresaInteger=^integer;
 var i, j, k : AdresaInteger;
     r, s, t : ^real;
 begin
  {создание динамических переменных типа integer }
  new(i); new(j); new(k);
  {операции над созданными переменными}
  i^:=1; j^:=2;
  k^:=i^+j^;
  writeln(k^);
  {создание динамических переменных типа real}
  new(r); new(s); new(t);
  {операции над созданными переменными}
  r^:=1.0; s^:=2.0;
  t^:=r^/s^;
  writeln(t^);
  {уничтожение динамических переменных}
  dispose(i); dispose(j); dispose(k);
  dispose(r); dispose(s); dispose(t);
  readln;
 end.
```

В отличие от статических переменных, которые занимают области памяти, выделенные компилятором, динамические переменные занимают области памяти, предлагаемые функцией `new`. Соответствующие области памяти освобождаются с помощью процедуры `dispose` и могут быть повторно использованы для размещения новых динамических переменных. Таким образом, процедуры `new` и `dispose` обеспечивают **динамическое распределение памяти**: область памяти предоставляется динамической переменной только на время ее существования.

Количество динамических переменных, способных существовать одновременно при выполнении программы, зависит от типа переменных и области памяти, имеющейся в распоряжении. Когда в памяти не остается свободного места, вызов процедуры `new` приведет к ошибке выполнения.

Пример:

```
Program P118;
 {Ошибка: переполнение памяти}
 label 1;
 var i : ^integer;
```

```
begin
 1 : new(i);
     goto 1;
end.
```

Динамическое распределение памяти требует особого внимания от программиста, который обязан создавать и уничтожать динамические переменные, а также обеспечивать правильную ссылку на них.

Вопросы и упражнения

❶ В чем разница между статическими и динамическими переменными?

❷ Как идентифицируются динамические переменные?

❸ Укажите на синтаксической диаграмме *рис. 2.1* пути, которые соответствуют описаниям ссылочных типов из программы `P117`.

❹ Даны описания:

```
type AdresaReal=^real;
var  r : AdresaReal;
```

Укажите множество значений типа данных `AdresaReal` и множество значений, которые может принимать динамическая переменная `r^`.

❺ Какие операции можно применять к *ссылочным* типам данных? А к динамическим переменным?

❻ Даны описания:

```
type AdresaTablou = ^array [1..10] of integer;
var  t : AdresaTablou;
```

Укажите множество значений типа данных `AdresaTablou` и множество значений, которые может принимать динамическая переменная `t^`.

❼ Прокомментируйте программу:

```
Program P119;
 {Ошибка}
 var r, s : ^real;
 begin
  r^:=1; s^:=2;
  writeln('r^=', r^, '  s^=', s^);
  readln;
 end.
```

❽ Напишите программу, в которой создаются две динамические переменные типа *строка символов*. Созданным переменным присвойте любые значения и выведите на экран результат конкатенации соответствующих строк.

❾ Что выводит на экран следующая программа?

```
Program P120;
 var i : ^integer;
```

```
begin
 new(i); i^:=1;
 new(i); i^:=2;
 new(i); i^:=3;
 writeln(i^);
 readln;
end.
```

⑩ Прокомментируйте программу:

```
Program P121;
 {Ошибка}
var i, j : ^integer;
begin
 new(i);
 i^:=1;
 dispose(i);
 new(j);
 j^:=2;
 dispose(j);
 writeln('i^=', i^, '  j^=', j^);
 readln;
end.
```

⑪ Объясните выражение *динамическое распределение памяти*.

2.2. Структуры данных

Структура данных состоит из самих данных и связей между ними. В зависимости от метода организации структура данных может быть явной или неявной.

Массивы, строки символов, записи, файлы и множества, изученные в предыдущих главах, относятся к **неявным структурам данных**. Связи между компонентами указанных структур являются предопределенными и неизменяемыми. Например, все компоненты строки символов имеют общее имя, а символ `s[i+1]`, с точки зрения занимаемой позиции, всегда следует за символом `s[i]`. Так как структура массивов, строк символов, записей, файлов и множеств не изменяется в процессе выполнения любой программы или подпрограммы, соответствующие структуры являются **статическими**.

Используя данные со статической структурой, можно решать лишь ограниченный класс задач. Очень часто связи между компонентами не только динамически изменяются, но и могут оказаться очень сложными.

Например, в случае очереди за билетами в кассу связи между людьми изменяются: вновь прибывшие становятся в очередь; те, у которых мало времени, уходят, так и не купив билеты; отошедшие на некоторое время возвращаются на свои места и т.д. При проектировании, с помощью компьютера, расписаний дорожного движения, остановки, маршруты, параметры транспортных средств

и т. д. могут интерактивно устанавливаться пользователем. В таких случаях использование данных со статической структурой становится неестественным, сложным и неэффективным.

Таким образом, в ряде задач необходимо использовать структуры данных, связи между компонентами которых представляются и обрабатываются явно. Этого можно достичь, если к каждой компоненте добавить информацию, характеризующую ее связи с другими компонентами структуры. В большинстве случаев дополнительная информация, называемая **структурной информацией,** представляется с помощью ссылочных переменных.

Структуры данных, компоненты которых создаются и уничтожаются во время выполнения программы, называются **динамическими структурами**. Часто используемыми динамическими структурами данных являются односвязные и двусвязные списки, очереди, стеки, деревья и др.

Структура называется **рекурсивной**, если ее можно разложить на данные точно такой же структуры. В качестве примера можно привести односвязные и двусвязные списки.

Вопросы и упражнения

1. Объясните термин *структура данных*. Приведите примеры.
2. В чем разница между явными и неявными структурами данных?
3. Структура данных является **однородной**, если все ее компоненты относятся к одному и тому же типу. В противном случае структура является **неоднородной**. Приведите примеры однородных и неоднородных структур данных.
4. В чем разница между статическими и динамическими структурами данных?
5. Объясните термин *рекурсивная структура данных*.

2.3. Односвязные списки

Односвязные списки – это явные динамические структуры данных, составленные из ячеек. Каждая **ячейка** представляет собой динамическую переменную типа **record**, которая в основном состоит из двух полей: поле данных и поле связей. В **поле данных** запоминается обрабатываемая информация, связанная с ячейкой. **Поле связей** содержит указатель адреса ячейки, в которую можно попасть из текущей ячейки. Предполагается, что в любую ячейку можно попасть, начиная с первой ячейки, называемой **базой списка**.

В качестве примера на *рис. 2.2* представлен односвязный список, составленный из четырех ячеек. Ячейки содержат элементы *A*, *B*, *C* и *D*.

Данные, необходимые для создания и обработки односвязного списка, определяются с помощью следующих описаний:

```
type AdresaCelula=^Celula;
  Celula=record
        Info : string;
```

```
        Urm   : AdresaCelula;
      end;
  var P : AdresaCelula;
```

Полезная информация, относящаяся к конкретной ячейке, запоминается в поле Info, а адрес следующей ячейки – в поле Urm. Для простоты считается, что тип Info является *строковым*. В поле Urm последней ячейки списка содержится значение **nil**. Адрес первой ячейки (базовый адрес) запоминается в переменную ссылочного типа P (*рис.* 2.2).

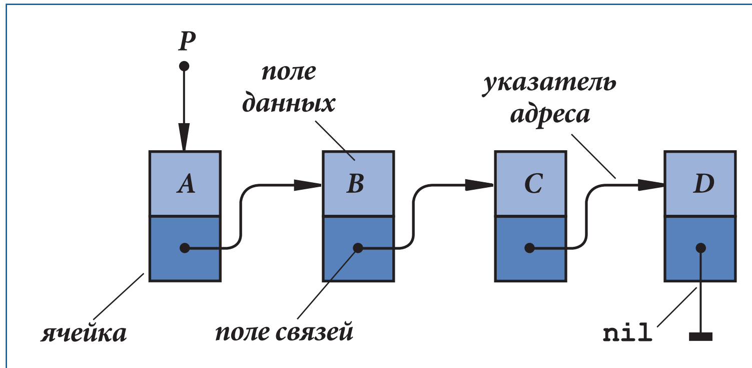


Рис. 2.2. Односвязный список

Отметим, что при описании ссылочного типа AdresaCelula базовый тип Celula еще не известен. Согласно правилам языка ПАСКАЛЬ, это возможно только в случае динамических переменных при условии, что базовый тип определяется ниже в этом же описании.

Односвязный список может быть создан путем добавления в вершину списка по одному элементу. Естественно, перед началом работы создаваемый список является пустым, то есть не содержит ни одной ячейки.

Пример:

```
Program P122;
 {Создание односвязного списка A->B->C->D}
 type AdresaCelula=^Celula;
      Celula=record
                   Info : string;
                    Urm : AdresaCelula;
             end;
 var P,         {базовый адрес}
     R, V : AdresaCelula;
 begin
  {1 - сперва список является пустым}
  P:=nil;
  {2 - добавление ячейки A}
  new(R);        {создание ячейки}
  P:=R;          {инициализация базового адреса}
```

```
  R^.Info:='A'; {ввод полезной информации}
  R^.Urm:=nil;  {запись индикатора «конец списка»}
  V:=R;         {запоминание адреса вершины списка}
  {3 - добавление ячейки B}
  new(R);       {создание ячейки}
  R^.Info:='B'; {ввод полезной информации}
  R^.Urm:=nil;  {запись индикатора «конец списка»}
  V^.Urm:=R;    {создание связи A -> B}
  V:=R;         {обновление адреса вершины списка}
  {4 - добавление ячейки C}
  new(R);       {создание ячейки}
  R^.Info:='C'; {ввод полезной информации}
  R^.Urm:=nil;  {запись индикатора «конец списка»}
  V^.Urm:=R;    {создание связи B -> C}
  V:=R;         {обновление адреса вершины списка}
  {5 - добавление ячейки D}
  new(R);       {создание ячейки}
  R^.Info:='D'; {ввод полезной информации}
  R^.Urm:=nil;  {запись индикатора «конец списка»}
  V^.Urm:=R;    {создание связи C -> D}
  V:=R;         {обновление адреса вершины списка}
  {вывод созданного списка на экран}
  R:=P;
  while R<>nil do begin
                    writeln(R^.Info);
                    R:=R^.Urm
                  end;
  readln;
 end.
```

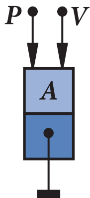
Процесс составления данного списка представлен на *рис. 2.3*. Для того чтобы установить указатель адреса `V^.Urm`, адрес только что созданной ячейки запоминается в переменной `V` (адрес вершины). Это необходимо потому, что на момент заполнения полей `R^.Info` и `R^.Urm` текущей ячейки адрес последующей ячейки еще не известен.

С помощью соответствующих процедур из программы `P123` можно создавать и выводить на экран списки с произвольным количеством ячеек:

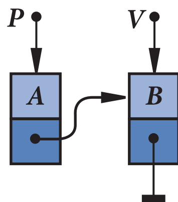
```
Program P123;
{Создание линейных списков}
type AdresaCelula=^Celula;
     Celula=record
              Info : string;
               Urm : AdresaCelula;
            end;
```

1 – сперва список является пустым

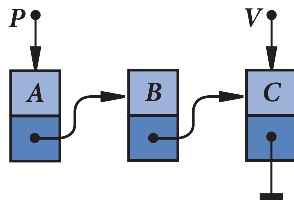
2 – добавление ячейки A



3 – добавление ячейки B



4 – добавление ячейки C



5 – добавление ячейки D

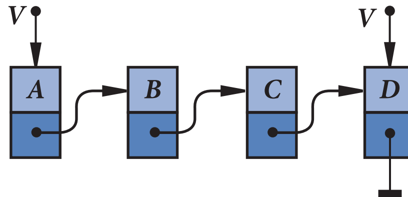


Рис. 2.3. Создание односвязного списка

```
var p,q,r : AdresaCelula;
    s : string;
    i: integer;
procedure Creare;
begin
 p:=nil;
write('s='); readln(s);
 new(r); r^.Info:=s; r^.Urm:=nil;
 p:=r; q:=r;
 write('s=');
  while not eof do
   begin
     readln(s);write('s=');
     new(r); r^.Info:=s; r^.Urm:=nil;
     q^.Urm:=r; q:=r
   end;
end; { Creare }
procedure Afisare;
begin
 r:=p;
 while r<>nil do
   begin
     writeln(r^.Info);
     r:=r^.Urm;
   end;
readln;
end; {Вывод на экран}

begin
 Creare;
 Afisare;
end.
```

Любой односвязный список можно определить **рекурсивно** следующим образом:

а) отдельная ячейка является односвязным списком;

б) ячейка, которая содержит указатель на другой односвязный список, также является односвязным списком.

Для того чтобы подчеркнуть тот факт, что односвязные списки представляют собой рекурсивные типы данных, соответствующие описания можно записать в виде:

```
 type  Lista=^Celula;
   Celula=record
            Info : string;
            Urm  : Lista
          end;
 var P : Lista;
```

Частично свойства односвязных списков можно воспроизвести путем запоминания соответствующих элементов в одномерный массив. Например, данные из *рис. 2.2* можно представить в виде:

```
var L : array [1..4] of string;
    ...
   L[1]:= 'A';
   L[2]:= 'B';
   L[3]:= 'C';
   L[4]:= 'D';
   ...
```

Однако такое представление является неприемлемым в случае списков с заранее неизвестным числом элементов.

Вопросы и упражнения

❶ Как описываются данные, необходимые для создания списка?

❷ Для чего нужно поле данных, входящее в состав ячейки? Для чего нужно поле связей? Какая информация записывается в данное поле?

❸ Напишите программу, которая создает односвязный список *рис. 2.2*, добавляя к базе списка по одному элементу.

❹ Напишите процедуру, которая меняет местами два элемента списка.

❺ С клавиатуры считываются числа, отличные от нуля. Создайте два списка, один из которых состоит из отрицательных чисел, а другой – из положительных.

❻ Чем можно объяснить тот факт, что односвязные списки являются рекурсивными структурами данных?

2.4. Обработка односвязных списков

Наиболее часто к односвязным спискам применяются следующие **операции**:
– проход по списку и обработка информации из каждой ячейки;
– поиск определенного элемента, заданного своим значением;
– вставка заданного элемента в указанное место списка;
– удаление некоторого элемента из списка и т.д.

Предположим, что существует непустой список (*рис. 2.2*), определенный описанием:

```
type   AdresaCelula=^Celula;
       Celula=record
              Info : string;
              Urm  : AdresaCelula
             end;
```

Переменная `P` является базовым адресом списка.

Проход по списку осуществляется с помощью следующей последовательности операторов:

```
R:=P; {установление указателя на базовую ячейку}
while R<>nil do
begin
  {обработка информации из поля R^.Info}
  R:=R^.Urm; {установление указателя на следующую ячейку}
end;
```

Поиск ячейки, которая содержит элемент, заданный переменной `Cheie`, осуществляется с помощью следующей последовательности операторов:

```
R:=P;
while R^.Info<>Cheie do R:=R^.Urm;
```

Адрес найденной ячейки заносится в переменную `R`.

Отметим, что приведенная последовательность операторов выполняется правильно только в случае, когда список содержит, по крайней мере, одну ячейку с искомой информацией. В противном случае, по достижении вершины списка, переменная `R` принимает значение **`nil`**, и операция адресации `R^` приведет к ошибке выполнения. Для исключения таких ошибок используется следующая последовательность операторов:

```
  R:=P;
  while R<>nil do
  begin
    if R^.Info=Cheie then goto 1;
    R:=R^.Urm
  end;
1: ...
```

Так как односвязные списки являются рекурсивными структурами данных, операцию поиска можно реализовать и с помощью рекурсивной подпрограммы:

```
type Lista=^AdresaCelula;
     Celula=record
              Info : string;
               Urm : Lista;
            end;
var  P : Lista;
...
function Caut(P : Lista; Cheie : string):Lista;
begin
  if P=nil then Caut:=nil
           else
              if P^.Info=Cheie then Caut:=P
                               else Caut:=Caut(P^.Urm, Cheie)
end;
```

Функция `Caut` возвращает базовый адрес списка, который содержит в первой ячейке, если существует, значение, указываемое параметром `Cheie`.

Вставка ячейки, заданной указателем `Q`, за ячейкой, заданной указателем `R` (*рис. 2.4*), выполняется с помощью следующей последовательности операторов:

```
Q^.Urm:=R^.Urm;
R^.Urm:=Q;
```

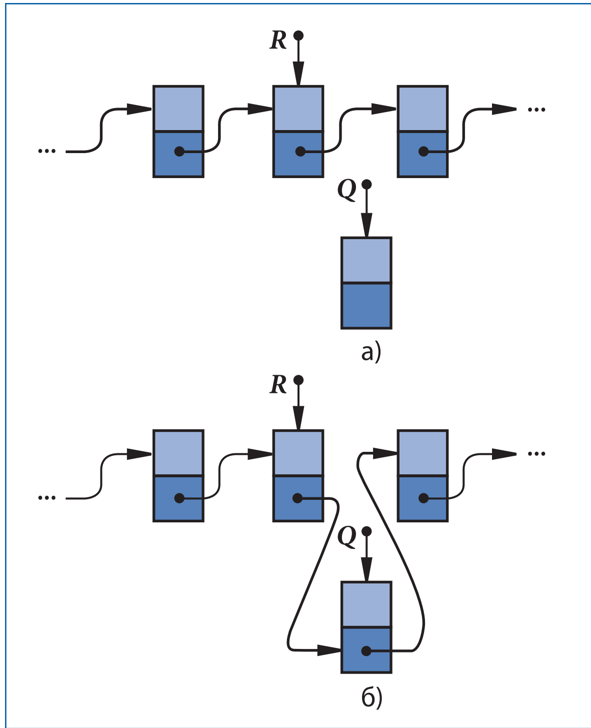


Рис. 2.4. Вставка элемента в список:
а – список до операции вставки; *б* – список после операции вставки

Для **удаления** элемента из списка необходимо найти адрес `Q` предыдущей ячейки и изменить указатель адреса `Q^.Urm` (*рис. 2.5*):

```
Q:=P;
while Q^.Urm<>R do Q:=Q^.Urm;
Q^.Urm:=R^.Urm;
```

Отметим, что для вставки или удаления базового элемента списка необходимо обновить базовый адрес `P`.

Пример:

```
Program P124;
 {Создание и обработка односвязного списка}
 type AdresaCelula=^Celula;
      Celula=record
```

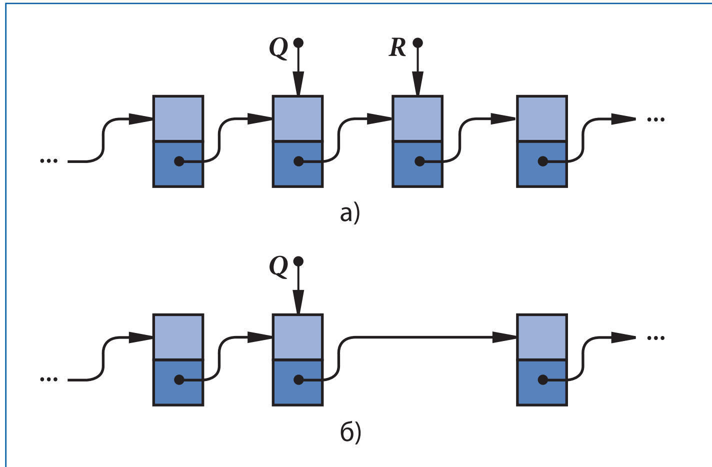


Рис. 2.5. Удаление элемента из списка:
а – список до операции удаления; *б* – список после операции удаления

```
              Info : string;
               Urm : AdresaCelula;
              end;
  var P : AdresaCelula;  {базовый адрес}
      c : char;
procedure Creare;
var R, V : AdresaCelula;
begin
  if P<>nil then writeln('Список уже существует')
    else begin
           writeln('Введите список:');
           while not eof do
             begin
               new(R);
               readln(R^.Info);
               R^.Urm:=nil;
               if P=nil then begin P:=R; V:=R end
                     else begin V^.Urm:=R; V:=R end;
          end;
         end;
end; {Creare }

procedure Afis;
var R : AdresaCelula;
begin
  if P=nil then writeln('Список пустой')
           else begin
                  writeln('Текущий список:');
                  R:=P;
                  while R<>nil do
```

```
            begin
              writeln(R^.Info);
              R:=R^.Urm;
            end;
          end;
   readln;
  end; {Afis }

procedure Includ;
label 1;
var Q, R : AdresaCelula;
    Cheie : string;
begin
  new(Q);
  write('Введите элемент, который');
  writeln(' будет вставлен:');
  readln(Q^.Info);
  write('Укажите элемент, после которого');
  writeln(' будет осуществляться вставка:');
  readln(Cheie);
  R:=P;
  while R<>nil do
     begin
       if R^.Info=Cheie then goto 1;
       R:=R^.Urm;
     end;
 1:if R=nil then begin
                  writeln('Несуществующий элемент');
                  dispose(Q);
                end;
            else begin
                  Q^.Urm:=R^.Urm;
                  R^.Urm:=Q;
                end;
end; { Includ }

procedure Exclud;
label 1;
var Q, R : AdresaCelula;
    Cheie : string;
begin
  write('Укажите элемент, который');
  writeln(' будет удален:');
  readln(Cheie);
  R:=P;
  Q:=R;
```

```
  while R<>nil do
     begin
       if R^.Info=Cheie then goto 1;
       Q:=R;
       R:=R^.Urm;
     end;
 1:if R=nil then writeln('Несуществующий элемент')
            else begin
                   if R=P then P:=R^.Urm else Q^.Urm:=R^.Urm;
                   dispose(R);
                 end;
end; { Exclud }
begin
  P:=nil; { сперва список пустой }
  repeat
    writeln('Меню:');
    writeln('C - Создание списка');
    writeln('I - Вставка элемента');
    writeln('E - Удаление элемента');
    writeln('A - Вывод списка на экран');
    writeln('O - Остановка программы');
    write('Выбор='); readln(c);
    case c of
        'C' : Creare;
        'I' : Includ;
        'E' : Exclud;
        'A' : Afis;
        'O' :
        else writeln('Недопустимая операция')
    end;
  until c='O';
 end.
```

Процедура `Creare` создает односвязный список с произвольным количеством ячеек. Полезная информация, связанная с каждой ячейкой, считывается с клавиатуры. Процедура `Afis` выводит элементы списка на экран. Процедура `Includ` считывает с клавиатуры элемент, который должен быть вставлен, и элемент, после которого должна выполняться вставка. Затем реализуется поиск ячейки, содержащей указанный элемент. Если такой элемент существует, вставляется новая ячейка. Процедура `Exclud` осуществляет поиск ячейки, содержащей элемент, считанный с клавиатуры, и удаляет ее, если таковая существует.

Вопросы и упражнения

❶ Напишите нерекурсивную функцию, которая возвращает адрес вершины односвязного списка.

❷ Перепишите процедуры `Includ` и `Exclud` из программы `P124` без использования оператора **goto**.

❸ Даны следующие типы данных:

```
type   AdresaCandidat=^Candidat;
       Candidat=record
                  NumePrenume : string;
                  NotaMedie : real;
                  Urm : AdresaCandidat
                end;
```

Напишите программу, которая:

а) создает односвязный список с компонентами типа `Candidat`;

б) выводит список на экран;

в) удаляет из списка кандидата, который забирает документы;

г) включает в список кандидата, который подает документы;

д) выводит на экран список кандидатов, средний балл которых больше 7,5;

е) создает дополнительный список, составленный из кандидатов, средняя оценка которых больше 9,0;

ж) удаляет из списка всех кандидатов, средний балл которых меньше 6,0.

❹ Напишите процедуру, которая:

а) упорядочивает элементы односвязного списка согласно заданному критерию;

б) склеивает два односвязных списка;

в) разделяет заданный список на два отдельных списка;

г) выбирает из списка элементы, которые удовлетворяют заданному критерию.

❺ Элементы односвязного списка запоминаются в одномерном массиве. Напишите процедуры, необходимые для:

а) прохода по списку;

б) поиска определенного элемента;

в) вставки заданного элемента;

г) удаления заданного элемента.

Каковы преимущества и недостатки такого представления? Подразумевается, что список содержит не более 100 элементов.

❻ Напишите рекурсивную функцию, которая возвращает количество ячеек в односвязном списке.

❼ Напишите рекурсивную подпрограмму, которая исключает из списка указанную ячейку.

❽ Под **словом** понимается любая непустая последовательность из букв латинского алфавита. Напишите программу, которая формирует список слов, встречающихся в текстовом файле, и считает, сколько раз появляется каждое слово. Рассмотрите следующие случаи:

а) слова вставляются в список в порядке их первого появления в тексте;

б) слова вставляются в список в алфавитном порядке.

Прописные и строчные буквы считаются идентичными.

2.5. Стеки

Под **стеком** (по-английски *stack*) понимается односвязный список, вставку и удаление элементов в который можно производить только с одного конца списка. Ячейка, которую занимает последний введенный элемент, называется **вершиной стека**. Стек, не содержащий ни одного элемента, называется **пустым**.

В качестве примера на *рис. 2.6* представлен стек, который содержит элементы *A*, *B*, *C*.

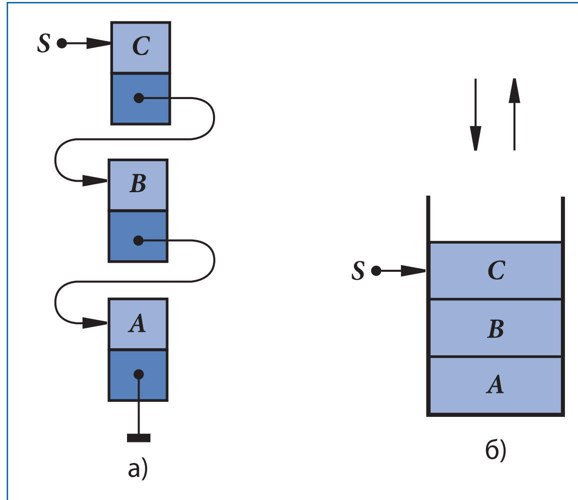


Рис. 2.6. Удаление элемента из списка:
а – детальное представление; *б* – обобщенное представление

Данные, необходимые для создания и обработки стека, можно определять с помощью описаний вида:

```
type AdresaCelula=^Celula;
      Celula=record
               Info : string;
               Prec : AdresaCelula
             end;
var  S : AdresaCelula;
```

Адрес вершины стека запоминается в переменной *ссылочного* типа `S`. Адрес предыдущей ячейки стека запоминается в поле `Prec`.

Операция вставки элемента в стек (*рис. 2.7*) выполняется с помощью следующей последовательности операторов:

```
new(R); {создание ячейки}
{введение полезной информации в поле R^.Info}
R^.Prec:=S; {создание связи с предыдущей ячейкой стека}
S:=R; {обновление адреса вершины}
```

где `R` является переменной типа `AdresaCelula`.

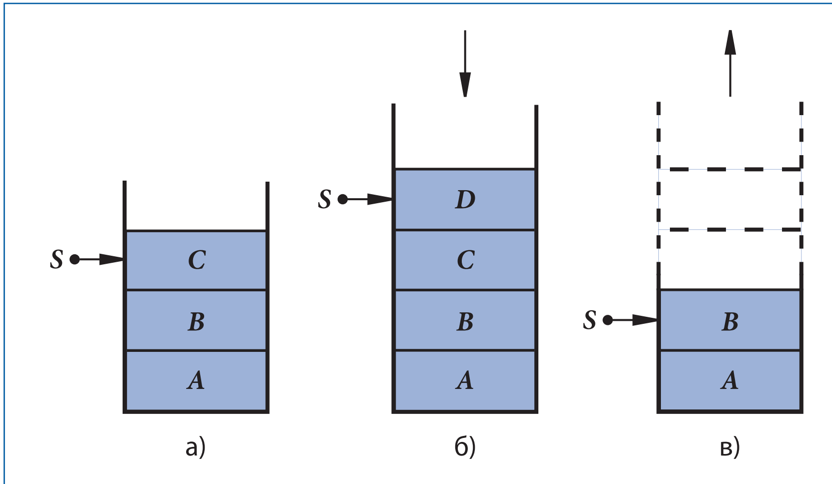


Рис. 2.7. Вставка и удаление элемента из стека:
а – исходный стек; *б* – вставка элемента *D*; *в* – извлечение элементов *D*, *C*

Извлечение элемента из стека (*рис. 2.7*) выполняется с помощью следующей последовательности операторов:

```
R:=S; {запоминание адреса удаляемого элемента}
 {обработка информации из поля R^.Info}
S:=S^.Prec; {удаление ячейки из стека}
dispose(R); {уничтожение удаленной ячейки}
```

Пример:

```
Program P127;
 {Создание и обработка стека}
 type AdresaCelula=^Celula;
      Celula=record
                   Info : string;
                   Prec : AdresaCelula;
            end;
 var S : AdresaCelula; {адрес вершины}
     c : char;
 procedure Introduc;
 var R : AdresaCelula;
 begin
  new(R);
  write('Введите элемент, который необходимо');
  writeln('вставить:');
  readln(R^.Info);
  R^.Prec:=S;
  S:=R;
 end; { Includ }
```

```
procedure Extrag;
var R : AdresaCelula;
begin
 if S=nil then writeln('Стек пустой')
          else begin
                R:=S;
                write('Извлечен');
                writeln('элемент:');
                writeln(R^.Info);
                S:=S^.Prec;
                dispose(R);
               end;
end; { Extrag }

procedure Afis;
var R : AdresaCelula;
begin
 if S=nil then writeln('Стек пустой')
   else begin
          writeln('Стек содержит элементы:');
          R:=S;
          while R<>nil do begin
                            writeln(R^.Info);
                            R:=R^.Prec;
                          end;
        end;
 readln;
end; { Afis }
begin
 S:=nil;  {Сперва стек пустой}
 repeat
   writeln('Меню:');
   writeln('I - Вставка элемента;');
   writeln('E - Извлечение элемента');
   writeln('A - Вывод стека на экран');
   writeln('O - Остановка программы');
   write('Ваш выбор='); readln(c);
   case c of
       'I' : Introduc;
       'E' : Extrag;
       'A' : Afis;
       'O' :
       else writeln('Недопустимая операция')
   end;
 until c='O';
end.
```

Стеки называют еще ***списками LIFO*** (*last in, first out* – последний элемент, введенный в стек, будет извлечен первым) и часто используют для динамического выделения памяти в случае рекурсивных процедур и функций. Очевидно, стеки можно имитировать, используя одномерные массивы: **array[1..n] of** ..., однако такое представление ограничено только *n* ячейками.

Вопросы и упражнения

❶ В каком порядке вставляются и удаляются элементы стека?

❷ С клавиатуры вводится последовательность вещественных чисел. Выведите данные числа на экран в порядке, обратном чтению.

❸ На *рис. 2.8* представлена схема движения железнодорожных вагонов в депо. Напишите программу, которая вводит с клавиатуры и выводит на экран данные о каждом вагоне, который вошел или вышел из депо. Эти данные должны включать:
– регистрационный номер (integer);
– станцию регистрации (**string**);
– год выпуска (**1960 ... 2000**);
– тип вагона (**string**);
– грузоподъемность (real);
– данные о владельце вагона (**string**).

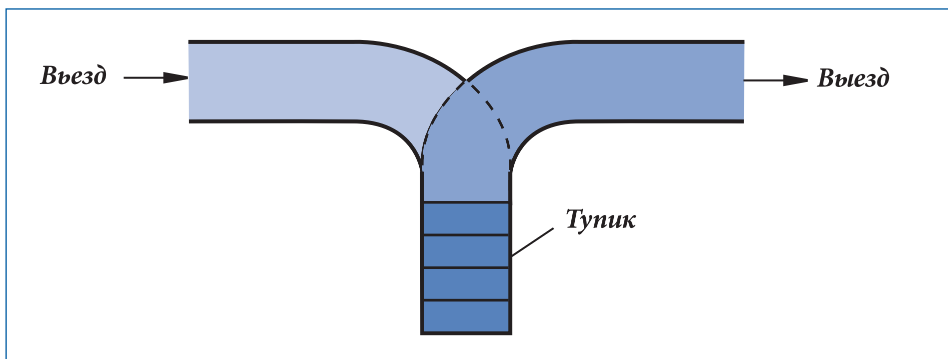


Рис. 2.8. Схема движения железнодорожных вагонов в депо

❹ Временные рабочие группы создаются и упраздняются в следующем порядке: “Последний принятый на работу будет уволен первым”. Напишите программу, которая вводит с клавиатуры и выводит на экран данные о каждом принятом или уволенном специалисте. Эти данные должны включать:
– фамилию (**string**);
– имя (**string**);
– год рождения (1930–1985);
– дату приема на работу (день, месяц, год).

❺ Даны конечные строки символов, составленные из скобок: (,), [,], {, }. Строка является правильной, если она составлена по следующим правилам:
а) пустая строка является правильной;
б) если *A* является правильной строкой, то (*A*), [*A*], {*A*} – правильные строки;
в) если *A* и *B* – правильные строки, то *AB* – правильная строка.

Например, строки: ( ), [ ], { }, [( )], ((({[ ]}))([ ])) – правильные, а строки: ([, ( )[ ]{{, ([)] – неправильные. Напишите программу, которая проверяет, является ли строка символов, считанная с клавиатуры, правильной.

Указание. Задачу можно решить, осуществляя проверку с помощью единственного просмотра строки. Если текущий символ (, [, или {, то он заносится в стек. Если вершина списка и текущий символ образуют пару ( ), [ ] или { }, то соответствующая скобка удаляется из стека. Если после проверки последнего символа строки стек остается пустым, то строка символов – правильная.

❻ Элементы стека запоминаются в одномерном массиве. Напишите процедуры, необходимые для вставки и удаления элементов из стека. Каковы недостатки и преимущества такого представления? Считается, что стек содержит не более 100 элементов.

2.6. Очереди

Под **очередью** (по-английски *queue*) понимается односвязный список, в котором вставка элементов производится в один конец, а удаление элементов – из другого конца. Очередь, не содержащая ни одного элемента, называется **пустой**. В качестве примера на *рис. 2.9* представлена очередь, которая содержит элементы *A, B, C*.

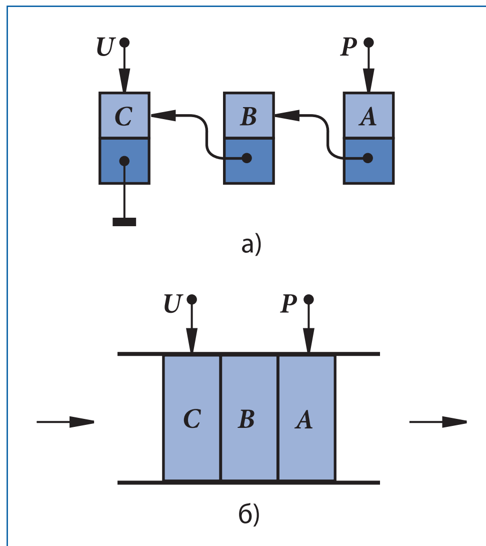


Рис. 2.9. Очередь: *а* – детальное представление;
б – обобщенное представление

Данные, необходимые для создания и обработки очереди, можно определять описаниями вида:

```
type  AdresaCelula=^Celula;
      Celula=record
                Info : string;
```

```
                    Urm : AdresaCelula
          end;
  var P, U : AdresaCelula;
```

Адрес первого элемента очереди запоминается в переменную ссылочного типа `P`, а адрес последнего элемента – в переменную `U`. Адрес последующего элемента очереди запоминается в поле `Urm`.

Операция включения элемента в очередь (*рис. 2.10*) реализуется с помощью следующей последовательности операторов:

```
new(R); {создание ячейки }

{введение полезной информации в поле R^.Info }

R^.Urm:=nil; { установка признака "последний элемент" }
U^.Urm:=R;   { включение ячейки в очередь }
U:=R;        { обновление адреса последней ячейки }
```

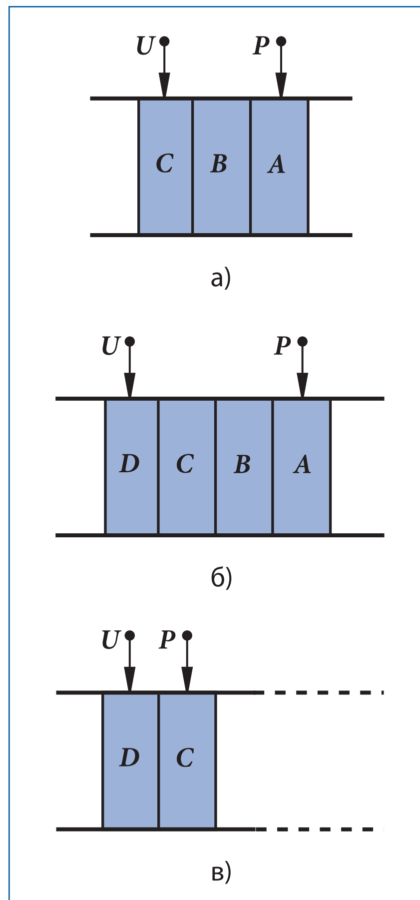


Рис. 2.10. Включение и удаление элементов из очереди: *а* – исходная очередь; *б* – включение элемента *D*; *в* – удаление элементов *A*, *B*

Удаление элемента из очереди (*рис. 2.10*) выполняется с помощью следующей последовательности операторов:

```
R:=P; { запоминание адреса первой ячейки }
{обработка информации из поля R^.Info }
P:=P^.Urm; {удаление первой ячейки }
dispose(R); {уничтожение удаленной ячейки }
```

Пример:

```
Program P128;
 {Создание и обработка очереди}
 type AdresaCelula=^Celula;
      Celula=record
               Info : string;
               Urm : AdresaCelula;
             end;
 var P,                  {адрес первого элемента}
     U : AdresaCelula; {адрес последнего элемента}
     c : char;

 procedure Introduc;
 var R : AdresaCelula;
 begin
  new(R);
  write('Введите элемент');
  writeln('который нужно включить в очередь:');
  readln(R^.Info);
  R^.Urm:=nil;
  if P=nil then begin P:=R; U:=R end
           else begin U^.Urm:=R; U:=R end;
 end; { Introduc }

 procedure Extrag;
 var R : AdresaCelula;
 begin
  if P=nil then writeln('Очередь пустая')
           else begin
                  R:=P;
                  write('Удален');
                  writeln('элемент:');
                  writeln(R^.Info);
                  P:=P^.Urm;
                  dispose(R);
                end;
 end; { Extrag }
```

```
procedure Afis;
var R : AdresaCelula;
begin
 if P=nil then writeln('Очередь пустая')
         else begin
               write('Очередь содержит');
               writeln('элементы:');
               R:=P;
               while R<>nil do
                begin
                  writeln(R^.Info);
                  R:=R^.Urm;
                end;
              end;
 readln;
end; { Afis }

begin
 P:=nil; U:=nil; {Сперва очередь пустая}
 repeat
   writeln('Меню:');
   writeln('I - Включение элемента;');
   writeln('E - Удаление элемента;');
   writeln('A - Вывод очереди на экран;');
   writeln('O - Остановка программы');
   write('Ваш выбор='); readln(c);
   case c of
       'I' : Introduc;
       'E' : Extrag;
       'A' : Afis;
       'O' :
      else writeln('Недопустимая операция')
   end;
 until c='O';
end.
```

Очереди называют еще **списками *FIFO*** (*first in, first out* – первый элемент, который был введен в очередь, будет первым, который будет извлечен из нее). Отметим, что имитация очередей с помощью одномерных массивов является неэффективной по причине сдвига элементов очереди в сторону последнего элемента массива.

Вопросы и упражнения

❶ Напишите функцию, которая возвращает количество элементов очереди.

❷ Самолеты, запрашивающие разрешение на посадку, образуют очередь. Напишите программу, которая вводит с клавиатуры и выводит на экран данные о каждом са-

молете, запросившем разрешение на посадку и о приземляющемся самолете. Эти данные должны включать:

– регистрационный номер (`integer`);
– тип самолета (**`string`**);
– номер рейса (`integer`).

❸ **Очередью с приоритетами** является очередь, в которой элемент вставляется не после последнего, а перед всеми элементами с меньшими приоритетами. Приоритеты элементов задаются целыми числами. Напишите программу, которая:

а) создает очередь с приоритетами;

б) вставляет в очередь элементы, заданные пользователем;

в) удаляет элементы из очереди;

г) выводит очередь с приоритетами на экран.

2.7. Двоичные деревья

Под **узлом** понимается динамическая переменная типа **`record`**, которая содержит поле, предназначенное для запоминания полезной информации и поля для двух указателей адреса.

Двоичное дерево определяется рекурсивно следующим образом:

а) отдельный узел является двоичным деревом;

б) узел, содержащий связи с другими двумя двоичными деревьями, тоже является двоичным деревом.

Считается, что **пустое дерево** не содержит ни одного узла. В качестве примера на *рис. 2.11* представлено двоичное дерево, в узлах которого содержится полезная информация *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*. Данные, необходимые для создания и обработки двоичного дерева, можно определять с помощью описаний вида:

```
type AdresaNod=^Nod;
      Nod=record
             Info : string;
             Stg, Dr : AdresaNod
          end;
var  T : AdresaNod;
```

Для того чтобы подчеркнуть тот факт, что двоичные деревья являются рекурсивными структурами данных, эти описания можно переписать в виде:

```
type Arbore=^Nod;
      Nod=record
             Info : string;
             Stg, Dr : Arbore;
        end;
var  T : Arbore;
```

Узел, к которому нет ни одной ссылки от других узлов, называется **корнем**. Адрес корня запоминается в переменной ссылочного типа `T`. В случае пустого дерева `T = nil`.

Деревья, связанные с корнем, называются **левым** и **правым поддеревом**. Адрес левого поддерева запоминается в поле `Stg`, адрес правого – в поле `Dr`.

Считается, что корневой узел находится на **нулевом уровне**, а уровень узла, связанного с узлом *i*-го уровня, равен (*i*+1). Обычно в графическом представлении двоичного дерева каждый узел располагается на своем уровне: корень – на уровне 0, узлы, связанные с корнем, – на уровне 1 и т. д. (*рис. 2.11*). Узлы (*i*+1)-го уровня, связанные с некоторым узлом i-го уровня, называются его **потомками**. На *рис. 2.11* узел *B* является левым потомком, а узел *C* – правым потомком узла *A*; узел *D* является левым потомком, а узел *E* – правым потомком узла *B* и т. д.

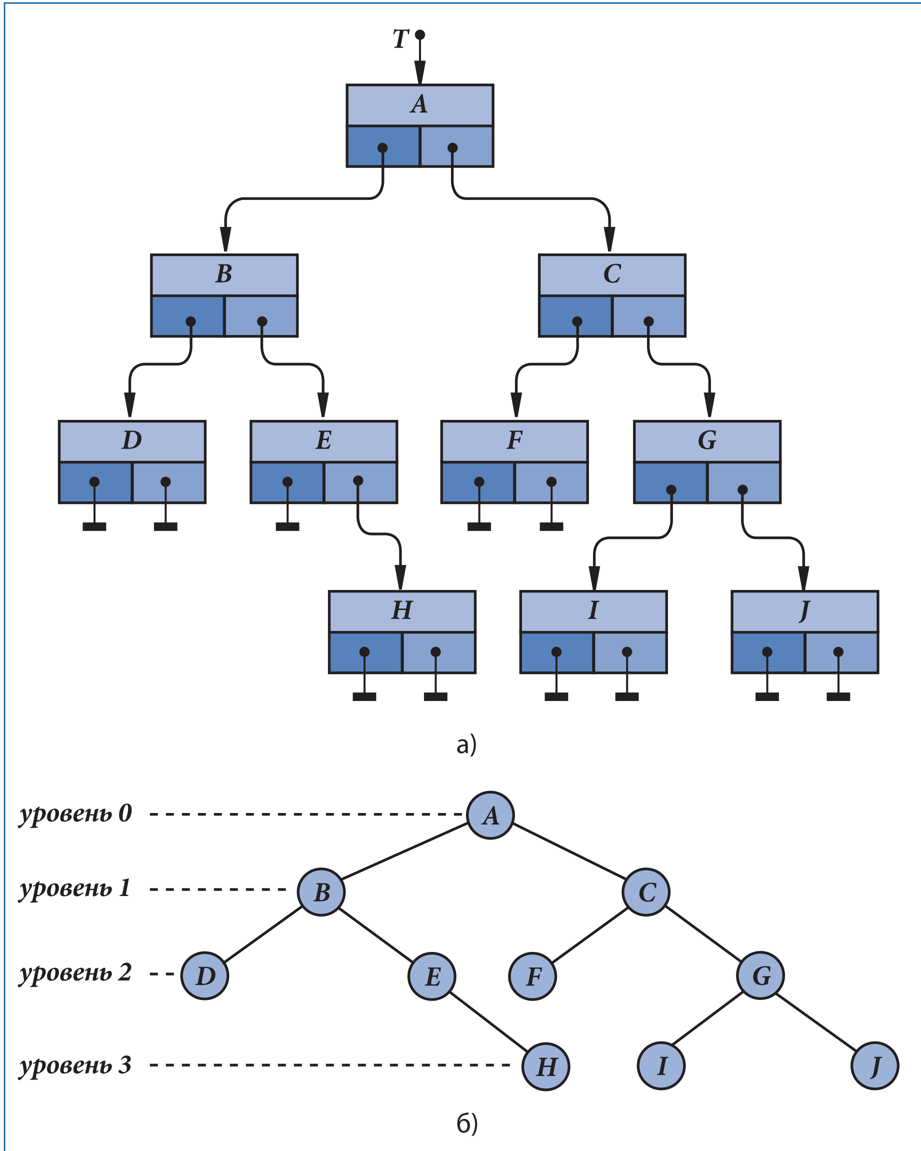


Рис. 2.11. Двоичное дерево: *а* – детальное представление; *б* – обобщенное представление

Если узел *x* является потомком узла *y*, то узел *y* называется **предком** или **родителем** узла *x*. На *рис. 2.11* узел *A* является родителем узлов *B* и *C*; узел *B* является родителем узлов *D* и *E* и т.д.

Узел, к которому не подсоединяется ни одно поддерево, называется **терминальным узлом**. В противном случае узел является **нетерминальным**. Под **высотой** двоичного дерева понимается наибольший из уровней, на котором содержатся терминальные узлы. Высота дерева на *рис. 2.11* равняется 3; узлы *D*, *H*, *F*, *I* и *J* являются терминальными узлами, а узлы *A*, *B*, *C*, *E* и *G* – нетерминальными.

Двоичные деревья могут создаваться в памяти компьютера с помощью итеративных или рекурсивных алгоритмов.

Итеративный алгоритм создает узлы в порядке их появления на уровнях:

– создается корневой узел;

– корневой узел заносится в очередь;

– для каждого узла, удаленного из очереди, создается левый и правый потомок, если таковые существуют;

– вновь созданные узлы заносятся в очередь;

– процесс создания дерева завершается, когда очередь становится пустой.

Узлы дерева на *рис. 2.11* будут созданы с помощью итеративного алгоритма в следующем порядке: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*.

Подобный алгоритм можно также использовать для обхода двоичного дерева и вывода соответствующих узлов на экран:

– создается очередь, состоящая из одного элемента – корневого узла;

– узел, извлеченный из очереди, выводится на экран;

– потомки удаленного узла заносятся в очередь;

– процесс вывода на экран завершается, когда очередь становится пустой.

Пример:

```
Program P129;
 {Создание двоичного дерева - итерация}
 type AdresaNod=^Nod;
      Nod=record
                Info : string;
             Stg, Dr : AdresaNod
          end;

      AdresaCelula=^Celula;
      Celula=record
                   Info : AdresaNod;
                   Urm : AdresaCelula
             end;

 var  T : AdresaNod;     {корень}
    Prim,                {первый элемент очереди}
  Ultim : AdresaCelula; {последний элемент очереди}

procedure IntroduInCoada(Q : AdresaNod);
 var R : AdresaCelula;
```

```
begin
  new(R);
 R^.Info:=Q;
 R^.Urm:=nil;
 if Prim=nil then begin Prim:=R; Ultim:=R end
          else begin Ultim^.Urm:=R; Ultim:=R end;
end; {IntroduInCoada}

procedure ExtrageDinCoada(var Q : AdresaNod);
var R : AdresaCelula;
begin
 if Prim=nil then writeln('Очередь пустая')
          else begin
                 R:=Prim;
                 Q:=R^.Info;
                 Prim:=Prim^.Urm;
                 dispose(R);
               end;
end; { ExtrageDinCoada }
procedure CreareArboreBinar;
var R, Q : AdresaNod;
       s : string;
begin
 T:=nil;                {Сперва дерево пустое}
 Prim:=nil; Ultim:=nil; {Сперва очередь пустая}
 writeln('Введите корень:'); readln(s);
 if s<>'' then
     begin
     new(R); {Создание корня}
     R^.Info:=s;
     T:=R;   {Инициализация адреса корня}
     IntroduInCoada(T);
    end;
 while Prim<>nil do  {Пока очередь не станет пустой}
    begin
     ExtrageDinCoada(R);
     writeln('Введите потомки узла', R^.Info);
     write('левый:  '); readln(s);
     if s='' then R^.Stg:=nil
        else
            begin
             new(Q); R^.Stg:=Q;
             Q^.Info:=s;
             IntroduInCoada(Q);
            end; { else }
   write('правый:  '); readln(s);
     if s='' then R^.Dr:=nil
```

```
         else
             begin
              new(Q); R^.Dr:=Q;
              Q^.Info:=s;
              IntroduInCoada(Q);
             end; { else }
     end; { while }
end; { CreareArboreBinar }
procedure AfisareArboreBinar;
var R : AdresaNod;
begin
 if T=nil then writeln('Пустое дерево')
     else
      begin
       writeln('Дерево состоит из:');
       Prim:=nil; Ultim:=nil;
       IntroduInCoada(T);
       while Prim<>nil do
             begin
              ExtrageDinCoada(R);
              writeln('Корень', R^.Info);
              write('  Потомки: ');
              if R^.Stg=nil then write('nil, ')
                 else begin
                       write(R^.Stg^.Info, ', ');
                       IntroduInCoada(R^.Stg);
                      end;

              if R^.Dr=nil then writeln('nil')
                 else begin
                       writeln(R^.Dr^.Info);
                       IntroduInCoada(R^.Dr);
                      end;
             end; { while }
      end; { else }
 readln;
end; { AfisareArboreBinar }

begin
 CreareArboreBinar;
 AfisareArboreBinar;
end.
```

Полезная информация, связанная с каждым узлом, вводится с клавиатуры. Признаком отсутствия потомка является нажатие клавиши <*ENTER*> (программа считывает с клавиатуры пустую строку символов). Отметим, что очередь, созданная в программе `P129`, содержит не узлы, а только их адреса.

С помощью **рекурсивного алгоритма** двоичные деревья создаются в соответствии со следующими правилами:

– создается корневой узел;
– строится левое поддерево;
– строится правое поддерево.

Согласно рекурсивному алгоритму узлы двоичного дерева на *рис. 2.11* будут созданы в следующем порядке: *A, B, D, E, H, C, F, G, I, J*.

Пример:

```
Program P130;
 {Создание двоичного дерева - рекурсия}
 type Arbore=^Nod;
      Nod=record
                    Info : string;
                  Stg, Dr : Arbore
          end;

 var  T : Arbore;    {корень}

 function Arb : Arbore;
  {создание двоичного дерева}
 var R : Arbore;
     s : string;
 begin
  readln(s);
  if s='' then Arb:=nil
     else begin
          new(R);
          R^.Info:=s;
          write('Введите левого потомка');
          writeln(' корня', s, ':');
          R^.Stg:=Arb;
          write('Введите правого потомка');
          writeln('узла', s, ':');
          R^.Dr:=Arb;
          Arb:=R;
        end;
 end; {Arb }

procedure AfisArb(T : Arbore; nivel : integer);
  {Вывод двоичного дерева на экран}
 var i : integer;
begin
  if T<>nil then
     begin
      AfisArb(T^.Stg, nivel+1);
      for i:=1 to nivel do write('    ');
     writeln(T^.Info);
```

```
      AfisArb(T^.Dr, nivel+1);
    end;
end; {AfisareArb }

begin
 writeln('Введите корень:');
 T:=Arb;
 AfisArb(T, 0);
 readln;
end.
```

Функция `Arb` вводит с клавиатуры полезную информацию для узла, находящегося в процессе создания. Если считывается пустая строка, то узел не создается и функция возвращает значение **nil**. В противном случае функция создает новый узел, записывает строку символов в поле `Info` и возвращает адрес созданного узла. В момент, когда необходимо заполнить поля `Stg` (адрес левого поддерева) и `Dr` (адрес правого поддерева), функция вызывает саму себя, переходя таким образом к построению соответствующего поддерева.

Процедура `AfisArb` выводит двоичное дерево на экран. Сначала она выводит на экран левое поддерево, корень и затем правое поддерево. Уровень каждого узла указывается на экране путем вывода соответствующего числа пробелов.

При сравнении программ `P129` и `P130` видно, что процесс обработки рекурсивных структур данных, а именно: двоичных деревьев, является более естественным и эффективным в случае использования именно рекурсивных алгоритмов.

Двоичные деревья имеют очень много применений, одним из которых является обработка арифметических выражений в трансляторах с языков программирования.

Вопросы и упражнения

❶ Как определяется *двоичное дерево*? Объясните термины: *корень, левое поддерево, правое поддерево, потомок, уровень, терминальный узел, нетерминальный узел, высота двоичного дерева.*

❷ Сформулируйте итеративные алгоритмы, предназначенные для создания двоичных деревьев и их вывода на экран.

❸ Как строится двоичное дерево с помощью рекурсивного алгоритма?

❹ Напишите программу, которая строит ваше генеалогическое дерево, рассчитанное на три-четыре поколения. Корневой узел содержит вашу фамилию, имя и год рождения, а разветвляющиеся узлы – соответствующую информацию о родителях.

❺ Как можно изменить процедуру `AfisArb` из программы `P130` для того, чтобы двоичное дерево было выведено на экран в следующем порядке: правое поддерево, корневой узел, левое поддерево?

❻ Напишите рекурсивную функцию, которая возвращает количество узлов двоичного дерева. Перепишите данную функцию в нерекурсивной форме.

❼ Организация турнира “на выбывание” представлена с помощью двоичного дерева. Узлы данного дерева содержат следующую информацию:

– фамилия (**string**);
– имя (**string**);
– дата рождения (день, месяц, год);
– гражданство (**string**).
Каждому игроку соответствует терминальный узел, а каждому матчу – нетерминальный узел. В каждый нетерминальный узел записываются данные о победителе матча, в котором участвовали два игрока из разветвляющихся узлов. Очевидно, корень содержит данные о победителе турнира.
Напишите программу, которая создает в памяти компьютера и выводит на экран дерево турнира "на выбывание".
Указание. Вначале составляется список игроков. Победители матчей первого тура заносятся во второй список. Далее создается список победителей второго тура и т. д.

❽ Как нужно изменить функцию `Arb` из программы `P130` для того, чтобы двоичное дерево строилось в следующем порядке: *A*, *C*, *G*, *J*, *I*, *F*, *B*, *E*, *H*, *D*?

❾ Функция `Arb` из программы `P130` строит двоичные деревья в следующем порядке: корневой узел, левое поддерево, правое поддерево. Напишите нерекурсивную процедуру, которая строит двоичные деревья в таком же порядке.

Указание. Используется стек, элементы которого являются узлами. Вначале стек содержит только корневой узел. Для каждого узла из вершины стека строится левое поддерево, а затем – правое. Созданные узлы заносятся в стек. После построения правого дерева соответствующий узел удаляется из стека.

2.8. Обход двоичных деревьев

Операции, которые можно выполнять с двоичными деревьями, подразделяются на две большие категории:

– операции, которые изменяют структуру дерева (вставка или удаление узлов);

– операции, которые не затрагивают структуру дерева (поиск информации, печать информации из заданного узла и т. д.).

Одной из наиболее часто встречаемых проблем при выполнении таких операций является необходимость обхода двоичного дерева.

Под **обходом дерева** понимается последовательное исследование его узлов, причем информация каждого узла извлекается только один раз. Существуют три способа обхода двоичных деревьев. Эти три способа являются рекурсивными: если дерево пустое, то при его обходе не выполняется никаких операций, иначе обход осуществляется в три этапа.

Обход *КЛП*:
– исследуется корень;
– обходится левое поддерево;
– обходится правое поддерево.

Обход *ЛКП*:
– обходится левое поддерево;
– исследуется корень;
– обходится правое поддерево.

Обход *ЛПК*:

– обходится левое поддерево;
– обходится правое поддерево;
– исследуется корень.

Обозначения *КЛП*, *ЛКП* и *ЛПК* указывают порядок обхода: *К* – корень, *Л* – левое поддерево, *П* – правое поддерево. Методы обхода двоичных деревьев показаны на *рис. 2.12*.

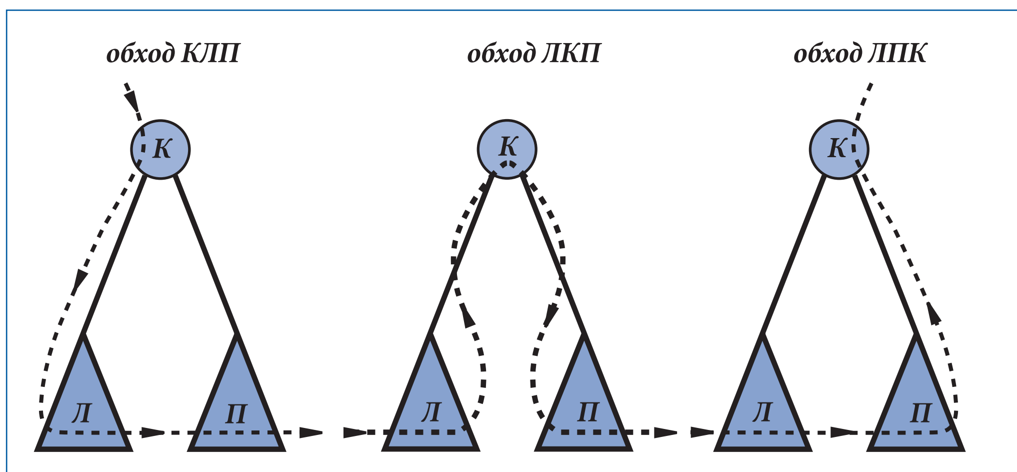


Рис. 2.12. Методы обхода двоичных деревьев

При обходе дерева на *рис. 2.11* методом *КЛП* получаем следующий порядок посещения узлов:

A, B, D, E, H, C, F, G, I, J;

при обходе методом *ЛКП* получаем следующий порядок посещения узлов:

D, B, E, H, A, F, C, I, G, J;

обход методом *ЛПК* приводит к следующему порядку посещения узлов:

D, H, E, B, F, I, J, G, C, A.

Ниже представлена программа обхода двоичного дерева по всем трем методам.

```
Program P131;
 {Обход двоичного дерева}
 type Arbore=^Nod;
      Nod=record
               Info : string;
             Stg, Dr : Arbore
          end;
var  T : Arbore;    {корень}
```

```
 function Arb : Arbore;
  {создание двоичного дерева}
 var R : Arbore;
     s : string;
 begin
  readln(s);
  if s='' then Arb:=nil
     else begin
          new(R);
          R^.Info:=s;
          write('Введите левого потомка');
          writeln(' узла ', s, ':');
          R^.Stg:=Arb;
          write('Введите правого потомка');
          writeln(' узла ', s, ':');
          R^.Dr:=Arb;
          Arb:=R;
         end;
 end; {Arb }

 procedure AfisArb(T : Arbore; nivel : integer);
  {вывод двоичного дерева на экран}
 var i : integer;
 begin
  if T<>nil then
     begin
      AfisArb(T^.Stg, nivel+1);
      for i:=1 to nivel do write('    ');
      writeln(T^.Info);
     AfisArb(T^.Dr, nivel+1);
     end;
 end; {AfisareArb }

 procedure Preordine(T : Arbore);
  {обход КЛП}
 begin
  if T<>nil then begin
                  writeln(T^.Info);
                  Preordine(T^.Stg);
                  Preordine(T^.Dr)
                end;
 end; {Preordine }

procedure Inordine(T : Arbore);
  {обход ЛКП}
 begin
  if T<>nil then begin
```

```
                    Inordine(T^.Stg);
                    writeln(T^.Info);
                    Inordine(T^.Dr)
                  end;
end; {Preordine }

procedure Postordine(T : Arbore);
 {обход ЛПК}
begin
 if T<>nil then begin
                    Postordine(T^.Stg);
                    Postordine(T^.Dr);
                    writeln(T^.Info)
                  end;
end; { Postordine }

begin
 writeln('Введите корень:');
 T:=Arb;
 AfisArb(T, 0);
 readln;
 writeln('обход КЛП:');
 Preordine(T);
 readln;
writeln('обход ЛКП:');
 Inordine(T);
 readln;
 writeln('обход ЛПК:');
 Postordine(T);
 readln;
end.
```

Отметим, что функция `Arb` создает узлы, совершая в процессе построения обход двоичного дерева методом *КЛП*. Процедура `AfisArb` выводит на экран узлы, совершая обход двоичного дерева методом *ЛКП*.

Вопросы и упражнения

1. Какие операции можно применять к двоичным деревьям?
2. Объясните методы обхода двоичных деревьев. Приведите примеры.
3. Напишите списки узлов, получаемые при каждом методе обхода двоичного дерева на *рис. 2.13*.
4. Перепишите в нерекурсивной форме процедуры `Preordine`, `Inordine` и `Postordine` из программы `P131`.
5. Напишите подпрограмму, которая вычисляет высоту двоичного дерева.
6. Напишите программу, которая выводит на экран все узлы, находящиеся на заданном уровне двоичного дерева.

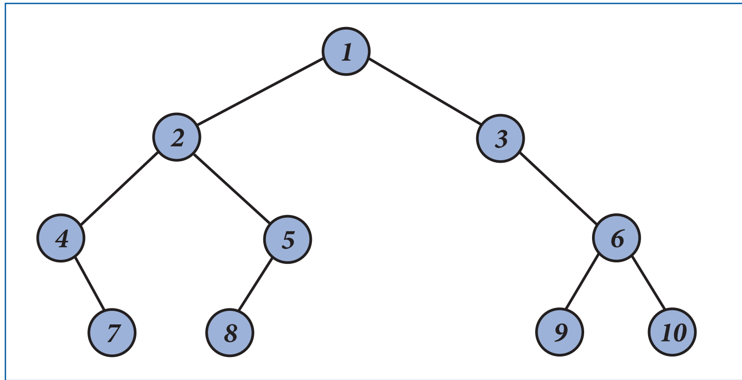


Рис. 2.13. Двоичное дерево

❼ Напишите рекурсивную процедуру, которая обходит двоичное дерево в порядке:

а) КПЛ (корень – правое поддерево – левое поддерево);

б) ПКЛ (правое поддерево – корень – левое поддерево);

в) ПЛК (правое поддерево – левое поддерево – корень):

Перепишите созданную процедуру в нерекурсивном виде.

❽ Напишите программу, которая выводит на экран уровень каждого узла двоичного дерева.

❾ Дано двоичное дерево, в котором терминальные узлы представляют целые числа, а нетерминальные – бинарные операции: +, –, *, **mod, div**. Данное дерево можно рассматривать как представление арифметического выражения. Значение этого выражения вычисляется, начиная с корня, путем выполнения операций над значениями подвыражений, представленных левыми и правыми поддеревьями. Напишите функцию, которая возвращает значение арифметических выражений, представленных двоичными деревьями.

❿ Даны арифметические выражения, составленные из операндов и двоичных операций: +, –, *, /. Операндами являются переменные, имена которых состоят из одной буквы, и константы, состоящие из одной цифры. Каждому арифметическому выражению *i* можно поставить в соответствие двоичное дерево:

а) арифметическому выражению, составленному из одного операнда *i*, ставится в соответствие двоичное дерево, состоящее только из корня, который содержит соответствующий операнд;

б) арифметическому выражению вида $E_1 \circ E_2$, где E_1 и E_2 являются арифметическими выражениями, ставится в соответствие двоичное дерево, в корне которого содержится оператор "∘", левым поддеревом является выражение E_1, а правым – выражение E_2.

Значение выражения вычисляется, начиная с корня, путем выполнения операций над значениями подвыражений, представленных левыми и правыми поддеревьями. Напишите программу, которая:

а) строит двоичные деревья для арифметических выражений, вводимых с клавиатуры;

б) вычисляет арифметические выражения, представленные двоичными деревьями.

Указание. Алгоритм должен следовать рекурсивному определению рассматриваемого дерева. В качестве текущего оператора "∘" может выступать любой оператор +, или – из выражения, подлежащего обработке. Операторы * и / могут выступать в роли текущих операторов только в том случае, когда выражение, подлежащее обработке, не содержит операторов +, –.

2.9. Деревья *m*-го порядка

Рассмотрим динамические переменные типа **record**, в поле связей которых содержатся $m \geq 2$ указателей адреса. Как и в случае двоичных деревьев, будем называть такие переменные **узлами**.

Дерево *m*-го порядка определяется рекурсивно:

а) отдельный узел является деревом *m*-го порядка;

б) узел, который содержит не более *m* связей с другими деревьями *m*-го порядка, также является деревом *m*-го порядка.

Считается, что дерево содержит, по крайней мере, хоть один узел, от которого ответвляется ровно *m* поддеревьев. По соглашению, **пустое дерево** не содержит ни одного узла.

Деревья 2-го порядка называются **двоичными деревьями** и изучались в предыдущих параграфах. Деревья 3-го, 4-го, 5-го порядка и т. д. называются **сильно ветвящимися деревьями** (по-английски *multiway trees*).

В качестве примера на *рис. 2.14* представлено дерево 4-го порядка. Очевидно, что для деревьев *m*-го порядка термины: *корень, поддерево, уровень, родитель, потомок, терминальный узел, нетерминальный узел, высота* имеют такое же значение, как и для двоичных деревьев. Терминология, используемая для таких структур данных, включает даже такие слова, как *сын, отец, братья, дядя, двоюродные братья, прадедушка* и др., смысл которых для узлов, находящихся на разных уровнях, совпадает со значениями этих слов в обычной речи. Проще говоря, такие структуры данных выражают отношения "разветвления" между узлами, подобно строению обычных деревьев, с той лишь разницей, что в информатике деревья "растут" вниз головой.

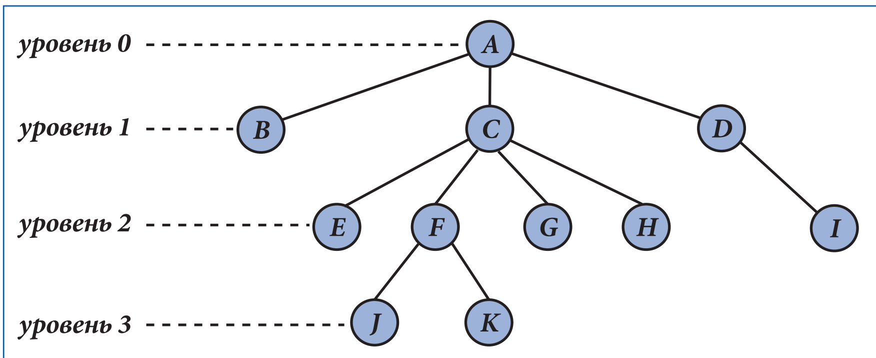


Рис. 2.14. Дерево 4-го порядка

Данные, необходимые для создания и обработки дерева *m*-го порядка, можно описывать с помощью следующих объявлений:

```
type Arbore = ^Nod;
      Nod = record;
      Info : string;
      Dsc : array [1..m] of Arbore
        end;
var T : Arbore;
```

Адреса потомков узла запоминаются в компонентах: Dsc[1], Dsc[2], ... , Dsc[m] массива Dsc. Адрес корня сохраняется в переменной ссылочного типа T.

Самыми распространенными методами обхода деревьев *m*-го порядка являются обход в ширину и обход в глубину.

При **обходе в ширину** узлы обходятся в порядке их появления на уровнях. Например, для дерева на *рис. 2.14* обход узлов будет осуществлен в следующем порядке: *A, B, C, D, E, F, G, H, I, J, K.*

Обычно обход в ширину реализуется с помощью итеративного алгоритма, который использует вспомогательную структуру данных, а именно: очередь, составленную из адресов обходимых узлов.

Обход в глубину определяется рекурсивно: если дерево – пустое, то при обходе не выполняется никаких действий; если непустое, то сначала производится обход корня, затем обход деревьев слева направо. При обходе в глубину дерева на *рис. 2.14* узлы посещаются в следующем порядке: *A, B, C, E, F, J, K, G, H, D, I*. Обход в глубину реализуется легко с помощью рекурсивного алгоритма.

Пример:

```
Program P133;
 {Деревья m-го порядка}
 const m=4;
 type Arbore=^Nod;
      Nod=record
                    Info  : string;
                    Dsc  : array [1..m] of Arbore
          end;

      AdresaCelula=^Celula;
      Celula=record
                       Info  : Arbore;
                        Urm  : AdresaCelula
             end;

 var  T  : Arbore;      {корень}
    Prim,               {первый элемент очереди}
  Ultim : AdresaCelula; {последний элемент очереди}

 procedure IntroduInCoada(Q : Arbore);
 var R : AdresaCelula;
 begin
  new(R);
  R^.Info:=Q;
  R^.Urm:=nil;
  if Prim=nil then begin Prim:=R; Ultim:=R end
           else begin Ultim^.Urm:=R; Ultim:=R end;
 end; {IntroduInCoadă }
```

```
 procedure ExtrageDinCoada(var Q : Arbore);
 var R : AdresaCelula;
 begin
  if Prim=nil then writeln('Очередь пустая')
           else begin                  R:=Prim;
                  Q:=R^.Info;
                  Prim:=Prim^.Urm;
                  dispose(R);
                end;
 end; {ExtrageDinCoadă }

procedure CreareArbore(var T : Arbore);
 var R, Q : Arbore;
        s : string;
        i : integer;
 begin
 T:=nil;                {изначально дерево пустое}
  Prim:=nil; Ultim:=nil; {изначально очередь пустая}
  writeln('Введите корень: '); readln(s);
  if s<>'' then
     begin
       new(R); {создание корня}
       R^.Info:=s;
       T:=R;   {инициализация адреса корня}
       IntroduInCoada(T);
     end;
  while Prim<>nil do  {пока очередь не является пустой}
     begin
       ExtrageDinCoada(R);
       for i:=1 to m do R^.Dsc [i]:=nil;
       writeln('Введите потомков узла',R^.Info);
       i:=1; readln(s);
       while (i<=m) and (s<>'') do
         begin
           new(Q); R^.Dsc [i]:=Q; Q^.Info:=s;
           IntroduInCoada(Q);
           i:=i+1; readln(s);
         end;
  end;
 end; {CreareArbore }

 procedure AfisareArbore(T : Arbore);
 var R : Arbore;
     i : integer;
begin
  if T=nil then writeln('Дерево пустое')
```

```
      begin
        writeln('Дерево содержит:');
        Prim:=nil; Ultim:=nil;
        IntroduInCoada(T);
        while Prim<>nil do
           begin
             ExtrageDinCoada(R);
             writeln('Узел', R^.Info);
             write('Потомки: ');
             for i:=1 to m do
               if R^.Dsc [i]<>nil then
                  begin
                     write(R^.Dsc [i]^.Info, ' ');
                     IntroduInCoada(R^.Dsc [i]);
                  end; {then }
             writeln;
           end; {while }
      end; {else }
 readln;
end; {AfisareArbore }

procedure InLatime(T : Arbore);
var R : Arbore;
    i : integer;
begin
 if T<>nil then
    begin
      Prim:=nil; Ultim:=nil;
      IntroduInCoada(T);
      while Prim<>nil do
        begin
          ExtrageDinCoada(R);
          writeln(R^.Info);
          for i:=1 to m do
           if R^.Dsc [i]<>nil then IntroduInCoada(R^.Dsc [i]);
        end; {while }
      end; {then }
end; {InLatime }

procedure InAdincime(T : Arbore);
var i : integer;
begin
 if T<>nil then
   begin
      writeln(T^.Info);
      for i:=1 to m do InAdincime(T^.Dsc [i]);
    end;
end; {InAdincime }
```

```
begin
 CreareArbore(T);
 AfisareArbore(T);
 writeln('Обход дерева в ширину:');
 InLatime(T);
 readln;
 writeln('Обход дерева в глубину:');
 InAdincime(T);
 readln;
end.
```

Полезная информация каждого узла вводится с клавиатуры. Признаком отсутствия потомков является нажатие клавиши <*ENTER*>. Отметим, что процедура `CreareArbore` в процессе создания узлов осуществляет обход дерева в ширину. Очевидно, что процедура `AfisareArbore` обходит узлы в порядке их создания.

К деревьям *m*-го порядка наиболее часто применяются следующие **операции**: вставка или удаление узла, поиск информации, обработка полезной информации, связанной с узлами, и т. д. Обычно деревья *m*-го порядка используются в приложениях, которые влекут за собой обработку большого количества данных, организованных иерархически на внешних носителях информации. В качестве примера отметим организацию данных на магнитных и оптических дисках в операционных системах MS-DOS, UNIX и т.д. Данные деревья также используются в графических приложениях для представления динамических связей между компонентами изображений.

Вопросы и упражнения

1. Приведите примеры деревьев 3-го, 5-го и 6-го порядка.
2. Как определяется дерево *m*-го порядка? Какие операции можно применять к данным деревьям?
3. Объясните методы обхода деревьев *m*-го порядка. Приведите примеры.
4. Напишите рекурсивную программу, которая строит в памяти компьютера дерево *m*-го порядка. Полезная информация, связанная с узлами, считывается с клавиатуры.
5. Напишите функцию, которая возвращает:
   а) количество узлов дерева *m*-го порядка;
   б) уровень определенного узла дерева;
   в) высоту дерева.
6. Перепишите в нерекурсивной форме процедуру `InAdincime` из программы `P133`.
7. Как можно изменить процедуру `InLatime` из программы `P133`, чтобы обход узлов дерева на *рис. 2.14* осуществлялся в следующем порядке: *A*, *D*, *C*, *B*, *I*, *H*, *G*, *F*, *E*, *K*, *J*?
8. Как можно изменить процедуру `InAdincime` из программы `P133`, с тем чтобы обход узлов дерева на *рис. 2.14* осуществлялся в следующем порядке: *A*, *D*, *I*, *C*, *H*, *G*, *F*, *K*, *J*, *E*, *B*?
9. Дано дерево *m*-го порядка, в котором информация, находящаяся в узлах, представляет собой строки символов. Вывести на экран все строки символов четной длины.

⑩ Организация данных на магнитных дисках задается с помощью дерева *m*-го порядка. Терминальные узлы представляют собой файлы, а нетерминальные – каталоги. Полезная информация, связанная с каждым узлом, включает:
– имя файла или каталога (**string**[8]);
– расширение (**string[3]**);
– дату и время последнего изменения (соответственно день, месяц, год и час, минута, секунда);
– длину (integer);
– атрибуты ('A', 'H', 'R', 'S').
Напишите программу, которая имитирует операции поиска, создания и удаления файлов и каталогов.

⑪ Иногда порядок *m* дерева не известен на момент написания программы, поэтому становится невозможным использование типа данных **array** [1..m] **of** Arbore. Для устранения этого недостатка соответствующий массив можно заменить на односвязный список.

Напишите подпрограммы, необходимые для создания и обработки деревьев с произвольным заранее неизвестным порядком.

2.10. Тип данных pointer

Данный параграф полностью относится к Turbo PASCAL.

Множество значений предопределенного типа pointer (указатель) состоит из адресов и специального значения **nil**. В отличие от ссылочных типов данных, адреса которых указывают только на динамические переменные, принадлежащие базовому типу, значения типа pointer указывают на динамические переменные любого типа. Очевидно, значение **nil** не указывает ни на какую динамическую переменную. По соглашению, тип данных pointer **совместим** с любым ссылочным типом.

К значениям типа данных pointer можно применять **операции**: = и <>. Значения этого типа не могут быть считаны с клавиатуры или выведены на экран.

Переменная типа pointer вводится описанием вида:

```
var p : pointer;
```

Так как такое описание не содержит информацию о базовом типе, тип динамической переменной p^ неизвестен. Следовательно, на переменные типа pointer нельзя ссылаться с помощью символа ^, наличие которого после такой переменной является ошибкой.

Следующая программа показывает использование переменных типа pointer для временного запоминания значений переменных ссылочного типа.

```
Program P134;
  { Тип данных pointer }
 var p : pointer;
  i, j : ^integer;
  x, y : ^real;
  r, s : ^string;
```

```
begin
  {p будет указывать на динамическую переменную типа integer }
  new(i); i^:=1;
  p:=i;
  new(i); i^:=2;
  j:=p;
  writeln('j^=', j^); { на экран выводится 1 }
  {p будет указывать на динамическую переменную типа real }
  new(x); x^:=1;
  p:=x;
  new(x); x^:=2;
  y:=p;  writeln('y^=', y^); {на экран выдается 1.0000000000E+00 }
  {p будет указывать на динамическую переменную string }
  new(r); r^:='AAA';
  p:=r;
  new(r); r^:='BBB';
  s:=p;
  writeln('s^=', s^); { на экран выводится AAA }
  readln;
 end.
```

Основной областью использования переменных типа pointer является распределение внутренней памяти компьютера. В Turbo PASCAL динамические переменные заносятся в специальную область внутренней памяти, называемой **heap** (куча). Адрес начала этой области, называемый **базовым адресом**, заносится в предопределенную переменную `HeapOrg` типа `pointer`. Переменная `HeapPtr` типа `pointer` содержит адрес первой свободной зоны, называемой **вершиной** *heap*-а (*рис.2.15*).

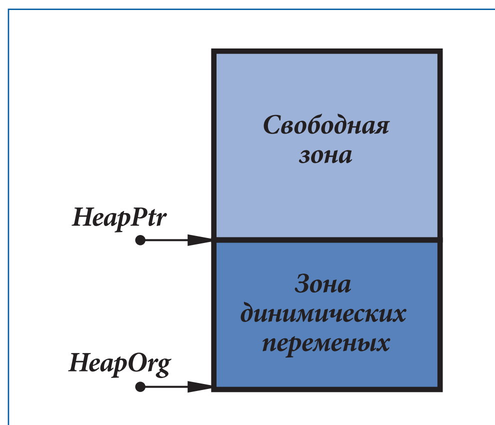


Рис. 2.15. Структура области *heap*

Динамические переменные создаются и заносятся в *heap* с помощью процедуры `new`. Когда в вершине области *heap* создается динамическая переменная, содержание переменной `HeapPtr` изменяется: текущее значение увеличива-

ется на размер области памяти, необходимой для размещения динамической переменной.

Память, занимаемая в области heap динамической переменной, освобождается с помощью процедуры dispose. Размер освобождаемого пространства зависит от типа динамической переменной.

Порядок вызова процедуры dispose произвольный и, как правило, не совпадает с порядком создания динамических переменных с помощью процедуры new. Вследствие этого в области *heap* возникают пустые зоны. Пустые зоны могут использоваться процедурой new только в том случае, если вновь создаваемая динамическая переменная «вмещается» в соответствующую зону.

Область памяти, занимаемую динамической структурой данных, можно освободить, применяя процедуру dispose к каждой компоненте. Так как программе известны только адреса отдельных компонент, как правило: базы и вершины списка, корня дерева и т. д., поиск остальных компонент является задачей программиста. Более того, порядок вызова процедуры dispose должен быть таким, чтобы сохранились связи к компонентам, которые еще не были уничтожены. В противном случае ни один указатель адреса не будет ссылаться на соответствующие компоненты, и они становятся недоступными. Таким образом, использование процедуры dispose для освобождения области памяти, занимаемой сложными структурами данных, является сложным и неэффективным. Этого можно избежать, применяя стандартные процедуры mark и release.

Вызов процедуры mark имеет вид:

```
mark(p)
```

где p является переменной типа pointer. Процедура запоминает адрес вершины из HeapPtr в переменную p.

Вызов процедуры release имеет вид:

```
release(p)
```

Эта процедура переводит адрес вершины на позицию, зарегистрированную ранее процедурой mark: значение, содержащееся в переменной p типа pointer, заносится в указатель HeapPtr.

Распределение области памяти, предназначенной для динамических переменных, можно осуществлять с помощью следующего алгоритма:

1) с помощью процедуры mark запоминается адрес вершины;

2) с помощью процедуры new создаются динамические переменные;

3) созданные динамические переменные используются;

4) когда динамические переменные уже не нужны, пространство, занимаемое ими в области heap, освобождается с помощью процедуры release.

Пример:

Даны следующие описания:

```
var i, j, k, m, n : ^integer;
    p : pointer;
```

Предположим, что выполняются операторы:

```
new(i); i^:=1;
new(j); j^:=2;
mark(p);
new(k); k^:=3;
new(m); m^:=4;
new(n); n^:=5;
```

Состояние области *heap* представлено на *рис. 2.16 а*. Перед тем как создать динамическую переменную `k^`, оператор `mark(p)` запоминает в переменной `p` типа `pointer` фактическое значение из `HeapPtr`.

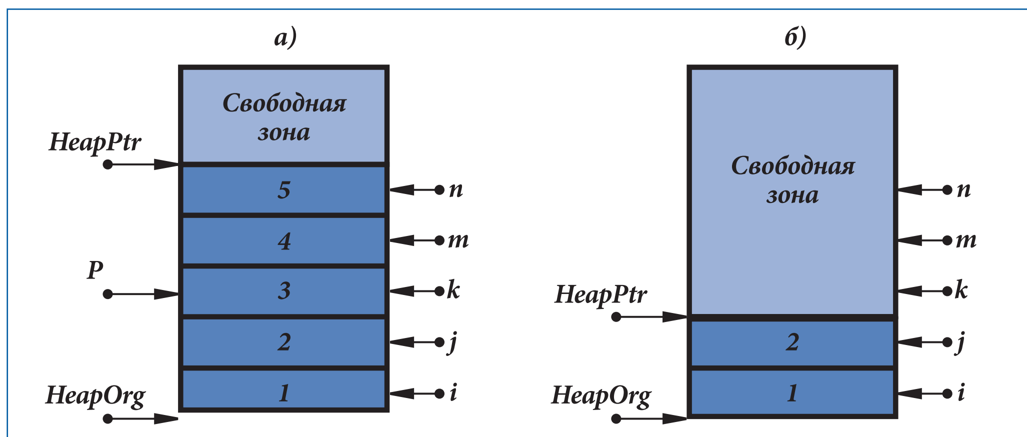


Рис. 2.16. Состояние области *heap* до (*а*) и после (*б*) выполнения оператора `release(p)`

Если теперь выполняется оператор

```
release(p)
```

то область памяти, занимаемая динамическими переменными после вызова процедуры `mark`, а именно: `k^`, `m^` и `n^`, будет освобождена (*рис. 2.16 б*).

Так как предопределенная переменная `HeapOrg` сохраняет базовый адрес области *heap*, то все пространство, предназначенное для динамических переменных, можно освободить с помощью оператора:

```
release(HeapOrg)
```

Следующая программа наглядно показывает использование процедур `mark` и `release`.

```
Program P135;
  {Управление внутренней памятью }
 type Lista=^Celula;
      Celula=record
                   Info : string;
                    Urm : Lista
             end;
```

```
      Stiva=Lista;
          end;
 var L : Lista;
     S : Stiva;
     T : Arbore;
     p : pointer;

function Lst : Lista;
  {Создание односвязного списка}
 var R : Lista;
     s : string;
 begin
  write('Info='); readln(s);
  if s='' then Lst:=nil
     else
         begin
           new(R);
           R^.Info:=s;
           R^.Urm:=Lst;
           Lst:=R;
         end;
 end; {Lst }

 procedure AfisLst(L : Lista);
  {вывод списка на экран}
 begin
  if L<>nil then
     begin
       writeln(L^.Info);
       AfisLst(L^.Urm);
     end;
 end; {AfisLst }

 procedure Stv(var S : Stiva);
  {создание стека}
 var R : Stiva;
    st : string;
 begin
  S:=nil;
  write('Info='); readln(st);
  while st<>'' do
       begin
          new(R);
          R^.Info:=st;
          R^.Urm:=S;
         S:=R;
```

```
        write('Info='); readln(st);
      end;
 end; { Stv }

function Arb : Arbore;
  {создание двоичного дерева}
 var R : Arbore;
     s : string;
 begin
  readln(s);
  if s='' then Arb:=nil
     else begin
           new(R);
           R^.Info:=s;
           write('Введите левого потомка');
           writeln(' узла ', s, ':');
           R^.Stg:=Arb;
           write('Введите правого потомка');
           writeln(' узла ', s, ':');
           R^.Dr:=Arb;
           Arb:=R;
          end;
 end; { Arb }

 procedure AfisArb(T : Arbore; nivel : integer);
  { вывод двоичного дерева на экран }
 var i : integer;
 begin
  if T<>nil then
     begin
      AfisArb(T^.Stg, nivel+1);
      for i:=1 to nivel do write('    ');
      writeln(T^.Info);
      AfisArb(T^.Dr, nivel+1);
     end;
 end; {AfisArb }
 begin
  writeln('Введите список:');
  L:=Lst;
  writeln('Созданный список:');
  AfisLst(L);
  mark(p);    { p сохраняет адрес из HeapPtr }
  writeln('Введите корень:');
  T:=Arb;
  writeln('Созданное дерево:');
  AfisArb(T, 0);
 release(p);{освобождение области памяти, занимаемой деревом}
```

```
   writeln('Введите стек:');
   Stv(S);
   writeln('Созданный стек');
   AfisLst(S);
   release(HeapOrg); {освобождение области памяти, занимаемой
 списком и стеком}
   readln;
 end.
```

Отметим, что в нынешних версиях Turbo PASCAL процедуры `dispose` и `release` не присваивают значение **`nil`** указателю адреса переменной, которая была уничтожена (*рис. 2.16 б*). Так как освобожденная область памяти используется повторно, то присвоение значений уже уничтоженным переменным может изменить значения вновь созданных динамических переменных.

Вопросы и упражнения

❶ Из чего состоит множество значений типа данных `pointer`? Какие операции можно применять к этим значениям?

❷ Прокомментируйте следующую программу:

```
Program P136;
  {Eroare }
 var i : ^integer;
   j, k : integer;
   p : pointer;
 begin
  new(i); i^:=1;
  p:=i;
  new(i); i^:=2;
  j:=i^; k:=p^;
  writeln('j+k=', j+k);
 end.
```

❸ Когда целесообразно использование переменных типа `pointer`?

❹ Является ли область `heap` структурой данных типа `стек`? Аргументируйте ваш ответ.

❺ Запустите следующие программы. Объясните результаты, выводимые на экран.

```
Program P137;
 var i, j, k, m, n : ^integer;
                p : pointer;
 begin
  {создание переменных i^, j^, k^ }
  new(i); new(j); new(k);
  i^:=1; j^:=2; k^:=3;
  p:=j; {p сохраняет адрес из j }
  {уничтожение переменной j^ и создание переменной m^}
  dispose(j); new(m); m^:=4;
```

```
   j:=p; {восстановление адреса из j }
   writeln('i^=', i^, '  j^=', j^, '  k^=', k^);
   {уничтожение переменной m^ и создание переменной n^}
   dispose(m); new(n); n^:=5;
   writeln('i^=', i^, '  j^=', j^, '  k^=', k^);
   readln;
  end.
```

```
Program P138;
 var i, j, k, m : ^integer;
 begin

  {создание переменных i^, j^ }
  new(i); new(j);
  i^:=1; j^:=2;

  {освобождение области heap}
  release(HeapOrg);

  {создание переменных: k^ и m^}
  new(k); new(m);

  k^:=1; m^:=2;
  writeln('k^=', k^, ' m^=', m^);
  i^:=3; j^:=4;
  writeln('k^=', k^, ' m^=', m^);
  readln;
 end.
```

❻ Напишите процедуру, которая освобождает область памяти, занимаемую:

а) односвязным списком;

б) двоичным деревом;

в) деревом *m*-го порядка.

Область памяти должна освобождаться путем применения процедуры `dispose` ко всем компонентам динамической структуры данных.

❼ Напишите программу, которая вначале создает очередь, а затем дерево *m*-го порядка. Область памяти, освобожденную после уничтожения очереди, необходимо использовать для размещения создаваемого дерева.

Глава 3

МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ

3.1. Модульное программирование

Модульное программирование снижает уровень сложности больших программ путем их разбиения на модули.

Модуль – это программный продукт, состоящий из описания данных и подпрограмм для их обработки. В процессе написания программы модули могут создаваться независимо и компилироваться отдельно, до их включения в программу. Отметим, что сама программа называется главным модулем.

В стандартном языке не предусмотрены средства для создания модулей. Программа на языке ПАСКАЛЬ является единым целым и должна компилироваться вместе со всеми своими подпрограммами. Однако это не удобно в случае больших программ, которые содержат десятки или даже сотни подпрограмм.

В версии Turbo PASCAL модули реализуются с помощью **unit**-ов. Общая форма модуля имеет вид:

```
unit <Имя>;
interface
[uses <Имя>{,<Имя>};]
[<Константы>]
[<Типы>]
[<Переменные>]
[{<Заголовок функции>; | <Заголовок процедуры>;}]

implementation
[uses <Имя> {,<Имя};]
[<Метки>]
[<Константы>]
[<Типы>]
[<Переменные>]
[<Подпрограммы>]
[{function <Идентификатор>;
<Corp>; |
 procedure <Идентификатор>;
 <Блок>;}]
 [begin
 [<Оператор> {; <Оператор}]]
 end.
```

Модуль программы состоит из трех разделов: интерфейсный раздел, раздел реализации, раздел инициализации.

Интерфейсный раздел начинается с ключевого слова **interface.** Здесь находятся описания констант, типов, переменных и подпрограмм, экспортируемых модулем. Эти элементы доступны любому модулю, который использует рассматриваемый модуль непосредственно или по транзитивности. Отметим, что в интерфейсном разделе находятся только заголовки экспортируемых процедур и функций. Если в данном модуле используются другие модули, то их имена указываются в строке **uses**.

Раздел реализации начинается с ключевого слова **implementation.** В этом разделе находятся локальные описания меток, констант, типов, переменных и подпрограмм. Элементы, определенные здесь, «скрыты» и не доступны для других модулей. После локальных описаний следует тело процедур и функций, заголовки которых были определены в интерфейсном разделе. Каждая подпрограмма, указанная в интерфейсном разделе, должна иметь тело. После ключевого слова **function** или **procedure** следует только имя подпрограммы. Отметим, что здесь нет необходимости описывать список параметров и возвращаемых значений.

Раздел инициализации, если таковой существует, начинается с ключевого слова **begin**. Раздел состоит из последовательности операторов и служит для присвоения исходных значений переменным, определенным в интерфейсном разделе. Если программа использует несколько модулей, то выполнению программы предшествует выполнение разделов инициализации в порядке их появления в строке **uses** программы.

Пример:

```
Unit U1;
  {Обработка массивов}
 interface

  const nmax=100;
  type Vector=array [1..nmax] of real;
  var n : 1..nmax;
  function sum(V : Vector) : real;
  function min(V : Vector) : real;
  function max(V : Vector) : real;
  procedure Citire(var V : Vector);
  procedure Afisare(V : Vector);

 implementation

  var i : 1..nmax;
      s : real;

  function sum;
  begin
       s:=0;
       for i:=1 to n do s:=s+V [i];
```

```
      sum:=s;
  end; {sum }

  function min;
  begin
       s:=V [1];
       for i:=2 to n do
       if s>V [i] then s:=V [i];
       min:=s;
  end; {min }

  function max;
  begin
       s:=V [1];
       for i:=2 to n do
       if s<V [i] then s:=V [i];
       max:=s;
  end; {max }

  procedure Citire;
  begin
       for i:=1 to n do readln(V [i]);
  end; {Citire }

  procedure Afisare;
  begin
       for i:=1 to n do writeln(V [i]);
  end; {Afisare }

 begin
  write('n='); readln(n);
 end.
```

Модуль U1 экспортирует константу nmax, тип Vector, переменную n, функции sum, min, max, процедуры Citire и Afisare. Исходное значение переменной n считывается с клавиатуры. Эти элементы доступны любой программе, которая содержит строку: **uses U1**.

Пример:

```
Program P139;
  {Использование модуля U1}
 uses U1;
var A : Vector;
 begin
  writeln('Введите массив:');
  Citire(A);
  writeln('Был введен массив:');
```

```
  Afisare(A);
  writeln('sum=', sum(A));
  writeln('min=', min(A));
  writeln('max=', max(A));
  readln;
 end.
```

Область видимости описаний из модулей программы устанавливается согласно следующим правилам:

1. Описания из раздела реализации видимы только в данном модуле.
2. Описания из интерфейсного раздела видимы:
   – в данном модуле;
   – в модулях, которые непосредственно используют данный модуль;
   – в модулях, которые используют данный модуль по транзитивности.
3. Если один и тот же идентификатор описан в нескольких модулях, то во внимание принимается самое последнее описание.

Пример:

```
Program P140;
 uses U2;
 var x : integer;
 begin
  x:=4;
  writeln('Программа P140:');
  writeln('n=', U3.n);
  writeln('m=', m);
  writeln('x=', x);
  readln;
 end.
```

```
Unit U2;
interface
 uses U3;
 var m : integer;
     x : real;
 implementation
 begin
  writeln('Модуль U2:');
  m:=2;
  writeln('    m=', m);
  x:=3.0;
  writeln('    x=', x);
 end.
Unit U3;
interface
 var n : integer;
implementation
```

```
begin
 writeln('Модуль U3:');
 n:=1;
 writeln('n=', n);
end.
```

В программе P140 модуль U2 используется непосредственно, а модуль U3 по транзитивности. Ссылка на переменную n из модуля U3 производится через U3.n. Идентификатор x появляется в описаниях **var** x: real в модуле U2 и **var** x: integer в программе P140. Компилятор принимает во внимание последнее описание.

Модули программ классифицируются на стандартные модули, содержащиеся в компиляторе Turbo PASCAL, и модули, написанные пользователем. Далее следует краткая характеристика часто используемых стандартных модулей:

System – содержит все стандартные подпрограммы из Turbo PASCAL. Данный модуль автоматически вставляется во все программы, без включения строки **uses**.

Crt – позволяет использовать процедуры и функции для работы с экраном в текстовом режиме, а также с клавиатурой и громкоговорителем. Доступ к подпрограммам осуществляется через строку **uses** crt.

Graph – включает подпрограммы, предназначенные для графической обработки: выбор окон и страниц, выбор цвета и палитры, создание дуг, окружностей, многоугольников и других фигур, сохранение изображений и т. д. Доступ к подпрограммам данного модуля осуществляется через строку **uses** graph.

Printer – заменяет принтер на текстовой файл с именем lst. При использовании данного модуля программисту не требуется объявлять, открывать и закрывать данный файл. Модуль Printer становится доступным программе или модулю программы при наличии строки **uses** printer.

Назначение и способ использования констант, типов данных, переменных, функций и процедур стандартных модулей можно найти в руководствах по использованию и в системе помощи *Turbo PASCAL's Online Help*.

При разработке собственных модулей любой пользователь может создавать библиотеки подпрограмм, описывающих алгоритмы из различных областей: решение уравнений, статистические вычисления, обработка текстов, создание и обработка динамических структур данных и т. д. Разбиение большой программы на модули облегчает процесс разработки сложных программных продуктов. В таких случаях программист разрабатывает и отлаживает несколько сравнительно простых модулей, что улучшает качество программного продукта.

Вопросы и упражнения

❶ В чем преимущества модульного программирования? Предусмотрены ли в стандартном языке средства для модульного программирования?

❷ Какова стандартная форма модуля программы? Объясните структуру и назначение интерфейсного раздела, раздела реализации и раздела инициализации.

❸ Как определяется область видимости описаний в модулях программы?

❹ Что выведет на экран следующая программа?

```
Program P141;
uses U4;
var s : string;
begin
  s:='BBB';
  writeln('U5.k=', U5.k);
  writeln('U5.m=', U5.m);
  writeln('U5.s=', U5.s);
  writeln('U4.m=', U4.m);
  writeln('U4.s=', U4.s);
  writeln('m=', m);
  writeln('s=', s);
  readln;
end.
```

```
Unit U4;
interface
uses U5;
var m : real;
   s : char;
implementation
begin
  m:=4.0;
  s:='A';
end.
Unit U5;
interface
var k, m : integer;
   s : real;
implementation
begin
  k:=1;
  m:=2;
  s:=3.0;
end.
```

❺ Прокомментируйте следующую программу:

```
Program P142;
  {Ошибка}
uses U6;
begin
  writeln('k=', k);
  writeln('m=', m);
  readln;
end.
```

```
Unit U6;
interface
 var k : integer;
implementation
 var m : integer;
begin
  k:=1;
  m:=2;
end.
```

❻ Дополните модуль U1 из данного параграфа подпрограммой, которая:
а) возвращает среднее арифметическое значение элементов массива;
б) располагает элементы массива в порядке их возрастания;
в) возвращает произведение элементов массива;
г) возвращает количество положительных элементов массива;
д) располагает элементы массива в порядке убывания.

❼ Напишите модуль, который содержит описания типов данных и подпрограммы для обработки матриц.

❽ Целые числа *n*, $n \leq 10^{254}$, могут быть представлены в компьютере в виде строк, составленных из символов: '+', '-', '0', '1', '2', ..., '9'. Напишите модуль, который содержит процедуры и функции, необходимые для выполнения следующих операций:
а) чтение чисел с клавиатуры;
б) вывод чисел на экран
в) +, –, *, **mod**, **div**;
г) вычисление факториала;
д) чтение и запись чисел в файлы последовательного доступа.

❾ В программах Turbo PASCAL строки символов длины *n*, $n \leq 500$ могут представляться переменными типа:

```
type lungime = 0..500;
     SirDeCaractere = record
                        n: lungime;
                        s: array [1..500] of char
                      end;
```

Напишите модуль, который содержит процедуры и функции, необходимые для выполнения следующих операций:
а) чтение строк с клавиатуры;
б) вывод строк на экран;
в) конкатенация (склеивание) строк;
г) лексикографическое сравнение;
д) вычисление длины строки.

❿ Напишите модуль для обработки:
а) односвязных списков;
б) очередей;
в) стеков;

г) двоичных деревьев;

д) деревьев *m*-го порядка.

Для этого используйте описания типов, функций и процедур из главы 2.

⓫ Найдите в системе помощи *Turbo PASCAL's Online Help* описание стандартных модулей, установленных на вашем компьютере. Выведите на экран текст каждого модуля, определите назначение и способ использования соответствующих процедур и функций.

3.2. Проверка и отладка программ

Программа является **правильной**, если:

а) процесс вычислений завершается за конечное время;

б) полученные результаты являются решением именно той задачи, для которой была написана программа.

В противном случае программа содержит **ошибки**.

Теоретически правильность программы можно проверить, запуская ее при всевозможных значениях входных данных. В большинстве практических случаев проверить правильность программы таким образом невозможно, так как множество значений входных данных практически бесконечно, а соответствующие решения неизвестны. Поэтому в информатике термин «проверка программы» имеет более узкий смысл.

Проверка программы – это этап разработки программ, целью которых является исправление ошибок. Проверка реализуется при выполнении программы для определенных входных данных, называемых **тестовыми данными**. Существуют два метода отбора тестовых данных:

– функциональная проверка или метод черного ящика;

– структурная проверка или метод прозрачного ящика.

Напомним, что термин *черный ящик* используется для системы, внутренняя структура которой неизвестна.

В случае **функционального тестирования** входные данные подбираются так, чтобы проверить выполнение каждой функции программы. Программа представляется черным ящиком, а ее правильное функционирование определяется вводом определенных данных и анализом полученных результатов. Отбор входных данных в большой степени зависит от умения и опыта того, кто осуществляет проверку. Обычно из множества входных данных отбираются **типичные** и **нетипичные значения**.

Пример. Рассмотрим программу `P143`. Текст самой программы не представлен, поскольку при функциональном тестировании он не нужен. Программа выполняет следующие функции:

– считывает с клавиатуры последовательность вещественных чисел;

– выводит на экран среднее арифметическое значение положительных чисел последовательности.

Очевидно, множество входных данных состоит из всевозможных последовательностей вещественных чисел.

Тестовые данные будут включать:
а) нетипичные значения:
– пустую последовательность;
– последовательность, которая не содержит положительных чисел;
– последовательность, которая содержит только одно положительное число;
б) типичные значения:
– последовательность с двумя положительными числами;
– последовательность с тремя и более положительными числами.

При **структурном тестировании** входные данные выбираются исходя из структуры программы: описание данных, процедуры и функции, простые операторы, составные операторы и т. д. Тестовые данные должны обеспечить:

а) выполнение каждого простого оператора (оператора присваивания, оператора вызова процедуры, оператора перехода **goto**);

б) выполнение каждого цикла **for** ноль, один и более раз;

в) выполнение каждого оператора **if**, **repeat**, **while** для значений `true`, `false` соответствующих булевых управляющих выражений;

г) выполнение каждого случая, описанного в операторе выбора **case**.

Пример. Рассмотрим текст программы, которая вычисляет среднее арифметическое значение положительных чисел некоторой последовательности:

```
Program P143;
  {Среднее арифметическое значение положительных чисел}
var n, k : integer;
    x, s : real;
begin
  n:=0;
  k:=0;
  s:=0;
  writeln('Введите последовательность вещественных чисел:');
 while not eof do
    begin
      readln(x);
      n:=n+1;
      if x>0 then
        begin
          k:=k+1;
          s:=s+x;
        end;
    end; { while }
 if n=0 then writeln('Пустая последовательность')
       else if k=0 then writeln('Последовательность не
                              содержит положительных чисел')
                 else writeln('Среднее=', s/k);
 readln;
end.
```

Входные данные должны обеспечить выполнение операторов **while** и **if** для значений true и false булевых выражений: **not eof**, x>0, n=0 и k=0. Следовательно, тестовые данные будут включать:

– пустую последовательность (**not eof** = false, n=0);

– непустую последовательность (**not eof** = true, n≠0);

– последовательность, которая содержит, по крайней мере, одно положительное число (*x*>0, *k*≠0);

– непустую последовательность, которая содержит только отрицательные числа (*x*≤0, *k*=0).

Метод прозрачного ящика можно использовать независимо или вместе с **методом черного ящика** для того, чтобы улучшить качество выполняемых проверок. Например, в случае программы P143 последовательность, содержащую, по крайней мере, одно положительное число, можно заменить на три последовательности, которые содержат соответственно одно, два, три или более положительных числа. Эти данные обеспечат выполнение операторов k:=k+1; s:=s+x и вычисление выражения s/k для типичных и нетипичных значений переменных k и s.

Применение структурной проверки связано со значительными трудностями из-за наличия условных операторов (**if**, **case**), операторов цикла (**for**, **while**, **repeat**) и операторов перехода (**goto**). Очевидно, существует огромное количество комбинаций, для которых необходимо выполнять все операторы присваивания и вызова процедур.

Отладка программы заключается в локализации (выявлении) тех зон программы, которые привели к появлению ошибок, установлении причин ошибок и их устранении. Отладка может проводиться статически (после выполнения программы) и динамически (в процессе выполнения программы).

При **статической отладке** причины ошибок устанавливаются путем анализа результатов программы и сообщений операционной системы. Для облегчения процесса отладки в программу временно вставляются операторы, которые выводят на экран промежуточные значения основных переменных.

При **динамической отладке** локализация ошибок осуществляется путем отслеживания процесса выполнения программы на уровне операторов. Существующие версии языка допускают для динамической отладки применение следующих операций:

– пошаговое выполнение программы;

– наблюдение за значениями выбранных выражений;

– создание и удаление временных точек останова программы;

– установка значений указанных переменных;

– отслеживание вызовов процедур и функций;

– исправление ошибок ввода-вывода, ошибок переполнения и т.д.

Подробное описание указанных средств можно найти в системе помощи *Turbo PASCAL's Online Help*.

Эффективность отладки зависит от способа написания и проверки программы, качества сообщений об ошибках, выдаваемых компилятором и типа ошибок. Как правило, после первой проверки, показавшей наличие ошибки, проводятся другие проверки, организованные таким образом, чтобы изолировать ошибку и получить информацию для ее исправления.

Проверка и отладка программы занимает больше половины времени, необходимого для разработки программного продукта. Сложность этих процессов снижается путем разбиения больших программ на подпрограммы или модули и применения методов структурного программирования.

Отметим, что проверка программ является эффективным средством **выявления ошибок**, а не доказательства их отсутствия. Несмотря на то, что проверка не доказывает правильность программы, она пока является единственным практическим методом сертификации программных продуктов. В настоящее время разрабатываются методы проверки, основанные на формальном доказательстве правильности программ, однако теоретические результаты, достигнутые в этом направлении, не применимы к сложным программам.

Вопросы и упражнения

❶ Когда программа на языке ПАСКАЛЬ является правильной? Как обеспечивается правильность программы?

❷ Как производится отбор входных данных по методу функционального тестирования?

❸ Осуществите функциональную проверку программы `P124` из параграфа 2.4. Программа выполняет следующие функции:
- создает односвязный список;
- выводит список на экран;
- вставляет в список заданный элемент;
- удаляет из списка элемент, заданный пользователем.

❹ Определите функции, которые выполняют следующие программы, и выполните функциональные проверки следующих программ:
а) `P117` и `P120` из параграфа 2.1;
б) `P122` и `P123` из параграфа 2.2;
в) `P127` из параграфа 2.5 и `P128` из параграфа 2.6.

❺ Как производится отбор входных данных по методу структурной проверки?

❻ Выполните структурную проверку следующих программ:
а) `P117` и `P120` из параграфа 2.1;
б) `P122` и `P123` из параграфа 2.2;
в) `P127` из параграфа 2.5.

❼ В чем разница между *статической* и *динамической отладкой*?

❽ Найдите в системе помощи *Turbo PASCAL's Online Help* описание средств для динамической отладки. Выполните отладку ваших программ.

3.3. Элементы структурного программирования

С первых лет компьютерной обработки данных стало ясно, что проверка, отладка и модификация программ требуют большого объема работы. Более того, сложные программы, содержащие сотни и тысячи операторов, становятся непонятными даже для их создателей.

Структурное программирование представляет собой определенный стиль, манеру написания программ в соответствии с определенными правилами, основанными на теореме о полноте. Согласно **теореме о полноте**, любой алгоритм можно представить в виде комбинации трех управляющих структур:

– блок (последовательность двух или более операторов присваивания и/или операторов вызова процедур);

– условие (**if**... **then**... или **if**... **then**... **else**...);

– цикл с предусловием (**while**... **do**...).

Структурное программирование допускает использование и других управляющих структур, таких как:

– выбор (**case**... **of**...);

– цикл с постусловием (**repeat**... **until**...);

– цикл со счетчиком (**for**... **do**...).

Основные правила структурного программирования:

1. Структура любой программы или подпрограммы должна представлять собой комбинацию только допустимых управляющих структур: блок, условие, выбор, цикл.

2. Структуры данных, используемых в программе, должны соответствовать специфике решаемой задачи.

3. Максимальная длина функции или процедуры 50–100 строк. Не рекомендуется использовать глобальные переменные.

4. Идентификаторы, используемые для обозначения констант, типов, переменных, функций, процедур и модулей программы, должны нести в себе информацию о соответствующих объектах.

5. Для того чтобы текст программы был ясным, необходимо вставлять комментарии и располагать строки в соответствии с логической и синтаксической структурой операторов.

6. Операции ввода-вывода необходимо выделять в отдельные подпрограммы. Правильность входных данных нужно проверять сразу после их считывания.

7. Вложенные операторы **if** необходимо заменять оператором **case**.

Программы, составленные согласно этим правилам, являются простыми, понятными, не содержат переходов и возвращений. Напомним, что согласно теореме о полноте, любую программу можно написать без использования оператора перехода **goto**. Однако некоторые авторы допускают использование этого оператора при условии, что оно будет сведено к минимуму, а переходы будут осуществляться только по ходу чтения текста.

Вопросы и упражнения

❶ Что такое структурное программирование?

❷ Укажите структуры управления, необходимые и достаточные для представления любого алгоритма.

❸ Сформулируйте основные правила структурного программирования.

❹ В чем преимущества структурного программирования?

❺ Удовлетворяют ли программы `P124`, `P130` и `P135` из главы 2 основным правилам структурного программирования?

❻ Программа `P144` выводит на экран всевозможные представления натурального числа *n* в виде суммы последовательных натуральных чисел.

```
Program P144;
var a,i,l,s,n : integer;
b : boolean;
begin
write('n='); readln(n);
b:=true;
for i:=1 to ((n+1) div 2) do
begin
a:=i;
s:=0;
while (s<n) do
begin
s:=s+a;
a:=a+1;
end;
if s=n then
begin
b:=false;
write(n, '=', i);
for l:=i+1 to (a-1) do write('+',l);
writeln;
end;
end;
if b then writeln('Представлений не существует');
readln;
end.
```

Например, для $n = 15$ получаем:

`15=1+2+3+4+5;`

`15=4+5+6;`

`15=7+8.`

Эти представления вычисляются с помощью метода перебора, рассматривая последовательности:

1, 2, 3, ..., *k*;

2, 3, ..., *k*;

3, ..., *k*

и т. д., где $k = (n+1)$ **div** 2. Члены 1, 2, 3, … *k* вычисляются с помощью рекуррентного соотношения $a = a+1$.

Расположите текст программы согласно логической и синтаксической структуре каждого оператора.

Глава 4

АНАЛИЗ АЛГОРИТМОВ

4.1. Сложность алгоритмов

Практическая ценность программ на языке ПАСКАЛЬ в значительной степени зависит от сложности используемых в них алгоритмов. Напомним, что алгоритм представляет собой конечную последовательность известных операций (инструкций, команд), которые выполняются в сторого определенном порядке, обеспечивая этим решение определенной задачи.

Известно, что один и тот же алгоритм может быть описан различными способами: логическими схемами, математическими формулами, текстом, записанном на привычном языке общения, и наконец, с помощью языков программирования. Очевидно, в данном учебнике изучаемые алгоритмы будут описываться при помощи средств языка ПАСКАЛЬ: операторов, функций, процедур и программ, которые могут быть выполнены на компьютере.

Сложность алгоритма характеризуется *объемом требуемой памяти и временем (продолжительностью, длительностью) его выполнения*. Методы оценивания этих показателей изучаются в специальном разделе информатики, который называется **анализ алгоритмов**. В рамках этого раздела используются следующие обозначения:

n – натуральное число, характеризующее размер (объем) данных на входе изучаемого алгоритма. В большинстве случаев n представляет собой количество элементов обрабатываемого множества, степень решаемого уравнения, количество элементов обрабатываемого массива и т.п.;

$V(n)$ – объем внутренней памяти, необходимой для хранения используемых алгоритмом данных;

$T(n)$ – время, необходимое для выполнения алгоритма. Обычно в нем не учитываются операции по вводу исходных данных и выводу результатов.

Очевидно, что объем памяти $V(n)$ и время выполнения $T(n)$ зависят, в первую очередь, от характеристики n входных данных. Указанная зависимость подчеркивается и тем фактом, что V и T представлены в виде функций от аргумента n.

Практическая применимость алгоритма возможна лишь тогда, когда расход памяти и требуемое время не превышают ограничений, накладываемых на них средой программирования и техническими характеристиками используемого компьютера.

Например, предположим, что для решения некоторой задачи существуют два различных алгоритма, обозначенных как A_1 и A_2. Объем памяти и время выполнения алгоритма A_1 составляют:

$$V_1(n) = 100n^2 + 4;$$
$$T_1(n) = n^3 \cdot 10^{-3},$$

а для алгоритма A_2:

$$V_2(n) = 10n + 12;$$
$$T_2(n) = 2^n \cdot 10^{-6}.$$

В приведенных формулах объем памяти измеряется в байтах, а время – в секундах.

Объем памяти и время выполнения алгоритмов A_1, A_2 для различных величин n представлены в *таблице 4.1*.

Таблица 4.1

Сложность алгоритмов A_1 и A_2

n	10	20	30	40	50
$V_1(n)$	9,77 Кбайт	39,06 Кбайт	87,89 Кбайт	156,25 Кбайт	244,14 Кбайт
$V_2(n)$	112 байтов	212 байтов	312 байтов	412 байтов	512 байтов
$T_1(n)$	1 секунда	8 секунд	9 секунд	16 секунд	25 секунд
$T_2(n)$	0,001 секунды	1,05 секунды	18 секунд	13 дней	36 лет

Из *таблицы 4.1* видим, что алгоритм A_2 становится практически неприменим для входных данных с характеристикой $n>30$. Для таких данных время выполнения алгоритма A_1 намного меньше, но объем требуемой памяти $V_1(n)$ может превысить ограничение, накладываемое средой программирования (64 *Кбайта* для статических переменных программ в Turbo PASCAL 7.0).

Определение объема памяти $V(n)$ и времени выполнения $T(n)$ представляет особый интерес на этапе разработки соответствующих алгоритмов и программ. Очевидно, что именно на этом этапе можно заранее исключить из рассмотрения те алгоритмы, которые потребуют слишком больших объемов памяти или неприемлемого времени выполнения.

Отметим, что постоянное увеличение внутренней памяти современных компьютеров направляет основное внимание информатиков на изучение времени выполнения алгоритмов или, другими словами, на их **временную сложность**.

Вопросы и упражнения

❶ Объясните термин *сложность алгоритма*. Назовите показатели, характеризующие сложность алгоритмов.

❷ Как Вы считаете, какие факторы влияют на сложность алгоритма?

❸ От чего зависит расход памяти и время выполнения алгоритма? Когда возможно практическое применение алгоритма?

❹ Алгоритмы A_1 и A_2 (смотри *таблицу 4.1*) будут выполняться в среде программирования Turbo PASCAL 7.0. Как Вы считаете, какой из алгоритмов нужно использовать в случае входных данных с характеристикой: a) $n = 10$; b) $n = 20$; c) $n = 30$? Для каких значений n алгоритм A_1 может быть использован в среде программирования Turbo PASCAL 7.0?

❺ Сложность некоторого алгоритма, обозначенного как A_3, описывается формулами

$$V_3(n) = 600n^3 + 18;$$
$$T_3(n) = 3^n \cdot 10^{-2}.$$

Как Вы считаете, для каких значений n алгоритм A_3 может выполняться в среде программирования Turbo PASCAL 7.0?

❻ Определите размер входных данных наиболее распространенных алгоритмов, разработанных Вами в процессе изучения языка программирования PASCAL.

4.2. Оценка объема памяти

Оценка объема памяти $V(n)$ может быть осуществлена путем суммирования количества байтов, занимаемых каждой из переменных программы. Количество байтов, занимаемых простой переменной каждого из типов – integer, real, boolean, char, *перечисление, интервального и ссылочного*, – зависит от конкретной реализации языка. Так, в среде программирования Turbo PASCAL 7.0 память для переменных выделяется в соответствии с *таблицей 4.2*.

Таблица 4.2

Выделение внутренней памяти в Turbo PASCAL 7.0

Тип переменной	*Число байтов*
integer	2
real	6
boolean	1
char	1
перечисление	1
интервальный	В соответствии с базовым типом
ссылочный	4
pointer	4

Для составных типов данных объем необходимой памяти для одной переменной вычисляется путем суммирования числа байтов, выделяемых для каждого элемента.

Например, объем памяти для переменных A, B, p и s из объявлений

```
var A : array[1..n, 1..n] of real;
    B : array[1..n] of integer;
    p : boolean;
    s : string[10];
```

составляет

$$V(n) = 6n^2 + 2n + 11 \text{ (байтов)}.$$

В общем случае объем памяти произвольной программы на ПАСКАЛЕ зависит не только от типа использованных переменных, но и от способа управления внутренней памятью компьютера. При выполнении программы на ПАСКАЛЕ внутренняя память компьютера делится на три части (*рис. 4.1*):

– ***сегмент данных***, который предназначен для размещения глобальных переменных. Указанные переменные объявляются в секции **`var`** программы на ПАСКАЛЕ;

– ***стек***, предназначенный для размещения фактических параметров, локальных переменных, значений, возвращаемых функциями и адресов возврата вызываемых подпрограмм ПАСКАЛЯ. Вызов каждой подпрограммы ведет к размещению соответствующих данных в стеке, а выход из подпрограммы – к их удалению. Подчеркнем, что для параметров-переменных в стеке запоминается только адрес переменной из вызывающей программы, а в случае параметров-значений в стек будет помещена копия данных из списка фактических параметров;

– ***куча*** (***heap***), предназначенная для размещения динамических переменных. Указанные переменные создаются при помощи процедуры `new` и удаляются из памяти при помощи процедуры `dispose`.

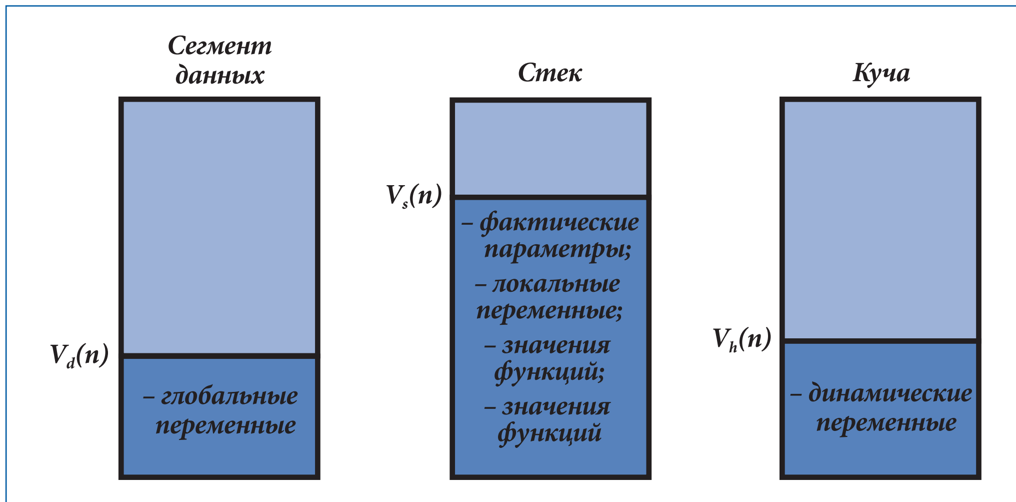


Рис. 4.1. Управление внутренней памятью

Таким образом, оценка объема памяти предполагает вычисление следующих характеристик программы (*рис. 4.1*):

$V_d(n)$ – объем памяти, занимаемый глобальными переменными в *сегменте данных*;

$V_s(n)$ – объем памяти, занимаемый фактическими параметрами и локальными переменными в *стеке*;

$V_h(n)$ – объем памяти, занимаемый динамическими переменными в *куче*.

Обычно в среде Turbo PASCAL 7.0 требуется, чтобы $V_d(n) \le 64$ *Кбайт*, $V_s(n) \le 16$ *Кбайт* и $V_h(n) \le 256$ *Кбайт*. Размеры *стека* и *кучи* могут быть изменены при помощи директив компилятора или через систему меню среды программирования.

Пример:

```
Program P145;
 {Управление внутренней памятью}
const n = 100;
type Matrice = array[1..n, 1..n] of real;
     Vector = array[1..n] of real;
var A : Matrice;
    i : integer;
 p, q : ^Matrice;

procedure Prelucrare(var B:Matrice);
var C : Vector;
begin
  {...обработка элементов матрицы B...}
end; { Prelucrare }

begin
  {...ввод матрицы A...}
 Prelucrare(A);
 new(p);
 new(q);
  {...обработка динамических переменных p^ и q^...}
 dispose(p);
 dispose(q);
  {...вывод результатов...}
 writeln('Конец');
 readln;
end.
```

Глобальные переменные `A`, `i`, `p` и `q` программы `P145` будут размещены в сегменте данных (*рис. 4.2*). Объем памяти для этих переменных:

$$V_d(n) = 6n^2 + 2 + 2 \cdot 4 = 6n^2 + 10.$$

Вызов процедуры `Pr(A)` приведет к размещению в стеке адреса матрицы `A`, адреса возврата в главную программу и локальной переменной `C`.

Объем памяти, необходимой для этих данных:

$$V_s(n) = 6n + 8.$$

После выхода из процедуры соответствующие данные будут удалены из стека.

Операторы `new(p)` и `new(q)` создают в *куче* динамические переменные `p^` и `q^` типа `Matrice`. Объем памяти для этих переменных::

$$V_h(n) = 6n^2 + 6n^2 = 12n^2 .$$

После выполнения операторов `dispose(p)` и `dispose(q)` динамические переменные в *куче* будут уничтожены, а занятое ими место освободится.

Вопросы и упражнения

❶ Пользуясь системой помощи среды программирования, в которой Вы работаете, определите объем памяти, занимаемый простыми переменными.

❷ Как вычисляется объем внутренней памяти, необходимой для хранения данных, используемых в определенном алгоритме?

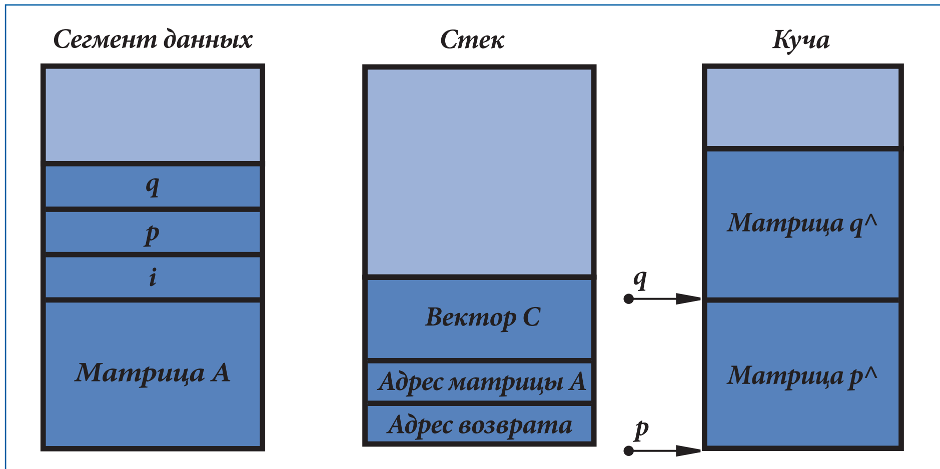


Рис. 4.2. Управление памятью в программе `P145`

❸ Объясните, как происходит управление внутренней памятью в случае программ на ПАСКАЛЕ.

❹ Пользуясь системой помощи среды программирования, определите размеры памяти сегмента данных, *стека* и *кучи*. Каким образом могут быть изменены размеры *стека* и *кучи*?

❺ Вычислите объем памяти, необходимый для переменных из приведенных ниже объявлений:

a)

```
var A : array[1..n, 1..n] of integer;
    B : string;
    C : array [1..n, 1..n, 1..n] of boolean;
```

b)

```
type Vector  = array[1..n] of real;
     Matrice = array[1..n] of Vector;
var A, B, C : Matrice;
    D : Vector;
```

c)

```
type Elev = record
              Nume : string;
              Prenume : string;
              NotaMedie : real
           end;
    ListaElevi = array[1..n] of Elev;
var A, B, C : ListaElevi;
```

d)
```
type Angajat = record
                 NumePrenume : string;
                 ZileLucrate : 1..31;
                 PlataPeZi : real;
                 PlataPeLuna : real
               end;
    ListaDePlata = array[1..n] of Angajat;
var L1, L2 : ListaDePlata;
```

e)
```
type Elev = record
               Nume : string;
               Prenume : string;
               NotaMedie : real
            end;
     FisierElevi = file of Elev;
var FE : FisierElevi;
    E : Elev;
    str : string;
    i, n : integer;
```

❻ Вычислите объем памяти, необходимый для приведенной ниже программы. Скомпилируйте эту программу для значений 50, 60, 70, 80 и 100 константы *n*. Объясните выводимые на экран сообщения.

```
Program P146;
   { Размеры сегмента данных }
const n = 50;
type Matrice = array[1..n, 1..n] of real;
var A, B : Matrice;
begin
  {...ввод данных...}
  {...обработка матриц A и B...}
 writeln('Конец');
 readln;
end.
```

❼ Рассмотрим следующую программу:

```
Program P147;
 { Размеры стека }
var n : integer;

function S(n:integer):real;
begin
 if n=0 then S:=0
        else S:=S(n-1)+n;
end; { S }
```

```
begin
 write('n='); readln(n);
 writeln('s=', S(n));
 readln;
end.
```

В приведенной программе сумма:

$$S(n) = 0 + 1 + 2 + ... + n$$

вычисляется при помощи рекурсивной функции:

$$S(n) = \begin{cases} 0, & \text{если } n = 0; \\ S(n-1) + n, & \text{если } n > 0. \end{cases}$$

Оцените объем памяти, необходимой программе `P147`. Определите максимальное значение n, при котором программа `P147` выполняется без ошибок.

❽ Определите объем памяти, необходимой для приведенной ниже программы. При каких значениях n программа будет выполняться без ошибок?

```
Program P148;
 { Размеры кучи (heap) }
type Vector = array[1..100] of real;
var p : ^Vector;
    i, n : integer;
begin
 write('n='); readln(n);
 for i:=1 to n do new(p);
 writeln('Конец');
 readln;
end.
```

4.3. Измерение времени выполнения алгоритма

Для уже разработанных программ время их выполнения $T(n)$ можно измерить непосредственно. С этой целью будем использовать модуль `U7`:

```
Unit U7;
 { Измерение времениi }
interface
function TimpulCurent : real;
implementation
uses Dos;
var ore : word;
    minute : word;
    secunde : word;
    sutimi : word;
```

```
function TimpulCurent;
begin
GetTime(ore, minute, secunde, sutimi);
TimpulCurent:=3600.0*ore+60.0*minute+
              1.0*secunde+0.01*sutimi;
end; { TimpulCurent }
end.
```

Программный модуль U7 дает программисту возможность использовать предопределенную функцию TimpulCurent, которая возвращает значение вещественного типа – текущее время, выраженное в секундах. Показания системных часов компьютера (часы, минуты, секунды и сотые доли секунды) считываются при помощи процедуры GetTime, содержащейся в модуле DOS среды программирования Turbo PASCAL 7.0.

В качестве примера приведем программу P149, в которой измеряется время выполнения процедуры Sortare:

```
Program P149;
 { Время выполнения процедуры Sortare }
uses U7;
type Vector = array[1..10000] of real;
var   A   : Vector;
    i, n : integer;
  T1, T2 : real; { время в секундах }

procedure Sortare(var A:Vector; n:integer);
  { Сортировка элементов массива A }
var i, j : integer;
       r : real;
begin
  for i:=1 to n do
     for j:=1 to n-1 do
        if A[j]>A[j+1] then
           begin
             r:=A[j];
             A[j]:=A[j+1];
             A[j+1]:=r;
           end;
end; { Sortare }

begin
 write('Введите число элементов n=');
 readln(n);
 { массиву A присваивается значение (n, n-1, ..., 3, 2, 1) }
 for i:=1 to n do A[i]:=n-i+1;
 T1:=TimpulCurent;
```

```
  Sortare(A, n);
  T2:=TimpulCurent;
  writeln('Длительность выполнения', (T2-T1):7:2, '  сек');
  readln;
end.
```

Процедура `Sortare` упорядочивает элементы массива `A` методом пузырька. В этом методе массив `A` просматривается n раз, причем при каждом проходе выполняется n–1 сравнение соседних элементов `A[j]` и `A[j+1]`. Если `A[j]>A[j+1]`, то такие элементы обмениваются местами.

Для того чтобы избежать ввода с клавиатуры большого количества данных, в программе `P149` массиву `A` присваивается начальное значение:

$$A = (n, n\text{-}1, n\text{-}2, ..., 3, 2, 1).$$

Например, для n=4 исходный массив имеет вид:

$$A=(4, 3, 2, 1).$$

В процессе сортировки получаем:

$$i = 1, \quad A = (3, 2, 1, 4);$$
$$i = 2, \quad A = (2, 1, 3, 4);$$
$$i = 3, \quad A = (1, 2, 3, 4);$$
$$i = 4, \quad A = (1, 2, 3, 4).$$

Время выполнения процедуры `Sortare` для компьютера *Pentium* с тактовой частотой 500 *MHz* приведено в *таблице 4.3*, а соответствующий график – на *рис. 4.3*.

Таблица 4.3

Время выполнения процедуры `Sortare`

n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
T(n), s	0,27	1,10	2,47	4,50	7,03	10,16	13,84	18,02	22,85	28,18

Вопросы и упражнения

❶ Как Вы считаете, какая связь существует между временем, необходимым для выполнения программы на ПАСКАЛЕ, тактовой частотой и производительностью компьютера?

❷ Измерьте время выполнения процедуры `Sortare` (смотри программу `P149`) для компьютера, на котором Вы работаете. Постройте график, аналогичный представленному на *рис. 4.3*.

❸ Представьте графически на одном рисунке время выполнения приведенных ниже процедур.

a)

```
procedure N2(n : integer);
var i, j, k : integer;
          r : real;
```

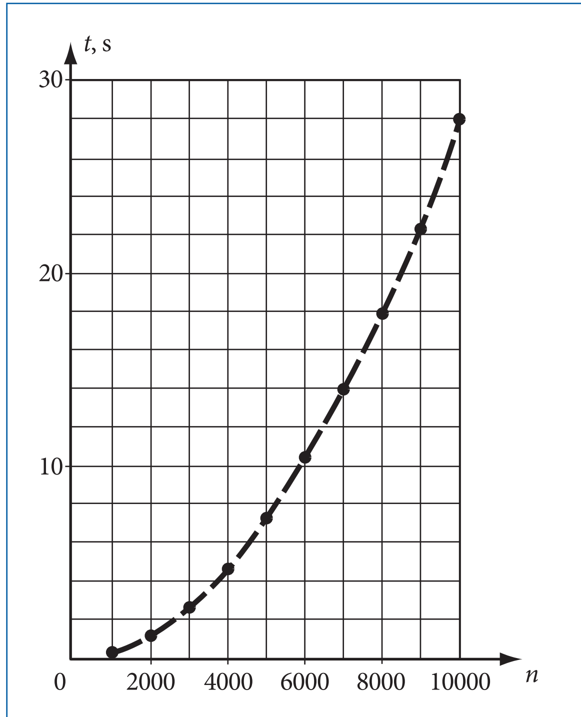


Рис. 4.3. Время выполнения процедуры `Sortare`

```
begin
   for i:=1 to n do
     for j:=1 to n do
        for k:=1 to 300 do
           r:=1.0;
end; { N2 }
```

b)

```
procedure N3(n : integer);
var i, j, k : integer;
          r : real;
begin
   for i:=1 to n do
     for j:=1 to n do
        for k:=1 to n do
           r:=1.0;
end; { N3 }
```

c)

```
procedure N4(n : integer);
var i, j, k, m : integer;
             r : real;
begin
   for i:=1 to n do
```

```
      for j:=1 to n do
         for k:=1 to n do
            for m:=1 to n do
               r:=1.0;
end; { N4 }
```

Для примера, на *рис. 4.4* представлены соответствующие графики для компьютера *Pentium*, с тактовой частотой 500 *MHz*.

❹ Какова точность измерения времени с помощью функции `TimpulCurent`? Обоснуйте Ваш ответ.

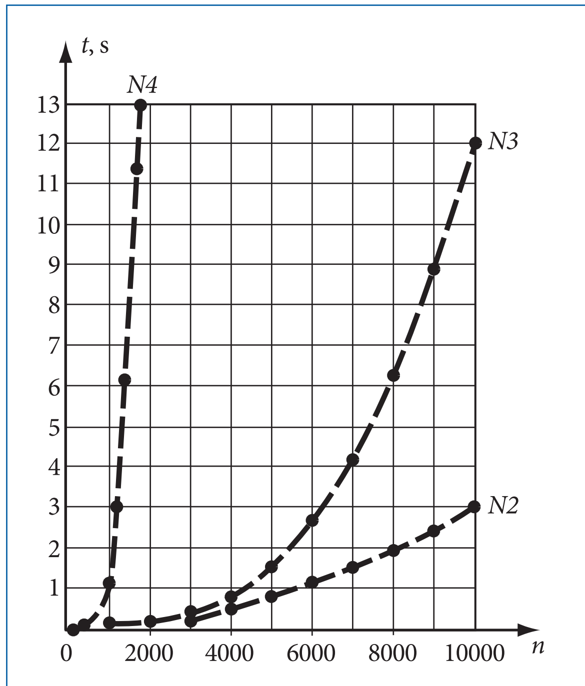


Рис.4.4. Время выполнения процедур `N2`, `N3` и `N4`

4.4. Оценка времени выполнения алгоритма

При практической реализации любого алгоритма появляется необходимость в предварительной оценке времени его выполнения *T(n)* до окончательного тестирования и отладки самой программы. К сожалению, теоретически очень тяжело найти точное выражение для *T(n)*. По этой причине обычно ищут верхний предел времени, требуемого для выполнения алгоритма, анализируя только самые неблагоприятные случаи.

Предположим, в учебных целях, что на выполнение любой из операций на языке ПАСКАЛЬ (`+`, `-`, **`or`**, `*`, `/`, **`div`**, **`and`**, `<`, `<=`, **`not`** и др.) затрачива-

ется не более Δ единиц времени. Такое же время Δ затрачивается на индексирование элементов произвольного массива [ ], на присваивание := и выполнение оператора **goto**. Конкретное значение величины Δ зависит от среды программирования, производительности используемого компьютера и составляет порядка 10^{-9} ... 10^{-7} секунды. В дальнейшем будем оценивать время $T(n)$ по формуле

$$T(n) = Q(n) \cdot \Delta,$$

где $Q(n)$ – количество элементарных операций – сложения, вычитания, умножения, сравнения и т.д. – необходимых для решения некоторой задачи.

Допустим, что в выражении E встречается m операций на ПАСКАЛЕ и k вызовов функции F. Очевидно, что общее число Q_E элементарных операций, необходимых для вычисления выражения E, определяется как

$$Q_E = m + k\, Q_F\,,$$

где Q_F – количество элементарных операций, необходимых для вычисления функции F.

Примеры:

	Выражение E	*Число элементарных операций* Q_E
a)	`a*b+c`	2
b)	`(a<b)`**`or`**`(c>d)`	3
c)	`sin(x)+cos(y)`	$1 + Q_{\sin} + Q_{\cos}$
d)	`a+M[i]`	2
e)	`sin(x+y)+sin(x-y)`	$3 + 2Q_{\sin}$

Число элементарных операций Q_I , необходимых для выполнения некоторого оператора I языка ПАСКАЛЬ, оценивается по формулам *таблицы 4.4*.

Таблица 4.4

Число элементарных операций, необходимых для выполнения некоторых команд на языке ПАСКАЛЬ

№	***Оператор***	***Число элементарных операций***
1	**2**	**3**
1	Присваивание $\upsilon := E$	$Q_E + 1$
2	Вызов процедуры P	$Q_P + 1$
3	Выбор **if** E **then** I_1 **else** I_2	$Q_E + \max\{Q_{I_1}, Q_{I_2}\} + 1$
4	Множественный выбор **case** E **of** $I_1; I_2; ...; I_k$ **end**	$Q_E + \max\{Q_{I_1}, Q_{I_2}, ..., Q_{I_k}\} + k + 1$
5	Цикл со счетчиком **for** $v := E_1$ **to/downto** E_2 **do** I	$Q_{E_1} + Q_{E_2} + mQ_I + m + 1$
6	Цикл с предусловием **while** E **do** I	$(m + 1)Q_E + mQ_I + 1$

1	2	3
7	Цикл с постусловием **repeat** *I* **until** *E*	$mQ_I + mQ_E + 1$
8	Составной оператор **begin** I_1; I_2; ...; I_k **end**	$Q_{I_1} + Q_{I_2} + ... + Q_{I_k} + 1$
9	Оператор **with** v **do** *I*	$Q_I + 1$
10	Переход **goto**	1

Формулы из *таблицы 4.4* могут быть выведены исходя из способа выполнения каждого оператора ПАСКАЛЬ. В *таблице 4.4* используются следующие обозначения:

v – переменная или имя функции;

E – выражение;

I – оператор языка ПАСКАЛЬ.

Количество повторений оператора *I* внутри цикла **for**, **while** или **repeat** обозначено через *m*. Подчеркнем, что циклы в ПАСКАЛЕ могут быть организованы и при помощи операторов **if** и **goto**, хотя такое применение указанных операторов противоречит правилам структурного программирования.

Для примера оценим количество элементарных операций *Q(n)*, необходимых для упорядочивания элементов массива методом пузырьковой сортировки:

```
procedure Sortare(var A:Vector; n:integer);
var i, j : integer;
       r : real;
{1}  begin
{2}  for i:=1 to n do
{3}     for j:=1 to n-1 do
{4}        if A[j]>A[j+1] then
{5}            begin
{6}              r:=A[j];
{7}              A[j]:=A[j+1];
{8}              A[j+1]:=r;
               end;
end; { Sortare }
```

Операторы I_1, I_2, ... , I_8 процедуры `Sortare` помечены при помощи комментариев `{1}`, `{2}`, ..., `{8}` в левой части строк программы. Через Q_j будем обозначать количество элементарных операций, необходимых для выполнения оператора I_j:

$$Q_6 = 2;$$
$$Q_7 = 4;$$
$$Q_8 = 3;$$
$$Q_5 = Q_6 + Q_7 + Q_8 + 1 = 10;$$
$$Q_4 = 4 + Q_5 + 1 = 15;$$
$$Q_3 = 0 + 1 + (n–1)Q_4 + (n–1) + 1 = 16n – 14;$$

$$Q_2 = 0 + 0 + nQ_3 + n + 1 = 16n^2 - 13n + 1;$$
$$Q_1 = Q_2 + 1 = 16n^2 - 13n + 2.$$

Следовательно, количество элементарных операций

$$Q(n) = 16n^2 - 13n + 2,$$

а время выполнения процедуры `Sortare`

$$T(n) = (16n^2 - 13n + 2)\Delta .$$

Из рассмотренного выше примера замечаем, что порядок прохода (выполнения) операторов диктуется структурой самой программы. Очевидно, что сначала анализируются простые операторы, а затем – составные. Для вложенных операторов в первую очередь анализируются внутренние, а затем – внешние операторы.

Аналитические выражения для $T(n)$, полученные в результате анализа программ на ПАСКАЛЕ, могут быть использованы для экспериментального нахождения времени Δ, нужного для выполнения одной элементарной операции.

Например, для процедуры `Sortare` (смотри *таблицу 4.3*) $n = 10000$ и $T(n) = 28{,}18$ с. Из уравнения

$$(16n^2 - 13n + 2)\Delta = 28{,}18$$

получаем $\Delta \approx 1{,}8 \cdot 10^{-8}$ с.

Очевидно, что найденная величина применима только для среды программирования Turbo PASCAL 7.0 и компьютера *Pentium* с тактовой частотой 500 *MHz*, использовавшегося в ходе измерения времени выполнения процедуры `Sortare`.

Например, в случае компьютера *Pentium* с тактовой частотой 150 *MHz* получается величина $\Delta \approx 6{,}0 \cdot 10^{-8}$ с.

Вопросы и упражнения

❶ Определите с помощью программы `P149` значение Δ для компьютера и среды программирования, с которыми Вы работаете.

❷ Определите количество элементарных операций Q_I, необходимых для выполнения следующих операторов ПАСКАЛЬ программы:

a)
```
x:=2*a-6*(y+z);
```

b)
```
p:=not(a=b)and(c>d);
```

c)
```
p:=(a in R)and(b in P);
```

d)
```
if a>b then x:=0 else x:=a+b;
```

e)
```
case i of
   1: x:=0;
   2: x:=a+b;
   3: x:=a+b+c;
end;
```

f)
```
for i:=1 to n do A[i]:=2*A[i];
```

g)
```
for i:=1 to n do A[i]:=B[i+1]-C[i-2];
```

h)
```
i:=0; while i<n do begin i:=i+1 end;
```

i)
```
i:=n; repeat i:=i-1 until i=0;
```

j)
```
begin i:=0; s:=0; r:=0 end;
```

k)
```
with A do begin x:=0; y:=0 end.
```

❸ Оцените количество элементарных операций *Q(n)* в приведенных ниже процедурах:

a)
```
procedure N2(n : integer);
var i, j, k : integer;
          r : real;
begin
   for i:=1 to n do
     for j:=1 to n do
        for k:=1 to 300 do
           r:=1.0;
end; { N2 }
```

b)
```
procedure N3(n : integer);
var i, j, k : integer;
          r : real;
begin
   for i:=1 to n do
     for j:=1 to n do
        for k:=1 to n do
           r:=1.0;
end; { N3 }
```

c)
```
procedure N4(n : integer);
var i, j, k, m : integer;
             r : real;
begin
   for i:=1 to n do
     for j:=1 to n do
        for k:=1 to n do
           for m:=1 to n do
              r:=1.0;
end; { N4 }
```

❹ Для процедуры `Sortare` компьютера *Pentium* с тактовой частотой $f_1 = 500$ *MHz* получено $\Delta_1 \approx 1{,}8 \cdot 10^{-8}$ с. Для такого же типа компьютера, но с тактовой частотой $f_2 = 150$ MHz получено $\Delta_2 \approx 6{,}0 \cdot 10^{-8}$ с. Сопоставляя полученные данные, замечаем, что

$$\frac{f_1}{f_2} = \frac{500 \cdot 10^6}{150 \cdot 10^6} \approx 3{,}3 \quad \text{и} \quad \frac{\Delta_2}{\Delta_1} = \frac{6{,}0 \cdot 10^{-8}}{1{,}8 \cdot 10^{-8}} \approx 3{,}3.$$

Как Вы считаете, чем объясняется этот факт?

❺ В процессе компиляции программы на языке ПАСКАЛЬ каждый оператор транслируется в одну и более команд языка машинных кодов. Количество получаемых в результате компиляции команд зависит от конкретной реализации среды программирования и типа используемого компьютера. Придумайте план эксперимента, который позволил бы оценить количество команд машинного кода, в которые транслируется каждый оператор ПАСКАЛЯ.

❻ Известно, что время выполнения команд языка машинного кода зависит от их типа. Например, команда суммирования двух целых чисел выполняется быстрее, чем команда суммирования двух вещественных чисел. Вследствие этого, значения Δ, найденные в результате измерения времени выполнения программы на ПАСКАЛЕ, зависят от типа использованных данных. Проверьте экспериментально это утверждение для компьютера, с которым Вы работаете.

❼ Производительность современных компьютеров измеряется в специальных единицах *Mips* – Мега (миллионов) команд в секунду. Например, персональные компьютеры обладают производительностью 500–800 *Mips*. Для измерения указанной характеристики изготовители компьютеров используют команды языка машинных кодов. Очевидно, что для программирующих на ПАСКАЛЕ представляет интерес производительность, выраженная в количестве ПАСКАЛЬ-операторов в секунду. Разработайте план эксперимента, который позволил бы оценить эту характеристику для компьютера, с которым Вы работаете.

4.5. Временная сложность алгоритмов

В информатике временная сложность характеризуется временем выполнения $T(n)$ или количеством элементарных операций $Q(n)$. Поскольку современные компьютеры обладают очень большой скоростью вычислений – порядка $10^8 \dots 10^{10}$ команд в секунду, то проблема времени выполнения возникает только при больших значениях n. Следовательно, в формулах, выражающих число элементарных операций $Q(n)$, представляет интерес только **доминирующий член**, т.е. тот, который быстрее остальных стремится к бесконечности с ростом n. Преобладание доминирующего члена над остальными демонстрируется в *таблице 4.5*.

Таблица 4.5

Значения доминирующих членов

n	$\log_2 n$	n^2	n^3	n^4	2^n
2	1	4	8	16	4
4	2	16	64	256	16
8	3	64	512	4096	256
16	4	256	4096	65536	65536
32	5	1024	32768	1048576	4294967296

Например, количество элементарных операций процедуры `Sortare` выражается формулой:

$$Q(n) = 16n^2 - 13n + 2.$$

Доминирующий член в этом выражении $16n^2$. Очевидно, что для больших значений n количество элементарных операций:

$$Q(n) \approx 16n^2,$$

а время выполнения:

$$T(n) \approx 16n^2\Delta.$$

В зависимости от временной сложности, алгоритмы классифицируются на:
– полиномиальные алгоритмы;
– экспоненциальные алгоритмы;
– полиномиально-недетерминированные алгоритмы.

Алгоритм называется **полиномиальным**, если доминирующий член имеет вид Cn^k, то есть

$$Q(n) \approx Cn^k; \quad T(n) \approx Cn^k\Delta,$$

где n – это характеристика входных данных, C – положительная константа, а k – натуральное число.

Временная сложность полиномиальных алгоритмов отражена в обозначении $O(n^k)$, которое читается как “алгоритм со временем выполнения порядка n^k”, или короче, “**алгоритм порядка n^k**”. Очевидно, что существуют полиномиальные алгоритмы порядка n, n^2, n^3 и т.д.

Например, упорядочивание элементов произвольного массива методом пузырьковой сортировки представляет собой полиномиальный алгоритм порядка n^2. Это видно из графика для времени выполнения $T(n)$ процедуры `Sortare`, представленного на *рис. 4.3*.

Алгоритм называется **экспоненциальным**, если доминирующий член имеет вид Ck^n, то есть

$$Q(n) \approx Ck^n; \quad T(n) \approx Ck^n\Delta,$$

где $k>1$. Временная сложность экспоненциальных алгоритмов отражена в обозначении $O(k^n)$.

Отметим, что экспоненциальными считаются также алгоритмы сложности $n^{\log n}$, хотя эта функция и не является экспоненциальной в строго математическом смысле.

Полиномиально-недетерминированные алгоритмы изучаются в углубленных курсах информатики.

Из сравнения скоростей роста экспоненциальной и полиномиальной функций (смотрите *таблицу 4.5*) следует, что экспоненциальные алгоритмы становятся неприменимыми даже при небольших значениях n. Для примера обратимся к *таблице 4.1*, в которой представлено время выполнения $T_1(n)$ полиномиального алгоритма порядка $O(n^3)$ и время выполнения $T_2(n)$ экспоненциального алгоритма порядка $O(2^n)$.

В зависимости от временной сложности считается, что задача **легкоразрешима**, если для ее решения существует полиномиальный алгоритм. Задача, для решения которой не существует полиномиального алгоритма, называется

трудноразрешимой. Подчеркнем, что указанная классификация относится только к асимптотическому анализу алгоритмов, то есть касается лишь больших значений n. Для малых n ситуация может сильно отличаться.

Например, предположим, что для решения некоторой задачи существуют два алгоритма, первый из которых полиномиальный со временем выполнения $T_1(n)$, а второй – экспоненциальный со временем выполнения $T_2(n)$:

$$T_1(n) = 1000n^2\Delta\ ;$$

$$T_2(n) = 2^n\Delta.$$

Непосредственными вычислениями можно проверить, что для $n = 1, 2, 3, ..., 18$ время $T_2(n) < T_1(n)$. Следовательно, в случае $n \le 18$ для применения предпочтителен экспоненциальный алгоритм.

На практике разработка компьютерных программ предполагает выполнение следующих этапов:

– точное формулирование задачи;

– определение временной сложности задачи – является ли она легко- или трудноразрешимой;

– разработку соответствующего алгоритма и реализацию его на определенной вычислительной системе.

Очевидно, что в случае легкоразрешимых задач программист должен направить все усилия на разработку полиномиального алгоритма т.е. алгоритма порядка n^k, стремясь к тому, чтобы параметр k принял как можно меньшее значение. В случае трудноразрешимых задач приоритет отдается тем алгоритмам, которые минимизируют время выполнения хотя бы для наиболее часто встречающихся на практике размеров входных данных. Методы классификации задач на легко- и трудноразрешимые изучаются в углубленных курсах информатики.

Вопросы и упражнения

❶ Укажите доминирующий член в следующих выражениях:

a) $12n + 5$;

b) $6n^2 + 100n + 18$;

c) $15n^3 + 1000n^2 - 25n + 6000$;

d) $2000n^3 + 2^n + 13$;

e) $n^{\log_2 n} + n^5 + 300n^2 + 6$;

f) $3^n + 2^n + 14n^3 + 21$;

g) $n^5 + 10n^4 + 200n^3 + 300n^2 + 1000n$.

❷ Как классифицируются алгоритмы в зависимости от их временной сложности?

❸ Определите тип алгоритма по его временной сложности:

a) $Q(n) = 200n + 15$;

b) $Q(n) = 2^n + 25n^2 + 1000$;

c) $Q(n) = n^3 + 3^n + 6000n^2 + 10^6$;

d) $Q(n) = 3^n + 2^n + n^{10} + 4000$

e) $Q(n) = n^{\log_2 n} + n^3 + n^2 + 1500$;

f) $Q(n) = 100n^2 + 15n^3 + 8^n + 900$.

❹ Как Вы считаете, чем объясняется важность доминирующего члена при анализе временной сложности алгоритмов?

❺ Рассмотрим процедуры `N2`, `N3` и `N4` из предыдущего параграфа. В результате оценки количества элементарных операций получаем:

$$Q_{N2}(n) = 602n^2 + 2n + 2;$$
$$Q_{N3}(n) = 2n^3 + 2n^2 + 2n + 2;$$
$$Q_{N4}(n) = 2n^4 + 2n^3 + 2n^2 + 2n + 2.$$

Определите порядок временной сложности алгоритмов, описанных этими процедурами. Сравните скорость роста времен выполнения $T_{N2}(n)$, $T_{N3}(n)$ и $T_{N4}(n)$, используя с этой целью графики на *рис. 4.4*.

❻ Рассмотрим алгоритм, состоящий из *k* вложенных циклов:

```
for i1:=1 to n do
   for i2:=1 to n do
      ...
          for ik:=1 to n do P
```

Количество элементарных операций Q_P , выполняемых в процедуре P, – постоянная величина. Оцените временную сложность алгоритма.

❼ Придумайте алгоритм решения следующей задачи:
Дано множество *A*, состоящее из *n* целых чисел. Определите, существует ли хотя бы одно подмножество *B*, $B \subseteq A$, сумма элементов которого равна *m*. Например, для $A=\{-3, 1, 5, 9\}$ и $m=7$, такое подмножество существует, а именно $B=\{-3, 1, 9\}$.
Оцените временную сложность разработанного алгоритма.

Глава 5

МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ

5.1. Итерация или рекурсия?

Развитие информатики помогло установить, что многие практически важные задачи могут быть решены с помощью определенного набора стандартных приемов, называемых **методами программирования:** рекурсии, перебора, метода перебора с возвратом, эвристических методов и других.

Одним из самых распространенных методов программирования является **рекурсия**. Напомним, что рекурсия определяется как ситуация, когда подпрограмма обращается к самой себе либо непосредственно, либо посредством другой подпрограммы. Рассматриваемые методы, изученные в рамках темы „Функции и процедуры", называются соответственно **прямой рекурсией** и **косвенной рекурсией**.

В общем случае, разработка любой рекурсивной программы возможна только тогда, когда соблюдается следующее **правило сходимости:** решение задачи должно вычисляться непосредственно либо вычисляться при помощи некоторых непосредственно вычисляемых значений. Другими словами, правильное определение любого рекурсивного алгоритма предполагает, что в процессе осуществления вычислений должны существовать:

–элементарные случаи, которые вычисляются непосредственно;

–случаи, которые хотя и не могут быть решены явно, однако процесс вычислений обязательно сводится к элементарному случаю.

Например, в рекурсивном определении функции факториала $fact$: $\mathbf{N} \rightarrow \mathbf{N}$,

$$fact(n) = \begin{cases} 1, \text{ если } n = 0; \\ n \cdot fact(n-1), \text{ если } n > 0, \end{cases}$$

различаем:

– элементарный случай $n = 0$. В этом случае значение $fact(0)$ вычисляется непосредственно, а именно $fact(0) = 1$;

– неэлементарные случаи $n > 0$. В таких случаях значения $fact(n)$ не могут быть вычислены непосредственно, но процесс вычислений сводится к элементарному случаю $fact(0)$.

Например, для $n = 3$ получаем:

$$fact(3) = 3 \cdot fact(2) = 3 \cdot 2 \cdot fact(1) = 3 \cdot 2 \cdot 1 \cdot fact(0) = 3 \cdot 2 \cdot 1 \cdot 1 = 6.$$

Следовательно, рекурсивное определение функции $fact(n)$ является **сходящимся определением**. Напомним, что функция $fact(n)$ может быть представлена на языке PASCAL, в соответствии с рекурсивным определением в форме:

```
function Fact(n:Natural):Natural;
begin
  if n=0 then Fact:=1
  else Fact:=n*Fact(n-1)
end;
```

Способ управления стеком для случая вызова функции `Fact(3)` представлен на *рис. 5.1*.

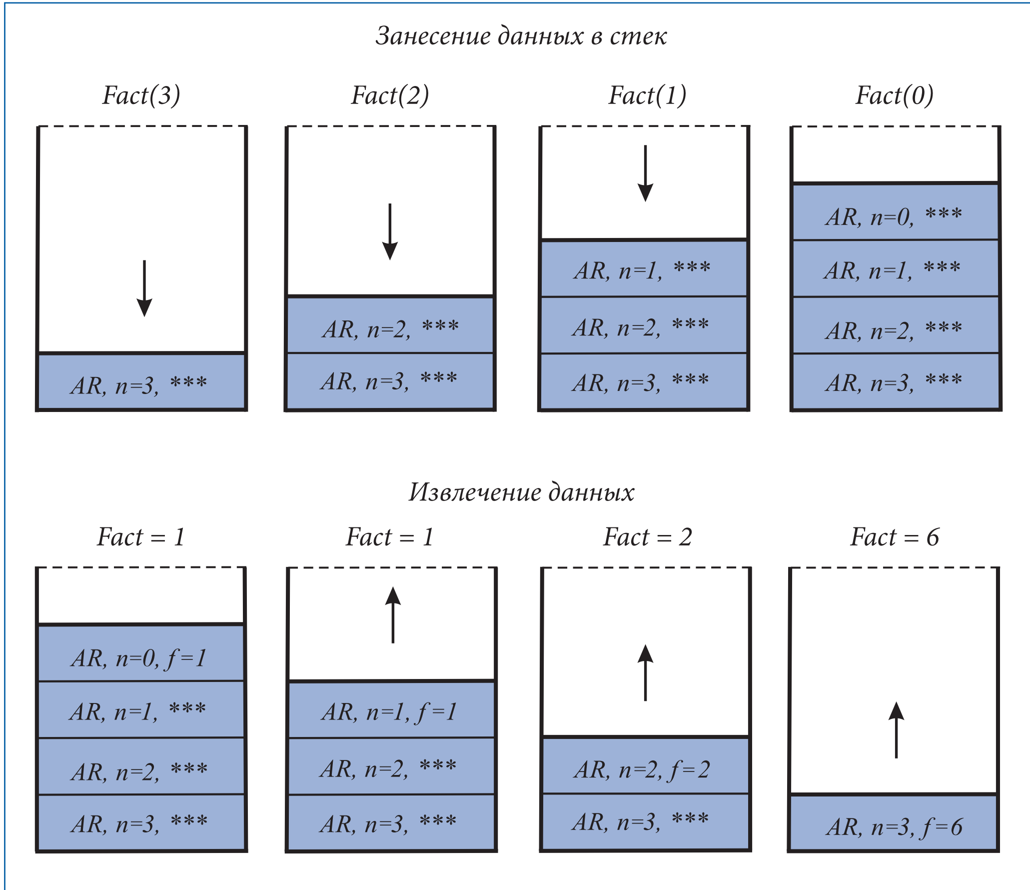


Рис. 5.1. Управление стеком в случае вызова функции `Fact(3)`:
AR – адрес возврата; *n* – текущее значение фактического параметра;
*** – место для запоминания значений *f*, возвращаемых функцией `Fact`

Аналогично в рекурсивном определении функции *incons*: **N**→**N**,

$$incons(n) = \begin{cases} 1, & \text{если } n = 0; \\ n \cdot incons(n+1), & \text{если } n > 0, \end{cases}$$

можно выделить элементарный случай $n = 0$ и неэлементарные случаи $n>0$. Но в отличие от функции $fact(n)$, для $n > 0$ значения $incons(n)$ не могут быть вычислены, поскольку процесс вычисления ведет к $incons(\infty)$.

Например, для $n = 3$ получаем:

$$incons(3) = 3 \cdot incons(4) = 3 \cdot 4 \cdot incons(5) = 3 \cdot 4 \cdot 5 \cdot incons(6) = \cdots.$$

Следовательно, приведенное выше рекурсивное определение функции $incons(n)$ является **расходящимся определением** и теоретически процесс вычислений будет продолжаться бесконечно. На практике же вычисления будут остановлены операционной системой в момент переполнения памяти, выделенной для стека, или превышения разрядности арифметического устройства.

Подчеркнем тот факт, что современные среды для разработки программ не выполняют автоматическую проверку сходимости рекурсивных алгоритмов, оставляя этот контроль программисту.

Как было показано в предыдущих главах, любой рекурсивный алгоритм может быть преобразован в итеративный и обратно. Выбор метода программирования – **итеративного** или **рекурсивного** – зависит от желания программиста. Очевидно, что при выборе должны учитываться преимущества и недостатки каждого метода, которые варьируются от случая к случаю. Для примера, в *таблице 5.1* представлены результаты сравнения обоих методов программирования для решения задач, связанных с компьютерной обработкой текстов.

Таблица 5.1

Сравнение итерации и рекурсии (компьютерная обработка текстов)

№	*Характеристики*	*Итерация*	*Рекурсия*
1.	Расход памяти	Малый	Большой
2.	Время выполнения	одинаковое	
3.	Структура программы	Сложная	Простая
4.	Трудоемкость написания программы	Большая	Малая
5.	Тестирование и отладка программы	Простые	Сложные

Предлагаем читателю в качестве упражнения сопоставить итеративный и рекурсивный подходы при разработке алгоритмов, предназначенных для создания и обработки динамических структур данных из *Главы 2*.

В общем, рекурсивные алгоритмы рекомендуется применять для решения задач, описанных с помощью рекуррентных соотношений, таких как: синтаксический анализ текстов, обработка динамических структур данных, обработка изображений и т.п. Типичным примером задач подобного рода является грамматический анализ программ на ПАСКАЛЕ, синтаксис которых, как известно, определяется при помощи рекуррентных соотношений.

Вопросы и упражнения

1. Объясните термин *методы программирования*.
2. В чем разница между *прямой рекурсией* и *косвенной рекурсией*? Приведите примеры.
3. Какие условия должны быть соблюдены, чтобы определение рекурсивного алгоритма было правильным?
4. В чем разница между сходящейся и расходящейся рекурсиями?

❺ Рассмотрим следующие рекурсивные определения функций. Какие из этих определений являются сходящимися? Обоснуйте Ваш ответ.

a) $f: \mathbf{N} \to \mathbf{N}$, $$f(n) = \begin{cases} 1, \text{ если } n = 0; \\ n + f(n-1), \text{ если } n > 0; \end{cases}$$

b) $f: \mathbf{N} \to \mathbf{N}$, $$f(n) = \begin{cases} 1, \text{ если } n = 0; \\ n + f(n), \text{ если } n > 0; \end{cases}$$

c) $f: \mathbf{Z} \to \mathbf{Z}$, $$f(i) = \begin{cases} 1, \text{ если } i = 0; \\ i + f(i-1), \text{ если } i \neq 0; \end{cases}$$

d) $f: \mathbf{N} \to \mathbf{N}$, $$f(n) = \begin{cases} 1, \text{ если } n = 0; \\ n + g(n-1), \text{ если } n > 0; \end{cases}$$

$g: \mathbf{N} \to \mathbf{N}$, $$g(n) = \begin{cases} 2, \text{ если } n = 0; \\ n + f(n-1), \text{ если } n > 0; \end{cases}$$

e) $f: \mathbf{N} \to \mathbf{N}$, $$f(n) = \begin{cases} n, \text{ если } n < 10; \\ (n \bmod 10) + f(n \operatorname{div} 10), \text{ если } n \geq 10. \end{cases}$$

❻ Запустите на выполнение следующую программу. Объясните сообщения, выводимые на экран.

```
Program P150;
 { Переполнение стека }
type Natural = 0..Maxint;

function Incons(n:Natural):Natural;
 { расходящееся рекурсивное определение }
begin
  writeln('Рекурсивный вызов n=', n);
  if n=0 then Incons:=1
         else Incons:=n*Incons(n+1);
end; { Incons }

begin
  writeln(Incons(3));
  readln;
end.
```

Представьте на рисунке, аналогичном *рис. 5.1*, способ управления стеком для случая вызова функции `Incons(3)`.

❼ Укажите элементарные и неэлементарные случаи для следующих рекурсивных алгоритмов:

а) функции `Arb` и процедуры `AfisArb` программы `P130`;

б) процедуры `Preordine`, `Inordine` и `Postordine` программы `P131`;

в) процедуры `InAdincime` программы `P133`.

❽ Сумма *S*(*n*) первых *n* натуральных чисел может быть вычислена при помощи итеративной функции:

$$S(n) = 0 + 1 + 2 + \ldots + n = \sum_{i=o}^{n} i$$

или рекурсивной функции:

$$S(n) = \begin{cases} 0, \text{ если } n = 0; \\ n + S(n-1), \text{ если } n > 0. \end{cases}$$

Используя в качестве модели *таблицу 5.1*, укажите достоинства и недостатки алгоритмов вычисления суммы *S*(*n*).

9. Рассмотрим следующие металингвистические формулы:
   <Цифра> ::= `0|1|2|3|4|5|6|7|8|9`
   <Число> ::= *<Цифра>* {*<Цифра>*}
   <Знак> ::= `+|-|*|/`
   <Выражение> ::= *<Число>* | `(`*<Выражение>*`)` |*<Выражение><Знак><Выражение>*

   Напишите рекурсивную функцию, возвращающую значение `true`, если строка *S* соответствует определению лексической единицы <*Выражение*> и `false` – в противном случае.

10. Укажите достоинства и недостатки итеративных и рекурсивных алгоритмов, предназначенных для создания и обработки следующих динамических структур данных:
    а) односвязных списков;
    б) стека;
    в) очереди;
    г) двоичных деревьев;
    д) деревьев порядка *m*.

11. Напишите итеративную функцию, возвращающую значение `true`, если строка символов *S* соответствует определению лексической единицы <*Выражение*> из *упражнения 9* и `false` – в противном случае. Используя в качестве модели *таблицу 5.1*, проведите сопоставительное сравнение итеративного и рекурсивного алгоритмов.

12. Напишите рекурсивную подпрограмму, вычисляющую сумму цифр произвольного натурального числа *n*. Исследуйте случаи $n \leq$ `MaxInt` и $n \leq 10^{250}$.

13. Черно-белые изображения (*рис. 5.2*) могут быть закодированы с помощью двоичной матрицы $B = ||b_{ij}||_{n \times m}$, $1 \leq n$, $m \leq 30$. Элемент b_{ij} указывает цвет соответствующей микрозоны: черный (b_{ij} =1) или белый (b_{ij} =0).

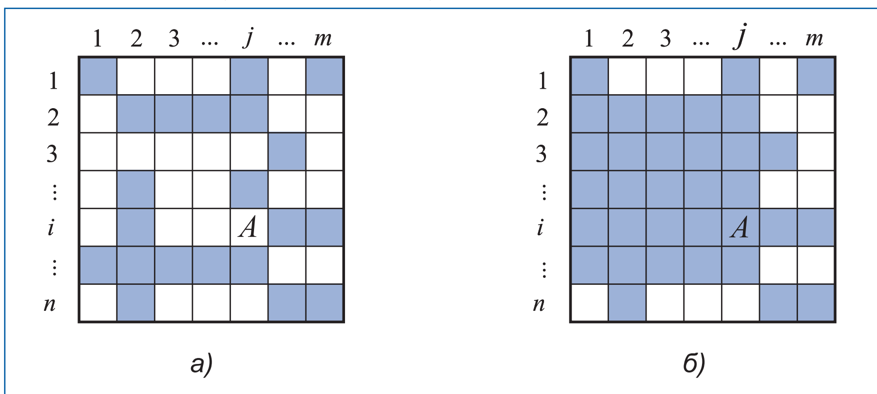


Рис. 5.2. Закраска замкнутой поверхности:
а – исходное изображение; *б* –изображение после закраски

Разработайте рекурсивную процедуру закраски замкнутой поверхности, определенной координатами (*i, j*) произвольной внутренней точки.

⓮ Напишите процедуру, которая определяет, сколько объектов содержит черно-белое изображение. Изображение делится на микрозоны и кодируется при помощи двоичной матрицы $B = ||b_{ij}||_{n \times m}$. Элемент b_{ij} определяет цвет микрозоны с координатами (*i, j*).

⓯ Укажите достоинства и недостатки итеративных и рекурсивных алгоритмов, предназначенных для решения задач из *упражнений 13* и *14*.

5.2. Метод полного перебора

В методе полного перебора предполагается, что решение рассматриваемой задачи может быть найдено путем последовательного анализа элементов s_i, принадлежащих конечному множеству

$$S = \{s_1, s_2, \ldots, s_i, \ldots, s_k\},$$

называемому **множеством возможных решений**. В простейших случаях, элементы s_i, $s_i \in S$, могут быть представлены значениями какого-нибудь порядкового типа: `integer`, `boolean`, `char`, *перечисления* или *интервального*. В более сложных задачах для представления таких элементов используются массивы, записи (`record`) или множества. Подчеркнем, что в большинстве задач возможные решения $s_1, s_2, \ldots, s_k$ не представлены явным образом и разработка алгоритма их нахождения ложится целиком на плечи программиста.

Общая схема алгоритма, основанного на методе полного перебора, может быть представлена с помощью цикла:

```
for i:= 1 to k do
   if SolutiePosibila(si) then PrelucrareaSolutiei(si)
```

где `SolutiePosibila` – это логическая функция, возвращающая значение `true`, если элемент s_i удовлетворяет условиям задачи, и `false` – в противном случае, а `PrelucrareaSolutiei` – это процедура, осуществляющая обработку выбранного элемента. Обычно эта процедура выводит решение s_i на экран.

Рассмотрим два примера, которые показывают достоинства и недостатки алгоритмов, основанных на методе перебора.

Пример 1. Рассмотрим натуральные числа, принадлежащие множеству {0, 1, 2, ..., *n*}. Напишите программу, определяющую, для скольких чисел *K* этого множества сумма цифр каждого числа равна *m*. В частности, для *n* = 100 и *m* = 2, множество {0, 1, 2, …, 100} содержит 3 числа, удовлетворяющие условию задачи: 2, 11 и 20. Следовательно, *K* = 3.

Решение. Очевидно, что множество возможных решений *S* = {0, 1, 2, …, *n*}. В следующей программе сумма цифр любого натурального числа *i*, *i* ∈ *S*, вычисляется с помощью функции `SumaCifrelor`. Выделение десятичных цифр из записи натурального числа *i* осуществляется справа налево путем деления нацело числа *i* и соответствующих частных на основание системы счисления 10.

```
Program P151;
 { Сумма цифр натурального числа }
type Natural=0..MaxInt;
var i, K, m, n : Natural;
function SumaCifrelor(i:Natural):Natural;
var suma : Natural;
begin
  suma:=0;
  repeat
    suma:=suma+(i mod 10);
    i:=i div 10;
  until i=0;
  SumaCifrelor:=suma;
end; { SumaCifrelor }

function SolutiePosibila(i:Natural):boolean;
begin
  if SumaCifrelor(i)=m then SolutiePosibila:=true
                       else SolutiePosibila:=false;
end; { SumaCifrelor }

procedure PrelucrareaSolutiei(i:Natural);
begin
  writeln('i=', i);
  K:=K+1;
end; { PrelucrareaSolutiei }

begin
  write('Введите n='); readln(n);
  write('Введите m='); readln(m);
  K:=0;
  for i:=0 to n do
    if SolutiePosibila(i) then PrelucrareaSolutiei(i);
  writeln('K=', K);
  readln;
end.
```

Анализ программы P151 показывает, что временная сложность соответствующего алгоритма составляет O(n).

Пример 2. Рассмотрим множество $P = \{P_1, P_2, \ldots, P_n\}$, образованное n точками ($2 \le n \le 30$) на евклидовой плоскости. Каждая точка P_j определена ее координатами x_j, y_j. Напишите программу, которая выводит на экран координаты точек P_a, P_b , расстояние между которыми максимально.

Решение. Множество возможных решений $S = P \times P$. Элементы (P_j, P_m) декартова произведения $P \times P$ могут быть получены при помощи двух вложенных циклов:

```
for j:=1 to n do
  for m:=1 to n do
    if SolutiePosibilă(Pj, Pm) then PrelucrareaSolutiei(Pj, Pm)
```

Расстояние между точками P_j, P_m вычисляется по формуле:

$$d_{jm} = \sqrt{(x_j - x_m)^2 + (y_j - y_m)^2}.$$

```
Program P152;
 { Точки на евклидовой плоскости }
const nmax=30;
type Punct = record
              x, y : real;
             end;
     Indice = 1..nmax;
var P : array[Indice] of Punct;
    j, m, n : Indice;
    dmax : real; { максимальное расстояние }
    PA, PB : Punct;

function Distanta(A, B : Punct) : real;
begin
  Distanta:=sqrt(sqr(A.x-B.x)+sqr(A.y-B.y));
end; { Distanta }

function SolutiePosibila(j,m:Indice):boolean;
begin
  if j<>m then SolutiePosibila:=true
          else SolutiePosibila:=false;
end; { SolutiePosibila }

procedure PrelucrareaSolutiei(A, B : Punct);
begin
  if Distanta(A, B)>dmax then
    begin
      PA:=A; PB:=B;
      dmax:=Distanta(A, B);
    end;
end; { PrelucrareaSolutiei }

begin
  write('Введите n='); readln(n);
  writeln('Введите координаты x, y точек');
  for j:=1 to n do
    begin
      write('P[', j, ']: '); readln(P[j].x, P[j].y);
    end;
  dmax:=0;
```

```
  for j:=1 to n do
     for m:=1 to n do
        if SolutiePosibila(j, m) then
           PrelucrareaSolutiei(P[j], P[m]);

  writeln('Решение: PA=(', PA.x:5:2, ',', PA.y:5:2, ')');
  writeln('         PB=(', PB.x:5:2, ',', PB.y:5:2, ')');
  readln;
end.
```

Временная сложность алгоритма, описанного с помощью программы `P152`, составляет $O(n^2)$.

Рассмотрев данные примеры, замечаем, что в алгоритмах, основанных на методе перебора, вычисляется, явно или неявно, множество возможных решений S. В относительно простых задачах (*Пример 1*) элементы множества возможных решений могут быть перечислены непосредственно. В более сложных задачах (*Пример 2*) генерирование возможных решений требует разработки специального алгоритма. В общем случае, такие алгоритмы реализуют операции, связанные с обработкой конечных множеств:

– объединение;
– пересечение;
– разность;
– генерирование всевозможных подмножеств;
– генерирование элементов декартова произведения множеств;
– генерирование всевозможных перестановок, размещений или сочетаний элементов и т.п.

Главным достоинством алгоритмов полного перебора является то, что соответствующие программы относительно просты, а их отладка не требует сложных тестов. Временная сложность таких алгоритмов определяется количеством элементов k множества возможных решений S. Для большинства практически важных задач метод перебора приводит к экспоненциальным алгоритмам. Поскольку экспоненциальные алгоритмы неприемлемы в случае больших объемов входных данных, метод перебора применяется только в учебных целях или для разработки таких программ, время выполнения которых не является критичным.

Вопросы и упражнения

❶ Объясните общую структуру алгоритмов полного перебора.

❷ Как может быть реализован перебор возможных решений с использованием циклов **`while`** и **`repeat`**?

❸ Оцените время выполнения программ `P151` и `P152`. Измените программу `P152` так, чтобы время ее выполнения уменьшилось примерно в два раза.

❹ Приведите примеры программ, время выполнения которых не является критичным.

❺ Какими преимуществами и недостатками обладают алгоритмы, основанные на методе полного перебора?

❻ Рассмотрим множество $P = \{P_1, P_2, \ldots, P_n\}$, образованное n точками ($3 \le n \le 30$) на евклидовой плоскости. Каждая точка P_j определена своими координатами x_j, y_j. Напишите программу, которая находит такие три точки из множества P, для которых площадь соответствующего треугольника максимальна. Оцените время выполнения написанной программы.

❼ Напишите функцию ПАСКАЛЬ, которая, получив в качестве параметра натуральное число n, возвращает значение `true`, если n является простым, и `false` – в противном случае. Оцените временную сложность соответствующей функции.

❽ В обозначении $(a)_x$ буква x представляет основание системы счисления, а буква a – число, записанное в соответствующей системе. Напишите программу, которая вычисляет, если существует, хотя бы один корень уравнения

$$(a)_x = b,$$

где a и b натуральные числа, а x неизвестно. Каждая цифра натурального числа a принадлежит множеству {0, 1, 2, …, 9}, а число b записано в десятичной системе счисления. Например, корнем уравнения

$$(160)_x = 122$$

является $x = 8$, а уравнение

$$(5)_x = 10$$

не имеет решений. Оцените временную сложность разработанной программы.

❾ В копилке находится N монет различных достоинств общим весом G граммов. Вес каждой монеты определенной стоимости дан в таблице, приведенной ниже.

Стоимость монеты, *лей*	***Вес монеты***, *грамм*
1	1
5	2
10	3
25	4
50	5

Напишите программу, которая определит минимальную сумму S, которая может находиться в копилке.

❿ Напишите программу, которая определяет, сколько точек с целочисленными координатами содержится в сфере радиуса R с центром в начале системы координат. Подразумевается, что R – натуральное число, $1 \le R \le 30$. Расстояние d между точкой с координатами (x, y, z) и началом системы координат определяется по формуле $d = \sqrt{x^2 + y^2 + z^2}$.

5.3. Метод *Greedy* – “жадный” алгоритм

Данный метод предполагает, что решаемая задача имеет следующую структуру:

– задано множество $A=\{a_1, a_2, \ldots, a_n\}$, образованное из n элементов;

– требуется определить подмножество B, $B \subseteq A$, удовлетворяющее определенным условиям, чтобы оно могло быть принято в качестве решения.

В принципе, задачи рассматриваемого типа могут быть решены с помощью метода полного перебора путем последовательной генерации 2^n подмножеств A_i множества *A*. Недостаток метода полного перебора состоит в том, что время выполнения соответствующего алгоритма очень велико.

Чтобы избежать полного перебора всевозможных подмножеств A_i, $A_i \subseteq A$, в методе *Greedy* применяется **критерий (правило)** прямого выбора необходимых элементов из множества *A*. Обычно критерии отбора не заданы явно в описании задачи, и их формулировка является задачей программиста. Очевидно, что при отсутствии подобного критерия метод *Greedy* не может быть применен.

Общая схема алгоритма, основанного на методе *Greedy*, может быть представлена с помощью следующего цикла:

```
while ExistaElemente do
begin
  AlegeUnElement(x);
  IncludeElementul(x);
end
```

Функция `ExistaElemente` возвращает значение `true`, если во множестве *A* существуют элементы, удовлетворяющие критерию отбора. Процедура `AlegeUnElement` извлекает из множества *A* такой элемент *x*, а процедура `IncludeElementul` записывает выбранный элемент в подмножество *B*. Первоначально подмножество *B* является пустым.

Как видим, в методе *Greedy* решение задачи ищется путем последовательного тестирования элементов множества *A* и включения некоторых из них в подмножество *B*. Образно говоря, подмножество *B* пытается “проглотить” “вкусные” элементы множества *A*, откуда и происходит название метода (*greedy* – жадный, ненасытный).

Пример. Рассмотрим множество $A=\{a_1, a_2, ..., a_i, ..., a_n\}$, элементы которого являются вещественными числами и хотя бы один из них удовлетворяет условию $a_i>0$. Напишите программу, которая вычисляет подмножество *B*, $B \subseteq A$, такое, чтобы сумма элементов подмножества *B* была максимальна. Например, для $A=\{21{,}5; –3{,}4; 0; –12{,}3; 83{,}6\}$ получим $B=\{21{,}5; 83{,}6\}$.

Решение. Заметим, что если подмножество *B*, $B \subseteq A$, содержит элемент $b \le 0$, тогда сумма элементов подмножества $B \setminus \{b\}$ больше или равна сумме элементов *B*. Следовательно, правило отбора очень простое: на каждом шаге в подмножество *B* включается любой произвольный положительный элемент множества *A*.

В приведенной ниже программе множества *A* и *B* представлены векторами (одномерными массивами) `A` и `B`, а количество элементов каждого множества – целыми переменными соответственно `n` и `m`. Первоначально *B* – пустое множество, соответственно $m=0$.

```
Program P153;
 { Tehnica Greedy }
const nmax=1000;
var A : array [1..nmax] of real;
    n : 1..nmax;
```

```
    B : array [1..nmax] of real;
    m : 0..nmax;
    x : real;
    i : 1..nmax;

Function ExistaElemente : boolean;
var i : integer;
begin
  ExistaElemente:=false;
  for i:=1 to n do
    if A[i]>0 then ExistaElemente:=true;
  end; { ExistaElemente }

procedure AlegeUnElement(var x : real);
var i : integer;
begin
  i:=1;
  while A[i]<=0 do i:=i+1;
  x:=A[i];
  A[i]:=0;
end; { AlegeUnElement }

procedure IncludeElementul(x : real);
begin
  m:=m+1;
  B[m]:=x;
end; { IncludeElementul }

begin
  write('Введите n='); readln(n);
  writeln('Введите элементы множества A:');
  for i:=1 to n do read(A[i]);
  writeln;
  m:=0;
  while ExistaElemente do
    begin
      AlegeUnElement(x);
      IncludeElementul(x);
    end;
  writeln('Элементы множества B:');
  for i:=1 to m do writeln(B[i]);
  readln;
end.
```

Отметим, что процедура `AlegeUnElement` обнуляет компоненту массива `A`, содержащую элемент `x`, извлеченный из соответствующего множества.

Временная сложность алгоритмов, основанных на методе *Greedy*, может быть оценена в соответствии с общей схемой, представленной выше. Обычно

время, затрачиваемое процедурами `ExistaElemente`, `AlegeUnElement` и `IncludeElementul`, составляет n. В составе цикла **while** эти процедуры выполняются не более n раз. Следовательно, алгоритмы, основанные на методе *Greedy*, представляют собой полиномиальные алгоритмы. Для сравнения отметим, что алгоритмы, основанные на переборе всех подмножеств A_i, $A_i \subseteq A$, являются алгоритмами порядка $O(2^n)$, т.е. экспоненциальными. К сожалению, метод *Greedy* применим только для задач, из условия которых может быть выведено правило, обеспечивающее прямой выбор нужных элементов из множества A.

Вопросы и упражнения

❶ Объясните общую схему алгоритмов, основанных на методе *Greedy*.

❷ Каковы преимущества и недостатки алгоритмов, основанных на методе *Greedy*?

❸ Оцените время выполнения программы `P153`.

❹ Опишите в общих чертах алгоритм полного перебора, который определяет подмножество *B* из вышеприведенного примера. Оцените временную сложность полученного алгоритма.

❺ **Хранение файлов на магнитных лентах.** Рассмотрим n файлов $f_1, f_2, ..., f_n$, которые должны быть записаны на магнитную ленту. Напишите программу, определяющую такой порядок размещения файлов на ленте, чтобы среднее время доступа к ним было минимальным. Предполагается, что частота обращения ко всем файлам (для чтения) одинакова, а на чтение файла f_i (i=1, 2, ..., n) затрачивается t_i секунд.

❻ **Непрерывная задача о рюкзаке.** Имеются n предметов. Для каждого предмета i (i=1, 2, ..., n) известен вес g_i и прибыль c_i, которая получается при транспортировке. Имеется рюкзак, в котором можно переносить один и более предметов, суммарный вес которых не превышает величины G_{max}. Напишите программу, которая определяет, каким образом необходимо загрузить рюкзак с тем, чтобы суммарная прибыль *C* была максимальна. При необходимости, переносимые предметы можно разделять на меньшие части.

❼ **Криковские подвалы.** После посещения знаменитых Криковских подвалов* один информатик сконструировал робота, который может двигаться по полю, разделенному на квадраты (*рис. 5.3*). Робот может выполнять следующие команды: `ВВЕРХ`, `ВНИЗ`, `НАПРАВО`, `НАЛЕВО`, в соответствии с которыми он перемещается в один из соседних квадратиков. Если в квадрате имеется препятствие, например стена или бочка, то происходит столкновение и робот ломается.

Напишите программу, которая по известному плану подвалов проведет робота по подземным помещениям от входа в подвалы до выхода. Поскольку коллекция вин очень большая, нет необходимости посещать все подземные помещения Криковских подвалов.

Входные данные. План подвалов нарисован на листе бумаги в клетку. Заштрихованные квадратики обозначают препятствия, а незаштрихованные – свободные пространства. Квадратики по периметру плана, за исключением входа и выхода

* В Криковских подвалах на протяжении многих десятилетий закладываются на хранение лучшие вина Республики Молдова.

закрашены по определению. В цифровом виде план подвалов представлен в виде массива A из m строк и n столбцов. Элементы $A[i, j]$ указанного массива имеют следующие значения: 0 – свободно; 1 – препятствие; 2 – вход в подвалы; 3 – выход из подвалов. Первоначально робот находится в квадратике, для которого $A[i, j]=2$.

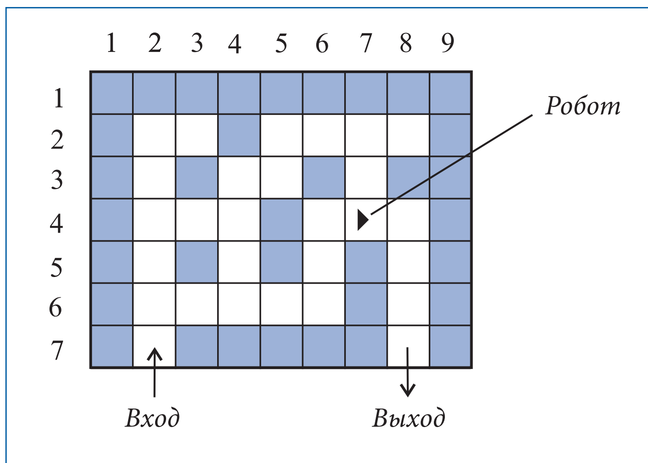


Рис. 5.3. Рабочее поле робота

Файл HRUBE.IN содержит в первой строке числа m, n, разделенные пробелом. В каждой из следующих m строк содержится по n чисел $A[i, j]$, разделенных пробелами.

Выходные данные. Файл HRUBE.OUT должен содержать в каждой строке по одной из команд `ВВЕРХ`, `ВНИЗ`, `НАПРАВО`, `НАЛЕВО`, записанных в порядке их выполнения роботом.

Ограничения. $5 \le m, n \le 100$. Время выполнения программы не должно превышать 3 секунды.

Пример:

HRUBE.IN

```
7 9
1 1 1 1 1 1 1 1 1
1 0 0 1 0 0 0 0 1
1 0 1 0 0 1 0 1 1
1 0 0 0 1 0 0 0 1
1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 1 0 1
1 2 1 1 1 1 1 3 1
```

HRUBE.OUT

```
ВВЕРХ
НАПРАВО
НАПРАВО
НАПРАВО
НАПРАВО
ВВЕРХ
ВВЕРХ
НАПРАВО
НАПРАВО
ВНИЗ
ВНИЗ
ВНИЗ
```

Указание. На каждом шаге выбирайте из множества {`ВВЕРХ`, `ВНИЗ`, `НАПРАВО`, `НАЛЕВО`} по одной команде таким образом, чтобы робот все время двигался только вдоль некоторой стены.

5.4. Метод перебора с возвратом

В методе перебора с возвратом предполагается, что решение задачи, которую необходимо решить, может быть представлено с помощью вектора

$$X=(x_1, x_2, ..., x_k, ... , x_n).$$

Каждая компонента x_k вектора X может принимать значения, принадлежащие конечному множеству A_k, $k = 1, 2, ..., n$. Преполагается, что все m_k элементов каждого множества A_k упорядочены по некоторому заранее определенному критерию, например по порядку их записи в памяти компьютера.

В принципе, такие задачи могут быть решены методом полного перебора, где вектор X рассматривается как элемент декартова произведения $S = A_1 \times A_2 \times ... \times A_n$. Недостаток метода полного перебора состоит в том, что временная сложность соответствующих алгоритмов очень большая. Например, для множеств $A_1, A_2, ..., A_n$, каждое из которых состоит всего из двух элементов, время вычислений имеет порядок $O(2^n)$, т.е. алгоритм является экспоненциальным.

Чтобы избежать перебора всех элементов декартова произведения $A_1 \times A_2 \times ... \times A_n$, в методе перебора с возвратом компоненты вектора X принимают значения по очереди, т.е. значение для x_k вычисляется только тогда, когда уже присвоены значения компонентам $x_1, x_2, ..., x_{k-1}$. Более того, при подборе значений для x_k переход к вычислению значений для x_{k+1} осуществляется не сразу, а после проверки определенных **условий продолжения**, касающихся ранее вычисленных значений $x_1, x_2, ..., x_k$. Указанные условия выявляют ситуации, при которых можно перейти к вычислению x_{k+1}. Если данные условия не выполняются, необходимо выбрать другое значение для x_k или, если элементы множества A_k исчерпаны, следует уменьшить k на единицу, осуществив тем самым возврат к выбору новых значений для x_{k-1}.

Подчеркнем, что именно упомянутое выше уменьшение значения k и дало название изучаемому методу. "Возврат" означает, что мы возвращаемся к другим вариантам выбора переменных $x_1, x_2, ..., x_{k-1}$. Английское название изучаемого метода – *backtracking* (*back* – назад, *track* – след).

Для примера, на *рис. 5.4* представлен порядок, в котором рассматриваются элементы множеств $A_1 = \{a_{11}, a_{12}\}$, $A_2 = \{a_{21}, a_{22}\}$ и $A_3 = \{a_{31}, a_{32}, a_{33}\}$. В учебных целях считается, что каждое из множеств A_1 и A_2 содержат по два элемента ($m_1 = 2$, $m_2 = 2$), а множество A_3 – три элемента ($m_3 = 3$). Элементы a_{kj} соответствующих множеств обозначены кружочками. Результаты проверки условий продолжения представлены на рисунке двоичными значениями 0 (`false`) и 1 (`true`).

На *рис. 5.4* замечаем, что первый элемент a_{11} множества A_1 не удовлетворяет условиям продолжения и, исходя из этого, переходим ко второму элементу a_{12} этого же множества. Далее в вектор X включаем первый элемент a_{21} множества A_2, который удовлетворяет условиям продолжения, и переходим к рассмотрению элементов из множества A_3.

Поскольку ни один из элементов a_{31}, a_{32}, a_{33} не удовлетворяет условиям продолжения, возвращаемся к множеству A_2, из которого выбираем второй элемент, а именно a_{22}. После этого заново проверяются элементы множества A_3, где элемент a_{32} удовлетворяет условиям продолжения.

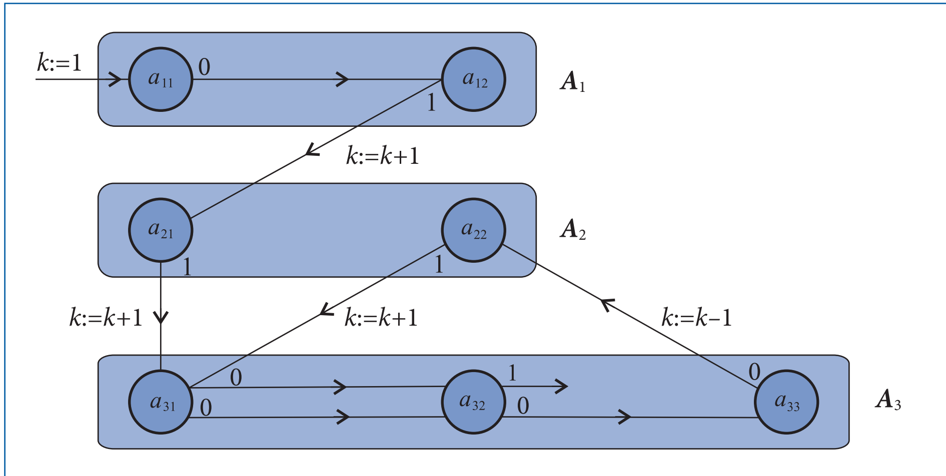


Рис. 5.4. Поиск решения методом перебора с возвратом

Общая схема рекурсивного алгоритма, основанного на методе перебора с возвратом, может быть представлена при помощи следующей процедуры:

```
procedure Reluare(k:integer);
begin
  if k<=n then
    begin
      X[k]:=PrimulElement(k);
      if Continuare(k) then Reluare(k+1);
      while ExistaSuccesor(k) do
        begin
          X[k]:=Succesor(k);
          if Continuare(k) then Reluare(k+1)
        end; { while }
    end { then }
  else PrelucrareaSolutiei;
end; {Reluare}
```

Процедура `Reluare` обменивается с вызывающей программой и вызываемыми подпрограммами через глобальные переменные, которые представляют вектор X и множества $A_1, A_2, ..., A_n$. Вызываемые подпрограммы выполняют следующие действия:

`PrimulElement(k)` – возвращает первый элемент множества A_k;

`Continuare(k)` – возвращает значение `true`, если элементы, записанные в первые k компоненты вектора X, удовлетворяют условиям продолжения и `false` – в противном случае;

`ExistaSuccesor(k)` – возвращает значение `true`, если в упорядоченном множестве A_k за элементом, записанном в компоненте x_k, следует другой элемент и `false` – в противном случае;

`Succesor(k)` – возвращает элемент, следующий в множестве A_k за элементом из компоненты x_k;

`PrelucrareaSolutiei` – обычно эта процедура выводит решение, содержащееся в полученном векторе X , на экран.

Для лучшего понимания рекурсивного процесса проследим выполнение процедуры `Reluare` для случая, изображенного на *рис. 5.4*.

При выполнении вызова `Reluare(1)` в компоненту x_1 вектора X записывается первый элемент множества A_1:

$$X=(a_{11}).$$

Поскольку в случае этого вектора вызов `Continuare(1)` возвращает значение `false`, переходим к выполнению оператора **`while`**. Обращение `ExistaSuccesor(1)` возвращает значение `true` и, как следствие, в компоненту x_1 вектора X записывается элемент следующий за a_{11}:

$$X=(a_{12}).$$

В этом случае вызов `Continuare(1)` возвращает значение `true` и, следовательно, переходит к выполнению рекурсивного вызова `Reluare(2)`. Очевидно, что в компоненту x_2 вектора X будет записан первый элемент множества A_2:

$$X=(a_{12}, a_{21}).$$

Для этого вектора функция `Continuare` возвращает значение `true`, что ведет к началу выполнения рекурсивного вызова `Reluare(3)`.

В случае вызова `Reluare(3)` компонента x_3 вектора X принимает рекурсивно значения a_{31}, a_{32} и a_{33}:

$$X=(a_{12}, a_{21}, a_{31});$$
$$X=(a_{12}, a_{21}, a_{32});$$
$$X=(a_{12}, a_{21}, a_{33}),$$

но ни один из этих векторов не удовлетворяет условиям продолжения. После просмотра последнего элемента множества A_3 функция `ExistaSuccesor` возвращает значение `false` и, следовательно, процесс вычислений возвращается в контекст вызова `Reluare(2)`. В данном контексте в компоненту x_2 вектора X записывается элемент, следующий за элементом a_{21}:

$$X=(a_{12}, a_{22}).$$

Поскольку и для этого вектора функция `Continuare` возвращает значение `true`, снова осуществляется рекурсивный вызов `Reluare(3)`. Но, в отличие от предыдущего вызова, в данном случае вектор

$$X=(a_{12}, a_{22}, a_{32})$$

удовлетворяет условиям продолжения, что запускает рекурсивный вызов `Reluare(4)`. При этом $k>n$ и, как следствие, процедура `PrelucrareaSolutiei` выводит координаты вектора X на экран.

Пример. Рассмотрим множества A_1, A_2, ..., A_n, каждое из которых образовано из m_k натуральных чисел. Выберите из каждого множества по одному числу так, чтобы их сумма была равна q.

Решение. Представим множества A_1, A_2, ..., A_n строками двумерного массива (матрицы) $A = ||a_{kj}||$. Число элементов m_k каждого множества A_k будет содержаться в соответствующей компоненте одномерного массива (вектора) $M = ||m_k||$.

Из описания задачи следует, что условия продолжения соблюдаются только тогда, когда сумма элементов, заданных первыми k компонентами вектора X, не превосходит значения q для $k<n$ и равна q для $k = n$. Для упрощения программы в векторе X запоминаются только индексы j элементов a_{ij}, выбранных из множеств $A_1, A_2, ..., A_n$.

```
Program P154;
 { Программа, основанная на методе перебора с возвратом }
const mmax=50; { максимальное количество множеств }
      nmax=50; { максимальное количество элементов }
type Natural = 0..MaxInt;
     Multime = array [1..nmax] of Natural;

var  A : array[1..nmax] of Multime;
     n : 1..nmax;                      { количество множеств }
     M : array[1..nmax] of 1..mmax; { кардинал множества S[k] }
             X : array[1..nmax] of 1..mmax; { индексы выбранных
                                                    элементов }
     q : Natural;
     k, j : integer;
     Indicator : boolean;

function PrimulElement(k : integer) : Natural;
begin
  PrimulElement:=1;
end; {PrimulElement }

function Continuare(k : integer) : boolean;
var j : integer;
    suma : Natural;
begin
  suma:=0;
  for j:=1 to k do suma:=suma+A[j, X[j]];
  if ((k<n) and (suma<q)) or ((k=n) and (suma=q))
    then Continuare:=true
    else Continuare:=false;
end; { Continuare }

function ExistaSuccesor(k : integer) : boolean;
begin
  ExistaSuccesor:=(X[k]<M[k]);
end; { ExistaSuccesor }

function Succesor(k : integer) : integer;
begin
  Succesor:=X[k]+1;
end; { Succesor }
```

```
procedure PrelucrareaSolutiei;
var k : integer;
begin
  write('Решение: ');
  for k:=1 to n do write(A[k, X[k]], ' ');
  writeln;
  Indicator:=true;
end; { PrelucrareaSolutiei }

procedure Reluare(k : integer);
 { Метод перебора с возвратом - рекурсивный вариант }
begin
  if k<=n then
       begin
         X[k]:=PrimulElement(k);
         if Continuare(k) then Reluare(k+1);
         while ExistaSuccesor(k) do
           begin
             X[k]:=Succesor(k);
             if Continuare(k) then Reluare(k+1);
           end { while }
       end { then }
     else PrelucrareaSolutiei;
end; { Reluare }

begin
  write('Введите количество множеств n='); readln(n);
  for k:=1 to n do
    begin
      write('Введите кардинал A[', k, ']='); readln(M[k]);
      write('Введите элементы множества A[', k, ']: ');
      for j:=1 to M[k] do read(A[k, j]);
      writeln;
    end;
  Write('Введите q='); readln(q);
  Indicator:=false;
  Reluare(1);
  if Indicator=false then writeln('Решения нет');
  readln;
end.
```

Из анализа процедуры `Reluare` и порядка извлечения элементов множеств $A_1, A_2, ..., A_n$ (*рис. 5.4*) следует, что при выполнении алгоритма, в наихудшем случае, будут сгенерированы всевозможные векторы X декартового произведения $A_1 \times A_2 \times ... \times A_n$. Следовательно, **временная сложность** алгоритмов, основанных на методе перебора с возвратом, как и в случае метода полного перебора, составляет $O(m^n)$, где $m=\max(m_1, m_2, ..., m_n)$.

На практике, конкретное время выполнения алгоритмов перебора с возвратом в большой степени зависит от природы решаемых задач, точнее, от вида условий продолжения вычислений и порядка расположения элементов a_{kj} во множествах A_1, A_2, ..., A_n. Правильный выбор условий продолжения ведет к существенному снижению объема вычислений. В частности, в наилучшем случае будет рассмотрено только по одному элементу из каждого множества A_1, A_2, ..., A_n и, следовательно, время выполнения алгоритма будет пропорционально n.

К сожалению, в настоящее время не существует универсального правила, которое позволило бы сформулировать "хорошие" условия продолжения для любой задачи, встречающейся на практике. Нахождение таких условий и, следовательно, уменьшение времени вычислений зависит от мастерства программиста. Отметим, что для большинства задач, множества A_1, A_2, ..., A_n заранее не известны, а компоненты вектора X могут принадлежать составным типам – массивам, множествам или записям.

Вопросы и упражнения

❶ Объясните общую структуру алгоритмов, основанных на методе перебора с возвратом.

❷ Укажите элементарные и неэлементарные случаи рекурсивной процедуры `Reluare`.

❸ Напишите итеративный вариант процедуры `Reluare`.

❹ Сопоставьте итеративные и рекурсивные алгоритмы, основанные на методе перебора с возвратом.

❺ Представьте на рисунке, похожем на *рис. 5.4*, порядок, в котором извлекаются элементы множеств A_1, A_2, ..., A_n из программы `P154`:

a) $A_1=\{1\}$, $A_2=\{2\}$, $A_3=\{3\}$, $q=4$;

b) $A_1=\{1\}$, $A_2=\{2, 3\}$, $A_3=\{4, 5\}$, $q=10$;

c) $A_1=\{1, 2, 3\}$, $A_2=\{4, 5\}$, $A_3=\{6\}$, $q=14$;

d) $A_1=\{1, 2\}$, $A_2=\{3, 4\}$, $A_3=\{5, 6\}$, $A_4=\{7, 8\}$, $q=20$.

❻ Укажите условия продолжения вычислений для программ, в которых нужно сгенерировать всевозможные элементы декартового произведения $A_1 \times A_2 \times ... \times A_n$. Для проверки вашего ответа напишите соответствующую программу на ПАСКАЛЕ и отладьте ее.

❼ Укажите на *рис. 5.4* порядок перебора элементов множеств A_1, A_2, A_3 в наилучшем и наихудшем случаях.

❽ Рассмотрим множество $B=\{b_1, b_2, ..., b_n\}$, содержащее первые n букв латинского алфавита. Напишите программу, основанную на методе перебора с возвратом, которая вычисляет всевозможные подмножества B_i, $B_i \subseteq B$, содержащие ровно по q букв.

Указание. Каждое подмножество B_i может быть представлено характеристическим вектором $X = ||x_k||_n$, где

$$x_k = \begin{cases} 1, & \text{если } b_k \in B_i; \\ 0, & \text{в противном случае.} \end{cases}$$

Очевидно, что подмножество B_i удовлетворяет условиям задачи, если $x_1+x_2+...+x_n=q$.

❾ **Раскраска географических карт.** На карте представлены n стран, $n \le 30$. В памяти компьютера карта представлена при помощи матрицы $A=||a_{ij}||_{n \times m}$, где

$$a_{ij} = \begin{cases} 1, & \text{если страны } i, j \text{ – соседние;} \\ 0, & \text{в противном случае.} \end{cases}$$

Найдите минимальное число цветов *m,* необходимых для раскраски карты. Естественно, любые две соседние страны должны быть закрашены различными цветами.

❿ **Лабиринт.** Рассмотрим план произвольного лабиринта, изображенного на листе бумаги "в клетку" (*рис. 5.5*). Закрашенные клетки представляют препятствия, а незакрашенные – комнаты и коридоры лабиринта.

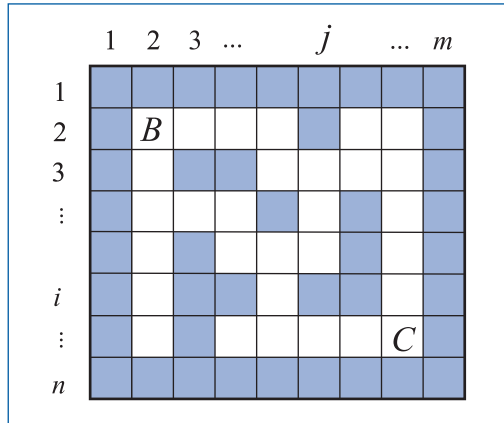


Рис. 5.5. План лабиринта

В памяти компьютера план лабиринта представляется при помощи матрицы $A=||a_{ij}||_{n \times m}$, $1 \le n, m \le 30$, где

$$a_{ij} = \begin{cases} 1, & \text{если клетка } (i, j) \text{ закрашена;} \\ 0, & \text{в противном случае.} \end{cases}$$

Путешественник может перемещаться из одной незакрашенной клетки в другую только в том случае, если у них есть общая сторона. Напишите программу, которая определяет, существует ли путь из клетки *B* в клетку *C*.

⓫ Дано n ($n \le 30$) кошельков, пронумерованных числами 1, 2, 3, ..., n. Кошелек с номером i, (i = 1, 2, 3, ..., n), содержит m_i монет одного и того же достоинства V_i. Напишите программу, которая выводит на экран, если это возможно, один из способов выплаты суммы *S* ровно *p* монетами, взятыми из рассматриваемых кошельков.

⓬ Напишите программу, которая выводит на экран все способы разложения натурального числа n в виде суммы k взаимно различных натуральных чисел. Например, для n=9 и k=3 решениями являются суммы 1+2+6, 2+3+4 и 1+3+5.

⓭ Выполните сравнительный анализ алгоритмов, основанных на методе полного перебора и методе перебора с возвратом, разработанных для решения задач из *упражнений 8* и *12*.

5.5. Метод *“разделяй и властвуй”*

Метод *“разделяй и властвуй”* (на латыни *divide et impera*) представляет собой обобщенный подход к разработке алгоритмов, основанный на:

1) повторяющемся разделении решаемой задачи большого размера на две или более подзадачи такого же типа, но меньшего размера;

2) решении полученных подзадач прямым способом, если их размеры это позволяют, или их разделении на другие подзадачи еще меньшего размера;

3) обработке частичных решений соответствующих подзадач с целью получения решения исходной задачи.

Предположим, что на определенном шаге алгоритма вычислено упорядоченное множество:

$$A = (a_i, a_{i+1}, \ldots, a_j)$$

но для получения решения необходима дальнейшая обработка его элементов. Для того чтобы разделить задачу на две близкие по размерам подзадачи, положим

$$m = (j - i) \text{ div } 2$$

и разделим множество A на два подмножества, которые будут обработаны по отдельности:

$$A_1 = (a_i, a_{i+1}, \ldots, a_{i+m}); \;\; A_2 = (a_{i+m+1}, a_{i+m+2}, \ldots, a_j).$$

Далее, каждое из множеств A_1 и A_2 снова делится на два подмножества A_{1-1}, A_{1-2} и A_{2-1}, A_{2-2} соответственно. Этот процесс будет продолжаться до тех пор, пока частичные решения для соответствующих подмножеств не могут быть найдены непосредственно, без последующего разделения.

Для примера, на *рис. 5.6* представлен способ разделения множества $A=(a_1, a_2, \ldots, a_7)$ в случае разделения каждой из текущих задач на две подзадачи того же типа.

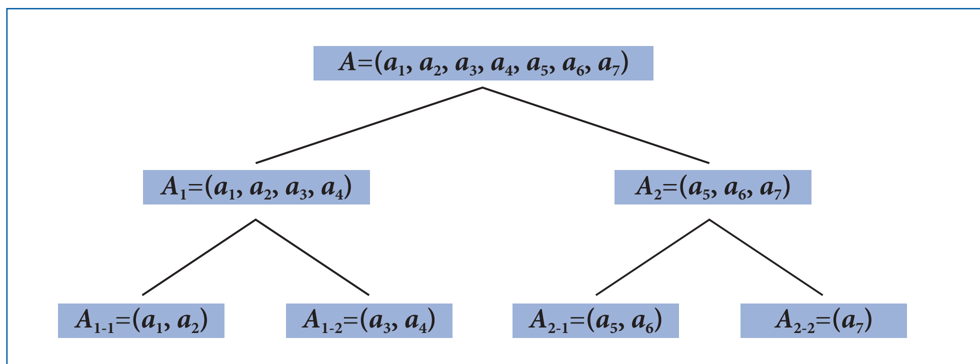


Рис. 5.6. Разделение множества A на подмножества

Общая схема алгоритмов, основанных на методе *”разделяй и властвуй”*, может быть представлена с помощью следующей рекурсивной процедуры:

```
procedure DesparteSiStapineste(i, j : integer; var x : tip);
var m : integer;
    x1, x2 : tip;
```

```
procedure DesparteSiStapineste(i, j : integer; var x : tip);
var m : integer;
    x1, x2 : tip;
begin
   if SolutieDirecta(i, j) then Prelucrare(i, j, x)
     else
         begin
           m:=(j-i) div 2;
           DesparteSiStapineste(i, i+m, x1);
           DesparteSiStapineste(i+m+1, j, x2);
           Combina(x1, x2, x);
         end;
end;
```

Параметры `i` и `j` заголовка процедуры определяют текущее подмножество (a_i, a_{i+1}, ..., a_j), подлежащее обработке. Функция `SolutieDirecta` возвращает значение `true`, если текущая подзадача может быть решена напрямую, и `false` – в противном случае. Процедура `Prelucrare` возвращает через параметр `x` решение текущей задачи, вычисленное непосредственно. Если прямое решение невозможно, то происходит два рекурсивных вызова – один для подмножества (a_i, a_{i+1}, ..., a_m) и другой для подмножества (a_{i+m+1}, a_{i+m+2}, ..., a_j). Процедура `Combina` обрабатывает решения `x1` и `x2` соответствующих подзадач и возвращает решение `x` текущей задачи.

Пример 1. Рассмотрим множество A={a_1, a_2, ..., a_n}, состоящее из n вещественных чисел. Напишите программу, которая находит максимальное число из этого множества.

Решение. В приведенной программе множество A представлено с помощью одномерного массива из n элементов (компонент). Допустим, что решение текущей подзадачи может быть получено непосредственно только тогда, когда множество (a_i, ..., a_j) содержит одно или два числа. Очевидно, что в таких случаях $x = a_i$ или $x = \max(a_i , a_j)$.

```
Program P155;
 { Нахождение максимального элемента методом разделяй
 и властвуй }
const nmax=100;
var A : array[1..nmax] of real;
 i, n : 1..nmax;
    x : real;

function SolutieDirecta(i, j : integer) : boolean;
begin
  SolutieDirecta:=false;
  if (j-i<2) then SolutieDirecta:=true;
end; { SolutieDirecta }
```

```
procedure Prelucrare(i, j : integer; var x : real);
begin
  x:=A[i];
  if A[i]<A[j] then x:=A[j];
end; { Prelucrare }

procedure Combina(x1, x2 : real; var x : real);
begin
  x:=x1;
  if x1<x2 then x:=x2;
end; { Combina }

procedure DesparteSiStapineste(i, j : integer; var x : real);
var m : integer;
    x1, x2 : real;
begin
  if SolutieDirecta(i, j) then Prelucrare(i, j, x)
     else
       begin
         m:=(j-i) div 2;
         DesparteSiStapineste(i, i+m, x1);
         DesparteSiStapineste(i+m+1, j, x2);
         Combina(x1, x2, x);
       end;
end; { DesparteSiStapineste }

begin
  write('Введите n='); readln(n);
  writeln('Введите ', n, ' вещественных чисел');
  for i:=1 to n do read(A[i]);
  writeln;
  DesparteSiStapineste(1, n, x);
  writeln('Максимальное число x=', x);
  readln;
  readln;
end.
```

В программе P155 при каждом вызове процедуры DesparteSiStapineste текущая задача решается непосредственно либо делится на две подзадачи приблизительно равных размеров. Очевидно, что метод *"разделяй и властвуй"* позволяет разделять текущие задачи на произвольное число подзадач, не обязательно на две. В следующем примере текущая задача – разрезка пластины максимальной площади – разделяется на четыре подзадачи того же типа, но меньшего размера.

Пример 2. Рассмотрим прямоугольную пластину размером $L \times H$. На пластинке имеется n отверстий точечной формы. Каждое отверстие задается своими координатами (x_i, y_i). Напишите программу, которая "вырезает" из пластины прямоугольный фрагмент максимальной площади без отверстий. Разрешены

только непрерывные разрезы от одного до противоположного края пластины в направлениях, параллельных ее сторонам, – вертикальном или горизонтальном (*рис. 5.7*).

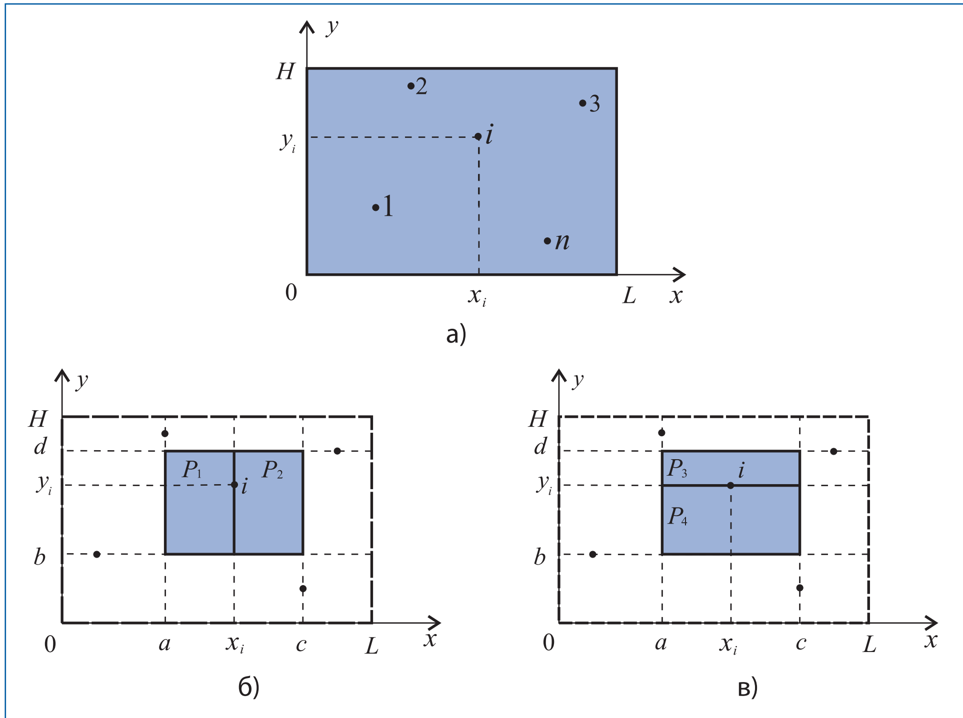


Рис. 5.7. Разрезка пластины максимальной площади: *а* – исходная пластина; *б* – разрезка по вертикали; *в* – разрезка по горизонтали

Решение. Зададим текущую пластину с помощью вектора $P=(a, b, c, d)$, где a и b – это координаты левого нижнего угла, а c и d – координаты правого верхнего угла. Очевидно, что исходная пластина определяется вектором $(0, 0, L, H)$. Метод "*разделяй и властвуй*" может быть реализован следующим образом:

– сначала положим максимальную площадь $S_{max}= 0$;

– если текущая пластина не содержит отверстий, то задача может быть решена непосредственно путем сравнения текущей площади со значением S_{max};

– в противном случае, выбираем произвольное отверстие (x_i, y_i), через которое разрезаем текущую пластину на меньшие части, показанные на *рис 5.7*:

$$P_1=(a, b, x_i, d),\ P_2=(x_i, b, c, d) \quad \text{или} \quad P_3=(a, y_i, c, d),\ P_4=(a, b, c, y_i);$$

– далее, таким же образом рассматриваем каждую из пластин, полученных в результате разрезки, запоминая последовательно в переменной S_{max} наибольшую из площадей текущих пластин.

```
Program P156;
 { Разрезка пластины методом разделяй и властвуй }
const nmax=100;
var L, H : real;
```

```
    n : 1..nmax;
    X,Y : array[1..nmax] of real;
    Smax, amax, bmax, cmax, dmax : real;
    i : integer;

function SolutieDirecta(a, b, c, d : real;
                        var i : integer) : boolean;
label 1;
var j : integer;
begin
  SolutieDirecta:=true;
  for j:=1 to n do
    if (X[j]>a) and (X[j]<c) and (Y[j]>b) and (Y[j]<d) then
      begin
        SolutieDirecta:=false;
        i:=j;
        goto 1;
      end;
1:end; { SolutieDirecta }

procedure PrelucrareaSolutiei(a, b, c, d : real);
var S : real;
begin
  S:=(c-a)*(d-b);
  if S>=Smax then
    begin
      Smax:=S;
      amax:=a; bmax:=b; cmax:=c; dmax:=d;
    end;
end; { PrelucrareaSolutiei }

procedure DesparteSiStapineste(a, b, c, d : real);
var i : integer;
begin
  writeln('Рассматривается пластина (', a:5:1,' ', b:5:1, ' ',
                               c:5:1, ' ', d:5:1, ')');
  readln;
 if SolutieDirecta(a, b, c, d, i)
     then PrelucrareaSolutiei(a, b, c, d)
    else begin
           DesparteSiStapineste(a, b, X[i], d);
           DesparteSiStapineste(X[i], b, c, d);
           DesparteSiStapineste(a, Y[i], c, d);
           DesparteSiStapineste(a, b, c, Y[i]);
         end;
end; { DesparteSiStapineste }
```

```
begin
  writeln('Введите размеры L, H'); readln(L, H);
  write('Введите n='); readln(n);
  writeln('Введите координаты X[i], Y[i]');
  for i:=1 to n do read(X[i], Y[i]);
  writeln;
  Smax:=0;
  DesparteSiStapineste(0, 0, L, H);
  writeln('Пластина максимальной площади (',
          amax:5:1, ' ', bmax:5:1, ' ',
          cmax:5:1, ' ', dmax:5:1, ')');
  writeln('Smax=', Smax:5:2);
  readln;
end.
```

Функция `SolutieDirecta` из программы `P156` возвращает значение `true`, если в пластине (a, b, c d) нет отверстий и `false` – в противном случае. В случае значения `false`, функция дополнительно возвращает индекс i одного из отверстий пластины. Указанное отверстие выбирается исходя из необходимости соблюдения условий $a<x_i<c$ и $b<y_i<d$. Процедура `PrelucrareaSolutiei` сравнивает площадь текущей пластины $S=(c-a)(d-b)$ со значенем S_{max}. Если $S \geq S_{max}$, процедура запоминает текущую пластину в векторе (a_{max}, b_{max}, c_{max}, d_{max}).

Временная сложность алгоритмов, основанных на методе *"разделяй и властвуй"*, зависит от количества подзадач k, на которое делится текущая задача, а также от сложности алгоритмов непосредственного решения соответствующих подзадач. Доказано, что в большинстве случаев порядок времени выполнения алгоритмов типа *"разделяй и властвуй"* составляет $n \log_2 n$ или $n^2 \log_2 n$, т.е. является полиномиальным.

Программы, разработанные на основе метода *"разделяй и властвуй"*, просты, а время их выполнения относительно невелико. К сожалению, этот метод не универсален, поскольку может быть применен только в тех случаях, когда допустимо разделение текущей задачи на подзадачи меньших размеров. Обычно это свойство не указано явно в условии задачи, и нахождение его, если это возможно, является задачей программиста.

Вопросы и упражнения

❶ Объясните структуру алгоритмов, основанных на методе *"разделяй и властвуй"*.

❷ Каковы достоинства и недостатки метода *"разделяй и властвуй"*?

❸ Используя метод *"разделяй и властвуй"*, напишите программу, которая определяет сумму элементов множества $A=\{a_1, a_2, ..., a_n\}$, состоящего из n вещественных чисел.

❹ Укажите на рисунке, аналогичном тому, что изображен на *рис. 5.7*, порядок просмотра пластин в ходе выполнения программы `P156`:
– начальные размеры пластины: 3 × 4;
– количество отверстий: 3;
– координаты отверстий: (1, 1); (1, 2); (2, 2).

❺ Оцените расход памяти и временную сложность программ `P155` и `P156`.

❻ Разработайте алгоритм, который решает задачу разрезки произвольной пластины максимальной площади методом полного перебора. Оцените временную сложность и требуемый объем памяти разработанного алгоритма. Проведите сравнительный анализ алгоритмов, основанных на методе полного перебора, и алгоритмов, основанных на методе “*разделяй и властвуй*”.

❼ **Двоичный поиск.** Рассмотрим множество $A=\{a_1, a_2, ..., a_n\}$, элементами которого являются упорядоченные в порядке возрастания целые числа. Напишите программу, которая определяет, содержит ли множество A заданное число p. Оцените временную сложность написанной программы.

❽ Используя метод “*разделяй и властвуй*”, напишите программу, которая находит наибольший общий делитель натуральных чисел $a_1, a_2, ..., a_n$.

❾ **Сортировка слиянием.** Напишите программу, которая сортирует элементы последовательности $(a_1, a_2, ..., a_n)$ в порядке возрастания в соответствии со следующим алгоритмом:

– делим текущую последовательность на две подпоследовательности приблизительно одинаковой длины;

– если подпоследовательность содержит только два элемента, то упорядочиваем их в порядке возрастания;

– имея две отсортированные подпоследовательности, объединяем (“сливаем”) их в общую отсортированную последовательность.

Предполагается, что элементами исходной последовательности являются целые числа. Например, в результате объединения отсортированных подпоследовательностей (–3, 4, 18) и (–2, –1, 15), получаем (–3, –2, –1, 4, 15, 18).

❿ **Задача о Ханойских башнях.*** Рассмотрим три стержня, обозначенные 1, 2 и 3, а также n дисков различного размера с центральными отверстиями (*рис. 5.8*). Вначале все диски нанизаны на стержень 1, в порядке уменьшения диаметров от основания к вершине стержня. Напишите программу, которая вычисляет последовательность переноса всех дисков на стержень 2 с использованием стержня 3 как вспомогательного и соблюдением следующих правил:

– за один шаг переносится ровно один диск;

– любой диск можно класть только на диск большего размера.

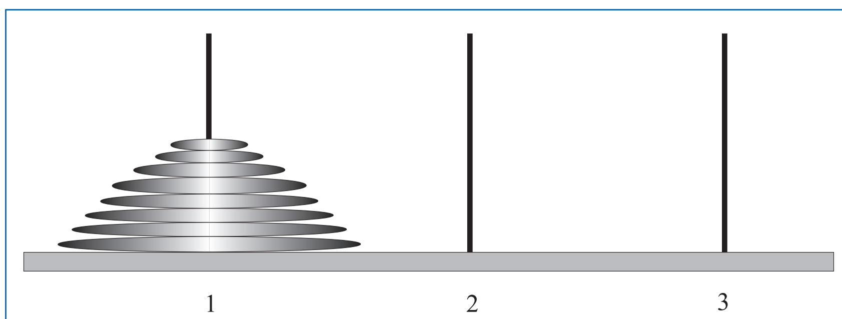


Рис. 5.8. Задача о Ханойских башнях

* Название задачи происходит от древней легенды, согласно которой, после перемещения всех 64 дисков, наступит конец света.

Указания. Перемещение произвольного диска со стержня i на j может быть представлено в виде пары (i, j), где $i, j \in \{1, 2, 3\}$, $i \neq j$. Через $H(m, i, j)$ обозначим последовательность перемещений, необходимых для переноса первых m дисков (естественно, тех, что расположены сверху) со стержня i на стержень j. Например,

$H(1, 1, 2)=(1, 2)$;
$H(2, 1, 2)=(1, 3), (1, 2), (3, 2)$;
$H(3, 1, 2)=(1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2)$.

В общем случае,

$$H(m, i, j) = \begin{cases} (i, j), \text{ для } m = 1; \\ H(m-1, i, k), (i, j), H(m-1, k, j), \text{ для } m > 1, \end{cases}$$

где $k=6-i-j$. Следовательно, задача для n дисков сводится к решению двух подзадач того же типа для $(n-1)$ диска.

5.6. Метод динамического программирования

Для задач, общее решение которых может быть получено как результат решений некоторого ряда подзадач $(d_1, d_2, \ldots, d_p, \ldots, d_q)$, применяется метод динамического программирования.

Каждое решение d_p должно являться **локальным решением,** которое оптимизировало бы некоторый **глобальный критерий качества**, например, стоимость путешествия, длину пройденного пути, массу перевезенного груза, место, занимаемое файлом на диске и т.п. Для того чтобы данный метод был применим, необходимо, чтобы решаемая задача отвечала **принципу оптимальности**: если $(d_1, d_2, \ldots, d_p, d_{p+1}, \ldots, d_q)$ – оптимальный ряд принимаемых решений, то и ряды $(d_1, d_2, \ldots, d_p)$ и $(d_{p+1}, \ldots, d_q)$ должны быть оптимальными.

Например, если кратчайшая дорога от *Кишинэу* до *Бэлць* проходит через *Орхей*, то и оба участка этой дороги – *Кишинэу–Орхей* и *Орхей–Бэлць* – также должны быть самыми короткими. Следовательно, задачи нахождения кратчайшего пути удовлетворяют принципу оптимальности.

В ПАСКАЛЕ метод динамического программирования может быть реализован с помощью массивов, элементы которых вычисляются при помощи определенных **рекуррентных соотношений**. В общем случае, рекуррентные соотношения бывают следующих двух типов:

1) Каждое принимаемое решение d_p зависит от решений $d_{p+1}, \ldots, d_q$. Будем говорить, что в этом случае применяется **метод “вперед”**. В этом методе решения будут приниматься в порядке $d_q, d_{q-1}, \ldots, d_1$.

2) Каждое принимаемое решение d_p зависит от решений $d_1, \ldots, d_{p-1}$. Будем говорить, что в этом случае применяется **метод “назад”**. В этом методе решения будут приниматься в порядке $d_1, d_2, \ldots, d_q$.

Очевидно, что для каждой задачи программист должен проверять в первую очередь соблюдение принципа оптимальности и, в случае положительного ответа, вывести соответствующие рекуррентные соотношения. В противном случае, рассматриваемая задача не может быть решена с помощью метода динамического программирования.

Пример. План некоторого золотоносного участка прямоугольной формы размерами $n \times m$ состоит из квадратных зон с длиной стороны, равной 1 (*рис. 5.9*). В зоне, расположенной в северо-западном углу, находится робот. Из зоны с координатами (i, j), $1 \le i \le n$, $1 \le j \le m$, робот может добыть не более a_{ij} граммов золота. По технологическим соображениям, на участке существуют ограничения: на каждом шаге робот может переместиться из текущей зоны только в одну из соседних зон восточнее или южнее. Напишите программу, которая находит максимальное количество золота C_{max} , которое может добыть робот.

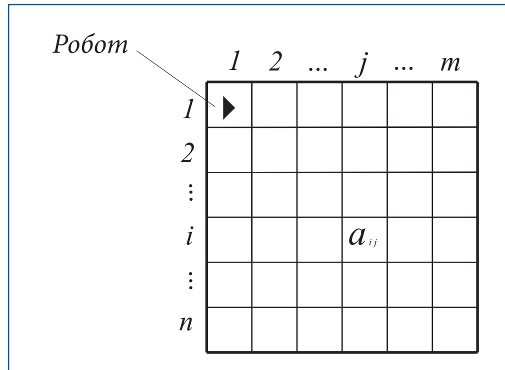


Рис. 5.9. Зоны золотоносного участка

Решение. В компьютере информация о золотоносном участке может быть представлена с помощью двумерного массива $A = ||a_{ij}||_{n \times m}$, где a_{ij} – это количество золота, которое можно добыть в зоне с координатами (i, j). В качестве примера далее рассматривается золотоносный участок, описанный с помощью приведенного ниже массива:

A =

	1	2	3	4	5
1	1	2	3	4	5
2	3	4	6	4	6
3	6	2	7	7	5
4	7	2	3	4	5

Для систематизации вычислений введем в рассмотрение двумерный массив $C = ||c_{ij}||_{n \times m}$, элементами c_{ij} которого будет служить максимальное количество золота, которое может добыть робот, переместившись из начальной зоны (1,1) в зону с координатами (i, j). Очевидно, что $C_{max} = c_{nm}$.

В соответствии с условиями задачи, в каждую зону (i, j) можно войти только из соседних зон: из западной с координатами $(i, j-1)$ либо из северной с координатами $(i-1, j)$. Следовательно, максимальное количество золота, которое может собрать робот достигнув зоны (i, j), вычисляется по рекуррентной формуле:

$$c_{ij} = a_{ij} + max(c_{i,j-1}, c_{i-1,j}).$$

Порядок вычисления элементов c_{ij} массива C определяется способом перемещения робота, а именно:

шаг 1: c_{11};
шаг 2: c_{21}, c_{12};
шаг 3: c_{31}, c_{22}, c_{13};
...
шаг $n+m-1$: c_{nm}.

Отметим, что на шаге k будут вычислены только те элементы c_{ij} массива C, для которых выполняется равенство $i+j-1=k$. Порядок, в котором вычисляются элементы c_{ij} в случае массива A, представлен на *рис. 5.10*.

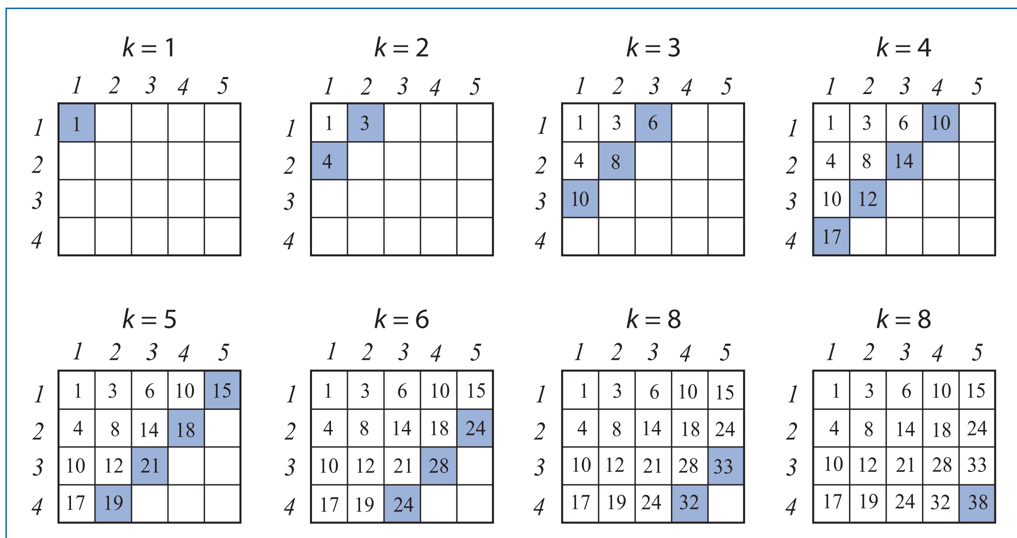


Рис. 5.10. Вычисление элементов массива C

Для того чтобы избежать проверки, связанной с вычислением индексов i и j, в приведенной ниже программе элементы c_{ij} из первой строки и первого столбца вычисляются отдельно от остальных компонент массива C.

```
Program P157;
 { Перемещение робота по золотоносному участку }
var A , C : array [1..50, 1..50] of real;
    m, n, i, j, k : integer;

function Max(a, b : real) : real;
begin
  if a>b then Max:=a else Max:=b;
end; { Max }

begin
  write('Введите значения n, m:  '); readln(n, m);
  writeln('Введите элементы массива A');
  for i:=1 to n do
    for j:=1 to m do read(A[i,j]);
  writeln;
```

```
  C[1,1]:=A[1,1];
   for i:=2 to n do C[i,1]:=A[i,1]+C[i-1,1];
   for j:=2 to m do C[1,j]:=A[1,j]+C[1,j-1];
   for k:=2 to n+m-1 do
     for i:=2 to n do
       for j:=2 to m do
         if (i+j-1)=k then
           C[i,j]:=A[i,j]+Max(C[i,j-1], C[i-1,j]);
   writeln('Cmax=', C[n,m]);
   readln;
 end.
```

Временная сложность алгоритмов, основанных на методе динамического программирования, зависит от характера рекуррентных соотношений и в большинстве случаев полиномиальна. Например, программа P157 имеет порядок сложности $O(n^3)$.

Вопросы и упражнения

❶ Объясните сущность принципа оптимальности.

❷ В каком порядке будут приниматься частичные решения в процессе решения задачи методом динамического программирования?

❸ Объясните порядок принятия решений в задаче перемещения робота по золотоносному участку.

❹ Докажите, что задача о перемещении робота по золотоносному участку удовлетворяет принципу оптимальности.

❺ Оцените расход памяти и время выполнения программы P157.

❻ Как Вы считаете, каковы достоинства и недостатки алгоритмов, основанных на методе динамического программирования?

❼ **Дискретная задача о рюкзаке.** Имеется рюкзак, в котором можно переносить один и более предметов, суммарный вес которых не превосходит значения G_{max}. Для каждого предмета i (i=1, 2, ..., n) известен вес g_i и прибыль c_i, которая получается при его перевозке. Напишите программу, которая определяет, каким образом необходимо загрузить рюкзак с тем, чтобы суммарная прибыль C была максимальной. Предметы не могут быть разделены на меньшие части.

❽ **Путь наименьшей стоимости.** Дана сеть из n городов. Между некоторыми городами есть прямые автобусные маршруты. Стоимость прямого проезда из города i в город j составляет c_{ij} леев. Очевидно, что стоимость проезда прямым маршрутом всегда меньше, чем стоимость проезда с пересадкой: $c_{ij} < c_{ik} + c_{kj}$. Напишите программу, которая вычисляет минимальную стоимость проезда из города i в город j.

❾ **Архивирование файлов.** Известно, что на внешних носителях информации любой файл хранится как последовательность двоичных цифр. Для экономии места программы архивирования разбивают исходный файл на последовательность двоичных слов, принадлежащих определенному словарю, и записывают на диск только порядковые номера соответствующих слов. Например, в случае словаря, состоящего из пяти двоичных слов:

1) `0`; 4) `110`;
2) `1`; 5) `1101101`,
3) `000`;

исходный файл `10001101101110` будет записан на диск в виде `2354`. Напишите программу, которая разбивает любую двоичную последовательность в минимальное количество слов, принадлежащих заданному словарю.

⑩ **Триангуляция многоугольников.** Обработка изображений с помощью компьютеров предполагает разложение многоугольников на более простые геометрические фигуры, а именно на треугольники. Допустим, что выпуклый многоугольник $P_1P_2...P_n$ определен координатами (x_i, y_i) его вершин P_i, $i=1, 2, ..., n$. Напишите программу, которая разбивает многоугольник $P_1P_2...P_n$ на треугольники, проводя с этой целью набор диагоналей P_jP_m, $j \neq m$, $j, m \in \{1, 2, ..., n\}$, которые не пересекаются между собой. Требуется, чтобы суммарная длина диагоналей была минимальна.

5.7. Метод *ветвей и границ*

В методе *ветвей и границ* (на английском – *branch and bound*) поиск решения задачи осуществляется путем прохода некоторого дерева порядка m, именуемого **деревом решений**. Узлы этого дерева представляют множество возможных состояний $S = \{s_1, s_2, ..., s_n\}$, в которых может находиться изучаемая система. Например, такими состояниями могут быть: расположение фигур на шахматной доске, положение автомобиля на карте национальных дорог, положение головки записи-чтения накопителя на магнитных дисках. На дереве решений связи s_i–s_j типа *предок–потомок* указывают на то, что из состояния s_i система может непосредственно перейти в состояние s_j в результате относительно простых преобразований или операций (*рис. 5.11*). В качестве примера вспомним перемещение фигур игроками в ходе шахматной партии, переезд автомобиля из одной местности в другую, выполнение отдельной команды записи-чтения и т.п. Очевидно, в подобных случаях решение задачи может быть представлено как путь, связывающий корень дерева с одним из терминальных узлов и оптимизирующий определенный **критерий качества**, например, количество ходов в шахматной партии, общее расстояние, которое проехал автомобиль, суммарное время чтения-записи файла и т.п.

Формально задачи рассматриваемого типа могут быть решены методом полного перебора, для чего осуществляются следующие операции:

– обходим дерево решений в глубину или в ширину и находим конечные состояния системы;

– строим множество всех возможных путей, соединяющих начальное состояние с конечными состояниями;

– из множества полученных путей выбираем те, которые оптимизируют заданный критерий качества.

Поскольку полный обход дерева требует очень большого времени, с целью уменьшения числа посещаемых узлов, в методе *ветвей и границ* порядок обхода выбирается в зависимости от текущих значений специальной функции $f : S \to R$, называемой **функцией стоимости**. Указанная функция определена на множестве узлов дерева решений, причем конкретное значение $f(s_i)$ характери-

зует “затраты”, необходимые для того, чтобы перейти из текущего сотояния s_i в одно из конечных состояний соответствующего поддерева с корнем s_i.

Посещение узлов в порядке, задаваемом функцией стоимости, называется **обходом минимальной стоимости**. Такой обход осуществляется путем запоминания активных узлов в списке, из которого на каждом шаге извлекается узел минимальной стоимости. Для сравнения вспомним, что при обходе деревьев в ширину или глубину активные узлы заносятся в очередь или соответственно в стек.

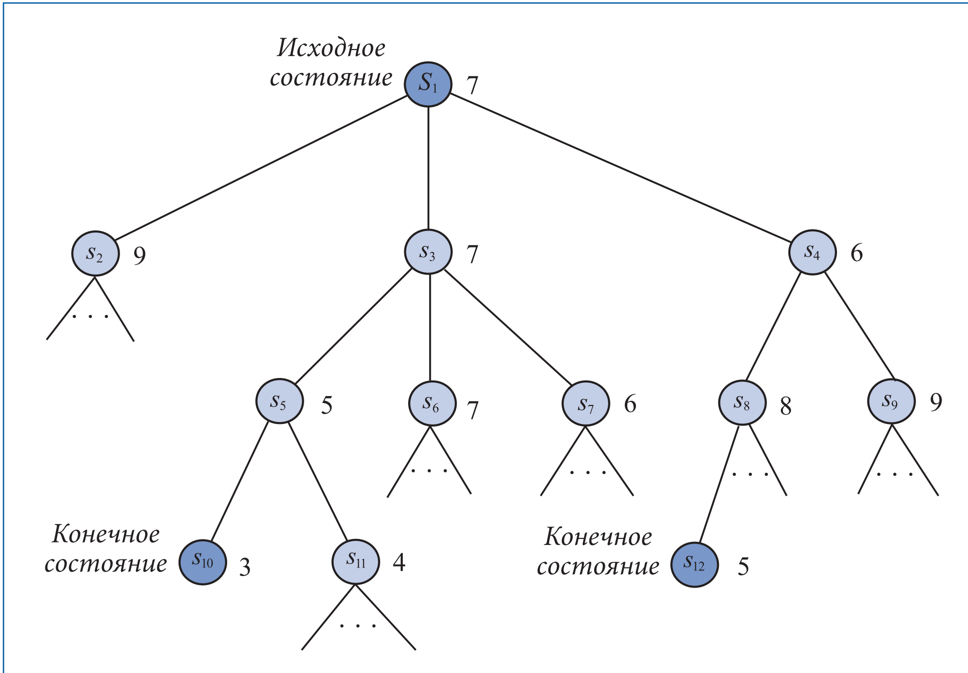


Рис. 5.11. Дерево решений

В случае, когда функция стоимости известна, метод *ветвей и границ* предполагает выполнение следующих операций:

1. *Начальный этап*: создается список активных узлов. Сначала этот список содержит только один элемент – корень дерева решений.

2. *Этап ветвей*: из списка активных узлов извлекается узел минимальной стоимости и генерируются все его потомки.

3. *Этап границ*: для каждого потомка вычисляется значение функции стоимости. Потомки, стоимость которых неприемлема (такие поддеревья, как правило, не содержат конечных состояний), исключаются из дальнейшего рассмотрения. Оставшиеся потомки включаются в список активных узлов.

Этапы 2 и 3 повторяются до тех пор, пока не будет достигнуто одно из конечных состояний, или список активных узлов не опустеет. В последнем случае предполагается, что решение задачи не найдено.

Для примера, в *таблице 5.2* представлен порядок, в котором посещаются узлы дерева, изображенного на *рис. 5.11*. Значения функции стоимости показаны непосредственно на рисунке возле соответствующих узлов. Считается, что потомок, стоимость которого превышает значение 9, является неприемлемым.

Таблица 5.2

Обход минимальной стоимости

№	*Этап*	*Список активных узлов*	*Узел минимальной стоимости*	*Потомки узла минимальной стоимости*
1.	начальный	(s_1)	-	-
2.	ветвей	$\varnothing$	s_1	s_2, s_3, s_4
3.	границ	(s_3, s_4)	-	-
4.	ветвей	(s_3)	s_4	s_8, s_9
5.	границ	(s_3, s_8)	-	-
6.	ветвей	(s_8)	s_3	s_5, s_6, s_7
7.	границ	(s_8, s_5, s_6, s_7)	-	-
8.	ветвей	(s_8, s_6, s_7)	s_5	s_{10}, s_{11}
9.	границ	$(s_8, s_6, s_7, s_{10}, s_{11})$	-	-
10.	конечный	(s_8, s_6, s_7, s_{11})	s_{10}	-

В случае дерева, изображенного на *рис. 5.11,* обход минимальной стоимости заканчивается, когда в списке активных узлов появляется конечное состояние s_{10}. Поскольку любой путь, связывающий конечное состояние с начальным, может быть построен отслеживая связи типа потомок–предок, то решением задачи для рассматриваемого примера является путь (s_1, s_3, s_5, s_{10}).

Схема вычислений метода *ветвей и границ* может быть изображена следующей последовательностью операторов ПАСКАЛЯ:

```
InițializareaListei;
repeat
  Ramifica;
  Margineste;
until Gasit or ListaEsteVida
```

Сложность алгоритмов, основанных на методе *ветвей и границ*, зависит от того, как определена функция стоимости $f: S \to R$. В случае неудачного определения потребуется посещение всех узлов дерева решений, причем временная сложность соответствующего алгоритма будет такой же, как и для алгоритмов с полным перебором. Напротив, удачный выбор функции стоимости позволяет исключить из рассмотрения поддеревья, не содержащие требуемые конечные состояния. В углубленных курсах информатики доказывается, что "хорошее" определение функции стоимости может быть представлено в форме:

$$f(s_i) = niv(s_i) + h(s_i),$$

где $niv(s_i)$ – это уровень узла s_i, а функция $h(s_i)$ оценивает сложность преобразований (операций), необходимых для того, чтобы из текущего состояния s_i достичь одно из конечных состояний поддерева с корнем s_i.

Например, в случае шахматной партии функция $h(s_i)$ может выражать количество сильных фигур противника, для автомобиля – это расстояние до пункта назначения, а в случае накопителя на магнитных дисках – время, необходимое для чтения или записи нужной информации.

Отметим, что метод *ветвей и границ* применим лишь тогда, когда текущие значения $f(s_i)$ функции стоимости могут быть вычислены без предварительного построения поддеревьев с корнем s_i. К сожалению, к настоящему времени неизвестны универсальные правила, позволяющие по типу заданной задачи находить однозначные определения функций стоимости. Следовательно, сложность алгоритмов, основанных на методе *ветвей и границ*, зависит в значительной степени от опыта и умения программиста.

Вопросы и упражнения

❶ Для какого типа задач может быть применен метод *ветвей и границ*?

❷ Как строится дерево решений? Какую информацию содержит каждый узел этого дерева?

❸ Как представляется решение рассматриваемой задачи в методе *ветвей и границ*?

❹ Для чего предназначена функция стоимости? Как определяется такая функция?

❺ Объясните вычислительную схему метода *ветвей и границ*.

❻ Напишите ПАСКАЛЬ-процедуру, которая реализует обход минимальной стоимости. Считается, что каждый узел дерева содержит поле, в котором указано текущее значение функции стоимости. Оцените расход памяти и временную сложность разработанной процедуры.

❼ Укажите порядок, в котором будут посещены узлы дерева на *рис. 5.11*, в следующих случаях:
а) обход в ширину;
б) обход в глубину;
в) обход минимальной стоимости.

❽ Как Вы считаете, посещаются ли все узлы дерева решений в случае обхода минимальной стоимости? Обоснуйте Ваш ответ.

❾ Набросайте алгоритм, который ищет оптимальное решение путем обхода дерева решений в ширину (в глубину). Оцените расход памяти и временную сложность разработанного алгоритма.

❿ Приведите примеры задач, которые могут быть решены с помощью метода *ветвей и границ*.

⓫ Попробуйте нарисовать узлы уровней 0, 1 и 2 дерева решений, представляющего Вашу любимую игру, например, морской бой, шахматы, шашки и т.п.

⓬ Как Вы считаете, какими преимуществами и недостатками обладают алгоритмы, основанные на методе *ветвей и границ*?

5.8. Применения метода *ветвей и границ*

Метод *ветвей и границ* может быть применен для решения тех задач, из условия которых следуют правила или операции, необходимые для непосредственного построения дерева решений. Обычно в таких задачах моделируется поведение воображаемого партнера (противника), цели или намерения которого

противоположны Вашим, например: военные игры, сценарии экономического развития в условиях конкуренции, игра в шахматы или в шашки и т.п. Отметим, что существование однозначных правил непосредственного генерирования потомков узлов дерева решений делает ненужным явное построение самого дерева решений, что ведет к уменьшению требуемого объема внутренней памяти.

Практическая реализация метода *ветвей и границ* предполагает решение следующих задач:

– определение множества состояний системы, которое, в зависимости от природы (характера) решаемой задачи, может быть конечным или бесконечным;

– установление элементарных преобразований (операций), в результате которых система переходит из одного состояния в другое;

– определение функции стоимости;

– определение структур данных, необходимых для представления состояний системы, дерева решений и списка активных узлов;

– разработка подпрограмм, необходимых для построения потомков узлов дерева решений, вычисления функции стоимости, выбора узлов с минимальной стоимостью и т.д.

В качестве примера рассмотрим реализацию метода *ветвей и границ* в случае **игры Perspico**, известной также под названием **игра 15**. В этой игре имеются 15 квадратных пластинок, пронумерованных от 1 до 15 (*рис. 5.12*).

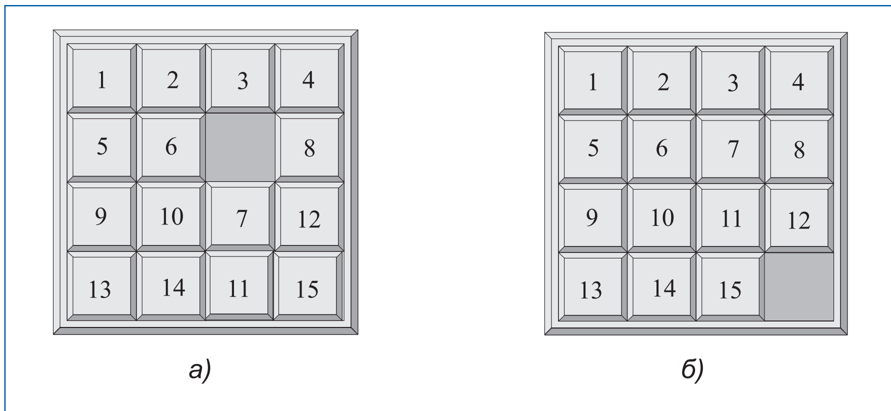


Рис. 5.12. Игра Perspico: *а* – начальное состояние;
б – конечное состояние

Пластинки уложены в квадратную коробочку размерами 4×4, одно место в которой свободно. Любая пластинка, соседствующая со свободным местом, может быть перемещена на него. Необходимо перейти, используя только разрешенные ходы, от заданного начального состояния к требуемому конечному состоянию пластинок.

Текущее состояние игры Perspico может быть выражено распределением 15 пластинок на 16 свободных местах. Пронумеровав свободное место числом 0, можно определить тип данных, описывающих состояние системы:

```
type Stare = array [1..4, 1..4] of 0..15;
```

Отметим, что количество возможных состояний системы задается числом размещений 16 из 16 и составляет:

$$A_{16}^{16} = P_{16} = 16! \approx 2,09 \cdot 10^{13},$$

в то время как объем внутренней памяти персональных компьютеров составляет всего порядка 10^8 байтов.

Переход из одного состояния в другое может быть осуществлен путем “перемещения“ отсутствующей пластинки влево, вверх, вправо или вниз (естественно, лишь тогда, когда соответствующее перемещение возможно). В ПАСКАЛЕ рассматриваемые “перемещения“ описываются путем изменения соответствующих элементов переменных типа `Stare`.

Зная состояния системы и правила перехода из одного состояния в другое, можно построить **дерево решений**, часть которого представлена на *рис. 5.13*.

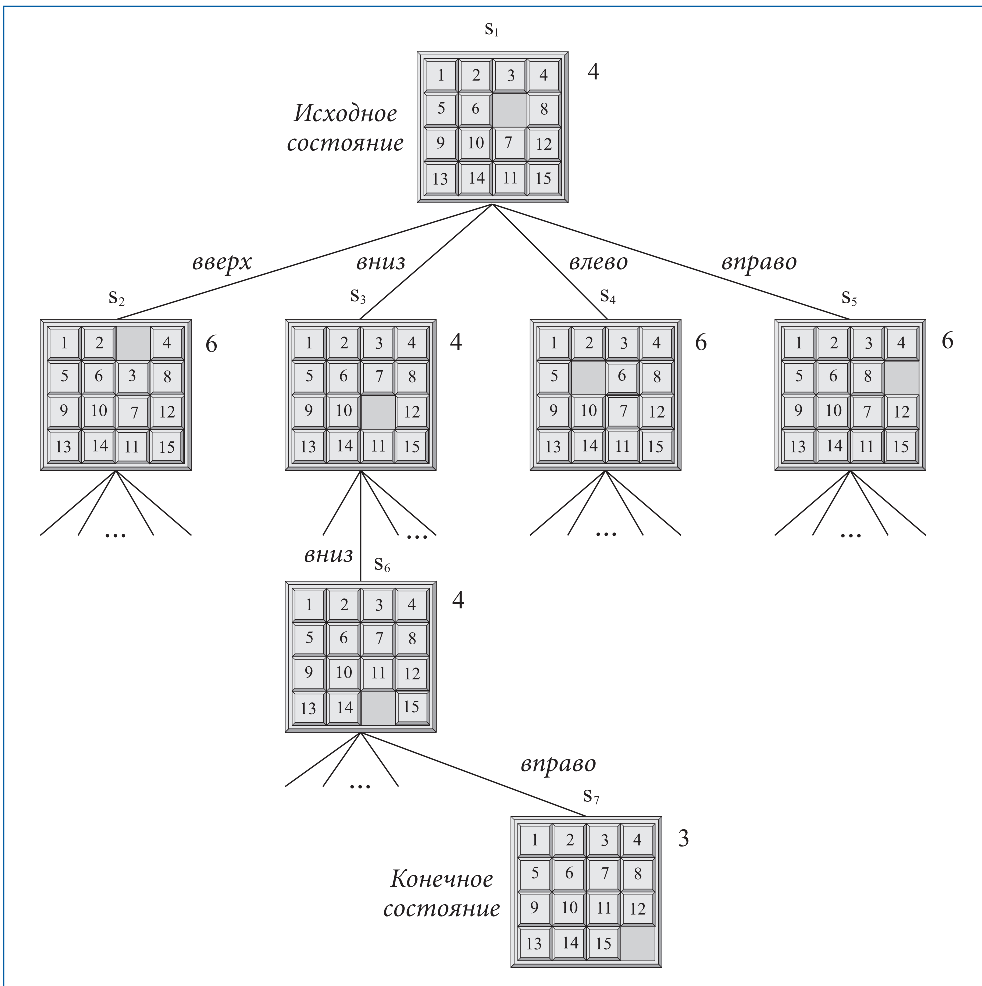


Рис. 5.13. Дерево решений в игре Perspico

Поскольку свободную пластинку можно перемещать и по циклическому маршруту, в общем случае дерево решений является бесконечным. Поэтому,

для того чтобы избежать возврата к уже ранее рассмотренным состояниям, будем включать в дерево решений только новые текущие состояния. Также принимая во внимание, что внутренняя память персональных компьютеров не позволяет сохранять все возможные состояния, в дальнейшем будем рассматривать только первые 10–15 уровней дерева решений. Очевидно, при введенных ограничениях больше не гарантируется нахождение последовательности перемещений, переводящих систему из начального состояния в конечное, даже если такая последовательность и существует.

Функция стоимости $f : S \to R$ может быть определена в виде:

$$f(s_i) = niv(s_i) + h(s_i),$$

где член $h(s_i)$ характеризует количество перестановок, необходимых для перехода из текущего состояния s_i в конечное состояние. Поскольку это число не может быть вычислено без предварительного построения деревьев с корнем s_i, будем использовать нижнюю оценку требуемого числа перемещений:

$h(s_i)$ – число пластинок, которые не находятся на своих местах.

Например, для дерева на *рис. 17* имеем:

$$f(s_1) = niv(s_1) + h(s_1) = 0 + 4 = 4;$$
$$f(s_2) = niv(s_2) + h(s_2) = 1 + 5 = 6;$$
$$f(s_3) = niv(s_3) + h(s_3) = 1 + 3 = 4;$$
$$f(s_6) = niv(s_6) + h(s_6) = 2 + 2 = 4;$$
$$f(s_7) = niv(s_7) + h(s_7) = 3 + 0 = 3.$$

Дерево решений и список активных узлов можно представить в памяти компьютера следующими структурами данных:

```
type AdresaNod = ^Nod;
     Nod = record
             S : Stare;
             Nivel, Cost : integer;
             Drum : boolen;
             D : array [1..4] of AdresaNod;
             Tata : AdresaNod;
           end;
     AdresaCelula = ^Celula;
     Celula = record
                ReferintaNod : AdresaNod;
                Urm : AdresaCelula;
              end;
var Radacina : AdresaNod;
    BazaListei : AdresaCelula;
```

Каждый узел дерева решений содержит информационные поля `S`, `Nivel`, `Cost`, `Drum` и поля связи `D`, `Tata`. Поле `Drum` используется для того, чтобы отмечать узлы, принадлежащие пути, связывающему корень с узлом, содержащим конечное состояние. Поле `Tata` используется, чтобы хранить адрес узла-предка.

Каждая ячейка списка активных узлов содержит информационное поле `ReferintaNod`, в которое заносится адрес узла, включенного в список. Поле связи `Urm` содержит адрес следующей ячейки списка. Следовательно, список активных узлов содержит не сами узлы, а их адреса.

Подпрограммы, необходимые для реализации метода *ветвей и границ* для игры Perspico, включены в программу `P158`.

```
Program P158;
 { Игра Perspico - метод ветвей и границ }
const NivelMaxim=15;
type Stare=array[1..4, 1..4] of 0..15;
     AdresaNod=^Nod;
     Nod=record
           S : Stare;
           Nivel, Cost : integer;
           Drum : boolean;
           D : array [1..4] of AdresaNod; {адреса потомков}
           Tata : AdresaNod;
         end;
     AdresaCelula=^Celula;
     Celula=record
              ReferintaNod : AdresaNod;
              Urm : AdresaCelula;
            end;
var Radacina : AdresaNod;
    BazaListei : AdresaCelula;
    StareaInitiala, StareaFinala : Stare;
    Gasit : boolean;  { true когда найден путь }
    Finput, Foutput : text; { входные и выходные данные }
    Str : array[1..4] of Stare; { следующие состояния }
    m : 0..4;                   { текущее число состояний }
    i, j : integer;

procedure CalculareaCostului(Adresa : AdresaNod);
var i, j, C : integer;
begin
  C:=0;
  for i:=1 to 4 do
    for j:=1 to 4 do
      if Adresa^.S[i,j] <> StareaFinala[i,j] then C:=C+1;
  Adresa^.Cost:=Adresa^.Nivel+C;
end; { CalculareaCostului }

procedure Initializare;
 { Создает корневой узел и список активных узлов }
 { Заносит в список активных узлов корневой узел }
var i : integer;
```

```
begin
  Gasit:=false;
  new(Radacina);
  Radacina^.S:=StareaInitiala;
  Radacina^.Nivel:=0;
  CalculareaCostului(Radacina);
  Radacina^.Drum:=false;
  for i:=1 to 4 do Radacina^.D[i]:=nil;
  Radacina^.Tata:=nil;
  new(BazaListei);
  BazaListei^.ReferintaNod:=Radacina;
  BazaListei^.Urm:=nil;
end; { Initializare }

procedure Ramifica(Adresa : AdresaNod);
 { Записывает в Str все состояния, которые могут быть получены }
 { из текущего путем осуществления разрешенных перемещений }
label 1;
var   St : Stare;
      i, j, k : integer;
begin
  { поиск пластинки 0 }
  for i:=1 to 4 do
    for j:=1 to 4 do
      if Adresa^.S[i,j]=0 then goto 1;
1: m:=0;
  { Перемещение пластинки сверху }
  if i<>1 then
    begin
      St:=Adresa^.S;
      St[i,j]:=St[i-1, j];
      St[i-1, j]:=0;
      m:=m+1;
      Str[m]:=St;
    end;
  { Перемещение пластинки снизу }
  if i<>4 then
    begin
      St:=Adresa^.S;
      St[i,j]:=St[i+1, j];
      St[i+1, j]:=0;
      m:=m+1;
      Str[m]:=St;
    end;
  { Перемещение пластинки слева }
 if j<>1 then
    begin
      St:=Adresa^.S;
```

```
      St[i,j]:=St[i, j-1];
      St[i, j-1]:=0;
      m:=m+1;
      Str[m]:=St;
    end;
 { Перемещение пластинки справа }
 if j<>4 then
   begin
      St:=Adresa^.S;
      St[i,j]:=St[i, j+1];
      St[i, j+1]:=0;
      m:=m+1;
      Str[m]:=St;
    end;
end; { Desparte }

procedure IncludeInLista(Adresa : AdresaNod);
 { Включает адрес узла в список активных узлов }
var R : AdresaCelula;
begin
  new(R);
  R^.ReferintaNod:=Adresa;
  R^.Urm:=BazaListei;
  BazaListei:=R;
end; { IncludeInLista }

procedure ExtrageDinLista(var Adresa : AdresaNod);
 { Извлечение узла минимальной стоимости из списка }
label 1;
var P, R : AdresaCelula;
    C : integer;  { costul curent }
begin
  if BazaListei=nil then goto 1;
  { поиск узла минимальной стоимости }
  C:=MaxInt;
  R:=BazaListei;
  while R<>nil do
    begin
      if R^.ReferintaNod^.Cost < C then
        begin
          C:=R^.ReferintaNod^.Cost;
          Adresa:=R^.ReferintaNod;
          P:=R;
        end; { then }
      R:=R^.Urm;
    end; { while }
  { исключение узла минимальной стоимости из списка }
```

```
  if P=BazaListei then BazaListei:=P^.Urm
    else
      begin
        R:=BazaListei;
        while P<>R^.Urm do R:=R^.Urm;
        R^.Urm:=P^.Urm;
      end;
    dispose(P);
1:end; { ExtrageDinLista }

function StariEgale(S1, S2 : Stare) : boolean;
 { Возвращает TRUE, если состояния S1 и S2 совпадают }
var i, j : integer;
    Coincid : boolean;
begin
  Coincid:=true;
  for i:=1 to 4 do
    for j:=1 to 4 do
      if S1[i,j]<>S2[i,j] then Coincid:=false;
  StariEgale:=Coincid;
end; { StariEgale }

function StareDejaExaminata(St : Stare) : boolean;
 { Возвращает TRUE, если текущее состояние уже включено
   в дерево }
var EsteInArbore : boolean;

procedure InAdincime(R : AdresaNod);
 { Обход дерева в глубину }
label 1;
var i : integer;
begin
  if R<>nil then
    begin
      if StariEgale(St, R^.S) then
        begin
          EsteInArbore:=true;
          goto 1;
        end;
      for i:=1 to 4 do InAdincime(R^.D[i]);
    end;
1:end; { InAdincime }

begin
  EsteInArbore:=false;
 InAdincime(Radacina);
```

```
  StareDejaExaminata:=EsteInArbore;
end; { StareDejaExaminata }

procedure Margineste(Adresa : AdresaNod);
  { Отбор потомков и включение их в список }
label 1;
var i, k : integer;
    R : AdresaNod;
begin
  k:=0;
  if (Adresa^.Nivel+1) > NivelMaxim then goto 1;
  for i:=1 to m do
    if not StareDejaExaminata(Str[i]) then
      begin
        k:=k+1;
        new(R);
        R^.S:=Str[i];
        R^.Nivel:=Adresa^.Nivel+1;
        CalculareaCostului(R);
        for j:=1 to 4 do R^.D[j]:=nil;
        Adresa^.D[i]:=R;
        R^.Tata:=Adresa;
        R^.Drum:=false;
        if StariEgale(R^.S, StareaFinala) then
             begin
               R^.Drum:=true;
               Gasit:=true;
             end;
        IncludeInLista(R);
      end;
  writeln(Foutput);
  writeln(Foutput, 'В список были включены ', k,
                    'потомка');
  writeln(Foutput);
1:end; { Margineste }

procedure AfisareaNodului(R : AdresaNod);
var i, j : integer;
begin
  writeln(Foutput, Путь=', R^.Drum, ' Уровень=', R^.Nivel,
                   ' Стоимость=', R^.Cost);
  for i:=1 to 4 do
    begin
      for j:=1 to 4 do write(Foutput, R^.S[i, j] : 3);
      writeln(Foutput);
    end;
```

```
    for i:=1 to 4 do
      if R^.D[i]<>nil then write(Foutput, '***  ')
                      else write(Foutput, 'nil  ');
    writeln(Foutput); writeln(Foutput);
end; { AfisareaNodului }

procedure RamificaSiMargineste;
var NodulCurent : AdresaNod;
begin
  Initializare;
  repeat
    ExtrageDinLista(NodulCurent);
    writeln(Foutput, '    УЗЕЛ ИЗВЛЕЧЕННЫЙ ИЗ СПИСКА');
    writeln(Foutput, '    =======================');
    AfisareaNodului(NodulCurent);
    Ramifica(NodulCurent);
    Margineste(NodulCurent);
  until Gasit or (BazaListei=nil);
end; { RamificaSiMargineste }

procedure AfisareaDrumului;
label 1;
var R : AdresaCelula;
    P, Q : AdresaNod;
begin
  if not Gasit then
    begin
      writeln(Foutput, 'ПУТЬ НЕ НАЙДЕН');
      goto 1;
    end;
  writeln(Foutput, '     ПУТЬ НАЙДЕН:');
  writeln(Foutput, '     =============');
  { поиск в списке терминального узла }
  R:=BazaListei;
  while (R<>nil) and (not R^.ReferintaNod^.Drum) do R:=R^.Urm;
  { пометка узлов, образующих путь }
  P:=R^.ReferintaNod;
  while P<>nil do
    begin
      P^.Drum:=true;
      P:=P^.Tata;
    end;
  { вывод пути }
  P:=Radacina;
 while P<>nil do
    begin
      AfisareaNodului(P);
```

```
      Q:=nil;
      for i:=1 to 4 do
        if (P^.D[i]<>nil) and P^.D[i]^.Drum then Q:=P^.D[i];
      P:=Q;
    end;
  writeln(Foutput, 'Конец пути');
1:end; { AfisareaDrumului }

begin
  { Чтение начального состояния }
  assign(Finput, 'FINPUT.TXT');
  reset(Finput);
  for i:=1 to 4 do
    for j:=1 to 4 do read(Finput, StareaInitiala[i, j]);
  { Чтение конечного состояния }
  for i:=1 to 4 do
    for j:=1 to 4 do read(Finput, StareaFinala[i, j]);
  close(Finput);
  { Открытие выходного файла }
  assign(Foutput, 'FOUTPUT.TXT');
  rewrite(Foutput);
  RamificaSiMargineste;
  AfisareaDrumului;
  close(Foutput);
  writeln('Найден=', Gasit);
  readln;
end.
```

В программе P158 начальное и конечное состояния игры считываются из файла INPUT.TXT, а узлы, извлекаемые на каждом шаге из списка, и найденный путь записываются в файл FOUTPUT.TXT. Содержимое этих файлов может быть просмотрено или отредактировано при помощи любого простого текстового редактора.

Вопросы и упражнения

❶ Укажите порядок, в котором будут посещены узлы дерева на *рис. 5.13* в случае обхода минимальной стоимости. Используя в качестве образца *таблицу 5.2*, составьте таблицу с данными об этом дереве.

❷ Объясните назначение каждой подпрограммы из программы P158.

❸ Запустите на выполнение программу P158 для начальных данных, приведенных на *рис. 5.12*. Прокомментируйте результаты, записанные в выходном файле.

❹ Запустите на выполнение программу P158 для следующих начальных состояний игры Perspico:

a)

1	2	0	4
5	6	3	8
9	11	7	12
13	10	14	15

b)

1	2	3	4
5	7	6	8
9	0	10	11
13	14	15	12

c)

1	3	4	8
5	2	6	0
9	10	7	11
13	14	15	12

d)

0	1	2	3
6	7	8	4
5	9	10	11
13	14	15	12

Прокомментируйте результаты, записанные программой в выходном файле `FOUTPUT.TXT`.

❺ Известно, что для приведенных ниже начальных состояний:

a)

1	3	4	4
9	5	7	8
13	6	10	11
0	14	15	12

b)

1	2	3	4
5	0	7	8
9	6	10	11
13	14	15	12

программа `P158` находит соответствующие решения. Но после внесения изменения

```
const NivelMaxim = 5;
```

решение для случая (*a*) больше не находится. Как Вы считаете, с чем это связано?

❻ Рассмотрим упрощенный вариант игры Perspico, называемый **игра 9** (*рис. 5.14*). Какие изменения нужно внести в программу `P158` для того, чтобы вычислить последовательность разрешенных ходов, переводящих игру из начального состояния в конечное?

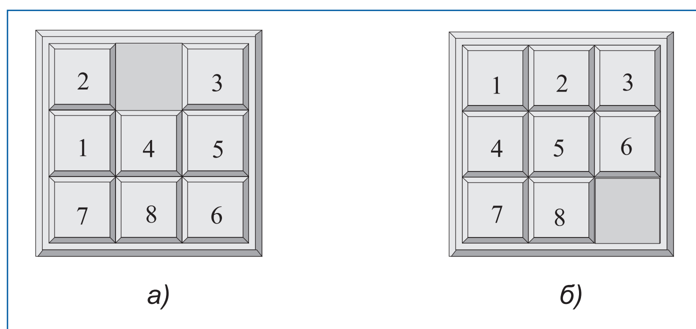


Рис. 5.14. Упрощенный вариант игры Perspico:
а – начальное состояние; *б* – конечное состояние

❼ Измените программу `P158`, используя следующую функцию стоимости:

$$F(s_i) = niv(s_i) + g(s_i),$$

где $g(s_i)$ – сумма расстояний между каждой пластинкой и ее местоположением в заданном конечном состоянии, выраженная числом соответствующих перемещений. Например, для начального состояния на *рис. 5.12* имеем:

$$g(s_i) = 0 + 0 + 0 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 3 = 6.$$

Проверьте, как выполняется измененная программа в случае начальных состояний из упражнений 4 и 5. Разработайте план эксперимента, который позволил бы установить, какая из функций $h(s_i)$, $g(s_i)$ уменьшает время вычислений.

❽ Сравните сложность программы `P158` со сложностью программы, основанной на методе полного перебора.

❾ Дана шахматная доска, на которой находится единственная фигура – белый конь. Напишите программу, которая определяет, можно ли, используя разрешенные ходы, переместить коня из позиции (α, β) в позицию (γ, δ). Напоминаем, что в случае шахматной доски $\alpha, \gamma \in$ {A, B, ..., H}, а $\beta, \delta \in \{1, 2, ..., 8\}$.

❿ **Задача коммивояжера***. Даны *n* городов, между которыми существуют прямые авиалинии. В городе i находится коммивояжер, который должен посетить остальные n–1 городов и вернуться в город, из которого стартовал. Напишите программу, которая по известным расстояниям между городами d_{ij} определит порядок, в котором они должны быть посещены. Требуется, чтобы расстояние, которое преодолел коммивояжер, было как можно короче, а каждый город был бы посещен только один раз.

⓫ Напишите программу, которая решает задачу коммивояжера методом полного перебора. Сравните сложность разработанной программы со сложностью программы, основанной на методе *ветвей и границ*.
Указания: Путь, проделанный коммивояжером может быть представлен с помощью упорядоченного множества городов $(1, i, j, ..., k, 1)$, в котором $i, j, ..., k \in \{2, 3, ..., n\}$. Для генерирования всех возможных путей вычисляются перестановки множества $\{2, 3, ..., n\}$.

⓬ Проведите сопоставительный анализ алгоритмов, базирующихся на методе *ветвей и границ*, и алгоритмов, базирующихся на методе полного перебора.

5.9. Точные и эвристические алгоритмы

Алгоритмы, применяемые для решения задач на компьютере, делятся на две различные категории: точные алгоритмы и эвристические алгоритмы.

Алгоритм называется **точным** только тогда, когда он находит оптимальные решения задач, для которых он был разработан. Конечно, этот факт должен быть подтвержден строгими математическими доказательствами.

Алгоритм является **эвристическим**, когда он находит хорошие, но не обязательно оптимальные решения. В таких случаях соответствующего математического доказательства либо не существует, либо оно не известно. Обычно, эвристические алгоритмы содержат последовательности операций по обработке данных, которые хотя и не обоснованы математически, выбраны на основе опыта и интуиции программиста.

* Человек, который посредничает в продаже товаров, перемещаясь с места на место в поисках прибыли.

Очевидно, что методы, основанные на полном переборе, являются точными, поскольку в ходе последовательного рассмотрения всех возможных решений обязательно будет найдено и оптимальное решение. Подтверждение точности этих алгоритмов сводится к доказательству того факта, что в процессе выполнения программы на компьютере будут сгенерированы и проанализированы все элементы из множества возможных решений. Для алгоритмов, основанных на других методах программирования – методе *Greedy*, методе перебора с возвратом, методе ветвей и границ и т.д., – алгоритм будет точным или эвристическим в зависимости от характера условий, которые помогают избежать полного перебора всех возможных решений. Если эти условия выбраны неудачно, оптимальные решения могут быть потеряны и, как следствие, соответствующий алгоритм не будет точным. Для примера рассмотрим следующую задачу.

Задача нахождения кратчайшего пути. Даны n городов, связанных сетью дорог (*рис. 5.15*). Зная расстояния d_{ij} между соседними городами, определите самый короткий путь из города a в город b.

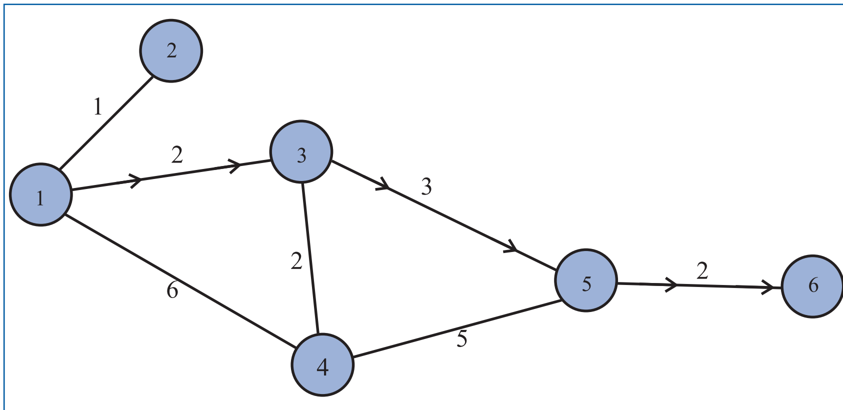


Рис. 5.15. Сеть дорог

Исходные данные рассматриваемой задачи могут быть описаны с помощью матрицы (двумерного массива) $D = ||d_{ij}||_{n \times n}$ из n строк и n столбцов, называемой **матрицей расстояний**. В этой матрице компонента d_{ij} равна расстоянию между городами i, j тогда, когда они являются соседними, и 0 – в противном случае. По определению, $d_{ii} = 0$, $i = 1, 2, ..., n$.

Например, для городов на *рис. 5.15* имеем:

$D =$

	1	*2*	*3*	*4*	*5*	*6*
1	0	1	2	6	0	0
2	1	0	0	0	0	0
3	2	0	0	2	3	0
4	6	0	2	0	5	0
5	0	0	3	5	0	2
6	0	0	0	0	2	0

Кратчайший путь, связывающий города $a = 1$, $b = 6$, имеет длину 7 и включает города 1, 3, 5, 6.

В принципе, кратчайший путь может быть найден с помощью метода **полного перебора,** путем последовательного генерирования всех возможных перестановок упорядоченных множеств

$$X = (a, \underbrace{i, j, \ldots, k,}_{q \text{ городов}} b),$$

где q принимает значения от 0 до n–2. Очевидно, что алгоритмы, основанные на генерировании всех возможных перестановок, всегда находят минимальный путь, однако временная сложность O(n!) таких алгоритмов неприемлема.

Чтобы уменьшить объем вычислений, попробуем найти кратчайший путь методом **перебора с возвратом**. В данном методе построение пути начинается с города $x_1 = a$ и продолжается первым из еще непосещенных соседей, пусть это будет x_2, далее переходим к первому из непосещенных соседей для x_2 и т.д. Чтобы построить кратчайшую дорогу, воспользуемся следующим **интуитивным правилом**: на каждом шаге в первую очередь будем рассматривать непосещенных еще соседей, которые находятся ближе всего к текущему городу.

Текущий путь может быть представлен в виде массива:

$$X = (a, x_2, \ldots, x_{k-1}, x_k, \ldots, b)$$

причем элементом x_k должен быть один из соседей города x_{k-1}. Для систематизации вычислений будем запоминать еще непосещенных соседей города i во множестве A_i , $i = 1, 2, \ldots, n$. Очевидно, в соответствии с интуитивным правилом, сформулированным выше, города из каждого множества A_i должны быть отсортированы в порядке увеличения расстояний d_{ij}, $j \in A_i$.

Например, для *рис. 5.15* перед началом вычислений имеем:

$$A_1 = (2, 3, 4);$$
$$A_2 = (1);$$
$$A_3 = (1, 4, 5);$$
$$A_4 = (3, 5, 6);$$
$$A_5 = (6, 3, 4);$$
$$A_6 = (5).$$

Условия продолжения в методе перебора с возвратом следуют непосредственно из условия задачи: город x_k может быть добавлен к уже построенной части пути $(a, x_2, \ldots, x_{k-1})$ только тогда, когда:

1) x_k является еще непосещенным соседом города x_{k-1}, т.е. $x_k \in A_{k-1}$;

2) в процессе построения город x_k не был включен ранее в путь, т.е. $x_k \neq a$, $x_k \neq x_2$, ..., $x_k \neq x_{k-1}$.

Например, для *рис. 5.15* вектор (массив) X будет последовательно принимать следующие значения:

$$X = (1);$$
$$X = (1, 2);$$
$$X = (1);$$
$$X = (1, 3);$$
$$X = (1, 3, 4);$$

$X = (1, 3, 4, 5);$
$X = (1, 3, 4, 5, 6).$

Путь (1, 3, 4, 5, 6), построенный методом перебора с возвратом, имеет длину 11 и, очевидно, не является минимальным. Однако этот путь лучше пути (1, 4, 5, 6), длина которого составляет 13.

В приведенной ниже программе элементы множеств $A_1, A_2, ..., A_n$ размещены в начале строк `A[1]`, `A[2]`, ..., `A[n]` двумерного массива А, а остальные позиции содержат нулевые значения.

```
Program P159;
 { Задача о кратчайшем пути - метод перебора с возвратом }
const nmax=50;
var n : integer; { число городов }
    D : array[1..nmax, 1..nmax] of real; { матрица расстояний }
    a, b : 1..nmax;
    X : array [1..nmax] of integer;      { построенный путь }
    V : array[1..nmax, 1..nmax] of integer; { соседи }
    Finput : text;

procedure InitializareVecini;
 { Записывает в V[k] соседей города k }
var k, i, j, p, q, r : integer;
begin
  for k:=1 to n do
    begin
      { Сперва множество V[k] пустое }
      for i:=1 to n do V[k,i]:=0;
      { вычисляем элементы множества V[k] }
      i:=0;
      for j:=1 to n do
        if D[k,j]<>0 then
          begin
            i:=i+1;
            V[k,i]:=j;
          end; { then }
      { сортировка элементов множества V[k] методом пузырька }
      for j:=1 to i do
        for p:=1 to i-1 do
          if D[k, V[k,p]]>D[k, V[k, p+1]] then
            begin
              q:=V[k,p];
              V[k,p]:=V[k, p+1];
              V[k, p+1]:=q;
            end; { then }
    end; { for }
end; { InitializareVecini }
```

```
procedure Initializare;
var i, j : integer;
begin
  assign(Finput, 'DRUM.IN');
  reset(Finput);
  readln(Finput, n);
  readln(Finput, a, b);
  writeln('n=', n, '   a=', a, '  b=', b);
    for i:=1 to n do
      for j:=1 to n do read(Finput, D[i,j]);
  close(Finput);
  InitializareVecini;
end; { Initializare }

function MultimeVida(k : integer) : boolean;
begin
  MultimeVida:=(V[X[k-1], 1]=0);
end; { MultimeVida }

function PrimulElement(k : integer) : integer;
var i : integer;
begin
  PrimulElement:=V[X[k-1], 1];
  for i:=1 to n-1 do V[X[k-1],i]:=V[X[k-1], i+1];
end; { PrimulElement }

function ExistaSuccesor(k : integer) : boolean;
begin
  ExistaSuccesor:=(V[X[k-1], 1]<>0);
end; { ExistaSuccesor }

function Succesor(k : Integer) : integer;
var i : integer;
begin
  Succesor:=V[X[k-1], 1];
  for i:=1 to n-1 do V[X[k-1], i]:=V[X[k-1], i+1];
end; { Succesor }

function Continuare(k : integer) : boolean;
var i : integer;
Indicator : boolean;
begin
  Continuare:=true;
  for i:=1 to k-1 do
   if X[i]=X[k] then Continuare:=false;
end; { Continuare }
```

```
procedure PrelucrareaSolutiei(k : integer);
var i : integer;
    s : real;
begin
  writeln('Найденный путь:');
  for i:=1 to k-1 do write(X[i] : 3);
 writeln;
  s:=0;
  for i:=1 to k-1 do s:=s+D[X[i], X[i+1]];
  writeln('Длина пути', s : 10:2);
  readln;
  halt;
end; { PrelucrareaSolutiei }

procedure Reluare(k : integer);
label 1;
var i : integer;
begin
  if MultimeVida(k) then goto 1;
  if X[k-1]<>b then
    begin
      X[k]:=PrimulElement(k);
      if Continuare(k) then Reluare(k+1);
      while ExistaSuccesor(k) do
        begin
          X[k]:=Succesor(k);
          if Continuare(k) then Reluare(k+1);
        end; { while }
    end { then}
  else PrelucrareaSolutiei(k);
1:end; { Reluare }

begin
  Initializare;
  X[1]:=a;
  Reluare(2);
end.
```

Анализ программы P159 показывает, что алгоритм, основанный на методе перебора с возвратом, имеет сложность $O(m^n)$, т.е. является экспоненциальным. В данной формуле m – максимальное из числа соседей каждого города. Отметим, что конкретное количество операций, выполняемых в программе P159, в большой степени зависит от конфигурации сети дорог и соответствующих расстояний. Очевидно, что “платой“ за уменьшение числа операций в эвристическом алгоритме, основанном на методе перебора с возвратом, по отношению к точному алгоритму, основанному на полном переборе, является потеря оптимальных решений.

Для эффективного решения задач реальной практической значимости в информатике тратятся значительные усилия для разработки точных алгоритмов полиномиальной сложности. В задаче нахождения кратчайшего пути такой алгоритм может быть разработан на основе метода **динамического программирования**.

Вспомним, что метод динамического программирования может применяться только для решения задач, соответствующих принципу оптимальности, который соблюдается в случае минимальных путей. Для того чтобы сформулировать соответствующие рекуррентные соотношения, введем в рассмотрение **матрицу стоимостей** $C = ||c_{ij}||_{n\times n}$, где элемент c_{ij} равен длине минимального пути между городами i и j. Компоненты c_{ij} матрицы C вычисляются следующим образом:

1) сперва рассматриваются минимальные пути, связывающие соседние города:

$$c_{ij} = \begin{cases} 0, & \text{если } i = j; \\ d_{ij}, & \text{если города } i, j \text{ соседние;} \\ \infty, & \text{в противном случае;} \end{cases}$$

2) далее рассматриваются минимальные пути между городами i, j, составленные при помощи непосредственно связанного с ними города k:

$$c_{ij} = \min(c_{ik}, c_{ik}+c_{jk}), \quad i, j \in \{1, 2, ..., n\}, \quad i \neq j, \quad i \neq k, \quad j \neq k;$$

3) пункт 2 повторяется для $k = 1, 2, ..., n$.

Текущие значения матрицы стоимостей для городов на *рис. 5.15* представлены на *рис. 5.16*.

Поскольку матрица стоимостей C содержит только длины, но не сами минимальные пути, будем строить минимальный путь, связывающий города a, b с помощью метода *Greedy*. В соответствии с этим методом, построение кратчайшего пути $X = (x_1, ..., x_{k-1}, x_k, ...)$ начинается с города $x_1 = a$. Далее на каждом шаге k выбирается такой город x_k, $x_k \in A_{k-1}$, который соответствует принципу оптимальности:

$$C[a, x_k] + C[x_k, b] = C[a, b].$$

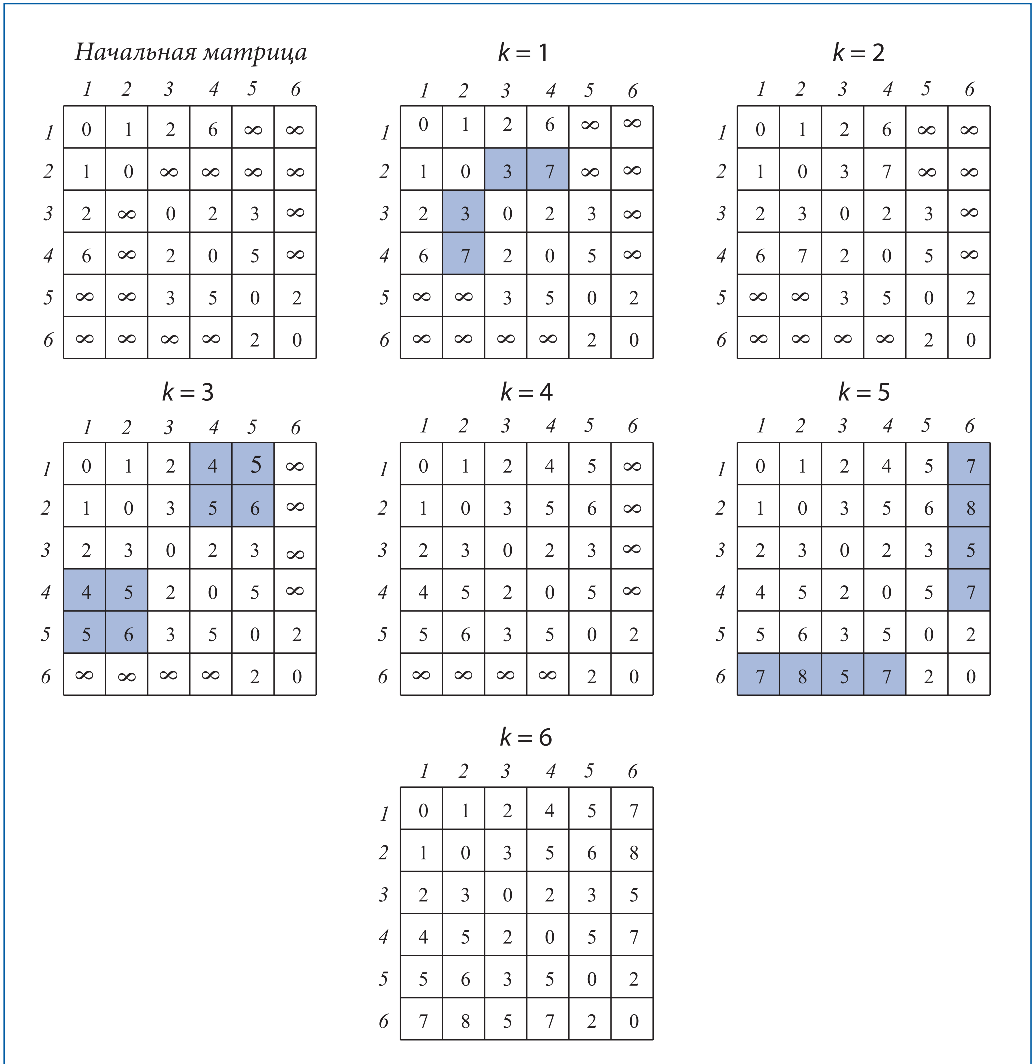
Например, для городов $a = 1$, $b = 6$ на *рис. 5.15* имеем:

$$X = (1);$$
$$X = (1, 3);$$
$$X = (1, 3, 5);$$
$$X = (1, 3, 5, 6).$$

Алгоритм, находящий кратчайшие пути, основанный на методе динамического программирования известен в специальной литературе под названием *алгоритма Роя-Флойда*. В качестве упражнения предлагаем Вам доказать, что этот алгоритм точен, то есть всегда строятся только кратчайшие пути.

В приведенной ниже программе точный алгоритм реализован с помощью процедуры `RoyFloyd`.

```
Program P160;
 {Задача о кратчайшем пути – метод динамического
программирования}
const nmax=50;
      Infinit=1.0E+35;
```



Начальная матрица

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	∞	∞	∞	∞
3	2	∞	0	2	3	∞
4	6	∞	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 1$

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	3	7	∞	∞
3	2	3	0	2	3	∞
4	6	7	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 2$

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	3	7	∞	∞
3	2	3	0	2	3	∞
4	6	7	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 3$

	1	2	3	4	5	6
1	0	1	2	4	5	∞
2	1	0	3	5	6	∞
3	2	3	0	2	3	∞
4	4	5	2	0	5	∞
5	5	6	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 4$

	1	2	3	4	5	6
1	0	1	2	4	5	∞
2	1	0	3	5	6	∞
3	2	3	0	2	3	∞
4	4	5	2	0	5	∞
5	5	6	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 5$

	1	2	3	4	5	6
1	0	1	2	4	5	7
2	1	0	3	5	6	8
3	2	3	0	2	3	5
4	4	5	2	0	5	7
5	5	6	3	5	0	2
6	7	8	5	7	2	0

$k = 6$

	1	2	3	4	5	6
1	0	1	2	4	5	7
2	1	0	3	5	6	8
3	2	3	0	2	3	5
4	4	5	2	0	5	7
5	5	6	3	5	0	2
6	7	8	5	7	2	0

Рис. 5.16. Текущие значения матрицы стоимостей

```
var n : integer; { число городов }
    D : array[1..nmax, 1..nmax] of real; { матрица расстояний }
    a, b : 1..nmAx;
    X : array [0..nmax+1] of integer;       { построенный путь }
    V : array[1..nmax, 1..nmax] of integer; { соседи }
    C : array[1..nmax, 1..nmax] of real; { матрица стоимостей }
    Finput : text;

procedure InitializareVecini;
 { Записывает в V[k] соседей города k }
var k, i, j, p, q, r : integer;
```

```
begin
  for k:=1 to n do
    begin
      { Сначала множество V[k] пустое }
      for i:=1 to n do V[k,i]:=0;
      { вычисляем элементы множества V[k] }
      i:=0;
      for j:=1 to n do
       if D[k,j]<>0 then
          begin
            i:=i+1;
            V[k,i]:=j;
          end; { then }
    end; { for }
end; { InitializareVecini }

procedure Initializare;
var i, j : integer;
begin
  assign(Finput, 'DRUM.IN');
  reset(Finput);
  readln(Finput, n);
  readln(Finput, a, b);
  writeln('n=', n, '    a=',  a, '  b=', b);
    for i:=1 to n do
      for j:=1 to n do read(Finput, D[i,j]);
  close(Finput);
  InitializareVecini;
end; { Initializare }

procedure AfisareaDrumului;
var k : integer;
begin
  write('Найденный путьт: ');
  k:=1;
  repeat
    write(X[k] : 3);
    k:=k+1;
  until X[k]=0;
  writeln;
  writeln('Длина пути ', C[a, b] : 5);
  readln;
end; { PrelucrareaSolutiei }

function Min(p, q : real) : real;
 { Возвращает минимум из p и q }
var s : real;
```

```
begin
  if p<q then s:=p else s:=q;
  if s>Infinit then s:=Infinit;
  Min:=s;
end; { Min }

procedure RoyFloyd;
var i, j, k : integer;
    s : real;
    ors : integer; { город-кандидат на включение
         в кратчайший путь }
    cnd : boolean; { условия включения города в путь }
begin
  { Инициализация матрицы стоимостей }
  for i:=1 to n do
    for j:=1 to n do
    if (D[i,j]=0) and (i<>j) then C[i,j]:=Infinit
                             else C[i,j]:=D[i,j];
  { Вычисление матрицы стоимостей }
  for k:=1 to n do
    for i:=1 to n do
      if i<>k then
        for j:=1 to n do
          if j<>k then C[i,j]:=Min(C[i,j], C[i,k]+C[j,k]);
  { Нахождение пути - метод Greedy }
  for k:=1 to n do X[k]:=0;
  k:=1; X[1]:=a;
  while X[k]<>b do
    begin
      i:=1;
      while V[X[k], i]<>0 do
        begin
          ors:=V[X[k], i];
          cnd:=true;
          for j:=1 to k do if ors=X[j] then cnd:=false;
          if cnd and (C[A, ors]+C[ors, B]=C[a,b])
            then X[k+1]:=ors;
          i:=i+1;
        end; { while }
      k:=k+1
    end; { while }
end; { RoyFloyd }

begin
  Initializare;
  RoyFloyd;
  AfisareaDrumului;
end.
```

Анализ процедуры `RoyFloyd` показывает, что временная сложность точного алгоритма составляет $O(n^3)$, т.е. он полиномиален.

На практике при решении конкретной задачи сперва делается попытка разработать точный алгоритм полиномиальной сложности. Если это не удается, переходят к разработке эвристического алгоритма. Для систематизации этого процесса следует учитывать все условия, которым должно удовлетворять оптимальное решение. Соответствующие условия могут быть разделены на два класса:

1) необходимые условия, то есть условия, невыполнение которых препятствует получению возможного решения задачи;

2) компромиссные условия, то есть условия, которые можно заменить на другие, более простые, приближаясь при этом к оптимальному решению.

Например, в случае кратчайшего пути $X = (a, x_2, ..., x_{k-1}, x_k, ..., b)$, то что x_k должен быть соседом города x_{k-1}, является необходимым условием, а соблюдение принципа оптимальности – компромиссное условие. В методе перебора с возвратом, компромиссное условие было заменено на более простое, а именно: x_k должен быть как можно ближе к городу x_{k-1}. Очевидно, что формулировка условий, допускающих компромисс и замена их более простыми, является задачей программиста.

Вопросы и упражнения

❶ Какая разница между точными и эвристическими алгоритмами?

❷ В каких случаях эвристические алгоритмы “лучше”, чем точные?

❸ Напишите программу, которая находит кратчайший путь методом полного перебора. Оцените временную сложность разработанной программы.

❹ Проведите сопоставительное сравнение точных и эвристических алгоритмов, предназначенных для решения задачи нахождения кратчайшего пути.

❺ Вычислите матрицу расстояний для сети дорог, связывающих районные центры. Найдите кратчайшие пути между районные центрами, полученные при помощи точных и эвристических алгоритмов.

❻ Как Вы считаете, может ли быть применен метод полного перебора для нахождения кратчайшего пути между любыми двумя населенными пунктами? Аргументируйте Ваш ответ.

❼ Сформулируйте необходимые и компромиссные условия для решения задачи о роботе, работающем на золотоносном участке (смотрите параграф 5.6).

❽ Оцените сложность точных и эвристических алгоритмов, предназначенных для решения следующих задач:

а) хранение файлов на магнитных лентах (упражнение 5, параграф 5.3);

б) непрерывная задача о рюкзаке (упражнение 6, параграф 5.3);

в) раскраска карты (упражнение 9, параграф 5.4);

г) дискретная задача о рюкзаке (упражнение 7, параграф 5.6);

д) архивирование файлов (упражнение 9, параграф 5.6);

е) триангуляция многоугольников (упражнение 10, параграф 5.6);

ж) игра Perspico (параграф 5.8);

з) задача коммивояжера (упражнение 10, параграф 5.8).

Глава 6

АЛГОРИТМЫ РЕШЕНИЯ НЕКОТОРЫХ МАТЕМАТИЧЕСКИХ ЗАДАЧ

6.1. Операции над множествами

В случае многих практически важных задач генерирование возможных решений включает обработку элементов различных множеств.

Пусть дано произвольное множество A, состоящее из n элементов:

$$A = \{a_1, a_2, \ldots, a_j, \ldots, a_n\}.$$

Поскольку язык ПАСКАЛЬ не предусматривает непосредственного доступа к каждому из элементов множеств, описанных с помощью типа данных `set`, будем запоминать элементы множества A в одномерном массиве размерности n: **A**=$(a_1, a_2, \ldots, a_j, \ldots, a_n)$. Очевидно, что такое представление устанавливает однозначное соответствие между порядком элементов исходного множества и порядком следования компонент в массиве **A**.

Пусть A_i – произвольное подмножество множества A. В ПАСКАЛЕ рассматриваемое подмножество может быть представлено **характеристическим вектором подмножества**, который имеет вид:

$$B_i = (b_1, b_2, \ldots, b_j, \ldots, b_n),$$

где

$$b_j = \begin{cases} 1, & \text{если } a_j \in A_i; \\ 0, & \text{в противном случае.} \end{cases}$$

Между подмножествами A_i множества A и характеристическими векторами B_i существует однозначное соответствие:

$$\begin{array}{ll}
A_1 = \varnothing & \leftrightarrow B_1 = (0, 0, \ldots, 0); \\
A_2 = \{a_1\} & \leftrightarrow B_2 = (1, 0, \ldots, 0); \\
A_3 = \{a_2\} & \leftrightarrow B_3 = (0, 1, \ldots, 0); \\
A_4 = \{a_1, a_2\} & \leftrightarrow B_3 = (1, 1, \ldots, 0); \\
\ldots & \quad \ldots \\
A_k = \{a_1, a_2, \ldots, a_n\} & \leftrightarrow B_k = (1, 1, \ldots, 1);
\end{array}$$

Очевидно, что количество всех подмножеств множества A равно $k = 2^n$.

Представление подмножеств с помощью характеристических векторов предполагает разработку специального программного модуля, который должен содержать отдельные процедуры для каждой из часто используемых операций над множествами: $\cup$ – объединения, $\cap$ – пересечения, $\setminus$ – разности,

– дополнения. В качестве упражнения предлагаем ученику самостоятельно разработать соответствующие процедуры.

Другой операцией, часто используемой в алгоритмах, основанных на методах перебора, является **генерирование всех подмножеств** заданного множества. Программная реализация указанной операции на ПАСКАЛЕ представлена в следующем примере.

Пример 1. Дано множество $A=\{a_1, a_2, \ldots, a_n\}$, состоящее из n целых чисел. Определите, существует ли хотя бы одно подмножество A_i, $A_i \subseteq A$, сумма элементов которого равна m.

Решение. Возможные решения $\varnothing$, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$ и т.д. можно получить последовательно формируя двоичные вектора $B_1, B_2, \ldots, B_k$.

```
Program P161;
 { Генерирование всех подмножеств заданного множества }
const nmax=50;
type Multime = array [1..nmax] of integer;
     CifraBinara = 0..1;
     VectorCaracteristic = array[1..nmax] of CifraBinara;
var A : Multime;
    B : VectorCaracteristic;
    n, m, j : integer;

function SolutiePosibila : boolean;
var j, suma : integer;
begin
  suma:=0;
  for j:=1 to n do
    if B[j]=1 then suma:=suma+A[j];
  if suma=m then SolutiePosibila:=true
            else SolutiePosibila:=false;
end; { SolutiePosibila }

procedure PrelucrareaSolutiei;
var j : integer;
begin
  write('Подмножество: ');
  for j:=1 to n do
    if B[j]=1 then write(A[j], ' ');
  writeln;
end; { PrelucrareaSolutiei }

procedure GenerareSubmultimi(var t:CifraBinara);
var j : integer;
begin
  t:=1; { transportul }
  for j:=1 to n do
    if t=1 then
```

```
      if B[j]=1 then B[j]:=0
                 else begin B[j]:=1; t:=0 end;
end; { GenerareSubmultimi }

procedure CautareSubmultimi;
var i : integer;
    t : CifraBinara;
begin
  for j:=1 to n do B[j]:=0;
  { начинаем с характеристического вектора B=(0, 0, ..., 0) }
  repeat
    if SolutiePosibila then PrelucrareaSolutiei;
    GenerareSubmultimi(t);
  until t=1;
end; { CautareSubmultimi }

begin
  write('Введите n='); readln(n);
  writeln('Введите ', n, ' целых чисел:');
  for j:=1 to n do read(A[j]);
  write('Введите m='); readln(m);
  CautareSubmultimi;
  writeln('Конец');
  readln;
end.
```

В программе `P161` последовательный перебор всех возможных решений реализован в процедуре `CautareSubmultimi` при помощи цикла **repeat... until**. Характеристические векторы соответствующих подмножеств формируются при помощи процедуры `GenerareSubmultimi`. В данной процедуре характеристический вектор B рассматривается как двоичное число, значение которого увеличивается на единицу при каждом вызове процедуры.

Операторы **if** в процедуре `GenerareSubmultimi` имитируют работу полусумматора, складывающего двоичные числа b_j и t, где переменная t представляет собой перенос. Значение $t = 1$ переноса из разряда n указывает на то, что после конечного вектора $B_k = (1, 1, \dots, 1)$ следует переходить к начальному вектору $B_1 = (0, 0, \dots, 0)$.

Временная сложность алгоритмов, основанных на генерировании всех подмножеств конечного множества, составляет $O(2^n)$.

В ряде других практически важных задач множество возможных решений S может быть получено как **декартово произведение других множеств**.

Пример 2. Пусть даны n множеств $A_1, A_2, \dots, A_n$, причем каждое множество A_j состоит из m_j целых чисел. Выберите из каждого множества A_j по одному целому числу a_j таким образом, чтобы произведение $a_1 \times a_2 \times \dots \times a_n$ было максимальным.

Решение. Множество возможных решений составляет $S=A_1 \times A_2 \times \dots \times A_n$. Очевидно, что количество возможных решений $k = m_1 \cdot m_2 \cdot \ldots \cdot m_n$. Каждый эле-

мент s_i декартова произведения $A_1 \times A_2 \times ... \times A_n$ может быть представлен **вектором индексов:**

$$C_i = (c_1, c_2, \ldots, c_j, \ldots, c_n),$$

где c_j – индексы соответствующего элемента из множества A_j. Например, для

$$\underset{}{A_1 = (\overset{①}{-6}, \overset{②}{2}, \overset{③}{1})}; \quad A_2 = (\overset{①}{4}, \overset{②}{9}); \quad A_3 = (\overset{①}{-8}, \overset{②}{3}, \overset{③}{5}),$$

получаем:

s_1 = (-6, 4, -8) ↔ C_1 = (①, ①, ①);
s_2 = (2, 4, -8) ↔ C_2 = (②, ①, ①);
s_3 = (1, 4, -8) ↔ C_3 = (③, ①, ①);
s_4 = (-6, 9, -8) ↔ C_4 = (①, ②, ①);
s_5 = (2, 9, -8) ↔ C_5 = (②, ②, ①);
…
s_{18}= (1, 9, 5) ↔ C_{18} = (③, ②, ③),

где ①, ② и ③ являются индексами элементов соответствующих множеств.

Векторы C_1, C_2, …, C_k могут быть сгенерированы в лексикографическом порядке, начиная с вектора C_1=(1, 1, …, 1).

```
Program P162;
 { Генерирование элементов декартова произведения }
const nmax=50; { максимальное количество множеств }
      mmax=50; { максимальное количество элементов }
type Multime = array [1..mmax] of integer;
     VectorIndicii = array[1..nmax] of 1..mmax;

var A : array[1..nmax] of Multime;
    n : 1..nmax;                        { количество множеств }
    M : array[1..nmax] of 1..mmax; { кардинал множества A[i] }
    Pmax : integer;                     { максимальное maximal }
    C, Cmax : VectorIndicii;
    i, j : integer;

procedure PrelucrareaSolutieiPosibile;
var j, p : integer;
begin
  p:=1;
  for j:=1 to n do p:=p*A[j, C[j]];
  if p > Pmax then begin Pmax:=p; Cmax:=C end;
end; { PrelucrareaSolutieiPosibile }

procedure GenerareProdusCartezian(var t:integer);
var j : integer;
begin
  t:=1; { перенос }
```

```
 for j:=1 to n do
     begin
       C[j]:=C[j]+t;
       if C[j]<=M[j] then t:=0 else C[j]:=1;
     end; { for }
end; { GenerareProdusCartezian }

procedure CautareaProdusuluiMaximal;
var j : integer;
    t : integer;
begin
  Pmax:=-MaxInt;
  writeln('Pmax=', Pmax);
  for j:=1 to n do C[j]:=1;
  { începem cu vectorul indiciilor C=(1, 1, ..., 1) }
  repeat
    PrelucrareaSolutieiPosibile;
    write('Декартово произведение: ');
    for j:=1 to n do write(A[j, C[j]], ' '); writeln;
    GenerareProdusCartezian(t);
  until t=1;
end; { CautareaProdusuluiMaximal }

begin
  write('Введите количество множеств n='); readln(n);
  for i:=1 to n do
    begin
      write('Введите кардинал M[', i, ']='); readln(M[i]);
      write('Введите элементы множества A[', i, ']: ');
      for j:=1 to M[i] do read(A[i, j]);
      writeln;
    end;

  CautareaProdusuluiMaximal;

  writeln('Pmax=', Pmax);
  write('Выбранные элементы: ');
  for j:=1 to n do write(A[j, Cmax[j]], ' ');
  writeln;
  readln;
  readln;
end.
```

В процедуре `GenerareProdusCartezian` вектор индексов C обрабатывается как целое число, записанное в **смешанной системе счисления**. В такой системе цифра c_1 записана по основанию m_1, цифра c_2 – по основанию m_2, цифра c_3 – по основанию m_3 и т.д. При каждом вызове процедуры `GenerareProdusCartezian`

значение числа, записанного в векторе C, увеличивается на единицу. Значение $t = 1$ переноса из разряда n указывает на то, что после конечного вектора $C_k = (m_1, m_2, \ldots, m_n)$ следует переходить к начальному вектору $C_1 = (1, 1, \ldots, 1)$.

Отметим, что если количество множеств n известно заранее, до написания самой программы, то генерирование элементов декартова произведения $A_1 \times A_2 \times \ldots \times A_n$ можно выполнить гораздо проще, а именно при помощи n вложенных циклов:

```
for j1:=1 to m1 do
  for j2:=1 to m2 do
   ...
    for jn:=1 to mn do
            if SolutiePosibila(aj1, aj2 ,..., ajn)
              then PrelucrareaSoluţiei(aj1, aj2,  ..., ajn)
```

Временная сложность алгоритмов, основанных на генерировании всех элементов декартова произведения, составляет $O(m^n)$, где $m = \max(m_1, m_2, \ldots, m_n)$.

Вопросы и упражнения

❶ Подмножества A_i, A_j множества A представлены характеристическими векторами. Напишите процедуры, реализующие следующие операции: $A_i \cap A_j$, $A_i \cup A_j$, $A_i \setminus A_j$, $\bar{A}_i$.

❷ Любые подмножества A_i, $A_i \subseteq A$, можно представить вектором из n компонент (n-элементным массивом), где элементы подмножества A_i размещены в начале вектора, а оставшиеся позиции имеют значения, не принадлежащие множеству A. Напишите процедуры, реализующие следующие операции, известные из теории множеств: $\cup$, $\cap$, $\setminus$, $\bar{\ }$. Как Вы считаете, какое из представлений множеств удобнее: при помощи характеристических векторов или при помощи векторов, содержащих сами элементы подмножества?

❸ Оцените время выполнения процедуры `CautareSubmultimi` из программы `P161`. Проверьте полученные оценки путем непосредственного измерения времени выполнения алгоритма для различных значений n.

❹ Пусть дано множество A, состоящее из первых n букв латинского алфавита. Напишите программу, которая выводит на экран все подмножества этого множества.

❺ Дано натуральное число вида n = 32*35*17*, состоящее из 9 десятичных цифр. Найдите цифры, которые должны быть записаны в позиции, отмеченные символом *, таким образом, чтобы полученное число делилось без остатка на m.

❻ Оцените время выполнения процедуры `CautareaProdusuluiMaximal` из программы `P162`. Проверьте полученные Вами оценки путем прямого измерения времени выполнения алгоритма для различных значений n и $m_1, m_2, \ldots, m_n$.

❼ В корзине лежат m яблок и p груш. Сформируйте все возможные варианты выбора f фруктов таким образом, чтобы k из них были яблоками.

❽ Разработайте рекурсивную процедуру, которая генерирует все элементы декартова произведения заданных множеств.

❾ Пусть $A = (a_1, a_2, \ldots, a_j, \ldots, a_n)$ – упорядоченное множество символов, называемое **алфавитом**. Назовем **словом длины *p*** любую последовательность p символов из алфавита A. Напишите процедуру, которая генерирует все слова длины p.

6.2. Комбинаторный анализ

Решение многих задач, связанных с последовательным анализом возможных решений, предполагает генерирование перестановок, размещений и сочетаний элементов заданного множества.

Генерирование перестановок. Известно, что количество возможных перестановок множества $A=\{a_1, a_2, ..., a_n\}$ из n элементов составляет $P_n = n!$. Это число может быть вычислено с помощью функции *factorial*, представленной в итеративной

$$P_n = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$$

или рекурсивной форме:

$$P_n = \begin{cases} 1, & \text{если } n = 0; \\ P_{n-1} \cdot n, & \text{если } n > 0. \end{cases}$$

Например, для $A=\{a_1, a_2\}$ количество перестановок $P_2 = 2! = 2$, а перестановками являются (a_1, a_2) и (a_2, a_1). Для $A=\{a_1, a_2, a_3\}$ количество $P_3 = 3! = 6$, а перестановками являются:

$$(a_1, a_2, a_3); \quad (a_1, a_3, a_2); \quad (a_2, a_1, a_3);$$
$$(a_2, a_3, a_1); \quad (a_3, a_1, a_2); \quad (a_3, a_2, a_1).$$

Поскольку между перестановками элементов множества $A=\{a_1, a_2, ..., a_n\}$ и перестановками множества $I=\{1, 2, ..., n\}$ существует однозначное соответствие, в общем случае, задача генерирования перестановок произвольного множества A из n элементов сводится к генерированию перестановок множества $\{1, 2, ..., n\}$, называемых **перестановками порядка *n*.**

Существует множество хитроумных методов генерирования перестановок, причем самым распространенным из них является **лексикографический метод.** В этом методе начальной является наименьшая в лексикографическом смысле перестановка, а именно **тождественная перестановка** $(1, 2, ..., n)$. Имея очередную текущую перестановку $p = (p_1, ..., p_{i-1}, p_i, p_{i+1}, ..., p_n)$, для того чтобы получить следующую в лексикографическом порядке перестановку p', ищем такой индекс i, который удовлетворяет следующим условиям:

$$p_i < p_{i+1}; \; p_{i+1} > p_{i+2} > ... > p_n.$$

Далее, элемент p_i заменяется на наименьший из элементов $p_{i+1}, ..., p_n$, которые больше чем p_i, пусть это будет p_k:

$$(p_1, ..., p_{i-1}, p_k, p_{i+1}, ..., p_{k-1}, p_i, p_{k+1}, ..., p_n).$$

Искомая перестановка p' получается путем инвертирования порядка последних $(n-1)$ элементов этого вектора так, чтобы они располагались в возрастающем порядке.

Если в процессе вычислений больше не существует индекса i, удовлетворяющего указанным выше условиям, это означает, что получена самая большая в лексикографическом порядке перестановка, т.е. $(n, (n-1), ..., 1)$ и выполнение алгоритма на этом заканчивается.

Например, для $n=3$ получаем перестановки:

$$(1, 2, 3); \quad (1, 3, 2); \quad (2, 1, 3);$$
$$(2, 3, 1); \quad (3, 1, 2); \quad (3, 2, 1).$$

В приведенной ниже программе лексикографический метод реализуется при помощи процедуры GenerarePermutari.

```
Program P163;
 { Генерирование перестановок }
const nmax=100;

type Permutare=array[1..nmax] of 1..nmax;
var P : Permutare;
    n : 2..nmax;
    Indicator : boolean;
    i : integer;

procedure GenerarePermutari(var Indicator : boolean);
label 1;
var i, j, k, aux : integer;
begin
  { тождественная перестановка }
  if not Indicator then
     begin
       for i:=1 to n do P[i]:=i;
       Indicator:=true;
       goto 1;
     end;

  { поиск индекса i }
  i:=n-1;
  while P[i]>P[i+1] do
    begin
      i:=i-1;
      if i=0 then
        begin
          { такого индекса нет }
          Indicator:=false;
          goto 1;
        end; { then }
    end; {while }

  { поиск индекса k }
  k:=n;
  while P[i]>P[k] do k:=k-1;

  { interschimbarea P[i] - P[k] }
  aux:=P[i]; P[i]:=P[k]; P[k]:=aux;

  { упорядочивание последних (n-i) элементов }
 for j:=1 to (n-i) div 2 do
```

```
    begin
      aux:=P[i+j];
      P[i+j]:=P[n-j+1];
      P[n-j+1]:=aux;
    end; { for }
  Indicator:=true;
1:end; { GenerarePermutari }

begin
  write('Введите n='); readln(n);
  Indicator:=false;
  repeat
    GenerarePermutari(Indicator);
    if Indicator then
      for i:=1 to n do write(P[i] : 3);
      writeln;
  until not Indicator;
  readln;
end.
```

Для того чтобы начать с исходной перестановки до первого обращения к процедуре `GenerarePermutari`, параметру `Indicator` присваивается значение `false`. При каждом вызове процедуры в вектор `P` заносится следующая в лексикографическом порядке перестановка, а параметру `Indicator` присваивается значение `true`. После генерирования всех перестановок процедура `GenerarePermutari` присваивает параметру `Indicator` значение `false`.

К сожалению, независимо от используемого метода, время, требуемое для генерирования всех перестановок составляет как минимум $O(n!)$. Вследствие этого алгоритмы, предназначенные для поиска решений путем генерирования всех возможных перестановок, применимы только для малых значений n.

Генерирование размещений. Количество размещений m элементов множества $A=\{a_1, a_2, ..., a_n\}$ из n элементов вычисляется по формуле:

$$A_n^m = \frac{n!}{(n-m)!}$$

или, без использования функции *факториал*:

$$A_n^m = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot (n-m+2) \cdot (n-m+1).$$

Как и в случае перестановок, задача генерирования размещений произвольного множества A сводится к генерированию размещений множества $I=\{1, 2, ..., n\}$. Например, для $I=\{1, 2, 3\}$ и $m=2$ соответствующими размещениями являются:

$$(1, 2); \quad (2, 1); \quad (1, 3);$$
$$(3, 1); \quad (2, 3); \quad (3, 2).$$

Генерирование размещений может быть осуществлено в лексикографическом порядке, начиная с наименьшего размещения $a=(1, 2, ..., m)$.

Пусть $a=(a_1, a_2, ..., a_i, ..., a_m)$ текущее размещение. Для того чтобы найти наследника a' размещения a, ищем сперва самый большой индекс i, такой что a_i может быть увеличен. Элемент a_i может быть увеличен, если существует хотя бы одно из значений a_i+1, a_i+2, ..., n, которым может быть заменен a_i. Для облегчения указанных проверок используется вектор $D=(d_1, d_2, ..., d_i, ..., d_n)$, где d_i равно 0 или 1 в зависимости от того, содержится или не содержится значение i в текущем размещении a. После того как определены индексы i, элементы a_i, a_{i+1}, ..., a_m принимают в качестве значений в возрастающем порядке наименьшие из имеющихся в распоряжении чисел.

Если индекса i с указанным выше свойством не существует, это означает, что достигнуто размещение $(n-m+1, n-m+2, ..., n)$ и процесс генерирования завершен. Для того чтобы указать факт завершения вычислений, в приведенной ниже программе используется логическая переменная `Indicator`.

```
Program P164;
 { Генерирование размещений }
const nmax=100;
      mmax=100;

type Aranjament=array[1..mmax] of 1..nmax;

var A : Aranjament;
    D : array[1..nmax] of 0..1;
    n : 1..nmax;
    m : 1..nmax;
    i : integer;
    Indicator : boolean;

procedure GenerareAranjamente(var Indicator : boolean);
label 1;
var i, j, k, l : integer;
begin
  { исходное размещение }
  if not Indicator then
    begin
      for i:=1 to m do
        begin
          A[i]:=i; D[i]:=1;
        end;
     for i:=m+1 to n do D[i]:=0;
      Indicator:=true;
      goto 1;
    end;
  { наследник текущего размещения }
  for i:=m downto 1 do
    begin
      D[A[i]]:=0;
```

```
      for j:=A[i]+1 to n do
         if D[j]=0 then
           begin
             A[i]:=j; D[j]:=1; k:=0;
             for l:=i+1 to m do
               begin
                 repeat k:=k+1 until D[k]=0;
                A[l]:=k; D[k]:=1;
               end; { for }
             goto 1;
           end; { if }
     end; { for }
   Indicator:=false;
 1:end; { GenerareAranjamente }

 begin
   write('Введите n='); readln(n);
   write('Введите m='); readln(m);
   Indicator:=false;
   repeat
     GenerareAranjamente(Indicator);
     if Indicator then
       for i:=1 to m do write(A[i] : 3);
     writeln;
   until not Indicator;
   readln;
 end.
```

Очевидно, временная сложность алгоритмов, основанных на генерировании всех размещений, составляет O(n!).

Генерирование сочетаний. Количество сочетаний из n элементов по m ($m \le n$) вычисляется по известной формуле:

$$C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{m!(n-m)!}.$$

Например, для I={1, 2, 3} и m=2 имеем сочетания:

{1, 2}; {1, 3}; {2, 3}.

Генерирование сочетаний может быть осуществлено в лексикографическом порядке, начиная с исходного сочетания {1, 2, ..., m}.

Пусть дано сочетание $c=\{c_1, c_2, ..., c_i, ..., c_m\}$. Непосредственно следующее за ним в лексикографическом порядке сочетание находится следующим образом:

– находим индекс i, удовлетворяющий соотношениям $c_i<n-m+1$, $c_{i+1}=n-m+i+1$, $c_{m-1}=n-1$, $c_m=n$;

– переходим к сочетанию $c'=\{c_1, ..., c_{i-1}, c_i+1, c_i+2, ..., c_i+n-i+1\}$.

Если индекса i, удовлетворяющего вышеприведенным условиям, не существует, то это означает, что уже были сгенерированы все сочетания.

В приведенной ниже программе сочетания множества $I=\{1, 2, ..., n\}$ генерируются последовательно в векторе (одномерном массиве) C.

```
Program P165;
  { Генерирование сочетаний }
const nmax=100;
      mmax=100;

type Combinare=array[1..mmax] of 1..nmax;
var C : Combinare;
    n : 1..nmax;
    m : 1..mmax;
    i : integer;
    Indicator : boolean;

procedure GenerareCombinari(var Indicator : boolean);
label 1;
var i, j : integer;
begin
  { исходное сочетание }
  if not Indicator then
    begin
      for i:=1 to m do C[i]:=i;
      Indicator:=true;
      goto 1;
    end;
  { наследник текущего сочетания }
  for i:=m downto 1 do
     if C[i]<(n-m+i) then
        begin
          C[i]:=C[i]+1;
          for j:=i+1 to m do C[j]:=C[j-1]+1;
          goto 1;
        end; { then }
  Indicator:=false;
1:end; { GenerareCombinari }

begin
  write('Введите n='); readln(n);
  write('Введите m='); readln(m);
  Indicator:=false;
 repeat
    GenerareCombinari(Indicator);
    if Indicator then
       for i:=1 to m do write(C[i] :3);
    writeln;
```

```
  until not Indicator;
  readln;
end.
```

Можно доказать, что временная сложность алгоритмов, основанных на генерировании всех сочетаний, составляет $O(n^k)$, где k=min(m, n–m+1), т.е. полиномиальна.

Вопросы и упражнения

❶ Напишите на ПАСКАЛЕ программу, которая выводит на экран количество перестановок P_n, количество размещений и количество сочетаний. Значения n и m вводятся с клавиатуры.

❷ Разработайте рекурсивную процедуру, генерирующую все возможные перестановки множества I={1, 2, ..., n}.

❸ Используя метод перебора с возвратом, составьте алгоритмы генерирования перестановок, размещений и сочетаний произвольного множества, состоящего из n различных элементов.

❹ Рассмотрим двумерный массив `T[1..n, 1..n]` целых чисел. Напишите программу, которая находит такую перестановку столбцов массива, после выполнения которой сумма элементов на главной диагонали была бы минимальна.

❺ Напишите программу, которая выводит на экран все возможные строки, состоящие из символов `A`, `b`, `C`, `d`, `E`. Каждый символ должен встречаться в строке ровно один раз.

❻ Из списка n кандидатов нужно выбрать m игроков, которые войдут в футбольную команду. Напишите программу, которая выводит на экран все возможные варианты выбора m игроков.

❼ Дано множество целых чисел $A=\{a_1, a_2, ..., a_n\}$. Напишите программу, которая находит подмножество, содержащее ровно m элементов множества A таким образом, чтобы их сумма была максимальна.

❽ Как Вы считаете, в чем преимущества и недостатки алгоритмов, основыванных на генерировании всех возможных перестановок, размещений и сочетаний?

❾ Существуют ли методы программирования, которые позволяют избежать полного перебора всех возможных перестановок, размещений и сочетаний?

❿ Дано множество целых чисел $A=\{a_1, a_2, ..., a_n\}$. Напишите программу, которая находит такое разбиение множества A на два непустых множества B и C, при котором сумма элементов подмножества B равна сумме элементов подмножества C. Например, для A={–4, –1, 0, 1, 2, 3, 9} получаем B={–4, 0, 9} и C={–1, 1, 2, 3}.

Глава 7

ЗАДАЧИ НА ПОВТОРЕНИЕ

Задачи, приведенные ниже, предлагались на различных конкурсах по информатике. Их решение требует углубленных знаний методов программирования, а также методов оценки сложности алгоритмов.

1. Круги. Квадрат со стороной a см содержит n кругов ($n \le 100$). Для каждого круга i задаются координаты его центра (x_i, y_i) и радиус r_i. Напишите программу, которая не более чем за t секунд вычисляет как можно точнее площадь, покрываемую этими n кругами.

2. Точки. На плоскости заданы n точек, $n \le 100$. Необходимо вычислить максимальное число коллинеарных точек.

3. Мосты. Даны n островов, связанных m мостами. Известно, что по мосту между островами i, j могут передвигаться транспортные средства, вес которых не превышает g_{ij} тонн. Найдите максимальный вес транспортного средства G_{ab}, которое может доехать от острова a до острова b.

4. Тексты. Даны n текстов, которые должны быть напечатаны на листах бумаги. Текст i состоит из r_i строк. На одном листе может быть напечатано не более m строк. Если на одном листе можно напечатать два или более текстов, для их отделения друг от друга между ними вставляется пустая строка. Фрагментирование текстов запрещено, т.е. все строки каждого текста должны размещаться на одном и том же листе. Напишите программу, которая находит минимальное количество листов, необходимых для распечатки всех текстов.

5. Окружность. Рассматриваются n точек на декартовой плоскости. Каждая точка i задана ее координатами x_i, y_i. Напишите программу, которая проверяет, принадлежат ли заданные точки некоторой окружности.

6. Телефонная сеть. Имеются n городов, $n \le 100$, которые с помощью кабеля необходимо соединить в телефонную сеть. Для каждого города i известны его декартовы координаты x_i, y_i. Кабель, соединяющий два города, не может иметь ответвлений. Абоненты телефонной сети связываются между собой напрямую или через телефонные станции промежуточных городов. Длина кабеля, соединяющего города i, j, равна расстоянию между ними. Найдите суммарную минимальную длину кабеля, необходимого для того, чтобы между любыми двумя абонентами существовала телефонная связь.

7. Вычисление выражений. Даны арифметические выражения, состоящие из целых чисел, скобок (,) и знаков бинарных операций `+`, `-`, `*`, **`mod`**, **`div`**. Напишите программу, вычисляющую значения рассматриваемых выражений.

8. Психология. Имеется n служащих, $n \leq 100$, которых необходимо распределить по m бригадам. Каждая бригада состоит из k служащих, $k \cdot m = n$. Совместимость между служащими i, j характеризуется коэффициентом r_{ij}, который может принимать следующие целые значения: 0 (полная несовместимость), 1, 2, ... , 10 (отличная совместимость). Совместимость всего коллектива C вычисляется суммированием коэффициентов r_{ij} для всех возможных пар (i, j) внутри каждой бригады. Найдите максимальную совместимость C_{max}, которую можно обеспечить подходящим распределением служащих по бригадам.

9. Компас. Рассматриваются n точек на декартовой плоскости. Каждая точка i задана ее координатами x_i, y_i. Напишите программу, которая проверяет, можно ли нарисовать окружность с центром в одной из имеющихся точек так, чтобы она прошла через все остальные точки.

10. Пересечение прямоугольников. Даны n прямоугольников ($n \leq 10$), стороны которых параллельны осям координат, а координаты вершин являются натуральными числами из множества {0, 1, 2, ..., 20}. Напишите программу, которая вычисляет площадь фигуры, получаемой в результате пересечения рассматриваемых прямоугольников.

11. Объединение прямоугольников. Даны n прямоугольников ($n \leq 10$), стороны которых параллельны осям координат, а координаты вершин являются действительными числами. Напишите программу, которая вычисляет площадь фигуры, получаемой в результате объединения n прямоугольников.

12. Простые числа. Вычислите все простые числа, состоящие из 4 цифр, инверсии (числа, записанные теми же цифрами, но в обратном порядке) которых также являются простыми числами, причем суммы их цифр также являются простыми числами.

13. Видимость. Дана замкнутая несамопересекающаяся ломаная линия $P_1P_2 \ldots P_nP_1$, $n \leq 20$. В точке A внутри линии расположен наблюдатель. В общем случае, некоторые отрезки ломаной линии могут быть частично или полностью невидимыми для наблюдателя. Напишите программу, которая вычисляет количество отрезков, полностью невидимых для наблюдателя.

14. Фонари. Парк прямоугольной формы разделен на квадраты одинакового размера. В любом из квадратов парка может быть установлен фонарь. Каждый фонарь освещает не только квадрат, в котором он находится, а также восемь соседних квадратов. Напишите программу, которая вычисляет минимальное количество фонарей, необходимых для освещения всего парка.

15. Лазер. Дана прямоугольная пластина размерами $m \times n$, где m и n натуральные числа. Пластина должна быть разрезана на $m \times n$ маленьких пластин размером 1×1 каждая. Поскольку исходная пластина неоднородна, то для каждого ее участка указывается плотность d_{xy}, где x, y – это координаты нижнего левого угла соответствующего квадрата.

Для операции разрезания используется лазер. Каждая операция разрезания включает в себя:

– установку разрезаемой пластины на крепежном столе;

– установку мощности лазерного луча в соответствии с плотностью разрезаемого материала;

– одноразовое перемещение лазера вдоль прямой, параллельной одной из осей координат (собственно разрезание);

– снятие двух полученных пластин с крепежного стола.

Стоимость одной операции разрезания определяется по формуле $c = d_{max}$, где d_{max} – это максимальная из плотностей кусков 1×1, по границам которых прошел луч лазера. Очевидно, что общая стоимость T может быть получена путем сложения стоимостей c всех операций разрезания, необходимых для получения пластин размером 1×1. Напишите программу, которая находит минимальную стоимость T.

16. Уезды. Территория некоторой страны разделена на уезды (*рис. 7.1*). На контурной карте страны границы каждого уезда представлены многоугольником, определенным координатами его вершин (x_i, y_i). Предполагается, что вершины многоугольника пронумерованы 1, 2, 3, ..., n, а сами координаты вершин представляют собой целые числа. Внутри каждого уезда не существует других уездов.

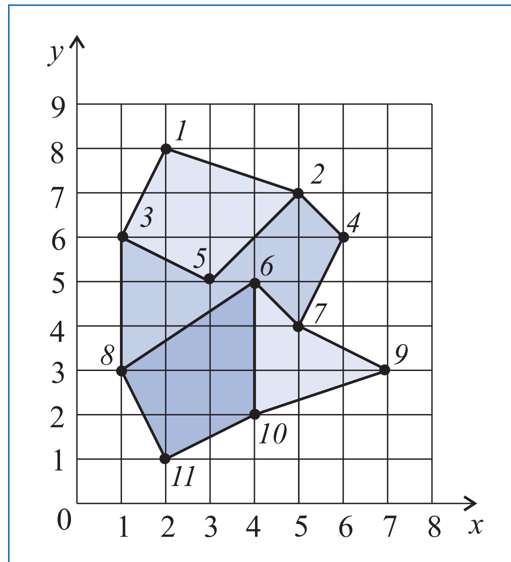


Рис. 7.1. Уезды некоторой страны

Компьютерный вирус частично разрушил информацию об административных границах, оставив нетронутыми следующие данные:

– количество n и координаты (x_i, y_i) вершин всех многоугольников;

– количество отрезков m, которое образует стороны многоугольников и информацию о концах каждого из отрезков.

Напишите программу, которая определяет количество уездов d и вершины каждого из многоугольников, являющихся административной границей уезда.

17. Башни. Даны n прямоугольных плит, пронумерованных от 1 до n. Каждая плита i описывается толщиной h_i и длинами сторон x_i, y_i. Напишите программу, которая находит максимальную высоту башни, которую можно сложить из рассматриваемых плит. Для обеспечения устойчивости башни необходимо соблюдать следующие правила (*рис. 7.2*):

– плиты можно размещать друг на друге только горизонтально, не на ребра или иным другим способом;

– соответствующие ребра укладываемых плит должны быть параллельны друг другу;

– любая плита башни должна целиком размещаться на поверхности нижней по отношению к ней плиты (естественно, плита в основании башни размещается на земле);

– часть плит может остаться неиспользованной.

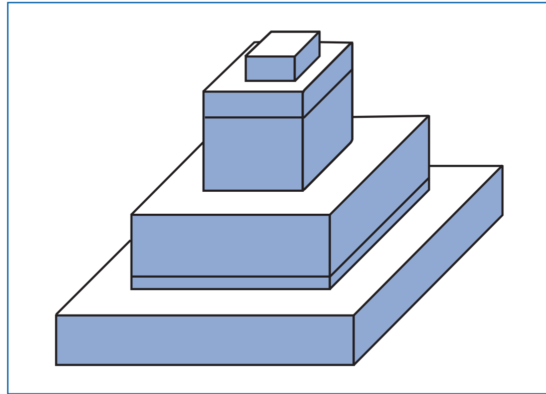


Рис. 7.2. Башня, сложенная из прямоугольных плит

Ввод. Текстовый файл `TURNURI.IN` содержит в первой строке число n. В каждой из следующих n строк содержится по три целых положительных числа x_i, y_i, h_i, разделенных пробелами.

Вывод. Текстовый файл `TURNURI.OUT` должен содержать в одной строке единственное целое число – максимальную высоту башни.

Пример.

TURNURI.IN

```
5
3 4 2
3 4 3
4 3 2
1 5 4
2 2 1
```

TURNURI.OUT

```
8
```

Ограничения: $n \leq 1000$; x_i, y_i, h_i < 1000. Время работы программы не должно превышать 10 сек.

18. Спелеология. Спелеолог – специалист, занимающийся изучением пещер. Работа спелеолога предполагает прохождение различных подземных лабиринтов. Один из таких лабиринтов состоит из n, $n \leq 100$, пещер и коридоров (*рис. 7.3*). У каждой пещеры есть собственное название, состоящее максимум из 10 символов (букв и цифр), написанных слитно на стене этой пещеры. Начальная пещера называется *INTRARE*, а конечная – *IESIRE*. На входе в каждый коридор написано название пещеры, в которую он ведет.

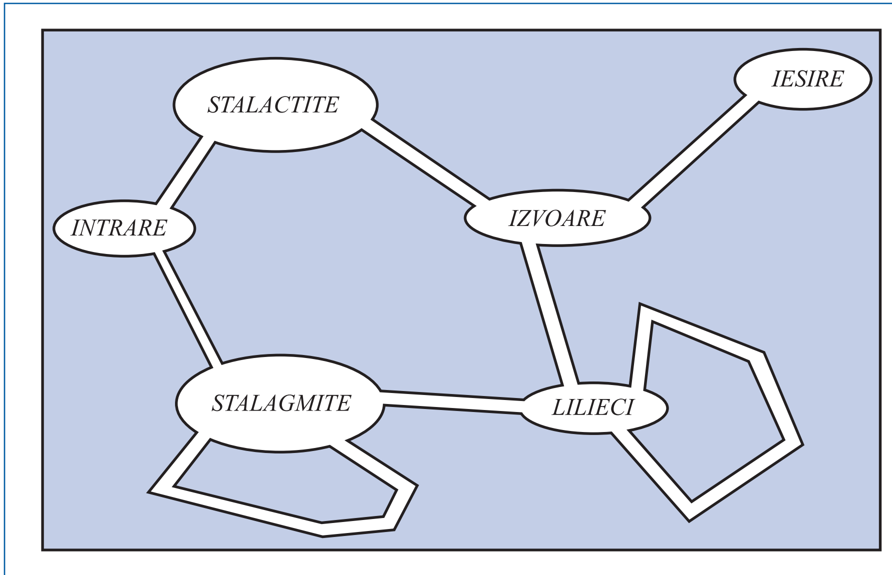


Рис. 7.3. План некоторой пещеры

Спелеологу не известен план лабиринта. Но у него есть тетрадь, карандаш и фонарь, которыми он может пользоваться, чтобы читать надписи на стенах и делать заметки. Будем называть путем последовательность пещер, в которой любые две соседние пещеры соединены коридором. Под длиной пути будем понимать количество пещер, из которых он состоит. Например, путь *INTRARE, STALAGMITE, LILIECI, IZVOARE, IESIRE* имеет длину, равную 5.

Напишите программу, которая находит один из кратчайших путей от пещеры *INTRARE* к пещере *IESIRE*.

Ввод. Входного файла не существует. Однако характеристика очередной пещеры может быть определена путем обращения к предопределенной функции `UndeMaAflu` типа `string`. Функция возвращает строку, содержащую название пещеры, в которой в данный момент находится спелеолог, двоеточие и надписи над соответствующими коридорами, записанные через пробел. Например, если спелеолог находится в пещере *LILIECI*, то функция возвращает значение:

```
LILIECI: STALAGMITE IZVOARE LILIECI LILIECI
```

Переход из текущей пещеры в пещеру, в которую ведет коридор *c*, осуществляется путем вызова предопределенной процедуры `TreciCoridorul(c)`, где `c` – это выражение типа `string`. Если указан несуществующий коридор, то спелеолог остается на месте. Чтобы данные подпрограммы были доступными, включите в декларативную часть разрабатываемой Вами программы строку:

```
uses LABIRINT;
```

Вывод. Текстовый файл `SPEOLOG.OUT` должен содержать в первой строке целое число – длину кратчайшего пути. В следующих строках записывается сам путь. Название каждой пещеры должно быть записано в отдельной строке.

Если требуемого пути не существует, то файл должен содержать единственную строку со словом FUNDAC.

Пример. Для лабиринта на *рис. 7.3* имеем:

SPEOLOG.OUT

```
4
INTRARE
STALACTITE
IZVOARE
IESIRE
```

Время работы программы не должно превышать 20 сек.

19. Сад. План некоторого сада прямоугольной формы размерами $n \times m$ состоит из квадратных участков со сторонами, равными 1. На каждом участке растут деревья только одной породы. Любая порода может занимать один и более участков, причем не обязательно соседних. Напишите программу, которая находит прямоугольную область минимальной площади, содержащую не менее k пород деревьев. Стороны области должны совпадать со сторонами участков на плане.

20. Дискотека. Посетители дискотеки пронумерованы от 1 до n. Сначала только одному из них, посетителю с номером i, известна очень важная новость, которую он позже сообщает своим друзьям. В дальнейшем, любой посетитель j, который уже узнал эту новость, также сообщает ее только своим друзьям. Напишите программу, которая определяет число участников p, которые узнали указанную новость. Отношение "друзья" на множестве посетителей задается с помощью m различных пар типа $\{j, k\}$ со значением "посетители j, k являются друзьями". Считается, что $3 \le n \le 1000$ и $2 \le m \le 30000$.

21. Сапер. Участок дороги разделен на n отрезков. На каждом отрезке установлено не более одной мины. Сапер записал информацию об установленных минах в одномерном массиве $M = ||m_i||$, в котором $m_i = 1$, если отрезок i содержит мину, и $m_i = 0$ в противном случае. На всякий случай, сапер перекодировал информацию из массива M в другой массив $C = ||c_i||_n$, элементы которого определяются по следующим правилам:

$$c_i = \begin{cases} m_1 + m_2, & \text{для } i = 1; \\ m_{i-1} + m_i + m_{i+1}, & \text{для } 1 < i < n; \\ m_{n-1} + m_n, & \text{для } i = n. \end{cases}$$

В боевых условиях данные из массива M были утеряны. Напишите программу, которая восстанавливает массив M, получая на вход массив C. Предполагается, что задача имеет хотя бы одно решение и $3 \le n \le 10000$.

22. Коробки. Даны n коробок в виде прямоугольных параллелепипедов. Для каждой коробки известны ее размеры x_i, y_i, z_i. В зависимости от размеров коробок, некоторые из них могут быть помещены одна в другую, причем, при необходимости, их можно поворачивать. Коробка i может быть помещена в коробку j только в том случае, если в результате всех возможных вращений

найдется такое положение, для которого размеры коробки i окажутся строго меньше соответствующих размеров коробки j. Из соображений красоты, требуется, чтобы соответствующие стороны вложенных коробок были параллельными (*рис.7.4*). В общем случае, любая вкладываемая коробка может содержать другие вложенные коробки или, другими словами, может быть образован ряд номеров коробок i_1, i_2, i_3, ..., i_k такой, что коробка i_1 находится в коробке i_2; коробка i_2 находится в коробке i_3 и т.д. Напишите программу, которая находит максимальное число коробок $k_{\max}$, которые могут быть вложены друг в друга.

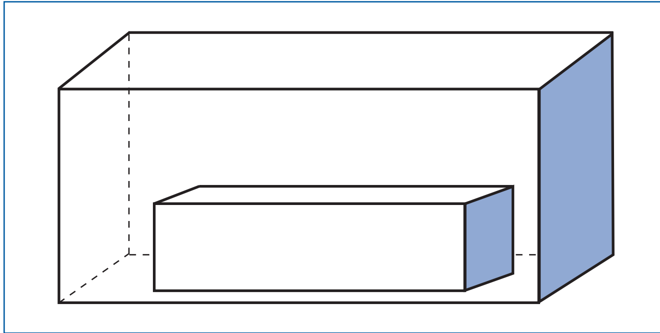


Рис. 7.4. Вложенные коробки

Ввод. Текстовый файл `CUTII.IN` содержит в своей первой строке натуральное число n. Каждая из следующих n строк содержит по три натуральных числа x_i, y_i, z_i, записанных через пробел.

Вывод. Текстовый файл `CUTII.OUT` должен содержать в единственной строке натуральное число $k_{\max}$.

Пример.

CUTII.IN	CUTII.OUT
5 4 4 4 1 3 5 2 2 3 1 1 1 1 1 2	3

Ограничения. $2 \le n \le 500$; $1 \le x_i$, y_i, z_i ≤ 30000. Время выполнения программы не должно превышать 2 секунд.

Acest manual este proprietatea Ministerului Educației al Republicii Moldova.

Liceul/gimnaziul ____

Manualul nr. ____

Nr. crt.	Numele de familie şi prenumele elevului	Anul şcolar	Aspectul manualului	
			la primire	la restituire
1.				
2.				
3.				
4.				
5.				

Dirigintele verifică dacă numele elevului este scris corect.

Elevul nu trebuie să facă niciun fel de însemnări în manual.

Aspectul manualului (la primire şi la restituire) se va aprecia folosind termenii: *nou, bun, satisfăcător, nesatisfăcător*.

Imprimare la Tipografia „BALACRON” SRL,
str. Calea Ieşilor, 10; MD-2069
Chişinău, Republica Moldova
Comanda nr. 561