ГЛАВА 5. МЕТОДЫ ПОСТРОЕНИЯ АЛГОРИТМОВ

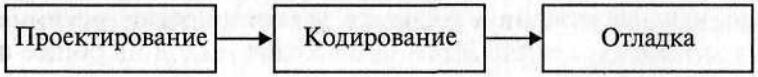
5.1. Основные понятия структурного программирования

Прошло уже более полувека со времени появления первой ЭВМ. Все это время вычислительная техника бурно развивалась. Менялась элементная база ЭВМ, росли быстродействие, объем памяти, менялись средства взаимодействия человека с машиной. Безусловно, эти изменения сказывались самым непосредственным образом на работе программиста. Определенный общепринятый способ производства чего-либо (в данном случае — программ) называют технологией. Далее мы будем говорить о *технологии программирования*.

На первых ЭВМ с «тесной» памятью и небольшим быстродействием основным показателем качества программы была ее экономичность по занимаемой памяти и времени счета. Чем программа получалась короче, тем класс программиста считался выше. Такое сокращение программы часто давалось большими усилиями. Иногда программа получалась настолько «хитрой», что могла «перехитрить» самого автора. Возвращаясь через некоторое время к собственной программе, желая что-то изменить, программист мог запутаться в ней, забыв свою «гениальную идею».

Так как вероятность выхода из строя сложного технического устройства больше, чем простого, очень сложный алгоритм всегда увеличивает вероятность ошибки в программе.

В процессе изготовления программного продукта программист должен пройти определенные этапы.



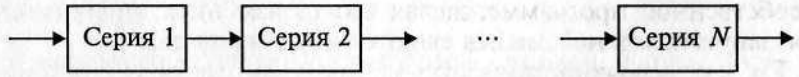
На стадии проектирования строится алгоритм будущей программы, например, в виде блок-схемы. Кодирование — это составление текста программы на языке программирования. Отладка осуществляется с помощью тестов, т.е. программа выполняется с некоторым заранее продуманным набором исходных данных, для которого известен результат. Чем сложнее программа, тем большее число тестов требуется для ее проверки. Очень «хитрую» программу трудно протестировать исчерпывающим образом. Всегда есть шанс, что какой-то «подводный камень» остался незамеченным.

С ростом памяти и быстродействия ЭВМ, с совершенствованием языков программирования и трансляторов с этих языков проблема экономичности программы становится менее острой. Все более важной качественной характеристикой программ становится их простота, наглядность, надежность. С появлением машин третьего поколения эти качества стали основными.

В конце 60-х — начале 70-х гг. XX столетия вырабатывается дисциплина, которая получила название структурного программирования. Ее появление и развитие связаны с именами Э.В. Дейкстры, Х.Д. Милса, Д.Е. Кнута и других ученых. Структурное программирование до настоящего времени остается основой технологии программирования. Соблюдение его принципов позволяет программисту быстро научиться писать ясные, безошибочные, надежные программы.

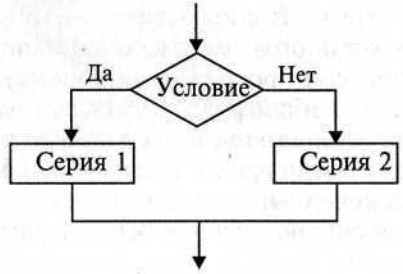
В основе структурного программирования лежит теорема, которая была строго доказана в теории программирования. Суть ее в том, что *алгоритм для решения любой логической задачи можно составить только из структур «следование, ветвление, цикл»*. Их называют базовыми алгоритмическими структурами. Из предыдущих разделов учебника вы уже знакомы с этими структурами. По сути дела, мы и раньше во всех рассматриваемых примерах программ придерживались принципов структурного программирования.

Следование — это линейная последовательность действий:



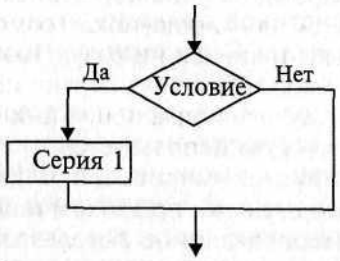
Каждый блок может содержать в себе как простую команду, так и сложную структуру, но обязательно должен иметь один вход и один выход.

Ветвление — алгоритмическая альтернатива. Управление передается одному из двух блоков в зависимости от истинности или ложности условия. Затем происходит выход на общее продолжение:



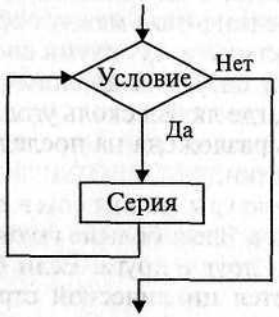

```
если условие
то   серия 1
иначе серия 2
кв
```

Неполная форма ветвления имеет место, когда на ветви «нет» пусто:




```
если условие
то   серия 1
иначе
кв
```

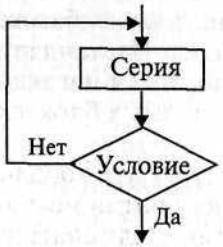
Цикл — повторение некоторой группы действий по условию. Различаются два типа цикла. Первый — цикл с предусловием (цикл-пока):




```
пока условие
повторять
нц
  серия
кц
```

Пока условие истинно, выполняется серия, образующая тело цикла.

Второй тип циклической структуры — цикл с постусловием (цикл-до):




```
повторять
    серия
до условие
```

Здесь тело цикла предшествует условию цикла. Тело цикла повторяет свое выполнение, если условие ложно. Повторение кончается, когда условие станет истинным.

Теоретически необходимым и достаточным является лишь первый тип цикла — цикл с предусловием. Любой циклический алгоритм можно построить с его помощью. Это более общий вариант цикла, чем цикл-до. В самом деле, тело цикла-до хотя бы один раз обязательно выполнится, так как проверка условия происходит после завершения его выполнения. А для цикла-пока возможен такой вариант, когда тело цикла не выполнится ни разу. Поэтому в любом языке программирования можно было бы ограничиться только циклом-пока. Однако в ряде случаев применение цикла-до оказывается более удобным, и поэтому он используется.

Иногда в литературе структурное программирование называют программированием без GOTO. Действительно, при таком подходе нет места безусловному переходу. Неоправданное использование в программах оператора GOTO лишает ее структурности, а значит, всех связанных с этим положительных свойств: прозрачности и надежности алгоритма. Хотя во всех процедурных языках программирования этот оператор присутствует, однако, придерживаясь структурного подхода, его употребления следует избегать.

Сложный алгоритм состоит из соединенных между собой базовых структур. Соединяться эти структуры могут двумя способами: *последовательным* и *вложенным*. Эта ситуация аналогична тому, что мы наблюдаем в электротехнике, где любая сколь угодно сложная электрическая цепь может быть разложена на последовательно и параллельно соединенные участки.

Вложенные алгоритмические структуры не являются аналогом параллельно соединенных проводников. Здесь больше подходит аналогия с матрешками, помещенными друг в друга. Если блок, составляющий тело цикла, сам является циклической структурой, то, значит, имеют место вложенные циклы. В свою очередь, внутренний цикл может иметь внутри себя еще один цикл и т. д. В связи с этим вводится представление о *глубине вложенности* циклов. Точно так же и ветвления могут быть вложенными друг в друга.

Структурный подход требует соблюдения стандарта в изображении блок-схем алгоритмов. Чертить их нужно так, как это делалось во всех приведенных примерах. Каждая базовая структура должна иметь один вход и один выход. Нестандартно изображенная блок-схема плохо читается, теряется наглядность алгоритма. Вот несколько примеров структурных блок-схем алгоритмов (рис. 47).

Такие блок-схемы легко читаются. Их структура хорошо воспринимается зрительно. Структуре каждого алгоритма можно дать название. У приведенных на рис. 47 блок-схем следующие названия:

1. Вложенные ветвления. Глубина вложенности равна единице.
2. Цикл с вложенным ветвлением.
3. Вложенные циклы-пока. Глубина вложенности — единица.
4. Ветвление с вложенной последовательностью ветвлений на положительной ветви и с вложенным циклом-пока на отрицательной ветви.

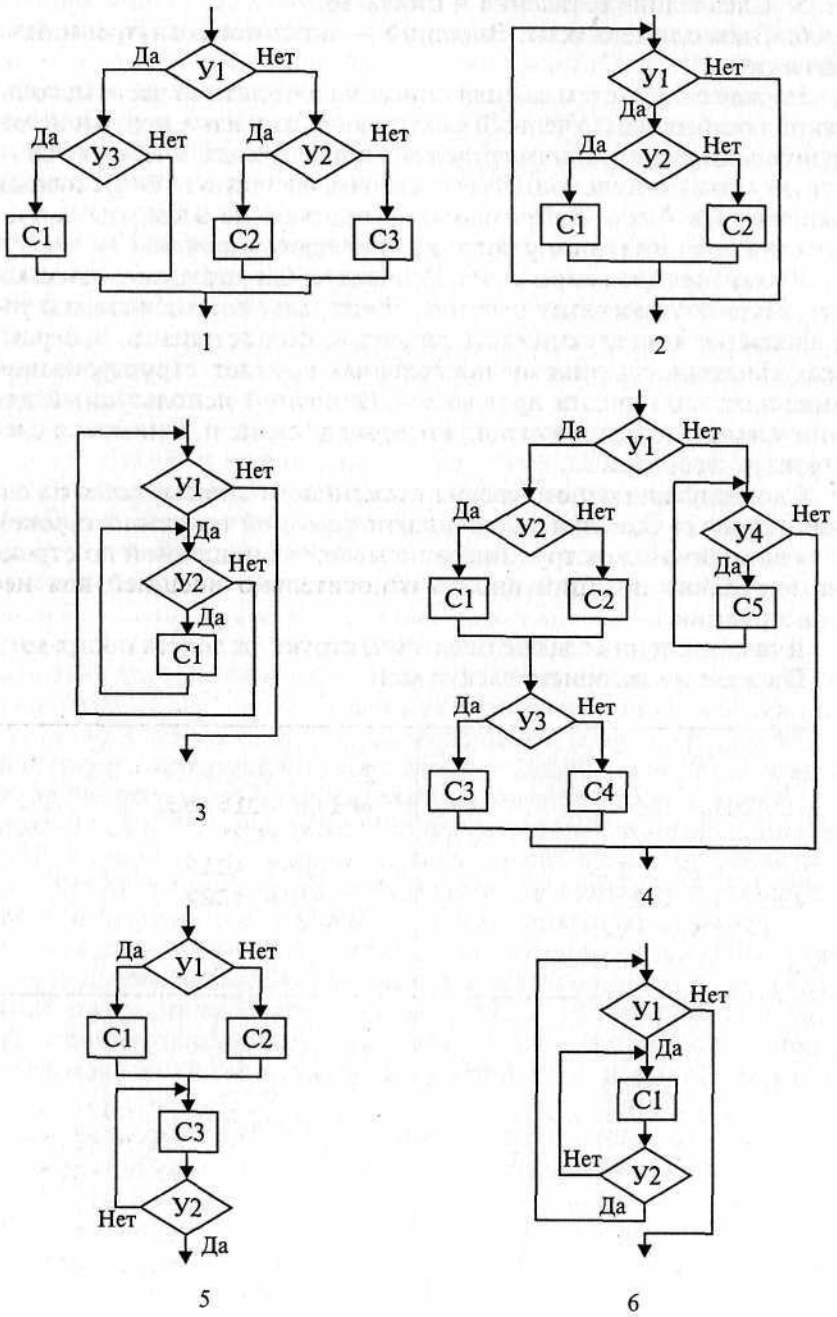


Рис. 47

5. Следование ветвления и цикла-до.

6. Вложенные циклы. Внешний — цикл-пока, внутренний — цикл-до.

Наряду с блок-схемами для описания алгоритмов часто используются псевдокоды. Учебный алгоритмический язык школьной информатики является примером такого псевдокода. Учебный АЯ — структурный псевдокод. В нем вообще отсутствует безусловный переход. Обучение составлению алгоритмов на этом языке способствует «структурному воспитанию» программиста.

Языки программирования Паскаль и Си называют языками структурного программирования. В них есть все необходимые управляющие конструкции для структурного построения программы. Наглядность такому построению придает структуризация внешнего вида текста программы. Основной используемый для этого прием — сдвиги строк, которые должны подчиняться следующим правилам:

- конструкции одного уровня вложенности записываются на одном вертикальном уровне (начинаются с одной позиции в строке);
- вложенная конструкция записывается смещенной по строке на несколько позиций вправо относительно внешней для нее конструкции.

Для приведенных выше блок-схем структура текста программы на Паскале должна быть следующей:

1.
```
if <U1>
then if <U2>
     then <C1>
else if <U2>
     then <C2>
     else <C3>
```

2.
```
while <U1> do
   if <U2>
   then <C1>
   else <C2>
```

3.
```
while <U1> do
   while <U2> do
    <C1>
```

4.
```
if <U1>
then  begin if <U2>
              then <C1>
              else <C2>;
              if<U3>
              then <C3>
              else <C4>
      end
else  while <U4> do
       <C5>
```

5.	6.

`if <U1>`
`then <C1>`
`else <C2>;`
`repeat`
`  C3`
`  until <U2>`
	`while <U1> do`
	`  repeat`
	`  <C1>`
	`  until <U2>`

Структурная методика алгоритмизации — это не только форма описания алгоритма, но это еще и *способ мышления программиста.* Создавая алгоритм, нужно стремиться составлять его из стандартных структур. Если использовать строительную аналогию, можно сказать, что структурная методика построения алгоритма подобна сборке здания из стандартных секций в отличие от складывания по кирпичику.

Еще одним важнейшим технологическим приемом структурного программирования является *декомпозиция решаемой задачи на подзадачи* — более простые с точки зрения программирования части исходной задачи. Алгоритмы решения таких подзадач называются *вспомогательными алгоритмами.* В связи с этим возможны два пути в построении алгоритма:

- «сверху вниз»: сначала строится основной алгоритм, затем вспомогательные алгоритмы;
- «снизу вверх»: сначала составляются вспомогательные алгоритмы, затем основной.

Первый подход еще называют методом *последовательной детализации*, второй — *сборочным* методом.

Сборочный метод предполагает накопление и использование библиотек вспомогательных алгоритмов, реализованных в языках программирования в виде подпрограмм, процедур, функций. При последовательной детализации сначала строится основной алгоритм, а затем в него вносятся обращения к вспомогательным алгоритмам первого уровня. После этого составляются вспомогательные алгоритмы первого уровня, в которых могут присутствовать обращения к вспомогательным алгоритмам второго уровня, и т.д. Вспомогательные алгоритмы самого нижнего уровня состоят только из простых команд.

Метод последовательной детализации применяется в любом конструировании сложных объектов. Это естественная логическая последовательность мышления конструктора: постепенное углубление в детали. В нашем случае речь идет тоже о конструировании, но только не технических устройств, а алгоритмов. Достаточно сложный алгоритм другим способом построить практически невозможно.

Методика последовательной детализации позволяет организовать работу коллектива программистов над сложным проектом. Например, руководитель группы строит основной алгоритм, а разработку вспомогательных алгоритмов и написание соответствующих подпрограмм поручает своим сотрудникам. Участники группы должны лишь договориться об интерфейсе (т.е. взаимосвязи) между разрабатываемыми программными модулями, а внутренняя организация программы — личное дело программиста.

Пример разработки программы методом последовательной детализации будет рассмотрен в следующем разделе.

5.2. Метод последовательной детализации

Суть метода была описана выше. Сначала анализируется исходная задача. В ней выделяются подзадачи. Строится иерархия таких подзадач (рис. 48).

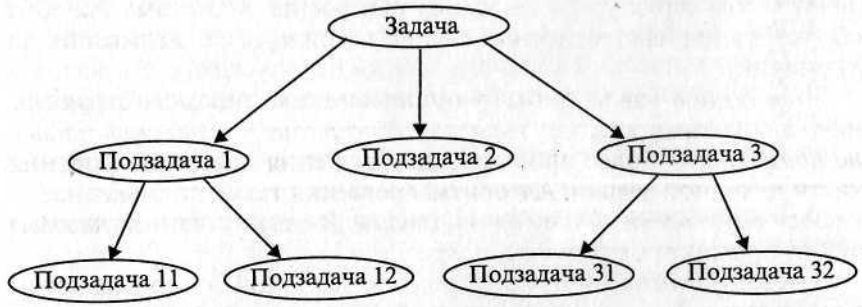


Рис. 48

Затем составляются алгоритмы (или программы), начиная с основного алгоритма (основной программы), далее — вспомогательные алгоритмы (подпрограммы) с последовательным углублением уровня, пока не получим алгоритмы, состоящие из простых команд.

Вернемся к задаче «Интерпретатор», которая рассматривалась в разд. 3.16. Напомним условие: дана исходная символьная строка, имеющая следующий вид:

`a` ⊕ `b` =

На месте `a` и `b` стоят десятичные цифры; значком ⊕ обозначен один из знаков операций: `+`, `-`, `*`. Нужно, чтобы машина вычислила это выражение и после знака = вывела результат. Операция деления не рассматривается для того, чтобы иметь дело только с целыми числами.

Сформулируем требования к программе *Interpretator*, которые сделают ее более универсальной, чем вариант, рассмотренный в разд. 3.16:

1. Операнды `a` и `b` могут быть многозначными целыми положительными числами в пределах `MaxInt`.

2. Между элементами строки, а также в начале и в конце могут стоять пробелы.

3. Программа осуществляет синтаксический контроль текста.

Ограничимся простейшим вариантом контроля: строка должна состоять только из цифр, знаков операций, знака = и пробелов.

4. Проводится семантический контроль: строка должна быть построена по схеме a ⊕ b =. Ошибка, если какой-то элемент отсутствует или нарушен их порядок.

5. Осуществляется контроль диапазона значений операндов и результата (не должны выходить за пределы `MaxInt`).

Уже из перечня требований становится ясно, что программа будет непростой. Составлять ее мы будем, используя метод последовательной детализации. Начнем с того, что представим в самом общем виде алгоритм как линейную последовательность этапов решения задачи:

1. Ввод строки.

2. Синтаксический контроль (нет ли недопустимых символов?).

3. Семантический контроль (правильно ли построено выражение?).

4. Выделение операндов. Проверка операндов на допустимый диапазон значений. Перевод в целые числа.

5. Выполнение операции. Проверка результата на допустимый диапазон.

6. Вывод результата.

Этапы 2, 3, 4, 5 будем рассматривать как подзадачи первого уровня, назвав их (и будущие подпрограммы) соответственно `Sintax`, `Semantika`, `Operand`, `Calc`. В свою очередь, для их реализации потребуется решение следующих подзадач: пропуск лишних пробелов (`Propusk`), преобразование символьной цифры в целое число (`Cifra`). Кроме того, при выделении операндов понадобится распознавать операнд, превышающий максимально допустимое значение (`Error`). Обобщая все сказанное в схематической форме, получаем некоторую структуру подзадач. Этой структуре будет соответствовать аналогичная структура программных модулей (рис. 49).

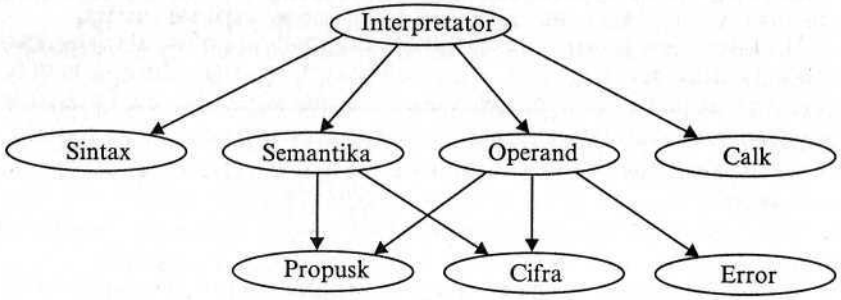


Рис. 49

Первый шаг детализации. Сначала наметим все необходимые подпрограммы, указав лишь их заголовки (спецификации). На месте тела подпрограмм запишем поясняющие комментарии (такой вид подпрограммы называется «заглушкой»). Напишем основную часть программы. А потом вернемся к детальному программированию процедур и функций. На первом этапе программирования вместо тела подпрограммы опишем ее назначение в форме комментария.

```
Program Interpretator;
Type PozInt=0..MaxInt;
Var Line: String;
      A,B: PozInt;
      Znak: Char;
      Flag: Boolean;
      Rezult: Integer;
Procedure Propusk(Stroka: String; M: Byte; Var N
                  :Byte);
Begin
{----------------------------------------
Пропускает подряд стоящие пробелы в строке Stroka,
начиная с символа номер M. В переменной N получает
номер первого символа, не являющегося пробелом
----------------------------------------}
End;
Function Cifra(C: Char): Boolean;
Begin
{----------------------------------------
Получает значение True, если символ с является
цифрой, и False - в противном случае.
----------------------------------------}
End;
Function Sintax(Stroka: String): Boolean;
Begin
{----------------------------------------
Синтаксический контроль. Получает значение True,
если в строке нет других символов, кроме цифр,
знаков операций, знака "=" и пробелов. В противном
случае - False.
----------------------------------------}
End;
Function Semantika(Stroka: String): Boolean;
Begin
{----------------------------------------
Проверяет, соответствует ли структура строки
формату "a⊕b=". Если да, то получает значение
True, если нет - False.
```

```
-----------------------------------}
End;
Procedure Operand(Stroka: String; Var A,B: PozInt;
Var Z: Char; Var Flag: Boolean);
Begin
{-----------------------------------
Выделяет операнды. Проверяет их на допустимый
диапазон значений. Переменная Flag получает
значение True, если результат проверки
положительный, и False - в противном случае.
Преобразует операнды в целые положительные числа A,
B. В переменной Z получает знак операции.
-----------------------------------}
End;
Procedure Calc(A,B: PozInt; Znak: Char; Var Rez:
Integer; Var Flag: Boolean);
Begin
{-----------------------------------
Вычисляет результат операции. Проверяет
принадлежность результата диапазону от - MaxInt до
MaxInt. Flag - признак результата проверки.
Результат операции получается в переменной Rez.
-----------------------------------}
End;
Begin {--- Начало основной части программы ---}
WriteLn("Введите выражение:");
WriteLn;
Read(Line);
If Not Sintax(Line)
Then WriteLn("Недопустимые символы")
Else If Not Semantika(Line)
     Then WriteLn("Неверное выражение")
     Else
       Begin
         Operand(Line,A,B,Znak,Flag);
         If Not Flag
         Then WriteLn("Слишком большие операнды")
         Else
           Begin
             Calc(A,B,Znak,Rezult,Flag);
             If Not Flag
             Then WriteLn("Большой результат")
             Else WriteLn(Rezult)
           End
       End
End.
```


Второй шаг детализации. Теперь будем составлять подпрограммы.

```
Procedure Propusk(Stroka: String; M: Byte; Var N
:Byte);
Begin
  N:=M;
  While (Stroka[N]=' ') And (N<Length(Stroka)) Do
  N:=N+1
End;
Function Cifra(C: Char): Boolean;
Begin
  Cifra:=(C>='0') And (C<='9')
End;
Function Sintax(Stroka: String): Boolean;
Var I: Byte;
Begin
  Sintax:=True;
  For I:=1 To Length(Stroka) Do
  If Not ((Stroka[I]=" ") Or Cifra(Stroka[I]) Or
  (Stroka[I]='+' Or (Stroka[I]='-') Or
  (Stroka[I]='*') Or (Stroka[I]='='))
  Then Sintax:=False
End;
Function Semantika(Stroka: String): Boolean;
Var I: Byte;
Begin
  Semantika:=True;
  Propusk(Stroka,1,I);
  If Not Cifra(Stroka[I])
  Then
  Begin
    Semantika:=False;
    Exit
  End;
  While Cifra(Stroka[I]) Do
  I:=I+1;
  Propusk(Stroka,I,I);
  If Not((Stroka[I]='+') Or (Stroka[I]='-')
  Or(Stroka[I]='*'))
  Then
  Begin
    Semantika:=False;
    Exit
  End;
  I:=I+1;
  Propusk(Stroka,I,I);
```

```
  If Not Cifra(Stroka[I])
  Then
  Begin
     Semantika:=False;
     Exit
  End;
  While Cifra(Stroka[I]) Do
  I:=I+1;
  Propusk(Stroka,I,I);
  If Stroka[I]<>'='
  Then
  Begin
     Semantika:=False;
     Exit
  End;
  I:=I+1;
  Propusk(Stroka,I,I);
  If I<Length(Stroka)
  Then
  Begin
     Semantika:=False;
     Exit
  End
End;
Procedure Operand(Stroka: String; Var A,B: PozInt;
Var Z: Char; Var Flag: Boolean);
Var P,Q: String;
  I: Byte;
  Code: Integer;
Function Error(S: String): Boolean;
{ Функция принимает значение True, если в строке S
находится число, превышающее максимальное целое, и
False – в противном случае }
Const Lim='32767';
Var I:Byte;
Begin
  If Length(S)>5
  Then Error:=True
  Else If Length(s)<5
       Then Error:=False
       Else Error:=S>Lim
End;
Begin
  Flag:=True;
  I:=1;
  Propusk(Stroka,I,I);
```



```
  Q:='';
  While Cifra(Stroka[I]) Do
  Begin
       Q:=Q+Stroka[I];
       I:=I+1
  End;
  If Error(Q)
  Then
  Begin
       Flag:=False;
       Exit
  End;
  Propusk(Stroka,I,I);
  Z:=Stroka[I];
  I:=I+1;
  Propusk(Stroka,I,I);
  P:='';
  While Cifra(Stroka[I]) Do
  Begin
       P:=P+Stroka[I];
       I:=I+1
  End;
  If Error(P)
  Then
  Begin
       Flag:=False;
       Exit
  End;
  Val(Q,A,Code);
  Val(P,B,Code)
{Стандртная процедура Val преобразует цифровую
строку в соответствующее число; Code - код ошибки}
End;
Procedure Calc(A,B:PozInt;Znak:Char;
Var Rez: Integer; Var Flag:Boolean);
Var X,Y,Z: Real;
Begin
  Flag:=True;
  Y:=A;
  Z:=B;
  Case  Znak Of
        '+':X:=Y+Z;
        '-':X:=Y-Z;
        '*':X:=Y*Z
  End;
If Abs(X)>MaxInt
```

```
    Then
    Begin
        Flag:=False;
        Exit
    End;
    Rez:=Round(X)
End;
```

Обратите внимание на то, что функция `Error` определена как внутренняя в процедуре `Operand`. Это объясняется тем, что указанная функция используется только в данной процедуре. Другим программным модулям она «не нужна».

Окончательно объединив тексты подпрограмм с основной программой, получаем рабочий вариант программы *Interpretator*. Теперь ее можно вводить в компьютер.

Отладка и тестирование программы. Никогда нельзя быть уверенным, что одним махом написанная программа будет верной (хотя такое и возможно, но с усложнением программы становится все менее вероятным). До окончательного рабочего состояния программа доводится в процессе *отладки*.

Ошибки могут быть «языковые», могут быть алгоритмические. Первый тип ошибок, как правило, помогает обнаружить компилятор с Паскаля. Это ошибки, связанные с нарушением правил языка программирования. Их еще называют ошибками *времени компиляции*, ибо обнаруживаются они именно во время компиляции. Сам компилятор в той или иной форме выдает пользователю сообщение о характере ошибки и ее месте в тексте программы. Исправив очередную ошибку, пользователь повторяет компиляцию. И так продолжается до тех пор, пока не будут ликвидированы все ошибки этого уровня.

Алгоритмические ошибки приводят к различным последствиям. Во-первых, могут возникнуть невыполнимые действия. Например, деление на нуль, корень квадратный из отрицательного числа, выход индекса за границы строки и т. п. Это ошибки *времени исполнения*. Они приводят к прерыванию выполнения программы. Как правило, имеются системные программные средства, помогающие в поиске таких ошибок.

Другая ситуация, когда алгоритмические ошибки не приводят к прерыванию выполнения программы. Программа выполняется до конца, получаются какие-то результаты, но они не являются верными. Для окончательной отладки алгоритма и анализа его правильности производится тестирование. *Тест* — это такой вариант решения задачи, для которого заранее известны результаты. Как правило, один тестовый вариант не доказывает правильность программы. Программист должен придумать систему тестов, построить план тестирования для исчерпывающего испытания всей программы.

Мы уже говорили о том, что качественная программа ни в каком варианте не должна завершаться аварийно. Тесты программы *Interpretator* должны продемонстрировать, что при правильном вводе исходной строки будут всегда получаться верные результаты, а при наличии ошибок (синтаксических, семантических, выхода за диапазон) будут получены соответствующие сообщения. План тестирования нашей программы может быть, например, таким:

№	Что вводим	Что должно получиться
1	`25+73=`	`98`
2	`5274–12315=`	`-7041`
3	`614*25=`	`15350`
4	`25,5+31,2=`	`Недопустимые символы`
5	`5=6–1`	`Неверное выражение`
6	`72315–256=`	`Слишком большие операнды`
7	`32000*100=`	`Большой результат`

Успешное прохождение всех тестов есть необходимое условие правильности программы. Заметим, что при этом оно необязательно является достаточным. Чем сложнее программа, тем труднее построить исчерпывающий план тестирования. Опыт показывает, что даже в «фирменных» программах в процессе эксплуатации обнаруживаются ошибки. Поэтому проблема тестирования программы — очень важная и одновременно очень сложная проблема.

5.3. Рекурсивные методы

Суть рекурсивных методов — сведение задачи к самой себе. Вы уже знаете, что как в Паскале, так и в Си существует возможность рекурсивного определения функций и процедур. Эта возможность представляет собой способ программной реализации рекурсивных алгоритмов. Однако увидеть рекурсивный путь решения задачи (рекурсивный алгоритм) часто очень непросто.

Рассмотрим классическую задачу, известную в литературе под названием «Ханойская башня» (рис. 50).

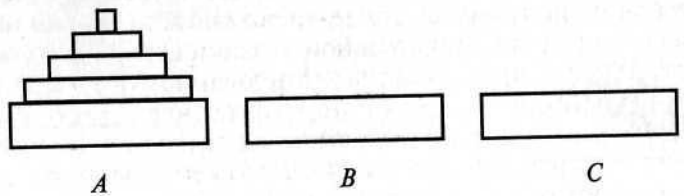


Рис. 50

На площадке (назовем ее A) находится пирамида, составленная из дисков уменьшающегося от основания к вершине размера.

Эту пирамиду в том же виде требуется переместить на площадку B. При выполнении этой работы необходимо соблюдать следующие ограничения:

- перекладывать можно только по одному диску, взятому сверху пирамиды;
- класть диск можно либо только на основание площадки, либо на диск большего размера;
- в качестве вспомогательной можно использовать площадку C.

Название «Ханойская башня» связано с легендой, согласно которой в давние времена монахи одного ханойского храма взялись переместить по этим правилам башню, состоящую из 64 дисков. С завершением их работы наступит конец света. Монахи все еще работают и, надеемся, еще долго будут работать!

Нетрудно решить эту задачу для двух дисков. Обозначая перемещения диска, например, с площадки A на B так: $A \to B$, напишем алгоритм для этого случая

$$A \to C;\ A \to B;\ C \to B.$$

Всего 3 хода! Для трех дисков алгоритм длиннее:

$$A \to B;\ A \to C;\ B \to C;\ A \to B;\ C \to A;\ C \to B;\ A \to B.$$

В этом случае уже требуются 7 ходов.

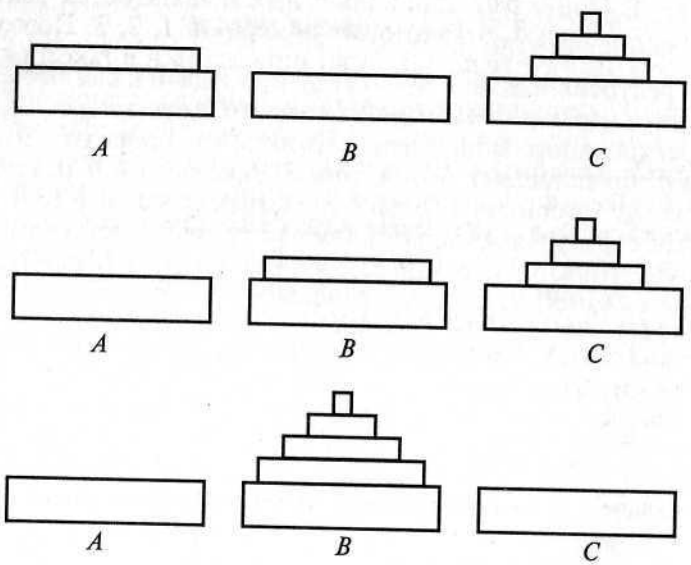


Рис. 51

Подсчитать количество ходов (N) для k дисков можно по следующей рекуррентной формуле:

$$N(1) = 1;\ N(k) = 2 \times N(k-1) + 1.$$

Например, $N(10) = 1023$, $N(20) = 104857$. А вот сколько перемещений нужно сделать ханойским монахам:

$$N(64) = 18446744073709551615.$$

Попробуйте прочитать это число.

Теперь составим программу, по которой машина рассчитает алгоритм работы монахов и выведет его для любого значения n (количества дисков). Пусть на площадке A находятся n дисков. Алгоритм решения задачи будет следующим:

1. Если $n = 0$, то ничего не делать.
2. Если $n > 0$, то переместить $n-1$ диск на C через B;
   переместить диск с A на B ($A \to B$)
   переместить $n-1$ диск с C на B через A.

При выполнении пункта 2 последовательно будем иметь три состояния (рис. 51).

Описание алгоритма имеет явно рекурсивный характер. Перемещение n дисков описывается через перемещение $n-1$ диска. А где же выход из этой последовательности рекурсивных ссылок алгоритма самого на себя? Он в пункте 1, каким бы ни показалось странным его тривиальное содержание.

А теперь построим программу на Паскале. В ней имеется рекурсивная процедура *Hanoy*, выполнение которой заканчивается только при $n = 0$. При обращении к процедуре используются фактические имена площадок, заданные их номерами: 1, 2, 3. Поэтому на выходе цепочка перемещений будет описываться в таком виде:

$$1 \to 2\ \ 1 \to 3\ \ 2 \to 3 \text{ и т.д.}$$

```
Program Monahi;
Var N: Byte;
Procedure Hanoi(N: Byte;A,B,C:Char);
Begin
   If N>0 Then
   Begin Hanoi(N-1,A,C,B);
   WriteLn(A,'=>',B);
   Hanoi(N-1,C,B,A)
   End
End;
Begin
   WriteLn("Укажите число дисков:");
   ReadLn(N);
   Hanoi(N,'1','2','3')
End.
```

Это одна из самых удивительных программ! Попробуйте воспроизвести ее на машине. Проследите, как изменяется число ходов с ростом n. Для этой цели можете сами добавить в программу счетчик ходов и в конце вывести его значение или печатать ходы с порядковыми номерами.

5.4. Методы перебора в задачах поиска

В данном разделе мы рассмотрим некоторые задачи, связанные с проблемой поиска информации. Это огромный класс задач, достаточно подробно описанный в классической литературе по программированию (см., например, книги Н. Вирта, Д. Кнута и другие).

Общий смысл задач поиска сводится к следующему: из данной информации, хранящейся в памяти ЭВМ, выбрать нужные сведения, удовлетворяющие определенным условиям (критериям).

Подобные задачи мы уже рассматривали. Например, поиск максимального числа в числовом массиве, поиск нужной записи в файле данных и т. п. Такой поиск осуществляется перебором всех элементов структуры данных и их проверкой на удовлетворение условию поиска. Перебор, при котором просматриваются все элементы структуры, называется *полным* перебором.

Полный перебор является «лобовым» способом поиска и, очевидно, не всегда самым лучшим.

Рассмотрим пример. В одномерном массиве X заданы координаты n точек, лежащих на вещественной числовой оси. Точки пронумерованы. Их номера соответствуют последовательности в массиве X. Определить номер первой точки, наиболее удаленной от начала координат.

Легко понять, что это знакомая нам задача определения номера наибольшего по модулю элемента массива X. Она решается путем полного перебора следующим образом:

```
Const N=100;
Var X: Array[1..N] Of Real; I,Number: 1..N;
Begin
{____Ввод массива X_____}
.......................
Number:=1;
For I:=2 To N Do
    If Abs(X[I])>Abs(X[Number])
    Then Number:=I;
WriteLn(Number)
End.
```

Полный перебор элементов одномерного массива производится с помощью одной циклической структуры.

А теперь такая задача: исходные данные — те же, что и в предыдущей; требуется определить пару точек, расстояние между которыми наибольшее.

Применяя метод перебора, эту задачу можно решать так: перебрать все пары точек из N данных и определить номера тех, расстояние между которыми наибольшее (наибольший модуль разности координат). Такой полный перебор реализуется через два вложенных цикла:

```
Number1:=1;
Number2:=2;
For I:=1 To N Do
For J:=1 To N Do
  If Abs(X[I]-X[J])>Abs(X[Number1]-X[Number2])
  Then
  Begin
    Number1:=I;
    Number2:=J
End;
```

Очевидно, что такое решение задачи нерационально. Здесь каждая пара точек будет просматриваться дважды, например при $i=1$, $j=2$ и $i=2, j=1$. Для случая $n=100$ циклы повторят выполнение $100 \times 100 = 10000$ раз.

Выполнение программы ускорится, если исключить повторения. Исключить также следует и случай совпадения значений i и j. Тогда число повторений цикла будет равно $\frac{n(n-1)}{2}$. При $n=100$ получается 4950.

Для исключения повторений нужно в предыдущей программе изменить начало внутреннего цикла с 1 на $i+1$. Программа примет вид:

```
Number1:=1;
Number2:=2;
For I:=1 To N Do
For J:=I+1 To N Do
  If Abs(X[I]-X[J])>Abs(X[Number1]-X[Number2])
  Then
  Begin
    Number1:=I;
    Number2:=J
  End;
```

Рассмотренный вариант алгоритма назовем перебором *без повторений*.

Замечание. Конечно, эту задачу можно было решить и другим способом, но в данном случае нас интересовал именно алгоритм,

связанный с перебором. В случае точек, расположенных не на прямой, а на плоскости или в пространстве, поиск альтернативы такому алгоритму становится весьма проблематичным.

В следующей задаче требуется выбрать все тройки чисел без повторений, сумма которых равна десяти, из массива *X*.

В этом случае алгоритм будет строиться из трех вложенных циклов. Внутренние циклы имеют переменную длину.

```
For I:=1 To N Do
For J:=I+1 To N Do
For K:=J+1 To N Do
If X[I]+X[J]+X[K]=10
Then WriteLn(X[I],X[J],X[K]);
```

А теперь представьте, что из массива *X* требуется выбрать все группы чисел, сумма которых равна десяти. В группах может быть от 1 до *n* чисел. В этом случае количество вариантов перебора резко возрастает, а сам алгоритм становится нетривиальным.

Казалось бы, ну и что? Машина работает быстро! И все же посчитаем. Число различных групп из *n* объектов (включая пустую) составляет 2^n. При $n = 100$ это будет $2^{100} \approx 10^{30}$. Компьютер, работающий со скоростью миллиард операций в секунду, будет осуществлять такой перебор приблизительно 10 лет. Даже исключение перестановочных повторений не сделает такой переборный алгоритм практически осуществимым.

Путь практической разрешимости подобных задач состоит в нахождении способов исключения из перебора бесперспективных с точки зрения условия задачи вариантов. Для некоторых задач это удается сделать с помощью алгоритма, описанного в следующем разделе.

Перебор с возвратом. Рассмотрим алгоритм перебора с *возвратом* на примере задачи о прохождении лабиринта (рис. 52).

Дан лабиринт, оказавшись внутри которого нужно найти выход наружу. Перемещаться можно только в горизонтальном и вертикальном направлениях. На рисунке показаны все варианты путей выхода из центральной точки лабиринта.

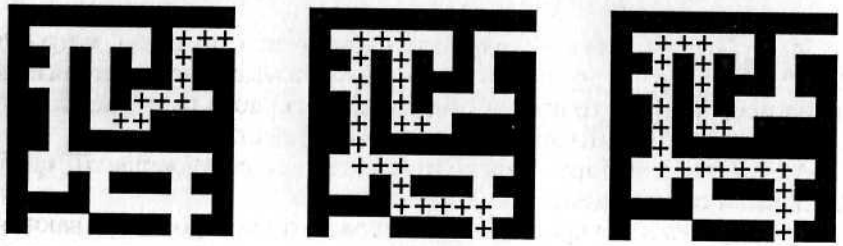


Рис. 52

Для получения программы решения этой задачи нужно решить две проблемы:

- как организовать данные;
- как построить алгоритм.

Информацию о форме лабиринта будем хранить в квадратной матрице LAB символьного типа размером $N \times N$, где N — нечетное число (чтобы была центральная точка). На профиль лабиринта накладывается сетка так, что в каждой ее ячейке находится либо стена, либо проход.

Матрица отражает заполнение сетки: элементы, соответствующие проходу, равны пробелу, а стене — какому-нибудь символу (например, букве M). Путь движения по лабиринту будет отмечаться символами +.

Например, приведенный выше рисунок (в середине) соответствует следующему заполнению матрицы LAB:

```
M M M M M M M M M M M
M   + + +     M
M M + M + M   M     M M
M   + M + M M M     M M
M M + M + M       M
M M + M + +   M M M M
M M + M M M M M M M M
M M + + +             M
M     M + M M M     M M
M M M M + + + + +     M
M             M M M + M M
```

Исходные данные — профиль лабиринта (исходная матрица LAB без крестиков); результат — все возможные траектории выхода из центральной точки лабиринта (для каждого пути выводится матрица LAB с траекторией, отмеченной крестиками).

Алгоритм перебора с возвратом еще называют *методом проб*. Суть его в следующем:

1. Из каждой очередной точки траектории просматриваются возможные направления движения в одной и той же последовательности; договоримся, что просмотр будет происходить каждый

раз против часовой стрелки — справа-сверху-слева-снизу; шаг производится в первую же обнаруженную свободную соседнюю клетку; клетка, в которую сделан шаг, отмечается крестиком.

2. Если из очередной клетки дальше пути нет (тупик), то следует возврат на один шаг назад и просматриваются еще не испробованные пути движения из этой точки; при возвращении назад покинутая клетка отмечается пробелом.

3. Если очередная клетка, в которую сделан шаг, оказалась на краю лабиринта (на выходе), то на печать выводится найденный путь.

Программу будем строить методом последовательной детализации. Первый этап детализации:

```
Program Labirint;
Const N=11; {размер лабиринта NxN клеток}
Type Field=Array[1..N,1..N] Of Char;
Var LAB: Field;
Procedure GO(LAB: Field; X,Y: Integer);
Begin
{Поиск путей из центра лабиринта до края
- каждый найденный путь печатается}
End;
Begin
{Ввод лабиринта}
GO(LAB, N Div 2+1, N Div 2+1) {начинаем с середины}
End.
```

Процедура GO пытается сделать шаг в клетку с координатами *x*, *y*. Если эта клетка оказывается на выходе из лабиринта, то пройденный путь выводится на печать. Если нет, то в соответствии с установленной выше последовательностью делается шаг в соседнюю клетку. Если клетка тупиковая, то выполняется шаг назад. Из сказанного выше следует, что процедура носит рекурсивный характер.

Запишем сначала общую схему процедуры без детализации:

```
Procedure GO(LAB: Field; X,Y: Integer);
Begin
    If {клетка (x,y) свободна}
    Then
    Begin
       {шаг на клетку (x,y)}
       If {дошли до края лабиринта}
       Then {печатается найденный путь}
       Else {попытка сделать шаг в соседние клетки
       в условленной последовательности}
       {возвращение на один шаг назад}
    End
End;
```


Для вывода найденных траекторий составляется процедура PRINTLAB.

В окончательном виде программа будет выглядеть так:

```
Program Labirint;
Const N=11; {размер лабиринта NxN клеток}
Type Field=Array[1..N, 1..N] Of Char;
Var LAB: Field; X,Y: Integer;
Procedure PRINTLAB(LAB: Field);
{Печать найденного пути в лабиринте}
Var X,Y: Integer;
Begin
     For X:=1 To N Do
     Begin
          For Y:=1 To N Do
            Write(LAB[X,Y]);
            WriteLn
     End;
     WriteLn
End; {печати}
Procedure GO(LAB: Field; X,Y: Integer);
Begin
     If LAB[X,Y]=' ' {если клетка свободна}
     Then
     Begin
       LAB[X,Y]:='+'; {делается шаг}
       If (X=1)Or(X=N)Or(Y=1)Or(Y=N) {край}
       Then
        PRINTLAB(LAB){печатается найденный путь}
       Else
         Begin {поиск следующего шага}
           GO(LAB,X+1,Y);
           GO(LAB,X, Y+1);
           GO(LAB,X-1,Y);
           GO(LAB,X,Y-1)
         End;
       LAB[X,Y]:='' {возвращение назад}
     End
End; {процедуры GO}
Begin {основной программы}
      {ввод лабиринта}
       For X:=1 To N Do
       Begin
          For Y:=1 To N Do
              Read(LAB[X,Y]);
              ReadLn
       End;
```

```
    GO(LAB,N Div 2+1,N Div 2+1)  {начинаем с
                                  середины}
End.
```

Еще один пример красивой программы с использованием рекурсивного определения процедуры (вспомните ханойскую башню!).

Схема алгоритма данной программы типична для метода перебора с возвратом. По аналогичным алгоритмам решаются, например, популярные задачи об обходе шахматной доски фигурами или о расстановке фигур на доске так, чтобы они «не били» друг друга; множество задач оптимального выбора (задачи о коммивояжере, об оптимальном строительстве дорог и т.п.).

Замечание. Из-за использования массива `LAB` в качестве параметра-значения в процедуре `GO` могут возникнуть проблемы с памятью при реализации программы на ЭВМ. В таком случае можно перейти к глобальной передаче массива.

Упражнения

1. Даны декартовы координаты N точек на плоскости. Составить программы решения следующих задач:

а) найти две самые близкие друг к другу точки;

б) найти две самые удаленные друг от друга точки;

в) найти три точки, лежащие в вершинах треугольника с наибольшим периметром;

г) найти две ближайшие точки, отрезок между которыми может служить радиусом окружности, заключающей внутри себя все остальные точки; указать, какая из них является центральной.

2. Изменить программу `Labirint` таким образом, чтобы на печать выводился лишь кратчайший путь из центра лабиринта до края.

3. Составить программу, в соответствии с которой шахматный конь обойдет всю доску, побывав на каждом поле всего один раз.

4. Составить программу расстановки на шахматной доске восьми ферзей так, чтобы они не угрожали друг другу.

5.5. Эвристические методы

Под эвристическими понимаются такие методы, правильность которых строго не доказывается. Они выглядят правдоподобными; кажется, что в большинстве случаев они должны давать верные решения. На уровне экспертной оценки алгоритма часто не удается придумать контрпример, доказывающий ошибочность или неуниверсальность метода. Это, разумеется, не является строгим обоснованием правильности метода. Тем не менее практика использования эвристических методов дает положительные результаты.

Эвристические методы разнообразны, поэтому нельзя описать какую-то общую схему их разработки. Чаще всего они применяются совместно с методами перебора для сокращения числа проверяемых вариантов. Некоторые варианты согласно выбранной эвристике считаются заведомо бесперспективными и не проверяются. Такой подход ускоряет работу алгоритма по сравнению с полным перебором. Платой за это является отсутствие гарантии того, что выбрано правильное или наилучшее из всех возможных решение.

5.6. Сложность алгоритмов

Традиционно принято оценивать степень сложности алгоритма по объему используемых им основных ресурсов компьютера: процессорного времени и оперативной памяти. В связи с этим вводятся такие понятия, как временная сложность алгоритма и объемная сложность алгоритма.

Параметр временной сложности становится особенно важным для задач, предусматривающих интерактивный режим работы программы, или для задач управления в режиме реального времени. Часто программисту, составляющему программу управления каким-нибудь техническим устройством, приходится искать компромисс между точностью вычислений и временем работы программы. Как правило, повышение точности ведет к увеличению времени.

Объемная сложность программы становится критической, когда объем обрабатываемых данных оказывается на пределе объема оперативной памяти ЭВМ. На современных компьютерах острота этой проблемы снижается благодаря как росту объема ОЗУ, так и эффективному использованию многоуровневой системы запоминающих устройств. Программе оказывается доступной очень большая, практически неограниченная область памяти (виртуальная память). Недостаток основной памяти приводит лишь к некоторому замедлению работы из-за обменов с диском. Используются приемы, позволяющие минимизировать потери времени при таком обмене. Это использование кэш-памяти и аппаратного просмотра команд программы на требуемое число ходов вперед, что позволяет заблаговременно переносить с диска в основную память нужные значения. Исходя из сказанного можно заключить, что минимизация емкостной сложности не является первоочередной задачей. Поэтому в дальнейшем мы будем интересоваться в основном временной сложностью алгоритмов.

Время выполнения программы пропорционально числу исполняемых операций. Разумеется, в размерных единицах времени (секундах) оно зависит еще и от скорости работы процессора (так-

товой частоты). Для того чтобы показатель временной сложности алгоритма был инвариантен относительно технических характеристик компьютера, его измеряют в относительных единицах. Обычно временная сложность оценивается числом выполняемых операций.

Как правило, временная сложность алгоритма зависит от исходных данных. Это может быть зависимость как от величины исходных данных, так и от их объема. Если обозначить значение параметра временной сложности алгоритма α символом T_α, а буквой V обозначить некоторый числовой параметр, характеризующий исходные данные, то временную сложность можно представить как функцию $T_\alpha(V)$. Выбор параметра V зависит от решаемой задачи или от вида используемого алгоритма для решения данной задачи.

Пример 1. Оценим временную сложность алгоритма вычисления факториала целого положительного числа.

```
Function Factorial(x:Integer): Integer;
Var m,i: Integer;
Begin  m:=1;
    For i:=2 To x Do m:=m*i;
    Factorial:=m
End;
```

Подсчитаем общее число операций, выполняемых программой при данном значении x. Один раз выполняется оператор `m:=1`; тело цикла (в котором две операции: умножение и присваивание) выполняется $x - 1$ раз; один раз выполняется присваивание `Factorial:=m`. Если каждую из операций принять за единицу сложности, то временная сложность всего алгоритма будет $1 + 2(x - 1) + 1 = 2x$. Отсюда понятно, что в качестве параметра V следует принять значение x. Функция временной сложности получилась следующей:

$$T_\alpha(V) = 2V.$$

В этом случае можно сказать, что временная сложность зависит линейно от параметра данных — величины аргумента функции факториал.

Пример 2. Вычисление скалярного произведения двух векторов $A = (a_1, a_2, ..., a_k)$, $B = (b_1, b_2, ..., b_k)$.

```
AB:=0;
For i:=1 To k Do  AB:=AB+A[i]*B[i];
```

В этой задаче объем входных данных $n = 2k$. Количество выполняемых операций $1 + 3k = 1 + 3(n/2)$. Здесь можно взять $V = k = n/2$. Зависимости сложности алгоритма от значений элементов векторов A и B нет. Как и в предыдущем примере, здесь можно говорить о линейной зависимости временной сложности от параметра данных.

С параметром временной сложности алгоритма обычно связывают две теоретические проблемы. Первая состоит в поиске ответа на вопрос: до какого предела значения временной сложности можно дойти, совершенствуя алгоритм решения задачи? Этот предел зависит от самой задачи и, следовательно, является ее собственной характеристикой.

Вторая проблема связана с классификацией алгоритмов по временной сложности. Функция $T_\alpha(V)$ обычно растет с ростом V. Как быстро она растет? Существуют алгоритмы с линейной зависимостью T_α от V (как это было в рассмотренных нами примерах), с квадратичной зависимостью и с зависимостью более высоких степеней. Такие алгоритмы называются полиномиальными. А существуют алгоритмы, сложность которых растет быстрее любого полинома. Проблема, которую часто решают теоретики — исследователи алгоритмов, заключается в следующем вопросе: возможен ли для данной задачи полиномиальный алгоритм?

5.7. Методы сортировки данных

Существует традиционное деление алгоритмов на численные и нечисленные. Численные алгоритмы предназначены для математических расчетов: вычисления по формулам, решения уравнений, статистической обработки данных и т. п. В таких алгоритмах основным видом обрабатываемых данных являются числа. Нечисленные алгоритмы имеют дело с самыми разнообразными видами данных: символьной, графической, мультимедийной информацией. К этой категории относятся многие алгоритмы системного программирования (трансляторы, операционные системы), систем управления базами данных, сетевого программного обеспечения и т. д.

Для программных продуктов второй категории наиболее часто используемыми являются алгоритмы сортировки данных — упорядочения информации по некоторому признаку. От эффективности, прежде всего скорости, их выполнения во многом зависит эффективность работы всей программы.

Различают алгоритмы внутренней сортировки — во внутренней памяти и алгоритмы внешней сортировки — сортировки файлов. Далее мы будем рассматривать только внутреннюю сортировку.

Как правило, сортируемые данные располагаются в массивах. В простейшем случае это числовые массивы. Однако для нечисленных алгоритмов более характерна ситуация, когда сортируется массив записей (в терминологии Паскаля) или массив структур (в терминологии Си). Поле, по значению которого производится сортировка, называется *ключом сортировки*. Обычно оно имеет числовой тип. Например, массив сортируемых записей содержит

два поля: наименование товара и количество товара на складе. В программе на Паскале он описан так:

```
Const n=1000;
Type tovar=Record
             name: String;
             key: Integer
           End;
Var A: array[1..n] Of tovar;
```

Сортировка производится либо по возрастанию, либо по убыванию значения ключа `A[i].key`.

Во всех дальнейших примерах программ предполагается, что приведенные выше описания в программе присутствуют глобально и область их действия распространяется на процедуры сортировки. Хотя все примеры приводятся на Паскале, но по тому же принципу можно разработать функции сортировки на Си/Си++.

Алгоритм сортировки «методом пузырька» рассматривался в разделе 3.17. Здесь мы обсудим два алгоритма: сортировку простым включением и быструю сортировку.

Сортировка простым включением. Предположим, что на некотором этапе работы алгоритма левая часть массива с 1-го по $(i-1)$-й элемент включительно является отсортированной, а правая часть с i-го по n-й элемент остается такой, какой она была в первоначальном, неотсортированном массиве. Очередной шаг алгоритма заключается в расширении левой части на один элемент и, соответственно, сокращении правой части. Для этого берется первый элемент правой части (с индексом i) и вставляется на подходящее ему место в левую часть так, чтобы упорядоченность левой части сохранилась.

Процесс начинается с левой части, состоящей из одного элемента `A[1]`, а заканчивается, когда правая часть становится пустой.

```
Procedure StraightInsertion;
Var i,j: Integer; x: tovar;
Begin  For i:=2 To n Do
Begin x:=A[i]; {В переменной x запоминается
значение, которое нужно поставить на свое место в
левой части}
      j:=i-1; {Правый край левой части}
      While (x.key<A[j].key) and (j>=1) Do
      Begin A[j+1]:=A[j]; j:=j-1; {Продвижение
"дырки" в левой части массива справа налево до той
позиции, в которую должен быть включен элемент
      A[i]}
      End;
      A[j+1]:=x {включение A[i] в "дырку" в левой
части}
```



```
End
End;
```

Теперь оценим сложность алгоритма сортировки простым включением. Очевидно, что временная сложность зависит как от размера сортируемого массива, так и от его исходного состояния в смысле упорядоченности элементов. Временная сложность будет минимальной, если исходный массив уже отсортирован в нужном порядке значений ключа (в данном случае — по возрастанию). Максимальное значение сложности будет соответствовать противоположной упорядоченности исходного массива, т.е. упорядоченности исходного массива по убыванию значений ключа. Обычно для алгоритмов сортировки временная сложность оценивается количеством пересылок элементов.

Оценим величину минимальной временной сложности алгоритма. Если массив уже отсортирован, то тело цикла `while` не будет выполняться ни разу. Выполнение процедуры сведется к работе следующего цикла:

```
For i:=2 To n do
Begin x:=A[i];
      j:=i-1;
      A[j+1]:=x
End;
```

Поскольку тело цикла `for` исполняется $n-1$ раз, то число пересылок элементов массива

$$M_{\min} = 2(n-1),$$

а число сравнений ключей равно

$$C_{\min} = n-1.$$

Сложность алгоритма будет максимальной, если исходный массив упорядочен по убыванию. Тогда каждый элемент `A[i]` будет «прогоняться» к началу массива, т.е. устанавливаться в первую позицию. Цикл `while` выполнится 1 раз при $i=2$, 2 раза при $i=3$ и т.д., $n-1$ раз при $i=n$. Таким образом, общее число пересылок записей равно:

$$M_{\max} = 2(n-1) + \sum_{i=2}^{n}(i-1) = 2(n-1) + n(n-1)/2 = \frac{1}{2}(n^2+3n-4).$$

Более подходящей для реальной ситуации является *средняя оценка сложности*. Для ее вычисления надо предположить, что все элементы исходного массива — случайные числа и их значения никак не связаны с их номерами. В таком случае результат очередной проверки условия `x.key<A[j].key` в цикле `While` также является случайным.

Разумно допустить, что среднее число выполнений цикла While для каждого конкретного значения i равно $i/2$, т.е. в среднем каждый раз приходится просматривать половину последовательности до тех пор, пока не найдется подходящее место для очередного элемента. Тогда формула для среднего числа пересылок (средняя оценка сложности) будет следующей:

$$M_{ср} = 2(n-1) + \sum_{i=2}^{n} i/2 = 2(n-1) + (n-2)(n-1)/4 = \frac{1}{4}(n^2 + 9n - 10).$$

Как максимальная, так и средняя оценка сложности алгоритма квадратична (является полиномом второй степени) по параметру n — размеру сортируемого массива.

Алгоритм быстрой сортировки. Этот алгоритм был разработан Э. Хоаром. В алгоритме быстрой сортировки используются три идеи:

- разделение сортируемого массива на 2 части, левую и правую;
- взаимное упорядочение двух частей (подмассивов) так, чтобы все элементы левой части не превосходили элементов правой части;
- рекурсия, при которой подмассив упорядочивается точно таким же способом, как и весь массив.

Для разделения массива на две части нужно выбрать некоторое «барьерное» значение ключа. Это значение должно удовлетворять единственному условию: лежать в диапазоне значений для данного массива (т.е. между минимальной и максимальной величиной). За «барьер» можно выбрать значение ключа любого элемента массива, например первого, или последнего, или находящегося в середине.

Далее нужно сделать так, чтобы в левом подмассиве оказались все элементы с ключом, меньшим барьера, а в правом — с большим. Затем, просматривая массив слева направо, необходимо найти позицию первого элемента с ключом, большим барьера, а просматривая справа налево — найти первый элемент с ключом, меньшим барьера. Следует поменять эти значения, затем продолжить встречное движение до следующей пары элементов, предназначенных для обмена. Необходимо повторять эту процедуру, пока индексы левого и правого просмотров не совпадут. Место совпадения станет границей между двумя взаимно упорядоченными подмассивами. Далее алгоритм рекурсивно применяется к каждому из подмассивов (левому и правому). В конечном счете приходим к совокупности из n взаимно упорядоченных одноэлементных массивов, которые делить дальше невозможно. Эта совокупность образует один полностью упорядоченный массив. Сортировка завершена!

```
Procedure Quicksort;
  Procedure Sort(L,R:Integer);
```



```
Var i,j,bar: Integer; w:tovar;
Begin i:=L; j:=R;
    bar:=A[(L+R)div2].key;{Установка барьера}
    Repeat
      {Поиск элемента слева для обмена}
      While A[i].key<bar Do i:=i+1;
      {Поиск элемента справа для обмена}
      While A[j].key>bar Do j:=j-1;
      {Обмен элементов и смещение по массиву}
      If i<=j Then
      Begin w:=A[i]; A[i]:=A[j]; A[j]:=w;
            i:=i+1; j:=j-1
      End;
    Until i>j;
{сформированы взаимно упорядоченные подмассивы}
    {Сортировка левого подмассива}
    If L<j Then Sort(L,j);
    {Сортировка правого подмассива}
    If i<R Then Sort(i,R);
End; {Sort}
Begin Sort(1,n)
End. {Quicksort}
```

Сложность алгоритма быстрой сортировки. Исследование временной сложности алгоритма быстрой сортировки является очень трудоемкой задачей, и поэтому мы здесь приводить его не будем. Рассмотрим лишь окончательный результат этого анализа. Временная сложность T как функция от n — размера массива — по порядку величины выражается следующей формулой:

$$T(n) = O(n \ln(n)).$$

Здесь использовано принятое в математике обозначение: $O(x)$ обозначает величину порядка x. Следовательно, временная сложность алгоритма быстрой сортировки есть величина порядка $n \ln(n)$. Эта величина для целых положительных n меньше, чем n^2 (вспомним, что алгоритм сортировки простым включением имеет сложность порядка n^2). И чем больше значение n, тем эта разница существеннее. Например:

$n = 10$;	$n^2 = 100$;	$n \ln(n) = 23{,}03$;
$n = 100$;	$n^2 = 10\,000$;	$n \ln(n) = 460{,}52$.

Определить три точки, являющиеся вершинами треугольника, для которого разность точек вне его и внутри является минимальной.

14. Некоторое число содержится в каждом из трех целочисленных неубывающих массивов $x[1] \le ... \le x[p]$, $y[1] \le ... \le y[q]$, $z[1] \le ... \le z[r]$. Найти одно из таких чисел. Число действий должно быть порядка $p+q+r$.

15. Выяснить, есть ли одинаковые числа в каждом из трех целочисленных неубывающих массивов $x[1] \le ... \le x[p]$, $y[1] \le ... \le y[q]$, $z[1] \le ... \le z[r]$. Найти одно из таких чисел или сообщить о его отсутствии.

16. Дана целочисленная таблица $A[n]$. Найти наименьшее число K элементов, которые можно выкинуть из данной последовательности, так чтобы осталась возрастающая подпоследовательность.

17. Разделить массив на две части, поместив в первую элементы, большие среднего арифметического их суммы, а во вторую — меньшие (части не сортировать).

Сортировка массивов

1. Заданы два одномерных массива с различным количеством элементов и натуральное число k. Объединить их в один массив, включив второй массив между k-м и $(k+1)$-м элементами первого, при этом не используя дополнительный массив.

2. Даны две последовательности $a_1 \le a_2 \le ... \le a_n$ и $b_1 \le b_2 \le ... \le b_m$. Образовать из них новую последовательность чисел так, чтобы она тоже была неубывающей. *Примечание.* Дополнительный массив не использовать.

3. **Сортировка выбором.** Дана последовательность чисел $a_1, a_2, ..., a_n$. Требуется переставить элементы так, чтобы они были расположены по убыванию. Для этого в массиве, начиная с первого, выбирается наибольший элемент и ставится на первое место, а первый — на место наибольшего. Затем, начиная со второго, эта процедура повторяется. Написать алгоритм сортировки выбором.

4. **Сортировка обменами.** Дана последовательность чисел $a_1, a_2, ..., a_n$. Требуется переставить числа в порядке возрастания. Для этого сравниваются два соседних числа a_i и a_{i+1}. Если $a_i > a_{i+1}$, то делается перестановка. Так продолжается до тех пор, пока все элементы не станут расположены в порядке возрастания. Составить алгоритм сортировки, подсчитывая при этом количества перестановок.

5. **Сортировка вставками.** Дана последовательность чисел $a_1, a_2, ..., a_n$. Требуется переставить числа в порядке возрастания. Делается это следующим образом. Пусть $a_1, a_2, ..., a_i$ — упорядоченная последовательность, т.е. $a_1 \le a_2 \le ... \le a_i$. Берется следующее число a_{i+1} и вставляется в последовательность так, чтобы новая

последовательность была тоже возрастающей. Процесс производится до тех пор, пока все элементы от $i+1$ до n не будут перебраны. *Примечание.* Место помещения очередного элемента в отсортированную часть производить с помощью двоичного поиска. Двоичный поиск оформить в виде отдельной функции.

6. **Сортировка Шелла.** Дан массив n действительных чисел. Требуется упорядочить его по возрастанию. Делается это следующим образом: сравниваются два соседних элемента a_i и a_{i+1}. Если $a_i \le a_{i+1}$, то продвигаются на один элемент вперед. Если $a_i > a_{i+1}$, то производится перестановка и сдвигаются на один элемент назад. Составить алгоритм этой сортировки.

7. Пусть даны две неубывающие последовательности действительных чисел $a_1 \le a_2 \le ... \le a_n$ и $b_1 \le b_2 \le ... \le b_m$. Требуется указать те места, на которые нужно вставлять элементы последовательности $b_1, b_2, ..., b_m$ в первую последовательность так, чтобы новая последовательность оставалась возрастающей.

8. Даны дроби $\frac{p_1}{q_1}, \frac{p_2}{q_2}, ..., \frac{p_n}{q_n}$ (p_i, q_i — натуральные). Составить программу, которая приводит эти дроби к общему знаменателю и упорядочивает их в порядке возрастания.

9. **Алгоритм фон Неймана.** Упорядочить массив $a_1, a_2, ..., a_n$ по неубыванию с помощью алгоритма сортировки слияниями:

1) каждая пара соседних элементов сливается в одну группу из двух элементов (последняя группа может состоять из одного элемента);

2) каждая пара соседних двухэлементных групп сливается в одну четырехэлементную группу и т.д.

При каждом слиянии новая укрупненная группа упорядочивается.

10. **Сортировка подсчетом.** Выходной массив заполняется значениями -1. Затем для каждого элемента определяется его место в выходном массиве путем подсчета количества элементов строго меньших данного. Естественно, что все одинаковые элементы попадают на одну позицию, за которой следует ряд значений -1. После этого оставшиеся в выходном массиве позиции со значением -1 заполняются копией предыдущего значения.

11. **«Хитрая» сортировка.** Из массива путем однократного просмотра выбирается последовательность элементов, расположенных в порядке возрастания, переносится в выходной массив и заменяется во входном на -1. Затем оставшиеся элементы включаются в полученную упорядоченную последовательность методом «погружения», когда очередной элемент путем ряда обменов «погружается» до требуемой позиции в уже упорядоченную часть массива.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов В.Г., Трифонов Н.П., Трифонова Г.Н. Введение в язык Паскаль. — М.: Наука, 1988.
2. Березин Б.И., Березин С.Б. Начальный курс С и С++. — М.: ДИАЛОГ-МИФИ, 1996.
3. Бондарев В.М., Рублинецкий В.И., Качко Е.Г. Основы программирования. — Харьков: Фолио, Ростов н/Д: Феникс, 1997.
4. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1981.
5. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989.
6. Гладков В.П. Задачи по информатике на вступительном экзамене в вуз и их решения: Учебное пособие. — Пермь: Перм. техн. ун-т, 1994.
7. Гладков В.П. Курс лабораторных работ по программированию: Учебное пособие для специальностей электротехнического факультета ПГТУ. Пермь: Перм. техн. ун-т, 1998.
8. Грогоно П. Программирование на языке Паскаль. — М.: Мир, 1982.
9. Дагене В.А., Григас Г.К., Аугутис К.Ф. 100 задач по программированию. — М.: Просвещение, 1993.
10. Епашников А.М., Епашников В.А. Программирование в среде Турбо Паскаль 7.0. — М.: МИФИ, 1994.
11. Заварыкин В.М., Житомирский В.Г., Лапчик М.П. Основы информатики и вычислительной техники. — М.: Просвещение, 1989.
12. Задачи по программированию / С.А.Абрамов, Г.Г.Гнездилова, Е.Н.Капустина, М.И.Селюн. — М.: Наука, 1988.
13. Зубов В.С. Программирование на языке Turbo Pascal (версии 6.0 и 7.0). — М.: Информационно-издательский дом «Филинъ», 1997.
14. Зуев Е.А. Практическое программирование на языке Turbo Pascal 6.0, 7.0. — М.: Радио и связь, 1994.
15. Информатика. Задачник-практикум: В 2 т. / Под ред. И.Г.Семакина, Е.К.Хеннера. — М.: Лаборатория Базовых Знаний, 1999.
16. Йенсен К., Вирт Н. Паскаль — руководство для пользователей и описание языка. — М.: Мир, 1982.
17. Касаткин В.Н. Информация. Алгоритмы. ЭВМ. — М.: Просвещение, 1991.
18. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. — М.: Финансы и статистика, 1992.
19. Культин Н.Б. Программирование в Turbo Pascal и Delphi. — СПб.: BHV — Санкт-Петербург, 1998.
20. Ляхович В.Ф. Руководство к решению задач по основам информатики и вычислительной техники. — М.: Высшая школа, 1994.

21. Марченко А.И., Марченко Л.А. Программирование в среде Turbo Pascal 7.0 / Под ред. В.П.Тарасенко. — Киев: ВЕК+; М.: Бином Универсал, 1998.
22. Миков А.И. Информатика. Введение в компьютерные науки. — Пермь: Изд-во ПГУ, 1998.
23. Могилев А.В., Пак Н.И., Хеннер Е.К. Информатика: Учеб. пособие для студ. пед. вузов / Под ред. Е.К.Хеннера. — М.: Изд. центр «Академия», 1999.
24. Олимпиады по информатике. Задачи и решения: методические рекомендации для учителей и учащихся школ. — Красноярск, 1991.
25. Основы информатики и вычислительной техники. Ч. 1, 2 / Под ред. А.П.Ершова и В.М.Монахова. — М.: Просвещение, 1988.
26. Основы информатики и вычислительной техники в базовой школе / Л.А.Залогова, С.В.Русаков , И.Г.Семакин и др. — Пермь, 1995.
27. Паппас К., Мюррей У. Программирование на С и С++. — Киев: «Ирина»; BHV, 2000.
28. Пильщиков В.Н. Сборник упражнений по языку Паскаль. — М.: Наука, 1989.
29. Подбельский В.В., Фомин С.С. Программирование на языке Си. — М.: Финансы и статистика, 1999.
30. Подбельский В.В. Язык Си++. — М.: Финансы и статистика, 1996.
31. Попов Б.В. TURBO PASCAL для школьников. Версия 7.0. — М.: Финансы и статистика, 1996.
32. Романов Е.Л. http://ermak.cs.nstu.ru/cprog — электронный учебник по дисциплине «Информатика».
33. Сборник задач по программированию / Авт.-сост. А.П.Шестаков. — Пермь: Перм. ун-т, 1999.
34. Сычев Н.А. Задания для вступительных экзаменов по информатике в НГУ // Информатика и образование. — 1995. — №2.
35. Хонсбергер Р. Математические изюминки. — М.: Наука, 1992.
36. Шень А. Программирование: теоремы и задачи. — М.: МЦНМО, 1995.
37. http://eldorado.mirea.ac.ru/zao5