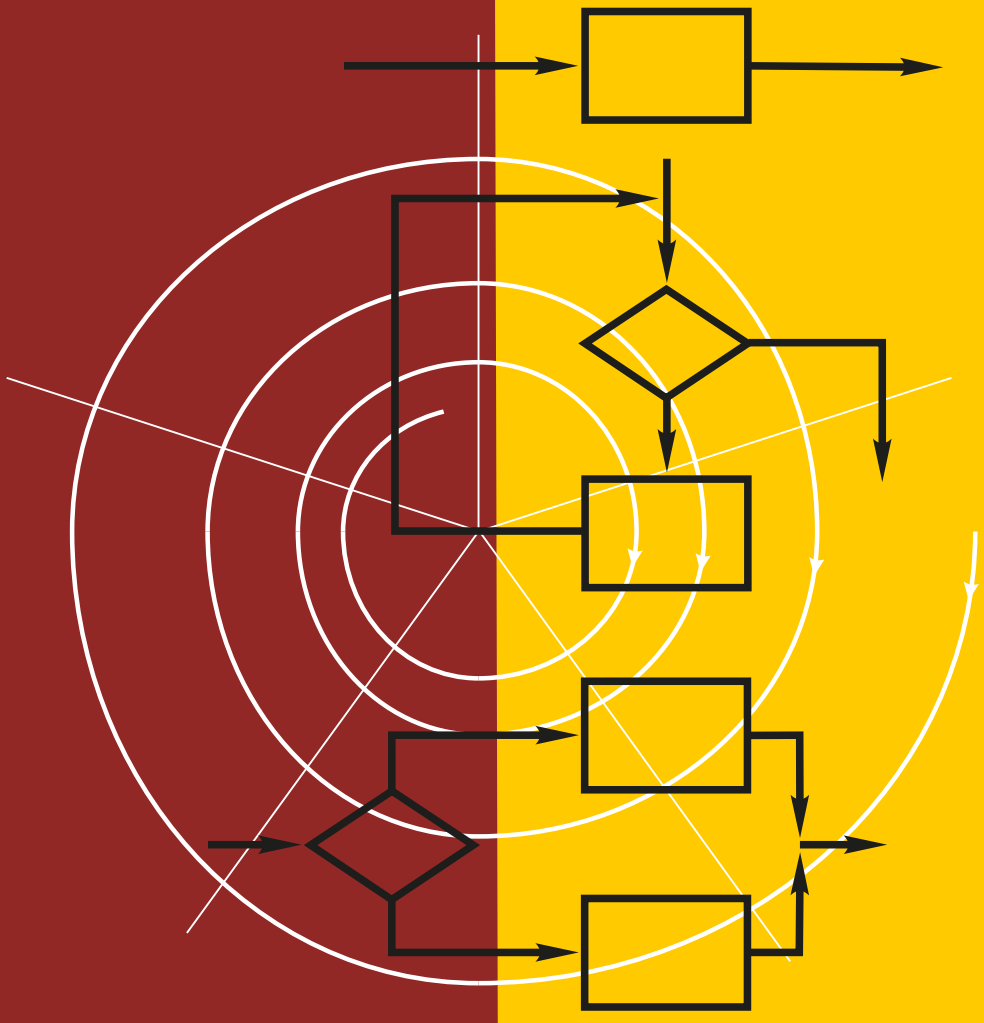
Д. ЗЛАТОПОЛЬСКИЙ

ПРОГРАММИРОВАНИЕ:

ТИПОВЫЕ ЗАДАЧИ, АЛГОРИТМЫ, МЕТОДЫ



Лаборатория ЗНАНИЙ

Д. ЗЛАТОПОЛЬСКИЙ

ПРОГРАММИРОВАНИЕ:

ТИПОВЫЕ ЗАДАЧИ, АЛГОРИТМЫ, МЕТОДЫ

4-е издание, электронное

Москва
Лаборатория знаний
2020

УДК 004.42
ББК 32.973-018
З-67

Златопольский Д. М.

З-67 Программирование: типовые задачи, алгоритмы, методы / Д. М. Златопольский. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 226 с. — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-789-9

Эта книга для тех, кто хочет научиться программировать. В ней представлена методика решения типовых задач программирования, не привязанная к конкретному языку. Разъяснения по методике решения задач и программы приведены на школьном алгоритмическом языке. Русский синтаксис делает программы понятными и легко переносимыми на любой язык программирования.

Для школьников и студентов, начинающих изучать программирование или знакомых с его основами, а также для всех, кого заинтересует решение сложных задач, в том числе встречающихся на олимпиадах по программированию. Книга будет полезна преподавателям различных учебных заведений и студентам педагогических вузов.

УДК 004.42
ББК 32.973-018

Деривативное издание на основе печатного аналога: Программирование: типовые задачи, алгоритмы, методы / Д. М. Златопольский. — М. : БИНОМ. Лаборатория знаний, 2007. — 223 с. : ил. — ISBN 978-5-94774-461-3.

ISBN 978-5-00101-789-9

1. Какая программа лучше?

Если вас, уважаемый читатель, спросят: «Какая программа лучше — меньшая по объему, работающая быстрее, или та, исходный текст которой более понятен?», то, подумав, вы, скорее всего, ответите: «Более быстрая, понятная и занимающая меньше места». Конечно, вы при этом будете правы. Но, к сожалению, такие программы встречаются крайне редко. Как правило, улучшение одного из показателей (размера, скорости работы и понятности листинга) приводит к ухудшению другого.

Рассмотрим в качестве примера задачу определения наибольшего общего делителя (НОД) двух натуральных чисел.

Сначала составим программу по принципу, который по-английски называется «KISS». Правда, сразу скажу: с поцелуями он ничего общего не имеет. «KISS» — это всего лишь аббревиатура фразы: «Keep it simple, stupid», что в переводе означает: «Делай это проще, дурачок» ☺. Этот принцип призывает нас решать поставленные задачи более простыми методами, прибегая к изощренным алгоритмам и программным средствам только в крайнем случае.

Будем рассуждать так. Если два заданных числа `a` и `b` равны между собой, то их НОД равен одному из них; в противном случае НОД равен либо минимальному из заданных чисел, либо некоему целому числу, которое меньше этого минимального.

Программа, реализующая такие рассуждения, на школьном алгоритмическом языке будет иметь следующий вид, где `mod` — функция, определяющая остаток[1]:

```
алг Определение_НОД
нач цел a, b, мин, НОД
| |Ввод чисел
| вывод "Задайте первое число"
| ввод a
| вывод нс, "Задайте второе число"
| ввод b
```

[1] В других языках программирования для определения остатка используется не функция, а специальная операция (как правило, знак этой операции — `mod`).

```
| если a = b
| |то
| |   НОД = a
| |иначе
| | |Находим минимальное из заданных чисел
| | если a > b
| | |то
| | | мин = b
| | |иначе
| | | мин = a
| | все
| | |и проверяем, не является ли оно
| | |и меньшие него числа
| | |общим делителем заданных чисел
| | НОД = мин
| | |проверку прекращаем, как только
| | |будет найдено число, являющееся НОД
| | нц пока mod(a, НОД) <> 0 или mod(b, НОД) <> 0
| | | НОД = НОД - 1
| | кц
| все
| |Выводим результат
| вывод нс, "Наибольший общий делитель этих чисел равен", NOD
кон
```

Эту программу можно немного сократить, если не сравнивать заданные числа между собой, а сразу проверять одно из этих чисел (например, первое) и меньшие, чем оно:

```
алг Определение_НОД
нач цел a, b, НОД
| |Ввод чисел
| ...
| |Проверяем, не является ли число a
| |и меньшие него числа
| |общим делителем заданных чисел
| НОД = a
| нц пока mod(a, НОД) <> 0 или mod(b, НОД) <> 0
| | НОД = НОД - 1
| кц
| |Выводим результат
| ...
кон
```

Выигрыш в размере программы существенный! «Понятность» ее хуже, пожалуй, не стала — благодаря записи комментария[2] относительно проверки числа a и меньших него чисел. Но если число a будет больше числа b, то количество проверяемых чисел по сравнению с первым вариантом возрастет, причем чем больше отношение a/b, тем значительней растет количество проверяемых чисел!

Какой из двух приведенных выше вариантов программы лучше, определять вам, уважаемый читатель.

По сути дела, оба рассмотренных варианта работают по методу перебора. Вместе с тем, существует алгоритм нахождения НОД двух чисел, не использующий перебор. Он носит имя древнегреческого математика Евклида, который изложил его в своей знаменитой работе «Начала» (330–320 гг. до н. э.). Суть этого, считающегося самым древним, алгоритма состоит в следующем. Если число a больше числа b, то нужно из a вычесть b; в противном случае, наоборот, нужно из b вычесть a и повторять эти действия до тех пор, пока a не станет равно b. После этого искомый НОД будет равен одному из полученных чисел. Приведем программу, работающую по этому алгоритму:

```
алг Определение_НОД
нач цел a, b, НОД
| |Ввод чисел
| ...
| нц пока a <> b
| | если a > b
| | |то
| | |  a = a - b
| | |иначе
| | |  b = b - a
| | все
| кц
| НОД = a
| |Вывод результата
| ...
кон
```

[2] *Комментарии* — оформляемые специальным образом фрагменты текста программы, играющие исключительно информационную роль. *Транслятор* (системная программа, переводящая прикладную программу, написанную на языке программирования высокого уровня — на Бейсике, Си, Паскале или др., в машинный код) в процессе работы игнорирует комментарии. Наличие комментариев считается «хорошим тоном» в программировании, так как позволяет сделать листинг понятнее и снабдить его подробными объяснениями (в том числе о назначении переменных, смысле выполняемых операций и т. д.).

Как улучшить эту программу? Прежде всего, можно многократные вычитания заменить на определение остатка от деления одного целого числа на другое:

```
нц
| если a > b
| |то
| |  a = mod(a, b)
| |иначе
| |  b = mod(b, a)
| все
кц при a = 0 или b = 0
NOD = a + b
```

Обратим внимание на условие окончания оператора цикла и на способ определения НОД. Очевидно дальнейшее усовершенствование: можно отказаться от многократного сравнения величин `a` и `b` — достаточно сделать это один раз, а потом учесть, что получаемый в цикле остаток всегда будет меньше, чем второй параметр функции `mod`.

В приведенной ниже программе, учитывающей это обстоятельство, использованы следующие новые величины: `макс` — максимальное из двух заданных чисел, `мин` — минимальное из двух заданных чисел, `остаток` — остаток от деления `макс` на `мин`.

```
алг Определение НОД
нач цел a, b, макс, мин, остаток, НОД
| |Ввод чисел
|  ...
| если a > b
| |то
| |  макс = a
| |  мин = b
| |иначе
| |  макс = b
| |  мин = a
| все
| остаток = mod(макс, мин)
| нц пока остаток <> 0
| | макс = мин
| | мин = остаток
| | остаток = mod(макс, мин)
| кц
| НОД = мин
| вывод нс, "Наибольший общий делитель этих чисел равен", НОД
кон
```

Насколько понятной получилась эта программа — решите сами.

И наконец, самая короткая возможная программа нахождения НОД:

```
алг Определение_НОД
нач цел a, b, остаток, НОД
| |Ввод чисел
| ...
| нц пока остаток <> 0
| | остаток = mod(a, b)
| | a = b
| | b = остаток
| кц
| НОД = a
| вывод нс, "Наибольший общий делитель этих чисел равен", НОД
кон
```

В особенностях ее работы разберитесь самостоятельно.

Итак, мы убедились, что часто одна и та же задача может быть решена различными способами, каждый из которых имеет свои преимущества и недостатки. Какой же путь разработки программ выбрать? По мнению автора, начинающему программисту лучше руководствоваться принципом «KIS» (четвертая буква здесь пропущена сознательно ☺), а после приобретения достаточного опыта программирования надо всегда стремиться к усовершенствованию программы с точки зрения ее размера, быстродействия и т. д. Ведь, как писал выдающийся математик XIX века К. Гаусс, «достоинство науки требует, чтобы прилежно совершенствовались средства, ведущие к достижению цели...».

2. Обмен значениями между двумя переменными

Рассмотрим такую задачу: *Даны значения двух переменных* a *и* b. *Требуется произвести взаимный обмен их значений.*

«Очевидное» решение этой задачи, что называется, «в лоб»:

```
a := b      или      b := a
b := a               a := b
```

требуемого результата не даст (кстати — подумайте сами почему?). Как же быть? Надо действовать так, как производят обмен содержимого двух чашек, в одной из которых находится молоко, а в другой — чай. Нужна третья чашка![3] То есть в нашей задаче для ее решения потребуется третья (вспомогательная) переменная. С ее помощью обмен значениями может быть проведен следующим образом:

```
c := a |Запоминаем значение a
a := b |Переменной a присваиваем значение b
b := c |Переменной b присваиваем "старое" значение a
```

или

```
c := b
b := a
a := c
```

Но оказывается, что обменять значения двух переменных можно и без использования вспомогательной переменной. «Не может быть!» — скажете вы. Еще как может, — и сделать это можно даже несколькими способами. Вот один из них:

```
a := a + b
b := a - b
a := a - b
```

Красиво, не правда ли? С молоком так не сделаешь! ☺ Другие подобные способы решения (а их, по крайней мере, еще 7) найдите самостоятельно. А заодно решите следующую задачу: *Даны значения трех переменных* a, b *и* c. *Требуется составить программу, после выполнения которой переменная* b *будет иметь значение переменной* a, *переменная* c *— значение переменной* b, *а переменная* a *— значение переменной* c. *Дополнительные величины не применять.* Интересно, сколько способов ее решения вы найдете?

[3] Можно, конечно, использовать и две дополнительные чашки, но это уже перебор!

3. Об операторах цикла

Как вы, очевидно, знаете, операторами цикла называют операторы, обеспечивающие повторение одних и тех же действий (поэтому их иногда также называют операторами повторения). В большинстве современных языков программирования существует несколько разновидностей операторов цикла. Опишем некоторые их особенности.

3.1. Оператор цикла с параметром (со счетчиком)

Применяется в тех случаях, когда в программе какие-либо действия (операторы) повторяются заранее известное количество раз и при этом некоторая величина меняется с постоянным шагом.

Пример. Для вывода «столбиком» всех целых чисел от 10 до 20 без использования оператора цикла потребовалось бы записать:

```
вывод нс, 10
вывод нс, 11
...
вывод нс, 20
```

Хорошо видно, что здесь есть повторяющиеся действия (вывод числа на экран) и что при этом выводимое число меняется с шагом, равным 1. А значит, можно применить оператор цикла с параметром.

Общий вид этого оператора[4]:

```
нц для <параметр цикла> от <начальное значение>
                        до <конечное значение>
| <тело оператора цикла>
кц
```

где <параметр цикла> — имя переменной, являющейся параметром цикла (меняющейся при повторении действий), <начальное значение> — начальное значение этой переменной, <конечное значение> — конечное значение этой переменной (предполагается, что <конечное значение> больше, чем <начальное значение>, и

[4] С правилами оформления этого и других операторов цикла в конкретном языке программирования, который вы изучаете, ознакомьтесь самостоятельно.

что изменение от <начального значения> до <конечного значения> происходит с шагом, равным 1), а <тело оператора цикла> — операторы, повторяющиеся при работе оператора цикла.

В рассмотренном примере оператор оформляется следующим образом:

```
нц для i от 10 до 20 |i — имя величины-параметра цикла
| вывод нс, i
кц
```

Схема, иллюстрирующая работу оператора цикла с параметром, показана на рис. 3.1.

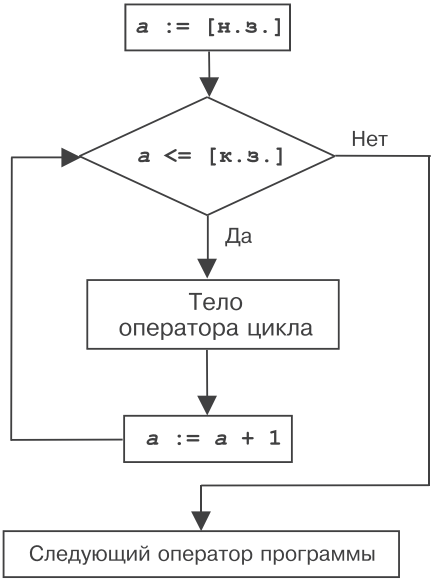


Рис. 3.1

На этой схеме буквой a обозначено имя переменной — параметра цикла, [н.з.] и [к.з.] — соответственно начальное и конечное значения параметра.

Возникает вопрос: какое значение имеет параметр цикла после окончания работы оператора? Ответы здесь различны для разных систем программирования: в программе на Бейсике оно равно (a + 1), в программе на Паскале — a, а в программе на школьном алгоритмическом языке оно вообще считается неопределенным (не кажется ли вам, уважаемый читатель, что в этом есть своя логика — ведь параметр цикла нужен только для работы оператора цикла, когда выполняются действия при различных его значениях?).

В некоторых языках программирования предусмотрена возможность изменения значения переменной — параметра цикла с шагом, отличным от 1. Например, в языке Паскаль[5] шаг может быть равен −1, а в языке Бейсик — вообще любому числу, в том числе вещественному. В таких случаях при оформлении оператора указывается значение шага, а приведенная на рис. 3.1 схема немного меняется (вместо `a:=a+1` в ней будет записано: `a:=a+[шаг]`).

1. Нарисуйте схему работы оператора цикла с параметром, значение которого меняется:
   - от большего начального значения до меньшего конечного с шагом, равным −1;
   - от меньшего начального значения до большего конечного с шагом, равным 2.
2. Может ли тело оператора цикла с параметром не выполниться ни разу?
3. Сколько раз будет выполняться тело оператора цикла с параметром при начальном (`a`) и конечном (`b`) значениях параметра цикла, равных:
   - `a` = 1, `b` = 10;
   - `a` = 10, `b` = 20;
   - `a` = `N`, `b` = `M`,

   если во всех случаях шаг изменения параметра цикла равен 1?
4. Сколько раз будет выполняться тело оператора цикла с параметром, если начальное значение параметра `a` = `N`, конечное значение `b` = `M`, а шаг изменения равен `s` (`s` ≠ 1)? (Предполагается, что значение `s` ≠ 1 допускается в данном языке программирования.)
5. Самостоятельно установите, можно ли в языке программирования, который вы используете, изменять значение параметра в теле оператора цикла, а также определите значение параметра после завершения работы оператора цикла. (Это очень важные вопросы, ответы на которые помогут вам избежать ошибок при разработке программ. Заметим, однако, что изменение параметра, а также его начального и конечного значений в теле оператора цикла может привести к непредсказуемым последствиям и считается плохим стилем программирования.)

[5] В языке программирования Паскаль при шаге, равном −1, в операторе вместо служебного слова `To` должно быть записано `Downto`.

Частным случаем оператора цикла с параметром является вариант, при котором само значение параметра цикла в его теле не используется. Например, если требуется повторить какие-то действия 10 раз, то соответствующий фрагмент может быть записан (с учетом схемы на рис. 3.1) как:

```
нц для i от 1 до 10
| ...
кц
```

или

```
нц для i от 2 до 11
| ...
кц
```

или другим подобным способом. Примером является задача обработки последовательности чисел (см. раздел 4). В ней для ввода чисел требуемой последовательности может быть использован следующий фрагмент программы:

```
нц для i от 1 до n |n — общее количество чисел в последовательности
| вывод нс, "Задайте очередное число последовательности"
| ввод a    |Очередное число
| ...       |Обработка заданного значения числа a
кц
```

Здесь видно, что переменная i в теле оператора цикла вообще никак не используется — она лишь играет роль счетчика количества повторений (вспомните второе название цикла с параметром).

Измените приведенный выше фрагмент так, чтобы на экране при вводе очередных значений последовательно появлялись следующие сообщения (при n = 20):

```
Задайте 1-е число последовательности
Задайте 2-е число последовательности
...
Задайте 20-е число последовательности
```

Оператор цикла с параметром также можно использовать в программе для реализации паузы, если в его теле не записывать никаких операторов (так называемый «*пустой цикл*»). Например:

```
нц для i от 1 до 2000
|
кц
```

Продолжительность такой паузы будет зависеть от быстродействия компьютера и от конечного значения параметра цикла.

Подберите на своем компьютере такое конечное значение параметра «пустого цикла», при котором будет получаться пауза длительностью примерно в 5 секунд.

3.2. Оператор цикла с предусловием

Этот оператор используется, когда требуемое количество повторений заранее не известно. Его общий вид:

```
нц пока <условие>
| <тело оператора цикла>
кц
```

где <условие> — условие, при котором выполняется <тело оператора цикла>.

Схема, иллюстрирующая работу этого оператора, показана на рис. 3.2.

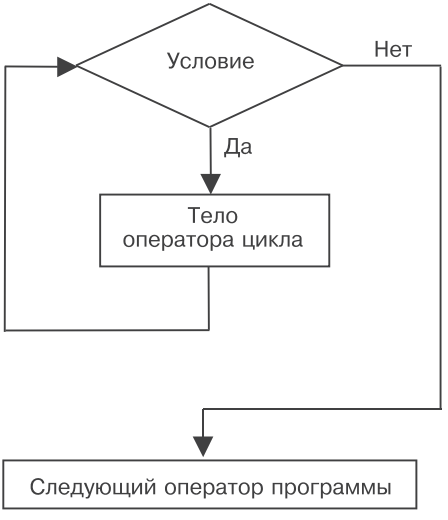


Рис. 3.2

Примечание. В языке программирования QuickBasic также имеется оператор цикла с предусловием, выполняемый, пока заданное ложное условие не станет истинным, — **DO UNTIL** <условие> **... LOOP**.

Существует полезное правило, с помощью которого легко можно сформулировать условие, записываемое в таком операторе цикла:

1) определите условие, при котором *нельзя* или *не нужно* выполнять повторяющиеся действия;
2) запишите в операторе цикла условие, противоположное только что найденному.

Рассмотрим, например, такую задачу: *Дано натуральное число* n. *Требуется найти его сумму цифр.*

Будем рассуждать следующим образом. Последняя цифра любого натурального числа равна остатку от деления этого числа на 10 (самостоятельно убедитесь в этом). Значит, для решения задачи нужно многократно выполнять следующие действия:

1) определить последнюю цифру числа n;
2) добавить ее к рассчитанной ранее сумме цифр;
3) изменить число n, отбросив в нем уже обработанную последнюю цифру.

Например, при n = 30 625 необходимо эти действия повторить 5 раз:

Шаг	Последняя цифра	Сумма обработанных цифр	Новое значение n
1	5	5	3062
2	2	5 + 2 = 7	306
3	6	7 + 6 = 13	30
4	0	13 + 0 = 13	3
5	3	13 + 3 = 16	0

Тогда фрагмент программы для подсчета суммы цифр можно оформить с использованием оператора цикла с параметром:

```
сумма_цифр := 0
нц для i от 1 до 5
| последняя_цифра := mod(n, 10)
| сумма_цифр := сумма_цифр + последняя_цифра
| n := div(n, 10)
кц
```

где div — функция, возвращающая результат целочисленного деления ее первого аргумента на второй[6].

[6] В других языках программирования для определения остатка обычно используется не функция, а специальная операция. В языке Паскаль знак этой операции — div, в языке Бейсик — обратная косая черта («\»).

Однако по условию задачи количество разрядов в заданном числе n неизвестно, а значит, неизвестно и требуемое количество повторений. Поэтому оператор цикла с параметром здесь использовать нельзя. Попробуем применить оператор цикла с предусловием:

```
нц пока ?
| последняя_цифра := mod(n, 10)
| ...
кц
```

Чтобы определить условие, обозначенное в данном листинге символом «?», вспомним приведенное ранее правило. Мы должны прекратить вычисления, когда n = 0 (см. таблицу для числа 30 625). Значит, условие[7], записываемое после служебного слова пока, должно быть таким: n > 0 (отрицательным n быть не может!). Следовательно, можно оформить наш фрагмент программы так:

```
сумма_цифр := 0
нц пока n > 0
| последняя_цифра := mod(n, 10)
| сумма_цифр := сумма_цифр + последняя_цифра
| n := div(n, 10)
кц
```

1. Может ли тело оператора цикла с предусловием не выполниться ни разу?
2. Может ли тело оператора цикла с предусловием выполняться бесконечно (или до того момента, когда пользователь вручную прервет выполнение программы одновременным нажатием клавиш Ctrl + Break)?

Оператор цикла с предусловием также может быть использован в программе для получения паузы. (Составьте соответствующий вариант самостоятельно.)

Обратим внимание на одну особенность работы оператора цикла с предусловием. Как следует из рис. 3.2, условие проверяется только *перед выполнением* тела цикла, но не проверяется в процессе его выполнения. Неучет этого обстоятельства может привести к ошибкам.

Это очень наглядно продемонстрировано в учебнике [1] на примере задачи с исполнителем «Робот». Пусть Робот, находящийся на клетчатом поле и умеющий перемещаться на одну клетку влево, вправо,

---

[7] Конечно, в данном случае это условие можно было определить и не используя указанное правило. Однако часто искомое условие далеко не так очевидно, и это правило способно оказать вам существенную услугу.

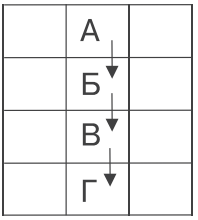


Рис. 3.3

вверх или вниз, находится в клетке *А* (рис. 3.3). Тогда, согласно схеме на рис. 3.2, при выполнении фрагмента программы:

```
нц пока снизу свободно
| вниз
| вниз
кц
```

результат ее работы будет следующим.

1. Исходное положение — Робот находится в клетке *А*. Проверяется условие: `снизу свободно` (оно истинно) и выполняется тело оператора. Тогда Робот, дважды сместившись вниз, окажется в клетке *В*.
2. Опять проверяется условие `снизу свободно`; оно также истинно, так что начнется выполнение тела оператора. При этом Робот сначала сместится в клетку *Г*, но затем вторая команда (оператор) `вниз` выполниться не сможет — появится сообщение об ошибке.

Конечно, Робот — это «условный» исполнитель. Однако нетрудно привести и пример «реальной» задачи на языке программирования. Пусть нам требуется вывести на экран числа в следующем порядке:

```
21  20
19  18
17  16
...
11
```

Казалось бы, достаточно записать в программе:

```
a := 21
нц пока a >= 11
| вывод нс, a, "  ", a - 1
| a := a - 2
кц
```

Однако в результате выполнения этого фрагмента на экран будет выведено следующее (также согласно схеме на рис. 3.2):

```
21  20
19  18
17  16
...
11  10
```

Но это не соответствует заданному условию (число 10 нам выводить было не нужно).

Еще более наглядный пример приведен в [2]. В этой статье представлена программа, моделирующая игру двух участников, каждый из которых берет некоторое количество спичек из одной из двух кучек. Соответственно, в такой программе должен быть фрагмент, блок-схема которого показана на рис. 3.4.

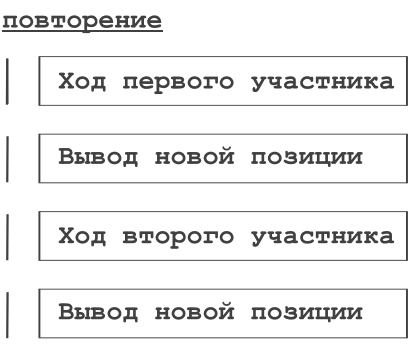


Рис. 3.4

Казалось бы, все правильно: участники по очереди делают ходы, пока игра не закончится (когда в обеих кучках не будет спичек). Однако при таком оформлении программы если после хода первого участника игра закончится, то блок `Ход второго участника` все равно будет выполняться (и на экран будут выводиться вопросы, связанные с ходом второго игрока), хотя условие окончания работы оператора уже соблюдается!

Аналогичные ситуации могут иметь место в других компьютерных программах, моделирующих игры двух и более участников, и в ряде других случаев. Помните об этом!

Подумайте, как изменить приведенную выше блок-схему игры в спички, чтобы в описанной ситуации операторы блока `Ход второго участника` не выполнялись.

3.3. Оператор цикла с постусловием

Этот вид циклов также применяется, когда количество повторений заранее неизвестно, но здесь проверка условия производится уже после выполнения тела цикла.

Общий вид этого оператора:

```
нц
| <тело оператора цикла>
кц при <условие>
```

где <условие> — условие, при котором оператор прекращает свою работу.

Например, в задаче о сумме цифр (см. выше) такой оператор можно оформить следующим образом:

```
сумма_цифр := 0
нц
| последняя_цифра := mod(n, 10)
| сумма_цифр := сумма_цифр + последняя_цифра
| n := div(n, 10)
кц при n = 0
```

Схема, иллюстрирующая работу оператора цикла с постусловием, показана на рис. 3.5.

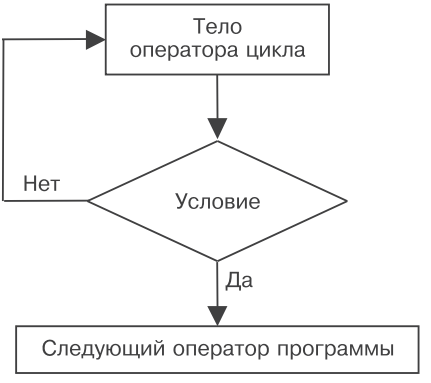


Рис. 3.5

Примечание. В языке программирования QuickBasic также имеется оператор цикла с постусловием, выполняемый, пока истинное условие не станет ложным — **DO ... LOOP WHILE** <условие>.

1. Может ли тело оператора цикла с постусловием выполняться бесконечно (или до того момента, когда пользователь вручную прервет выполнение программы одновременным нажатием клавиш Ctrl + Break)?
2. Может ли тело оператора цикла с предусловием не выполниться ни разу?

Оператор цикла с постусловием, в частности, можно использовать для проверки правильности вводимых в программу значений. Действия при этом обязательно выполняются хотя бы один раз и в случае ввода ошибочного значения повторяются. Например, следующий вариант обеспечивает проверку правильности введенного значения коэффициента a в программе нахождения корней квадратного уравнения вида $ax^2 + bx + c = 0$ $(a \neq 0)$:

```
нц
| вывод нс, "Задайте значение коэффициента a в уравнении"
| ввод a
| если a = 0
| | то
| |  вывод нс, "Коэффициент a не должен быть равен нулю!"
| все
кц при a <> 0
```

1. Напишите фрагмент программы, в которой пользователь в ответ на вопрос: «Чет (2) или нечет (1)?» должен ввести одно из двух значений — «1» или «2». В случае же ввода другого числа на экран должно выводиться сообщение об ошибке, после чего эти действия должны повторяться до ввода правильного значения.
2. Напишите фрагмент программы, в которой пользователь должен ввести установленный пароль. В случае ввода неправильного пароля на экран должно выводиться сообщение об ошибке, после чего действия должны повторяться до ввода правильного пароля.

Ясно, что рассматриваемая разновидность оператора цикла тоже может быть применена в программе для получения паузы в ее работе. Здесь же заметим, что в большинстве современных языков программирования оператор цикла с постусловием используется для приостановки программы до нажатия любой клавиши. Например, в языке Паскаль это делается так:

```
Repeat
Until Keypressed;
```

где Keypressed — функция, которая возвращает значение логического типа, указывающее, была ли нажата какая-либо клавиша. В языке Бейсик (вариант Quick Basic) та же программа будет выглядеть следующим образом:

```
DO
LOOP UNTIL INKEY$ <> " "
```

где INKEY$ — функция, возвращающая символ, считанный с клавиатуры. Если никакой символ не был считан (нажатия клавиш не

было), то возвращается «пустой» символ (""). Заметим, что функция INKEY$ не дублирует ввод с клавиатуры выводом символа на экран (т. е. нажатый символ на экран не выводится).

Часто в программе необходимо заменить одну разновидность оператора цикла на другую. Так как, зная условие продолжения работы оператора цикла с предусловием, можно определить условие окончания работы оператора цикла с постусловием, и наоборот (каждое из этих условий противоположно другому), то одну из этих разновидностей оператора всегда можно заменить на другую.

Более сложным является вопрос о возможности замены оператора цикла с параметром. Если проанализировать схему на рис. 3.1, то нетрудно увидеть, что для него:

1) известно значение величины a до начала выполнения тела оператора;
2) условие окончания работы оператора: a > [конечное значение].

Это означает, что оператор цикла с параметром всегда можно заменить на оператор цикла с постусловием (а следовательно, — и на оператор цикла с предусловием). Например, приведенный ранее фрагмент:

```
нц для i от 10 до 20
| вывод нс, i
кц
```

можно оформить в виде оператора цикла с постусловием:

```
i := 10
нц
| вывод нс, i
| i := i + 1
кц при i > 20
```

или с предусловием:

```
i := 10
нц пока i <= 20
| вывод нс, i
| i := i + 1
кц
```

А наоборот? Всегда ли можно заменить цикл с пред- или с постусловием на цикл с параметром? Вспомним задачу о сумме цифр некоторого натурального числа:

```
сумма_цифр:=0
нц
| последняя_цифра := mod(n, 10)
| сумма_цифр := сумма_цифр + последняя_цифра
| n := div(n, 10)
кц при n = 0
```

Можем ли мы определить начальное и конечное значения величины n и шаг ее изменения и тем самым применить оператор цикла с параметром? В данном случае — нет! Но иногда это возможно (см., например, приведенные выше операторы цикла с постусловием и с предусловием).

Сделаем вывод: возможность «взаимозаменяемости» циклов разного типа можно проиллюстрировать схемой, показанной на рис. 3.6. При этом иногда для реализации повторяющихся действий может быть использована любая из трех разновидностей оператора цикла, а в некоторых случаях возможно использование только операторов цикла с предусловием и с постусловием.

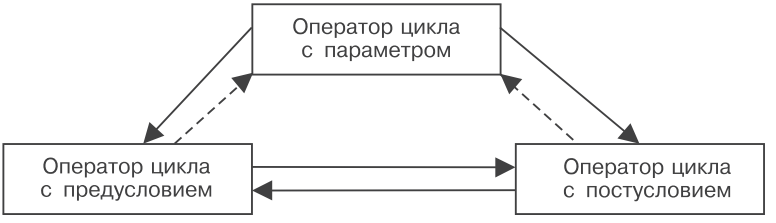


Рис. 3.6

А вот еще одна задача, связанная с заменой одной разновидности оператора цикла на другую: *Дан массив целых чисел. Требуется определить, имеется ли в массиве число 13.*

Можно, скажем, подсчитать количество чисел в массиве, равных 13, и по этому значению получить ответ на вопрос в условии задачи:

```
k := 0
нц для i от 1 до n
| если m[i] = 13
| | то
| |  k := k + 1
| все
кц
если k > 0
| то
|  вывод нс, "Число 13 в массиве есть"
| иначе
|  вывод нс, "Числа 13 в массиве нет"
все
```

Однако такой способ нерационален — в нем происходит проверка *всех* элементов массива, в то время как число 13 может оказаться записанным уже, например, во втором его элементе. Тогда желательно прекратить проверку сразу, как только встретится это число. Но поскольку количество проверок до этого момента заранее неизвестно, то применим оператор цикла с предусловием.

Условие, записываемое в нем, определим, используя то самое правило, корторое было описано при рассмотрении данного оператора. Будем рассуждать так: проверку значений элементов массива надо прекратить, как только встретится число 13 или когда массив будет исчерпан; соответственно, это условие можно записать в виде:

```
есть13 или i > n
```

где есть13 — переменная логического типа, принимающая истинное значение, если число 13 встретится в массиве, а i — индекс очередного проверяемого элемента массива. Тогда, согласно правилу, условие в операторе цикла с предусловием должно быть противоположным:

```
не есть13 и i <= n
```

а весь фрагмент программы для решения данной задачи оформляется следующим образом[8]:

```
i := 1; есть13 := нет
нц пока не есть13 и i <= n
| если m[i] = 13
| | то
| |  есть13 := да
| | иначе |Переходим к рассмотрению следующего элемента
| |  i := i + 1
| все
кц
если есть13
| то
|  вывод нс, "Число 13 в массиве есть"
| иначе
|  вывод нс, "Числа 13 в массиве нет"
все
```

Убедитесь, что применительно к массиву [21 6 13 7 77 12 0 120 12 15] работа оператора прекратится после проверки третьего элемента массива, а для массива [21 6 15 7 77 12 0 120 12 15], в котором число 13 отсутствует, результат работы программы также будет правильным: ее выполнение завершится после проверки всех элементов массива.

В заключение рассмотрим еще одну очень показательную задачу. Пусть требуется составить программу для вывода на экран чисел 0.1, 0.2, ..., 1.0. На языке Бейсик такая программа имеет вид:

```
FOR i = 0.1 TO 1 STEP 0.1
PRINT i
NEXT i
```

[8] Обратите внимание на условие, используемое в операторе если.

или с использованием оператора цикла с предусловием:

```
i = 0.1
WHILE i <= 1
PRINT i
i = i + 0.1
WEND
```

Однако в результате выполнения любой из этих программ мы получим последовательность чисел: `0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8000001 0.9000001`, т. е. требуемая единица выведена не будет!

Может быть, она будет выведена в результате работы программы на языке Паскаль? Так как в этом языке параметр цикла не может быть вещественным числом, используем для решения нашей задачи оператор цикла с условием (хотя суть от этого не меняется):

```
Var i: real;
BEGIN
i := 0.1;
While i <= 1 Do Begin
                  Write(i:4:1);
                  i: = i + 0.1
                End
END.
```

Здесь результат получится следующий:

```
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

Как видим, здесь нужной нам единицы тоже нет!

Причиной получаемых неправильных результатов является тот факт, что в памяти компьютера вещественные числа (а именно такой тип имеет в программах величина `i`), не всегда представлены точно. Так, используемое в обеих программах десятичное число 0.1 при переводе в двоичную систему счисления имеет вид периодической дроби, а поскольку количество разрядов (битов) в памяти ограничено, то число 0.1 будет представлено в ней приближенно. Поэтому сумма нескольких чисел 0.1 в памяти компьютера будет незначительно превышать 0.9, и после очередного сложения с 0.1 превысит 1. Эту особенность работы с вещественными числами следует учитывать при создании программ.

Напишите программу для решения рассматриваемой задачи, в которой будет выведено искомое число 1 (оформите оператор цикла, в условии которого используются переменные целого типа)[9].

[9] Большое число подобных задач на использование операторов цикла приведено в [3].

4. Типовые задачи на обработку последовательности чисел

В этом разделе рассмотрены методы решения восьми типовых задач, которые чаще всего встречаются при обработке последовательностей чисел (нахождение их суммы, максимального значения и др.). Для каждой задачи приведен фрагмент программы ее решения на школьном алгоритмическом языке. При этом использованы следующие основные величины: n — общее количество чисел в последовательности (оно может быть заранее задано в программе или вводится в ходе ее выполнения); a — очередное обрабатываемое число последовательности. Смысл остальных величин можно легко определить по их именам.

Так как количество обрабатываемых чисел n известно заранее, в программах использован оператор цикла с параметром.

4.1. Суммирование всех чисел последовательности

```
сумма := 0
нц для i от 1 до n
| |Ввод очередного числа a
| ...
| сумма := сумма + a
кц
|Вывод результата или его использование в расчетах
...
```

Пример. Известен вес каждого предмета, загружаемого в автомобиль. Определите общий вес груза.

Решение

```
вывод нс, "Укажите количество предметов"
ввод n
сумма := 0
нц для i от 1 до n
| вывод нс, "Введите вес", i, "-го предмета"
| ввод a
| сумма := сумма + a
кц
вывод нс, "Общий вес всех предметов равен", сумма
```

Примечание. В языках программирования Бейсик, Паскаль, Си и ряде других инициализирующий оператор `сумма := 0` не является обязательным (так как все используемые в программе переменные изначально обнуляются автоматически). Вместе с тем, начальное присваивание величине `сумма` нулевого значения является «правилом хорошего тона» (а в некоторых задачах это является обязательным действием).

4.2. Суммирование чисел последовательности, удовлетворяющих некоторому условию

```
сумма := 0
нц для i от 1 до n
| |Ввод очередного числа a
| ...
| если <условие>
| | то
| |  сумма := сумма + a
| все
кц
| Вывод результата или его использование в расчетах
...
```

Примечание. Здесь условие в условном операторе (команде `если`) может определяться значением числа `a` или его порядковым номером `i`.

4.3. Подсчет количества чисел последовательности, удовлетворяющих некоторому условию

```
количество := 0
нц для i от 1 до n
| |Ввод очередного числа a
| ...
| если <условие>
| | то
| |  количество := количество + 1
| все
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. В данном случае условие в условном операторе (команде `если`) определяется значением числа `a`. Если это условие зависит от порядкового номера `i`, то задача может быть решена без использования оператора циклов (самостоятельно убедитесь в этом).

4.4. Определение среднего арифметического чисел последовательности, удовлетворяющих некоторому условию

```
сумма := 0
количество := 0
нц для i от 1 до n
| |Ввод очередного числа a
| ...
| если <условие>
| | то
| |  сумма := сумма + a
| |  количество := количество + 1
| все
кц
|Подсчет и вывод результата
среднее_арифметическое := сумма / количество
вывод нс, "Среднее арифметическое: ", среднее_арифметическое
```

Обратим внимание на то, что многократно определять значение среднее_арифметическое в теле условного оператора нет необходимости. Это можно сделать один раз после окончания оператора цикла. Однако может оказаться, что чисел, удовлетворяющих заданному условию, в последовательности не окажется вовсе, и тогда произойдет деление на нуль, что недопустимо. Поэтому более правильным будет следующий фрагмент:

```
...
|Подсчет и вывод результата
если количество > 0
| то
|  среднее_арифметическое := сумма / количество
|  вывод нс, "Среднее арифметическое: ", среднее_арифметическое
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, нет"
все
```

4.5. Определение порядкового номера некоторого значения в заданной последовательности

Здесь значение — это число, номер которого (номер_значения) нужно найти.

```
нц для i от 1 до n
| |Ввод очередного числа a
| ...
| если a = значение
| | то
| |  номер_значения := i
| все
кц
```

```
|Вывод результата
если номер_значения <> 0
| то
|  вывод нс, "Номер этого значения: ", номер_значения
| иначе
|  вывод нс, "Такого числа нет"
все
```

Возникает вопрос: какой из порядковых номеров будет найден по приведенному алгоритму, если в последовательности окажется несколько чисел с искомым значением?

Примечание. Несколько изменив данный фрагмент, можно определять, имеется ли в заданной последовательности некоторое число.

4.6. Определение максимального значения в последовательности чисел

Алгоритм решения этой задачи аналогичен действиям человека, который определяет максимальное значение в некоторой последовательности, например 4, 2, 9, 6, 3, 16, 10, 2, 7. Сначала он запоминает первое число. Затем рассматривает второе число: если оно больше того числа, которое он запомнил, то человек запоминает уже новое максимальное число и переходит к следующему; в противном случае — просто переходит к следующему (третьему) числу и делает то же самое, пока массив не будет просмотрен весь.

Так и в нашей программе: сначала следует ввести и принять первое число в качестве максимального (его имя в программе — максимальное), затем ввести остальные числа (a) и каждое из них сравнивать со значением максимальное: если a > максимальное, то в качестве нового значения максимальное принимается значение числа a:

```
|Ввод первого числа a
...
максимальное := a
нц для i от 2 до n
| |Ввод остальных чисел последовательности a
| ...
| если a > максимальное
| | то
| |  максимальное := a
| все
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. Аналогичным способом, при незначительном изменении приведенного фрагмента, ищется минимальное число в последовательности.

Если минимальное значение в обрабатываемой последовательности заранее известно, то соответствующий фрагмент программы можно оформить короче:

```
максимальное := минимальное
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
| если a > максимальное
| |то
| | максимальное := a
| все
кц
```

где минимальное — минимальное число последовательности.

Например, если известно, что в обрабатываемой последовательности нет отрицательных значений, то можно записать:

```
максимальное := 0
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
| если a > максимальное
| | то
| |  максимальное := a
| все
кц
|Вывод результата или его использование в расчетах
...
```

4.7. Определение порядкового номера максимального значения в последовательности чисел

Здесь алгоритм решения задачи аналогичен приведенному выше, но кроме очередного «претендента на звание» максимального числа следует запоминать также и его порядковый номер.

Соответственно, в программе сначала надо ввести первое число и принять его в качестве максимального (максимальное), а искомый порядковый номер (номер_максимального) принять равным 1. Затем вводятся остальные числа a и каждое из них сравнивается со значением максимальное. Если a > максимальное, то в качестве нового значения максимальное принимается значение числа a, а в качестве нового значения номер_максимального — номер этого числа i.

```
|Ввод первого числа последовательности a
...
максимальное := a
номер_максимального := 1
нц для i от 2 до n
| |Ввод остальных чисел последовательности a
| ...
```

```
| если a > максимальное
| | то
| |  максимальное := a
| |  номер_максимального := i
| все
кц
|Вывод результата или его использование в расчетах
...
```

Если минимальное значение в обрабатываемой последовательности заранее известно, то, как и в предыдущей задаче, соответствующий фрагмент может быть оформлен короче:

```
максимальное := минимальное
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
| если a > максимальное
| |то
| | максимальное := a
| | номер_максимального := i
| все
кц
```

где минимальное — это минимальное число в последовательности.

4.8. Определение максимального значения чисел последовательности, удовлетворяющих некоторому условию

При этом возможны два случая:

1) точно известно, что числа, удовлетворяющие заданному условию, в обрабатываемой последовательности имеются;
2) чисел, удовлетворяющих заданному условию, в обрабатываемой последовательности может и не быть.

В первом случае задача решается следующим образом:

```
максимальное := X
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
| если <условие>
| | то |Встретилось число, удовлетворяющее заданному условию
| | |  |сравниваем его с величиной максимальное
| | | если a > максимальное
| | | |то
| | | | максимальное := a
| | | все
| | все
| все
кц
```

где X — число, о котором заранее известно, что оно не превышает минимальное из исходных чисел, для которых надо определить максимум.

Если же такое значение заранее неизвестно[10], то задача несколько усложняется. В приведенном фрагменте используется величина логического типа впервые[11], определяющая, впервые ли встретилось в последовательности число, удовлетворяющее заданному условию. Остальные особенности работы программы описаны в ее комментариях.

```
впервые := да
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
| если <условие>
| | то |Встретилось число, удовлетворяющее заданному условию
| |  если впервые        |Если оно встретилось впервые,
| |  | то
| |  |  максимальное:= a |принимаем его в качестве
| |  |                   |значения максимальное
| |  |  впервые := нет   |Следующие такие числа уже будут
| |  |                   |встречаться не впервые
| |  | иначе    |Если оно встретилось не впервые,
| |  |          |сравниваем число с величиной максимальное
| |  |  если a > максимальное
| |  |  |то
| |  |  | максимальное := a
| |  |  все
| |  все
| все
кц
```

Если же допускается, что чисел, удовлетворяющих заданному условию, в обрабатываемой последовательности может не быть, то для решения задачи можно использовать последний из приведенных фрагментов программы:

```
впервые := да
нц для i от 1 до n
| |Ввод очередного числа последовательности a
| ...
кц
```

Однако вывод ответа при этом должен быть уточнен:

```
если впервые = да
| то
|  вывод нс, "Чисел, удовлетворяющих условию, "
|  вывод "в последовательности нет"
| иначе
|  вывод нс, "Искомое максимальное значение: ", максимальное
все
```

---

[10] Вопрос о том, всегда ли можно найти такое значение, исследуйте самостоятельно.

[11] В программах на языке Бейсик и Си вместо величины логического типа следует использовать величину целого типа, например, принимающую значения 0 и 1.

1. Известно сопротивление каждого из шести элементов электрической цепи. Все элементы соединены последовательно. Определите общее сопротивление цепи.
2. Известна стоимость каждого товара из некоторой группы. Найдите общую стоимость товаров, которые стоят дороже 1000 рублей (количество таких товаров заранее неизвестно).
3. Известны данные о количестве осадков, выпавших в каждый день некоторого месяца. Определите общее количество осадков, выпавших второго, четвертого и т. д. числа этого месяца. (Оператор цикла с параметром, шаг изменения которого отличается от 1 или −1, не использовать!)
4. Известны оценки по информатике каждого ученика класса. Определите количество «пятерок».
5. Известен рост каждого ученика в классе. Рост мальчиков условно задан отрицательными числами. Определите средний рост мальчиков и средний рост девочек.
6. Известны расстояния от Москвы до нескольких городов. Найдите расстояние от Москвы до самого близкого от нее города (из представленных в списке).
7. Известны данные о температуре воздуха в течение месяца. Определите максимальную температуру в этом месяце.
8. Известны данные о количестве осадков, выпавших в каждый день месяца. Какого числа выпало больше всего осадков? Если таких дней несколько, то должна быть найдена дата последнего из них.
9. Дана последовательность из 10 целых чисел. Определите, имеется ли в ней число 13.
10. Дана последовательность из 20 целых чисел. Определите максимальное четное число в этой последовательности. (При решении надо учесть, что четных чисел в последовательности может не быть.)

5. Рекуррентные соотношения

И в математике, и в информатике часто встречаются последовательности чисел, в которых каждый последующий член выражается через предыдущие. Например, такими последовательностями являются *прогрессии* — арифметическая и геометрическая. В первой из них, как известно, каждый последующий член равен предыдущему, увеличенному на разность прогрессии: $a_i = a_{i-1} + d$, а во второй — предыдущему, умноженному на знаменатель прогрессии: $a_i = a_{i-1} \times k$.

Формулы, выражающие очередной член последовательности через один или несколько предыдущих членов, называют «*рекуррентными соотношениями*».

Как вычислить n-й член последовательности, заданной рекуррентным соотношением? Иногда для его расчета есть простая формула, например, для арифметической прогрессии — $a_n = a_1 + d(n-1)$. Чаще, однако, такой простой формулы нет или она неизвестна. В этом случае члены последовательности вычисляют по рекуррентному соотношению один за другим от $i = 1$ до $i = n$.

Возможно два способа таких вычислений: с использованием массивов или без них.

Первый способ проиллюстрируем на примере последовательности: 2, 3, 5, 9, 17, ..., задаваемой рекуррентным соотношением $a_i = 2 \times a_{i-1} - 1$, где $a_1 = 2$.

Составим программу для расчета n-го члена такой последовательности. Так как номер искомого ее члена заранее неизвестен, то массив, в котором будут храниться вычисляемые значения, следует описать, так сказать, «с запасом»[12], например, на 100 элементов (т. е. принимаем, что $n \le 100$). Далее будем вычислять члены последовательности по заданному рекуррентному соотношению один за другим от $i = 1$ до $i = n$, запоминая значение i-го члена в элементе a[i] массива a:

```
цел k      |константа - размер массива
k := 100
алг Расчет1
нач цел таб a[1:k], цел i, n
```

[12] В языке программирования Бейсик допускается описывать массив уже после задания количества его элементов, так что описанная здесь «хитрость» в программе на Бейсике вам не понадобится.

```
| вывод нс, "Задайте номер искомого члена последовательности"
| ввод n
| a[1] := 2            |Первый член последовательности
| нц для i от 2 до n
| | a[i] := 2 * a[i - 1] - 1 |Очередные члены последовательности
| кц
| вывод нс, "Искомый член последовательности равен ", a[n]
кон
```

Недостатком здесь является то, что, хотя нас интересует лишь n-й член последовательности, в процессе выполнения программы компьютер вычисляет и хранит в памяти все члены последовательности от 1-го до n-го. Кроме того, в памяти выделяется место для большого количества не используемых элементов массива (описанного, как указывалось, с запасом).

Перечисленных недостатков лишен второй метод вычислений — без использования массивов. Действительно, если для вычисления следующего члена последовательности нужно знать только значение предыдущего, то можно не запоминать все члены последовательности в массиве, а иметь одну переменную a и при увеличении номера i изменять ее значение (для рассмотренного примера — a := 2 * a - 1). Другими словами, рекуррентное соотношение $a_i = 2 \times a_{i-1} - 1$ можно заменить соотношением: $a_{новое} = 2 \times a_{старое} - 1$, которое записывается в виде оператора присваивания a := 2 * a - 1:

```
алг Расчет2
нач цел a, i, n
| вывод нс, "Задайте номер искомого члена последовательности"
| ввод n
| a := 2
| нц для i от 2 до n
| | a: = 2 * a - 1
| кц
| вывод нс, "Искомый член последовательности равен ", a
кон
```

В рассмотренных примерах (прогрессии и др.) очередной член последовательности выражается только через один предыдущий элемент. Но часто количество элементов, от которых зависит очередной член последовательности, больше одного. Рассмотрим, например, последовательность чисел: 1, 1, 2, 3, 5, 8, 13, ... Конечно, вы уже установили закон, по которому она строится, — каждый следующий член последовательности, начиная с третьего, равен сумме двух предыдущих. Знаете вы, наверное, и то, что такую последовательность называют *числами Фибоначчи*. Для нее рекуррентное соотношение имеет вид: $a_i = a_{i-2} + a_{i-1}$, $a_1 = 1$, $a_2 = 1$.

Очевидно, что если для расчета n-го члена последовательности Фибоначчи не использовать массивы, то в процессе вычислений надо хранить значения двух предшествующих элементов.

Разработаем соответствующую программу. Используем в ней следующие основные величины: n — номер искомого члена последовательности; очер — очередной рассчитываемый элемент последовательности; пред — элемент, предшествующий очередному элементу; предпред — элемент, предшествующий элементу пред.

Сначала имеем: пред = 1; предпред = 1. Затем рассчитываем очередной (третий) элемент по формуле: след = пред + предпред, после чего готовимся к расчету следующего элемента:

```
предпред = пред
пред = след
```

и определяем его: очер = пред + предпред, и т. д.

Ясно, что для определения n-го члена последовательности необходимо провести (n – 2) повторений описанных действий:

```
алг Последовательность_Фибоначчи
нач цел n, очер, пред, предпред, i
| вывод нс, "Задайте номер искомого члена последовательности"
| ввод n
| пред := 1
| предпред := 1
| нц для i от 1 до n - 2
| | очер := пред + предпред
| | предпред := пред
| | пред := очер
| кц
| вывод нс, "Искомый член последовательности равен ", очер
кон
```

В заключение рассмотрим интересный пример использования рекуррентных соотношений [1]. Пусть, например, требуется вычислить общее сопротивление гирлянды из n параллельно соединенных лампочек (каждая сопротивлением R2), если каждый соединительный провод имеет сопротивление R1 (рис. 5.1).

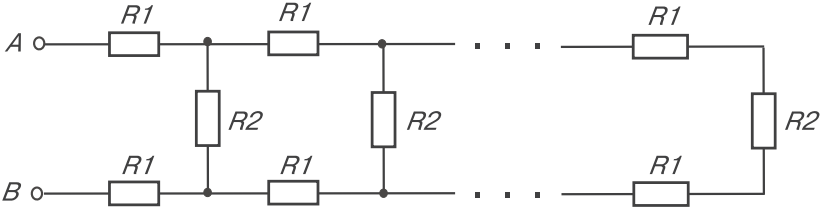


Рис. 5.1

Гирлянда с одной лампочкой выглядит, как показано на рис. 5.2, а ее сопротивление (элементы соединены последовательно) равно: $a_1 = 2R1 + R2$.

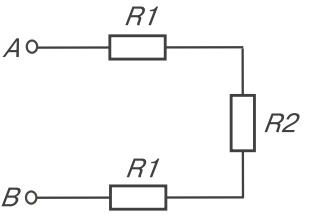


Рис. 5.2

Гирлянду с двумя лампочками (рис. 5.3) можно также представить по-другому (рис. 5.4), где Г1 — гирлянда с одной лампочкой.

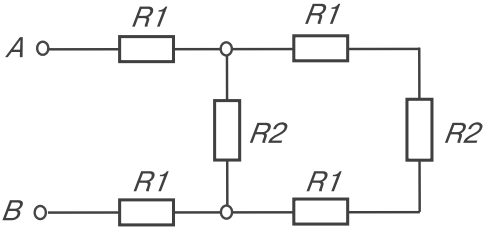


Рис. 5.3

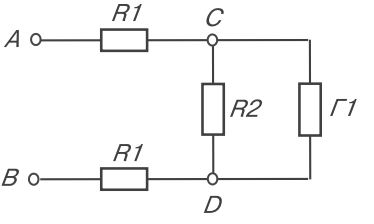


Рис. 5.4

Тогда общее сопротивление схемы на рис. 5.4 можно определить следующим образом. На участке CD параллельно соединены сопротивления R2 и a_1, где a_1 — сопротивление гирлянды с одной лампочкой, т. е. общее сопротивление участка CD равно:

$$a_{CD} = \frac{1}{\frac{1}{R2} + \frac{1}{a_1}} = \frac{a_1 \cdot R2}{a_1 + R2}.$$

Так как участки AC, CD и DB соединены последовательно, то общее сопротивление схемы с двумя лампочками равно:

$$a_2 = R1 + R1 + a_{CD} = 2R1 + \frac{a_1 \cdot R2}{a_1 + R2}.$$

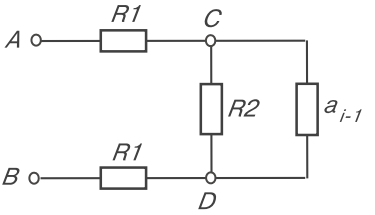


Рис. 5.5

Рассуждая аналогично, схему с i лампочками можно рассматривать в виде, показанном на рис. 5.5, где a_{i-1} — сопротивление гирлянды с $(i-1)$ лампочками, а общее сопротивление схемы на рис. 5.5 определить по формуле:

$$a_i = 2R1 + \frac{a_{i-1} \cdot R2}{a_{i-1} + R2}.$$

Таким образом, мы получили рекуррентное соотношение для расчета сопротивления исходной схемы. Программу для определения общего сопротивления схемы при любом количестве лампочек подготовьте самостоятельно.

---

1. В приведенной программе расчета *n*-го члена последовательности Фибоначчи последние вычисленные значения величин предпред и пред не используются, что не совсем рационально. Измените программу так, чтобы эти значения использовались для определения искомой величины.
2. Начав тренировки, лыжник в первый день пробежал 10 км. Каждый следующий день он увеличивал пробег на 10% от пробега предыдущего дня. Определите, какой суммарный путь он пробежал за первые 7 дней тренировок.
3. В некотором году (назовем его условно «первым») на участке в 100 гектаров средняя урожайность ячменя составила 20 центнеров с гектара. После этого каждый год площадь участка увеличивалась на 5%, а средняя урожайность — на 2%. Определите, какой урожай будет собран за первые шесть лет.
4. Последовательность чисел $a_0, a_1, a_2, \ldots$ образуется по закону: $a_0 = 1$; $a_k = a_{k-1} + 1/k$, где $k = 1, 2, \ldots$ Дано натуральное число n $(n \geq 2)$. Вычислите a_n.
5. Последовательность чисел $v_1, v_2, v_3, \ldots$ образуется по закону: $v_1 = v_2 = 0$; $v_3 = 1{,}5$, $v_i = \frac{i+1}{i^2+1} v_{i-1} - v_{i-2} \cdot v_{i-3}$, где $i = 4, 5, \ldots$ Дано натуральное число n $(n \geq 4)$. Вычислите v_n.

6. Составьте программу вычисления суммы:

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!},$$

где $k! = 1 \cdot 2 \cdot 3 \ldots k$ (факториал числа k). (При расчетах операцию возведения в степень и функцию для расчета факториала числа k не использовать.)

7. Найдите 10-й член последовательности, начинающейся с числа 2.5, в которой каждый следующий член равен сумме обратных величин всех предыдущих.

8. При положительном значении a рекуррентное соотношение:

$$x_i = \frac{x_{i-1}}{2} + \frac{a}{2x_{i-1}}$$

можно использовать для вычисления $\sqrt{a}$, так как элементы последовательности, построенной на таком соотношении, при увеличении i очень быстро приближаются к $\sqrt{a}$. Вот, например, как выглядит начало этой последовательности при $a = 2$:

$x_1 = 1;$
$x_2 = 1.5;$
$x_3 = 1.4166666667;$
$x_4 = 1.4142156863;$
$x_5 = 1.4142135624;$
$x_6 = 1.4142155624;$
…

Составьте программу для определения $\sqrt{a}$ при заданном a и заданной погрешности e. (Расчеты должны выполняться, пока рассчитываемое значение не изменится на величину, меньшую e, — см. приведенные выше значения членов последовательности. Принять $x_1 = 1$.)

6. Типовые задачи обработки одномерных числовых массивов

В этом разделе рассмотрены методы решения двадцати типовых задач, которые встречаются при обработке одномерных числовых массивов (поиск суммы их элементов, максимального значения и др.). Для каждой задачи приведен фрагмент программы на школьном алгоритмическом языке. При этом использованы следующие основные величины: a — имя массива; n — общее количество элементов массива (условно принято, что нумерация элементов массива начинается с 1). Смысл остальных величин можно легко определить по их именам.

6.1. Нахождение суммы всех элементов массива

```
сумма := 0[13]
нц для i от 1 до n
| сумма := сумма + a[i]
кц
|Вывод результата или его использование в расчетах
...
```

Пример. В массиве записан вес каждого из n предметов, загружаемых в автомобиль. Определите общий вес груза.

Решение:

```
сумма := 0
нц для i от 1 до n
| сумма := сумма + a[i]
кц
вывод нс, "Общий вес всех предметов равен ", сумма
```

[13] Как уже отмечалось в разделе 4, в языках программирования Бейсик, Паскаль, Си и ряде других начальное присваивание переменной нулевого значения не является обязательным.

6.2. Нахождение суммы элементов массива с заданными свойствами (удовлетворяющих некоторому условию)

```
сумма := 0
нц для i от 1 до n
| если <условие>
| | то
| |  сумма := сумма + a[i]
| все
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. Здесь условие в условном операторе (команде если) может определяться значением элемента массива a[i] или его индексом i.

6.3. Нахождение количества элементов массива с заданными свойствами

```
количество := 0
нц для i от 1 до n
| если <условие>
| | то
| |  количество := количество + 1
| все
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. В данном случае условие в условном операторе (команде если) определяется значением элемента массива a[i]. Количество элементов, зависящих от значения индекса i, может быть найдено без использования оператора цикла (самостоятельно убедитесь в этом).

6.4. Нахождение среднего арифметического элементов массива с заданными свойствами

```
сумма := 0
количество := 0
нц для i от 1 до n
| если <условие>
| | то
| |  сумма := сумма + a[i]
| |  количество := количество + 1
| все
кц
|Подсчет результата
  среднее_арифметическое := сумма / количество
|Вывод результата или его использование в расчетах
...
```

Обратим внимание, что многократно определять значение среднее_арифметическое в теле условного оператора (команды если) нет необходимости. Это можно сделать один раз после окончания оператора цикла. Однако если чисел, удовлетворяющих заданному условию, в массиве не окажется, то возникнет ситуация деления на нуль, что недопустимо. Поэтому правильным будет следующий фрагмент программы:

```
...
|Подсчет и вывод результата
если количество > 0
| то
|  среднее_арифметическое := сумма / количество
|  вывод нс, "Среднее арифметическое: ", среднее_арифметическое
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, в массиве нет"
все
```

6.5. Изменение значений элементов массива с заданными свойствами

```
нц для i от 1 до n
| если <условие>
| | то
| |  a[i] := ...
| все
кц
```

6.6. Вывод на экран элементов массива с заданными свойствами

```
вывод нс, "Элементы массива, удовлетворяющие условию: "
нц для i от 1 до n
| если <условие>
| | то
| |  вывод  a[i], " "
| все
кц
```

Этот фрагмент имеет единственный недостаток: если в массиве нет элементов, удовлетворяющих заданному условию, то первая команда вывод будет излишней. Поэтому желательно предварительно определить количество чисел в массиве, обладающих заданными свойствами (см. задачу 6.3), а затем оформить фрагмент программы в виде:

```
если количество > 0
| то
|  вывод нс, "Элементы массива, удовлетворяющие условию: "
|  нц для i от 1 до n
|  | если <условие>
|  | | то
|  | |  вывод  a[i], "  "
|  | все
|  кц
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, в массиве нет"
все
```

6.7. Нахождение номеров (индексов) элементов массива с заданными свойствами

```
вывод нс, "Номера элементов массива, удовлетворяющих условию: "
нц для i от 1 до n
| если <условие>
| | то
| |  вывод  i, "  "
| все
кц
```

Здесь тоже можно учесть замечание, сделанное при решении предыдущей задачи.

6.8. Определение индекса элемента массива, равного заданному числу

В приведенном ниже фрагменте программы `значение` — это число, индекс которого (`искомый_индекс`) требуется найти:

```
нц для i от 1 до n
| если a[i] = значение
| | то
| |  искомый_индекс := i
| все
кц
|Вывод результата или его использование в расчетах
...
```

Прежде чем обсуждать этот вариант листинга, предлагаю читателям ответить на вопрос: индекс какого элемента будет найден, если в массиве окажется несколько элементов, значения которых равны искомому?

Нетрудно также заметить, что в случае, если заданного числа значение в массиве нет, то результат работы показанной выше программы будет неправильным. Чтобы учесть это, необходимо изменить фрагмент следующим образом:

```
искомый_индекс := 0
нц для i от 1 до n
| если a[i] = значение
| | то
| |  искомый_индекс := i
| все
кц
|Вывод результата
если искомый_индекс > 0
| то
|  вывод нс, "Индекс этого элемента: ", искомый_индекс
| иначе
|  вывод нс, "Такого числа в массиве нет"
все
```

Однако и этот вариант нерационален: если искомое число находится, например, во втором элементе массива, то все равно придется просматривать весь массив. Желательно прекратить проверку элементов массива, как только встретится искомое значение; для этого следует использовать оператор цикла с условием[14]:

```
искомый_индекс := 1
нц пока искомый_индекс <= n и a[искомый_индекс] <> значение
| искомый_индекс := искомый_индекс + 1
кц
...
```

Но здесь опять возникла уже встречавшаяся нам проблема — в случае, если числа значение в массиве нет, результат работы программы будет неправильным (каким именно — определите самостоятельно). Учесть возможность такого случая можно при выводе результата следующим образом:

```
...
|Вывод результата
если искомый_индекс <= n
| то
|  вывод нс, "Индекс этого элемента: ", искомый_индекс
| иначе
|  вывод нс, "Такого числа в массиве нет"
все
```

[14] Условие, записываемое в нем, можно определить по правилу, описанному в разделе 4.

Интересно, что тем самым мы решили и другую задачу — определить, имеется ли в массиве элемент, равный некоторому значению. Напомним, что ее можно решить, также используя следующий фрагмент:

```
искомый_индекс := 1
есть := нет
нц пока не есть и искомый_индекс <= n
| если a[i] = значение
| | то
| |  есть := да
| | иначе |Переходим к рассмотрению следующего элемента
| |  искомый_индекс := искомый_индекс + 1
| все
кц
если есть
| то
|  вывод нс, "Индекс этого элемента: ", искомый_индекс
| иначе
|  вывод нс, "Такого числа в массиве нет"
все
```

где `есть` — величина логического типа, принимающая истинное значение, если `значение` встретится в массиве.

6.9. Определение индекса элемента, равного заданному числу, для массива, отсортированного по возрастанию

Эта задача отличается от предыдущей тем, что заданный массив *отсортирован*. Это обстоятельство можно использовать, чтобы значительно уменьшить количество проверяемых элементов. Идея решения здесь аналогична отгадыванию некоторого задуманного числа из заданного интервала (например, от 1 до 99) с помощью вопросов, на которые задумавший число отвечает «да» или «нет». Сначала надо спросить: «Ваше число больше 50?». Затем, в зависимости от ответа, — «Ваше число больше 25?» или «Ваше число больше 75?», и т. д. При таком способе решения (который называется *методом бинарного поиска*) диапазон возможного расположения заданного числа после каждого ответа уменьшается в 2 раза, в результате чего количество вопросов, задаваемых для отгадывания числа, не превышает $[\log_2 100] + 1$, где квадратные скобки соответствуют целой части числа, записанного между ними[15].

В программе для решения этой задачи, реализующей метод бинарного поиска, используем следующие основные величины: `значение` —

[15] Зависимость предельного количества вопросов для отгадывания числа из диапазона от 1 до n получите самостоятельно (оно не всегда равно $[\log_2 100] + 1$).

число, индекс которого (искомый_индекс) ищется; левая_граница — индекс начала рассматриваемой при поиске части массива; правая_граница — то же для конца рассматриваемой части массива; середина — величина, равная среднему арифметическому значений левая_граница и правая_граница.

Будем рассуждать так. Сначала имеем:

```
левая_граница := 1; правая_граница := n
```

Находим значение величины середина:

```
середина := div(левая_граница + правая_граница,2)
```

Далее рассматриваем три возможных варианта:

1) значение > a[середина]; в этом случае искомое число находится в правой половине массива (среди элементов с индексами от середина + 1 до правая_граница);

2) значение < a[середина]; при этом искомое число находится в левой половине массива (среди элементов с индексами от левая_граница до середина – 1);

3) значение = a[середина]; это означает, что заданное число находится в массиве в ячейке с индексом середина, т. е. дальнейшие действия можно прекратить, поскольку задача решена (искомый_индекс = середина).

В двух первых вариантах необходимо затем аналогичные рассуждения применить к новой части массива, и т. д. Действия прекращаются, когда левая_граница > правая_граница или когда наступит третий из указанных выше вариантов (чтобы зафиксировать его наступление, в программе можно использовать величину логического типа искомый_индекс_найден).

Соответствующий фрагмент программы оформим следующим образом:

```
левая_граница := 1; правая_граница := n
искомый_индекс_найден := нет
нц
| середина := div(левая_граница + правая_граница, 2)
| если значение > a[середина]
| |то
| | |Искомое число может находиться
| | |только в правой половине массива
| | левая_граница := середина + 1
| | иначе
| |  если значение < a[середина]
| |  | то
| |  |  |Искомое число может находиться
| |  |  |только в левой половине массива
| |  |  правая_граница := середина – 1
```

```
| |   | иначе       |значение = a[середина]
| |   |  искомый_индекс_найден := да
| |   |  искомый_индекс := середина
| |   все
| все
кц при искомый_индекс_найден = да или левая_граница > правая_граница
если искомый_индекс_найден
|то
| вывод нс, "Индекс этого числа равен "
| вывод искомый_индекс
|иначе
| вывод нс, "Такого числа в массиве нет"
все
```

Самостоятельно определите, как будет работать описанная программа в случае, когда в массиве имеется несколько элементов, равных заданному числу.

6.10. Определение максимального элемента в массиве

Алгоритм решения этой задачи аналогичен ранее рассмотренному алгоритму поиска максимального числа в последовательности:

```
максимальное := a[1]
нц для i от 2 до n
| если a[i] > максимальное
| | то
| |  максимальное := a[i]
| все
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. При незначительном изменении данного фрагмента можно обеспечить поиск минимального элемента массива.

Если минимальное число в массиве заранее известно, то указанный фрагмент программы решения задачи можно оформить так:

```
максимальное := минимальное
нц для i от 1 до n
| если a[i] > максимальное
| | то
| |  максимальное := a[i]
| все
кц
```

где минимальное — это минимальное число в массиве. Например, если известно, что в массиве все элементы неотрицательные, то можно записать:

```
максимальное := 0
нц для i от 1 до n
| если a[i] > максимальное
| | то
| |  максимальное := a[i]
| все
кц
```

6.11. Определение индекса максимального элемента в массиве

Здесь алгоритм решения задачи также аналогичен рассмотренному выше, но кроме очередного «максимального» числа нужно запоминать и его индекс:

```
максимальное := a[1]
номер_максимального := 1
нц для i от 2 до n
| если a[i] > максимальное
| | то
| |  максимальное := a[i]
| |  номер_максимального := i
| все
кц
|Вывод результата или его использование в расчетах
...
```

Интересно, что при решении этой задачи можно обойтись без использования переменной максимальное. В самом деле, если нам известно (запомнено) значение индекса максимального элемента массива, то мы знаем и его значение (оно равно a[номер_максимального]):

```
номер_максимального := 1
нц для i от 2 до n
| если a[i] > a[номер_максимального]
| | то
| |  номер_максимального := i
| все
кц
```

Красивое решение, не правда ли?

6.12. Определение максимального значения среди элементов массива, удовлетворяющих некоторому условию

Здесь возможны два случая:

1) заранее известно, что числа, удовлетворяющие заданному условию, в исследуемом массиве точно имеются;
2) чисел, удовлетворяющих заданному условию, в обрабатываемом массиве может и не быть.

В первом случае задача решается следующим образом:

```
максимальное := X
нц для i от 1 до n
| если <условие>
| | то  |Встретилось число, удовлетворяющее заданному условию
| |  |Сравниваем его с величиной максимальное
| |  если a[i] > максимальное
| |  | то
| |  |  максимальное := a[i]
| |  все
| все
кц
```

где X — число, о котором заранее известно, что оно не превышает минимального из чисел, для которых надо определить максимальное значение.

Если же такое значение заранее не известно[16], задача несколько усложняется. В приведенном ниже фрагменте используется величина логического типа впервые, определяющая, впервые ли встретилось в массиве число, удовлетворяющее заданному условию. Остальные особенности работы программы описаны в ее комментариях:

```
впервые := да
нц для i от 1 до n
| если <условие>
| | то |Встретился элемент массива, удовлетворяющий
| |    |заданному условию
| |  если впервые            |Если он встречен впервые,
| |  | то
| |  |  максимальное := a[i] |принимаем его в качестве
| |  |                       |значения максимальное
| |  |  впервые := нет       |Следующие такие числа уже будут
| |  |                       |встречаться не впервые
| |  | иначе                 |Если не впервые,
| |  |  |сравниваем число с величиной максимальное
```

[16] Вопрос о том, всегда ли можно найти такое значение, исследуйте самостоятельно.

```
| | | если a[i] > максимальное
| | | | то
| | | | максимальное := a[i]
| | | все
| | все
| все
кц
|Вывод результата или его использование в расчетах
...
```

Если же допускается, что чисел, удовлетворяющих заданному условию, в обрабатываемом массиве может и не быть, то для решения рассматриваемой задачи можно использовать последний приведенный фрагмент программы:

```
впервые := да
нц для i от 1 до n
| ...
кц
```

но вывод ответа при этом должен быть уточнен:

```
если впервые = да
| то
| вывод нс, "Чисел, удовлетворяющих условию, "
| вывод "в массиве нет"
| иначе
| вывод нс, "Искомое максимальное значение: ", максимальное
все
```

6.13. Определение места заданного числа в упорядоченном массиве

Речь идет об определении места, которое заданное число M должно занимать в массиве, упорядоченном по возрастанию (рис. 6.1), при сохранении его упорядоченности, если известно, что число M меньше последнего элемента массива.

Такая задача может быть решена несколькими способами. Один из них заключается в подсчете количества элементов массива, меньших M (см. ранее). Если это количество равно k, то искомое место (индекс) равен (k + 1). Недостаток этого способа заключается в том, что приходится рассматривать все элементы массива, в то время как может оказаться, что искомое место находится в самом начале массива.

2	4	5	12	20	24	31	45	50	53

Рис. 6.1

Более рационально применить метод, который обычно использует человек, решая такую задачу: он последовательно сравнивает первый, второй и т. д. элементы с числом M, и если рассматриваемый элемент меньше M, то переходит к следующему элементу, а в противном случае прекращает поиск (считается, что искомое место найдено). В программе это оформляется так:

```
искомый_индекс := 1;
нц пока а[искомый_индекс] < M
| искомый_индекс := искомый_индекс + 1
кц
|Вывод ответа
вывод нс, "Номер элемента, в котором должно находиться число ", M
вывод "равен ", искомый_индекс
```

1. Правильно ли будет работать эта программа, если число M меньше самого первого элемента массива?
2. Можно ли начинать проверку элементов с конца массива?

Нетрудно видеть, что в приведенном фрагменте, чтобы найти нужное место, в среднем необходимо n/2 проверок условия а[искомый_индекс] < M. Это количество проверок можно существенно сократить, если для нахождения искомого места в массиве применить идею бинарного поиска. (Программу для решения данной задачи с использованием этого метода разработайте самостоятельно.)

Если же о числе M точно известно, что оно больше первого элемента массива, то проверку можно проводить с конца массива:

```
искомый_индекс := n;
нц пока а[искомый_индекс] > M
| искомый_индекс := искомый_индекс - 1
кц
```

однако после окончания работы оператора цикла найденное значение величины искомый_индекс должно быть уточнено (почему — определите самостоятельно):

```
искомый_индекс := искомый_индекс + 1
|Вывод ответа
...
```

Ясно, что при этом число M не должно быть больше последнего элемента массива.

6.14. Обмен местами двух элементов массива с заданными номерами

Здесь, как и при обмене значениями двух простых переменных, необходимо использовать вспомогательную переменную или использовать прием, описанный в соответствующем разделе. Если индексы обмениваемых элементов — m1 и m2, то в первом случае фрагмент программы выглядит так:

```
вспомогательная := a[m1]
a[m1] := a[m2]
a[m2] := вспомогательная
```

а во втором:

```
a[m1] := a[m1] + a[m2]
a[m2] := a[m1] - a[m2]
a[m1] := a[m1] - a[m2]
```

6.15. Удаление из массива k-го элемента со сдвигом всех расположенных справа от него элементов на одну позицию влево

Операторы, осуществляющие сдвиг необходимых элементов:

```
a[k] := a[k + 1]
a[k + 1] := a[k + 2]
...
a[n - 1] := a[n]
```

можно оформить с использованием оператора цикла в виде:

```
нц для i от k до n - 1
| a[i] := a[i + 1]
кц
```

или

```
нц для i от k + 1 до n
| a[i - 1] := a[i]
кц
```

Примечание. После произведенных изменений следует не забывать, что количество элементов исходного массива уменьшилось на 1.

Обратим также внимание на прием, который был применен выше: если оператор цикла с параметром сразу оформить сложно, то следует выписать действия, являющиеся телом этого оператора, принять изменяющуюся величину в качестве параметра оператора цикла и определить его начальное и конечное значения. Этот прием будет нами использован и при решении нескольких следующих задач.

6.16. Вставка в массив заданного числа на k-е место со сдвигом k-го, (k + 1)-го, (k + 2)-го и т. д. элемента на одну позицию вправо

Прежде всего заметим, что для решения этой задачи размер массива должен быть задан при его описании с учетом дополнительного элемента. Еще одна особенность заключается в том, что в данном случае операторы, реализующие сдвиг элементов вправо, мы не можем записать как:

```
a[k + 1] := a[k]
a[k + 2] := a[k + 1]
...
a[n] := a[n - 1]
```

Понятно, почему не можем? Правильный ответ — сдвиг надо проводить, начиная с *конца* изменяемой части массива:

```
a[n] := a[n - 1]
a[n - 1] := a[n - 2]
...
a[k + 2] := a[k + 1]
a[k + 1] := a[k]
```

или используя при этом оператор цикла:

```
нц для i от n до k + 1 шаг -1
| a[i] := a[i - 1]
кц
```

А теперь, уже после сдвига, можно в k-ю ячейку записать заданное число:

```
a[k] := ...
```

6.17. Циклическое перемещение элементов массива влево

Циклическим перемещением элементов массива влево называется такая их перестановка, когда первый элемент массива записывается на место последнего, сдвинув второй, третий и т. д. элементы на одну позицию влево — рис. 6.2[17].

Ясно, что сразу записать в программе: `a[n] := a[1]` нельзя (так как исходное значение `a[n]` будет утеряно). Нельзя сначала провести и сдвиг элементов влево (`a[1] := a[2]` и т. д.). Как же быть? Правильно! Нужно предварительно запомнить значение первого элемента массива во вспомогательной переменной, затем провести сдвиг

[17] В общем случае задача формулируется следующим образом: «s-й элемент массива записать на место k-го, при этом сдвинув (s + 1)-й, (s + 2)-й, …, k-й элементы на одну позицию влево (s < k)».

5	12	8	0	14	5	1	3	23	4	8	0	9

а) исходный массив

12	8	0	14	5	1	3	23	4	8	0	9	5

б) конечный массив

Рис. 6.2

необходимых элементов влево, а потом запомненное значение записать в последнюю ячейку массива:

```
вспомогательная := a[1]
нц для i от 1 до n - 1
| a[i] := a[i + 1]
кц
a[n] := вспомогательная
```

6.18. Циклическое перемещение элементов массива вправо

Циклическим перемещением элементов массива вправо называется такая их перестановка, когда последний элемент записывается на место первого, сдвинув первый, второй и т. д. элементы на одну позицию вправо — рис. 6.3[18].

12	3	0	23	5	7	13	7	7	45	8	2	1

а) исходный массив

1	12	3	0	23	5	7	13	7	7	45	8	2

б) конечный массив

Рис. 6.3

Вспомнив решения предыдущих задач, мы можем записать:

```
вспомогательная := a[n]
нц для i от n до 2 шаг -1
| a[i] := a[i - 1]
кц
a[1] := вспомогательная
```

[18] В общем случае задача формулируется следующим образом: «s-й элемент массива записать на место k-го, сдвинув k-й, (k + 1)-й, ..., (s – 1)-й элементы на одну позицию вправо (s > k)».

6.19. Проверка массива на упорядоченность по неубыванию (определение, верно ли, что каждый его элемент, начиная со второго, не меньше предыдущего)

Здесь, как и при решении ряда предыдущих задач, всегда рассматривать весь массив нецелесообразно, поэтому желательно применить оператор цикла с условием (например, с постусловием). Условие окончания работы этого оператора — «встретился элемент, меньший предыдущего, или рассмотрены все элементы массива»:

```
i := 1
нц
| i := i + 1
кц_при a[i] < a[i - 1] или i > n
```

После этого ответ можно получить по значению величины i:

```
если i <= n
| то
|  вывод нс, "Массив не упорядочен"
| иначе
|  вывод нс, "Массив упорядочен"
все
```

6.20. Проверка наличия в массиве одинаковых элементов

Можно подсчитать количество в массиве каждого i-го элемента k_i (см. соответствующую задачу). Если одинаковых элементов в массиве нет, то сумма всех таких количеств равна общему числу элементов массива (самостоятельно убедитесь в этом). Соответствующая программа:

```
сумма_всех_количеств := 0
нц для i от 1 до n     |Для каждого i-го элемента
| определяем количество таких значений в массиве
| количество := 0      |Начальное присваивание[19]
| нц для j от 1 до n   |j - индексы элементов, с которыми
| |                    |сравнивается значение a[i]
| | если a[j] = a[i]
| | | то
| | |  количество := количество + 1
| | все
| кц
| |Учитываем это количество в общей сумме
| сумма_всех_количеств := сумма_всех_количеств + количество
кц
```

[19] Здесь начальное присваивание нулевого значения для каждого нового i является обязательным!

```
если сумма_всех_количеств = n
| то
|  вывод нс, "В массиве нет одинаковых элементов"
| иначе
|  вывод нс, "В массиве есть одинаковые элементы"
все
```

Количество сравнений значений элементов (которое в приведенной программе равно n^2) можно сократить вдвое, если каждый элемент сравнивать с элементами, расположенными правее него:

```
сумма_всех_количеств := 0
нц для i от 1 до n - 1
| количество := 0
| нц для j от i + 1 до n
| | если a[j] = a[i]
| | | то
| | |  количество := количество + 1
| | все
| кц
| сумма_всех_количеств := сумма_всех_количеств + количество
кц
```

В этом случае ответ определяется следующим образом:

```
если сумма_всех_количеств = 0 |Обратите внимание на условие!
| то
|  вывод нс, "В массиве нет одинаковых элементов"
| иначе
|  вывод нс, "В массиве есть одинаковые элементы"
все
```

Но и этот новый вариант программы имеет недостаток: если в массиве одинаковыми являются, например, уже второй и третий элементы, то рассматривать все равно приходится весь массив. Желательно прекратить проверку, как только встретятся два одинаковых элемента. Это можно сделать, используя в программе вместо цикла с параметром операторы цикла с условием:

```
сумма_всех_количеств := 0
i := 1
нц пока i <= n - 1
| количество := 0
| j := i + 1
| нц пока j <= n
| | если a[j] = a[i]
| | | то
| | |  количество := количество + 1
| | все
```

```
| | j := j + 1
| кц
| сумма_всех_количеств := сумма_всех_количеств + количество
| i := i + 1
кц
```

Однако и в приведенном варианте по-прежнему рассматриваются все элементы. Чтобы прекратить действия в нужный момент, введем переменную логического типа `есть_одинаковые`: как только встретятся два одинаковых элемента, присвоим ей значение `да`. При этом используем эту переменную в условиях:

```
i := 1; есть_одинаковые := нет
нц пока i <= n - 1 и не есть_одинаковые
| j := i + 1
| нц пока j <= n и не есть_одинаковые
| | если a[j] = a[i]
| | | то
| | |  есть_одинаковые := да
| | все
| | j := j + 1
| кц
| i := i + 1
кц
```

Искомый ответ можно получить по значению величины `есть_одинаковые`.

В заключение заметим, что еще одной достаточно распространенной задачей обработки числовых массивов является их *сортировка*, т. е. переразмещение элементов в некотором порядке (как правило, по возрастанию или убыванию). Изучению различных методов решения этой задачи будет посвящен один из следующих разделов.

Самостоятельно решите предлагаемые задачи с использованием массивов:

1. Известно сопротивление каждого из восьми элементов электрической цепи. Все элементы соединены последовательно. Определите общее сопротивление цепи.
2. Известны данные о стоимости каждого из 12 товаров. Найдите общую стоимость товаров, которые стоят дороже 1000 руб. (количество таких товаров заранее неизвестно).
3. Известны данные о количестве осадков, выпавших в каждый день января. Определите общее количество осадков, выпавших второго, четвертого и т. д. января. (Оператор цикла с параметром, шаг изменения которого отличается от 1 или –1, не использовать!)

4. Известны оценки по информатике каждого из 22 учеников класса. Определите количество «пятерок».
5. Известен рост каждого из 25 учеников класса. Рост мальчиков условно задан отрицательными числами. Определите средний рост мальчиков и средний рост девочек.
6. Все отрицательные элементы исходного массива замените на их абсолютную величину.
7. Выведите на экран элементы целочисленного массива, оканчивающиеся на «49».
8. Определите порядковые номера элементов массива, которые больше своих «соседей» (чисел, расположенных справа и слева от них).
9. Найдите индекс элемента в массиве, равного 13. Если такого числа в массиве нет, то выведите на экран соответствующее сообщение.
10. В массиве записано общее количество учащихся в каждой параллели школы (с V по XI класс). Найдите номер самой «старшей» параллели, в которой учится более 120 учащихся.
11. В массиве хранится рост каждого ученика класса (все значения расположены в порядке возрастания). Определите, есть ли в классе ученик, рост которого равен 170 см.
12. Известны расстояния от Москвы до 15 городов. Найдите расстояние от Москвы до самого близкого от нее города (из представленных в списке).
13. Известны данные о температуре воздуха в течение августа. Определите максимальную температуру в августе.
14. Известны данные о количестве осадков, выпавших в каждый день февраля. Какого числа выпало самое большое количество осадков? (Если таких дней несколько, то должна быть найдена дата последнего из них.)
15. Дан массив из 20 целых чисел. Определите максимальное четное число в нем. Рассмотрите два возможных случая:
    1) известно, что четные числа в массиве точно имеются;
    2) допускается, что четных чисел в массиве может и не быть.
16. Решите задачу 6.13, если:
    1) число `M` может быть больше последнего элемента массива, но точно известно, что оно больше его первого элемента;
    2) число `M` может быть меньше первого элемента массива, но точно известно, что оно меньше его последнего элемента.
17. Значения элементов массива упорядочены по убыванию. Определите, на какое место в массиве следует записать число 100, чтобы общая упорядоченность значений сохранилась. (Известно, что минимальное число в массиве меньше 100.)

18. В заданном массиве поменяйте местами имеющиеся в единственном экземпляре числа 5 и 15.
19. В массиве записаны значения роста 13 юношей класса. В середине учебного года ученик, рост которого был записан в третьей ячейке массива, был переведен в другую школу. Внесите это изменение в массив.
20. Измените массив

1	3	10	70	35	47	53	57	58	65	68	72	81

так, чтобы его элементы были расположены по возрастанию (при просмотре слева направо).

21. В массиве записаны значения роста каждого из 20 учеников класса. В середине учебного года в класс был принят новый ученик, рост которого должен быть записан в пятую ячейку массива. Внесите это изменение в массив.
22. Измените массив

1	3	10	23	35	47	53	57	58	65	18	72	81

так, чтобы его элементы были расположены по возрастанию (при просмотре слева направо).

23. Проверьте заданный массив на упорядоченность по невозрастанию, т. е. определите, верно ли, что каждый его элемент, начиная со второго, не превышает предыдущий.

7. Случайные числа в программах

При разработке программ иногда возникает необходимость в получении и использовании *случайных чисел*, значения которых заранее не известны. Приведем несколько примеров:

1) требуется заполнить случайным образом массив из 20 целых чисел значениями из диапазона от 10 до 100 и отсортировать его элементы по возрастанию;
2) требуется промоделировать бросание игрального кубика, т. е. получать при каждом запуске программы случайное целое число, равное 1, 2, 3, 4, 5 или 6;
3) требуется промоделировать тираж лотереи «Спортлото — 5 из 49», т. е. получить 5 различных случайных целых чисел из диапазона от 1 до 49.

Для получения случайных чисел практически во всех современных языках программирования имеется стандартная функция. В школьном алгоритмическом языке ее имя — `rnd`, в языке Бейсик — `RND` (прописной регистр букв, разумеется, здесь не обязателен); в языке Паскаль — `RANDOM`. Формулы для расчета случайного числа `x` различного типа на этих языках программирования приведены в таблице:

Тип величины x	**Диапазон возможных значений**	**Школьный алгоритмический язык**	**Паскаль**	**Бейсик**
Вещественный	$0 \le x \le 1$	x = rnd(1)	x = RANDOM	x = RND
	$0 \le x \le A$	x = rnd(A)	x = RANDOM * A	x = RND * A
	$A \le x \le B$	$x = A$ + rnd($B - A$)	$x = A$ + RANDOM * ($B - A$)	$x = A$ + RND * ($B - A$)
Целый	$0 \le x \le A$	x = rnd(A + 1)	$x = A$ + RANDOM(A + 1)	x = INT(RND * (A + 1)
	$A \le x \le B$	$x = A$ + rnd(B–A+1)	$x = A$ + RANDOM($B - A$ + 1)	$x = A$+ INT(RND*(B–A+1)

Примечание. В программах на языке Бейсик и Паскаль, если указанные функции используются несколько раз, при каждом новом запуске программы будут генерироваться одни и те же случайные числа. Чтобы исключить это, необходимо записать (желательно в начале программы):

1) в программах на Бейсике — оператор `RANDOMIZE` с параметром `TIMER` (в виде `RANDOMIZE TIMER`);
2) в программах на Паскале — оператор `RANDOMIZE` без параметров.

Интересной задачей является формирование *массива случайных чисел*, в котором все числа должны быть различными (см., например, выше задачу о тираже лотереи «Спортлото»). Она может быть решена несколькими методами.

Первый метод. Основная идея его алгоритма заключается в следующем:

1) генерируется случайное число;
2) проверяется, имеется ли уже такое число в заполненной части массива. Если не имеется, то число записывается в массив; в противном случае генерируется новое случайное число и т. д. (до генерации числа, которого еще нет в массиве);
3) описанные действия повторяются до заполнения всего массива.

В приведенной ниже программе, реализующей этот метод, используются следующие основные величины: `a` — заполняемый массив; `n` — общее количество элементов массива; `nn` — очередное случайное число, получаемое с помощью функции `rnd`; `k` — количество уже заполненных значениями элементов массива; `есть` — величина логического типа, фиксирующая факт наличия очередного случайного числа в заполненной части массива.

```
цел n
n := ...      |Размер массива
алг Заполнение_массива_неповторяющимися_числами
нач цел таб a[1:n], цел nn, k, i, лог есть
| a[1] := rnd(100) |Первый элемент массива -
|                  |всегда оригинальный
| k := 1
| нц
| | nn := rnd(100) |Очередное случайное число
| | |Проверяем, есть ли уже такое число в массиве
| | i := 1; есть := нет
| | нц
| | | если a[i] = nn
| | | |то
| | | | есть := да
```

```
| | | |иначе
| | | | i := i + 1
| | | все
| | кц при есть или i > k
| | если не есть  |Числа nn в массиве нет
| | |то
| | | k := k + 1   |Записываем
| | | a[k] := nn  |его в массив
| | все
| кц при k = n
| |Массив a заполнен случайными
| |неповторяющимися числами
| ...
кон
```

Обратим внимание на сложное условие окончания вложенного оператора цикла — условие `i > k` включено в него потому, что значение `i` может превысить значение `k`.

Второй метод. Используем вспомогательный массив `b` — целый по типу, но логический по содержанию. Его элементы принимают значения 0 или 1 и хранят информацию о том, имеется ли уже в массиве `a` число, совпадающее с номером элемента массива `b`. Например, если `b[5] := 0`, то это значит, что числа 5 в массиве `a` еще нет; если же `b[9] := 1`, то девятка в массиве `a` уже есть.

В приведенной ниже программе тело оператора цикла с параметром выполняется `n` раз, и каждый раз случайное число генерируется до тех пор, пока не выяснится, что такого числа в массиве `a` еще нет (`b[a[i]] = 0`):

```
цел n
n := ...            |Размер массива
алг Заполнение_массива_неповторяющимися_случайными_числами
нач цел таб a[1:n], b[1:n], цел i
| нц для i от 1 до n
| | b[i] := 0
| кц
| нц для i от 1 до n
| | нц
| | | |Очередное случайное число
| | | a[i] := 1 + rnd(10)
| | кц при b[a[i]] = 0
| | |Получено и записано в массив a
| | |очередное число a[i]
| | |Делаем "отметку" об этом в массиве b
| | b[a[i]] := 1
| кц
| |Массив a заполнен случайными
| |неповторяющимися числами
| ...
кон
```

Третий метод. Идею этого метода мы проиллюстрируем на примере двумерного массива из n строк и двух столбцов, где в первом столбце надо получить неповторяющиеся числа.

Сначала массив заполняется следующим образом: элементы первого столбца принимают значения номера строки, а второго — случайные целые значения из некоторого интервала; затем строки массива переразмещаются в порядке возрастания[20] значений элементов второго столбца. В результате неповторяющиеся элементы в первом столбце будут расположены «случайным» образом.

В приведенной ниже программе для упорядочивания чисел во втором столбце массива использован метод обмена («пузырьковая» сортировка — см. раздел 11):

```
цел n
n := ...     |Размер массива
алг Заполнение_массива_неповторяющимися_случайными_числами
нач цел таб a[1:n, 1:2], цел i, j, всп
| нц для i от 1 до n
| | a[i, 1] := i
| | a[i, 2] := rnd(20)
| кц
| |Сортируем строки массива
| |по числам во втором столбце
| нц для i от 1 до n - 1
| | нц для j от i + 1 до n
| | | если a[i, 2] > a[j, 2]
| | | |то
| | | | |Меняем местами строки i и j
| | | | |Значения в первом столбце (см. раздел 2)
| | | | всп := a[i, 1]
| | | | a[i, 1] := a[j, 1]
| | | | a[j, 1] := всп
| | | | |Значения во втором столбце
| | | | всп := a[i, 2]
| | | | a[i, 2] := a[j, 2]
| | | | a[j, 2] := всп
| | | все
| | кц
| кц
| |Первый столбец массива a
| |заполнен неповторяющимися
| |числами от 1 до n
| ...
кон
```

[20] Конечно, можно сортировать эти значения и в порядке убывания.

В заключение отметим, что хотя получаемые с помощью указанных выше функций числа называют «случайными», на самом деле они вычисляются по специальным правилам, — ведь, как вы знаете, любые действия в компьютере происходят только по жестко заданным алгоритмам.

1. Получите в программе:
   а) 8 случайных вещественных чисел n_i ($0 \le n_i < 1$);
   б) k вещественных чисел n_i ($0 \le n_i < 1$); значение k вводится с клавиатуры;
   в) 15 вещественных чисел n_i ($38 \le n_i < 39$);
   г) 20 вещественных чисел n_i ($0 \le n_i < 10$);
   д) натуральное k, не превосходящее A, и k вещественных чисел n_i ($0 \le n_i < B$); значения A и B вводятся с клавиатуры;
   е) 10 вещественных чисел n_i ($-50 \le n_i < 50$);
   ж) натуральное k, не превосходящее M, и k вещественных чисел n_i ($A \le n_i < B$); значения M, A и B вводятся с клавиатуры.
2. Получите в программе:
   а) 10 случайных целых чисел, лежащих в диапазоне от 0 до 10 включительно;
   б) k целых чисел, лежащих в диапазоне от 0 до A включительно; значения k и A вводятся с клавиатуры;
   в) 20 целых чисел, лежащих в диапазоне от 10 до 20 включительно;
   г) k целых чисел, лежащих в диапазоне от -10 до A включительно; значения k и A вводятся с клавиатуры;
   д) натуральное k, не превосходящее 15, и k целых чисел, лежащих в диапазоне от A до B включительно; значения A и B вводятся с клавиатуры.
3. Получите случайные натуральные числа m и n, не превосходящие 20, n целых чисел, лежащих в диапазоне от A до B включительно, и m неотрицательных вещественных чисел, не превосходящих n; значения A и B вводятся с клавиатуры.
4. Составьте программу, проверяющую знание таблицы умножения. В ней случайным образом должны выдаваться два целых числа, больших 0 и меньших 10, после чего на экран выводится вопрос о произведении этих чисел (например, в виде: «Чему равно произведение 4×9?»). После ввода ответа должно выводиться сообщение о его правильности. Разработайте следующие варианты такой программы:
   а) вопрос выводится один раз;
   б) вопрос выводится 10 раз, проводится подсчет и вывод на экран количества правильных и неправильных ответов;
   в) вопрос выводится до тех пор, пока в качестве ответа не будет указан «0».

5. Промоделируйте подбрасывание монеты и падение ее одной из сторон — лицевой («решка») или обратной («орел»), т. е. получите одно из случайных целых чисел 0 или 1.
6. Подсчитайте относительную частоту появления каждого из чисел 0 и 1 при 100 и при 1000 «подбрасываниях монеты» (см. предыдущую задачу).
7. Подсчитайте относительную частоту появления каждого из чисел 1, 2, ..., 6 при 100 и при 1000 «бросаниях игрального кубика» (промоделировав его на компьютере — см. задачу 2 в начале раздела 7).
8. Промоделируйте выбор «наугад» одной кости домино из полного набора костей этой игры (0–0, 0–1, ..., 6–6). Выведите состав этой кости в виде, аналогичном следующему: «Выбрана кость 4–3» (0–6, 2–2, 6–0 и т. п.).
9. Промоделируйте выбор «наугад» двух костей домино из полного набора костей этой игры (0–0, 0–1, ..., 6–6) и определите, можно ли приставить эти кости одна к другой в соответствии с правилами домино.
10. Промоделируйте выбор «наугад» одной карты из набора игральных карт одной масти, включающего карты следующих достоинств: «6», «7», «8», «9», «10», «валет», «дама», «король», «туз». Выведите достоинство этой карты.
11. Промоделируйте выбор «наугад» одной карты из полного набора игральных карт, включающего 4 масти («пики», «трефы», «бубны» и «червы») и по 9 достоинств карт в каждой масти («6», «7», «8», «9», «10», «валет», «дама», «король», «туз»). Выведите название этой карты в виде, аналогичном следующему: «Выбрана дама пик», «Выбрана шестерка бубен» и т. п.
12. Промоделируйте выбор «наугад» двух карт из полного набора игральных карт, включающего 4 масти («пики», «трефы», «бубны» и «червы») и по 9 достоинств карт в каждой масти («6», «7», «8», «9», «10», «валет», «дама», «король», «туз»). Выведите названия этих карт в виде, аналогичном следующему: «Выбрана дама пик и шестерка бубен». Определите, какая из двух этих карт «старше» (приведенные выше перечни мастей и карт одной масти даны в порядке увеличения их «старшинства», например любая бубновая карта «старше» любой карты масти «пики», а валет червей «старше» десятки червей). Рассмотрите также вариант, когда имеется козырная масть (любая карта козырной масти «старше» любой карты некозырной масти). Номер козырной масти выбирается случайным образом.
13. Заполните два массива длиной по 10 элементов каждый случайными целыми числами из диапазона от 10 до 100 так, чтобы все 20 чисел были различными. Задачу решите тремя методами.

14. Определите, какие преимущества и недостатки имеет второй метод решения рассмотренной в данном разделе задачи заполнения массива по сравнению с первым методом.
15. Составьте программу заполнения массива из 20 элементов неповторяющимися случайными целыми числами из диапазона от 100 до 200 (на основе второго метода решения задачи), где в качестве вспомогательного используется массив `b` с элементами логического типа (если это возможно в используемом вами языке программирования).
16. Составьте программу, в которой на экран выводятся 10 неповторяющихся случайных целых чисел из диапазона от 10 до 100 (массив не использовать!).
17. Разработайте вариант программы, реализующей третий метод решения задач со случайными числами, в котором вместо двумерного массива используется два одномерных, в первом из которых надо получить неповторяющиеся числа, а второй является вспомогательным.
18. Подумайте, как сделать так, чтобы при третьем методе формируемый массив (или первый столбец двумерного массива) был заполнен:

    а) двадцатью неповторяющимися целыми числами из диапазона от 20 до 39 (в случайном порядке);

    б) десятью неповторяющимися вещественными числами 0.3, 0.4, 0.5, …, 1.2 (в случайном порядке).

    Подготовьте соответствующие варианты программы.

8. Типовые задачи обработки двумерных числовых массивов

В этом разделе рассмотрены методы решения типовых задач, которые встречаются при обработке двумерных числовых массивов (нахождение суммы элементов, максимального значения и др.).

Для каждой задачи приведен фрагмент программы ее решения на школьном алгоритмическом языке. При этом использованы следующие основные величины: a — имя массива; n — количество строк в массиве; m — количество столбцов в массиве (условно принято, что нумерация строк и столбцов начинается с 1). Смысл остальных величин можно легко определить по их именам.

8.1. Нахождение суммы всех элементов массива

```
сумма := 0
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
кц
|Вывод результата или его использование в расчетах
...
```

Возможен и другой вариант программы, в котором элементы массива просматриваются по столбцам:

```
сумма := 0
нц для номер_столбца от 1 до m
| нц для номер_строки от 1 до n
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
кц
|Вывод результата или его использование в расчетах
...
```

Пример. В двумерном массиве из 8 строк и 14 столбцов записано количество жильцов каждой из 8 квартир, находящихся на каждом из 14 этажей жилого дома. Определите общее число жильцов.

Решение

```
сумма := 0
нц для номер_столбца от 1 до 8
| нц для номер_строки от 1 до 14
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
кц
вывод нс, "Общее число жильцов равно ", сумма
```

8.2. Нахождение суммы элементов массива с заданными свойствами (удовлетворяющих некоторому условию)

```
сумма := 0
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  сумма := сумма + a[номер_строки, номер_столбца]
| | все
| кц
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. Здесь условие в условном операторе (команде если) может определяться значением элемента массива a[номер_строки, номер_столбца] или его индексами номер_строки или/и номер_столбца.

8.3. Нахождение количества элементов массива с заданными свойствами

```
количество := 0
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  количество := количество + 1
| | все
|кц
кц
|Вывод результата или его использование в расчетах
...
```

Примечание. В данном случае условие в условном операторе (команде если) определяется значением элемента массива a[номер_строки, номер_столбца]. Количество элементов, зависящих от значений индексов, может быть найдено без использования оператора цикла (самостоятельно убедитесь в этом).

8.4. Нахождение среднего арифметического элементов массива с заданными свойствами

```
сумма := 0
количество := 0
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  сумма := сумма + a[номер_строки, номер_столбца]
| | |  количество := количество + 1
| | все
| кц
кц
|Подсчет результата
 среднее_арифметическое := сумма / количество
|Вывод результата или его использование в расчетах
...
```

Обратим внимание на то, что многократно определять значение среднее_арифметическое в теле условного оператора (команды если) нет необходимости. Это можно сделать один раз после операторов цикла. Однако может оказаться, что чисел, удовлетворяющих заданному условию, в массиве не окажется; тогда возникнет ситуация деления на нуль, что недопустимо. Поэтому правильное решение будет таким:

```
...
|Подсчет и вывод результата
если количество > 0
| то
|  среднее_арифметическое := сумма / количество
|  вывод нс, "Среднее арифметическое: ",
|             среднее_арифметическое
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, в массиве нет"
все
```

8.5. Изменение значений элементов массива с заданными свойствами

```
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  a[номер_строки, номер_столбца] := ...
| | все
| кц
кц
```

8.6. Вывод на экран элементов массива с заданными свойствами

```
вывод нс, "Элементы массива, удовлетворяющие условию: "
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  вывод  a[номер_строки, номер_столбца], " "
| | все
| кц
кц
```

Приведенный здесь фрагмент имеет существенный недостаток: если в массиве нет элементов, удовлетворяющих заданному условию, то первая команда `вывод` будет излишней. Для его исправления необходимо предварительно определить количество элементов массива с заданными свойствами (см. задачу 8.3), а затем оформить данный фрагмент в виде:

```
если количество > 0
| то
|  вывод нс, "Элементы массива, удовлетворяющие условию: "
|  нц для номер_строки от 1 до n
|  | нц для номер_столбца от 1 до m
|  | | если <условие>
|  | | | то
|  | | |  вывод  a[номер_строки, номер_столбца], " "
|  | | все
|  | кц
|  кц
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, в массиве нет"
все
```

8.7. Нахождение индексов элементов массива с заданными свойствами

```
вывод нс, "Индексы элементов массива, удовлетворяющих условию: "
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  вывод  номер_строки, ",", номер_столбца, " "
| | все
| кц
кц
```

Здесь также можно учесть замечание, сделанное при решении предыдущей задачи.

8.8. Определение максимального элемента массива

Решить эту задачу можно аналогично ее решению для одномерных массивов:

```
максимальное := a[1, 1]
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если a[номер_строки, номер_столбца] > максимальное
| | | то
| | |  максимальное := a[номер_строки, номер_столбца]
| | все
| кц
кц
|Вывод результата или его использование в расчетах
...
```

Нетрудно видеть, что в нашем случае значение элемента `a[1, 1]` будет использоваться дважды. Если же минимальное число в массиве заранее известно, то фрагмент программы для решения этой задачи может быть оформлен так:

```
максимальное := минимальное
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если a[номер_строки, номер_столбца] > максимальное
| | | то
| | |  максимальное := a[номер_строки, номер_столбца]
| | все
| кц
кц
```

где `минимальное` — это минимальное число в массиве.

Примечание. При незначительном изменении вышеприведенных фрагментов нетрудно решить задачу поиска минимального элемента массива.

8.9. Определение индексов максимального элемента массива

Здесь решение тоже достигается аналогично одномерному массиву:

```
максимальное := a[1, 1]
строка_максимального := 1
столбец_максимального := 1
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если a[номер_строки, номер_столбца] > максимальное
| | | то
| | |  максимальное := a[номер_строки, номер_столбца]
| | |  строка_максимального := номер_строки
| | |  столбец_максимального := номер_столбца
| | все
| кц
кц
```

При решении этой задачи можно обойтись и без использования величины максимальное. В самом деле, если нам известно значение индексов максимального значения среди рассмотренных элементов (строка_максимального и столбец_максимального), то мы знаем и значение соответствующего элемента (оно равно a[строка_максимального, столбец_максимального]). Соответствующий вариант программы разработайте самостоятельно.

8.10. Определение максимального значения среди элементов массива, удовлетворяющих некоторому условию

Здесь возможны два случая:

1) точно известно, что числа, удовлетворяющие заданному условию, в исследуемом массиве имеются;
2) чисел, удовлетворяющих заданному условию, в обрабатываемом массиве может и не быть.

В первом случае задача решается следующим образом:

```
максимальное := X
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то |Встретилось число, удовлетворяющее заданному условию
| | |    |Сравниваем его с величиной максимальное
| | |  если a[номер_строки, номер_столбца] > максимальное
| | |  | то
```

```
| | |  | максимальное := a[номер_строки, номер_столбца]
| | |  все
| | все
| кц
кц
|Вывод результата или его использование в расчетах
...
```

где X — число, о котором заведомо известно, что оно не превышает минимальное из чисел, для которых нужно определить максимальное значение.

Если же такое значение заранее не известно, то задача несколько усложняется. В приведенном ниже фрагменте программы используется величина логического типа впервые, определяющая, впервые ли встретилось в массиве число, удовлетворяющее заданному условию. Остальные особенности работы программы описаны в ее комментариях.

```
впервые := да
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то | Встретился элемент массива, удовлетворяющий
| | |    | заданному условию
| | |  если впервые            |Если он встретился впервые,
| | |  | то
| | |  |  максимальное := a[i] |принимаем его в качестве
| | |  |                       |значения максимальное
| | |  |  впервые := нет       |Следующие такие числа уже будут
| | |  |                       |встречаться не впервые
| | |  | иначе                 |Если не впервые,
| | |  |  |сравниваем число с величиной максимальное
| | |  |  если a[i] > максимальное
| | |  |  | то
| | |  |  |  максимальное := a[i]
| | |  |  все
| | |  все
| | все
| кц
кц
|Вывод результата или его использование в расчетах
...
```

Если же допускается, что чисел, удовлетворяющих заданному условию, в обрабатываемом массиве может не быть, то для решения задачи можно использовать следующий фрагмент программы:

```
впервые := да
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | ...
| кц
кц
```

но вывод ответа при этом должен быть уточнен:

```
если впервые = да
| то
|  вывод нс, "Чисел, удовлетворяющих условию, "
|  вывод "в массиве нет"
| иначе
|  вывод нс, "Искомое максимальное значение: ", максимальное
все
```

8.11. Нахождение суммы элементов в некоторой строке массива

```
сумма := 0
нц для номер_столбца от 1 до m
| сумма := сумма + a[номер_строки, номер_столбца]
кц
|Вывод результата или его использование в расчетах
...
```

где номер_строки — номер строки, для которой решается задача.

8.12. Нахождение суммы элементов с заданными свойствами в некоторой строке массива

```
сумма := 0
нц для номер_столбца от 1 до m
| если <условие>
| | то
| |  сумма := сумма + a[номер_строки, номер_столбца]
| все
кц
|Вывод результата или его использование в расчетах
...
```

8.13. Нахождение количества элементов с заданными свойствами в некоторой строке массива

```
количество := 0
нц для номер_столбца от 1 до m
| если <условие>
| | то
| |  количество := количество + 1
| все
кц
|Вывод результата или его использование в расчетах
...
```

8.14. Нахождение среднего арифметического значений элементов с заданными свойствами в некоторой строке массива

```
сумма := 0
количество := 0
нц для номер_столбца от 1 до m
| сумма := сумма + a[номер_строки, номер_столбца]
| количество := количество + 1
кц
если количество > 0
| то
|  среднее_арифметическое := сумма / количество
| |Вывод результата или его использование в расчетах
| ...
все
```

8.15. Изменение значений элементов массива с заданными свойствами в некоторой строке массива

```
нц для номер_столбца от 1 до m
| если <условие>
| | то
| |  a[номер_строки, номер_столбца] := ...
| все
кц
```

8.16. Вывод на экран элементов с заданными свойствами из некоторой строки массива

```
...
если количество > 0
| то
|  вывод нс, "Элементы, удовлетворяющие условию: "
|  нц для номер_столбца от 1 до m
|  | если <условие>
|  | | то
|  | |  вывод a[номер_строки, номер_столбца], "  "
|  | все
|  кц
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, "
|  вывод "в заданной строке массиве нет"
все
```

где количество — количество чисел с заданными свойствами в рассматриваемой строке массива (см. задачу 8.13).

8.17. Нахождение индексов элементов массива с заданными свойствами

```
...
если количество > 0
| то
|  вывод нс, "Индексы элементов, удовлетворяющих условию: "
|  нц для номер_столбца от 1 до m
|  | если <условие>
|  | | то
|  | |  вывод номер_строки, ",", номер_столбца, " "
|  | все
|  кц
| иначе
|  вывод нс, "Чисел, удовлетворяющих условию, "
|  вывод "в заданной строке массиве нет"
все
```

где количество — количество чисел с заданными свойствами в рассматриваемой строке массива (см. задачу 8.13).

8.18. Определение максимального элемента в некоторой строке массива

```
максимальное :=  a[номер_строки, 1]
нц для номер_столбца от 2 до m
| если a[номер_строки, номер_столбца] > максимальное
| | то
| |  максимальное := a[номер_строки, номер_столбца]
| все
кц
|Вывод результата или его использование в расчетах
...
```

8.19. Определение индекса столбца максимального элемента в некоторой строке массива

```
максимальное := a[номер_строки, 1]
столбец_максимального := 1
нц для номер_столбца от 2 до m
| если a[номер_строки, номер_столбца] > максимальное
| | то
| |  максимальное := a[номер_строки, номер_столбца]
| |  столбец_максимального := номер_столбца
| все
кц
|Вывод результата или его использование в расчетах
...
```

Самостоятельно разработайте вариант программы, в котором переменная максимальное не используется.

8.20. Нахождение суммы элементов в каждой строке массива

```
нц для номер_строки от 1 до n
| сумма := 0
| нц для номер_столбца от 1 до m
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
| |Вывод результата или его использование в расчетах
кц
```

Обратим внимание, что здесь начальное присваивание нулевого значения **является обязательным**. Заметим также, что найденные для каждой строки суммы могут быть записаны в одномерный массив из n элементов:

```
нц для номер_строки от 1 до n
| сумма := 0
| нц для номер_столбца от 1 до m
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
| сумма_в_строке[номер_строки] := сумма
кц
```

где сумма_в_строке — имя только что упомянутого массива.

Самостоятельно разработайте вариант программы, в котором переменная сумма не используется.

8.21. Нахождение суммы элементов с заданными свойствами в каждой строке массива

```
нц для номер_строки от 1 до n
| сумма := 0
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  сумма := сумма + a[номер_строки, номер_столбца]
| | все
| кц
| |Вывод результата или его использование в расчетах
| ...
кц
```

Здесь, равно как и в других задачах, связанных с обработкой всех строк двухмерного массива, также может быть использован одномерный массив для записи искомых значений.

8.22. Нахождение количества элементов с заданными свойствами в каждой строке массива

```
нц для номер_строки от 1 до n
| количество := 0
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  количество := количество + 1
| | все
| кц
| |Вывод результата или его использование в расчетах
| ...
кц
```

8.23. Нахождение среднего арифметического значений элементов с заданными свойствами в каждой строке массива

Для упрощения программы примем, что элементы с заданными свойствами имеются в каждой строке массива.

```
нц для номер_строки от 1 до n
| сумма := 0
| количество := 0
| нц для номер_столбца от 1 до m
| | если <условие>
| | | то
| | |  сумма := сумма + a[номер_строки, номер_столбца]
| | |  количество := количество + 1
| | все
| кц
| среднее_арифметическое := сумма / количество
| |Вывод результата или его использование в расчетах
| ...
кц
```

8.24. Определение максимального элемента в каждой строке массива

```
нц для номер_строки от 1 до n
| максимальное := a[номер_строки, 1]
| нц для номер_столбца от 2 до m
| | если a[номер_строки, номер_столбца] > максимальное
| | | то
| | |  максимальное := a[номер_строки, номер_столбца]
| | все
| кц
| |Вывод результата или его использование в расчетах
| ...
кц
```

8.25. Определение индекса столбца для максимального элемента в каждой строке массива

```
нц для номер_строки от 1 до n
| максимальное :=  a[номер_строки, 1]
| столбец_максимального := 1
| нц для номер_столбца от 2 до m
| | если a[номер_строки, номер_столбца] > максимальное
| | | то
| | |  максимальное := a[номер_строки, номер_столбца]
| | |  столбец_максимального := номер_столбца
| | все
| кц
| |Вывод результата или его использование в расчетах
| ...
кц
```

Самостоятельно разработайте вариант программы, в котором переменная `максимальное` не используется.

8.26. Определение максимальной суммы значений в строках массива

При использовании массива `сумма_в_строке` со значениями сумм элементов в каждой строке (см. задачу 8.20) искомое значение может быть найдено как максимальное число в этом массиве.

Без использования массива `сумма_в_строке` эта задача решается следующим образом:

```
|Рассчитываем сумму элементов 1-й строки
сумма := 0
нц для номер_столбца от 1 до m
| сумма := сумма + a[1, номер_столбца]
кц
|Принимаем ее в качестве максимальной
максимальная_сумма := сумма
|Определяем сумму элементов
|в каждой из остальных строк
нц для номер_строки от 2 до n
| сумма := 0
| нц для номер_столбца от 1 до m
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
| |Сравниваем ее с максимальной суммой
| если сумма > максимальная_сумма
| | то
| |  максимальная_сумма := сумма
| все
кц
|Вывод результата или его использование в расчетах
...
```

8.27. Определение номера строки массива с максимальной суммой значений

Эта задача также может быть решена с использованием массива сумма_в_строке или без него. Во втором случае соответствующий фрагмент имеет вид:

```
|Рассчитываем сумму элементов 1-й строки
сумма := 0
нц для номер_столбца от 1 до m
| сумма := сумма + a[1, номер_столбца]
кц
|Принимаем ее в качестве максимальной,
максимальная_сумма := сумма
|а в качестве номера строки
|с максимальной суммой принимаем 1
номер_строки_с_максимальной_суммой := 1
|Определяем сумму элементов
|в каждой из остальных строк
нц для номер_строки от 2 до n
| сумма := 0
| нц для номер_столбца от 1 до m
| | сумма := сумма + a[номер_строки, номер_столбца]
| кц
| |Сравниваем ее с максимальной суммой
| если сумма > максимальная_сумма
| | то
| |  максимальная_сумма:= сумма
| |  номер_строки_с_максимальной_суммой := номер_строки
| все
кц
|Вывод результата или его использование в расчетах
...
```

Самостоятельно разработайте вариант программы, в котором переменная максимальная_сумма не используется.

8.28. Обмен местами двух элементов массива с заданными индексами

Эта задача может быть решена двумя способами:

1) с использованием вспомогательной переменной:

```
вспомогательная := a[m1, m2]
a[m1, m2] := a[m3, m4]
a[m3, m4] := вспомогательная
```

2) без использования вспомогательной переменной (см. раздел 2):

```
a[m1, m2] := a[m1, m2] + a[m3, m4]
a[m3, m4] := a[m1, m2] - a[m3, m4]
a[m1, m2] := a[m1, m2] - a[m3, m4]
```

где m1 и m2 — индексы первого из обмениваемых элементов, а m3 и m4 — второго.

8.29. Обмен местами двух строк массива

Эта задача также может быть решена двумя способами:

1) с использованием вспомогательного одномерного массива вспомогательный_массив для временного хранения одной из обмениваемых строк;
2) с использованием вспомогательной величины вспомогательная для временного хранения значения отдельного элемента массива.

Если номера обмениваемых строк — s1 и s2, то в первом случае фрагмент программы выглядит так:

```
|Запоминаем все элементы строки с номером s1
|в массиве вспомогательный_массив
нц для номер_столбца от 1 до m
| вспомогательный_массив[номер_столбца] := a[s1, номер_столбца]
кц
|Переписываем все элементы строки с номером s2
|в соответствующие элементы строки с номером s1
нц для номер_столбца от 1 до m
| a[s1, номер_столбца] := a[s2, номер_столбца]
кц
|Заполняем строку с номером s2
|"старыми" элементами строки с номером s1
нц для номер_столбца от 1 до m
| a[s2, номер_столбца] := вспомогательный_массив[номер_столбца]
кц
```

а во втором:

```
|Каждый элемент строки с номером s1
|меняем местами с соответствующим элементом
|строки с номером s2
нц для номер_столбца от 1 до m
| вспомогательная := a[s1, номер_столбца]
| a[s1, номер_столбца] := a[s2, номер_столбца]
| a[s2, номер_столбца] := вспомогательная
кц
```

Самостоятельно разработайте вариант программы, в котором переменная вспомогательная не используется.

8.30. Удаление из массива k-й строки со сдвигом всех расположенных ниже нее элементов на одну строку вверх

Операторы, осуществляющие сдвиг необходимых элементов:

- с (k+1)-й строки на k-ю:
```
a[k, 1] := a[k + 1, 1]; a[k, 2] := a[k + 1, 2]; ... ;
a[k, m] := a[k + 1, m]
```
- с (k+2)-й строки на (k+1)-ю:
```
a[k + 1, 1] := a[k + 2, 1]; a[k + 1, 2] := a[k + 2, 2];... ;
a[k + 1, m] := a[k + 2, m]
...
```
- с n-й строки на (n–1)-ю:
```
a[n - 1, 1] := a[n, 1]; a[n - 1, 2] := a[n, 2]; ... ;
a[n - 1, m] := a[n, m]
```

Помня, что для каждой строки можно применить операторы цикла:

```
нц для номер_столбца от 1 до m
| a[k, номер_столбца] := a[k + 1, номер_столбца]
кц
нц для номер_столбца от 1 до m
| a[k + 1, номер_столбца] := a[k + 2, номер_столбца]
кц
...
нц для номер_столбца от 1 до m
| a[n - 1, номер_столбца] := a[n, номер_столбца]
кц
```

оформим программу с использованием вложенного оператора:

```
нц для номер_строки от k до n - 1
| нц для номер_столбца от 1 до m
| | a[номер_строки, номер_столбца] := a[номер_строки + 1,
| |                                    номер_столбца]
| кц
кц
```

Примечание. После произведенных изменений следует не забывать, что количество строк исходного массива уменьшилось на 1.

8.31. Вставка в массив заданного одномерного массива на k-ю строку со сдвигом k-й, (k+1)-й, (k+2)-й и т. д. строк на одну позицию вниз

Прежде всего, заметим, что для решения этой задачи массив должен быть описан с учетом дополнительной строки. Еще одна особенность заключается в том, что в данном случае операторы, реализующие сдвиг элементов вниз, мы не можем записать следующим образом:

```
нц для номер_строки от k + 1 до n + 1
| нц для номер_столбца от 1 до m
| | a[номер_строки, номер_столбца] := a[номер_строки - 1,
| |                                    номер_столбца]
| кц
кц
```

Понятно, почему не можем? Правильный ответ: сдвиг надо проводить, начиная с нижних строк изменяемой части массива:

```
нц для номер_строки от n + 1 до k + 1 шаг -1
| нц для номер_столбца от 1 до m
| | a[номер_строки, номер_столбца] := a[номер_строки - 1,
| |                                    номер_столбца]
| кц
кц
```

8.32. Циклическое перемещение строк массива вверх

При такой перестановке первая строка записывается на место последней, сдвинув вторую, третью и т. д. строки на одну позицию вверх[21].

Ясно, что сразу переписать элементы первой строки на соответствующее место в последней строке (`a[n,1]:=a[1,1];a[n,2]:=a[1,2],...`) нельзя — исходные значения элементов последней строки будут утеряны. Нельзя также сначала провести и сдвиг строк вверх. Нужно предварительно запомнить значение элементов первой строки во вспомогательном одномерном массиве, после чего провести сдвиг необходимых элементов вверх, а потом запомненные значения записать в последнюю строку массива:

```
|Запоминаем все элементы 1-й строки
|в массиве вспомогательный_массив
нц для номер_столбца от 1 до m
| вспомогательный_массив[номер_столбца] := a[1, номер_столбца]
кц
|Сдвигаем строки вверх
|(см. предыдущую задачу)
нц для номер_строки от 1 до n = 1
| нц для номер_столбца от 1 до m
| | a[номер_строки, номер_столбца] := a[номер_строки + 1,
| |                                    номер_столбца]
| кц
кц
|Записываем значения в последнюю строку
нц для номер_столбца от 1 до m
| a[n, номер_столбца] := вспомогательный_массив[номер_столбца]
кц
```

[21] В общем случае эта задача формулируется следующим образом: «s-ю строку массива записать на место k-й, при этом сдвинув (s + 1)-ю, (s + 2)-ю, ..., k-ю строки на одну позицию вверх (s < k)».

8.33. Циклическое перемещение строк массива вниз

При такой перестановке последняя строка записывается на место первой, при этом сдвинув первую, вторую и т. д. строку на одну позицию вниз[22]. С учетом решений двух предыдущих задач можно записать:

```
|Запоминаем все элементы последней строки
|в массиве вспомогательный_массив
нц для номер_столбца от 1 до m
| вспомогательный_массив[номер_столбца] := a[n, номер_столбца]
кц
|Сдвигаем строки вниз
|(см. предыдущую задачу)
нц для номер_строки от n до 2 шаг -1
| нц для номер_столбца от 1 до m
| | a[номер_строки, номер_столбца] := a[номер_строки - 1,
| |                                    номер_столбца]
| кц
кц
|Записываем значения в первую строку
нц для номер_столбца от 1 до m
| a[1, номер_столбца] := вспомогательный_массив[номер_столбца]
кц
```

8.34. Выяснение, имеется ли в массиве элемент, равный некоторому значению

Можно решить эту задачу, подсчитав количество элементов, равных заданному значению (см. задачу 8.3), и по найденному количеству получить нужный ответ. Однако такой вариант нерационален — если заданному числу равен, например, второй элемент первой строки, то все равно придется просматривать весь массив. Поэтому желательно прекратить проверку элементов массива, как только встретится заданное значение.

Для каждой строки двумерного массива мы с этой целью можем использовать фрагмент программы, аналогичный применявшемуся при решении задачи 8 при изучении одномерных массивов (см. соответствующий раздел):

```
есть := нет
номер_столбца := 1
нц пока не есть и номер_столбца <= m
| если a[номер_строки, номер_столбца] = значение
```

[22] В общем случае эта задача формулируется следующим образом: «*s*-ю строку массива записать на место k-й, при этом сдвинув k-ю, (k+1)-ю, ..., (s – 1)-ю строки на одну позицию вниз (s > k)».

```
| | то
| |  есть := да
| | иначе |Переходим к рассмотрению следующего элемента в строке
| |  номер_столбца := номер_столбца + 1
| все
кц
```

где `есть` — величина логического типа, принимающая истинное значение, если число `значение` встретится в массиве.

Можно также прекратить переход к следующей строке, если искомое значение уже встретилось в предыдущей. Приведем весь фрагмент программы решения данной задачи:

```
номер_строки := 1; есть := нет
нц пока не есть и номер_строки <= n
| номер_столбца := 1
| нц пока не есть и номер_столбца <= m
| | если a[номер_строки, номер_столбца] = значение
| | | то
| | |  есть := да
| | | иначе |Переходим к рассмотрению следующего элемента в строке
| | |  номер_столбца := номер_столбца + 1
| | все
| кц
| если не есть
| | то
| |  |Переходим к следующей строке
| |  номер_строки := номер_строки + 1
| все
кц
```

Ответ на поставленный в условии задачи вопрос можно получить по значению величины `есть`.

8.35. Проверка наличия в массиве одинаковых элементов

Можно подсчитать количество в массиве каждого из его элементов (см. задачу 8.3). Если одинаковых элементов в массиве нет, то сумма всех таких количеств будет равна $n \times m$ (убедитесь в этом самостоятельно). Соответствующая программа:

```
сумма_всех_количеств := 0
|Для каждого элемента массива
нц для номер_строки от 1 до n
| нц для номер_столбца от 1 до m
| | |Определяем количество таких значений
| | количество := 0
```

```
| | нц для ном_стр от 1 до n
| | | нц для ном_стол от 1 до m
| | | | |ном_стр и ном_стол - индексы
| | | | |элементов, с которыми сравнивается
| | | | |значение a[номер_строки, номер_столбца]
| | | | если a[номер_строки, номер_столбца] = a[ном_стр, ном_стол]
| | | | | то
| | | | |  количество := количество + 1
| | | | все
| | | кц
| | кц
| | |Учитываем это количество в общей сумме
| | сумма_всех_количеств := сумма_всех_количеств + количество
| кц
кц
если сумма_всех_количеств = n * m
| то
|  вывод нс, "В массиве нет одинаковых элементов"
| иначе
|  вывод нс, "В массиве есть одинаковые элементы"
все
```

Однако ясно, такой вариант нерационален: если в массиве одинаковыми являются, например, уже второй и третий элементы первой строки, то просматривать все равно приходится весь массив. Желательно прекратить проверку, как только встретятся два одинаковых элемента. Это можно сделать, используя в программе вместо операторов цикла с параметром операторы цикла с условием (соответствующий вариант программы разработайте самостоятельно).

1. В двумерном массиве хранится информация о количестве учеников в каждом классе каждой параллели школы с I по XI (в первой строке — информация о классах первой параллели, во второй — второй параллели и т. д.). В каждой параллели школы имеется по 4 класса. Определите:

   а) общее количество учащихся школы;

   б) количество учащихся в параллели V классов;

   в) в каком количестве классов численность учеников превышает 25 человек;

   г) общую численность учеников классов, в которых учатся менее 18 человек.

2. Таблица результатов футбольного чемпионата задана в виде двумерного массива из n строк и n столбцов, в котором все элементы, принадлежащие главной диагонали, равны нулю, а каждый элемент, не принадлежащий главной диагонали, равен 3, 1 или 0 (количеству очков, набранных в игре: 3 — выигрыш, 1 — ничья, 0 — проигрыш). Определите:

   а) количество ничьих у команды номер 2;

   б) количество выигрышей каждой команды;

   в) количество очков, набранных командой-чемпионом;

   г) номер команды-чемпиона (считая, что максимальное количество очков набрала только одна команда).

3. В двумерном массиве хранится информация о зарплате 18 человек за каждый месяц года (за январь — в первом столбце, за февраль — во втором и т. д.). Определите:

   а) общую сумму зарплаты, выплаченной в июне;

   б) среднюю зарплату за каждый месяц.

4. В двумерном массиве из 10 строк и 25 столбцов хранится информация о росте каждого ученика десяти классов. Рост мальчиков условно задан отрицательными числами. Если в классе менее 25 человек, то лишние элементы массива равны нулю. Определите:

   а) средний рост мальчиков в каждом классе;

   б) средний рост девочек в классе, информация о котором записана в третьей строке.

5. В двумерном массиве из 31 строки и 5 столбцов хранится информация о ночной температуре за каждый день месяца с ноября по март (для месяцев, число дней в которых меньше 31, в соответствующих элементах массива записаны нули). Составьте программы для определения средней отрицательной температуры:

   а) в заданном месяце;

   б) в каждом из представленных в массиве месяцев.

6. В двумерном массиве хранится информация о количестве студентов в той или иной группе каждого курса института с первого по пятый (в первом столбце — информация о группах первого курса, во втором — второго и т. д.). На каждом курсе имеется 10 групп. Для каждого курса определите:

   а) общее количество студентов в группах, численность которых превышает 25 человек;

   б) среднее количество студентов в таких группах.

   Составьте также программу для определения общего количества студентов в таких группах для заданного курса.

7. Фирма имеет 10 магазинов. Информация о доходе каждого магазина за каждый месяц года хранится в двумерном массиве (для первого магазина — в первой строке, для второго — во второй и т. д.). Так как не каждый магазин работал все 12 месяцев, то некоторые элементы массива равны нулю. Составьте программы:
   а) для расчета среднемесячного дохода заданного магазина (по его номеру);
   б) для определения максимального дохода за месяц для каждого магазина;
   в) для определения номера месяца, когда был получен максимальный доход (также для каждого магазина).
8. Дан двумерный массив натуральных чисел. Определите максимальное четное число в нем.
9. В двумерном массиве хранится информация о баллах, полученных спортсменами-пятиборцами в каждом из пяти видов спорта (в первом столбце — информация о баллах для первого спортсмена, во втором — для второго и т. д.). Общее количество спортсменов равно 20. Определите, в каком количестве видов спорта второй спортсмен получил более 50 баллов.
10. В зрительном зале 25 рядов, в каждом из которых 36 мест. Информация о проданных билетах хранится в двумерном массиве, номера строк которого соответствуют номерам рядов, а номера столбцов — номерам мест. Если билет на то или иное место продан, то соответствующий элемент массива равен 1, в противном случае — 0. Составьте программу для вывода на экран сведений (номера ряда и номера места) о свободных местах:
    а) во всем зале;
    б) в 10-м ряду.
11. В двумерном массиве хранится информация о зарплате каждого из 20 сотрудников четырех отделов. Составьте программу перерасчета зарплаты по правилу: если зарплата меньше 8000 руб., то она увеличивается на 20%, в противном случае — на 5%. Составьте также аналогичную программу для перерасчета зарплаты в одном только заданном отделе.
12. В поезде 18 вагонов, в каждом из которых 36 мест. Информация о проданных на поезд билетах хранится в двумерном массиве, номера столбцов которых соответствуют номерам вагонов, а номера строк — номерам мест. Если билет на то или иное место продан, то соответствующий элемент массива равен 1, в противном случае — 0. Определите:
    а) количество пассажиров в наиболее заполненном вагоне;
    б) номер одного из наиболее заполненных вагонов.

13. В двумерном массиве хранится информация о количестве осадков, выпавших за каждый день каждого месяца года. Составьте программы для определения:
    а) максимального количества осадков, выпавших за день;
    б) номера месяца и номера дня, в который это произошло.
14. В двумерном массиве хранится информация о баллах, полученных спортсменами-десятиборцами в каждом из десяти видов спорта (в первом столбце — информация о баллах для первого спортсмена, во втором — для второго и т. д.). Общее количество спортсменов равно 30. Составьте программы для определения:
    а) максимального количества баллов, полученных каждым спортсменом за каждый вид спорта;
    б) номера вида спорта, в котором каждый спортсмен набрал максимальное количество баллов;
    в) минимального количества баллов, полученных каждым спортсменом за заданный вид спорта (по его номеру);
    г) номера спортсмена, получившего минимальное количество баллов.
15. В двумерном массиве хранятся результаты (время в минутах), показанные каждым из 12 автогонщиков на каждом из 10 этапов соревнований «Формула-1» (в первой строке — результаты для первого гонщика, во второй — для второго и т. д.). После десятого этапа гонщик с порядковым номером 4 выбыл из соревнований. Измените массив, удалив из него результаты выбывшего гонщика.
16. В двумерном массиве хранятся результаты (время в минутах), показанные каждым из 16 велогонщиков на каждом из 12 этапов соревнований (в первом столбце — результаты для первого этапа, во втором — для второго и т. д.). Судейской коллегией результаты пятого этапа гонки были признаны недействительными. Измените массив, удалив из него результаты этого этапа.
17. В двумерном массиве хранится информация о зарплате каждого из 20 сотрудников отдела за 12 месяцев (для первого сотрудника — в первом столбце, для второго — во втором и т. д.). В связи с изменением фамилий сотрудниц, необходимо:
    а) поменять местами данные в 4-м и 10-м столбцах;
    б) информацию о 5-й сотруднице переместить в 15-й столбец, сдвинув при этом значения в 6-м, 7-м, … , 15-м столбцах на одну позицию влево;
    в) информацию о 18-й сотруднице переместить в 3-й столбец, сдвинув при этом значения в 3-м, 4-м, … , 17-м столбцах на одну позицию вправо.

    Составьте соответствующие программы.

18. В двумерном массиве хранится информация об оценках каждого из 25 учеников класса по 8 предметам за первую четверть (для первого ученика — в первой строке, для второго — во второй и т. д.). После окончания первой четверти в класс поступил новый ученик, фамилия которого такова, что информация о его оценках должна находиться в пятой строке. Составьте программу для внесения этих изменений в массив (информацию об оценках нового ученика считать известной).
19. В предыдущей задаче выяснилось, что по предмету под номером 5 необходимо поменять местами оценки 2-го и 4-го учеников. Разработайте программу, осуществляющую это.
20. В двумерном массиве из 5 строк и 25 столбцов хранится информация о весе каждого ученика пяти старших классов. Определите:

    а) имеется ли в этих классах ученик, вес которого равен 90 кг;
    б) имеются ли в этих классах ученики, вес которых больше 90 кг (если известно, что хотя бы один такой ученик есть).

    В обоих случаях не должен производиться полный перебор всех элементов массива.

9. Использование процедур и функций

Что такое процедура, и что такое функция? Обе они[23] представляют собой фрагменты программ, в которых решается какая-то часть общей задачи. Такой фрагмент оформляется особым образом и снабжается собственным именем.

Возникает вопрос: зачем нужно какую-то часть программы оформлять отдельно? Чтобы ответить на него, рассмотрим несколько примеров.

В программах часто встречаются повторяющиеся или похожие фрагменты — например, чтобы вывести на экран «заставку», представленную на рис. 9.1, в программе должно быть записано:

```
нц для i от 1 до 40
| вывод "*"
кц
нц для i от 1 до 5
| вывод "  "
кц
вывод "ЭТУ ПРОГРАММУ РАЗРАБОТАЛ ПЕТЯ ХАКЕРОВ"
вывод нс
нц для i от 1 до 40
| вывод "*"
кц
```

Как видим, в этой программе есть два абсолютно одинаковых фрагмента для вывода линий из звездочек. Эти фрагменты можно оформить как *процедуру*.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ЭТУ ПРОГРАММУ РАЗРАБОТАЛ ПЕТЯ ХАКЕРОВ
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Рис. 9.1

[23] В языке программирования Си оба этих понятия называются одинаково: «функция».

На школьном алгоритмическом языке общий вид процедуры следующий[24]:

```
алг <имя_процедуры>
нач <описания_переменных_величин_используемых_в_процедуре>
| <тело_процедуры
кон
```

где <описания_переменных_величин_используемых_в_процедуре> — перечень используемых в процедуре величин с указанием их типа; <тело_процедуры> — операторы, собственно реализующие решаемую с помощью процедуры задачу.

В нашей задаче процедура, с помощью которой можно получить линию из 40 звездочек, имеет вид:

```
алг Линия
нач цел i
| нц для i от 1 до 40
| | вывод "*"
| кц
кон
```

Теперь ее можно использовать в основной части программы. Вызов процедуры на выполнение осуществляется отдельным оператором, в котором указывается имя требуемой процедуры:

```
Линия
нц для i от 1 до 5
| вывод "  "
кц
вывод "ЭТУ ПРОГРАММУ РАЗРАБОТАЛ ПЕТЯ ХАКЕРОВ"
вывод нс
Линия
```

Нетрудно убедиться, что в данном случае применение процедуры уменьшает размер программы (это преимущество проявилось бы в еще большей степени, если бы в задаче требовалось рисовать линии три и более раза).

Еще один пример (хотя в значительной степени условный, но достаточно показательный): пусть необходимо на экране нарисовать из линий изображение слова «МУ». Соответствующая программа оформляется так:

```
алг МУ
нач
| ...
| сместиться на вектор(0, 100)
| сместиться на вектор(50, -50)
| сместиться на вектор(50, 50)
| сместиться на вектор(0, -100)
```

[24] С правилами оформления процедур в языке программирования, который вы изучаете, ознакомьтесь самостоятельно.

```
| поднять перо
| сместиться на вектор(10, 0)
| опустить перо
| сместиться на вектор(50, 0)
| сместиться на вектор(0, 100)
|...
кон
```

Легко ли в такой программе найти ошибочный оператор, из-за которого, например, буква «У» изображена неправильно? И где нужно добавить операторы, чтобы получить на экране вместо слова «МУ» слово «МЯУ»?

Если же рисование букв «М» и «У» оформить в виде отдельных процедур, то основная часть программы примет вид:

```
алг МУ
нач
| сместиться в точку(50, 100)
| М
| сместиться в точку(160, 100)
| У
кон
```

Конечно, при этом размер всей программы немного увеличится, но зато проявятся другие преимущества — программа станет более понятной, в ней легко найти нужное место, в том числе и связанное с ошибками! Если же нужно изобразить на экране слово «МУМУ», то применение процедур, кроме того, приведет уже и к уменьшению размера программы!

Возможность использования процедур и функций позволяет применять современные методы проектирования программ. Дело в том, что при программировании достаточно сложной задачи решить ее «одним махом», т. е. написать сразу всю программу от начала до конца, обычно невозможно. Процесс разработки программы — это творческий процесс, проходящий в несколько этапов. Вначале обычно стараются разработать наиболее общую схему алгоритма, не останавливаясь на технических деталях его реализации. В результате такой алгоритм представляется в виде последовательности относительно крупных блоков, реализующих более или менее самостоятельные смысловые части, решающих какие-то частные задачи. Эти блоки, в свою очередь, могут разбиваться на меньшие подблоки и т. д. Такой процесс последовательного структурирования программы продолжается, пока реализуемые блоками алгоритмы не станут простыми и легко программируемыми.

Рассказав о преимуществах применения процедур, перейдем к изучению их разновидностей.

Возможно использование так называемых «*процедур с параметром*» — процедур, результат выполнения которых зависит от неких величин — *параметров*. Пусть, например, в программе требуется рисовать из звездочек линии разной «длины»:

```
...
нц для i от 1 до 40
| вывод "*"
кц
...
нц для i от 1 до 20
| вывод "*"
кц
...
нц для i от 1 до 30
| вывод "*"
кц
```

Можно, конечно, для такой задачи создать три отдельные процедуры, решающие эти задачи. Но выиграем ли мы при этом в размере программы? Конечно, нет! Желательно разработать универсальную процедуру, которая умеет рисовать линии из звездочек любой длины. Это и будет процедура с параметром.

Общий вид таких процедур:

```
алг <имя_процедуры> (арг <список_формальных_параметров>)
нач <описания_переменных_величин_используемых_в_процедуре>
| <тело_процедуры>
кон
```

где <список_формальных_параметров> — это перечень величин, от которых зависит результат выполнения процедуры (с указанием их типа). Эти величины называют «формальными параметрами процедуры», и именно они используются в теле процедуры.

В нашем случае процедура, с помощью которой можно получить линию любой длины, имеет вид:

```
алг Линия(арг цел k)
нач цел i
| нц для i от 1 до k
| | вывод "*"
| кц
кон
```

Результат ее выполнения зависит от величины k, которая является единственным параметром этой процедуры.

Вызов такой процедуры осуществляется командой:

```
<имя_процедуры> (<список_фактических_параметров>)
```

где <список_фактических_параметров> — это перечень конкретных значений параметров процедуры для решения конкретной задачи. В нашем случае, чтобы использовать процедуру для рисования линии из 30 звездочек, вызов процедуры должен быть оформлен командой: `Линия(30)`.

В качестве значений фактических параметров можно указывать[25]:

1) константы (заданные, заранее известные значения) — именно так было сделано в приведенном выше примере;
2) имена величин, например `Линия(длина1)`; как видно из этого примера, имена фактических и формальных параметров не обязательно должны совпадать;
3) выражения `(Линия(длина1 + 10))`.

В каждом из этих трех случаев тип фактического параметра обязательно должен совпадать с типом формального параметра, указанным в заголовке процедуры.

Ясно, что в процедуре могут быть использованы два или более формальных параметра. В таких случаях при оформлении списка фактических параметров следует учитывать следующие требования:

1) количество фактических параметров должно быть таким же, как и в списке формальных параметров в описании процедуры;
2) тип каждого фактического параметра должен совпадать с типом соответствующего формального параметра;
3) соответствующие фактические и формальные параметры должны совпадать по смыслу.

Если первые два требования понятны, то третье требует дополнительных объяснений. Пусть, например, подготовлена процедура, изображающая на экране прямоугольник:

```
алг Линия(арг цел a, b)
нач
| ...
кон
```

Как с ее помощью нарисовать прямоугольник шириной 20 и высотой 100 пикселей? Какой из вариантов оформления вызова процедуры является правильным: `Линия(20, 100)` или `Линия(100, 20)`? Ответ зависит от того, какому размеру соответствует формальный параметр `a`, а какому — `b`. Если `a` — это высота прямоугольника, то правильный вариант — `Линия(100, 20)`; в противном случае первой должна быть указана ширина изображаемого прямоугольника, а второй — высота: `Линия(20, 100)`.

[25] Приведенный здесь перечень далее будет уточнен.

Рассмотрим характерный пример. Пусть в программе необходимо часто обменивать местами значения двух переменных. Для решения такой задачи составим[26] процедуру Обмен:

```
алг Обмен(арг цел a, b)
нач цел c
| c := a
| a := b
| b := c
кон
```

Используем эту процедуру для переменных x и y:

```
алг Основная_программа
нач цел x, y
| |Исходные значения
| x := 10
| y := 100
| |Вызываем процедуру Обмен
| Обмен(x, y)
| |Выводим на экран новые значения
| вывод нс, "x=", x, "y=", y
кон
```

Выполнив программу, обнаружим, что значения переменных не изменились! Почему? Ведь вызывалась и работала процедура Обмен, в которой происходили изменения значений параметров. А все дело в том, что в школьном алгоритмическом языке, в языке Паскаль и ряде других возможны два вида формальных параметров:

1) формальные *параметры-значения*;
2) формальные *параметры-переменные*.

Особенностью первых является то, что все действия над ними внутри процедуры никак не отражаются на соответствующих фактических параметрах, т. е. какими фактические параметры были до вызова процедуры, такими они и останутся после завершения ее работы. Именно поэтому значения фактических параметров x и y у нас не изменились — они были описаны как параметры-значения (со служебным словом арг). Чтобы в результате выполнения процедуры значения фактических параметров изменились согласно проведенным в процедуре действиям, формальные параметры должны быть описаны как параметры-переменные. В школьном алгоритмическом языке это делается путем сочетания служебных слов арг и рез в заголовке процедуры:

```
алг Обмен(арг рез цел a, b)
...
```

а в языке Паскаль — указанием в заголовке процедуры служебного слова **Var**:

```
Procedure Swap(Var a, b: integer);
...
```

[26] В языке программирования Бейсик для обмена значениями двух переменных имеется стандартная процедура SWAP.

Конечно, в описании процедуры можно и сочетать оба вида формальных параметров.

Примечание. При использовании формальных параметров-переменных в списке фактических параметров им может соответствовать только имя величины (константы и выражения в этом случае записывать нельзя).

Наглядной иллюстрацией механизма передачи значений в процедуру и из нее с формальными параметрами-переменными является следующий пример:

```
алг Обмен_местами_двух_элементов_массива
нач цел i, цел таб m[1:10]
| нц для i от 1 до n
| | m[i] := rnd(100)
| | вывод m[i], "  "
| кц
| Обмен(m[1], m[10])
| нц для i от 1 до n
| | вывод m[i], "  "
| кц
кон
```

Здесь в процедуре `Обмен` производится обмен значениями двух ее параметров, которые являются значениями элементов массива, а в результате изменяется и сам исходный массив.

Теперь поговорим об особенностях использования процедур в языке программирования Бейсик. В нем все формальные параметры процедур, как говорится, «по умолчанию» считаются параметрами-переменными. При этом использование в качестве фактических параметров констант и выражений ошибкой не считается (результат выполнения процедур при таком оформлении их вызова определите самостоятельно).

Прежде чем идти дальше, заметим, что переменные, описанные в процедуре, называют «*локальными*», а описанные в основной части программы — «*глобальными*». Локальные переменные существуют только во время выполнения процедуры и поэтому недоступны в основной части программы и в других процедурах. Например, при выполнении следующих четырех программ появится сообщение об ошибке, связанное с этим обстоятельством:

1)
```
алг Основная_программа[27]
нач
| вывод a
кон
алг Процедура1
нач цел a
| a := 100
кон
```

[27] В школьном алгоритмическом языке основная часть программы размещается раньше всех вспомогательных процедур.

```
2) алг Основная_программа
   нач
   | Процедура1
   | вывод a
   кон
   алг Процедура1
   нач цел a
   | a := 100
   кон

3) алг Основная_программа
   нач
   | Процедура1
   | Процедура2
   кон
   алг Процедура1
   нач цел a
   | вывод a
   кон
   алг Процедура2
   нач цел a
   | a := 100
   кон

4) алг Основная_программа
   нач
   | Процедура2
   | Процедура1
   кон
   алг Процедура2
   нач цел a
   | a := 100
   кон
   алг Процедура1
   нач цел a
   | вывод a
   кон
```

В то же время глобальные переменные доступны («видны») в процедурах:

```
алг Основная_программа
нач цел a
| a := 100
| Процедура1
кон
алг Процедура1
нач
| вывод a
кон
```

Еще один случай, который нужно здесь рассмотреть, — это совпадение имен локальной и глобальной переменных. Что, по вашему мнению, будет выведено на экран в результате выполнения следующей программы?

```
алг Основная_программа
нач цел a
| a := 100
| Процедура1
кон
алг Процедура1
нач цел a
| a := 1
| вывод a
кон
```

Правильный ответ — 1 (говорят, что в этом случае локальная переменная «закрывает» собой глобальную)[28].

Теперь — о том, что такое *функция*. Вы, конечно, знаете о так называемых «стандартных» (имеющихся в языке программирования) функциях (таких как `sin`, `sqr` и др.). Что они делают при их использовании в программах? Возвращают какое-то одиночное значение (результат расчетов и т. д.). Так вот, с такой же целью можно создавать и использовать собственные функции!

Пусть, например, в программе надо рассчитывать значения `x` по формуле:

$$x = \frac{2+\sqrt{2}}{5+\sqrt{5}} + \frac{5+\sqrt{5}}{13+\sqrt{13}} + \frac{11+\sqrt{11}}{8+\sqrt{8}}.$$

Видно, что в этом выражении имеются похожие фрагменты. Поэтому желательно для расчета дробей оформить функцию, которая вычисляла бы значения вида $\frac{a+\sqrt{a}}{b+\sqrt{b}}$ при любых значениях `a` и `b`.

Правила оформления функций в разных языках программирования различны. В школьном алгоритмическом языке общий вид функций такой:

```
алг <тип_результата> <имя_функции>(арг <список_
     формальных_параметров>)
нач <описания_переменных_величин_используемых_в_функции>
| <тело_функции>
кон
```

причем последним оператором тела функции здесь должен быть оператор присваивания, в левой части которого должно быть указано служебное слово `знач`.

Нетрудно видеть, что в этом языке отличие функции от процедуры заключается в наличии в заголовке функции типа результата, кото-

[28] Следует отметить интересный (и полезный) факт: в программе на языке Паскаль при отсутствии в последней процедуре оператора `a := 1` результат выполнения программы будет непредсказуемым — на экран может быть выведено любое число. Дело в том, что начальное присваивание нулевого значения локальным переменным процедур и функций в этом языке не производится! (См. также задачи 4.1–4.4, 6.1–6.4, 8.1–8.4, 8.11–8.14, 8.20–8.23.)

рый она возвращает. В других языках программирования, как правило, в описании функции указывается служебное слово Function.

В нашем случае функция для расчета значения дробей указанного выше вида оформляется так:

```
алг вещ Дробь(арг цел a, b)
нач цел числитель, знаменатель
| числитель := a + sqrt(a)
| знаменатель := b + sqrt(b)
| |Значение функции:
| знач := числитель / знаменатель
кон
```

Как и стандартные, функция, созданная программистом, используется в правой части оператора присваивания. Для этого надо указать ее имя, а в скобках — значения фактических параметров:

```
x := Дробь(2, 5) + Дробь(5, 13) + Дробь(13, 8)
```

Требования к оформлению списка фактических параметров те же, что и для процедур с формальными параметрами-значениями.

Результат, возвращаемый функцией, может быть любого типа, в том числе логического[29]. Например, функция, определяющая, является ли натуральное число a делителем натурального числа b, имеет вид:

```
алг лог Делитель(арг цел a, b)
нач цел
| |Значение функции:
| знач := mod(b, a) = 0
кон
```

Эта функция затем может быть использована в программе для решения задачи: *Дано натуральное число* a. *Подсчитать количество делителей этого числа*:

```
алг Количество_делителей
нач цел a, возможный_делитель, количество
| вывод нс, "Задайте число a"
| ввод a
| количество := 0
| нц для возможный_делитель от 1 до div(a, 2)
| | если Делитель(возможный_делитель, a)
| | | то
| | |  количество := количество + 1
| | все
| кц
| вывод нс, "Количество делителей числа a "
| вывод "равно ", количество
кон
```

[29] Если, конечно, такой тип предусмотрен в конкретном языке программирования.

Возникают закономерные вопросы: какая существует связь между процедурами и функциями? Можно ли вместо одной из них использовать другую? Чтобы ответить на них, надо вспомнить процедуры с формальными параметрами-переменными. Они, как и функции, возвращают в вызывающую их программу какие-то значения. Функция же всегда возвращает единственное значение. Это означает, что вместо функции всегда можно использовать соответствующим образом оформленную процедуру. О том, целесообразно ли это делать, можно судить, например, по следующей процедуре, созданной для расчета значения дроби в рассмотренной выше задаче:

```
алг Дробь(арг цел a, b, арг рез вещ частное)
нач цел числитель, знаменатель
| числитель := a + sqrt(a)
| знаменатель := b + sqrt(b)
| частное := числитель / знаменатель
кон
```

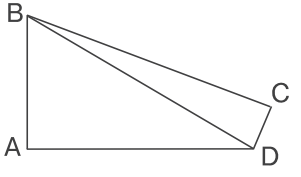
Использование процедуры `Дробь`:

```
алг Основная_программа
нач вещ слагаемое1, слагаемое2, слагаемое3, x
| слагаемое1 := 0
| |Вызываем процедуру Дробь
| |для расчета первого слагаемого
| Дробь(2, 5, слагаемое1)
| слагаемое2 := 0
| |Вызываем процедуру Дробь
| |для расчета второго слагаемого
| Дробь(5, 13, слагаемое2)
| слагаемое3 := 0
| |Вызываем процедуру Дробь
| |для расчета третьего слагаемого
| Дробь(11, 8, слагаемое3)
| |Рассчитываем значение x
| x := слагаемое1 + слагаемое2 + слагаемое3
| ...
кон
```

Вместо процедуры, возвращающей только один результат, также можно (и даже, как следует из приведенного примера, чаще всего целесообразно) оформить функцию. Однако ясно, что если количество возвращаемых процедурой значений больше одного, то применять функцию нельзя.

В заключение отметим, что процедура или функция может в процессе своей работы использовать любую другую процедуру (функцию) или даже саму себя. В последнем случае процедура (функция) называется *рекурсивной*. Рекурсивным процедурам и функциям будет посвящен следующий раздел.

1. Составьте процедуру, выводящую по периметру экрана прямоугольник из «звездочек» (*).
2. Напишите процедуру, выводящую на экран строку из любого количества копий заданного символа.
3. Составьте процедуру, осуществляющую обмен значениями для трех переменных a, b и c по следующей схеме: b присвоить значение a, c присвоить значение b, a присвоить значение c.
4. Даны стороны двух треугольников. Найдите сумму их периметров и сумму их площадей. (Напишите процедуру для расчета периметра и площади треугольника по его сторонам.)
5. Рассчитайте значение x, определив и использовав функцию:
$$x = \frac{\sqrt{6}+6}{2} + \frac{\sqrt{13}+13}{2} + \frac{\sqrt{21}+21}{2}.$$
6. Определите значение z = sign x + sign y, где
$$\text{sign } a = \begin{cases} -1 \text{ при } a < 0, \\ 0 \text{ при } a = 0, \\ 1 \text{ при } a > 0. \end{cases}$$
Значения x и y вводятся с клавиатуры. (При решении задачи требуется определить и использовать функцию sign.)
7. Найдите периметр фигуры **ABCD** по заданным сторонам **AB**, **AD** и **CD**. (Определите функцию для расчета гипотенузы прямоугольного треугольника по его катетам.)



∠BAD = 90°, ∠BCD = 90°

8. Найдите периметр треугольника, заданного координатами его вершин. (Определите функцию для расчета длины отрезка по координатам его вершин.)
9. Получите все шестизначные «счастливые» номера. «Счастливым» называют такое шестизначное число, для которого сумма первых трех цифр равна сумме последних трех цифр. (Определите функцию для расчета суммы цифр трехзначного числа.)
10. Даны два предложения. В каком из них встречаемость (в процентах) буквы «б» больше. (Определите функцию для расчета встречаемости заданной буквы в предложении.)

10. Рекурсия

Рекурсией (от латинского «recursio» — «возвращение») называют ситуацию в программе, когда процедура или функция вызывает в качестве вспомогательной саму себя. Непонятно? Тогда вспомните функцию Дробь, созданную нами в предыдущем разделе:

```
алг вещ Дробь(арг цел a, b)
нач цел числитель, знаменатель
| числитель := a + sqrt(a)
| знаменатель := b + sqrt(b)
| |Значение функции:
| знач := числитель / знаменатель
кон
```

В ней используется стандартная (имеющаяся в языке программирования) функция sqrt, рассчитывающая квадратный корень из своего аргумента.

Так вот, при оформлении собственной функции (или процедуры) в качестве вспомогательной может быть вызвана сама эта функция (или процедура). Это и есть рекурсия. Для иллюстрации создадим функцию с рекурсией (такие функции называются *рекурсивными*) для расчета факториала натурального числа n (факториал числа n, обозначаемый как n!, равен произведению чисел $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$). Такую функцию можно создать, зная, что $n! = (n-1)! \cdot n$:

```
алг цел Факториал(арг цел n)
нач
| знач := Факториал(n - 1) * n |Рекурсивный вызов
|                              |функции Факториал
кон
```

На самом деле эта функция оформлена не по правилам (и поэтому при выполнении такой программы появится сообщение об ошибке), но об этом — чуть позже. Здесь же мы обсудим закономерный вопрос, который уже, наверное, у вас появился: зачем нужно было «го-

родить огород», когда можно было создать такую естественную и понятную функцию:

```
алг цел Факториал(арг цел n)
нач цел произведение, i
| произведение := 1
| нц для i от 2 до n
| | произведение := произведение * i
| кц
| знач := произведение
кон
```

На этот вопрос в данном случае трудно ответить убедительно. Но давайте рассмотрим еще одну задачу: *Требуется разработать функцию для расчета k-го члена последовательности Фибоначчи (последовательность Фибоначчи образуют числа 1, 1, 2, 3, 5, 8, 13, …).*

Для начала попробуйте создать такую функцию самостоятельно, не используя рекурсию. (Если не получится, то воспользуйтесь приведенным ниже ответом и … зеркалом.)

```
|алг цел Фиб (арг цел k)
|нач цел очер, пред, предпред, i
||пред := 1
||предпред := 1
||нц для i от 2 до k = 2
|||очер := пред + предпред
|||предпред := пред
|||пред := очер
||кц
||знач := очер
|кон
```

Использованы следующие величины: очер — очередной рассчитываемый элемент последовательности; пред — элемент, предшествующий очередному элементу; предпред — элемент, предшествующий элементу пред

Быстро ли вы нашли решение? А теперь посмотрите, как просто и логично выглядит рекурсивный вариант этой функции[30]:

```
алг цел Фиб(арг цел k)
нач
| знач:=Фиб(k - 2) + Фиб(k - 1)
кон
```

[30] Он тоже пока оформлен не по правилам, но отражает основную суть идеи.

Такая запись полностью соответствует закону построения последовательности Фибоначчи — очередной ее элемент равен сумме двух предыдущих. При этом не требуется применять оператор цикла и думать над последовательностью расчета значений `пред` и `предпред`.

А если и этот пример не убедил вас в преимуществе использования рекурсии (конечно, в определенных случаях), то рассмотрим еще одну задачу: *Требуется получить на экране изображение, показанное на рис. 10.1.*

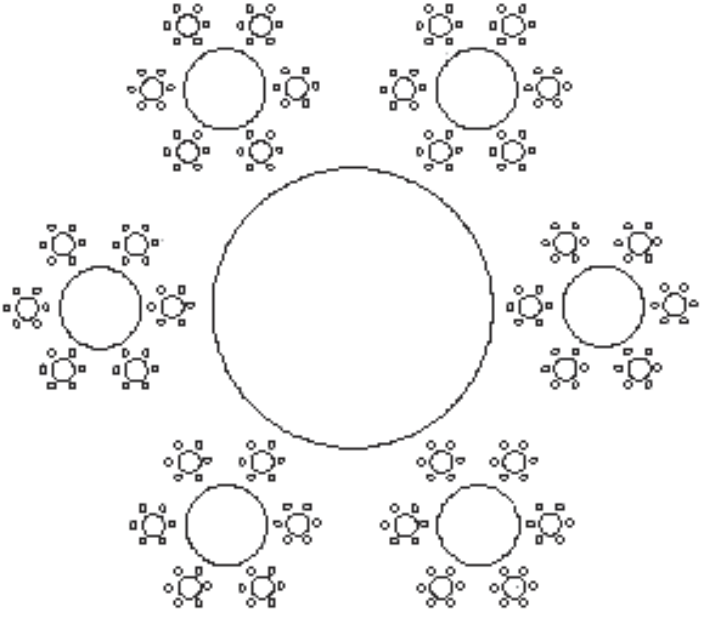


Рис. 10.1

Ну как, возьметесь ли вы решить эту задачу без рекурсии? Если нет, то давайте обсудим рекурсивный вариант.

Будем считать, что на рис. 10.1 изображена центральная планета с шестью спутниками, у каждого из которых есть свои спутники, у тех — свои и т. д. Очевидно, что каждый спутник может рассматриваться нами как планета с соответствующими спутниками. Поэтому если составить процедуру, с помощью которой можно изобразить на экране некоторую окружность-планету с несколькими окружностями-спутниками, а для рисования спутников использовать эту же процедуру, но с другими параметрами — координатами, радиусами и т. п. (т. е. применить рекурсию), то можно получить требуемое изображение.

Соответствующая рекурсивная процедура имеет вид:

```
алг Планета(арг цел x, y, рад, Nсп, вещ к_орб, к_спут)
| x, y – координаты центра планеты, рад – ее радиус
| Nсп – число спутников у каждой планеты
| к_спут – отношение радиуса спутника к радиусу "своей" планеты
| к_орб – то же для радиуса орбиты спутников
нач цел x1, y1, i, вещ угол, рад_орб
| поз(x, y)                 |Центр планеты
| окружность(рад)           |Рисуем планету
| рад_орб := рад * к_орб    |Радиус орбиты спутников
| угол := 6.28 / Nсп        |Угол между спутниками
| нц для i от 1 до Nсп      |Для каждого спутника
| | x1 := x + рад_орб * cos(угол * i) |Координаты центра
| | y1 := x + рад_орб * sin(угол * i) |i-го спутника
| | |Вызываем процедуру Планета с новыми параметрами
| | Планета(x1, y1, int(рад *к_спут), Nсп, к_орб, к_спут)
| кц
кон
```

Однако и эта процедура пока оформлена неправильно! Теперь настало время сказать почему. Вы, конечно, знаете, что при каждом вызове вспомогательной процедуры (функции) для нее отводится место в оперативной памяти, которое освобождается после завершения работы процедуры. Но если во вспомогательной процедуре имеется рекурсия, то вызовы вспомогательных процедур будут продолжаться, пока место в памяти не будет исчерпано (именно с этим связано сообщение об ошибке, которое было бы выдано при запуске рекурсивной функции Факториал).

Чтобы устранить этот недостаток, необходимо так оформлять процедуры (функции), чтобы рекурсивные вызовы осуществлялись по условию, которое в какой-то момент станет ложным. В приведенной выше процедуре для этого можно использовать в качестве аргумента некоторую величину n, которая при каждом новом вызове будет уменьшаться на 1, а в тело процедуры включить условие, что ее команды должны выполняться только при n > 0:

```
алг Планета(арг цел x, y, рад, n, Nсп, вещ к_орб, к_спут)
нач
| если n>0
| | то
| |  поз(x, y)
| |  ...
| |  Планета(x1, y1, int(рад*к_спут), n – 1, Nсп, к_орб, к_спут)
| все
кон
```

Другой способ ограничения количества «вложенных» рекурсивных вызовов — по достижению радиусом «основной» окружности некоторого минимально возможного значения.

Приведем также «правильные» варианты ранее рассмотренных функций `Факториал` и `Фиб`:

```
алг цел Факториал(арг цел n)
нач
| если n>1
| | то
| |  |Рекурсивный вызов функции Факториал
| |  знач := Факториал(n - 1) * n
| | иначе
| |  знач := 1
| все
кон
```

и

```
алг цел Фиб(арг цел k)
нач
| если k > 2
| | то
| |  |Рекурсивный вызов функции Фиб
| |  знач:=Фиб(k - 2) + Фиб(k - 1)
| | иначе
| |  знач := 1
| все
кон
```

Вернемся теперь к рисованию изображения на рис. 10.1 и напишем основную часть программы решения этой задачи:

```
алг Система_планет
нач цел n, Nсп, рад, x, y, вещ к_орб, к_спут
| видео(18)        |Установка графического режима
| n := 7; Nсп := 6
| x := максХ / 2    |Координаты центра
| y := максУ / 2    |планеты
| рад := 120; к_орб := 1.6; к_спут := 0.5
| Планета(x, y, рад, n, Nсп, к_орб, к_спут)
кон
```

Кстати, с помощью этой программы, задавая различные значения исходных величин (радиуса, количества спутников и др.), можно получать красивые картинки.

Удивить своих товарищей вы сможете также, получив на экране другие подобные рисунки (см. рис. 10.2 и 10.3; такие изображения называют «фрактальными»).

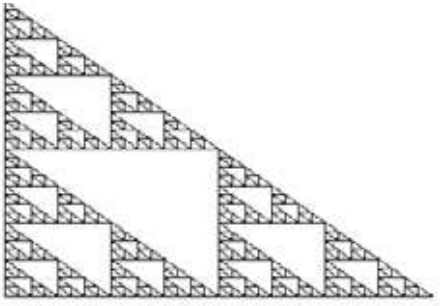


Рис. 10.2

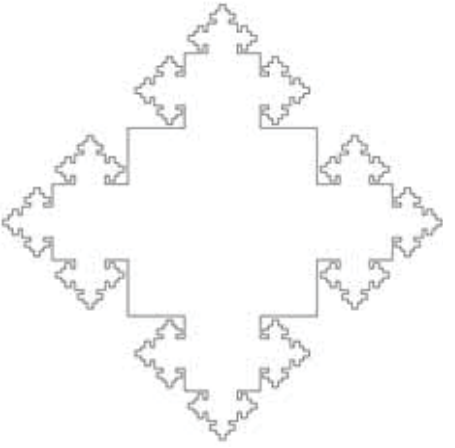


Рис. 10.3

Картинка на рис. 10.2 получена следующим способом. В треугольнике проводятся все средние линии; тем самым он разбивается на четыре треугольника. К трем из них, примыкающим к вершинам первоначального треугольника, применяются те же действия, и т. д. Рекурсивная процедура Треугольник, в которой все это реализуется, имеет вид:

```
алг Треугольник(арг цел xa, ya, xb, yb, xc, yc, n)
нач цел xp, yp, xq, yq, xr, yr
| если n > 0
| | то
| | |Координаты середин сторон треугольника
| | xp := (xb + xc) / 2; yp := (yb + yc) / 2
| | xq := (xa + xc) / 2; yq := (ya + yc) / 2
| | xp := (xa + xb) / 2; yr := (ya + yb) / 2
| | поз(xp, yp)
| | |Рисуем средние линии
| | линия(xq, yq)
| | линия(xr, yr)
| | линия(xp, yp)
| | |Рекурсивно вызываем процедуру Треугольник
| | Треугольник(xa, ya, xr, yr, xq, yq, n - 1) |для каждой
| | Треугольник(xb, yb, xp, yp, xr, yr, n - 1) |из трех
| | Треугольник(xc, yc, xq, yq, xp, yp, n - 1) |сторон
| все
кон
```

А вот и основная часть программы:

```
алг Множество_треугольников
нач цел xa, ya, xb, yb, xc, yc
| видео(18)
| | Координаты вершин самого большого треугольника
| xc := 0; yc := 0
| xb := максX; yb := максY
| xa := 0; ya := максY
| поз(xa, ya)
| |Рисуем самый большой треугольник
| линия(xb, yb)
| линия(xc, yc)
| линия(xa, ya)
| |Начинаем рисовать внутренние треугольники
| Треугольник(xa, ya, xb, yb, xc, yc, 6)
кон
```

Изображение же на рис. 10.3 получено иначе. На каждой из сторон внутреннего (самого большого) квадрата нарисованы 3 стороны малого квадрата, на каждой из сторон которого также изображены 3 стороны еще меньшего квадрата и т. д. Процедура, которая выполняет соответствующие действия на некотором отрезке с координатами концов `xa`, `ya`, `xb`, `yb`, **может быть оформлена следующим образом:**

```
алг Сторона(арг цел xa, ya, xb, yb, n, вещ k)
нач цел xp, yp, xq, yq, xr, yr, xs, ys, dx, dy
| если n = 0
| | то
| |   линия(xb, yb)
| | иначе
| |   dx := 0.5 * (1 - k) * (xb - xa) |k — коэф. уменьшения
| |   dy := 0.5 * (1 - k) * (yb - ya) |размера квадратов
| |   |Координаты фигуры, изображаемой на отрезке AB
| |   xp := xa + dx; yp := ya + dy
| |   xs := xb-dx; ys := yb - dy
| |   xq := xp + (ys - yp); yq := yp - (xs - xp)
| |   xr := xq + (xs - xp); yr := yq +(ys - yp)
| |   линия(xp, yp) |К началу малого квадрата
| |   Сторона(xp, yp, xq, yq, n - 1, k)
| |   Сторона(xq, yq, xr, yr, n - 1, k)
| |   Сторона(xr, yr, xs, ys, n - 1, k)
| |   линия(xb, yb) |К концу отрезка
| все
кон
```

Основная часть программы:

```
алг Картинка
нач цел n, xc, yc, b, вещ k
| видео (18)
| xc := максX / 2; yc := максY / 2
| b := 100  |Половина длины стороны базового квадрата
| n := 5; k := 0.4
| поз(xc - b, yc - b) |Точка, с которой начинается рисунок
| Сторона(xc – b, yc – b, xc + b, yc – b, n, k)
| Сторона(xc + b, yc – b, xc + b, yc + b, n, k)
| Сторона(xc + b, yc + b, xc – b, yc + b, n, k)
| Сторона(xc – b, yc + b, xc – b, yc – b, n, k)
кон
```

Когда вы напишете аналогичные программы на известном вам языке программирования, то получите, я уверен, истинное удовольствие. Красиво получается, не правда ли? Без рекурсии так, наверное, не сделаешь!..

Сложно (но можно — см., например, [4]) без рекурсии решить и известную задачу «Ханойские башни», где речь идет о перекладывании дисков с одного стержня на другой по определенным правилам.

Одним из самым ярких примеров использования рекурсии является метод сортировки массивов, разработанный в 1962 г. в Англии профессором Оксфордского университета Ч. Хоаром (C. Hoare). Этот метод, считающийся самым быстрым из всех известных, основан на рекурсии; о нем будет рассказано в разделе 11.

А сейчас давайте рассмотрим различные формы рекурсивных процедур.

В общем случае любая рекурсивная процедура **Р** включает в себя некоторое множество операторов **Д** и один или несколько операторов рекурсивного вызова **Р**. Как уже отмечалось, главное требование к рекурсивным процедурам и функциям заключается в том, что рекурсивный вызов должен осуществляться по условию, которое в какой-то момент станет ложным.

Структура рекурсивных процедур может принимать три разные формы:

1) с выполнением действий после рекурсивного вызова (или с выполнением действий на рекурсивном возврате):

```
алг Р                    или      алг Р
  нач                               нач
  | если <условие>                  | если <условие>
  | | то                            | | то
  | |   Р                           | |   Р
  | все                             | |  Д
  | Д                               | все
кон                                 кон
```

Примеры такой процедуры:

```
алг ВыводЧисла(арг цел n)
  нач
  | если n > 0
  | | то
  | |  ВыводЧисла(n - 1)
  | все
  | вывод n
кон
```

или

```
алг ВыводЧисла(арг цел n)
нач
| если n > 0
| | то
| |  ВыводЧисла(n - 1)
| |  вывод n
| все
кон
```

Нетрудно предсказать результат ее выполнения, например, при n = 5: 1 2 3 4 5;

2) с выполнением действий до рекурсивного вызова (или с выполнением действий на рекурсивном спуске):

```
алг Р
  нач
  | Д
  | если<условие>
  | | то
  | |  Р
  | все
кон
```

или

```
алг Р
нач
| если<условие>
| | то
| |  Д
| |  Р
| все
кон
```

Примеры:

```
алг ВыводЧисла(арг цел n)
  нач
  | вывод n
  | если n > 1
  | | то
  | |  ВыводЧисла(n - 1)
  | все
  кон
```

или

```
алг ВыводЧисла(арг цел n)
нач
| если n > 0
| | то
| |  вывод n
| |  ВыводЧисла(n - 1)
| все
кон
```

Результат (при n = 5): 5 4 3 2 1;

3) с выполнением действий как до, так и после рекурсивного вызова (или с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате):

```
алг Р
  нач
  | Д
  | если <условие>
  | | то
  | |  Р
  | все
  | Д
кон
```

или

```
алг Р
нач
| если <условие>
| | то
| |  Д
| |  Р
| |  Д
| все
кон
```

Примеры:

```
алг ВыводЧисла(арг цел n)
  нач
  | вывод n
  | если n > 1
  | | то
  | | ВыводЧисла(n - 1)
  | все
  | вывод n
кон
```

или

```
алг ВыводЧисла(арг цел n)
нач
| если n > 0
| | то
| |  вывод n
| |  ВыводЧисла(n - 1)
| |  вывод n
| все
кон
```

Результат: 5 4 3 2 1 1 2 3 4 5.

Многие задачи безразличны к тому, какая форма рекурсивной процедуры для них используется. Однако есть классы задач, при решении которых программисту требуется сознательно управлять ходом работы рекурсивных процедур и функций. Такими, в частности, являются задачи, использующие списки и древовидные структуры данных. Например, при разработке трансляторов широко применяются так называемые «атрибутированные деревья разбора», работа с которыми требует от программиста умения направлять ход рекурсии: одни действия можно выполнять только на спуске, другие — только на возврате. Поэтому глубокое понимание рекурсивного механизма и умение управлять им по собственному желанию является необходимым качеством квалифицированного программиста.

Существует также разновидность рекурсии, которую называют *косвенной*, или *непрямой*. Такой рекурсией является ситуация, когда процедура **А** вызывает себя в качестве вспомогательной не непосредственно, а через другую вспомогательную процедуру **Б**.

Образно косвенную рекурсию можно описать так. Перед зеркалом 1 стоит зеркало 2. Что видно в зеркале 1? Зеркало 2, в котором отражается само зеркало 1. В последнем видно зеркало 2 и т. д. (рис. 10.4).

Рис. 10.4

В качестве примера косвенной рекурсии можно привести несколько измененную программу рисования изображения на рис. 10.1:

```
алг Планета(арг цел x, y, рад, n, Nсп, вещ к_орб, к_спут)
нач
| если n > 0
| | то
| |  поз(x, y) |Центр планеты
| |  окружность(рад) |Рисуем планету
| |  Спутники(x, y, рад, n, Nсп, к_орб, к_спут)
| |  |Спутники – вспомогательный алгоритм
| все
кон
```

```
алг Спутники(арг цел x, y, рад, n, Nсп, вещ к_орб, к_спут)
нач цел x1, y1, i, вещ угол, рад_орб
| рад_орб := рад * к_орб |Радиус орбиты спутников
| угол := 6.28 / Nсп |Угол между спутниками
| нц для i от 1 до Nсп |Для каждого спутника
| | x1 := x + рад_орб * cos(угол * i) |Координаты центра
| | y1 := x + рад_орб * sin(угол * i) |i-го спутника
| | |Рекурсивный вызов процедуры Планета
| | Планета(x, y, int(рад * к_спут), Nсп - 1, к_орб, к_спут)
| кц
кон
```

Основная часть программы:

```
алг Система планет
нач цел n, Nсп, рад, x, y, вещ к_орб, к_спут
| видео(18) |Установка графического режима
| n := 7; Nсп := 6
| x := максX / 2 |Координаты центра
| x := максX / 2 |планеты
| рад := 120; к_орб := 1.6; к_спут := 0.5
| Планета(x, y, рад, n, Nсп, к_орб, к_спут)
кон
```

Процедура Планета здесь выводит на экран центральную окружность-планету, а для рисования ее спутников используется вспомогательная процедура Спутники. Последняя рисует спутники с помощью рекурсивного вызова процедуры Планета. Не кажется ли вам, читатель, что в таком виде программа даже более логична и понятна, чем с «обычной» рекурсией?

Примечание. При написании аналогичной программы на языке Паскаль возникает проблема, связанная с тем, что процедура Планета использует в качестве вспомогательной еще не описанную процедуру

Спутники, а это в Паскале недопустимо. Причем процедуру Спутники тоже нельзя описать первой, поскольку в ней имеется вызов процедуры Планета. Выходом здесь является применение так называемого «опережающего» описания процедур.

1. Составьте программу, в результате выполнения которой на экран 10 раз выводится текст всем известной истории о попе и его собаке («У попа была собака ...»).

   (Указания по выполнению. Чтобы наблюдать вывод каждой новой истории, предусмотрите в процедуре небольшую паузу в выполнении программы. Как это сделать, рассказано в разделе 3.)

2. Составьте программы для расчета:

   1) значения a^n для заданных значений a и n (a — вещественное число, n — натуральное);
   2) n-го члена арифметической прогрессии;
   3) n-го члена геометрической прогрессии;
   4) суммы n первых членов арифметической прогрессии;
   5) суммы n первых членов геометрической прогрессии;
   6) минимального элемента массива;
   7) индекса минимального элемента массива.

3. Составьте программу, в которой вводятся, а затем выводятся на экран в обратном порядке 10 заданных чисел. Каждое из заданных чисел должно вводиться один раз. (Массивы не применять!)
4. Дано натуральное число n. Выведите все его делители в порядке их возрастания. (Проверку возможных делителей числа n проводите только до значения, не превышающего $\sqrt{n}$; все остальные делители получаются в результате деления числа n на найденные делители.)

   Все указанные задачи следует решить с использованием рекурсивных функций и процедур.

---

11. Методы сортировки числовых массивов

Сортировкой называют процесс размещения заданного множества объектов в определенном порядке (в частности, множества чисел — в порядке их возрастания или убывания).

Зачем нужна сортировка? Очевидно, что с отсортированными данными в ряде случаев работать легче, чем с произвольно расположенными. Когда элементы отсортированы, их проще найти (как, например, в телефонном справочнике, в словаре, на панелях программы Windows Commander или FAR Manager). На отсортированных данных легче определить, имеются ли пропущенные элементы, и удостовериться, что все элементы были проверены. Легче найти общие элементы двух множеств, если оба этих множества отсортированы. Сортировка является мощным средством ускорения работы практически любого алгоритма, в котором нужно часто обращаться к определенным элементам данных.

Но, как пишет один из классиков теории программирования Д. Кнут, «даже если бы сортировка была почти бесполезна, нашлась бы масса причин заняться ею! Изобретательные методы сортировки говорят о том, что она и сама по себе интересна как объект исследования» [5]. Знакомство с этими методами очень полезно при изучении программирования, так как с их помощью можно наглядно проиллюстрировать многие принципы и ситуации, возникающие и в других задачах. Так что если вы хотите стать хорошим программистом, то обязательно должны знать различные методы сортировки.

В настоящее время известно множество методов сортировки, из которых мы рассмотрим восемь.

Сначала приведем четыре простых метода:

1) сортировка подсчетом;
2) сортировка вставками;
3) сортировка выбором;
4) сортировка обменом.

Затем рассмотрим более сложные, но и более эффективные[31] методы:

5) сортировка вставками с убывающим шагом (метод Шелла);
6) сортировка с разделением (быстрая сортировка);
7) сортировка слияниями;
8) пирамидальная сортировка.

[31] Эффективность метода также зависит от количества элементов массива `n`; при достаточно малых значениях `n` простые методы часто работают быстрее.

Каждому из представленных методов сортировки будет посвящен особый подраздел. Структура всех этих подразделов одинакова: сначала излагается сущность метода, затем приводится процедура[32], соответствующая данному методу, а также некоторые другие вспомогательные процедуры и функции, необходимые для работы программы.

Во всех случаях рассматривается сортировка массива, состоящего из n элементов — целых чисел. Представленные процедуры могут быть легко модифицированы и применительно к массивам, элементы которых — это данные комбинированного типа (записи), — главное только, чтобы одним из полей записи была величина числового типа, от значения которой зависит место каждого элемента в упорядоченном массиве. Массив упорядочивается в порядке неубывания.

Для иллюстрации работы процедур сортировки и для проверки их работоспособности должны также быть разработаны программы, реализующие заполнение исходного массива, вывод на экран исходного и отсортированного массивов.

Заполнение исходного массива удобнее всего проводить с помощью функции rnd, возвращающей случайное число из некоторого интервала (см. раздел 7). Для этого в программах используется вспомогательная процедура Заполнение:

```
алг Заполнение(арг рез цел таб a[1:n])
нач цел i
| нц для i от 1 до n
| | a[i] := rnd(100)
| кц
кон
```

Обратим внимание, что формальный параметр процедуры здесь описан как параметр-переменная (т. е. в результате выполнения процедуры массив должен измениться — см. раздел 9).

Вывод массива на экран может быть проведен с помощью процедуры Печать:

```
алг Печать(арг цел таб a[1:n])
нач цел i
| нц для i от 1 до n
| | вывод a[i], " "
| кц
кон
```

Здесь в описании процедуры использован параметр-значение.

---

[32] Если вы не знакомы с правилами оформления собственных процедур и функций, то можете включать соответствующие операторы в основную программу.

Вся программа сортировки массива в этом случае будет иметь вид:

```
цел n     |Глобальная константа
n := ...  |Размерность массива
алг Основная_часть_программы
|В школьном алгоритмическом языке
|основная часть программы размещается
|раньше всех вспомогательных процедур
нач цел таб a[1:n]
| |Заполняем массив a
| Заполнение(a)
| |и печатаем его
| вывод нс, "Исходный массив:"
| Печать(a)
| |Сортируем исходный массив:
| Сортировка(...)
| |Печатаем отсортированный массив:
| вывод нс, "Отсортированный массив:"
| Печать(a)
кон
```

где Сортировка — это процедура сортировки массива каким-либо конкретным методом.

И последнее: для некоторых процедур сортировки проводится их усовершенствование с целью уменьшения времени выполнения программ — предлагаю читателям еще раз прочитать слова К. Гаусса, приведенные в разделе 1.

11.1. Сортировка подсчетом

Этот метод основан на следующей, очень понятной, идее: место каждого элемента в отсортированном массиве зависит от количества элементов, меньших его. Иными словами, если значение некоторого элемента исходного массива превышает значения четырех других, то его место в упорядоченной последовательности — пятое (рис. 11.1).

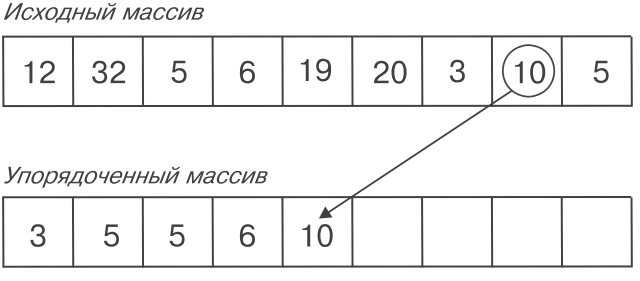


Рис. 11.1

И наоборот, если какое-либо число в отсортированном массиве находится на пятом месте, то это значит, что в исходном массиве есть четыре элемента, которые меньше или равны ему. Следовательно, для сортировки необходимо для каждого элемента заданного массива подсчитать количество элементов, меньших его, и затем разместить каждый рассмотренный элемент на соответствующем месте в новом, специально созданном, массиве.

Используем в процедуре сортировки следующие величины: a — исходный массив; b — отсортированный массив; n — количество элементов в массиве; i — индекс рассматриваемого элемента; j — индекс элемента, с которым сравнивается рассматриваемый; k — количество элементов, меньших, чем рассматриваемый.

Процедура, выполняющая сортировку массива a методом подсчета и заполняющая отсортированными элементами массив b, имеет вид:

```
алг Сортировка_подсчетом(арг цел таб a[1:n], арг рез цел таб b[1:n])
нач цел i, j, k
| |Для каждого элемента массива a
| нц для i от 1 до n
| | |определяем величину k,
| | |сравнивая его со всеми
| | |остальными элементами
| | k := 0
| | нц для j от 1 до i - 1
| | | если a[j] < a[i]
| | | | то
| | | |  k := k + 1
| | | все
| | кц
| | нц для j от i + 1 до n
| | | если a[j] < a[i]
| | | | то
| | | |  k := k + 1
| | | все
| | кц
| | |Размещаем элемент a[i]
| | |на (k + 1)-м месте в массиве b
| | b[k + 1] := a[i]
| кц
кон
```

Так как приведенная выше процедура использует для сортировки дополнительный массив, то при выводе на экран упорядоченной последовательности в основной части программы следует использовать массив b:

```
цел n
n := ...
алг Основная_часть_программы
нач цел таб a[1:n], b[1:n]
| Заполнение(a)
| вывод нс, "Исходный массив:"
| Печать(a)
| Сортировка_подсчетом(a, b)
| вывод нс, "Отсортированный массив:"
| Печать(b)
кон
```

Нетрудно заметить, что данная процедура не решает задачу сортировки при наличии в исходном массиве одинаковых элементов (в подобном случае все одинаковые элементы будут записаны один поверх другого (стирая тем самым предыдущий) в массиве b только в одну, самую левую ячейку из тех, которые они должны занимать на самом деле, а в остальных ячейках будет записан все тот же начальный нуль[33]).

Для массивов, заполненных неотрицательными числами, этот недостаток можно ликвидировать, если:

а) массив b предварительно заполнить значениями, равными нулю (в программах на школьном алгоритмическом языке, в отличие от других языков программирования, при инициализации процедуры нулевые значения величинам не присваиваются);

б) каждый элемент массива b, равный нулю, заменять значением предшествующего элемента (разумеется, кроме первого элемента массива)[34].

Процедура формирования упорядоченного массива b, учитывающая это обстоятельство, будет следующей:

```
алг Сортировка_подсчетом(арг цел таб a[1:n],
                         арг рез цел таб b[1:n])
нач цел i, j, k
| |Заполняем массив b нулями
| нц для i от 1 до n
| | b [i] := 0
| кц
| |Для каждого элемента массива a
| нц для i от 1 до n |
| | |определяем величину k
| | k := 0
```

[33] Проверьте это на примере массива, скажем, из 30 элементов, заполненного числами из интервала от 0 до 10.

[34] Для массива, элементами которого являются также и отрицательные числа, можно выполнить аналогичные действия, используя вместо нуля значение, минимальное среди всех значений элементов массива.

```
| | нц для j от 1 до i - 1
| | | ...
| | кц
| | |Размещаем элемент a[i] на (k + 1)-м месте в массиве b
| | b[k + 1] := a[i]
| кц
| |Элементы массива b, начиная со второго,
| |равные нулю, заменяем предшествующими элементами
| нц для i от 2 до n
| | если b[i] = 0
| | | то
| | |  b[i] := b[i - 1]
| | все
| кц
кон
```

Основная часть программы в этом случае оформляется так же, как и ранее.

Однако и этот вариант имеет недостаток: для проверки элементов на равенство нулю приходится заново проходить почти по всему массиву. Кроме того, надо обязательно знать минимальное среди всех элементов массива значение.

Самостоятельно разработайте вариант процедуры, не имеющий этих недостатков. В нем место для размещаемого элемента `a[i]` определяется сразу после подсчета значения величины `k` на основе следующих рассуждений. Перед тем как записать значение в (`k`+1)-ю ячейку, надо сравнить его с содержимым этой ячейки: если они равны, то проверяем следующую ячейку и т. д. до тех пор, пока не будет найдена ячейка с другим значением, куда и заносится рассматриваемый элемент. Эти рассуждения не распространяются на начальные нулевые элементы — все они уже расставлены на свои места.

Усовершенствование процедуры сортировки

Проанализировав приведенную выше процедуру сортировки, можно увидеть, что при подсчете значения `k` половина действий является излишней. В самом деле, после сравнения `i`-го элемента массива `a` с его `j`-м элементом уже не надо сравнивать `j`-й элемент с `i`-м. Можно сравнивать элементы так, как показано в таблице:

Значение индекса `i`	**Индексы `j` элементов, с которыми сравнивается `i`-й элемент**
1	**2, 3, ...,** `n`
2	**3, 4, ...,** `n`
...	
`n` **— 1**	`n`

Достаточно сравнить значения a[i] и a[j] при всех i и j, удовлетворяющих условиям: $1 \le i \le n-1$ и $i+1 \le j \le n$. Однако при этом возникает проблема подсчета количества элементов, меньших каждого i-го. Например, для массива 3, 2, 5, 1, ..., если подсчитывать значение k так же, как ранее, то при определении k для третьего элемента он не будет сравниваться с первым и вторым, и в результате значение k будет рассчитано неправильно. Чтобы решить эту проблему, надо использовать не одну переменную k для всех элементов, а массив с тем же именем из n элементов и при сравнении элементов для каждой пары индексов (i, j) определять значения k как для i-го элемента, так и для j-го.

Соответствующая процедура сортировки массива a методом подсчета имеет вид:

```
алг Сортировка_подсчетом(арг цел таб a[1:n], арг рез цел таб b[1:n])
нач цел таб k[1 : n], цел i, j
| |Подсчитываем значение k[i] для каждого элемента массива a
| нц для i от 1 до n
| | k[i] := 0
| кц
| нц для i от 2 до n
| | нц для j от 1 до i - 1
| | | если a[i] < a[j]
| | | | то
| | | |  |Увеличиваем значение k для j-го элемента
| | | |  k[j] := k[j] + 1
| | | | иначе
| | | |  |Увеличиваем значение k для i-го элемента
| | | |  k[i] := k[i] + 1
| | | все
| | кц
| кц
| |Размещаем все элементы массива a
| |на соответствующих им местах в массиве b
| нц для i от 1 до n
| | b[k[i] + 1] := a[i]
| кц
кон
```

Сопоставим количество сравнений элементов при заполнении массива b в процедуре сортировки, приведенной ранее, и в последнем ее варианте. В первом случае это количество равно n (n – 1), где n — количество элементов в массиве; во втором — n (n – 1) / 2, т. е. вдвое меньше.

Интересно, что проведенные усовершенствования не только уменьшают время выполнения программы, но и обеспечивают сортировку массива, в котором имеются одинаковые элементы, без последующего изменения значений нулевых элементов массива b (см. выше)[35].

[35] Почему это происходит, проанализируйте самостоятельно.

Частный случай метода подсчета

Существует еще и другая разновидность сортировки методом подсчета. Правда, применима она в основном, если в массиве имеется много одинаковых элементов и их значения лежат в диапазоне $u \le a[i] \le v$, где разность $(v - u)$ невелика.

Чтобы понять основную идею, предположим, что все элементы исходного массива a имеют значения от 1 до 50. Для получения упорядоченного массива можно при первом просмотре массива a подсчитать количество элементов, равных 1, 2, ..., 50, а при втором — записать эти элементы в соответствующие ячейки массива b (или a, заменив в нем исходные значения).

Эта разновидность метода сортировки называется *методом распределяющего подсчета* (вариант же, описанный ранее, обычно называется *методом сравнения и подсчета* [5]).

При использовании метода распределяющего подсчета количество выполняемых сравнений равно $n\ (v - u + 1)$, где n — количество элементов в сортируемом массиве, а $(v - u + 1)$ — количество возможных значений элементов этого массива.

Сопоставляя это количество сравнений с полученным ранее, можно сделать вывод, что сортировка методом распределяющего подсчета эффективнее метода сравнения и подсчета, если $(v - u + 1) < (n - 1) / 2$.

Этому требованию удовлетворяет, например, условие следующей задачи: *Дан массив, элементами которого являются цифры. Требуется упорядочить его по возрастанию.*

Представим процедуру сортировки такого массива цифр по методу распределяющего подсчета[36]. В ней мы используем следующие основные величины: a — сортируемый массив; кол_цифр — массив из 10 элементов, значениями которых являются количества каждой из цифр (0, 1, ..., 9) в массиве a (рис. 11.2); z — индекс элемента массива, которому присваивается значение при втором просмотре (см. выше).

```
алг Сортировка_подсчетом(арг рез цел таб a[1:n])
нач цел таб кол_цифр[0:9], цел i, j, f
| |1. Для каждой цифры подсчитываем
| |ее количество в массиве a
| нц для i от 0 до 9
| | кол_цифр[i] := 0
```

[36] Соответствующий фрагмент процедуры можно оформить и короче:

```
нц для i от 0 до 9
|кол_цифр[i] := 0
кц
нц для j от 1 до n
|кол_цифр[a[j]] := кол_цифр[a[j]] + 1
кц
```

```
| | нц для j от 1 до n
| | | если a[j] = i
| | | | то
| | | |  кол_цифр[i] := кол_цифр[i] + 1
| | | все
| | кц
| кц
| |2. Размещаем все элементы массива a
| |(т. е. каждую цифру) в порядке возрастания цифр
| |и в соответствующем количестве в этом же массиве
| z := 1
| нц для i от 0 до 9
| | |Каждую i-ю цифру записываем
| | |в кол_цифр[i] подряд идущих
| | |ячеек массива a
| | нц для j от 1 до кол_цифр [i]
| | | a[z] := i
| | | |Смещаем "указатель записи" z
| | | z := z + 1
| | кц
| кц
кон
```

Массив ***a:***

2	3	0	1	1	2	4	6	7	8	3	2	5	6	7	3	0	9	0	5

Массив ***кол_цифр:***

3	2	3	3	1	2	2	2	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Рис. 11.2

На известном вам языке программирования разработайте программы сортировки массива:

а) методом сравнения и подсчета;

б) методом распределяющего подсчета.

Во втором случае используйте массив из 100 элементов, значения которых находятся в диапазоне от 30 до 45. Проанализируйте, будет ли работать процедура сортировки, если каких-либо значений из указанного диапазона в массиве не окажется.

11.2. Сортировка выбором

Сортировка выбором состоит в том, что сначала в неупорядоченном массиве выбирается минимальный элемент[37]. Этот элемент исключается из дальнейшей обработки, а оставшаяся последовательность элементов принимается за исходную, и процесс повторяется до тех пор, пока все элементы не будут выбраны. Очевидно, что выбранные элементы образуют упорядоченную последовательность.

При этом выбранный в исходном массиве минимальный элемент может быть размещен на предназначенном ему месте упорядоченной последовательности разными способами. Один из них заключается в использовании дополнительного массива. После первого просмотра найденный минимальный элемент размещается на первом месте в этом массиве (рис. 11.3). Однако если на втором просмотре исходного массива вновь найти минимальный элемент, то им окажется все тот же самый (найденный ранее) элемент. Чтобы исключить эту ситуацию, надо в исходном массиве вместо выбранного записать число, заведомо превосходящее любой элемент сортируемого массива. Тогда, вновь найдя при втором проходе минимальный элемент, можно разместить его на втором месте в дополнительном массиве, и т. д. Тогда общая схема алгоритма сортировки рассмотренным методом имеет вид:

```
нц для i от 1 до n
| 1. Определить минимальный элемент массива.
| 2. Записать его на i-е место в дополнительном массиве.
| 3. В исходном массиве на место минимального элемента
| записать "большое" число.
кц
```

Что нужно знать, чтобы выполнить этап 3? Во-первых, число, которое превосходит любой элемент сортируемого массива. Будем считать, что такое число нам известно. Во-вторых, нужен индекс минимального элемента, а значит, на этапе 1 следует найти также и этот индекс.

Рис. 11.3

[37] Разумеется, можно выбрать и максимальный. В этом случае потребуется лишь небольшое изменение в алгоритме.

Программу сортировки массива обсуждаемым методом в виде исключения приведем сначала без применения вспомогательных процедур. В ней мы используем величины: мин — минимальный элемент массива[38]; индмин — его индекс; макс — число, которым заменяются выбранные минимальные элементы исходного массива.

```
|Глобальные константы
цел n, макс
n := ...
макс := ...
алг Основная_часть_программы
нач цел мин, индмин, i, j
| |Заполнение массива
| нц для i от 1 до n
| | a[i] := rnd(100)
| кц
| |Вывод массива на экран
| нц для i от 1 до n
| | вывод a[i], " "
| кц
| |Сортировка
| нц для i от 1 до n
| | |1. Находим минимальный элемент массива
| | |и его индекс
| | мин := a[1]; индмин := 1
| | нц для j от 2 до n
| | | если a[j] < a[индмин]
| | | | то
| | | |  мин := a[j]
| | | |  индмин := j
| | | все
| | кц
| | |2. Записываем минимальный элемент
| | |на i-е место в массиве b
| | b[i] := мин
| | |3. Заменяем минимальный элемент
| | |исходного массива "большим" числом
| | b[i] := макс
| кц
| |Вывод на экран
| |отсортированного массива b
| нц для i от 1 до n
| | вывод b[i], " "
| кц
кон
```

[38] Вариант программы, в котором величина min не используется, разработайте самостоятельно.

Приведем также вариант программы, в которой использованы процедуры Заполнение и Печать, а также две функции[39]:

1) Минимум, возвращающая значение минимального элемента массива:

```
алг цел Минимум(арг цел таб a[1:n])
нач цел i, m
| |m — текущее значение искомой величины
| m := a[1]
| нц для i от 2 до n
| | если a[i] < m
| | | то
| | |  m := a[i]
| | все
| кц
| знач := m   |Значение функции
кон
```

2) ИндексМинимума, определяющая индекс этого элемента:

```
алг цел ИндексМинимума(арг цел таб a[1:n])
нач цел индмин, i |индмин - текущее значение искомой величины
| индмин := 1
| нц для i от 2 до n
| | если a[i] < a[индмин]
| | | то
| | |  индмин := i
| | все
| кц
| знач := индмин   |Значение функции
кон
```

Используя созданные функции, разработаем саму процедуру сортировки массива:

```
алг Сортировка_выбором1(арг рез цел таб a[1:n],
                        рез цел таб b[1:n])
нач цел i, мин, индмин
| мин – минимальный элемент массива a
| индмин – его индекс
| нц для i от 1 до n |n раз
| | |1. Определяем минимальный элемент массива a
| | |и его индекс
| | мин := Минимум(a)
| | индмин := ИндексМинимума(a)
```

[39] Самостоятельно разработайте вариант программы, в котором используется только одна функция — ИндексМинимума.

```
| | |2. Минимальный элемент массива a
| | |записываем в i-ю ячейку массива b
| | b[i] := мин
| | |3. На месте минимального элемента
| | |размещаем число макс
| | a[индмин] := макс
| кц
кон
```

Так как приведенная здесь процедура применяет для сортировки дополнительный массив, то в основной части программы при выводе на экран упорядоченной последовательности следует использовать массив b:

```
алг Основная_часть_программы
нач цел таб a[1:n], b[1:n]
| Заполнение(a)
| вывод нс, "Исходный массив:"
| Печать(a)
| Сортировка_выбором1(a, b)
| вывод нс, "Отсортированный массив:"
| Печать(b)
кон
```

Надеемся, что приведенный вариант программы еще раз убедил читателя в преимуществах использования процедур и функций.

Второй способ сортировки выбором

Рассмотренный вариант сортировки массива методом выбора обладает двумя недостатками:

1) для его реализации требуется дополнительный массив;
2) для нахождения минимального элемента и его индекса на каждом проходе приходится просматривать все элементы массива.

Указанные недостатки устраняются, если все изменения проводить в исходном массиве. Тогда отсортировать его можно следующим образом:

1) найти минимальный элемент среди всех элементов массива и поменять его местами с первым элементом;
2) найти минимальный элемент среди второго, третьего и т. д. элементов массива и поменять его местами со вторым элементом;
3) ...

На последнем этапе определяется минимальный элемент среди двух последних по номеру индекса и найденный элемент меняется местами с предпоследним элементом исходного массива. Поскольку

после каждого просмотра упорядоченные элементы исключаются из дальнейшей обработки, размер каждого последующего обрабатываемого участка массива на 1 меньше размера предыдущего.

В программе, реализующей сортировку выбором по такому способу, будем использовать функцию ИндексМинимума, определяющую индекс минимального элемента из списка переменной длины, начиная с элемента массива a с некоторым индексом нач_инд и до последнего элемента этого массива:

```
алг цел ИндексМинимума(арг цел таб a [1:n], цел нач_инд)
нач цел i, индмин
| |индмин - текущее значение искомой величины
| индмин := нач_инд
| нц для i от нач_инд + 1 до n
| | если a[i] < a[индмин]
| | | то
| | |  индмин := i
| | все
| кц
| знач := индмин |Значение функции
кон
```

Напишем также процедуру, обеспечивающую обмен значениями двух величин:

```
алг Обмен(арг рез цел a, b)
нач цел c
| c := a; a := b; b := c
кон
```

Вся процедура сортировки массива a при этом оформляется очень кратко:

```
алг Сортировка_выбором2(арг рез цел таб a[1:n])
нач цел i
| нц для i от 1 до n - 1
| | Обмен(a[i], a[ИндексМинимума(a, i)])
| кц
кон
```

Красиво, не правда ли?

На известном вам языке программирования разработайте программу сортировки массива методом выбора (в двух описанных выше вариантах).

11.3. Сортировка обменом

Метод «пузырька»

Сортировка обменом — это метод, при котором все соседние элементы массива попарно сравниваются друг с другом и меняются местами, если предшествующий элемент больше последующего. В результате максимальный элемент постепенно смещается вправо и в конце концов занимает свое место (которое он должен занимать в упорядоченном массиве — крайнее правое), после чего этот элемент исключается из дальнейшей обработки. Затем процесс повторяется, и свое место занимает второй по величине элемент, который также исключается из дальнейшего рассмотрения. Так продолжается до тех пор, пока весь массив не будет упорядочен.

Например, пусть требуется провести сортировку массива: 30, 17, 73, 47, 22, 11, 65, 54. Ход сортировки этого массива отражен в таблице (представлены первые два прохода обработки):

№ прохода	Сравниваемые элементы	Обмен
Первый проход по массиву:		
1)	30 и 17	Проводится
2)	30 и 73	Нет
3)	73 и 47	Проводится
4)	73 и 22	Проводится
5)	73 и 11	Проводится
6)	73 и 65	Проводится
7)	73 и 54	Проводится
Полученный массив: 17, 30, 47, 22, 11, 65, 54, 73		
Второй проход по массиву:		
1)	17 и 30	Нет
2)	30 и 47	Нет
3)	47 и 22	Проводится
4)	47 и 11	Проводится
5)	47 и 65	Нет
6)	65 и 54	Проводится
Полученный массив: 17, 30, 22, 11, 47, 54, 65, 73		

Если же последовательность сортируемых чисел расположить вертикально (где первый элемент исходного массива — внизу) и проследить за перемещением элементов (рис. 11.4), то можно увидеть, что бóльшие элементы, подобно пузырькам воздуха в воде, «всплывают» на соответствующую позицию. Поэтому сортировку таким способом и называют *сортировкой методом «пузырька»*, или *«пузырьковой» сортировкой*.

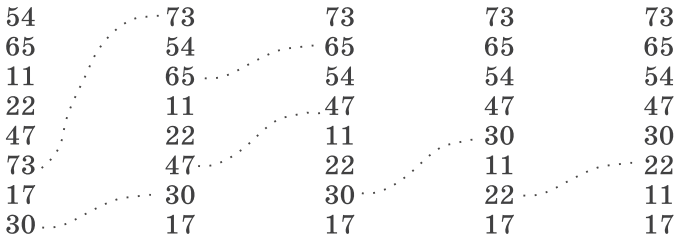


Рис. 11.4

Выпишем пары индексов элементов, сравниваемых на каждом проходе сортировки, в виде таблицы:

Номер прохода по массиву	Индексы
Первый	1—2 2—3 … (n—2)—(n—1) (n—1)—**n**
Второй	1—2 2—3 (n—2)—**(n—1)**
…	
Последний	1—**2**

Нетрудно убедиться, что общее количество проходов при этом равняется (n – 1).

Если индекс правого элемента в последней паре сравниваемых на каждом проходе чисел обозначить как посл_прав (соответствующие значения в таблице выделены жирным шрифтом), то можно составить следующую процедуру сортировки массива a методом «пузырька»:

```
алг Пузырьковая_сортировка1(арг рез цел таб a[1:n])
нач цел посл_прав, i
| нц для посл_прав от n до 2 шаг -1
| | нц для i от 1 до посл_прав – 1
| | | если a[i] > a[i + 1]
| | | | то
| | | |  Обмен(a[i], a[i + 1])
| | | все
| | кц
| кц
кон
```

где Обмен — процедура, обеспечивающая обмен значениями двух величин (см. подраздел 11.2).

Усовершенствованная «пузырьковая» сортировка

Нетрудно заметить, что в приведенной ранее процедуре сравнения элементов будут продолжаться даже после того, как массив станет упорядоченным после некоторого, не последнего прохода. Так, из рис. 11.4 следует, что уже после пятого прохода массив станет упорядоченным, но сравнения элементов (бесполезные!) будут продолжаться. Желательно прекратить проходы по массиву сразу, как только он станет упорядоченным.

Но как определить момент, в который массив стал упорядоченным? Можно, конечно, после каждого прохода попарно сравнивать все элементы, но данный способ нерационален: количество сравнений при этом существенно возрастает. Лучше применить следующие рассуждения. Ясно, что если на очередном проходе не было сделано ни одного обмена, то массив уже стал упорядоченным, и все действия можно прекратить. Если мы будем фиксировать данный факт, то небольшое усложнение программы может дать существенный выигрыш во времени ее выполнения. В частности, можно использовать величину логического типа был_обмен, определяющую, выполнялся ли обмен элементов при последнем проходе, и очередной проход проводить только в случае положительного ответа на этот вопрос, применив оператор цикла с условием.

Приведем процедуру, учитывающую это обстоятельство:

```
алг Пузырьковая_сортировка2(арг рез цел таб a[1:n])
нач цел посл_прав, i, лог был_обмен
| |Если обмен был, то был_обмен = да,
| |в противном случае — был_обмен = нет
| был_обмен := да |Условно
| посл_прав := n
| нц пока посл_прав >= 2 и был_обмен
| | был_обмен := нет
| | нц для i от 1 до посл_прав - 1
| | | если a[i] > a[i + 1]
| | | | то
| | | |  Обмен(a [i], a[i + 1])
| | | |  был_обмен := да
| | | все
| | кц
| | посл_прав := посл_прав - 1
| кц
кон
```

Здесь условие, записываемое в операторе цикла с предусловием, определяется по правилу, изложенному в разделе 3.

Процесс совершенствования метода «пузырька» можно продолжать, если фиксировать не только сам факт, но и место (индексы элементов) последнего обмена, а затем учитывать его для уточнения значения посл_прав.

Объясним сказанное на примере. Из рис. 11.4 видно, что на третьем проходе последними обменивались местами числа 30 и 11 (их индексы, соответственно, — 3 и 4). Число 30 заняло свое место в массиве, и следующий проход можно делать до значения посл_прав, равного 3.

Соответствующая процедура имеет вид:

```
алг Пузырьковая_сортировка3(арг рез цел таб a[1:n])
нач цел посл_прав, i, лог был_обмен
| был_обмен := да
| посл_прав := n
| нц пока посл_прав >= 2 и был_обмен
| | был_обмен := нет
| | нц для i от 1 до посл_прав - 1
| | | если a[i] > a[i + 1]
| | | | то
| | | |  Обмен(a [i], a[i + 1])
| | | |  был_обмен := да
| | | |  |Запоминаем место обмена
| | | |  |(новое значение величины посл_прав)
| | | |  посл_прав := i
| | | все
| | кц
| | |Следующий оператор уже не нужен
| | |и оформлен в виде комментария
| | |для наглядности
| | |посл_прав := посл_прав - 1
| кц
кон
```

Очевидно, что усовершенствования «пузырьковой» сортировки дают наибольший эффект, когда исходный массив уже почти отсортирован. Для массива же, таковым не являющегося, усовершенствованный метод «пузырьковой» сортировки дает даже несколько худшие показатели, чем обычный метод сортировки обменом [6].

«Шейкер»-сортировка

Несмотря на все сделанные выше усовершенствования, «пузырьковая» сортировка следующего (почти упорядоченного!) массива: 5, 7, 12, 17, 28, 36, 85, 2 будет проведена за 7 проходов. Это связано с тем, что при сортировке рассматриваемым методом за один проход любой элемент не может переместиться влево более чем на одну позицию. Так что если минимальный элемент массива находится в его правом

конце (как и обстоит дело в рассматриваемом примере), то придется выполнить максимальное число проходов. Поэтому, естественно, напрашивается еще одно улучшение метода «пузырьковой» сортировки: попеременные проходы массива в обоих направлениях, а не только от его начала к концу. В этом случае время сортировки может несколько сократиться. Такой метод сортировки называется *«шейкер»-сортировкой* (от английского слова «shake» — «трясти, встряхивать»). Его работа показана на рис. 11.5 на примере сортировки массива из восьми элементов: 67, 6, 18, 94, 42, 12, 55, 4.

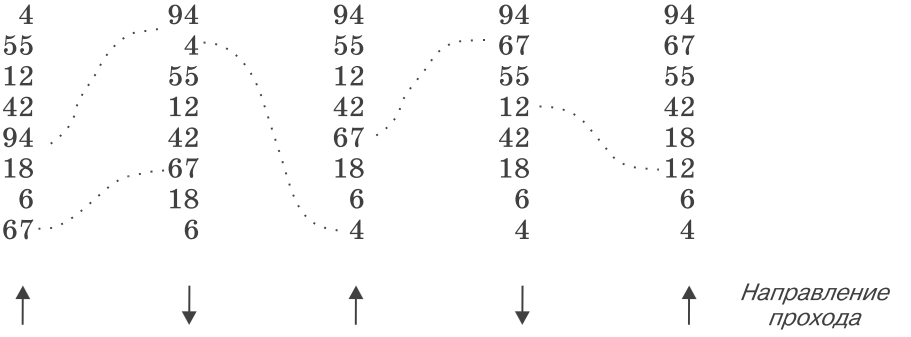


Рис. 11.5

В процедуре, реализующей «шейкер»-сортировку, мы будем использовать две новые величины: лев — левая граница (индекс) участка массива, обрабатываемого на каждом проходе; прав — то же, правая граница (индекс), а процесс упорядочивания массива будем проводить, пока соблюдается условие: лев < прав. Такая процедура может быть представлена следующим образом:

```
алг Шейкер_Сортировка(арг рез цел таб a[1:n])
нач цел лев, прав, i, лог был_обмен
| был_обмен := да
| лев := 1; прав := n
| нц пока лев < прав и был_обмен
| | был_обмен := нет
| | |Проход слева направо
| | нц для i от лев до прав - 1
| | | если a[i] > a[i + 1]
| | | | то
| | | |  Обмен(a[i], a[i + 1])
| | | |  был_обмен := да
| | | все
| | кц
| | |Меняем правую границу
| | прав := прав - 1
```

```
| | |Проход справа налево
| | нц для i от прав до лев + 1 шаг -1
| | | если a[i] < a[i - 1]
| | | | то
| | | |  Обмен(a[i], a[i - 1])
| | | |  был_обмен := да
| | | все
| | кц
| | |Меняем левую границу
| | лев := лев + 1
| кц
кон
```

Как и в случае метода «пузырька», «шейкер»-сортировку можно ускорить, если фиксировать места (индексы) последнего обмена при проходах слева направо и справа налево, а затем использовать их для уточнения границ обрабатываемого участка массива. (Соответствующую процедуру разработайте самостоятельно.)

На известном вам языке программирования разработайте программы сортировки массива методом:

1) «пузырьковой» сортировки (в трех описанных вариантах);

2) «шейкер»-сортировки.

11.4. Сортировка вставками

При сортировке вставками (или «включениями» [6]) из неупорядоченной последовательности элементов поочередно выбирается каждый элемент, сравнивается с предыдущим (уже упорядоченным) списком и помещается на соответствующее место в нем. Заметим, что такой метод упорядочивания обычно используют игроки в карты.

Сортировку вставками мы рассмотрим на примере следующей неупорядоченной последовательности элементов: 38, 12, 80, 15, 36, 23, 74, 62. Методику сортировки иллюстрирует рис. 11.6, где для каждого этапа жирным выделен очередной элемент, а стрелкой сверху отмечено место для его размещения (при необходимости). На второй строке для каждого этапа приведен вид массива после требуемого размещения элемента.

На первом этапе рассмотрим элемент 12 (второй по счету, пропуская первое число 38). Он меньше 38, поэтому ставится на первое место, а число 38 сдвигается вправо (элементы упорядоченной последовательности, смещаемые вправо, на рис. 11.6 подчеркнуты). Теперь упорядоченная последовательность состоит уже из двух элементов 12, 38.

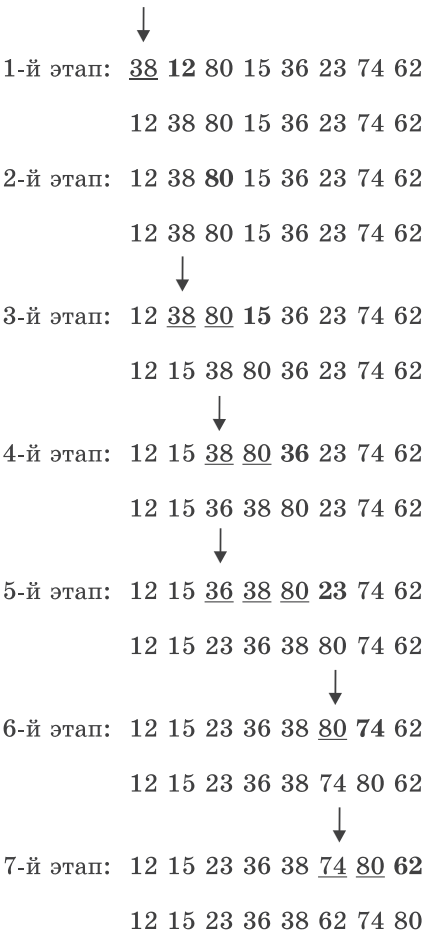


Рис. 11.6

На втором этапе рассматриваем первый элемент оставшейся неупорядоченной последовательности — 80. Он больше всех элементов ранее упорядоченной последовательности и потому остается на своем месте.

На третьем этапе упорядоченная последовательность состоит из чисел 12, 38, 80, а рассматриваемый элемент — 15, и т. д.

Подобным образом последовательность меняется до тех пор, пока она не станет упорядоченной (это достигается на седьмом этапе). Иными словами, общая схема алгоритма сортировки массива `a` методом вставок выглядит следующим образом:

```
цикл для i от 2 до n
| Разместить элемент a[i] на соответствующем ему
| месте в предшествующей, уже упорядоченной, последовательности
конец цикла
```

Размещение же элемента массива на соответствующем ему месте в предшествующей, уже упорядоченной последовательности может быть проведено двумя способами:

1) можно сначала определить место, соответствующее рассматриваемому элементу в упорядоченной последовательности, а затем поместить там этот элемент, сдвинув соответствующие элементы на одну позицию вправо, как показано на рис. 11.6; эту разновидность метода мы назовем «*поиск места и вставка*»;
2) можно последовательно сравнивать рассматриваемый элемент с элементом, расположенным слева от него, и обменивать их местами, пока слева от перемещаемого элемента не окажется элемент, меньший или равный ему. Такой способ мы будем называть «*сравнение и обмен*» (или «*просеивание*» [6]).

Рассмотрим процедуры, соответствующие этим вариантам.

Поиск места и вставка

В процедуре, реализующей сортировку вставками с таким вариантом размещения, целесообразно использовать вспомогательные функцию и процедуру:

1) функция `ПоискМеста` — определяет место элемента массива с индексом `i` в предшествующей ему упорядоченной части массива;
2) процедура `Вставка` — обеспечивает перемещение в массиве элемента с индексом `i` на `m`-ю позицию и смещение всех элементов с индексами от `m` до (`i` – 1) на одну позицию вправо.

Функция `ПоискМеста` имеет вид:

```
алг цел ПоискМеста(арг цел таб a[0:n], цел i)
нач цел m  |Текущее значение искомой величины
| m := 1
| нц пока a[m] < a[i]
| | m := m + 1
| кц
| знач := m |Значение функции
кон
```

Процедура `Вставка`:

```
алг Вставка(арг рез цел таб a[1:n], арг цел i, m)
нач цел всп, k
| |Запоминаем значение элемента a[i]
| |во вспомогательной переменной всп
| всп := a[i]
| |Элементы с индексами от i – 1 до m
| |смещаем вправо на одну позицию
```

```
| нц для k от i - 1 до m шаг -1
| | a[k + 1] := a[k]
| кц
| |Размещаем элемент a[i] на m-й позиции
| a[m] := всп
кон
```

Соответствующие задачи уже рассматривались нами в разделе «Типовые задачи обработки одномерных числовых массивов».

Процедура сортировки вставками в этом случае оформляется следующим образом:

```
алг Сортировка_Вставками1(арг рез цел таб a[1:n])
нач цел i, m
| нц для i от 2 до n
| | |Если нужно перемещать i-й элемент
| | если a[i] < a[i - 1]
| | | то
| | |  |Находим место m для элемента a[i]
| | |  m := ПоискМеста(a, i)
| | |  |Перемещаем i-й элемент на m-е место
| | |  Вставка(a, i, m)
| | все
| кц
кон
```

Примечание. В приведенной процедуре и далее использовано условие перемещения элементов (`a[i] < a[i - 1]`). Целесообразность этого приема обсуждается в конце данного раздела.

Сравнение и обмен

В этом варианте размещения `i`-го элемента соответствующие действия могут быть оформлены в виде следующих команд (`j` — индекс, который размещаемый элемент имеет в ходе его перемещения):

```
|Если нужно перемещать i-й элемент
если a[i] < a[i - 1]
| то
|  j := i
|  нц
|  | |Смещаем его влево путем обмена
|  | Обмен(a[j], a[j - 1])
|  | |Новый индекс перемещаемого элемента
|  | j := j - 1
|  кц при j = 1 или a[j] >= a[j - 1]
все
```

где `Обмен` — процедура, обеспечивающая обмен значениями двух величин.

Примечание. Использование для окончания цикла дополнительного условия j = 1 позволяет избежать сравнения j-го элемента с несуществующим (j – 1)-м, что имеет место при j = 1. Это, в свою очередь, достигается, когда размещаемый элемент меньше всех элементов, расположенных до него, например, как в массиве: 12 15 36 38 80 **3** 74 62.

Можно также применить оператор цикла с предусловием:

```
|Если нужно перемещать i-й элемент
если a[i] < a[i - 1]
| то
|  j := i
|  нц пока j > 1 и a[j] < a[j - 1]
|  | |Смещаем его влево путем обмена
|  | Обмен(a[j], a[j - 1])
|  | |Новый индекс перемещаемого элемента
|  | j := j - 1
|  кц
все
```

Процедура же сортировки, использующая соответствующий фрагмент с оператором цикла с постусловием, имеет вид:

```
алг Сортировка_Вставками2(арг рез цел таб a[1:n])
нач цел i, j
| нц для i от 2 до n
| | если a[i] < a[i - 1]
| | | то
| | |  j := i
| | |  нц
| | |  | Обмен(a[j], a[j - 1])
| | |  | j := j - 1
| | |  кц при j = 1 или a[j] >= a[j - 1]
| | все
| кц
кон
```

Ясно, что использование в процедурах дополнительного условия j = 1, о котором говорилось выше, приводит к существенному увеличению времени выполнения программ (так как для каждого элемента многократно проверяется это дополнительное условие). Данный недостаток можно устранить, если применить так называемый «барьер» [6] — создать в сортируемом массиве элемент с индексом 0 (соответственно, расширив диапазон индексов в описании массива[40]) и на каждом шаге сортировки присваивать ему значение, равное значению очередного размещаемого элемента.

[40] В языках программирования Бейсик и Си такое значение нижнего диапазона индексов массива принимается по умолчанию.

Процедура, применяющая «барьер», имеет вид:

```
алг Сортировка_Вставками3(арг рез цел таб a[0:n])
нач цел i, j
| нц для i от 2 до n
| | если a[i] < a[i - 1]
| | | то
| | |  |Создаем "барьер"
| | |  a[0] := a[i]
| | |  j := i
| | |  нц
| | |  | |Смещаем i-й элемент влево путем обмена
| | |  | Обмен(a[j], a[j - 1])
| | |  | |Новый индекс перемещаемого элемента
| | |  | j := j - 1
| | |  кц при a[j] >= a[j - 1]
| | все
| кц
кон
```

Можно также ускорить работу, если в ходе размещения не проводить обмен значениями двух соседних элементов массива (процедура `Обмен` выполняет 3 операции присваивания!). Ведь достаточно только смещать вправо на одну позицию элементы, меньшие рассматриваемого, а затем разместить последний на найденном для него месте:

```
алг Сортировка_Вставками4(арг рез цел таб a[0:n])
нач цел i, j
| нц для i от 2 до n
| | если a[i] < a[i - 1]
| | | то
| | |  |Создаем "барьер" (и запоминаем
| | |  |значение элемента a[i])
| | |  a[0] := a[i]
| | |  j := i
| | |  нц
| | |  | |Смещаем соседний слева элемент вправо
| | |  | a[j] := a[j - 1]
| | |  | j := j - 1
| | |  кц при a[0] >= a[j - 1]
| | |  |Место j, соответствующее элементу a[i], найдено
| | |  |На найденном месте размещаем элемент a[0]
| | |  a[j] := a[0]
| | все
| кц
кон
```

В приведенной выше процедуре величина `a[0]` хранит значение размещаемого элемента массива `a[i]`.

Заметим, что если все элементы массива — неотрицательные числа, то механизм использования «барьера» упрощается: достаточно только расширить нижнюю границу диапазона индексов до 0, и тогда элемент `a[0]`, в котором записано нулевое значение, будет сам выполнять роль «барьера» для всех размещаемых элементов.

Бинарная вставка

Место, соответствующее некоторому элементу массива в предшествующей ему подпоследовательности, можно найти значительно быстрее, пользуясь тем, что эта подпоследовательность уже упорядочена. Для этого необходимо использовать так называемый *бинарный поиск*, который исследует средний элемент упорядоченной последовательности и продолжает деление пополам, пока не будет найдено требуемое место включения. Такой метод поиска использовался при решении задачи 6.9 в разделе «Типовые задачи обработки одномерных числовых массивов». На его основе разработаем функцию `БинарныйПоискМеста`, определяющую индекс ячейки в упорядоченной части массива `а`, в которой должно размещаться некоторое число `m`, где упорядоченная часть массива имеет границы индексов `1:посл`.

Обозначим: `лев` — индекс левого элемента исследуемого в ходе поиска отрезка массива, `прав` — правого, `сер` — величину, равную `(лев+прав)/2`. В начале поиска принимаем `лев = 1`, `прав = посл`. Определяем величину `сер` и сравниваем значения `a[сер]` и `m`. Если `a[сер] > m`, то размещаемое число находится в левой половине исследуемого диапазона, т. е. можно принять `прав = сер - 1`; если `a[сер] < m` — оно находится в правой половине (`лев = сер + 1`); в противном случае (`a[сер] = m`) искомое место уже найдено. В двух первых вариантах аналогичные рассуждения необходимо далее применить к новой части массива, и т. д. Действия прекращаются, когда `прав` станет меньше `лев` либо когда `a[сер] = m`. В этой ситуации искомый индекс равен значению `сер`.

Тогда функция `БинарныйПоискМеста` имеет вид:

```
алг цел БинарныйПоискМеста(арг цел посл, цел таб a[1:посл], цел m)
нач цел лев, прав, сер
| лев := 1
| прав := посл
| нц
| | сер := div(лев + прав, 2)
| | если a[сер] > m
| | | то
| | |  прав := сер - 1
| | | иначе
| | |  если a[сер] < m
| | |  | то
| | |  |  лев := сер + 1
| | |  все
| | все
| кц при прав < лев или a[сер] = m
| знач := сер
кон
```

Процедура же сортировки методом бинарных вставок, использующая приведенную функцию, оформляется так:

```
алг Сортировка_Бинарными_Вставками(арг рез цел таб a[1:n])
нач цел i
| нц для i от 2 до n
| | если a[i] < a[i - 1]
| | | то
| | |  |Вставляем элемент a[i]
| | |  |на соответствующее ему место,
| | |  |найденное методом бинарного поиска
| | |  |на отрезке массива a
| | |  |с индексами от 1 до i - 1
| | |  Вставка(a, i, БинарныйПоискМеста(i - 1, a, a[i]))
| | все
| кц
кон
```

Разновидность метода вставок, применяющая бинарный поиск места размещения очередного элемента, называется *сортировкой бинарными вставками*. Сравним ее с вариантами, описанными выше (эти варианты сортировки носят название «*метод простых вставок*»).

Когда при сортировке простыми вставками на соответствующем месте размещается i-й элемент, то его значение сравнивается в среднем примерно с i / 2 ранее отсортированными значениями, поэтому общее количество сравнений равно: $(1 + 2 + \ldots + n) / 2 \approx n^2/4$. Это очень много, даже если n умеренно велико!

При использовании же бинарного поиска места размещения очередного элемента количество сравнений значительно меньше. Например, если очередной элемент — 64-й, то его значение сначала сравнивается с 32-м, затем с 16-м или 48-м и т. д., так что искомое место будет найдено максимум после всего лишь шести сравнений. Общее же количество сравнений для n элементов равно приблизительно $n \log_2 n$, что существенно меньше, чем $n^2/4$.

Неприятность здесь состоит в том, что бинарный поиск решает только половину задачи — после того как мы нашли место очередного элемента, все равно нужно «вставить» этот элемент на найденное место и сместить некоторый отрезок упорядоченной последовательности на одну позицию вправо. А поскольку в компьютере для целых чисел операция присваивания выполняется за значительно большее время, чем операция сравнения[41], то общая экономия времени будет незначительной, так что общее время работы при бинарном поиске, по существу, остается пропорциональным n^2 [5].

[41] Для вещественных чисел, наоборот, сравнение выполняется медленнее, чем присваивание.

Интересен вопрос о том, имеет ли смысл перед размещением очередного элемента проверять, меньше ли он предшествующего элемента? (В представленных выше процедурах такая проверка проводится; в работах [5, 6] — нет.) Как показали расчеты, проверка условия `a[i] < a[i – 1]` не приводит к увеличению продолжительности сортировки. В то же время, если такую проверку не проводить, то в случаях, когда исходный массив упорядочен или близок к таковому, бинарный поиск потребует больше времени, чем методы поиска, описанные ранее[42] [6]! Этот пример показывает, что в ряде случаев «очевидное улучшение» программ оказывается намного менее существенным, чем кажется вначале, а иногда может на самом деле оказаться их ухудшением.

На известном вам языке программирования разработайте программу сортировки массива методом вставок во всех описанных вариантах.

11.5. Сортировка вставками с убывающим шагом

Рассматриваемый в данном подразделе метод сортировки был разработан Д. Шеллом, чье имя он и носит. Он заключается в том, что сначала все элементы массива разбиваются на группы, где в каждую группу входят элементы, отстоящие друг от друга на некоторое количество позиций `L`. Затем элементы каждой группы независимо сортируются. После этого все элементы вновь объединяются и сортируются в группах, при этом расстояние между элементами уменьшается. Процесс заканчивается после того, как будет проведено упорядочивание элементов с расстоянием между ними, равным 1. При этом для сортировки элементов каждой группы используется метод простых вставок (см. раздел 11.4)

Проиллюстрируем метод Шелла на следующем массиве: 40, 11, 83, 57, 32, 21, 75, 64.

Обозначим расстояние между упорядочиваемыми на каждом проходе элементами в группах как `L`. Сначала рассмотрим вариант, когда первоначальное значение `L` равно половине количества элементов в массиве, а каждое последующее значение вдвое меньше предыдущего. Методика сортировки представлена на рис. 11.7.

Очевидно, что в приведенных на рис. 11.7 этапах на последнем проходе (при `L = 1`) имеет место сортировка всего массива методом простых вставок. Но тогда возникает вопрос: зачем нужны были предшествующие проходы? Дело в том, что при `L = 1` массив будет

[42] Поскольку поиск места, соответствующего очередному элементу, проводится даже в случаях, когда в этом нет необходимости.

Исходный массив (в нем все элементы разбиты на группы с **L** = 4):
40 *11* 83 57 **32** *21* 75 64

Массив после упорядочивания элементов в группах с **L** = 4:
32 11 **75** 57 **40** 21 **83** 64

Массив после упорядочивания элементов в группах с **L** = 2:
32 11 40 21 75 57 83 64

Массив после упорядочивания элементов в группах с **L** = 1:
11 21 32 40 57 64 75 83

Рис. 11.7

уже в значительной степени упорядочен в результате предыдущих проходов, и общее время сортировки станет меньше, чем при сортировке исходного массива простыми вставками[43].

В процедуре сортировки, реализующей метод Шелла, как и при сортировке простыми вставками, при размещении очередного элемента на соответствующем ему месте целесообразно использовать «барьер». Причем в данном случае для этого массив следует «продлить» влево еще на некоторое число элементов (а не на один, как это делалось при сортировке простыми вставками). Это дополнительное количество элементов зависит от начального значения величины L, т. е. описание сортируемого массива в рассматриваемом случае должно быть следующим:

```
цел таб a[-div(n, 2) + 1 : n].
```

В процедуре сортировки мы используем следующие основные величины: i — индекс очередного элемента, размещаемого при сортировке; расст — расстояние между сортируемыми элементами на каждом проходе; b — индекс элемента массива, в котором размещается «барьер».

Проанализировав зависимость места размещения «барьера» от значений i и расст, можно установить, что индекс бар ячейки, в которой следует разместить «барьер», определяется следующим образом:

$$\text{бар} = \begin{cases} 0, \text{ если } \text{i mod расст} = 0, \\ -(1 - \text{i mod расст}) \text{ — в противном случае,} \end{cases}$$

где i mod расст — остаток от деления i на расст.

[43] Дополнительное преимущество метода Шелла может выявиться с учетом все более широкого распространения многоядерных микропроцессоров, позволяющих выполнять сортировку в пределах разных групп одного и того же массива одновременно в качестве пареллельно выполняемых вычислительных процессов. Однако подробно в этой книге вопросы параллелизации вычислений не рассматриваются. — *Прим. ред.*

Процедура сортировки:

```
алг Сортировка_методом_Шелла(арг рез цел таб
                                a[-div(n, 2) + 1 : n])
нач цел i, j, расст, бар
| L := div (n, 2) |Начальное расстояние
|                 |между сортируемыми элементами
| нц пока L >= 1
| | нц для i от расст + 1 до n
| | | если a[i] < a[i - L]
| | | | то          |Перемещаем элемент a[i]
| | | |  |Создаем "барьер":
| | | |  |а) определяем место размещения "барьера"
| | | |  если mod (i, расст) = 0
| | | |  | то
| | | |  |  бар := 0
| | | |  | иначе
| | | |  |  бар := -(расст - mod (i, расст))
| | | |  все
| | | |  | б) размещаем "барьер"
| | | |  а[бар] := a[i] |и одновременно запоминаем значение a[i]
| | | |  |Помещаем элемент a[i]
| | | |  |на соответствующее ему место (см. раздел 11.4)
| | | |  j := i
| | | |  нц
| | | |  | a[j] := a[j - L]
| | | |  | j := j - расст
| | | |  кц при a[бар] >= a[j - L]
| | | |  |Место j, соответствующее элементу a[i], найдено
| | | |  a[j] := a[бар] |На найденном месте размещаем элемент a[i]
| | | все
| | кц
| | расст := div(расст, 2) |Уменьшаем расстояние между элементами
| кц
кон
```

Напомним, что, как и при сортировке вставками, в приведенной процедуре элемент-«барьер» хранит значение размещаемого элемента массива `a[i]`. Заметим также, что для вывода на экран массива (исходного и отсортированного) в основной программе должна быть использована процедура, не учитывающая элементы для «барьеров» (аналогично приведенной в разделе 11.4).

До сих пор не установлено, какая последовательность расстояний между сортируемыми элементами является оптимальной, но выявлен удивительный факт, что бóльшая экономия времени происходит, когда величины расстояний `расст` на разных проходах не

кратны друг другу! Д. Кнут в [5] указывает, что разумным выбором может быть такая последовательность (записанная в обратном порядке): 1, 4, 13, 40, 121, ..., в которой количество значений $m = Int(\log_3 n) - 1$, а сами значения определяются как $расст_m = 1$; $расст_k = расст_{k+1} \times 3 + 1$ ($k = m - 1, \ldots, 1$).

Здесь Int( ) означает округление значения аргумента, указанного в скобках, до ближайшего меньшего целого числа.

Чтобы учесть указанные рекомендации в программе, следует определить значения расстояний расст, записать их в массив массив_расст из m элементов и использовать эти расстояния на каждом проходе сортировки.

Процедура сортировки, в которой величина расст принимается по рекомендациям Д. Кнута, несколько изменена по сравнению с приведенной выше:

```
алг Сортировка_методом_Шелла2(арг рез цел таб a
                                [-массив_расст[1] + 1 : n])
 |Размер массива учитывает места
 |для размещения "барьеров"
нач цел i, j, k, расст, b
| нц для k от 1 до m
| | расст := массив_расст[k] |Расстояние между сортируемыми элементами
| | нц для i от расст + 1 до n
| | | если a[i] < a[i - расст]
| | | | то
| | | |  |Размещаем элемент a[i]
| | | |  |на соответствующем ему месте (см. выше)
| | | |  ...
| | | все
| | кц
| кц
кон
```

При этом в качестве глобальных переменных в основной программе должны быть описаны и определены количество значений расстояний m и массив этих значений массив_расст:

```
цел m
m := int(ln(n) / ln(3)) - 1  |Количество различных расстояний
цел таб массив_расст[1:m]   |Массив расстояний
|Заполняем массив массив_расст
массив_расст[m] := 1 |Значение последнего элемента
                     |массива расстояний
цел k
нц для k от m - 1 до 1 шаг -1
| массив_расст[k] := 3 * массив_расст[k + 1] + 1
кц
```

Кроме того, как и ранее, в основной программе при выводе на экран значений элементов массива (исходного и отсортированного) элементы, используемые для размещения «барьеров», учитываться не должны.

В заключение отметим следующее. Как и при сортировке вставками, интересным является вопрос о том, стоит ли перед размещением очередного элемента проверять, меньше ли он предшествующего элемента в группе (в представленных процедурах такая проверка проводится, в работах [5, 6] — нет). При сортировке вставками эта проверка, как отмечено в подразделе 11.4, обеспечивает ускорение расчетов только при сортировке почти упорядоченного массива. При сортировке же методом Шелла проверка условия a[i] < a[i – 1] дает существенную экономию времени работы программ (более чем в 2 раза!) и для массива, заполненного случайным образом.

11.6. Сортировка с разделением (быстрая сортировка Хоара)

Прежде чем изучать данный метод сортировки, рассмотрим такую задачу: *Требуется переразместить элементы массива так, чтобы в новом массиве все значения, меньшие некоторого или равные ему, находились слева от него, а большие или равные — справа* (в дальнейшем элемент, относительно которого должны быть размещены остальные элементы, мы будем называть «основным»). Например, массив 38, 8, 16, 6, 79, 76, 57, 24, 56, 2, 58, 48, 4, 70, 45, 47 при основном элементе, равном первому (38), после указанного преобразования должен принять вид: 4, 8, 16, 6, 2 , 24, 38, 57, 56, 76, 58, 48, 79, 70, 45, 47.

Это можно сделать следующим образом. Используем два «указателя» лев и прав, из которых лев ведет отсчет номеров элементов слева, а прав — справа. Сначала имеем лев = 1, прав = n (в рассматриваемом случае количество элементов массива n = 16). Значение основного элемента обозначим осн.

Сравниваем осн и a[прав]. Если a [прав] > осн, то устанавливаем прав = прав – 1 и проводим следующее сравнение осн и a[прав]. Продолжаем *уменьшать* прав до тех пор, пока не достигнем a[прав] ≤ осн. Тогда поменяем местами элементы a[прав] и a[лев] (см. строку 5 на рис. 11.8, обмен значений 38 и 4), установим значение лев = лев + 1 и, сравнивая элементы a [лев] со значением осн, будем *увеличивать* лев до тех пор, пока не получим a[лев] ≥ осн. После следующего обмена (строка 10, элементы со значениями 79 и 38) опять уменьшим прав и будем просматривать элементы справа налево и т. д. до тех пор, пока не станет истинным условие прав ≤ лев.

осн = 48

	лев														прав	
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	
38	8	16	6	79	76	57	24	56	2	58	48	4	70	45	47	**Действие**
38															47	Уменьшение прав
38														45		-“-
38													70			-“-
38												4				4 < 38
4												38				Обмен
	8											38				Увеличение лев
		16										38				-“-
			6									38				-“-
				79								38				79 > 38
				38								79				Обмен
				38							48					Уменьшение прав
				38						58						-“-
				38					2							2 < 38
				2					38							Обмен
					76				38							Увеличение лев
					38				76							Обмен
					38			56								Уменьшение прав
					38		24									24 < 38
					24		38									Обмен
						57	38									Увеличение лев
						58	57									Обмен

Полученный массив:
4 8 16 6 2 24 **38** 57 56 76 58 48 79 70 45 47

Рис. 11.8

Процедура, в которой решается задача разделения массива, может быть оформлена в виде[44]:

```
алг Разделение(арг рез цел таб a[1:n])
нач цел лев, прав, осн, всп
| лев := 1; прав := n
| осн := a[1] |Значение основного элемента
| нц
| | нц пока a[прав] > осн
| | | прав := прав - 1 |Уменьшаем указатель прав
| | кц
```

[44] Можно также разработать соответствующую отдельную процедуру для выполнения операции обмена значений `a[лев]` и `a[прав]`.

```
| | если лев <= прав
| | | то
| | |  |Обмен
| | |  всп := а[лев]; а[лев] := а[прав]; а[прав] := всп
| | |  |Меняем указатель лев
| | |  лев := лев + 1
| | все
| | нц пока а[лев] < осн
| | | лев := лев + 1 |Увеличиваем указатель лев
| | кц
| | если лев <= прав
| | | то
| | |  |Обмен
| | |  всп := а[лев]; а[лев] := а[прав]; а[прав] := всп
| | |  |Меняем указатель прав
| | |  прав := прав - 1
| | все
| кц при лев >= прав
кон
```

Заметим, что при использовании строгих отношений «>» и «<» в операторах цикла с предусловием при изменении указателей прав и лев (вместо «≥» и «≤»), значения, равные основному элементу, действуют как «барьер» для обоих просмотров. Это преимущество компенсирует такой недостаток, как «лишние» обмены элементов, равных осн, которые в среднем для случайных массивов происходят крайне редко.

Теперь пора вспомнить, что наша главная цель — не разделить исходный массив на большие и меньшие, а рассортировать его. Однако от разделения до сортировки нас отделяет всего лишь один небольшой шаг: разделив массив, можно сделать то же самое с обеими полученными частями, затем — с частями этих частей и т. д., пока каждая часть не будет содержать только один элемент. Соответствующие действия можно выполнить, используя рекурсию (см. раздел 10).

Данный метод сортировки был изобретен Ч. Хоаром (C. Hoare). Исследования показали, что этот метод обладает столь блестящими характеристиками (с точки зрения времени выполнения программ), что Хоар назвал его *быстрой сортировкой*.

Прежде чем написать процедуры быстрой сортировки, заметим, что в них используется вспомогательная процедура Разделение2, работающая аналогично приведенной выше процедуре Разделение, но выполняющая разделение массива на отрезке с границами (индексами) левгр и правгр. Эта процедура рекурсивно вызывает сама себя, при этом в качестве передаваемых при рекурсивных вызовах

параметров используются значения границ `левгр` и `правгр`, а также указателей `лев` и `прав` вызывающей процедуры:

```
алг Разделение2(арг цел левгр, правгр, арг рез цел таб a[1:n])
нач цел лев, прав, осн, всп
| если левгр < правгр |Условие продолжения рекурсивных вызовов
| | то
| |  лев := левгр; прав := правгр
| |  осн := a[лев]
| |  нц
| |  | ... (см. выше процедуру Разделение)
| |  кц при лев > прав
| |  |Рекурсивно вызываем процедуру Разделение2
| |  Разделение2(лев, правгр, a)   |Разделяем левую часть
| |  Разделение 2(левгр, прав, a)  |Разделяем правую часть
| все
кон
```

При этом процедура сортировки оформляется очень просто:

```
алг Быстрая_сортировка(арг рез цел таб a[1:n])
нач
| Разделение2(a)
кон
```

В заключение заметим, что приведенные здесь процедуры работают, только если в качестве основного принимается первый элемент массива. Очевидно, что при случайном характере сортируемого массива это вполне допустимо.

11.7. Сортировка слиянием

И здесь, прежде чем приступить к описанию данного метода сортировки, решим «вводную» задачу: *Даны два натуральных числа* `m` *и* `n` *и два упорядоченных массива:* `a[1] ≤ a[2] ≤ … ≤ a[n]` *и* `b[1] ≤ b[2] ≤ … ≤ b[m]`. *Требуется образовать из элементов этих массивов упорядоченный массив* `c[1] ≤ c[2] ≤ … ≤ c[n + m]`; *количество сравнений при этом не должно превышать* `(n + m)`.

Конечно, можно решить эту задачу, используя метод вставок (см. раздел 11.4): каждый элемент массива `a` разместить на соответствующем ему месте в массиве `b`. Однако при этом количество сравнений превысит (`m + n`). Поэтому обсудим другой способ решения задачи.

Рассмотрим первые элементы обоих массивов, сравним их и меньший занесем в массив `c`. В массиве, элемент из которого был занесен в массив `c`, в дальнейшем будем рассматривать следующий элемент. Снова сравним 2 элемента из массивов `a` и `b` и также выполним аналогичные действия, и т. д. Таким образом, после каждого сравнения в массив `c` добавляется один элемент; следовательно, количество сравнений будет равно (`m + n`).

На рис. 11.9 представлены первые этапы решения данной задачи (меньший из двух сравниваемых элементов подчеркнут).

Сравнение	Полученный массив	Исходные массивы
		503 703 765 ...
1		← 87 512 677 ...
		← 503 703 765 ...
2	87	512 677 ...
		703 765 ...
3	87 503	← 512 677 ...

Рис. 11.9

Основные величины, используемые в программе (кроме переменных m, n и массивов a, b, c): i — индекс элемента, рассматриваемого в массиве a; j — то же, в массиве b; k — индекс элемента в массиве c, которому присваивается очередное значение.

Присваивание значений элементам массива c и изменение значений i и j должны происходить по следующим правилам:

```
если a[i] > b[j]
| то
|  c[k] := b[j]
|  j := j + 1
| иначе
|  c[k] := a[i]
|  i := i + 1
все
```

Однако при этом программа не будет работать после исчерпания одного из массивов.

Необходимое условие для нахождения очередного значения c[k] можно установить, рассуждая следующим образом. Присваивание элементу массива c значения из массива b должно происходить в случаях, когда a[i] > b[j] или когда массив a исчерпан; при этом массив b не должен быть исчерпан. В противном случае для присваивания используется элемент из массива a. Это утверждение оформляется в виде:

```
если j <= m и (i > n или a[i] > b[j])
| то
|  c[k] := b[j]
|  j := j + 1
| иначе
|  c[k] := a[i]
|  i := i + 1
все
```

Полностью процедура «слияния» массивов a и b в единый упорядоченный массив c выглядит так:

```
алг Слияние(арг цел n, m, цел таб a[1:n], b[1:m],
            рез цел таб c[1:n + m])
нач цел i, j, k
| i := 1; j := 1
| нц для k от 1 до n + m
| | если j <= m и (i > n или a[i] > b[j])
| | | то
| | |  c[k] := b[j]
| | |  j := j + 1
| | | иначе
| | |  c[k] := a[i]
| | |  i := i + 1
| | все
| кц
кон
```

Используя идею из этой процедуры, можно теперь решить главную задачу — отсортировать массив. Действительно, если многократно провести слияние уже упорядоченных частей исходного массива, то в конце концов он станет отсортированным. Вначале массив рассматривается как совокупность упорядоченных групп по одному элементу в каждой. Слиянием соседних групп мы получаем упорядоченные группы, каждая из которых содержит по два элемента (кроме, может быть, последней группы, в которой содержится только один элемент). Далее упорядоченные группы укрупняются тем же способом, и т. д. Количество упорядоченных групп убывает, следовательно, рано или поздно настанет такой момент, когда весь обрабатываемый массив станет упорядоченным. Естественно, что для реализации этого алгоритма надо использовать дополнительный массив b и поочередно пересылать упорядоченные группы то из массива a в массив b, то наоборот.

Разработаем вспомогательную процедуру, которая обеспечивает «слияние» двух участков массива a и запись полученной последовательности элементов в массив b. Будем считать, что «слиянию» подлежат два отрезка массива a: длиной, соответственно, длина1 и длина2 элементов, начиная с элементов с индексами нач1 и нач2. Очевидно, что заполнение массива b при этом всегда начинается с элемента, индекс которого равен индексу первого элемента первого из «сливаемых» отрезков (т. е. нач1). Кроме величин i и j, указывающих исходные элементы, будем использовать также величину k — место записи в массиве b.

С учетом этого указанная процедура оформляется в виде:

```
алг Слияние2(арг цел таб a[1:n], цел нач1, длина1, нач2, длина2,
             арг рез цел таб b[1:n])
нач цел i, j, k
| i := нач1; j := нач2
| нц для k от нач1 до нач1 + длина1 + длина2 - 1
| | если j <= нач2+длина2 - 1 и (i>нач1+длина1 - 1 или a[i] > a[j])
| | | то
| | |  b[k] := a[j]
| | |  j := j + 1
| | | иначе
| | |  b[k] := a[i]
| | |  i := i + 1
| | все
| кц
кон
```

Далее для упрощения допустим, что количество элементов в сортируемом массиве равно 2 в некоторой степени. При таком допущении длины «сливаемых» пар последовательностей будут всегда одинаковыми, а значения этих длин равны 1, 2, 4, ..., n/2.

В процедуре сортировки слиянием используем следующие величины: длина — длина «сливаемых» последовательностей; первый — индекс первого элемента первой последовательности из пары «сливаемых»; напр — величина, определяющая направление пересылки упорядочиваемых групп элементов (если напр = 1, то из массива a в массив b; если напр = −1, то наоборот).

Тогда процедура сортировки массивов «слиянием» будет выглядеть так:

```
алг Сортировка_слиянием(арг рез цел таб a[1:n], b[1:n])
нач цел первый, длина, напр
| напр := 1
| длина := 1
| нц
| | первый := 1
| | нц пока первый < n
| | | если напр = 1
| | | | то
| | | |  Слияние2(a, первый, длина, первый + длина, длина, b)
| | | |  |Переходим к следующей паре последовательностей
| | | |  первый := первый + 2 * длина
| | | | иначе
| | | |  Слияние2(b, первый, длина, первый + длина, длина, a)
| | | |  первый := первый + 2 * длина
| | | все
| | кц при длина = n
```

```
| | длина := 2 * длина |Увеличиваем длину
| |                    |"сливаемых" последовательностей
| | напр := -напр      |Переключаем направление
| |                    |пересылки элементов
| кц
| вывод нс, "Отсортированный массив: "
| |Определяем, какой массив выводить: a или b
| если напр = 1
| | то
| |  вывод нс, a
| | иначе
| |  вывод нс, b
| все
кон
```

Примечание. Для упрощения программы в процедуру `Сортировка_слиянием` включен также вывод отсортированного массива на экран. Это должно быть учтено в основной программе (см. раздел 11). Кроме того, необходимо использовать следующую процедуру `Заполнение`:

```
алг Заполнение (арг рез цел таб a[1:n], b[1:n])
нач цел i
| нц для i от 1 до n
| | a[i] := ...
| | b[i] := 0
| кц
кон
```

Теперь осталось устранить ограничение, в соответствии с которым количество элементов в сортируемом массиве `n` обязательно должно быть равно степени двойки. На какую часть процедуры сортировки это повлияет? Очевидно, что в более общем случае лучше всего использовать прежний метод до тех пор, пока это возможно. Для нас это означает, что можно проводить «слияние» отрезков массива, пока длина оставшейся, еще не «слитой», части массива больше или равна `2 * длина` (в представленных выше процедурах общая длина сливаемых последовательностей равна этой величине). Последнее условие может быть записано в виде: `n - первый + 1 = 2 * длина`.

После этого следует определить длины оставшихся частей, а затем «слить» и их. Здесь возможно два случая:

1) «сливаются» два отрезка массива, один из которых имеет «полную» длину `длина`, а второй — неполную;
2) длина второго отрезка равна нулю.

Тогда процедура определения длин отрезков массива длина1 и длина2, подлежащих «слиянию» на последней стадии сортировки, оформляется следующим образом:

```
алг Расчет_длин(арг цел первый, длина, арг рез цел длина1, длина2)
нач
| если n - первый + 1 > длина
| | то    |"сливаются" 2 отрезка
| |  длина1 := длина
| |  длина2 := n - (первый + длина - 1)
| | иначе
| |  длина1 := n - первый + 1
| |  длина2 := 0
| все
кон
```

И наконец, в общем случае, чтобы обеспечить окончание сортировки, нужно заменить условие длина = n, управляющее внешним циклом, на длина ≥ n. После этих модификаций процедура сортировки принимает вид:

```
алг Сортировка_слиянием(арг рез цел таб a[1:n], b[1:n])
нач цел таб a[1:n], b[1:n], цел первый, длина, длина1,
|                                 длина2, напр, i
| длина := 1
| напр := 1
| нц
| | первый := 1
| | нц пока n - первый + 1 >= 2 * длина
| | | |"Сливаем" отрезки прежним методом (см. выше)
| | | если напр = 1
| | | | то
| | | |  Слияние2(a, первый, длина, первый + длина, длина, b)
| | | |  первый := первый + 2 * длина
| | | | иначе
| | | |  Слияние2(b, первый, длина, первый + длина, длина, a)
| | | |  первый := первый + 2 * длина
| | | все
| | кц
| | если первый <= n  |Если массив еще не исчерпан
| | | то
| | |  |Определяем длины оставшихся отрезков
| | |  Расчет_длин(первый, длина, длина1, длина2)
| | |  |и "сливаем" их
| | |  если напр = 1
| | |  | то
| | |  |  Слияние2(a, первый, длина1, первый + длина1, длина2, b)
```

```
| | |  | иначе
| | |  |  Слияние2(b, первый, длина1, первый + длина1, длина2, a)
| | |  все
| | все
| | длина := 2 * длина
| | напр := -напр
| кц при длина >= n
| вывод нс, "Отсортированный массив: "
| если напр = 1
| | то
| |  вывод нс, a
| | иначе
| |  вывод нс, b
| все
кон
```

В заключение отметим, что рассмотренный в данном разделе метод сортировки — один из самых первых методов, предназначенных для сортировки на ЭВМ; он был предложен Джоном фон Нейманом еще в 1945 г.

11.8. Пирамидальная сортировка

Здесь мы тоже не сразу опишем новый метод сортировки, а сначала проведем необходимую подготовительную работу.

Рассмотрим ряд чисел, расположенных следующим образом:

$$\begin{array}{ccccccc} & & & 06_{(1)} & & & \\ & & 42_{(2)} & & & 12_{(3)} & \\ & 55_{(4)} & & 95_{(5)} & 18_{(6)} & & 44_{(7)} \\ & & 61_{(8)} & & 58_{(9)} & 101_{(10)} & \ldots \end{array}$$

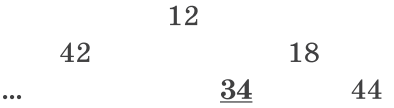
Рис. 11.10

Если пронумеровать числа a_i так, как это сделано на рис. 11.10 (номер числа указан в скобках), то нетрудно увидеть, что $a_i \le a_{2i}$ и $a_i \le a_{2i+1}$ для всякого $i = 1, 2, \ldots, n/2$, где n — количество чисел. В дальнейшем числа a_{2*i} и a_{2*i+1} мы будем называть «потомками» числа a_i.

Последовательность чисел, подчиняющаяся указанному условию, мы будем называть «*пирамидой*» [5, 6]. Очевидно, что на вершине пирамиды находится минимальный элемент. Это обстоятельство приводит к мысли о возможности сортировки чисел в пирамиде следующим способом. Можно взять число с вершины и разместить его на последнем месте в пирамиде, а число, находившееся на этом месте,

а) исходная ситуация (число 34 — в вершине пирамиды)

б) ситуация после первого шага «просеивания»



в) ситуация после второго шага «просеивания»

Рис. 11.11

расположить так, чтобы числа без этого первого минимума также составляли пирамиду (чтобы и для них соблюдалось указанное условие). В результате переразмещения всех элементов на вершине пирамиды опять окажется минимальное из всех чисел (без учета первого минимума). После этого новый минимум с вершины располагается на предпоследнем месте, а находившееся там число также перемещается на другое (соответствующее ему) место в новой пирамиде. При этом на вершине вновь окажется очередной минимальный элемент. В результате многократного повторения указанных действий вся последовательность чисел, взятых из пирамиды, будет упорядочена (правда, в обратном порядке).

Из сказанного следует, что нам необходимо уметь располагать какое-то число так, чтобы вновь получалась пирамида. Это можно сделать, поместив требуемое число сначала в вершину пирамиды, а затем, сравнивая это число с его «потомками», опускать его по пути, на котором находятся меньшие по сравнению с ним числа (которые одновременно поднимаются вверх). Процесс прекращается, когда встретятся «потомки», большие или равные размещаемому числу, т. е. происходит как бы его «просеивание» через числа в пирамиде (рис. 11.11а–в).

Разработаем процедуру «просеивания» некоторого числа в пирамиде. В ней мы будем использовать следующие величины: `i` — номер места, которое занимает размещаемое число в ходе просеивания;

j и j + 1 — номера «потомков» перемещаемого числа; вспо — величина, хранящая значение размещаемого числа; найдено — величина логического типа, определяющая факт нахождения места, соответствующего размещаемому элементу.

Числа в пирамиде мы будем рассматривать как элементы некоторого массива. Напомним, что должно обеспечиваться «просеивание» в пирамиде с переменным количеством чисел (в процессе сортировки некоторые числа в конце пирамиды могут не учитываться). Номер последнего учитываемого числа пирамиды обозначим как посл.

Вот процедура «просеивания» числа, находящегося в вершине пирамиды:

```
алг Просеивание1(арг цел посл, арг рез цел таб a[1:посл])
нач цел i, j, лог найдено, цел вспо
| i := 1; вспо := a[1]       |Запоминаем элемент с вершины
| найдено := нет
| j := 2 * i
| нц пока j <= посл и не найдено |Условие продолжения
| |                              |"просеивания"
| | |Определяем, по какой ветви будем опускаться
| | если a[j] > a[j + 1]
| | | то
| | |  j := j + 1
| | все
| | если вспо > a[j]
| | | то
| | |  a[i] := a[j]   |Поднимаем число a[j],
| | |  i := j         |а размещаемое число опускаем
| | |  j := 2 * I     |Номер нового потомка
| | | иначе           |Искомое место найдено
| | |  найдено := да
| | все
| кц
| |На найденном месте размещаем число с вершины
| a[i] := вспо
кон
```

Нетрудно видеть, что выше был рассмотрен случай, когда у каждого элемента имеются два «потомка» (естественно, кроме расположенных в нижнем ряду пирамиды), т. е. когда количество чисел в пирамиде нечетное. Незначительное усовершенствование дает возможность провести «просеивание» и при четном количестве чисел. Очевидно, что если j = посл, то это значит, что у i-го элемента есть только один «потомок», и в этом случае определять необходимую ветвь не нужно. С учетом этого фрагмент, относящийся к определению ветви,

по которой будет происходить дальнейшее «просеивание», должен быть несколько изменен:

```
...
| | если j < посл
| | | то  |У i-го элемента два "потомка"
| | |  |Определяем, по какой ветви будем опускаться
| | |  если a[j] > a[j + 1]
| | |  | то
| | |  |  j := j + 1
| | |  все
| | все
...
```

Выше уже говорилось, что числа в пирамиде можно рассматривать как элементы некоторого массива. Но в общем случае элементы массива, подлежащего сортировке, не обеспечивают соблюдение ранее указанного условия, т. е. исходный массив не является пирамидой в принятом понимании. Но если бы мы смогли преобразовать исходный массив в пирамиду, то проблема сортировки массива была бы решена!

Изящный способ построения пирамиды из элементов любого массива был предложен Р. У. Флойдом. Если дан массив a[1], a[2], ... , a[n], то ясно, что элементы a[n/2 + 1], a[n/2 + 2], ... , a[n] уже образуют пирамиду, поскольку у них нет «потомков», и поэтому можно считать, что требуемое условие для этих элементов не нарушается. Тогда можно поочередно брать элементы a[n/2], a[n/2 – 1], ... и при помощи «просеивания» размещать их на соответствующее место в имеющейся пирамиде. После того как будет «просеян» первый элемент исходного массива, процесс его преобразования в пирамиду будет завершен — можно проводить его сортировку с помощью процедуры, аналогичной процедуре Просеивание1. Последняя, правда, должна быть несколько модифицирована, поскольку «просеиваемый» в ходе построения пирамиды элемент не является первым, а номер последнего элемента в пирамиде равен n:

```
алг Просеивание(арг цел лев, посл, арг рез цел таб a [1:n])
нач цел i, j, лог найдено, цел всп
| i := лев; всп := a[лев]  |Размещаемый элемент
| найдено := нет
| j := 2 * i
| нц пока j <= посл и не найдено
| | если j < посл
| | | то
| | |  если a[j] > a[j + 1]
| | |  | то
| | |  |  j := j + 1
| | |  все
| | все
```

```
| | если всп > a[j]
| | | то
| | |  a[i] := a[j]
| | |  i := j
| | |  j := 2 * i
| | | иначе
| | |  найдено := да
| | все
| кц
| a[i] := всп
кон
```

Процедура сортировки массива описанным методом, который ее изобретатель Дж. Уильямс назвал *пирамидальной сортировкой*, оформляется в виде:

```
алг Пирамидальная_сортировка(арг рез цел таб a [1:n])
нач цел лев, посл, всп
| |Преобразуем исходный массив в пирамиду
| лев := div(n, 2)
| нц пока лев >= 1
| | Просеивание(лев, n, a)
| | лев := лев - 1
| кц
| |Сортируем пирамиду (массив)
| посл := n
| нц пока посл > 1
| | всп := a[1]              |Обмениваем местами
| | a[1] := a[посл]          |первый (минимальный) элемент
| | a[посл] := всп           |и элемент a[посл]
| | посл := посл - 1         |Уменьшаем значение посл
| | Просеивание(1, посл, a) |"Просеиваем" элемент с вершины
| кц
```

В заключение напомним, что рассмотренный метод сортирует массив в обратном порядке, и это необходимо учитывать при формулировании направления сортировки в программе, чтобы получить сортировку, требуемую в исходном условии задачи.

12. Динамическое программирование

Динамическое программирование[45] — это особый метод поиска оптимальных решений, специально приспособленный к так называемым «многошаговым», или «многоэтапным», операциям [7]. При этом «многошаговость» отражает реальное протекание процесса принятия решений во времени либо вводится в задачу искусственно за счет «расчленения» процесса принятия однократного решения на отдельные этапы.

Приведем несколько примеров задач, в которых рассматривается «многошаговая», т. е. представляющая собой последовательность «шагов» («этапов»), операция.

Задача 1. Нужно проложить путь, соединяющий пункты A и B (рис. 12.1), второй из которых лежит к северо-востоку от первого. Прокладка пути состоит из ряда шагов, где на каждом шаге мы можем двигаться только строго на север или строго на восток (т. е.

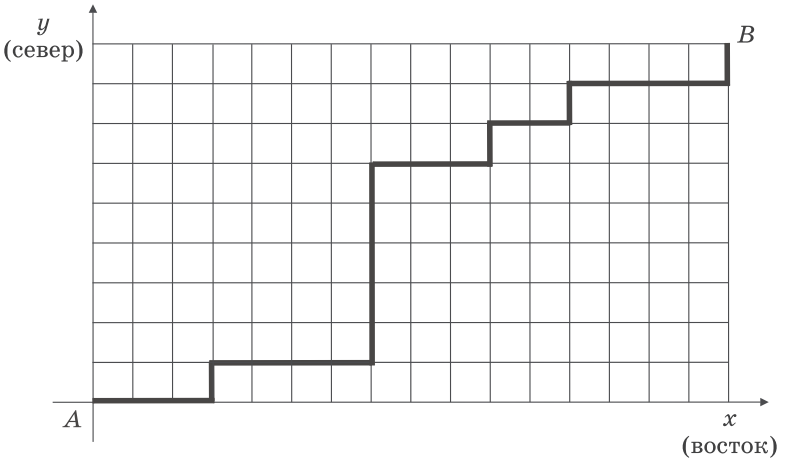


Рис. 12.1

[45] Термин «программирование» в названии этого метода происходит от слова «программа» в значении «план действий» и не связан с понятием «компьютерная программа». Поэтому динамическое программирование часто называют иначе — *динамическим планированием*. Происхождение же термина «динамический» здесь связано с использованием в задачах принятия решений выбранного метода периодически, через фиксированные промежутки времени.

любой путь из *A* в *B* представляет собой ступенчатую ломаную линию, отрезки которой параллельны одной из координатных осей). Затраты на прокладку каждого из подобных отрезков известны. Требуется проложить такой путь из *A* в *B*, при котором суммарные затраты минимальны.

Задача 2. Имеется определенный набор предметов — $П_1, П_2, ..., П_n$ (каждый — в единственном экземпляре). Известны их веса — $B_1, B_2, ..., B_n$ и стоимости — $C_1, C_2, ..., C_n$. Грузоподъемность машины равна Q. Какие из предметов нужно взять в машину, чтобы их суммарная стоимость (при суммарном весе не более Q) была максимальной?

Задача 3. Имеется некоторый объем денежных средств Q, который должен быть распределен между предприятиями $П_1, П_2, ..., П_n$. Каждое из предприятий $П_i$ при вложении в него каких-либо средств в объеме x приносит доход, зависящий от x, т. е. представляющий собой какую-то функцию $f_i(x)$. Все функции $f_i(x)$ ($i = 1, 2, .., n$) заданы (разумеется, все эти функции — неубывающие). Какие средства нужно выделить каждому предприятию, чтобы в сумме они дали максимальный доход?

В первой задаче операцией является строительство пути из точки *A* в точку *B*. Здесь шаги выделены уже в условии задачи: шагом является прокладывание очередного отрезка пути.

В постановке второй задачи нет упоминания о времени. Но процесс загрузки машины можно представить как состоящий из n шагов, считая за первый шаг принятие решения о том, брать или не брать первый предмет, за второй шаг — то же относительно второго предмета, и т. д.

В третьей задаче также можно операцию распределения средств мысленно разложить во времени в некоторую последовательность и рассматривать решение вопроса о вложении средств в предприятие $П_1$ как первый шаг, в предприятие $П_2$ — как второй шаг, и т. д.

Как решать подобные задачи? Можно перебрать все возможные варианты решения и выбрать среди них лучший. Например, в первой задаче рассмотреть все возможные варианты пути и выбрать тот, на котором затраты минимальны. Но можно выбирать лучший вариант пути шаг за шагом, на каждом этапе расчета оптимизируя только один шаг. Обычно такой способ оптимизации оказывается проще, чем предыдущий, особенно при большом количестве шагов[46].

---

[46] В ряде случаев при большом количестве шагов решение по первому способу вообще не может быть реализовано. В задаче 1, например, общее количество возможных вариантов пути равно числу сочетаний из $(N_1 + N_2)$ по N_1, где N_1 — количество отрезков от точки *A* до точки *B* в восточном направлении, N_2 — в северном. При $N_1 = 10$ и $N_2 = 10$ общее число вариантов равно 184756. При увеличении же N_1 и N_2 оно существенно возрастает. (В [8] показано, что при $N_1 = 30$ и $N_2 = 30$ компьютер, выполняющий миллион операций в секунду, рассмотрит все возможные варианты пути более чем за 100 000 лет!)

Такая идея постепенной, пошаговой оптимизации и лежит в основе динамического программирования. К ней мы вернемся чуть ниже, а пока заметим, что при решении задач методом динамического программирования используются следующие понятия [7]:

- система;
- шаг (этап);
- состояние системы;
- управление;
- выигрыш (либо проигрыш, который надо минимизировать).

Точные их определения дать достаточно сложно, поэтому мы просто проиллюстрируем их на примере перечисленных задач (табл. 12.1).

Таблица 12.1

Понятие	Задача 1	Задача 2	Задача 3
Система	Сооружаемый путь	Загружаемый автомобиль	Распределяемые средства
Шаг (этап)	Выбор очередного, i-го отрезка пути	Принятие решения о том, нужно ли брать в машину i-й предмет или нет (i = 1, 2, .. , n)	Определение объема средств, выделяемых i-му предприятию (i = 1, 2, .. , n)
Состояние	Точка, в которой может находиться конец уже построенного до i-го шага участка пути	Вес, который еще можно взять в машину после предыдущих шагов	Объем еще не вложенных средств
Управление	Строительство очередного участка в северном или восточном направлении	Решение: брать i-й предмет или не брать?	Объем средств, выделяемых i-му предприятию
Выигрыш (проигрыш)	Затраты на строительство	Стоимость загруженных в машину предметов	Доход от вложения средств

Вернемся теперь к идее пошаговой оптимизации. На первый взгляд она может показаться довольно тривиальной. В самом деле, чего, казалось бы, проще: если трудно оптимизировать процесс в целом, надо разбить его на ряд шагов и оптимизировать каждый шаг. Сделать это нетрудно — надо на этом шаге выбрать такое управление, чтобы эффективность данного шага была максимальна.

Нет, на самом деле все обстоит вовсе не так! Принцип динамического программирования вовсе не предполагает, что каждый шаг оптимизируется **отдельно**, независимо от других. Напротив, управ-

ление на каждом шаге должно выбираться дальновидно, с учетом его последствий в будущих этапах. Что толку, если мы выберем на данном шаге управление, при котором эффективность этого конкретного шага максимальна, но возникнет проигрыш на последующих шагах?

Но как выбрать оптимальное управление на том или ином шаге? Ведь на каждом из них (естественно, кроме первого) система может находиться (в общем случае) в разных возможных состояниях (в различных точках местности, на которой сооружается путь из *A* в *B*; с различным весом груза, который еще можно взять в машину, и т. п.), а в каждом из этих состояний в общем случае возможно несколько вариантов управления (строить очередной участок пути в северном или в восточном направлении; брать или не брать `i`-й предмет; вкладывать ли в `i`-е предприятие один из возможных объемов средств или не вкладывать вообще). Какое решение является оптимальным? Ведь мы даже не знаем точно, чем закончился предыдущий шаг!

Так как ответить на этот вопрос непросто, давайте сначала решим другую задачу — определим, какое управление является лучшим для **каждого** из состояний системы на том или ином шаге (иными словами, сделаем разные предположения о том, чем закончился предыдущий шаг). Такое управление мы назовем *условным оптимальным управлением* для этого состояния, а выигрыш, который дает это управление, — *условным оптимальным выигрышем* («условным» — потому что управление выбирается исходя из условия, что предыдущий шаг кончился так-то и так-то). Выбрать такой вариант управления можно, сравнив все возможные управления по следующему критерию: *сумма выигрыша от реализации того или иного управления на данном шаге и условного оптимального выигрыша в состоянии системы, в которое она перейдет в результате этого управления* (вы ведь помните о необходимости на каждом шаге принимать «дальновидные» решения?)[47].

Очевидно, что для использования такого критерия необходимо знать условные оптимальные выигрыши для всех состояний системы на следующем шаге. А из этого, в свою очередь, следует очень важный вывод: определение условных оптимальных параметров надо начинать с последнего шага, затем найти их для предпоследнего шага и т. д.

Для самого последнего шага сделать это не так уж сложно. Ведь для каждого состояния перед последним шагом известно, какое управление необходимо применить, — только то, которое приведет к конечному состоянию системы (путь должен закончиться в точке *B*; автомобиль должен быть полностью загружен; все средства должны быть вложены). Такое управление мы будем называть *вынужденным*. Запишем условные оптимальные показатели для последнего шага в

[47] Не вызывает сомнения факт, что именно по такому критерию пришлось бы выбирать оптимальный вариант управления в случае, когда какое-то состояние было бы *исходным*.

таблицу (табл. 12.2), где значения показателей показаны условно в виде символа «✓». То же самое мы будем делать и на других шагах.

Таблица 12.2

Состояние системы	Первый шаг		Второй шаг		...	Предпоследний шаг		Последний шаг	
	УОУ	УОВ	УОУ	УОВ		УОУ	УОВ	УОУ	УОВ
								✓	✓
								✓	✓
								...	...
								✓	✓

Примечание: УОУ — условное оптимальное управление, УОВ — условный оптимальный выигрыш.

Определив необходимые параметры для всех состояний последнего шага, можно перейти к рассмотрению ситуации перед предпоследним шагом. На этом этапе система также может находиться в нескольких состояниях, но для каждого состояния на этом шаге в общем случае возможен не один, а несколько вариантов управления (например, можно строить очередной участок пути в северном или в восточном направлении; брать или не брать i-й предмет; вкладывать в i-е предприятие один из возможных объемов средств или не вкладывать). Выбрать из них условное оптимальное управление мы сможем по критерию, который упоминался выше (ведь для каждого из возможных вариантов управления известно, в какое новое состояние перейдет система в результате, а для каждого состояния последнего шага уже найден условный оптимальный выигрыш). Выбор удобно проводить с использованием таблицы, подобной табл. 12.3.

Таблица 12.3

Возможное управление	Выигрыш от этого управления на данном шаге	Состояние, в которое перейдет система в результате управления	Условный оптимальный выигрыш для состояния в графе 3	Критерий сравнения вариантов управления (сумма граф 2 и 4)
1	2	3	4	5

Аналогичным способом можно определить условное оптимальное управление и условный оптимальный выигрыш для каждого состояния предпредпоследнего и других шагов, включая второй (соответствующая информация записывается в табл. 12.2).

После этого остается рассмотреть самый первый шаг. Его особенность состоит в том, что на нем система находится в одном, исходном, состоянии. Для этого состояния также можно выбрать условное оптимальное управление, которое на самом деле является уже не «условным», а безусловным. Заметим, кстати, что полученный для данного состояния оптимальный выигрыш есть наибольший выигрыш для *всех* шагов решения задачи.

Далее, применив оптимальное управление на первом шаге, мы переведем систему в какое-то новое состояние на втором шаге, для которого нам уже известен оптимальный вариант действий. Реализовав его, мы перейдем к следующему шагу, который также можем выполнить оптимальным способом, и т. д. до последнего шага, т. е. найдем решение задачи.

Как отмечается в [7], полное понимание основных положений динамического программирования, как правило, возможно только после рассмотрения ряда примеров, поэтому давайте перейдем к решению перечисленных ранее задач.

Решение задачи 1

Разделим расстояние от A до B в восточном направлении, скажем, на 7 частей, а в северном — на 5 (в принципе, такое дробление может быть сколь угодно мелким). Тогда любой путь из A в B состоит из $(7 + 5) = 12$ отрезков, направленных на восток или на север (рис. 12.2). Проставим на каждом из отрезков число, выражающее (в каких-то условных единицах) стоимость прокладки пути по этому отрезку. По требованию условия задачи необходимо выбрать такой путь из A в B, при котором сумма чисел, стоящих на отрезках, будет минимальной.

Будем рассматривать сооружаемый путь как управляемую систему, перемещающуюся (изменяющуюся) под влиянием управления (строительства очередного участка) из начального состояния A в конечное B. Состояние системы перед началом каждого шага мы будем характеризовать двумя целочисленными координатами: восточной (x) и северной (y), где $0 \le x \le 7$, $0 \le y \le 5$. Для каждого из состояний системы (узловой точки прямоугольной сетки на рис. 12.2) мы должны найти условное оптимальное управление: идти ли нам на север или на восток? Выбирается это управление так, чтобы стоимость всех оставшихся до конца шагов (включая данный) была минимальной. Эту стоимость мы договорились называть «условным оптимальным выигрышем» для данного состояния системы (хотя в данном случае это, конечно, не «выигрыш», а «проигрыш»).

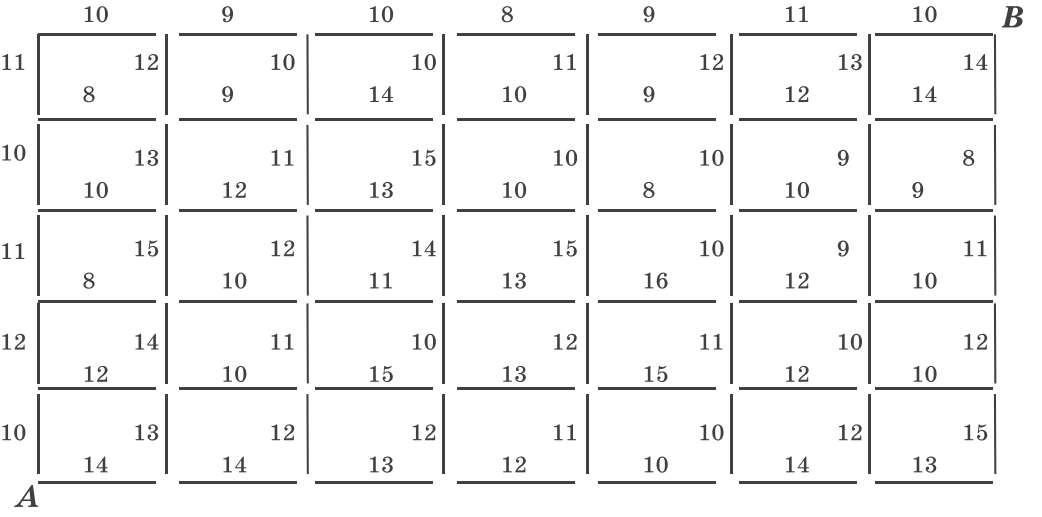


Рис. 12.2

Как указывалось выше, определение условных оптимальных параметров (управления и выигрыша/проигрыша) надо начинать с последнего, 12-го шага.

Рассмотрим отдельно правый верхний угол нашей прямоугольной сетки (рис. 12.3). Где мы можем находиться после 11-го шага? Только там, откуда за один (последний) шаг можно попасть в точку B, т. е. в одной из точек: B_1 или B_2. Если мы находимся в точке B_1, то у нас нет выбора (вынужденное управление): надо идти на восток, и это обойдется нам в 10 единиц. Запишем это число 10 в кружок у точки B_1, а оптимальное управление покажем короткой стрелкой, исходящей из B_1 и направленной на восток. Для точки B_2 управление тоже вынужденное (на север), а затраты равны 14; мы их запишем в кружок у точки B_2. Таким образом, условная оптимизация последнего шага сделана, условный оптимальный выигрыш для обоих состояний этого шага найден и записан в соответствующий кружок.

Теперь давайте оптимизировать предпоследний (11-й) шаг. После предпредпоследнего (10-го) шага мы могли оказаться в одной из точек: C_1, C_2, C_3 (рис. 12.4). Найдем для каждой из них условное оптимальное управление и условный оптимальный выигрыш.

Для точки C_1 управление вынужденное — идти на восток; обойдется это нам в итоге в 21 единицу (11 на данном шаге плюс 10 — условный оптимальный выигрыш для точки B_1, уже записанный в кружке). Это число 21 мы запишем в кружок при точке C_1.

Для точки C_2 управление уже не вынужденное: мы можем идти как на восток, так и на север. В первом случае мы затратим на данном шаге

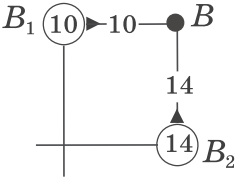


Рис. 12.3

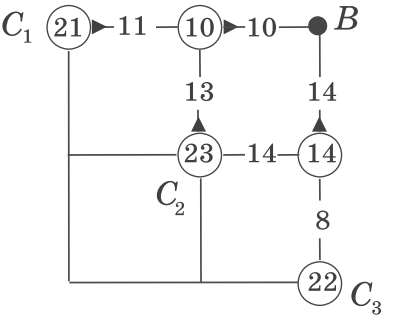


Рис. 12.4

14 единиц и от B_2 до конца пути — еще 14, или всего 28 единиц. Если же мы пойдем на север, то затратим 13 + 10 = 23 единицы. Значит, условное оптимальное управление в точке C_2 — идти на север (внимательный читатель, конечно, заметил, что это управление найдено по тому самому критерию, о котором говорилось ранее). Отмечаем его стрелкой, а число 23 записываем в кружок у точки C_2.

Для точки C_3 управление снова вынужденное — идти на север; обойдется это общим счетом в 22 единицы (ставим стрелку на север, а число 22 записываем в кружок у точки C_3).

Аналогично, «пятясь» от предпоследнего шага назад, найдем для каждой точки условное оптимальное управление, которое обозначим стрелкой, и условный оптимальный выигрыш, который запишем в соответствующем кружке. Вычисляется он так: расход на данном шаге складывается с **уже оптимизированным** расходом, записанным в кружке, к которому ведет стрелка. Таким образом, на каждом шаге мы оптимизируем только этот шаг, поскольку следующие за ним уже оптимизированы.

Конечный результат показан на рис. 12.5. Из него видно, что в какой бы из узловых точек мы ни находились, мы всегда знаем, куда идти (стрелка) и во что нам обойдется путь в итоге (число в кружке). Причем в кружочке у точки A записан оптимальный выигрыш (минимальные затраты) на все сооружение пути из A в B!

Теперь остается построить сам путь, ведущий из A в B, самым дешевым способом. Для этого нам надо только «слушаться стрелок», т. е. прочитать, что они предписывают делать на каждом шаге. Такая оптимальная траектория пути отмечена на рис. 12.5 кружочками серого цвета.

Как видим, для нахождения условного оптимального управления в каждой точке мы двигались по диагоналям, параллельным диагонали $B_1 - B_2$, причем каждая диагональ находилась юго-западнее предыдущей. Так, следующей после диагонали $B_1 - B_2$ была диагональ $C_1 - C_3$ и т. д.

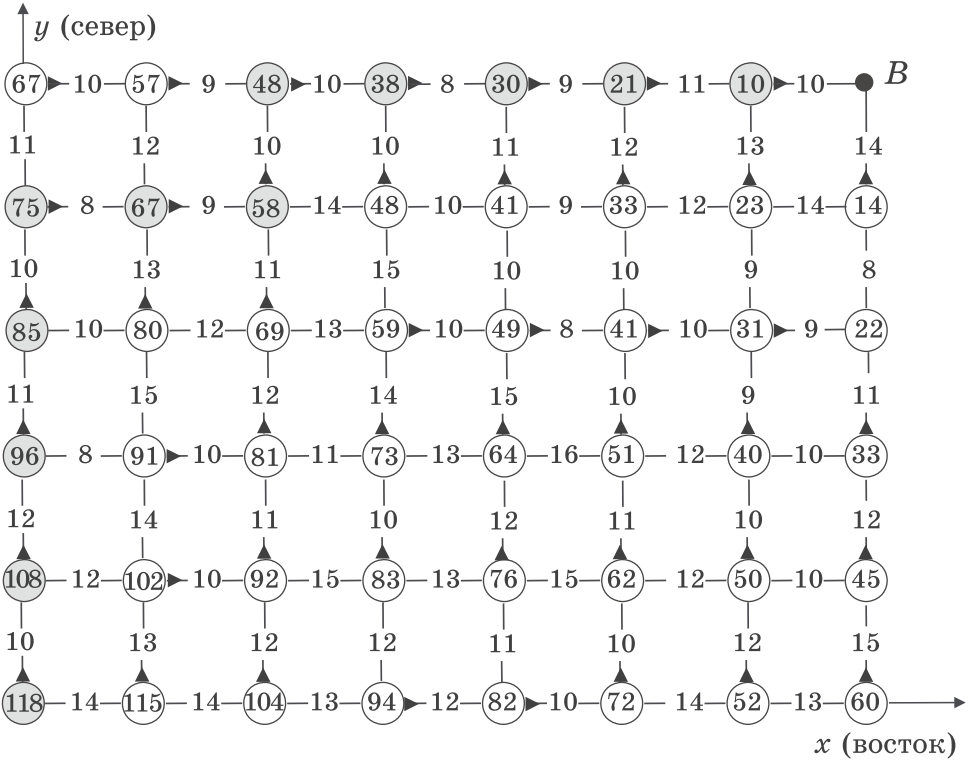


Рис. 12.5

Возможен и другой способ. Сначала найдем условное оптимальное управление в самых северных точках, учитывая, что здесь управление вынужденное — только на восток. Для этого мы движемся от точки *B* на запад до первой точки самой северной горизонтали. Затем найдем условное оптимальное управление в самых восточных точках. Здесь управление также вынужденное — только на север, и мы движемся от точки *B* на юг до первой точки самой восточной вертикали. Теперь можно найти условное оптимальное управление во всех остальных точках. Для этого мы движемся по горизонталям, начиная со второй с севера и кончая самой южной, а вдоль горизонтали — от второй точки с востока вплоть до самой западной. (Разумеется, тот же результат мы получим, двигаясь по вертикалям, начиная со второй с востока и вплоть до самой западной, а внутри вертикали — от второй точки с севера до самой южной.)

Наверное, движение по диагоналям выглядит более естественным, но для программной реализации более подходит второй способ. Обозначим количество отрезков на север и на восток как `Nсев` и `Nвост`. Значения условных оптимальных выигрышей для каждого состояния (числа в кружках на рис. 12.5) и соответствующее условное оп-

тимальное управление (направление отрезка пути в виде символов «в» и «с») будем записывать в массивах `УОВ[0:Nсев, 0:Nвост]` и `УОУ[0:Nсев, 0:Nвост]`. При этом первая строка каждого массива соответствует южной горизонтали, а первый столбец — западной вертикали, индексы точки A — это (0, 0), а точки B — (`Nсев`, `Nвост`). Данные о затратах на прокладку пути по восточным отрезкам (см. рис. 12.2) мы будем хранить в массиве `Свост[0:Nсев, 1:Nвост]`. Аналогично, затраты на прокладку северных отрезков будут храниться в массиве `Ссев[1:Nсев, 0:Nвост]`. При выборе условного оптимального управления из двух возможных вариантов по критерию, указанному ранее, используем величины `Ксев` и `Квост`. (Чтобы не усложнять работу с программой, значения элементов массивов задаются как константы.)

Вот листинг требуемой программы:

```
алг Задача_1
нач цел Nвост, Nсев, i, j, k, Ксев, Квост
| Nвост := 7 | Кол-во восточных отрезков
| Nсев := 5  | Кол-во северных отрезков
| сим таб УОУ[0:Nсев, 0:Nвост]
| цел таб УОВ[0:Nсев, 0:Nвост]
| цел таб Свост[0:Nсев, 1:Nвост], Ссев[1:Nсев, 0:Nвост]
| |Заполняем массив Свост
| Свост[0, 1] := 14; Свост[0, 2] := 14; Свост[0, 3] := 13
| Свост[0, 4] := 12; Свост[0, 5] := 10; Свост[0, 6] := 14
| Свост[0, 7] := 13; Свост[1, 1] := 12; Свост[1, 2] := 10
| Свост[1, 3] := 15; Свост[1, 4] := 13; Свост[1, 5] := 15
| Свост[1, 6] := 12; Свост[1, 7] := 10; Свост[2, 1] := 8
| Свост[2, 2] := 10; Свост[2, 3] := 11; Свост[2, 4] := 13
| Свост[2, 5] := 16; Свост[2, 6] := 12; Свост[2, 7] := 10
| Свост[3, 1] := 10; Свост[3, 2] := 12; Свост[3, 3] := 13
| Свост[3, 4] := 10; Свост[3, 5] := 8;  Свост[3, 6] := 10
| Свост[3, 7] := 9;  Свост[4, 1] := 8;  Свост[4, 2] := 9
| Свост[4, 3] := 14; Свост[4, 4] := 10; Свост[4, 5] := 9
| Свост[4, 6] := 12; Свост[4, 7] := 14; Свост[5, 1] := 10
| Свост[5, 2] := 9;  Свост[5, 3] := 10; Свост[5, 4] := 8
| Свост[5, 5] := 9;  Свост[5, 6] := 11; Свост[5, 7] := 10
| |Заполняем массив Ссев
| Ссев[1, 0] := 10; Ссев[2, 0] := 12; Ссев[3, 0] := 11
| Ссев[4, 0] := 10; Ссев[5, 0] := 11; Ссев[1, 1] := 13
| Ссев[2, 1] := 14; Ссев[3, 1] := 15; Ссев[4, 1] := 13
| Ссев[5, 1] := 12; Ссев[1, 2] := 12; Ссев[2, 2] := 11
| Ссев[3, 2] := 12; Ссев[4, 2] := 11; Ссев[5, 2] := 10
| Ссев[1, 3] := 12; Ссев[2, 3] := 10; Ссев[3, 3] := 14
| Ссев[4, 3] := 15; Ссев[5, 3] := 10; Ссев[1, 4] := 11
| Ссев[2, 4] := 12; Ссев[3, 4] := 15; Ссев[4, 4] := 10
```

```
| Сев[5, 4] := 11; Сев[1, 5] := 10; Сев[2, 5] := 11
| Сев[3, 5] := 10; Сев[4, 5] := 10; Сев[5, 5] := 12
| Сев[1, 6] := 12; Сев[2, 6] := 10; Сев[3, 6] := 9
| Сев[4, 6] := 9;  Сев[5, 6] := 13; Сев[1, 7] := 15
| Сев[2, 7] := 12; Сев[3, 7] := 11; Сев[4, 7] := 8
| Сев[5, 7] := 14
| |Заполняем массивы УОУ и УОВ
| |а) их последнюю строку (самые северные точки)
| УОВ[Nсев,Nвост] := 0
| нц для j от Nвост — 1 до 0 шаг —1
| | УОУ[Nсев, j] := "в" |Управление вынужденное
| | УОВ[Nсев, j] := Свост[Nсев, j + 1] + УОВ[Nсев, j + 1]
| кц
| |б) их правый столбец (самые восточные точки)
| нц для i от Nсев — 1 до 0 шаг —1
| | УОУ[i, Nвост] := "с" |Управление вынужденное
| | УОВ[i, Nвост] := Сев[i + 1, Nвост] + УОВ[i + 1, Nвост]
| кц
| |в) остальные элементы
| |Движемся по строкам от предпоследней строки к первой,
| |а вдоль строки — от предпоследнего элемента к первому
| нц для i от Nсев — 1 до 0 шаг —1
| | нц для j от Nвост — 1 до 0 шаг —1
| | | |Выбираем условное оптимальное управление
| | | |Значение критерия для управления "на север"
| | | Ксев := УОВ[i + 1, j] + Сев[i + 1, j]
| | | |Значение критерия для управления "на восток"
| | | Квост := УОВ[i, j + 1] +Свост[i, j + 1]
| | | |Выбираем условное оптимальное управление
| | | если Квост > Ксев
| | | | то |Идем на север
| | | |  УОУ[i, j] := "с"
| | | |  УОВ[i, j] := Ксев
| | | | иначе |Идем на восток
| | | |  УОУ[i, j] := "в"
| | | |  УОВ[i, j] := Квост
| | | все
| | кц
| кц
| |Печатаем результаты
| |Оптимальный выигрыш – в элементе УОВ[0, 0]
| |(в кружке у точки А)
| вывод нс, "Минимальные затраты на сооружение пути: ", УОВ[0, 0]
| вывод нс, "Оптимальные направления отрезков пути:"
| |Начальное направление
```

```
| |записано в элементе УОУ[0, 0]
| |(стрелка на кружке у точки А)
| i := 0; j := 0
| нц для k от 1 до Nсев + Nвост
| | вывод УОУ[i, j]," "
| | |Переходим к следующей точке
| | если УОУ[i, j] = "с"
| | | то
| | |  i := i + 1
| | | иначе
| | |  j := j + 1
| | все
| кц
кон
```

Выполнив программу с исходными данными, представленными на рис. 12.2, можно обнаружить, что оптимальный выигрыш (минимальные затраты) на сооружение пути из *А* в *В* равен 118, а достигается он при следующем наборе отрезков пути:

```
с, с, с, с, в, в, с, в, в, в, в, в
```

Интересно — какой результат был бы получен, если бы мы решали задачу наивным способом, о котором говорилось выше, т. е. выбирая на каждом шаге, начиная с первого, самое выгодное для данного шага направление? Из рис. 12.2 видно, что таким способом мы получили бы путь, состоящий из отрезков:

```
с, с, в, в, в, в, с, в, в, в, с, с
```

Подсчитаем расходы для такого пути. Они будут равны:

$$10 + 12 + 8 + 10 + 11 + 13 + 15 + 8 + 10 + 9 + 8 + 14 = 128,$$

что больше, чем найденное методом динамического программирования значение 118. В данном случае разница не очень велика, но в других ситуациях она может быть существеннее.

Отметим и еще один момент. В ходе условной оптимизации мы можем столкнуться со случаем, когда оба возможных варианта управления для какого-либо состояния (какой-либо узловой точки) являются оптимальными, т. е. приводят к одинаковому расходу средств от этой точки до конца пути. Например, в точке с координатами (1, 4) оба варианта управления являются оптимальными и дают итоговый расход, равный 67. Из них мы можем выбрать любой (в нашем случае мы выбрали «в», но с тем же успехом могли бы выбрать «с» — общий выигрыш был бы тем же). Такие случаи неоднозначного выбора оптимального управления часто встречаются при использовании метода динамического программирования.

Решение задачи 2

Обсудим методику решения этой задачи для следующих исходных данных: грузоподъемность машины Q = 35, количество предметов n = 6. Сведения об этих предметах приведены в таблице:

№	1	2	3	4	5	6
Вес	4	7	11	12	16	20
Стоимость	7	10	15	20	27	34

Напомним, что состоянием системы в данной задаче является вес, который еще можно взять в машину после предыдущих шагов (остаточная грузоподъемность). Грузоподъемность машины и вес каждого предмета считаются целыми числами, поэтому возможные состояния принимают все целочисленные значения от 0 до Q.

Управление в каждом состоянии и для каждого шага заключается в ответе на вопрос: берем ли мы данный предмет или не берем, что кодируется, соответственно, единицей и нулем.

Для решения задачи заполним табл. 12.4 (назовем ее главной; она является аналогом табл. 12.2).

В графе S записывается возможное состояние данной системы (вес, который еще можно взять в машину). В графе $УОУ_i$ будем записывать условное оптимальное управление на i-м шаге (1 — «берем»; 0 — «не берем»). В графе $УОВ_i$ записывается условный оптимальный выигрыш на i-м шаге.

Начнем оптимизацию с последнего, 6-го шага.

Вес 6-го предмета равен 20, поэтому если вес, который можно взять в машину, на 6-м шаге меньше 20, то 6-й предмет взять невозможно. Значит, в графе $УОУ_6$ в строках, где $S < 20$, ставим «0», а в остальных случаях ($S \geq 20$) ставим «1». В графе же $УОВ_6$ ставим «0» при $УОУ_6 = 0$ и 34 (стоимость 6-го предмета) при $УОУ_6 = 1$. Напомним, что такое управление мы условились называть *вынужденным*.

Теперь оптимизируем 5-й шаг.

Вес 5-го предмета равен 16, и если вес, который еще можно взять в машину, меньше 16, то мы не можем взять 5-й предмет (это также вынужденное управление), и в графе $УОУ_5$ в строках, где $S < 16$, ставим нуль. Соответствующее значение в графе УОВ для этих строк также равно нулю, поскольку после каждого из таких управлений система останется в том же состоянии, для которого на шаге 6 значение $УОУ_6 = 0$. А вот в строках, где $S \geq 16$, условное оптимальное управление надо будет выбирать из двух возможных — берем 5-й предмет или не берем.

Таблица 12.4

Состояние, s	1-й шаг		2-й шаг		3-й шаг		4-й шаг		5-й шаг		6-й шаг	
	$УОУ_1$	$УОВ_1$	$УОУ_2$	$УОВ_2$	$УОУ_3$	$УОВ_3$	$УОУ_4$	$УОВ_4$	$УОУ_5$	$УОВ_5$	$УОУ_6$	$УОВ_6$
0	—	—	0	0	0	0	0	0	0	0	0	0
1	—	—	0	0	0	0	0	0	0	0	0	0
2	—	—	0	0	0	0	0	0	0	0	0	0
…	…	…	…	…	…	…	…	…	…	…	…	…
6	—	—	0	0	0	0	0	0	0	0	0	0
7	—	—	1	10	0	0	0	0	0	0	0	0
8	—	—	1	10	0	0	0	0	0	0	0	0
9	—	—	1	10	0	0	0	0	0	0	0	0
10	—	—	1	10	0	0	0	0	0	0	0	0
11	—	—	0	15	1	15	0	0	0	0	0	0
12	—	—	0	20	0	20	1	20	0	0	0	0
13	—	—	0	20	0	20	1	20	0	0	0	0
14	—	—	0	20	0	20	1	20	0	0	0	0
15	—	—	0	20	0	20	1	20	0	0	0	0
16	—	—	0	27	0	27	0	27	1	27	0	0
17	—	—	0	27	0	27	0	27	1	27	0	0
18	—	—	0	27	0	27	0	27	1	27	0	0
19	—	—	1	30	0	27	0	27	1	27	0	0
20	—	—	0	34	0	34	0	27	0	34	1	34
21	—	—	0	34	0	34	0	34	0	34	1	34
22	—	—	0	34	0	34	0	34	0	34	1	34
23	—	—	1	37	1	35	0	34	0	34	1	34
24	—	—	1	37	1	35	0	34	0	34	1	34
25	—	—	1	37	1	35	0	34	0	34	1	34
26	—	—	1	37	1	35	0	34	0	34	1	34
27	—	—	1	44	1	42	0	34	0	34	1	34
28	—	—	0	47	0	47	1	47	0	34	1	34
29	—	—	0	47	0	47	1	47	0	34	1	34
30	—	—	0	47	0	47	1	47	0	34	1	34
31	—	—	0	49	0	49	1	47	0	34	1	34
32	—	—	0	54	0	54	1	54	0	34	1	34
33	—	—	0	54	0	54	1	54	0	34	1	34
34	—	—	0	54	0	54	1	54	0	34	1	34
35	0	57	1	57	0	54	1	54	0	34	1	34

Методику такого выбора проиллюстрируем на примере состояния $S = 16$ с помощью табл. 12.5 — аналога табл. 12.3.

Таблица 12.5

Возможное управление	Выигрыш от этого управления на данном шаге	Состояние, в которое перейдет система в результате управления	Условное оптимальное управление для этого состояния на 6-м шаге	Критерий выбора
Не берем (0)	0	16	0	0+0=0
Берем (1)	27	0	0	27+0=27

Значит, для состояния S = 16 условное оптимальное управление заключается в том, что 5-й предмет мы берем. Условный оптимальный выигрыш для такого управления равен 27, что мы и запишем в графу $УОВ_5$ главной таблицы.

Аналогичный результат можно получить и для состояний S = 17, S = 18, S = 19. Для состояния же S = 20 исходная информация для выбора условных оптимальных параметров приведена в табл. 12.6.

Таблица 12.6

Возможное управление	Выигрыш от этого управления на данном шаге	Состояние, в которое перейдет система в результате управления	Условное оптимальное управление для этого состояния на 6-м шаге	Критерий выбора
Не берем (0)	0	20	34	0+34=34
Берем (1)	27	0	0	27+0=27

Из этой таблицы следует, что условное оптимальное управление для этого состояния состоит в том, что 5-й предмет не берется, а условный оптимальный выигрыш равен 34. Запишем это в графы $УОУ_5$ и $УОВ_5$ главной таблицы.

Аналогично вычислим $УОУ_5$ и $УОВ_5$ для всех остальных элементов главной таблицы по i = 2 включительно.

Ситуация же для i = 1 проще: в самом начале машина пуста и S = Q = 35, а вычисления проводятся только для строки S = 35 (табл. 12.7).

Таблица 12.7

Возможное управление	Выигрыш от этого управления на данном шаге	Состояние, в которое перейдет система в результате управления	Условное оптимальное управление для этого состояния на 2-м шаге	Критерий выбора
Не берем (0)	0	35	57	0+57=57
Берем (1)	7	31	47	7+47=54

При этом автоматически дается ответ на вопрос, какова максимальная стоимость предметов, которыми можно загрузить машину. Ответ находится на пересечении 35-й строки и столбца $УОВ_1$ и равен 57.

Для получения ответа на вопрос, какими предметами надо нагрузить машину, чтобы их стоимость была максимальной, надо просмотреть таблицу справа налево:

- так как $УОУ_1 = 0$, то первый предмет мы не берем;
- так как мы не взяли первый предмет, то «свободного» веса так и осталось 35; находим элемент на пересечении строки `S` = 35 и столбца $УОУ_2$: он равен 1, а значит, второй предмет мы берем в машину;
- так как мы взяли второй предмет, а его масса равна 7, то после его погрузки «свободного» веса останется 28. Находим элемент на пересечении строки `S` = 28 и столбца $УОУ_3$: он равен 0, а значит, третий предмет мы не берем и «свободного» веса так и останется 28;

и т. д. В главной таблице полученные элементы выделены серым цветом.

При программной реализации описанного метода информацию из главной таблицы мы будем хранить следующим образом. Значения вариантов условного оптимального управления будем записывать в двумерный массив `УОУ[0:Q, 1:n]`. А для условных оптимальных выигрышей использовать массив с множеством столбцов необходимости нет — можно заметить, что после оптимизации `i`-го шага и заполнения графы $УОВ_i$ графа $УОВ_{i+1}$ станет ненужной и будет лишь занимать место в памяти. Поэтому для хранения условных оптимальных выигрышей, уже найденных на шаге `Nшага`, мы используем одномерный массив `УОВ[0:Q]`, а при переходе к предыдущему шагу (`Nшага` – 1) условные оптимальные выигрыши будем заносить в одномерный массив `УОВнов[0:Q]`, где после заполнения массива `УОВнов` значения его элементов переписываются в массив `УОВ` и используются для оптимизации на другом шаге.

Вес и стоимость предметов мы запишем в массивы с именами, соответственно, `B` и `C`, а состояние системы (вес, который еще можно взять в машину после предыдущих шагов) обозначим как `Ост`.

Как мы помним, поиск вариантов условного оптимального управления и выигрышей надо начинать с последнего шага. Здесь управление вынужденное: если вес `n`-го предмета не больше остаточной грузоподъемности `Ост`, то предмет берется, а в противном случае — не берется, поскольку это невозможно.

Запишем сказанное более формально:

```
если B[n] > Ост
| то
|  УОУ[Ост, n] := 0; УОВ[Ост] := 0
| иначе
|  УОУ[Ост, n] := 1; УОВ[Ост] := C[n]
все
```

Отсюда находим указанные величины при всех возможных значениях состояния Ост.

Теперь приступаем к нахождению условного оптимального управления и выигрышей на всех промежуточных шагах от предпоследнего до второго. Пусть на некотором шаге (Nшага + 1) условные оптимальные выигрыши уже найдены и хранятся в массиве УОВ. Определим их для предыдущего шага Nшага. По-прежнему, если вес предмета с номером Nшага шага больше остаточной грузоподъемности Ост, то предмет не берется; иначе выясняем, нужно ли его брать. Находим условные выигрыши (критерии выбора) Кберем, Кнет для обоих возможных вариантов управления и выбираем тот, при котором условный выигрыш больше.

Вот более подробная запись сказанного:

```
если B[Nшага] > Ост
| то
|  УОУ[Ост, Nшага] := 0; УОВнов[Ост] := 0
| иначе
|  |Выбираем лучший вариант
|  |Критерий для 1-го варианта
|  Кнет := УОВ[Ост]
|  |Критерий для 2-го варианта
|  Кберем := C[Nшага] + УОВ[Ост - B[Nшага]]
|  если Кберем > Кнет
|  | то
|  |  УОУ[Ост, Nшага] := 1
|  |  УОВнов[Ост] := Кберем
|  | иначе
|  |  УОУ[Ост, Nшага] := 0
|  |  УОВнов[Ост] := Кнет
|  все
все
```

Выполняем эти операции при всех возможных значениях состояния Ост, после чего копируем массив УОВнов в УОВ.

Наконец, рассматриваем первый шаг. Для него существует лишь одно состояние Q. Аналогично предыдущему, находим оптимальное управление на первом шаге: УОУ[Q, 1].

Осталось определить оптимальное управление на последующих шагах. Для этого, зная оптимальное управление и состояние на предыдущем шаге, находим состояние на очередном шаге и берем условное оптимальное управление этого состояния и шага (см. выше).

Программа:

```
цел Q, n |Глобальные параметры задачи
алг Задача_2
нач цел Nшага, Ост, цел таб B[1:n], C[1:n], УОУ[0:Q, 1:n],
|                            УОВ[0:Q], УОВнов[0:Q]
| Q := 35 |Грузоподъемность автомобиля
| n := 6  |Количество предметов
| |Заполняем массивы B и C
| B[1]:=4; B[2]:=7; B[3]:=11; B[4]:=12; B[5]:=16; C[6]:=20
| C[1]:=7; C[2]:=10; C[3]:=15; C[4]:=20; C[5]:=27; C[6]:=34
| |Определяем условные оптимальные параметры
| |Последний шаг
| нц для Ост от 0 до Q
| | если B[n] > Ост
| | | то |взять последний предмет нельзя
| | |  УОУ[Ост,n] := 0; УОВ[Ост] := 0
| | | иначе |можно взять последний предмет
| | |  УОУ[Ост,n] := 1; УОВ[Ост] := C[n]
| | все
| кц
| |Остальные шаги (кроме первого)
| нц для Nшага от n - 1 до 2 шаг -1
| | нц для Ост от 0 до Q
| | | если B[Nшага] > Ост
| | | | то     |Взять предмет с номером Nшага нельзя
| | | |  УОУ[Ост, Nшага] := 0; УОВнов[Ост] := 0
| | | | иначе   |можно
| | | |  |Выбираем лучший вариант
| | | |  Кнет := УОВ[Ост]
| | | |  Кберем := C[Nшага] + УОВ[Ост - B[Nшага]]
| | | |  если Кберем > Кнет
| | | |  | то
| | | |  |  УОУ[Ост, Nшага] := 1
| | | |  |  УОВнов[Ост] := Кберем
| | | |  | иначе
| | | |  |  УОУ[Ост, Nшага] := 0
| | | |  |  УОВнов[Ост] := Кнет
| | | |  все
| | | все
| | кц
```

```
| | |Переписываем массив УОВнов в массив УОВ
| | нц для Ост от 0 до Q
| | | УОВ[Ост] := УОВнов[Ост]
| | кц
| кц
| |Первый шаг
| |Возможное состояние одно - можно взять Q единиц груза
| |Находим оптимальное управление (безусловное)
| Кнет := УОВ[Q]
| Кберем := С[1] + УОВ[Q - В[1]]
| если Кберем > Кнет
| | то
| |  УОУ[Q,1] := 1
| |  УОВнов[Q] := Кберем
| | иначе
| |  УОУ[Q, 1] := 0
| |  УОВнов[Q] := Кнет
| все
| |Печатаем максимальный выигрыш
| вывод нс, "Максимальная суммарная стоимость: ", УОВнов[Q]
| |Определяем вариант управления, при котором он достигается
| вывод нс, "Оптимальное управление:"
| Ост := Q
| нц для Nшага от 1 до n
| | вывод нс, Nшага, "-й предмет "
| | если УОУ[Ост, Nшага] = 1
| | | то
| | |  вывод "берем"
| | |  Ост := Ост - В[Nшага]
| | | иначе
| | |  вывод "не берем"
| | все
| кц
кон
```

Выполнив программу, можно получить, что оптимальным является набор из 2-го, 4-го и 5-го предметов, при котором суммарная стоимость равна 57 (что соответствует данным главной табл. 12.4). Предлагаю читателю самому убедиться, что любой другой набор предметов с суммарным весом не более 35 обладает суммарной стоимостью, не большей 57.

Усовершенствуем программу. Так как в ней имеется два похожих фрагмента, связанных с выбором оптимальных параметров на первом шаге и на остальных шагах, кроме последнего, то соответствующие действия целесообразно оформить в виде процедуры ВыборВарианта:

```
алг ВыборВарианта(арг цел Ост, Nшага,
   цел таб B[1:n], C[1:n], УОВ[0:Q], арг рез цел УОУ, УОВн)
| |Выбор на шаге Nшага условного оптимального управления
| |из двух возможных: брать или не брать предмет,
| |а также определение условного оптимального выигрыша
нач цел Кберем, Кнет
| Кнет := УОВ[Ост]
| Кберем := C[i] + УОВ[Ост - B[i]]
| если Кнет > Кберем
| | то
| |  УОУ := 0; УОВн := Кнет
| | иначе
| |  УОУ := 1; УОВн := Кберем
| все
кон
```

Для шагов с номером от 2 до (n – 1) вызов этой процедуры должен осуществляться следующим образом:

```
ВыборВарианта(Ост, Nшага, B, C, УОВ, УОУ[Ост, Nшага], УОВнов[Ост])
```

а для первого шага:

```
ВыборВарианта(Q, 1, B, C, УОВ, УОУ[Q,1], УОВнов[Q])
```

Убедитесь, что использование этой процедуры существенно уменьшает размер программы и улучшает ее «читаемость».

Решение задачи 3

Прежде всего отметим, что исходный запас средств Q, а также средства, вложенные в предприятия, являются целыми величинами. Тогда любое возможное состояние, которое (см. табл. 12.1) характеризуется объемом еще не вложенных средств (назовем эту величину Ост), принимает целые значения в диапазоне от 0 до Q.

Для каждого возможного состояния возможно несколько вариантов управления: вложение в то или иное предприятие 0, 1, 2, ..., Ост единиц средств. Этим данная задача существенно отличается от предыдущих, где имелось лишь два варианта управления. С другой стороны, задача 3 довольно близка к предыдущей, второй задаче. Поэтому мы воспользуемся уже введенными обозначениями тех величин, которые являются общими для них: УОУ[Ост, Nшага] — условное оптимальное управление в состоянии Ост на шаге Nшага, УОВ[Ост] и УОВнов[Ост] — условные оптимальные выигрыши в состоянии Ост на шаге Nшага и (Nшага – 1).

Новым понятием здесь является *функция дохода* от вложения средств для данного предприятия. Поскольку вложенные средства

принимают все целые значения от 0 до Q, то функции дохода — табличные.

Объединим все эти таблицы в двумерный массив Доход[1:N, 0:Q]. Тогда Доход[Nшага, X] — это доход от вложения X единиц средств в предприятие Nшага.

Напомним, что общий подход к решению подобных задач — последовательно заполнять таблицу типа табл. 12.2; ее реализация для данной задачи приведена в табл. 12.8. Здесь принято, что количество предприятий N = 5, исходный запас Q = 10, а матрица дохода Доход имеет вид:

$$\begin{pmatrix} 0.0 & 0.5 & 1.0 & 1.4 & 2.0 & 2.5 & 2.8 & 3.0 & 3.0 & 3.0 & 3.0 \\ 0.0 & 0.1 & 0.5 & 1.2 & 1.8 & 2.5 & 2.9 & 3.5 & 3.5 & 3.5 & 3.5 \\ 0.0 & 0.6 & 1.1 & 1.2 & 1.4 & 1.6 & 1.7 & 1.8 & 1.8 & 1.8 & 1.8 \\ 0.0 & 0.3 & 0.6 & 1.3 & 1.4 & 1.5 & 1.5 & 1.5 & 1.5 & 1.5 & 1.5 \\ 0.0 & 1.0 & 1.2 & 1.3 & 1.3 & 1.3 & 1.3 & 1.3 & 1.3 & 1.3 & 1.3 \end{pmatrix}$$

Таблица 12.8

Состояние	1-й шаг		2-й шаг		3-й шаг		4-й шаг		5-й шаг	
	УОУ	УОВ	УОУ	УОВ	УОУ	УОВ	УОУ	УОВ	УОУ	УОВ
0	—	—	0	0	0	0	0	0	0	0
1	—	—	0	1	0	1	0	1	1	1
2	—	—	0	1.6	1	1.6	1	1.3	2	1.2
3	—	—	0	2.1	2	2.1	2	1.6	3	1.3
4	—	—	0	2.4	2	2.4	3	2.3	4	1.3
5	—	—	5	2.9	1	2.9	3	2.5	5	1.3
6	—	—	5	3.5	2	3.4	4	2.6	6	1.3
7	—	—	5	4.1	2	3.6	4	2.7	7	1.3
8	—	—	5	4.6	2	3.7	5	2.8	8	1.3
9	—	—	7	5.1	4	3.9	5	2.8	9	1.3
10	0	5.6	7	5.6	5	4.1	5	2.8	10	1.3

Значения, соответствующие полученному оптимальному управлению, здесь также выделены серым цветом.

При составлении табл. 12.8 оптимальный вариант удобно выбирать с помощью таблицы типа табл. 12.3. В нашей задаче продемонстрируем ее использование на примере определения условного оптимального управления и условного оптимального выигрыша на третьем шаге в состоянии, когда запас средств равен 7 (табл. 12.9).

Естественно, что к этому моменту 4-й шаг уже оптимизирован, т. е. заполнены соответствующие столбцы табл. 12.8. Из табл. 12.9 следует, что условное оптимальное управление в рассматриваемом состоянии равно двум единицам средств, а соответствующий условный оптимальный выигрыш — 3.6. Эти значения и записываются в табл. 12.8.

Таблица 12.9

Возможное управление	Выигрыш от этого управления на данном шаге	Состояние, в которое перейдет система в результате управления	Условный оптимальный выигрыш для состояния в графе 3	Критерий сравнения вариантов управления (сумма граф 2 и 4)
1	2	3	4	5
0	0	7	2.7	2.7
1	0.6	6	2.6	3.2
2	1.1	5	2.5	3.6
3	1.2	4	2.3	3.5
4	1.4	3	1.6	3.0
5	1.6	2	1.3	2.9
6	1.7	1	1.0	2.7
7	1.8	0	0	1.8

Переходим к систематическому изложению алгоритма. В нем существенную роль будет играть процедура ВыборВарианта для определения условного оптимального управления и условного оптимального выигрыша в состоянии Ост на шаге Nшага, аналогичная разработанной при решении задачи 2, но учитывающая то обстоятельство, что при выборе оптимального варианта требуется проверить все возможные значения управления от 0 до Ост:

```
алг ВыборВарианта(арг цел Ост, Nшага, вещ таб Доход[1: N, 0:Q],
    УОВ[0:Q], арг рез цел УОУ, вещ УОВн)
нач цел ВУ, ОУ, вещ Кву, Кмакс
| |ВУ – возможное управление
| |ОУ – оптимальное управление среди рассмотренных
| |Кву – значение критерия для выбора оптимального варианта
| |Кмакс – максимальное значение критерия
| Кмакс := Доход[Nшага, 0] + УОВ[Ост]
| нц для ВУ от 1 до Ост
| | Кву := Доход[Nшага, ВУ] + УОВ[Ост - ВУ]
| | если Кву > Кмакс
| | | то
| | |  ОУ := ВУ
| | |  Кмакс := Sx
| | все
| кц
| УОУ := ОУ
| УОВн := Кмакс
кон
```

В процедуре сначала принято ОУ := 0; Кмакс := Доход[Nшага,0]+ УОВ[Ост], а затем рассмотрены все значения управления ВУ от 1 до Ост.

Поиск оптимального варианта мы начинаем, как обычно при использовании динамического программирования, с последнего шага (рассмотрения вариантов вложения средств в 5-е предприятие). Здесь управление вынужденное — для любого состояния системы Ост мы вкладываем в 5-е предприятие все оставшиеся средства, а условный оптимальный выигрыш определяем из массива Доход:

```
нц для Ост от 0 до Q
| УОУ[Ост,n] := Ост
| УОВ[Ост] := Доход[n, Ост]
кц
```

Затем мы приступаем к нахождению вариантов условного оптимального управления и выигрышей на всех промежуточных шагах от предпоследнего до второго. Пусть при некотором (Nшага + 1) они уже найдены, при этом условные оптимальные выигрыши хранятся в массиве УОВ. Определим их для предыдущего шага Nшага, для чего выполняем обращение к процедуре:

```
ВыборВарианта(Ост,Nшага,Доход,УОВ,УОУ[Ост,Nшага],УОВнов[Ост])
```

при всех значениях состояния Ост от 0 до Q, а после этого копируем массив УОВнов в УОВ.

На первом шаге состояние единственно и равно Q. Поэтому, выполнив обращение:

```
ВыборВарианта(Q, 1, Доход, УОВ, УОУ[Q, 1], УОВнов[Q])
```

мы находим оптимальное управление для первого предприятия и максимальный доход.

Осталось определить оптимальное управление на последующих шагах. Для этого, зная состояние на предыдущем шаге и оптимальное управление, находим их разность. Состояние на очередном шаге равно этой разности. Отсюда получаем условное оптимальное управление для полученного состояния и очередного шага.

Прежде чем написать соответствующую программу, с целью ее упрощения, все исходные данные (исходный запас ресурсов, количество предприятий и функции дохода для каждого предприятия) примем известными заранее.

```
цел Q, N |Глобальные параметры задачи
алг Задача 3
нач цел Nшага, Ост, вещ таб Доход[1:n, 0:Q], УОВ[0:Q],
|    УОВнов[0:Q], цел таб УОУ[0:Q, 1:n]
| Q := 10 |Исходный запас средств
| n := 5  |Количество предприятий
```

```
| |Заполняем массив функций доходов Доход
| Доход[1, 0] := 0.0; Доход[2, 0] := 0.0; Доход[3, 0] := 0.0
| Доход[4, 0] := 0.0; Доход[5, 0] := 0.0; Доход[1, 1] := 0.5
| Доход[2, 1] := 0.1; Доход[3, 1] := 0.6; Доход[4, 1] := 0.3
| Доход[5, 1] := 1.0; Доход[1, 2] := 1.0; Доход[2, 2] := 0.5
| Доход[3, 2] := 1.1; Доход[4, 2] := 0.6; Доход[5, 2] := 1.2
| Доход[1, 3] := 1.4; Доход[2, 3] := 1.2; Доход[3, 3] := 1.2
| Доход[4, 3] := 1.3; Доход[5, 3] := 1.3; Доход[1, 4] := 2.0
| Доход[2, 4] := 1.8; Доход[3, 4] := 1.4; Доход[4, 4] := 1.4
| Доход[5, 4] := 1.3; Доход[1, 5] := 2.5; Доход[2, 5] := 2.5
| Доход[3, 5] := 1.6; Доход[4, 5] := 1.5; Доход[5, 5] := 1.3
| Доход[1, 6] := 2.8; Доход[2, 6] := 2.9; Доход[3, 6] := 1.7
| Доход[4, 6] := 1.5; Доход[5, 6] := 1.3; Доход[1, 7] := 3.0
| Доход[2, 7] := 3.5; Доход[3, 7] := 1.8; Доход[4, 7] := 1.5
| Доход[5, 7] := 1.3; Доход[1, 8] := 3.0; Доход[2, 8] := 3.5
| Доход[3, 8] := 1.8; Доход[4, 8] := 1.5; Доход[5, 8] := 1.3
| Доход[1, 9] := 3.0; Доход[2, 9] := 3.5; Доход[3, 9] := 1.8
| Доход[4, 9] := 1.5; Доход[5, 9] := 1.3; Доход[1, 10] := 3.0
| Доход[2, 10] := 3.5; Доход[3, 10] := 1.8; Доход[4, 10] := 1.5
| Доход[5, 10] := 1.3
| |Определяем условные оптимальные параметры
| |Последний шаг. Вкладываем все, что осталось
| нц для Ост от 0 до Q
| | УОУ[Ост, n] := Ост
| | УОВ[Ост] := Доход[n, Ост]
| кц
| |Остальные шаги (кроме первого)
| нц для Nшага от n - 1 до 2 шаг -1
| | нц для Ост от 0 до Q
| | | ВыборВарианта(Ост, Nшага, Доход, УОВ,
| | |               УОУ[Ост, Nшага], УОВнов[Ост])
| | кц
| | |Переписываем массив УОВнов в УОВ
| | нц для Ост от 0 до Q
| | | УОВ[Ост] := УОВнов[Ост]
| | кц
| кц
| |Первый шаг: Nшага = 1. Возможное состояние одно - Q
| ВыборВарианта(Q, 1, Доход, УОВ, УОУ[Q, 1], УОВнов[Q])
| |Печатаем результат
| вывод нс, "Максимальный доход: ", УОВнов[Q]
| вывод нс, "Для этого необходимо выделить"
| Ост := Q
| нц для Nшага от 1 до n
| | вывод нс, "- предприятию", Nшага, " - ", УОУ[Ост, Nшага]
| | Ост := Ост - УОУ[Ост, Nшага]
| кц
кон
```

Выполнив эту программу, получим сообщение:

```
Максимальный доход: 5.6
Для этого необходимо выделить:
- предприятию 1 — 0
- предприятию 2 — 7
- предприятию 3 — 2
- предприятию 4 — 0
- предприятию 5 — 1
```

Как видим, полученный результат вполне согласуется с табл. 12.9.

В заключение отметим следующее. Конечно, нами были рассмотрены простейшие задачи динамического программирования. Однако они дают понятие об общей идее метода: это пошаговая оптимизация, проводимая сначала в одном направлении (от конца к началу) «условно», а затем в другом (от начала к концу) — «безусловно». При этом при выборе условного оптимального управления следует соблюдать следующее правило: *каково бы ни было состояние системы перед очередным шагом, надо выбирать управление на этом шаге так, чтобы выигрыш на данном шаге плюс оптимальный выигрыш на всех последующих шагах был максимальным.* Это правило является общим принципом, лежащим в основе решения всех задач динамического программирования (его часто называют «*принципом оптимальности*»).

Сформулируем также несколько практических рекомендаций, полезных начинающему при постановке задач динамического программирования. Эту постановку удобно проводить в следующем порядке:

1) расчленяем исследуемую операцию на этапы (шаги);
2) выбираем параметры, характеризующие состояние системы перед каждым шагом;
3) для каждого i-го шага выясняем набор возможных управлений $У_i$ и налагаемые на них ограничения;
4) определяем, какой выигрыш приносит на i-м шаге управление $У_j$, если перед этим система была в состоянии S, т. е. записываем так называемую «функцию выигрыша»:

$$B_i^j = f_i(S, У_j);$$

5) определяем, как изменяется состояние S системы под влиянием управления $У_j$ на i-м шаге, — оно переходит в новое состояние S':

$$S' = \varphi(S, У_j); \qquad (12.1)$$

6) записываем для каждого состояния S основное рекуррентное уравнение динамического программирования, выражающее условный

оптимальный выигрыш УОВ (начиная с i-го шага и до конца) через уже известную функцию $B_{i+1}(S')$:

$$УОВ_i(S) = \max_{y_i}\{B_i^j + УОВ_{i+1}(S')\}. \qquad (12.2)$$

Этому выигрышу соответствует условное оптимальное управление $УОУ_j(S)$;

7) производим условную оптимизацию последнего (n-го) шага, задаваясь «гаммой» состояний S, из которых можно за один шаг дойти до конечного состояния, и вычисляя для каждого из них условный оптимальный выигрыш по формуле:

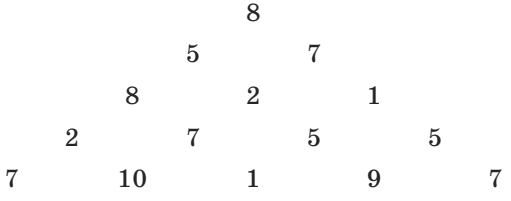
$$УОВ_n(S) = \max_{y_i}\{B_i^j\}$$

и находя условное оптимальное управление $УОУ_n(S)$, для которого достигается этот максимум;

8) производим условную оптимизацию (n−1)-го, (n−2)-го и т. д. шагов по формуле (12.2), полагая в ней i = (n−1), (n−2), …, 1, и для каждого из шагов указываем условное оптимальное управление УОУ(S), при котором максимум достигается (для каждого возможного состояния). Заметим, что если состояние системы в начальный момент известно (а обычно это бывает именно так), то на первом шаге варьировать состояние системы не нужно — мы прямо находим оптимальный выигрыш $ОВ_1$ для данного начального состояния S_0 (это и есть оптимальный выигрыш за всю операцию); при этом одновременно определяем оптимальное управление $ОУ_1$;

9) производим безусловную оптимизацию управления, «читая» соответствующие рекомендации на каждом шаге. Для этого надо взять найденное оптимальное управление на первом шаге $ОУ_1 = ОУ_1(S_0)$; изменить состояние системы по формуле (12.1), для вновь найденного состояния найти оптимальное управление на втором шаге $ОУ_2$ и т. д. до конца.

1. На рисунке изображен треугольник из чисел. Найдите максимальную сумму чисел, расположенных на пути, начинающемся в верхней точке треугольника и заканчивающемся на его основании:



2. Страховая компания распределяет 7 агентов по трем районам для заключения договоров страхования. Зависимость ожидаемого количества договоров от числа работающих в том или ином районе агентов приведена в таблице:

Кол-во агентов	1-й район	2-й район	3-й район
0	0	0	0
1	30	50	40
2	50	80	50
3	90	90	110
4	110	150	120
5	170	190	180
6	180	210	220
7	210	220	240

Распределите агентов по районам так, чтобы общее ожидаемое количество договоров было наибольшим.

3. Из предметов массой $m_1, m_2, .., m_n$ составьте набор массой M или, если это невозможно, массой, максимально близкой (снизу) к M.

13. Формирование комбинаторных объектов

В этом разделе рассмотрены алгоритмы формирования трех основных комбинаторных объектов — перестановок, сочетаний и размещений из некоторого исходного набора элементов.

Прежде всего, сделаем несколько предварительных замечаний.

Все такие объекты формируются из набора из n элементов. Для хранения этих элементов в программах применяется массив эл. При этом принимается, что элементы имеют строковый тип (что дает возможность использовать в качестве значений элементов символы, строки, а также числа — вернее, их «строковое» представление).

В то же время комбинаторные объекты формируются не из этих значений — приведенные алгоритмы основаны на обработке *индексов* элементов массива эл. Например, результатом выполнения процедур генерирования перестановок являются последовательности индексов всех исходных элементов. Зная их, можно легко получить соответствующие перестановки самих элементов[48].

Введем также понятие *лексикографического порядка* расположения последовательностей a и b. Последовательности равной длины называются *упорядоченными в лексикографическом порядке*, и последовательность a предшествует последовательности b, если первые s элементов этих последовательностей соответственно равны, а (s + 1)-й элемент последовательности a меньше (s + 1)-го элемента последовательности b. При этом $s \geq 0$.

Применительно к последовательностям, элементы которых представляют собой числа от 1 до n, можно дать и такое определение [8, 11]: порядок на множестве указанных последовательностей длины k называется лексикографическим, если для двух любых последовательностей справедлив следующий порядок их расположения: ранее должна стоять та из них, которая является меньшим числом в (n + 1)-ричной системе счисления. Например, при k = 3 и n = 14 последовательность 1 8 4 должна стоять раньше, чем 1 13 2, так как $184_{15} < 1D2_{15}$ (в 15-ричной системе 13 = D).

[48] См. далее процедуру Печать.

13.1. Перестановки

Напомним, что *перестановками* из n различных элементов называются все возможные варианты их размещения (при этом количество размещаемых элементов неизменно и равно n). При этом количество различных перестановок из n элементов равно n!, поэтому надо сразу отказаться от алгоритмов, в которых перестановки заносятся в массив.

Известно несколько алгоритмов формирования перестановок. Один из них состоит в следующем. Как указывалось чуть выше, в результате выполнения алгоритма должны быть получены последовательности индексов всех исходных элементов. Для хранения и обработки этих индексов мы будем использовать вспомогательный массив с именем инд. Потребуем, чтобы последовательности индексов элементов инд[1], инд[2], .., инд[n] формировались в лексикографическом порядке (см. выше). Например, для массива из четырех элементов в таком порядке:

```
1  2  3  4
1  2  4  3
1  3  2  4
1  3  4  2
1  4  2  3
1  4  3  2
2  1  3  4
   ...
4  1  3  2
4  2  1  3
4  2  3  1
4  3  1  2
4  3  2  1
```

Если проанализировать значения, меняющиеся при переходе от одной перестановки к другой в приведенном примере, то можно увидеть, в каком случае k-й член последовательности увеличивается, а (k – 1) предыдущих членов при этом не меняются: это происходит при таком максимальном k, которое меньше какого-либо элемента правее него. Чтобы найти такое значение, надо просматривать массив инд с правого конца и искать первую пару соседних элементов, для которых инд[k] < инд[k + 1].

Согласно этому неравенству, правее инд[k] существует по крайней мере один элемент, больший инд[k], а так как такое неравенство между соседними элементами встретилось впервые, то для уже просмотренных элементов справедливо соотношение:

```
инд[k+1] > инд[k+2] > инд[k+2] >...> инд[n]
```
(13.1)

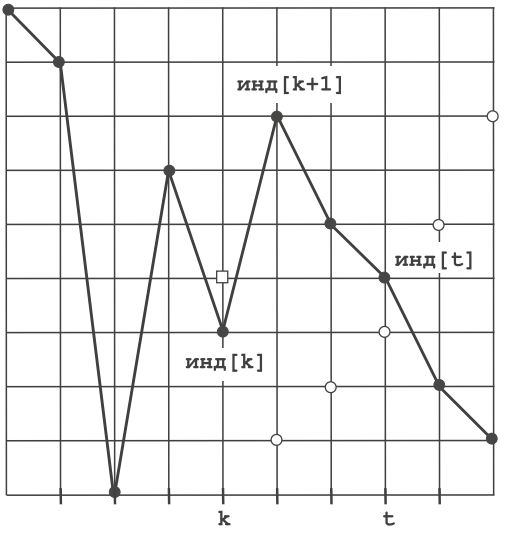


Рис. 13.1

Что еще происходит при переходе от одной перестановки к другой? Видно (см., например, переход от перестановки `1 4 3 2` к `2 1 3 4`), что:

1) найденный элемент `инд[k]` меняется местами с минимальным из элементов, больших него и находящихся правее него. Важно, что после такого обмена соотношение (13.1) остается справедливым (см. рис. 13.1[49]). Индекс минимального из элементов, о котором только что шла речь, обозначим как `t`;
2) (`k + 1`)-й, (`k + 2`)-й, .., `n`-й элементы располагаются так, чтобы новая перестановка была наименьшей с точки зрения лексикографического порядка, т. е. они должны быть размещены в порядке возрастания; это облегчается тем, что эти элементы уже расположены в убывающем порядке (см. рис. 13.1, где элементы, размещенные в порядке возрастания, обозначены кружочком).

Прежде чем написать соответствующую процедуру, заметим, что она оперирует с массивом `инд`, описанным в программе как глобальный (см. раздел 9).

```
алг Следующая
нач цел k, t, всп, лев, прав, i
| |Определяем значение k
| k := n - 1
| нц пока инд[k] > инд[k + 1]
| | k := k - 1
| кц
| |Определяем значение t
| t := n
```

[49] На рисунке место элемента `инд[t]` после обмена показано в виде квадратика.

```
| нц пока инд[t] < инд[k]
| | t := t - 1
| кц
| |Обмениваем значениями элементы с индексами k и t
| Обмен(k, t)
| |Располагаем элементы с индексами k + 1, ..., n
| |в обратном порядке[50]
| лев := k + 1 |Левая граница индексов
| прав := n    |Правая граница
| нц пока лев < прав
| | Обмен(лев, прав)
| | лев := лев + 1
| | прав := прав - 1
| кц
кон
```

где Обмен — процедура, осуществляющая обмен значениями двух элементов массива ind с заданными индексами n1 и n2:

```
алг Обмен(арг цел n1, n2)
нач цел всп
| всп := инд[n1]
| инд[n1] := инд[n2]
| инд[n2] := всп
кон
```

Основную программу можно оформить следующим образом (напомним, что результатом ее выполнения являются последовательности не индексов исходных элементов, а их значений):

```
цел n
n := ...
цел таб инд[1:n], лит таб эл[1:n]
алг Перестановки_из_n_элементов
нач цел i
| |Вводим исходные данные:
| нц для i от 1 до n
| | вывод нс, "Введите", i, "-й элемент"
| | ввод эл[i]
| кц
| |Формируем массив инд - первую последовательность
| |индексов исходных элементов
| нц для i от 1 до n
| | инд[i] := i
| кц
```

[50] Можно применить для этого также оператор цикла с параметром:

```
нц для i от k + 1 до div(k + 1 + n, 2)
| Обмен(i, n + k + 1 - I)
кц
```

```
| |Печатаем первую перестановку:
| Печать
| вывод " "
| нц
| | |Используем процедуру формирования
| | |следующей последовательности
| | Следующая
| | |Печатаем соответствующую перестановку
| | Печать
| | вывод " "
| кц при Последняя
кон
```

Здесь:

1) Последняя — **функция логического типа, определяющая, является ли последовательность** инд[1], ..., инд[n] **последней. Помня, что последней последовательностью является** n, n – 1, ..., 1, **эту функцию можно оформить в виде:**

```
алг лог Последняя(арг цел n, цел таб инд[1:n])
нач цел i, лог упоряд
| |упоряд – величина, контролирующая, является ли
| |последовательность инд[1], ..., инд[n]
| |упорядоченной по убыванию
| i := 1
| упоряд := да
| нц пока i <= n - 1 и упоряд
| | если инд[i] > инд[i + 1]
| | | то
| | |  i := i + 1
| | | иначе
| | |  упоряд := нет
| | все
| кц
| знач :=  упоряд |Значение функции
кон
```

2) Печать — **процедура печати перестановки (перечня элементов массива** эл**):**

```
алг Печать
нач цел i
| вывод нс
| нц для i от 1 до n
| | вывод эл[инд[i]]
| кц
кон
```

Можно несколько ускорить вычисления, «совместив» определение величины k и проверку того факта, что последовательность инд(1), ..., инд(n) является последней. В самом деле, если допустить, что k может быть равно 0 (при расширении в массиве инд диапазона индексов влево до 0), то в случае, когда k = 0, полученная перестановка является последней. С учетом этого основная программа примет вид:

```
алг Перестановки_из_n_элементов
нач цел k, i
| |Вводим исходные данные:
| ...
| |Формируем массив инд
| нц для i от 0 до n
| | инд[i] := i
| кц
| нц
| | Печать
| | |Определяем значение k
| | k := n - 1
| | нц пока x[k] > x[k + 1]
| | | k := k - 1
| | кц
| | если k > 0
| | | то
| | |  Следующая
| | |  Печать
| | все
| кц при k = 0
кон
```

Процедура Следующая при этом будет выглядеть так:

```
алг Следующая
нач цел t, всп, лев, прав
| |Определяем значение t
| t := k + 1
| нц пока t < n и инд[t + 1] > инд[k]
| | t := t + 1
| кц
| |Обмениваем значениями элементы с индексами k и t
| Обмен(k, t)
| |Располагаем элементы с индексами k + 1, ..., n
| |в обратном порядке
| лев := k + 1
| прав := n
| нц пока лев < прав
| | Обмен(лев, прав)
| | лев := лев + 1
| | прав := прав - 1
| кц
кон
```

Заметим, что при небольших[51] значениях *n* можно не использовать условие, проверяющее, является ли полученная перестановка последней, а применить оператор цикла с параметром, вычислив предварительно количество перестановок (n!).

Для решения рассматриваемой задачи также может быть использована рекурсия. Пусть массив инд первоначально заполнен числами 1, 2, ..., n. Создадим рекурсивную процедуру Перестан, формирующую все перестановки указанных чисел в лексикографическом порядке. Идея использования рекурсии в ней следующая: на первом месте в перестановках должны побывать все числа с 1 до n и для каждого из этих элементов должны быть получены все элементы массива инд со 2-го до n-го в лексикографическом порядке. Как это сделать — в деталях мы обсудим чуть ниже. Здесь же заметим, что для этого надо решить аналогичную задачу — на втором месте поочередно разместить все числа с 1 до n (кроме тех, которые уже размещены на первой позиции) и для каждого из них получить перестановки для оставшихся чисел, и т. д. для всех остальных позиций. Это означает, что параметром процедуры Перестан должен быть индекс элемента, начиная с которого должны быть получены все перестановки правой части массива (имя этой величины — начало). При начало = n очередная перестановка получена, т. е. начальный фрагмент процедуры может быть оформлен так:

```
алг Перестан(арг цел начало)
нач
| если начало = n
| |то
| | Печать
| |иначе
| | |Продолжаем формирование очередной перестановки
| | ...
```

Как получить все упорядоченные перестановки оставшихся чисел, начиная с позиции (начало + 1)? Для этого надо:

1) рекурсивно вызвать процедуру Перестан с параметром, меняющимся от (начало + 1) до n, и после каждого вызова менять местами элементы с индексами начало и i[52]:

```
нц для i от начало + 1 до n
| Перестан(начало + 1)
| Обмен(начало, i)
кц
```

Такой обмен нужен, чтобы на месте начало разместить все цифры из находящихся справа, кроме n;

---

[51] Учитывая, что значение n! растет очень быстро.

[52] Смысл величины i следует из приведенного далее оператора цикла.

2) еще раз рекурсивно вызвать процедуру Перестан для формирования очередной группы перестановок, когда на месте начало стоит число n.

В конце процедуры Перестан следует восстановить тот порядок элементов, который был при ее вызове. Так как последним на месте начало оказывается число n, то это можно сделать, сместив элементы с индексами начало + 1, начало + 2, ... , n на одну позицию влево, а элемент инд[начало] разместить в конце массива (как это сделать, было описано в разделе 6). Тогда вся процедура Перестан примет вид:

```
алг Перестан(арг цел начало)
нач цел i, всп
| если начало = n
| |то
| | Печать
| |иначе
| | нц для i от начало + 1 до n
| | | Перестан(начало + 1)
| | | Обмен(начало, i)
| | кц
| | Перестан(начало + 1)
| | |Сдвиг влево и перемещение элемента инд[начало]
| | всп := инд[начало]
| | нц для i от начало до n - 1
| | | инд[i] := инд[i + 1]
| | кц
| | инд[n] := всп
| все
кон
```

Если логика этой процедуры[53] показалась вам сложной, то рассмотрим еще один рекурсивный вариант. Его особенность заключается в следующем. Поменяем местами первый и второй, первый и третий, ..., первый и n-й элементы массива инд. После каждого обмена будем таким же образом (т. е. рекурсивно) получать перестановки (n–1)-элементного массива (отбросив первый элемент данного):

```
алг Перестановки(арг цел i)
нач цел j, всп
| если i = n
| |то
| | Печать
| |иначе
| | нц для j от i до n
| | | если i <> j
| | | | то
| | | |  Обмен(i, j)
```

[53] В основной части программы ее вызов выглядит так: Перестан(1).

```
| | | все
| | | Перестановки(i + 1)
| | | если i <> j
| | | | то
| | | |  Обмен(i, j)
| | | все
| | кц
| все
кон
```

Нетрудно убедиться, что в последнем варианте лексикографический порядок перестановок не обеспечивается.

Завершая обсуждение задачи формирования перестановок, заметим, что три алгоритма решения этой задачи приведены в [9]. В первом из них перестановки получаются в так называемом «антилексикографическом» порядке индексов элементов, во втором — каждая следующая перестановка получается из предыдущей путем выполнения одной *транспозиции* (обмена местами двух элементов). В третьем алгоритме, как и во втором, проводится одна транспозиция, но только соседних элементов. Второй из перечисленных алгоритмов приведен также в [8].

13.2. Сочетания

Напомним определение [10]. Если из n различных элементов составлять группы по m элементов в каждой, не обращая внимания на порядок элементов в группах, то получающиеся при этом комбинации называются *сочетаниями* из n элементов по m. Например, сочетаниями из пяти элементов a b c d e по три являются: a b c, a b d, a b e, a c d, a c e, a d e, b c d, b c e, b d e, c d e.

Решим сначала такую задачу: *Требуется получить все сочетания n различных элементов* (т. е. все подмножества данного множества).

Если использовать вспомогательный массив двоич из n элементов со значениями 1 или 0, то можно утверждать, что каждому искомому сочетанию соответствует некоторая совокупность значений всех элементов массива двоич, определяемая по правилу:

$$\text{двоич[i]} = \begin{cases} 1, \text{ если эл[i] входит в сочетание,} \\ 0 \text{ — в противном случае,} \end{cases} \qquad (13.2)$$

где эл — массив исходных элементов.

Значения всех элементов массива двоич будем рассматривать как некоторую последовательность единиц и нулей длиной n. Нетрудно заметить, что каждая такая последовательность есть двоичное представление какого-либо целого числа m из интервала $1 \le m \le 2^n - 1$ (без учета последовательности, состоящей из одних нулей). Поэтому,

чтобы получить все сочетания n различных элементов, необходимо получить все двоичные представления чисел 1, 2, ..., $2^n - 1$, а затем напечатать соответствующие сочетания, учитывая правило (13.2). Процедура печати в этом случае имеет вид:

```
алг Печать
нач цел i
| вывод нс
| нц для i от 1 до n
| | если двоич[i]=1
| | | то
| | |  вывод эл[i]
| | все
| кц
кон
```

Получить же двоичное представление указанных чисел, добавляя 1 на каждом шаге, можно с помощью массива двоич следующим образом. Если последний элемент массива двоич равен 0, то необходимо заменить его на 1, в противном случае надо:

а) найти первый (идя от конца массива) элемент, равный 0, и заменить его на 1; индекс такого элемента обозначим как инд0;

б) все элементы от (s+1)-го до n-го принять равными нулю.

Соответствующая процедура увеличения на 1 двоичного представления целого числа оформляется так:

```
алг Увелич1
нач цел инд0, i
| если двоич[n] = 0
| | то
| |  двоич[n] := 1
| | иначе
| |  |Определяем значение инд0
| |  инд0 := n
| |  нц
| |  | инд0 := инд0 - 1
| |  кц при s = 1 или двоич[инд0] = 0
| |  |Значение инд0 найдено
| |  |Заменяем двоич[инд0] на единицу
| |  двоич[инд0] := 1
| |  |а все оставшиеся справа элементы - на нуль
| |  нц для i от инд0 + 1 до n
| |  | двоич[i] := 0
| |  кц
| все
кон
```

Основная программа:

```
цел n
n := ...
цел таб инд[1:n], лит таб эл[1:n]
алг Все_сочетания_n_элементов
нач цел i
| |Вводим исходные данные
| нц для i от 1 до n
| | вывод нс, "Введите", i, "-й элемент"
| | ввод эл[i]
| кц
| |Формируем последовательность единиц и нулей
| |(массив двоич), соответствующую
| |двоичному представлению числа 1
| нц для i от 1 до n - 1
| | двоич[i] := 0
| кц
| двоич[n] := 1
| |Печатаем первое сочетание
| Печать
| нц
| | |Используем процедуру формирования
| | |следующей последовательности единиц и нулей
| | Увелич1
| | |Печатаем соответствующее сочетание:
| | Печать
| кц при Последняя
кон
```

где Последняя — это функция логического типа, определяющая, является ли последовательность чисел в массиве двоич последней. Зная, что последняя последовательность состоит из одних единиц, эту функцию можно оформить в виде:

```
алг лог Последняя
нач цел i, лог равны1 |равны1 – величина, контролирующая,
| |равны ли единице все элементы массива x
| i := 1
| равны1 := да
| нц пока i <= n и равны1
| | если двоич[i] = 1
| | | то
| | |  i := i + 1
| | | иначе
| | |  равны1 := нет
| | все
| кц
| знач :=  равны1 |Значение функции
кон
```

Теперь решим задачу получения сочетаний из n элементов по m. Очевидный способ решения — перебор всех сочетаний (как раньше) и отбор из них тех, в которых m элементов, — мы отбросим, считая его неэкономичным (значение m может быть намного меньше n).

Один из возможных «экономичных» вариантов решения задачи заключается в следующем. Если, как и раньше, использовать массив двоич из n элементов, равных 0 или 1, то можно утверждать, что каждому из искомых сочетаний соответствует некоторый набор значений всех элементов массива двоич, в котором имеется m единиц и к которому применимо правило (13.2).

Последовательности значений всех элементов двоич[1], .., двоич[n] мы будем получать в лексикографическом порядке (см. выше), т. е., например, при n = 4 и m = 2:

```
0 0 1 1
0 1 0 1
0 1 1 0
1 0 0 1
1 0 1 0
1 1 0 0
```

Здесь видно, что очередная последовательность получается из предыдущей по следующему правилу:

1) один из элементов, равный нулю (назовем его индекс s), надо заменить на единицу;
2) идущие за s-м элементом члены последовательности следует записать так, чтобы последовательность двоич[s + 1], .., двоич[n] была минимальной с точки зрения нашего порядка, т. е. чтобы сначала шли нули, а потом единицы (если они есть на указанном отрезке последовательности).

Для этого надо знать количество единиц на участке массива двоич[s], .. , двоич[n]. Определив это количество (имя величины — кол1), можно рассчитать и количество нулей на этом же участке — кол0:

```
кол0 = (n - s + 1) - кол1
```

А как определить само значение s? Если двоич[s] меняется с 0 на 1, то для сохранения общего количества единиц нужно справа от двоич[s] заменить 1 на 0. Таким образом, s — это индекс первого (при рассмотрении массива с конца) нуля, справа от которого стоит единица. Легко увидеть, что двоич[s + 1] = 1, иначе двоич[s] — не первый нуль. Следовательно, надо найти наибольшее s, для которого двоич[s] = 0 и двоич[s + 1] = 1.

Итак, основные этапы формирования последовательности двоич[1], .., двоич[n] из имеющейся будут следующими:

1) определение значения s;
2) замена двоич[s] на 1;

3) **запись в массив** кол0 **нулей (начиная с элемента с индексом** s + 1**);**
4) **запись в конце массива** кол1 **−** 1 **единиц (одна единица уже присвоена величине** двоич[s]**).**

Этим этапам соответствует такая процедура:

```
алг Следующая
нач цел s, кол1, кол0, i
| |Определяем значение s
| s := n - 1
| нц пока не(двоич[s]=0 и двоич[s + 1]=1)
| | s := s - 1
| кц
| |Определяем значение кол1:
| кол1 := 0
| нц для i от s + 1 до n
| | если двоич[i]=1
| | | то
| | |  кол1 :=  кол1 + двоич[i]
| | все
| кц
| |Заменяем элементы на участке двоич[s], .., двоич[n]
| |1) заменяем элемент двоич[s]
| двоич[s] := 1
| |Определяем число нулей кол0
| кол0 := (n - s + 1) - кол1
| |и записываем их:
| нц для i от s + 1 до s + кол0
| | двоич[i] := 0
| кц
| |2) записываем необходимое число единиц:
| нц для i от s + кол0 + 1 до n
| | двоич[i] := 1
| кц
кон
```

Основная программа:

```
цел n, m
n := ...
цел таб инд[1:n], лит таб эл[1:n]
алг Все_сочетания_n_элементов
нач цел i
| |Вводим исходные данные
| ... (см. выше)
| |в т. ч. значение m
```

```
| нц
| | вывод нс, "Введите значение m"
| | ввод m
| | если m > n
| | | то
| | |  вывод нс, "Значение m не должно быть больше n"
| | все
| кц при m <= n
| |Получаем и печатаем первое сочетание
| |а) заполняем первую последовательность (массив двоич)
| нц для i от 1 до n - m      |Она состоит из (n - m) нулей
| | двоич[i] := 0
| кц
| нц для i от n - m + 1 до n   |и m единиц
| | двоич[i] := 1
| кц
| |б) печатаем первое сочетание
| Печать
| нц
| | |Используем процедуру формирования
| | |следующей последовательности единиц и нулей
| | Следующая
| | |Печатаем соответствующее сочетание
| | Печать
| кц при Последняя
кон
```

Здесь процедура печати очередного сочетания `Печать` **аналогична приведенной выше, а функция** `Последняя` **определяет, является ли последовательность чисел в массиве** `двоич` **последней. Учитывая, что последней является последовательность:** `1 1 ... 1 0 0 ... 0` **(**`m` **единиц, (**`n - m`**) нулей), функцию** `Последняя` **можно оформить в виде:**

```
алг лог Последняя
нач цел i, лог равны1
| равны1 := да; i := 1
| нц
| | если двоич[i] <> 1
| | | то
| | |  равны1 := нет
| | |иначе
| | |  i := i + 1
| | все
| кц при i = m + 1 или не равны1
| знач := равны1
кон
```

Все сочетания из n элементов по m в лексикографическом порядке также можно получить, применив рекурсию. Как это обычно имеет место при использовании рекурсии, решение сводится к аналогичной задаче другой размерности (в нашем случае — меньшей). Первым элементом сочетания при этом может быть любой элемент массива инд, начиная с первого и кончая (n – (m – 1))-м (иначе мы не сможем разместить еще (m – 1) элементов — убедитесь в этом сами):

```
цел n, m
n := 5
m := 2
цел таб инд[1:n]
алг Все_сочетания_n_элементов
нач цел i
| нц для i от 1 до m
| | инд[i] := i
| кц
| Сочетания(n, m)
кон
алг Сочетания(арг цел n, m)
нач
| Сочетан(m, 1)
кон
алг Сочетан(арг цел k, начало)
нач цел i
| если k = 0
| |то
| | нц для i от 1 до m
| | | вывод инд[i]
| | кц
| | вывод нс
| |иначе
| | нц для i от начало до n - k + 1
| | | инд[m - k + 1] := i
| | | Сочетан(k - 1, i + 1)
| | кц
| все
кон
```

13.3. Размещения

Сначала также напомним определение [10]. Будем составлять из n различных элементов группы по m элементов в каждой, располагая взятые m элементов в различном порядке. Получающиеся при этом комбинации называются *размещениями* из n элементов по m. Например, размещения из четырех элементов a, b, c и d по 2 будут следующими:

ab, ba, ac, ca, ad, da, bc, cb, bd, db, cd, dc

Нетрудно увидеть, что размещения из n элементов по m могут быть получены путем формирования всех возможных сочетаний из n элементов по m с последующей генерацией для каждого сочетания элементов их перестановок. Предлагаю читателям разработать такой вариант решения самостоятельно. Здесь же мы опишем процедуру, с помощью которой можно получить все размещения из n элементов по m непосредственно.

Начнем с варианта, в котором размещаемыми элементами являются цифры. Пусть n = 4, m = 3. Все соответствующие размещения приведены (в лексикографическом порядке) в таблице:

123	124	132	134	142	143
213	214	231	234	241	243
312	314	321	324	341	342
412	413	421	423	431	432

Каждое формируемое размещение будем хранить в массиве разм, а элементы (цифры), уже использованные в размещении, — в массиве испол. Для проверки факта использования цифры i среди k уже имеющихся в размещении цифр применим функцию Использована:

```
алг лог Использована(арг цел таб испол[1:n], цел i)
нач лог ис, цел j
| j := 1; ис := нет
| нц пока j <= k и не ис
| | если испол[j]=i
| | |то
| | | ис := да
| | |иначе
| | | j := j + 1
| | все
| кц
| знач := ис
кон
```

Для формирования очередного размещения создадим рекурсивную процедуру Размещения; ее параметром сделаем номер позиции в размещении, для которой подбирается цифра:

```
алг Размещения(арг цел позиция)
нач цел i
| |Каждую цифру
| нц для i от 1 до n
| | |Проверяем, использована ли она в размещении
| | если не Использована(испол, i)
| | |то
```

```
| | | |Записываем ее в массив использованных цифр
| | | k := k + 1 |Номер очередной использованной цифры
| | | испол[k] := i
| | | |Записываем ее в размещение на место позиция
| | | разм[позиция] := i
| | | если позиция < m
| | | | то
| | | |  |Рекурсивно вызываем процедуру
| | | |  |для подбора цифры в следующей позиции
| | | |  Размещения(позиция + 1)
| | | | иначе
| | | |  |Очередное размещение получено
| | | |  |Выводим его
| | | |  вывод разм, " "
| | | все
| | | |Возвращаем цифру i в число неиспользованных,
| | | |уменьшив на 1 значение k
| | | k := k - 1
| | все
| кц
кон
```

Логика работы процедуры описана в комментариях.

Основная программа имеет вид:

```
цел n, m, k
цел таб разм[1:m]
цел таб испол[1:n]
алг Все размещения
нач
| n := 4; m := 3
| k := 0
| Размещения(1)
кон
```

Теперь обсудим, что должно измениться в случае, когда размещаемые элементы являются величинами строкового типа. Во-первых, понадобится массив эл с этими элементами. Строковый тип должны иметь и элементы массивов разм и испол. Следовательно, основная программа должна быть оформлена так:

```
цел n, m, k
n := 4; m := 3
лит таб эл[1:n]
лит таб разм[1:m]
лит таб испол[1:n]
```

```
алг Все_размещения
нач
| эл[1] := ...
| эл[2] := ...
| ...
| k := 0
| Размещения(1)
кон
```

Во-вторых, заголовок процедуры Использована **должен принять вид**[54]**:**

```
алг лог Использован(арг лит таб испол[1:n], лит i)
```

Более существенные изменения должна претерпеть процедура Размещения**:**

```
алг Размещения(арг цел позиция)
нач цел i |номер элемента
| |Каждый элемент
| нц для i от 1 до n
| | |Проверяем, использован ли он в размещении
| | если не Использован(испол, эл[i])
| | | то
| | |  |Записываем его в массив использованных элементов
| | |  k := k + 1 |Номер очередного использованного элемента
| | |  испол[k] := эл[i]
| | |  |Записываем его в размещение
| | |  разм[позиция] := эл[i]
| | |  если позиция < m
| | |  | то
| | |  |  |Рекурсивно вызываем процедуру
| | |  |  |для подбора элемента в следующей позиции
| | |  |  Размещения(t + 1)
| | |  | иначе
| | |  |  вывод разм, " "
| | |  все
| | |  |Возвращаем элемент эл[i] в число неиспользованных,
| | |  |уменьшив на 1 значение k
| | |  k := k - 1
| | все
| кц
кон
```

[54] С учетом того, что здесь речь идет не о цифре, а о символьном размещаемом элементе.

1. Дано слово «*колун*». Получите все пятибуквенные слова (в том числе бессмысленные, например «кнуол»), которые можно составить из букв этого слова.
2. Получите двоичные представления всех десятичных чисел от 1 до 31.
3. В сборную школы по баскетболу входит 8 человек: Александр, Андрей, Армен, Борис, Дмитрий, Егор, Ринат и Сергей. Получите все возможные варианты стартовой пятерки для этой команды.
4. В автомобиле имеется 5 мест: водительское, кресло рядом с ним и три места на заднем сидении. Получите все варианты размещения в автомобиле пяти пассажиров из имеющихся шести человек.

Все задачи решите, используя методы, описанные в данном разделе.

14. Обработка деревьев

Нет, уважаемый читатель, в этом разделе мы не будем рассказывать, как надо ухаживать за деревьями и кустарниками ☺. Речь пойдет о методике обработке данных, связи между которыми имеют «древовидный» характер (рис. 14.1–14.4).

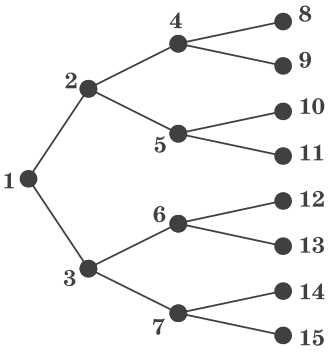


Рис. 14.1

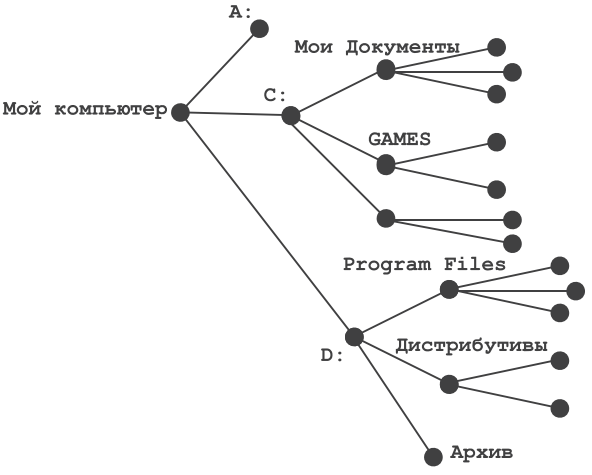


Рис. 14.2

Как решать задачи, связанные с обработкой данных, представленных в древовидной структуре? Прежде чем отвечать на этот вопрос, давайте обсудим терминологию. Конечные точки отрезков, из которых состоит *дерево*, называют *вершинами*. Вершина, из которой начинается («растет») дерево, называется *корнем*, вершины, к которым ведут линии из некоторой `i`-й вершины, — ее *потомками*[55], а сама `i`-я вершина называется *предком* вершин — ее потомков.

[55] Строго говоря, потомками `i`-й вершины являются не только указанные вершины, но и все их потомки.

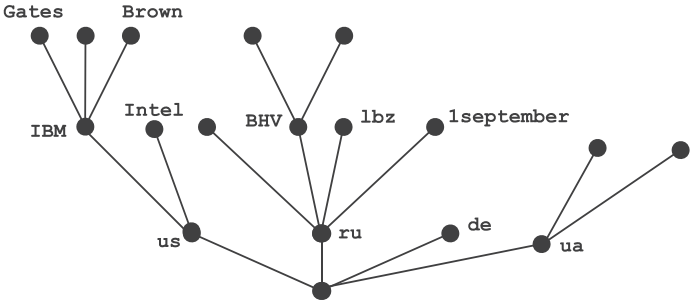


Рис. 14.3

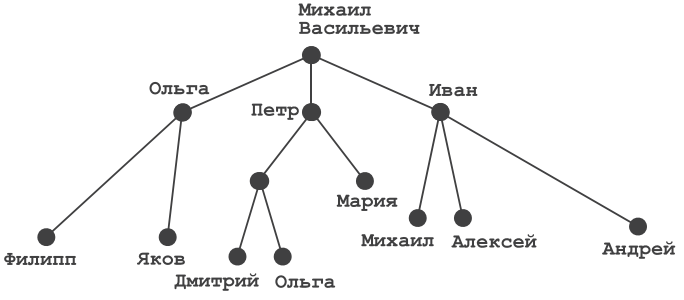


Рис. 14.4

Можно рассуждать так. Находясь в корне, следует обработать[56] его значение, а затем выполнить то же самое для каждого потомка (т. е. использовать прием, который в программировании называют рекурсией, — см. раздел 10).

Применим эти рассуждения, например, чтобы вывести номера всех вершин дерева, изображенного на рис. 14.1. Конечно, мы и так знаем эти номера, так что единственная цель, которую мы ставим при разработке соответствующей программы, — показать использование рекурсии для решения этой и подобных задач.

Так как между номером вершины-предка $N_{пр}$ и номерами вершин — ее потомков $N_{пот}$ в данном случае имеется зависимость:

$$N_{пот} = N_{пр} \times 2 + i \quad \text{при} \quad i = 0, 1$$

или

$$N_{пот} = N_{пр} \times 2 + (i - 1) \quad \text{при} \quad i = 1, 2,$$

то сделанные выше рассуждения об обработке данных в древовидной структуре можно оформить в виде следующей рекурсив-

[56] Под термином «обработать» здесь понимается любое действие, связанное с номером вершины или со значением, сопоставленным этой вершине (см. далее по тексту).

ной процедуры, моделирующей действия на очередном шаге обработки:

```
алг Очередной_шаг(арг цел номер_вершины)
нач
| |Выводим номер текущей вершины
| вывод номер_вершины
| если номер_вершины <= 7
| | то
| |  |Вызываем процедуру Очередной_шаг
| |  |для обработки ее (вершины) потомков
| |  Очередной_шаг(номер_вершины * 2)
| |  Очередной_шаг(номер_вершины * 2 + 1)
| все
кон
```

Как видим, ее аргументом является номер вершины, обрабатываемой на очередном шаге. Использованное в процедуре условие `номер_вершины <= 7` позволяет ограничить рекурсивные вызовы процедуры (в нашей задаче число 7 — максимальный номер вершины, имеющей потомков).

Можно также в процедуре применить оператор цикла с параметром:

```
алг Очередной_шаг(арг цел номер_вершины)
нач
| вывод номер_вершины
| если номер_вершины <= 7
| | то
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
кон
```

Основная часть программы выглядит так:

```
алг Вывод_номеров_всех_вершин
нач
| Очередной_шаг(1)
кон
```

В этом простейшем примере для нас представляет интерес последовательность вывода номеров вершин. Она будет следующей:

```
1  2  4  8  9  5  10  11  3  6  12  13  7  14  15
```

(убедитесь в этом и проанализируйте порядок рекурсивных вызовов процедуры `Очередной_шаг`).

В общем случае для дерева, каждая вершина которого имеет 2 потомка[57], соответствующая процедура оформляется так:

```
алг Очередной_шаг(арг цел номер_вершины)
нач
| вывод номер_вершины
| если номер_вершины <= максимальный_номер_вершины_с_потомками
| | то
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
кон
```

где `максимальный_номер_вершины_с_потомками` — это величина, смысл которой ясен из ее имени.

Составьте процедуры для вывода номеров всех вершин дерева, в котором количество потомков в каждой вершине равно: а) трем; б) целому числу n.

Научившись обходить все вершины дерева, мы можем решать задачи, связанные с различной обработкой данных, представленных в виде деревьев.

Задача 1. Каждой вершине дерева, представленного на рис. 14.1, сопоставлено число, записанное в таблице:

Номер вершины	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Число	10	22	16	11	45	25	25	4	10	7	8	25	10	1	9

Определите сумму всех чисел в вершинах (обойдя их так, как это делалось в предыдущем примере).

Решение

Числа, приведенные в первой строке этой таблицы, мы будем называть «пометками» в вершинах. В программе мы их будем хранить в массиве `a` из 15 элементов. Тогда рекурсивная процедура `Очередной_шаг`,

[57] Разумеется, кроме вершин, находящихся на последнем уровне дерева.

суммирующая значения «пометок» во всех вершинах[58], при соблюдении требований условия задачи должна иметь вид:

```
алг Очередной_шаг(арг цел номер_вершины)
нач
| |Обрабатываем текущую вершину
| сумма_чисел := сумма_чисел + a[номер_вершины]
| если номер_вершины <= 7
| | то
| |  |Обрабатываем ее потомков
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
кон
```

Примечание. Величина `сумма_чисел` и массив `a` должны быть описаны как глобальные.

Основная часть программы оформляется следующим образом:

```
цел таб a[1:15], цел сумма_чисел
алг Сумма_всех_чисел_в_вершинах_дерева
нач
| a[1] := 10; a[2] := 22; ...
| сумма_чисел := 0
| Очередной_шаг(1)
| вывод нс, "Сумма всех пометок равна ", сумма_чисел
кон
```

Для дерева, использованного в задаче 1, определите:

1) сумму «пометок», которые больше 15;

2) количество четных «пометок»;

3) среднее арифметическое нечетных «пометок»;

4) имеется ли в дереве «пометка», равная 100.

Примечание. При решении всех задач обход вершин должен осуществляться так, как это делалось в предыдущих примерах.

Задача 2. Для дерева, использованного в задаче 1, дойдите до вершины дерева с «пометкой» 25 и выведите ее номер.

---

[58] Для расчета суммы «пометок» также можно составить рекурсивную функцию.

Процедура Очередной_шаг для этой задачи оформляется следующим образом:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| если a[номер_вершины] = 25
| | то
| |  вывод номер_вершины
| | иначе
| |  |Продолжаем рекурсивные вызовы
| |  если номер_вершины <= 7
| |  | то
| |  |  нц для i от 1 до 2
| |  |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  |  кц
| |  все
| все
кон
```

Основная часть программы:

```
цел таб a[1:15]
a[1] := 10; a[2] := 22; ...
алг Задача_2
нач
| вывод нс, "Число 25 — это пометка в вершине номер "
| Очередной_шаг(1)
кон
```

Выполнив программу, можно обнаружить, что на экран будет выведен не только номер первой вершины (6), имеющей «пометку» 25, но и номер 7, а вот номер 12 выведен не будет. Это объясняется тем, что после выполнения процедуры Очередной_шаг с параметром, равным 6, рекурсивные вызовы «из нее» происходить не будут, но будет осуществлен вызов этой процедуры с параметром, равным 7 (поскольку процедуры Очередной_шаг(6) и Очередной_шаг(7) вызываются из процедуры Очередной_шаг(3)).

Чтобы получить номер только одной из вершин с «пометкой» 25, следует использовать глобальную величину логического типа с именем встретилось25:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| если a[номер_вершины] = 25 и не встретилось25
| | то
| |  встретилось25 := да
| |  вывод номер_вершины
```

```
| | иначе
| |  если номер_вершины <= 7
| |  | то
| |  |  нц для i от 1 до 2
| |  |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  |  кц
| |  все
| все
кон
```

Обратите внимание на место в процедуре, в котором применена величина `встретилось25`.

Для дерева, использованного в задаче 1:

1) выведите номера всех потомков вершин, которым сопоставлено число 25;

2) выведите номера всех потомков первой встретившейся при обходе дерева вершины, «пометка» которой равна 25.

Задача 3. Для дерева, использованного в задаче 1, определите сумму чисел, сопоставленных всем «маршрутам», которые связывают корень дерева с вершинами на его последнем уровне (с номерами 8–15).

Идея решения этой задачи следующая: при перемещении по вершинам дерева надо подсчитывать сумму соответствующих «пометок», а после достижения последнего уровня дерева — вывести полученную сумму. С учетом этого начальный фрагмент процедуры `Очередной_шаг` оформить несложно:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| сумма_чисел := сумма_чисел + a[номер_вершины]
| если номер_вершины > 7
| | то |Достигнут конец одного из "маршрутов"
| |  вывод нс, сумма_чисел
| | иначе
| |  |Продолжаем рекурсивные вызовы
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
| ...
```

Первый «маршрут» перемещения по дереву, для которого будет найдена и выведена на экран искомая сумма: 1–2–4–8. После выполнения процедуры Очередной_шаг(8) будет вызвана процедура Очередной_шаг(9), при выполнении которой будет достигнут конец второго «маршрута» (1–2–4–9), подсчитана и выведена вторая искомая сумма. Однако в ней будет учтено число, сопоставленное вершине 8, что неправильно. Чтобы устранить этот недостаток, надо перед окончанием работы процедуры Очередной_шаг(8) вычесть из найденной суммы это число. Аналогично надо вычитать уже учтенные в итоговой сумме «пометки» и в других вершинах дерева:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| сумма_чисел := сумма_чисел + a[номер_вершины]
| если номер_вершины > 7
| | то |Достигнут конец одного из "маршрутов"
| |  вывод нс, сумма_чисел
| | иначе
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
| |Исключаем из суммы число,
| |сопоставленное вершине, в которой находимся
| сумма_чисел := сумма_чисел - a[номер_вершины]
кон
```

Для дерева, изображенного на рис. 14.1, сформируйте все «маршруты», которые проходят от «корня» дерева до его последнего уровня.

Задача 4. Для дерева, использованного в задаче 1, определите минимальную из сумм, указанных в условии задачи 3.

Программа для решения этой задачи имеет вид:

```
цел таб a[1:15],
цел сумма_чисел, минимальная_сумма
алг Задача_4
нач
| a[1] := 10; a[2] := 22; ...
| сумма_чисел := 0
| минимальная_сумма := 10000 |Условно
| Очередной_шаг(1)
| вывод нс, минимальная_сумма
кон
```

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| сумма_чисел := сумма_чисел + a[номер_вершины]
| если номер_вершины > 7
| | |Достигнут конец одного из "маршрутов"
| | то
| |  если сумма_чисел < минимальная_сумма
| |  | то
| |  |  минимальная_сумма := сумма_чисел
| |  все
| | иначе
| |  нц для i от 1 до 2
| |  | Очередной_шаг(номер_вершины * 2 + i - 1)
| |  кц
| все
| сумма_чисел := сумма_чисел - a[номер_вершины]
кон
```

Из точки *А* в точку *Б* можно попасть по линиям, изображенным на рис. 14.5 (числа на рисунке соответствуют расстоянию между точками, обозначенными закрашенными кружочками). Найдите длину самого короткого маршрута от точки *А* до точки *Б*.

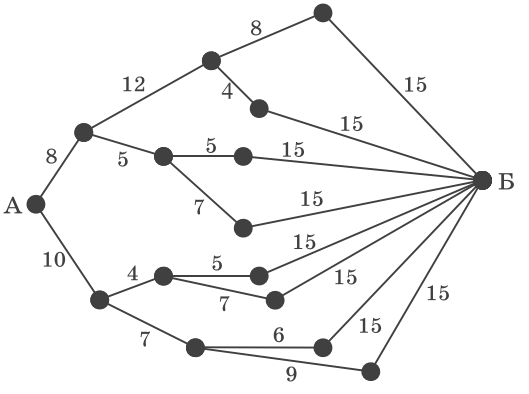


Рис. 14.5

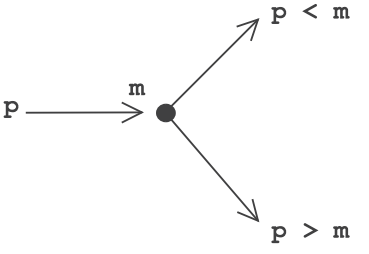
Теперь введем новые термины. Изображенное на рис. 14.1 дерево называют «*полным двоичным деревом*». Для каждой его i-й вершины определено *первое поддерево* — часть дерева, которое начинается с потомка с номером 2 × i, и *второе поддерево* — часть дерева, которое начинается с потомка с номером 2 × i + 1.

Двоичным упорядоченным деревом называют дерево, для которого справедливо следующее свойство: для любой i-й вершины все «пометки» в первом поддереве меньше «пометки» в i-й вершине, а все «пометки» во втором поддереве больше «пометки» в i-й вершине. Например, упорядоченным является дерево, изображенное на рис. 14.1, при «пометках», перечисленных в таблице:

Номер вершины	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
«Пометка»	20	10	26	8	18	24	30	1	9	11	19	22	23	28	50

Для дерева, описанного выше, найдите номер вершины с «пометкой», равной p (считая, что такое число среди «пометок» есть). Обход всех вершин дерева и последовательный просмотр элементов массива не использовать.

Указания по выполнению задания. Благодаря упорядоченности дерева число p в нем можно быстро найти следующим способом: придя в какую-либо вершину с «пометкой» m, можно определить, найдена ли искомая вершина, или в какое поддерево (первое или второе) нужно идти дальше:



при **p = m** вершина найдена

Начав с корня и двигаясь по этому правилу, можно всегда найти соответствующую вершину.

До этого были рассмотрены случаи, когда у каждой вершины дерева имелось одинаковое количество потомков (два). А как быть, если оно различно (рис. 14.6)? Как обойти и обработать все вершины дерева в этом случае?

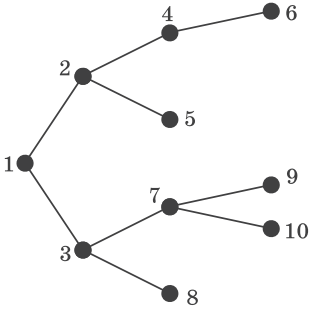


Рис. 14.6

Главная идея решения подобной задачи та же, что и раньше, — обработав каждую вершину, следует выполнить то же для каждого потомка этой вершины. Но как определить номера потомков? Ведь четкая зависимость между номером вершины-предка и номерами ее потомков в данном случае отсутствует! Здесь нам на помощь приходит следующая таблица, в которой отражена информация о связях вершин-предков и вершин-потомков:

Номер вершины-предка	Порядковый номер вершины-потомка	
	Первого	Второго
1	2	3
2	4	5
3	7	8
4	6	0
5	0	0
6	0	0
7	9	10
8	0	0
9	0	0
10	0	0

Примечание. Число 0 указывает, что соответствующий потомок для данной вершины отсутствует.

Обозначим порядковый номер потомка как `i` (в рассматриваемом случае `i` равно 1 или 2) и будем рассуждать так. Для каждого номера `i` проверяем, существует ли такой потомок у рассматриваемой вершины, и в случае положительного ответа рекурсивно переходим к обработке этого потомка, номер которого можно определить по приведенной таблице.

Разработаем соответствующую программу. Для хранения данных о порядковых номерах вершин-потомков из только что приведенной таблицы используем двумерный массив a из десяти строк и двух столбцов:

```
2   3
4   5
7   8
6   0
0   0
0   0
9  10
0   0
0   0
0   0
```

С использованием подобного массива рекурсивная процедура обхода вершин дерева и вывода их номеров оформляется следующим образом:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| вывод номер_вершины
| если номер_вершины <= 10
| | то
| |  нц для i от 1 до 2
| |  | |Если i-й потомок существует
| |  | если a[номер_вершины, i] <> 0
| |  | | то |Обрабатываем его
| |  | |  |Номер вершины — этого потомка
| |  | |  |равен a[номер_вершины, i]
| |  | |  Очередной_шаг(a[номер_вершины, i])
| |  | все
| |  кц
| все
кон
```

Вызов этой процедуры в основной части программы должен быть таким:

```
Очередной_шаг(1)
```

Еще раз обратим внимание, что зависимость между номером вершины-предка и номерами ее потомков в рассмотренном случае отсутствовала, но это не помешало нам решить задачу благодаря использованию двумерного массива, в котором отражена информация о связях вершин-предков и вершин-потомков.

Аналогично можно решить рассмотренную задачу и в случае, когда максимально возможное количество потомков в вершине равно Kmax. В этом случае структура таблицы будет выглядеть так:

Номер вершины-предка	Порядковый номер вершины-потомка			
	1	2	...	Kmax
1				
2				
...				
Ny				

Здесь Ny — общее количество вершин в дереве, а массив a должен иметь описание:

```
цел таб a[1:Ny, 1:Kmax]
```

При этом процедура Очередной_шаг оформляется следующим образом:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| вывод номер_вершины
| если номер_вершины <= Ny
| | то
| |  нц для i от 1 до Kmax
| |  | если a[номер_вершины, i] <> 0
| |  | | то
| |  | |  Очередной_шаг(a[номер_вершины, i])
| |  | все
| |  кц
| все
кон
```

Решенная выше задача позволит нам решать и другие задачи, в которых вершины дерева имеют произвольную нумерацию (рис. 14.7 — такие деревья в дальнейшем мы будем называть *произвольными*).

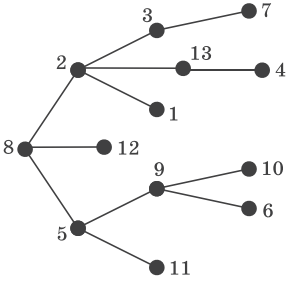


Рис. 14.7

Для представленного на рис. 14.7 произвольного дерева таблица, в которой отражена информация о связях вершин-предков и вершин-потомков, имеет вид:

Номер вершины-предка	Порядковый номер вершины-потомка		
	Первого	Второго	Третьего
1	0	0	0
2	3	13	1
3	7	0	0
4	0	0	0
5	9	11	0
6	0	0	0
7	0	0	0
8	2	12	5
9	10	6	0
10	0	0	0
11	0	0	0
12	0	0	0
13	4	0	0

На основе этой таблицы для хранения данных о порядковых номерах вершин-потомков в программе также может быть сформирован двумерный массив.

Вызов процедуры `Очередной_шаг` в основной части программы для рассматриваемого случая должен быть таким:

```
Очередной_шаг(8)
```

После решения нескольких последних задач нетрудно увидеть, что таким способом можно решать практически любые задачи по обработке данных, представленных в виде произвольного дерева, в котором вершинам дерева сопоставлены какие-либо значения («пометки») — числовые (как ранее) или текстовые (рис. 14.2–14.4). Что нового вносит это обстоятельство в методику решения таких задач? Только одно — для хранения информации кроме массива, моделирующего связи вершин-предков и вершин-потомков, следует использовать еще один одномерный массив с «пометками».

Рекурсивная процедура `Очередной_шаг` в этом случае будет отличаться от приведенных выше только тем, что в ней должны обрабатываться «пометки», соответствующие каждой вершине. Например,

вывод всех «пометок» при рекурсивном обходе дерева может быть проведен следующим способом:

```
алг Очередной_шаг(арг цел номер_вершины)
нач цел i
| вывод b[номер_вершины]
| если номер_вершины <= Ny
| | то
| |  нц для i от 1 до Kmax
| |  | если a[номер_вершины, i] <> 0
| |  | | то
| |  | |  Очередной_шаг(a[номер_вершины, i])
| |  | все
| |  кц
| все
кон
```

где `Ny` — общее число вершин в дереве, `Kmax` — максимально возможное количество потомков, а `b` — массив с «пометками» в вершинах.

1. Информация о структуре файлов и папок на компьютере представлена на рис. 14.2. Выведите на экран полный путь к каждому файлу, имеющемуся на жестком диске.
2. Информация о детях, внуках и т. д. человека по имени Михаил Васильевич показана на рис. 14.4. Определите, встречается ли у его потомков имя Мефодий. (Рассмотрите оба возможных случая — наличие этого имени и его отсутствие.)

   Примечание. Остальные сведения, необходимые для решения (кроме представленных на рисунках), добавьте самостоятельно.

Завершая обсуждение задач, заметим, что применяемый при их решении подход, по сути, представляет собой метод полного перебора всех возможных вариантов, поэтому он не может быть использован при большом количестве вершин дерева. Заметим также, что для многих задач обработки данных, представленных в древовидной структуре, существуют специальные, более эффективные способы решения (метод ветвей и границ [8], поиск в глубину и др.). Их читатели могут изучить самостоятельно.

Литература

1. *Кушниренко А. Г., Лебедев А. Г., Зайдельман Я. Н.* Информатика 7–9: Учебник для общеобразовательных учебных заведений. — М.: Дрофа, 2000.
2. Компьютерная игра «Две кучки спичек» // Информатика. 2004. № 40.
3. *Златопольский Д. М.* Сборник задач по программированию. — СПб.: БХВ-Петербург, 2007.
4. *Вьюкова Н. И., Галатенко В. А., Ходулев А. Б.* Систематический подход к программированию. — М.: Наука, 1988.
5. *Кнут Д.* Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. — М.: Мир, 1978.
6. *Вирт Н.* Алгоритмы + структуры данных = программы. — М.: Мир, 1985.
7. *Вентцель Е. С.* Исследование операций: задачи, принципы, методология. — М.: Наука, 1988.
8. *Окулов С. М.* Программирование в алгоритмах. — М.: БИНОМ. Лаборатория знаний, 2002.
9. *Липский В.* Комбинаторика для программистов. — М.: Мир, 1988.
10. *Выгодский М. Я.* Справочник по элементарной математике. — М.: Наука, 1989.
11. *Андреева Е. В.* Комбинаторные задачи / Библиотечка «Первого сентября». Серия «Информатика». — М., 2004.
12. *Усенков Д. Ю.* Рекурсивный генератор перестановок // Информатика. 1996. № 12.

Содержание

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Учебное электронное издание

Златопольский Дмитрий Михайлович

ПРОГРАММИРОВАНИЕ:
ТИПОВЫЕ ЗАДАЧИ, АЛГОРИТМЫ, МЕТОДЫ

Ведущий редактор *Д. Усенков*
Художник *Н. Лозинская*
Технический редактор *Е. Денюкова*
Корректор *Е. Клитина*
Компьютерная верстка: *С. Янковой*

Подписано к использованию 09.09.19.
Формат 155×225 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

Студентам, школьникам, преподавателям — всем, кто серьезно увлекается информатикой и программированием, хочет сделать их своей профессией, не ограничивается рамками стандартных образовательных программ, издательство «Лаборатория знаний» предлагает следующие книги

С. М. Окулов
Основы программирования

С. М. Окулов
Программирование в алгоритмах

И. А. Бабушкина, С. М. Окулов
Практикум
по объектно-ориентированному программированию

С. М. Окулов
Задачи по программированию

С. М. Окулов
Дискретная математика:
теория и практика решения задач по информатике

М. А. Плаксин
Тестирование и отладка программ
для профессионалов будущих и настоящих

Л. А. Залогова
Разработка Паскаль-компилятора