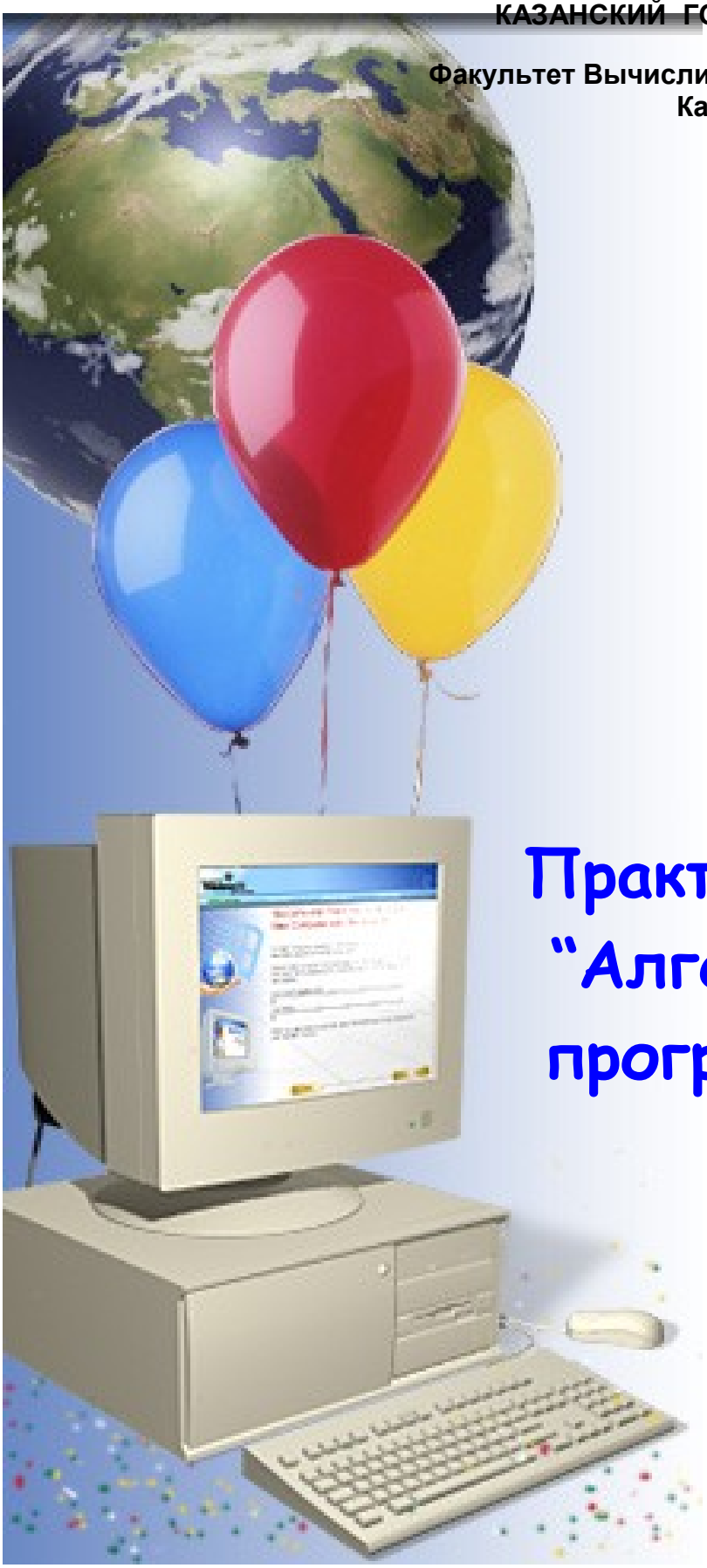
КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной Математики и Кибернетики
Кафедра Экономической Кибернетики



АНДРИАНОВА А.А.,
МУХТАРОВА Т.М.

Практикум по курсу "Алгоритмизация и программирование"

Часть 1

КАЗАНЬ - 2008

УДК 004.43

ББК 32.973.26 – 018.1

Печатается по постановлению редакционно-издательского совета
факультета вычислительной математики и кибернетики
Казанского государственного университета

Рецензенты:

кандидат физико-математических наук, заведующий кафедрой информационных технологий Института экономики, управления и права (г.Казань) **И.А. Фукин**

старший преподаватель кафедры системного анализа и информационных технологий Казанского государственного университета **Р.Р. Тагиров**

Андрианова А.А., Мухтарова Т.М.
Практикум по курсу “Алгоритмизация и программирование”. Часть 1: Учебное пособие / А.А. Андрианова, Т.М. Мухтарова. – Казань: Казанский государственный университет, 2008. – 95 с.

В данном учебном пособии рассматриваются принципы построения алгоритмов для решения классических задач программирования – разбираются основные приемы программирования, а также алгоритмы с использованием наиболее простых структур данных – массивов, символьных строк и двумерных массивов. Данное пособие предназначено для ведения практических занятий для студентов 1 курса специальностей 061800 “Математические методы в экономике” и 080700 “Бизнес-информатика”, а также может использоваться студентами других специальностей, начинающих изучать программирование.

Содержание

Предисловие

Высокие темпы развития вычислительной техники привели в последнее время к пересмотру государственного образовательного стандарта среднего образования по информатике. Главный акцент в школьной программе сейчас ставится на выработку навыков использования информационных технологий для простейшей обработки данных, для поиска информации в глобальных сетях, и лишь малая часть посвящена принципам построения алгоритмов и программированию. Проблема усугубляется тем, что в настоящее время достаточно мало учебников по информатике, в которых на очень простом уровне объясняются “азы” алгоритмизации, т.е. принципы построения алгоритмов решения задач, поэтому и было задумано это пособие.

В данном учебном пособии рассмотрены базовые алгоритмы, которые являются основой для решения более сложных задач программирования. Рассматриваются основные технологические приемы построения таких алгоритмов. Учебное пособие создано в поддержку проведения практических занятий для курсов “Алгоритмизация и программирование” и “Информатика и программирование” для студентов 1 курса факультета ВМК специальностей 061800 “Математические методы в экономике” и 080700 “Бизнес-информатика”, а также предназначено для студентов младших курсов и абитуриентов, начинающих изучать программирование.

Учебное пособие состоит из двух частей. В первой части будут рассмотрены принципы построения алгоритмов, проиллюстрированы основные понятия алгоритмизации (ветвление, циклы различных видов), а также рассмотрены базовые алгоритмы работы с основными структурами данных (массивами, символьными строками, матрицами).

Вторая часть учебного пособия будет посвящена структурированию программ (созданию пользовательских функций), а также работе с более сложными структурами данных (списками, деревьями, графами).

Введение

Алгоритм – это точно определенное описание способа решения задачи в виде конечной последовательности действий, ведущей от варьируемых начальных данных к искомому результату. Основное требование при составлении алгоритма заключается в его применимости для решения всех частных случаев решаемой задачи.

Алгоритмы реализуются на языках программирования. Все языки программирования являются искусственными языками, т.е. оперируют очень небольшим количеством команд, которые являются своеобразными “кирпичиками” программирования. В способности строить алгоритмы любой сложности из этих “кирпичиков” и заключается искусство программирования.

Помимо языков программирования существует еще целый ряд способов записи алгоритма, к числу которых относятся, в частности, блок-схемы и псевдокод. В пособии для этих целей будут использоваться блок-схемы, поскольку они являются наиболее наглядной и понятной формой записи алгоритма.

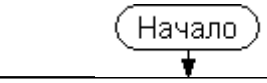
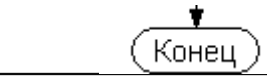
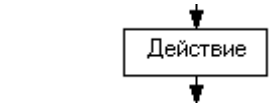
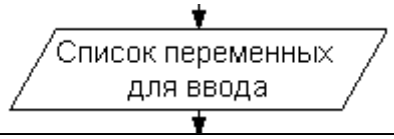
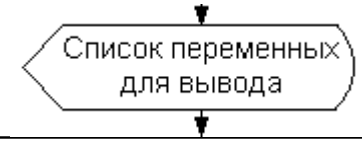
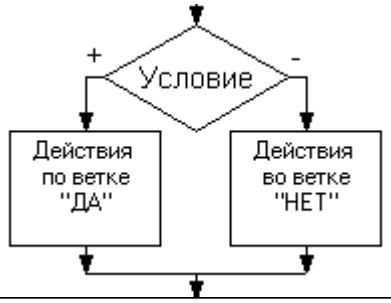
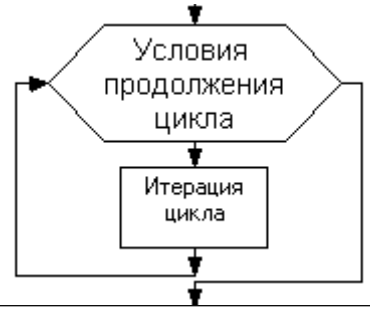
Блок-схема – это графическая интерпретация алгоритма. Каждый вид действия в ней обозначается с помощью некоторой геометрической фигуры (блока), внутри которой делается запись, конкретизирующая смысл действия. Блоки блок-схемы связаны с помощью линий потока, определяющих ход выполнения вычислительного процесса. Поток начинается в специальном блоке начала алгоритма и заканчивается в блоке конца алгоритма. Обозначения основных блоков, используемых в блок-схеме, приведены в таблице 1.

Часть 1 учебного пособия состоит из двух глав. Первая глава посвящена рассмотрению основных конструкций построения алгоритмов. Она разделена на три раздела: “Блок условия”, “Цикл “пока” (цикл с условием)”, “Цикл “для” (цикл с параметром)”. Во второй главе приводятся основные алгоритмы работы с наиболее используемыми структурами данных. Эта глава также разделена на три раздела, посвященных работе с массивами, символьными строками и двумерными массивами (матрицами).

В каждом разделе дается теоретическое описание приемов работы с рассматриваемыми конструкциями, подробно разбираются несколько задач, а также приводится список задач для самостоятельного решения (в некоторых случаях с методическими указаниями). Задачи имеют собственную нумерацию внутри разделов. Номера иллюстраций имеют нумерацию вида

“Номер_раздела.Номер_иллюстрации”. Например, Рис.1.2 – это второй по порядку рисунок, находящийся в Разделе 1.

Таблица 1. Обозначения основных блоков блок-схемы.

	Блок начала блок-схемы
	Блок конца блок-схемы
	Блок присваивания, например, вычисление значения некоторой переменной.
	Блок ввода значений. Значения вводятся в перечисленные в нем переменные
	Блок вывода значений перечисленных в блоке переменных
	Условный блок, или блок ветвления. В зависимости от условия алгоритм разделяется на две ветви – ветвь выполнения условия (ветвь “ДА”) и ветвь нарушения условия (ветвь “НЕТ”).
	Блок цикла. Это блок повторения некоторой последовательности действий (итерации цикла). Должно быть предусмотрено наличие условия выхода из цикла.

При разборе каждой задачи приводится ее постановка, краткое обсуждение используемого алгоритма решения, блок-схема алгоритма, а

также программа с комментариями, написанная на языке программирования C++. Программы отлажены с помощью оболочки проектирования Microsoft Visual Studio 6.0. Для некоторых задач приведены два алгоритма решения. Для создания блок-схем задач был использован редактор FCEditor 1.6.0.117 (http://www.fceditor.nm.ru).

Поскольку программы, приведенные в пособии, написаны на языке C++, в блок-схемах используются конструкции этого языка для обозначения некоторых операций. Эти обозначения указаны в таблице 2.

Таблица 2. Обозначения некоторых операций и символов.

%	Операция вычисления остатка от деления
/	Операция деления, а также операция целочисленного деления в случае целых операндов
!=	Операция сравнения “не равно”
[]	Получение элемента массива по индексу
‘\0’	EOL – признак конца строки
sqrt(x)	Функция вычисления квадратного корня из числа x

Глава 1. Простейшие алгоритмические конструкции

Раздел 1. Блок условия

Достаточно часто возникает ситуация, когда результат работы алгоритма зависит от выполнения того или иного условия. В этом случае требуется выполнить одну из нескольких альтернативных последовательностей действий. Для задания возможных альтернатив используется блок условия (или, как его еще называют, блок ветвления, логический блок).

Основой этой конструкции является логическое условие. Если условие истинно, выполняются команды, следующие по ветви “Да” (+). В противном случае – команды из ветви “Нет” (–). Блок условия завершен, когда обе ветви соединяются для продолжения единого вычислительного процесса.

Например, пусть даны две переменные **a** и **b**. Если **a** < **b**, переменная **c** должна принимать значение 1, иначе – значение 2. Фрагмент блок-схемы с таким условным блоком приведен на Рис.1.1.

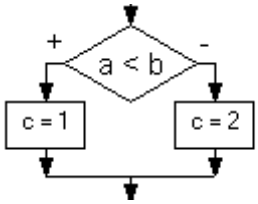


Рис. 1.1. Пример блока условия (логического блока).

Задача 1. Вычислить значение функции

$$y = sign(x) = \begin{cases} 1, если\ x > 0 \\ -1, если\ x < 0 \\ 0, если\ x = 0 \end{cases}$$

Алгоритм решения данной задачи очень прост. Сначала требуется ввести значение переменной **x**, которая является аргументом функции. Далее следуют два блока условия для выделения трех вариантов вычисления функции. Результат вычислений распечатывается.

Блок-схема решения задачи 1 приведена на Рис.1.2.

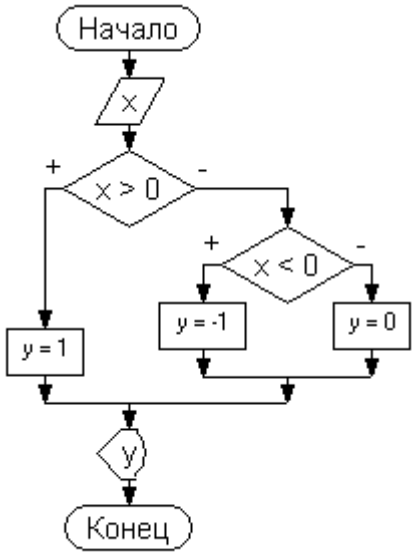


Рис.1.2. Блок-схема решения задачи вычисления функции sign(x).

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
{
      float x,y;
      printf("Введите  x:");
      scanf("%f",&x);
      if(x>0)
            y=1;
      else
            if(x<0)
                  y=-1;
            else y=0;
      printf("y=sign(%f)=%f\n",x,y);
}
```

Нередко бывает нужно осуществить выбор, заданный группой условий. Эти условия можно объединить с помощью логических операций. Существуют три основные логические операции – логическое НЕ, логическое ИЛИ и логическое И. С их помощью можно комбинировать несколько условий в блоке ветвления. Приоритет выполнения логических операций можно изменить с помощью круглых скобок.

Задача 2. Определить, принадлежит ли точка заштрихованной области, изображенной на Рис.1.3.

Заштрихованная область образуется пересечением двух множеств -

$$\{(x,y): x^2 + (y-1)^2 \le 1\} \text{ и } \{(x,y): y \le 1 - x^2\}.$$

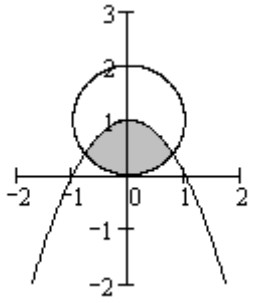


Рис.1.3. Область из задачи 2.

Для проверки, принадлежит ли точка одному множеству, требуется проверить, выполняется ли условие из определения этого множества. Для проверки принадлежности точки сразу двум множествам удобно использовать операцию **логическое И** (результат операции **a И b** будет истинным, если истинными являются одновременно и условие **a**, и условие **b**). Поэтому условие проверки принадлежности точки (x, y) заштрихованной области записывается так:

$$x^2 + (y-1)^2 \le 1 \;\; И \;\; y \le 1 - x^2 .$$

Блок-схема решения задачи 2 представлена на Рис.1.4.

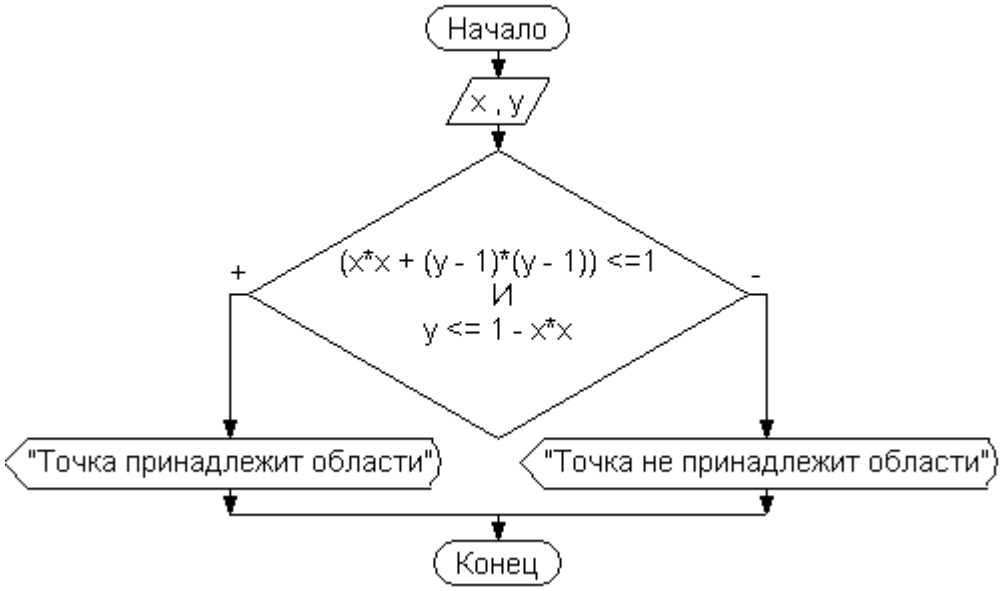


Рис.1.4. Блок-схема решения задачи о принадлежности точки области, изображенной на Рис.1.3.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
```

```
        float x,y;
        printf("Введите x:");
        scanf("%f",&x);
        printf("Введите y:");
        scanf("%f",&y);
        if(x*x+(y-1)*(y-1)<=1 && y<=1-x*x)
             printf("Точка принадлежит области\n");
        else
             printf("Точка не принадлежит области\n");
}
```

Задача 3. Определить, принадлежит ли точка заштрихованной области, изображенной на Рис.1.5.

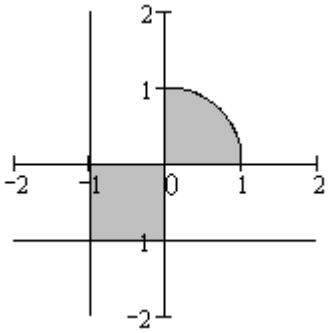


Рис. 1.5. Область из задачи 3.

В данном случае множество состоит из объединения двух множеств –

$$\{(x,y): x^2+y^2 \le 1, x \ge 0, y \ge 0\} \quad \text{и} \quad \{(x,y): -1 \le x \le 0, -1 \le y \le 0\}.$$

Точка принадлежит заштрихованной области, если она принадлежит хотя бы одному из этих двух множеств.

Для формулирования условия здесь подходит операция **логического ИЛИ**. Результат операции **a ИЛИ b** будет истинным, если выполняться будет хотя бы одно из условий **a** или **b**. Таким образом, условие имеет вид

$$(x^2+y^2 \le 1 \; И \; x \ge 0 \; И \; y \ge 0) \; ИЛИ \; (-1 \le x \le 0 \; И \; -1 \le y \le 0).$$

Блок-схема решения задачи 3 приведена на Рис.1.6.

Код программы для задачи 3.

```
# include <stdio.h>
void main(void)
{
     float x,y;
     printf("Введите x:");
     scanf("%f",&x);
     printf("Введите y:");
     scanf("%f",&y);
     if((x>=0 && y>=0 && x*x+y*y<=1)||
          (x<=0 && y<=0 && x>=-1 && y>=-1))
```

```
                printf("Точка принадлежит области\n");
        else
                printf("Точка не принадлежит области\n");
    }
```

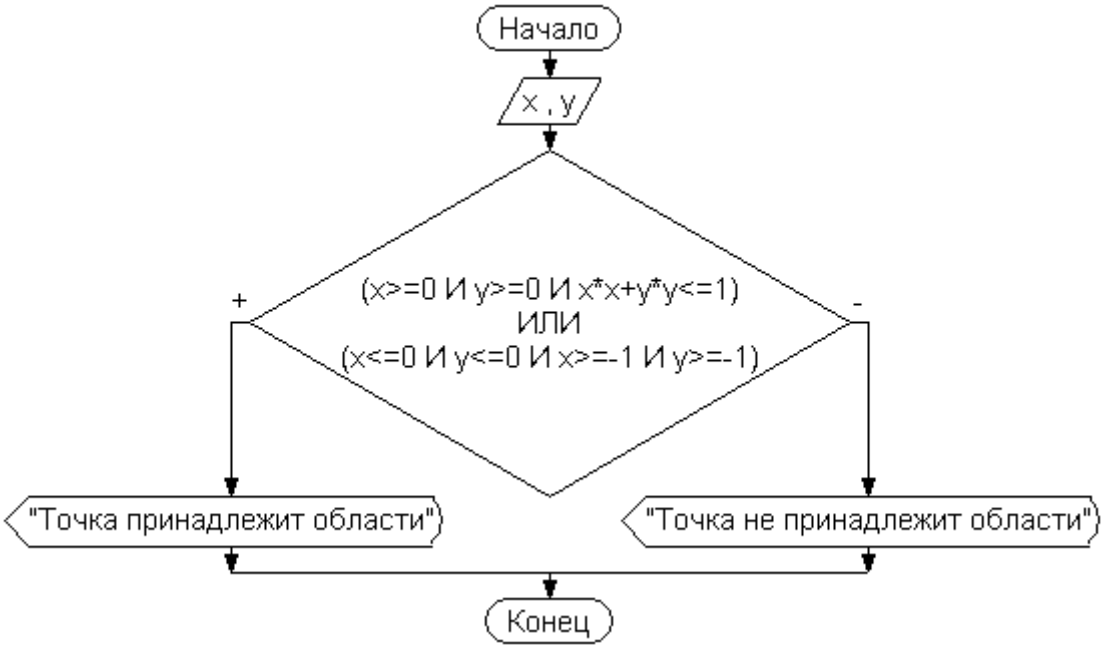


Рис.1.6. Блок-схема решения задачи о принадлежности точки области, изображенной на Рис.1.5.

Задача 4. Определить, принадлежит ли точка заштрихованной области, изображенной на Рис.1.7.

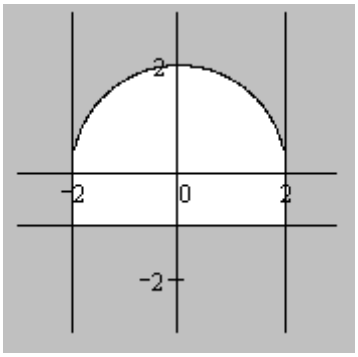


Рис.1.7. Область из задачи 4.

В данном случае проще описать множество точек, которые не входят в допустимое множество – это объединение множеств

$$\{(x,y): x^2 + y^2 < 4,\ y \ge 0\} \quad \text{и} \quad \{(x,y): -2 < x < 2, -1 < y \le 0\}.$$

Поэтому здесь лучше использовать операцию **логического НЕ** (если условие **а** ложно, то **(НЕ а)** истинно).

Блок-схема решения задачи 4 представлена на Рис.1.8.

Код программы для задачи 4.

```
# include <stdio.h>
void main(void)
{
     float x,y;
     printf("Введите x:");
     scanf("%f",&x);
     printf("Введите y:");
     scanf("%f",&y);
     if(!((x*x+y*y<4 && y>=0) ||
               (-2<x && x<2 && -1<y && y<=0)))
          printf("Точка принадлежит области\n");
     else
          printf("Точка не принадлежит области\n");
}
```

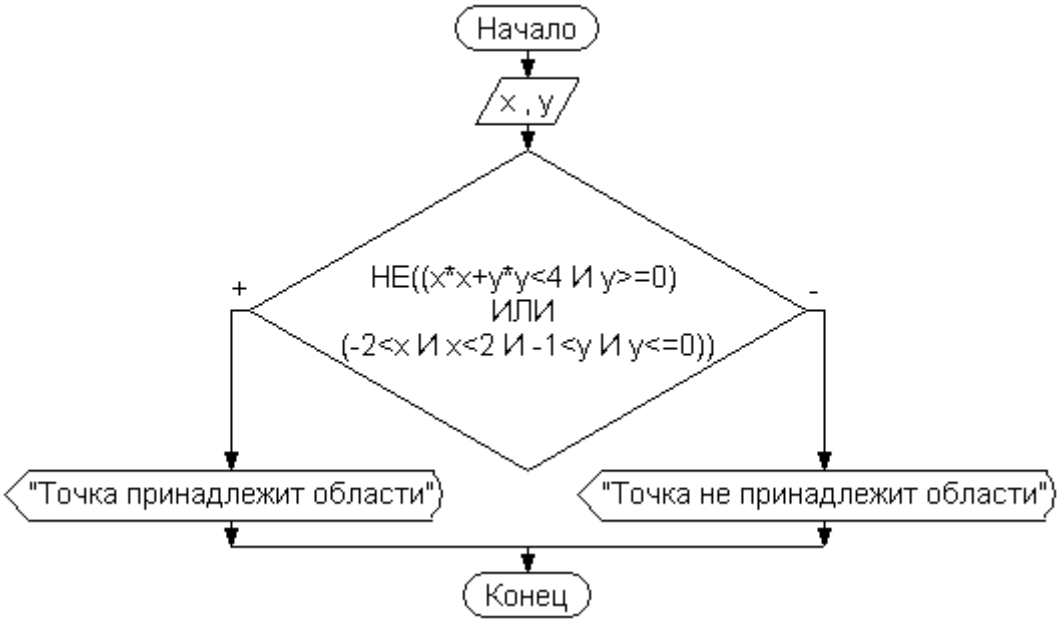


Рис.1.8. Блок-схема решения задачи о принадлежности точки области, изображенной на Рис.1.7.

Задача 5. Найти корни квадратного уравнения $ax^2 + bx \quad c + 0$.

Прежде чем приступать к поиску корней уравнения через вычисление дискриминанта, надо выяснить, не является ли нулем коэффициент a. При $a = 0$ уравнение становится линейным. Далее следует проверить на равенство нулю коэффициенты b и c. В зависимости от этого уравнение может не иметь решений (b=0, c≠0), иметь бесконечное множество решений ($b = 0, c \quad 0$) или иметь единственное решение (b≠0).

Если же a≠0 , надо вычислить дискриминант и найти корни обычного

квадратного уравнения.

Блок-схема решения задачи 5 представлена на Рис.1.9.

Код программы для задачи 5.

```
# include <stdio.h>
# include <math.h>
void main(void)
{
     float a,b,c;
     float D,x1,x2;
     printf("Введите коэффициенты уравнения:");
     scanf("%f%f%f",&a,&b,&c);
     if (a==0)
        if(b==0)
           if(c==0)
              printf("Бесконечное множество решений\n");
            else
               printf("Нет корней\n");
        else
        {
            x1= - c/b;
            printf("%f\n", x1);
        }
   else
   {
        D=b*b-4*a*c;
        if(D>0)
        {
             x1=(-b+sqrt(D))/(2*a);
             x2=(-b-sqrt(D))/(2*a);
             printf("%f %f\n",x1,x2);
        }
        else
             if (D==0)
             {
                  x1=-b/(2*a);
                  printf("%f\n",x1);
             }
             else
                  printf("Комплексные корни\n");
   }
}
```

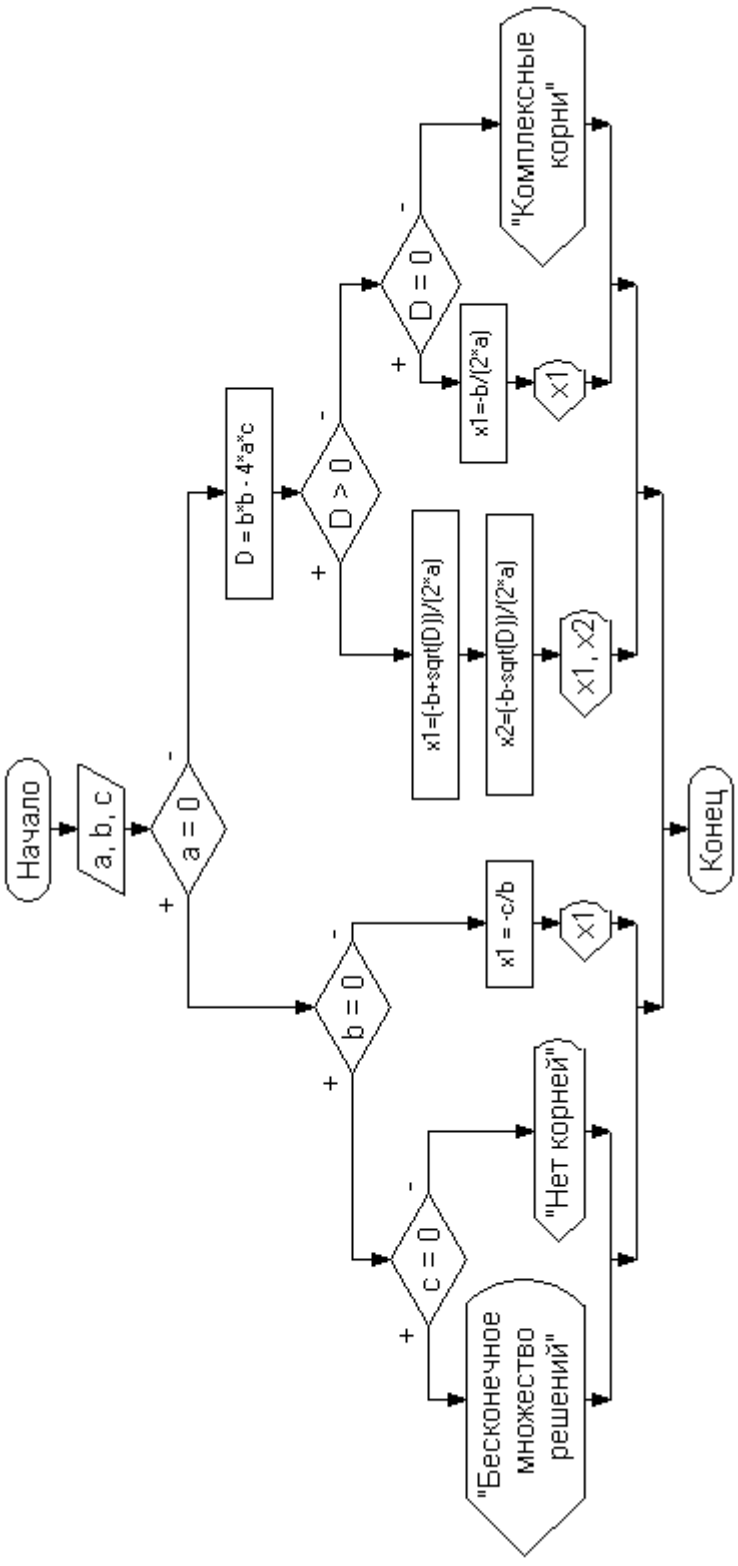


Рис.1.9. Блок-схема решения задачи о корнях квадратного уравнения.

Задача 6. Пусть пара чисел (x, y) задает координаты клетки шахматной доски. Пусть даны координаты двух клеток. Определить, являются ли они клетками одного цвета.

После ввода координат клеток следует проверить их корректность. Шахматная доска имеет размер 8x8. Следовательно, каждая из координат должна быть целым числом из интервала [1;8]. Если хотя бы одна из координат окажется некорректной, следует сообщить пользователю о неправильном вводе.

Заметим, что белые поля шахматной доски имеют четную сумму координат, т.е. $x + y$ – четное число, а черные поля – нечетную сумму.

Это следует из того, что при переходе от одной клетки некоторого цвета к другой сумма координат изменится на число, кратное 2. Например, при движении по горизонтали (вертикали) – клетки того же цвета расположены левее (выше) на 2, 4 и т.д, или правее (ниже) на 2, 4, т.е. меняется одна из координат на четное число. При движении по диагонали – либо сумма остается неизменной (одна координата увеличивается, другая – уменьшается на одно и то же число – движение по стрелке б) на Рис.1.10), либо сумма координат меняется на четное число (обе координаты меняются на одно и то же число, что означает неизменную разность координат – движение по стрелке а) на Рис.1.10).

Блок-схема решения задачи 6 представлена на Рис.1.11.

Код программы для задачи 6.

```
# include <stdio.h>
void main(void)
{
     int x1,y1,x2,y2;
     int s1,s2;
     printf("Введите координаты двух клеток
               шахматной доски:");
     scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
     if (!(x1>=1 && x1<=8 && y1>=1 && y1<=8 &&
           x2>=1 && x2<=8 && y2>=1 && y2<=8))
     {
          printf("Неправильно заданы координаты\n");
          return;
     }
     s1=x1+y1;
     s2=x2+y2;
     if (s1%2==s2%2)
          printf("Поля одного цвета\n");
     else
          printf("Поля разных цветов\n");
}
```

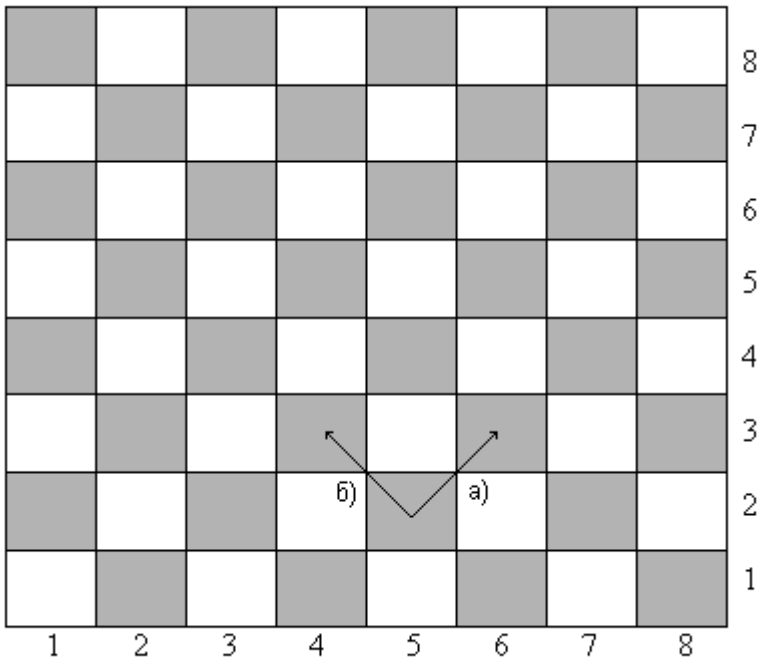


Рис.1.10. Схема шахматной доски.

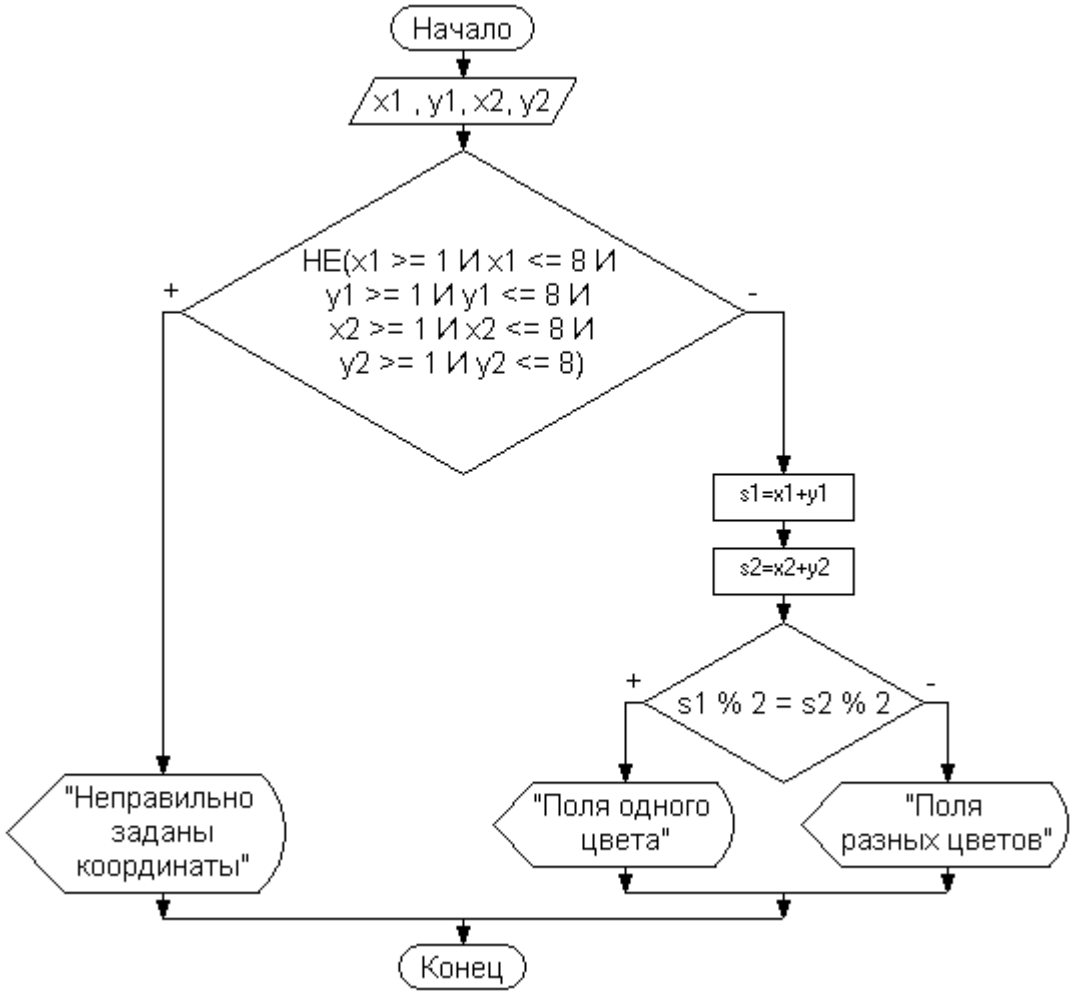


Рис.1.11. Блок-схема решения задачи о клетках шахматной доски.

Задача 7. Заданы координаты двух клеток шахматной доски. На первой из них расположен белый слон, на второй – фигура противника. Определить, бьет ли слон фигуру противника.

По правилам игры в шахматы слон бьет фигуру противника, если они находятся на одной диагонали. Через клетку, на которой стоит слон, может проходить одна или две диагонали. Условия проверки принадлежности двух клеток одной и той же диагонали обоснованы при объяснении задачи 6 - если сумма или разность координат обеих фигур совпадают, то слон бьет фигуру противника.

Блок-схема решения этой задачи изображена на Рис.1.12.

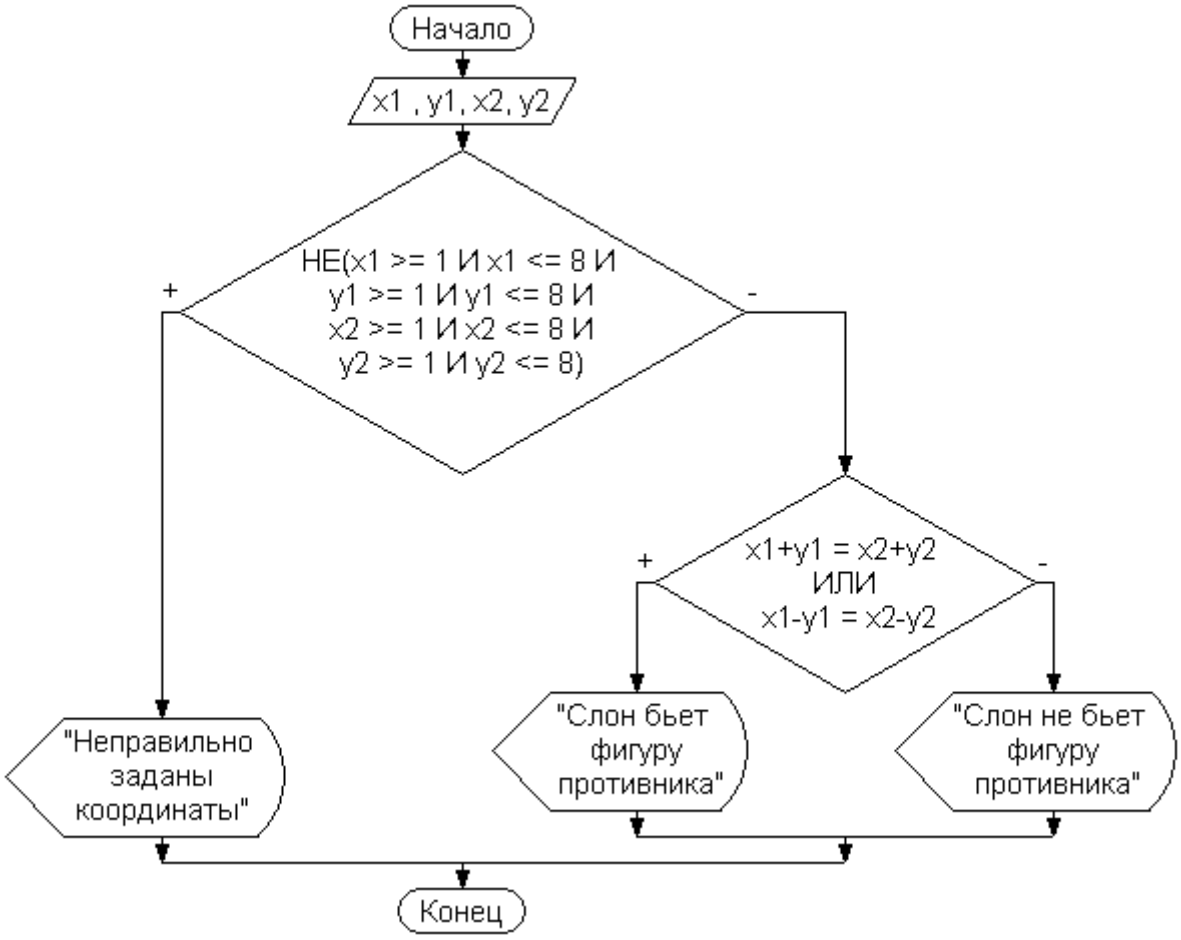


Рис.1.12. Блок-схема решения задачи о слоне и фигуре противника.

Код программы для задачи 7.

```
#include <stdio.h>
void main(void)
{
     int x1,y1,x2,y2;
     printf("Введите координаты двух клеток
               шахматной доски:");
     scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
     if (!(x1>=1 && x1<=8 && y1>=1 && y1<=8 &&
```

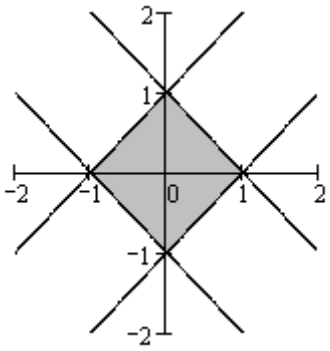
```
            x2>=1 && x2<=8 && y2>=1 && y2<=8))
    {
        printf("Неправильно заданы координаты\n");
        return;
    }
    if (x1+y1==x2+y2 || x1-y1==x2-y2)
        printf("Слон бьет фигуру противника\n");
    else
        printf("Слон не бьет фигуру противника\n");
}
```

Домашнее задание

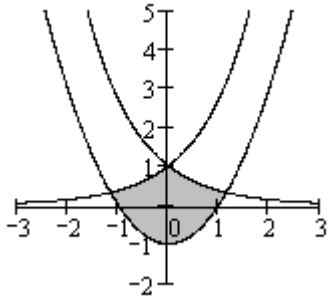
1. Определить, принадлежит ли точка (x, y) заштрихованной области, изображенной на Рис.1.13 (а,б,в,г,д).

2. Дана корректная дата в виде трех целых чисел. Получить дату следующего дня.

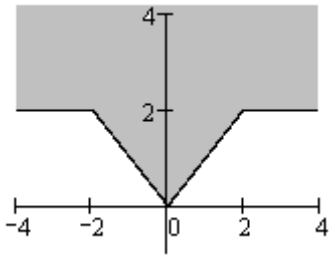
3. Дано целое число от 1 до 999. Распечатать его вместе со словом “рубль” в правильном падеже. Например, 32 рубля, 51 рубль, 112 рублей.



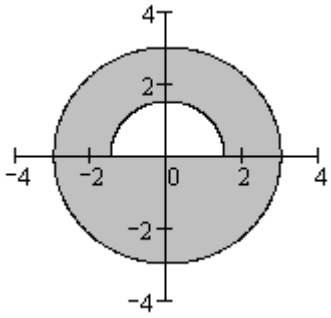
а)



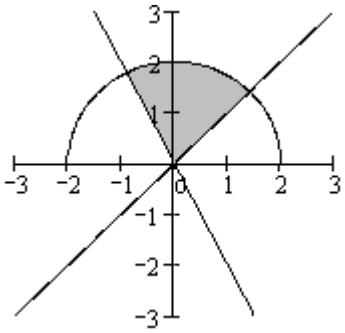
б)



в)



г)



д)

Рис.1.13. Области из задачи 1 домашнего задания.

Раздел 2. Цикл "пока" (цикл с условием)

Очень часто в алгоритмах приходится выполнять одну и ту же последовательность действий для варьируемых данных. В этом случае используется блок цикла, который обозначается в блок-схеме так:

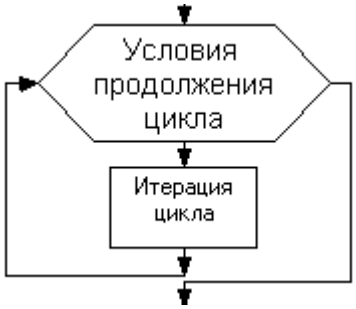


Рис.2.1. Блок цикла.

Цикл **"пока"** – одна из разновидностей блока цикла. Вход в цикл осуществляется, если условие принимает истинное значение. Далее выполняется тело цикла. После этого снова проверяется условие и так далее, **пока условие остается истинным**. Одно выполнение тела цикла называется итерацией цикла.

Примером использования цикла "пока" может являться ситуация ввода пароля, изображенная на Рис.2.2 – пока пользователь не введет правильный пароль, он будет запрашиваться снова.

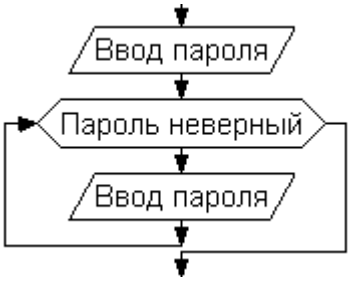


Рис.2.2. Фрагмент блок-схемы "Ввод и проверка пароля"

Задача 1. Найти сумму цифр числа.

Для решения задачи 1 требуется решить три подзадачи – получить последнюю цифру записи числа (вычислить остаток от деления числа на 10, например, 2345 % 10 = 5), перейти к числу, которое отличается от исходного отсутствием младшего разряда (выполнить целочисленное деление числа на 10, например, 3456/10 = 345), и определить, были ли просмотрены все разряды числа.

Алгоритм решения задачи 1 заключается в организации цикла, итерация которого выделяет последнюю цифру числа, добавляет ее к сумме и удаляет ее из числа. Как только разряды числа закончатся, т.е. число станет равным нулю (после учета последней цифры 3/10 = 0), цикл заканчивается.

Блок-схема решения задачи 1 приведена на Рис.2.3.

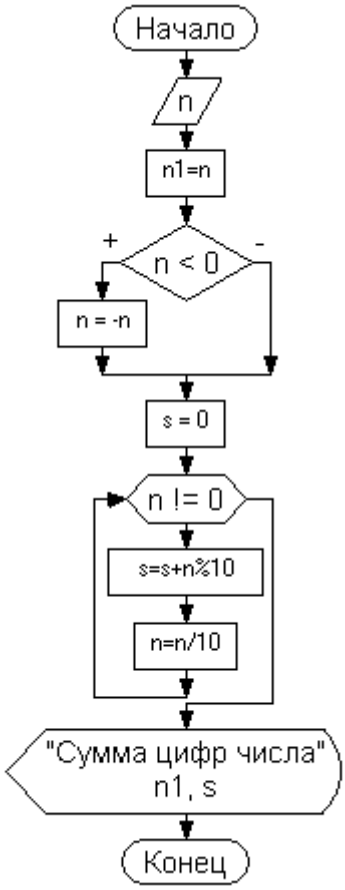


Рис.2.3. Блок-схема задачи о сумме цифр числа.

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
{
      int n,n1,s;
      printf("Введите n:");
      scanf("%d",&n);
      n1=n;
      // если число отрицательное, рассматриваем его модуль
      if(n<0) n=-n;
      s=0;
      while(n!=0)
      {
            s=s+n%10;
            n=n/10;
      }
      printf("Сумма цифр числа %d=%d\n",n1,s);
}
```

Задача 2. Найти значение числа π с заданной точностью ε из следующего соотношения:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \ldots + \frac{(-1)^{i-1}}{2i-1} + \ldots\right)$$

Для вычисления π нужно найти сумму бесконечной последовательности, которая будет накапливаться в переменной **pi**. Очередной член ряда заносится в переменную **drob**. Когда очередное слагаемое станет меньше значения ε, заданная точность будет достигнута и вычисления останавливаются.

Блок-схема решения задачи 2 приведена на Рис.2.4.

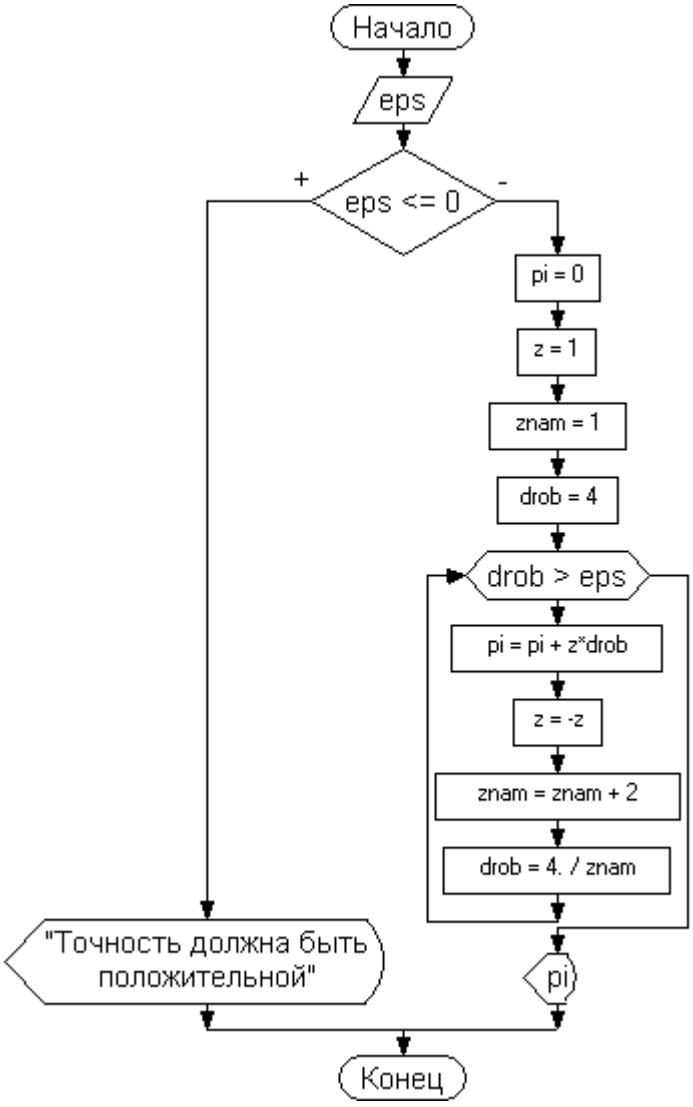


Рис.2.4. Блок-схема задачи о вычислении числа π.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
      float eps;
      float pi=0, drob;
      printf("Введите точность eps:");
      scanf("%f",&eps);
      if(eps<=0)
      {
            printf("Точность должна быть положительной\n");
            return;
      }
      int z=1;
      float znam=1;
      drob=4;
      while(drob>eps)
      {
            pi=pi+z*drob;
            z=-z;
            znam=znam+2;
            drob=4./znam;
      }
      printf("pi=%f\n", pi);
}
```

Задача 3. Найти НОД (наибольший общий делитель) двух неотрицательных чисел по алгоритму Евклида.

Алгоритм Евклида заключается в следующем. Пусть m и n – одновременно не равные нулю целые неотрицательные числа ($m \geq n$).

$$НОД(m,n) = \begin{cases} m, если\ n = 0 \\ НОД(n, ost), если\ n \neq 0, где\ ost = m\%n \end{cases}$$

Блок-схема решения задачи 3 приведена на Рис.2.5.

Код программы для задачи 3.

```
#include <stdio.h>
void main(void)
{
      int n, m,m1,n1;
      int t, ost, del;
      printf("Введите два числа:");
      scanf("%d%d",&m,&n);
      if(m<=0 || n<=0)
      {
            printf("Числа должны быть положительными\n");
            return;
      }
```

```
m1=m; n1=n;
```

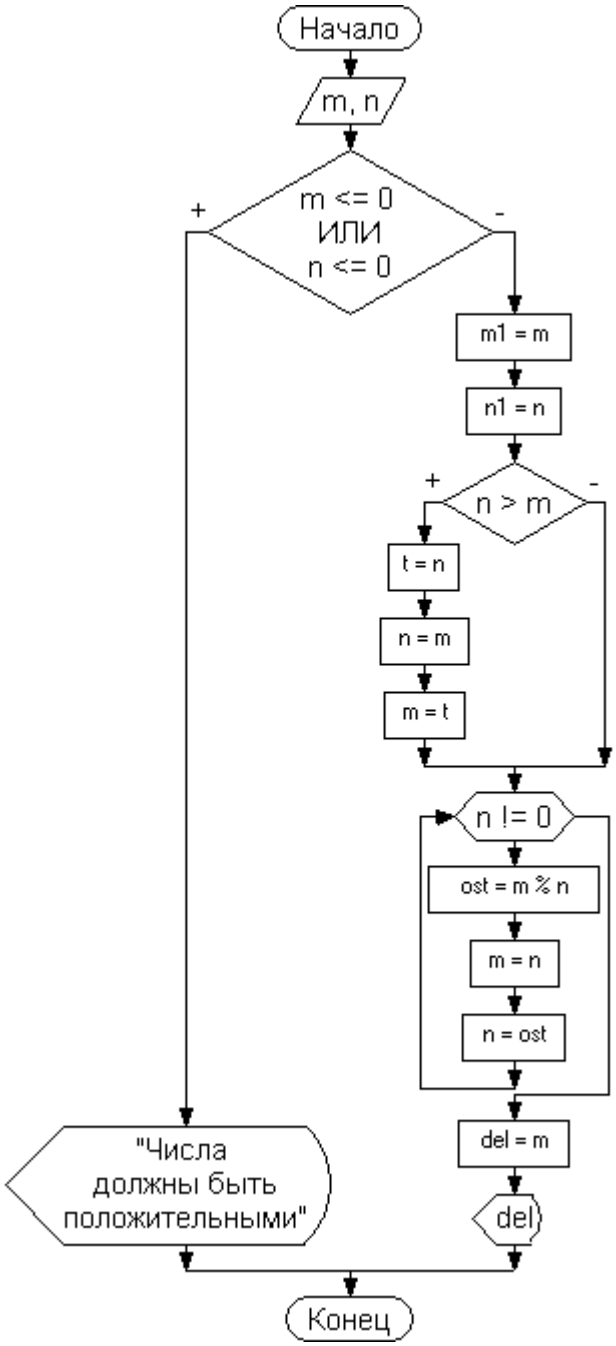


Рис.2.5. Блок-схема задачи о вычислении НОД двух чисел.

```
if(n>m)
{
      t=n;
      n=m;
      m=t;
}
while(n!=0)
{
      ost=m%n;
      m=n;
```

```
            n=ost;
        }
        del=m;
        printf("НОД двух чисел %d и %d=%d\n", m1,n1,del);
}
```

Задача 4. Найти минимальный элемент во входном потоке.

Сначала вводим количество элементов, которые будут извлекаться из входного потока. Далее используем прием, достаточно часто применяющийся для нахождения минимума или максимума. Пусть **E** – достаточно большое число (например, 100 000) такое, что каждый из элементов входного потока не превосходит его. Примем за текущий минимум (переменная **min**) значение **E**. Таким образом, первый же полученный элемент станет текущим минимумом. Далее, сравнивая последовательно все числа с текущим минимумом, меняем его при нахождении меньшего значения. В заключении распечатываем минимум.

Блок-схема решения задачи 4 приведена на Рис.2.6.

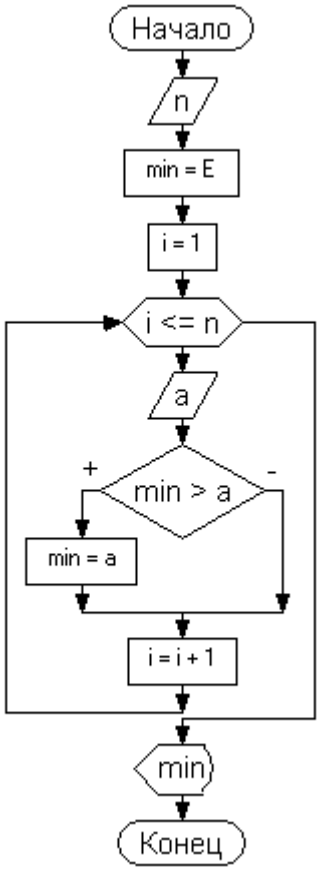


Рис.2.6. Блок-схема задачи о минимальном элементе
во входном потоке.

Код программы для задачи 4.

```
#include <stdio.h>
#define E 10000
void main(void)
{
     int n, a, min=E;
     printf("Введите количество чисел
                последовательности:");
     scanf("%d",&n);
     int i=1;
     while(i<=n)
     {
          scanf("%d", &a);
          if(min>a)
               min=a;
          i++;
     }
     printf("min=%d\n",min);
}
```

Домашнее задание

1. Найти методом деления отрезка пополам корни уравнения $f(x) = 0$ на отрезке $[a,b]$ с заданной точностью ε.

Полагаем, что отрезок задан так, что $f(a)f(b) < 0$, функция $f(x)$ непрерывна и унимодальна на $[a,b]$. В этом случае гарантируется, что корень на отрезке $[a,b]$ будет существовать (поскольку на границах отрезка функция принимает разные знаки, то она обязательно пересечет ось абсцисс, т.е. обязательно найдется x из $[a,b]$ такое, что $f(x) = 0$). Для примера можно рассмотреть функцию $f(x) = (x - 1)^2 - 5$, $[a,b] = [-1,5]$.

2. Дано число. Определить, входит ли заданная цифра в запись этого числа.

3. Из заданного числа сформировать новое, цифры в котором идут в обратном порядке.

4. Вычислить с заданной точностью ε сумму следующих рядов. Считается, что требуемая точность достигнута, если очередное слагаемое по модулю меньше ε.

а) $y = e^x - 1 = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^{n-1}}{(n-1)!} + \ldots$

б) $y = \cos x - 1 = -\frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + \frac{(-1)^{n-1} x^{2n-2}}{(2n-2)!} - \ldots$

в) $y = arctg\, x - x = -\frac{x^3}{3} + \frac{x^5}{5} - \ldots + \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)} - \ldots$

г) $y = \ln(1 - x) + x = -\frac{x^2}{2} - \frac{x^3}{3} - \ldots - \frac{(-1)^{n-1} x^n}{n} - \ldots$

Раздел 3. Цикл "для" (цикл с параметром)

Другим видом цикла является цикл **"для"**. Его удобно использовать в случаях, когда точно известно количество итераций цикла. Тогда создается переменная-счетчик, которая будет изменять свое значение при переходе от одной итерации к другой, пока не достигнет границы. Чаще всего эта переменная имеет определенный смысл: номер элемента массива, номер строки или столбца матрицы, степень числа и др.

В блоке цикла содержатся три части, которые разделяются запятыми. Первая часть служит для инициализации счетчика. Вторая задает граничное значение счетчика. Когда переменная-счетчик становится больше граничного значения, цикл завершается. Третья часть представляет собой приращение значения переменной-счетчика при переходе к следующей итерации цикла. Оно происходит в конце каждой итерации. Значение приращения называют также шагом цикла.

В качестве примера использования цикла "для" приведем фрагмент алгоритма решения следующей задачи – требуется вывести значения средних температур за год по месяцам. Фрагмент блок-схемы показан на Рис.3.1.

Рис.3.1. Фрагмент блок-схемы с циклом "для"

Задача 1. Найти

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Алгоритм не требует особенных комментариев. Значение переменной **n** должно быть неотрицательным. Цикл с параметром используется для перебора всех сомножителей искомого произведения (от 2 до **n**). Блок-схема решения этой задачи представлена на Рис.3.2.

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
```

```
{
      int n;
      printf("Enter n:");
      scanf("%d", &n);
      if (n<0)
      {
            printf("Введите неотрицательное n\n");
            return;
      }
      int p=1;
      for(int i=2;i<=n;i++)
            p=p*i;
      printf("%d!=%d\n",n,p);
}
```

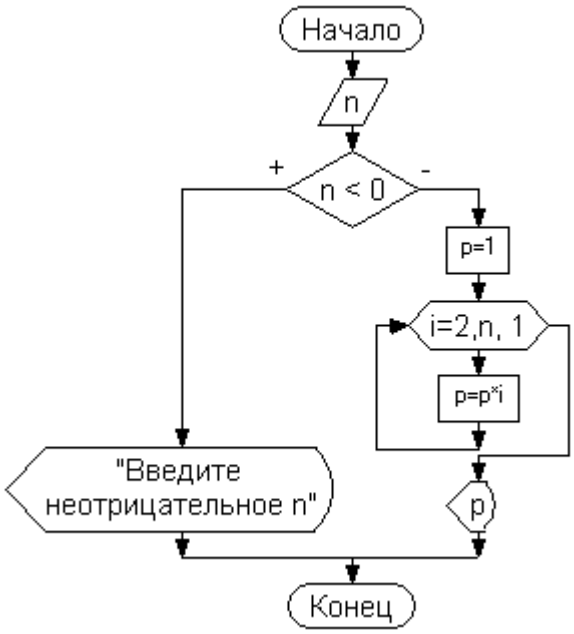


Рис.3.2. Блок-схема решения задачи о вычислении n!

Задача 2. Найти максимальное совершенное число, не превышающее заданного **M**. Число является совершенным, если оно равно сумме всех своих делителей, включая 1, и исключая само себя.

В данном случае приходится использовать один цикл внутри итерации другого. Такие конструкции алгоритма называются вложенными циклами.

При решении данной задачи во внешнем цикле просматриваются все натуральные числа, начиная с заданного **M** в обратном порядке. В этом случае шаг при переходе от итерации к итерации равен –1. Во внутреннем цикле определяется, является ли число совершенным. Для этого находятся все возможные делители числа, и вычисляется их сумма.

Блок-схема решения задачи 2 показана на Рис.3.3.

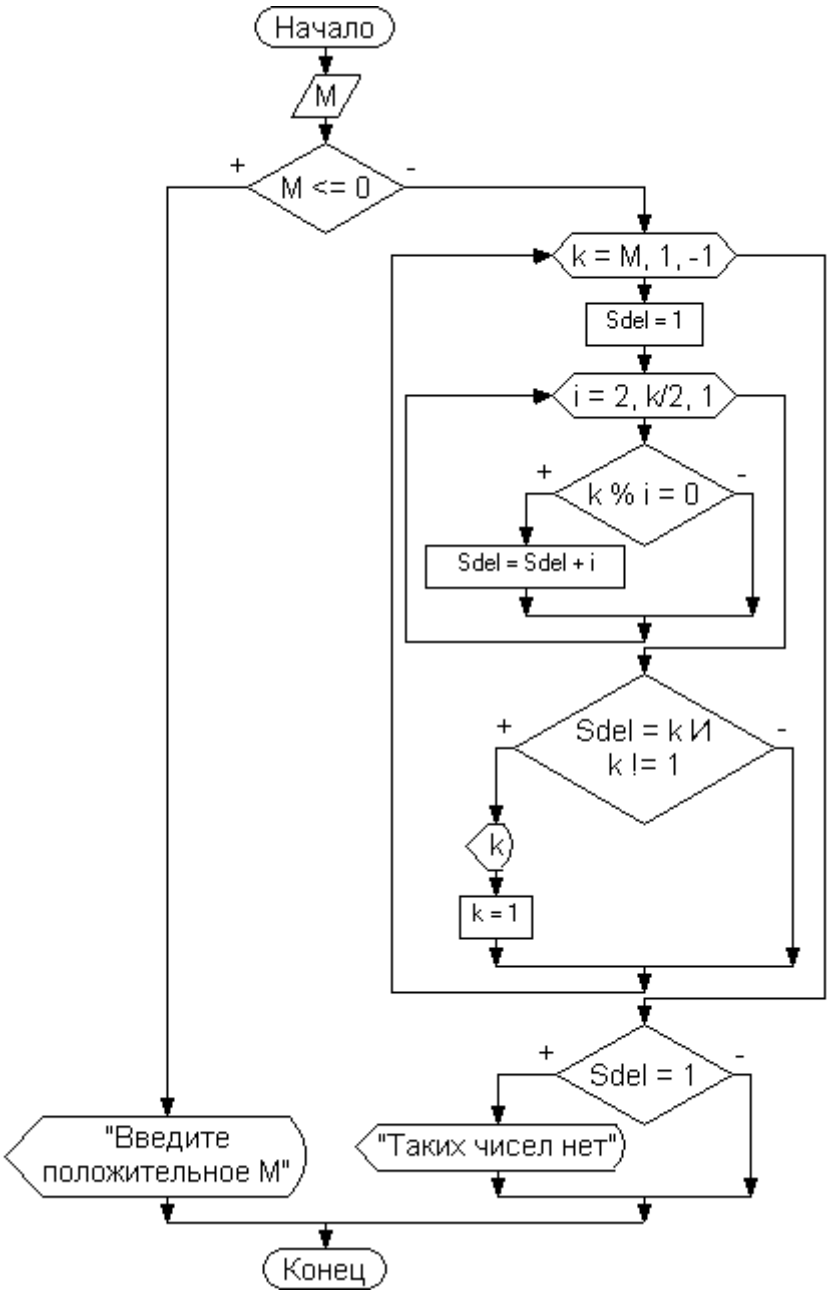


Рис.3.3. Блок-схема решения задачи о совершенном числе.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
      int M,
       Sdel; //переменная для хранения суммы делителей числа
      printf("Введите M:");
      scanf("%d", &M);
      if(M<=0)
      {
            printf("Введите положительное M\n");
            return;
      }
```

```
        for(int k=M; k>0; k--)
        {
              Sdel=1;
              for(int i=2; i<=k/2; i++)
                    if(k%i==0)
                          Sdel=Sdel+i;
              if(Sdel==k && k!=1)
              {
                    printf("Совершенное число=%d\n", k);
                    k=1;
              }
        }
        if (Sdel==1)
              printf("Таких чисел нет\n");
  }
```

Задача 3. Цилиндр объема 1 имеет радиус основания **r**. Определить высоту для значений r=0.5, 1, 1.5 ...5.

Алгоритм решения этой задачи очень прост. Нестандартным здесь является только шаг изменения переменной-счетчика – шаг равен 0.5. Поэтому тип переменной-счетчика надо задать как "число с плавающей точкой".

Блок-схема решения задачи 3 показана на Рис.3.4.

Код программы для задачи 3.

```
 # include <stdio.h>
 void main(void)
 {
       for(double r=0.5; r<=5; r=r+0.5)
       {
             double h=1./(3.14*r*r);
             printf("r=%f\th=%f\n",r,h);
       }
 }
```

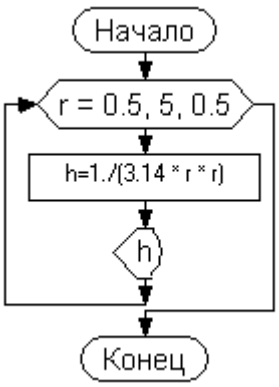


Рис.3.4. Блок-схема решения задачи о высоте цилиндра.

Задача 4. Распечатать все простые числа до 100.

Число 2 распечатываем без проверки, так как это единственное простое четное число. Все остальные нечетные числа проверяем на “простоту”. Для проверки простоты числа **k** требуется исследовать в качестве делителей все нечетные числа от 3 до **k**/2. Если найдется хотя бы один делитель, то число **k** не является простым. Факт нахождения такого делителя можно определить с помощью приема, называемого “использованием флажка”. Флажком является переменная, которая имеет всего два значения. Одно из них соответствует состоянию “истина”, другое – “ложь”. Значение переменной меняется при выполнении определенных условий (в данной задаче – при нахождении делителя числа **k**). Проверив все возможные делители, по значению переменной-флажка определяем, выполнилось ли требуемое условие.

Заметим, что для проверки на простоту числа **k** достаточно было проверить на делимость все нечетные числа до квадратного корня из **k**. Например, если **k** =15, то вовсе необязательно проверять в качестве делителя число 5. Поскольку **k**=5·3, то меньший делитель (3) будет найден на более ранних итерациях. Поэтому проверку следует осуществлять до квадратного корня из **k**, например, 25=5·5.

Блок-схема показана на Рис.3.5.

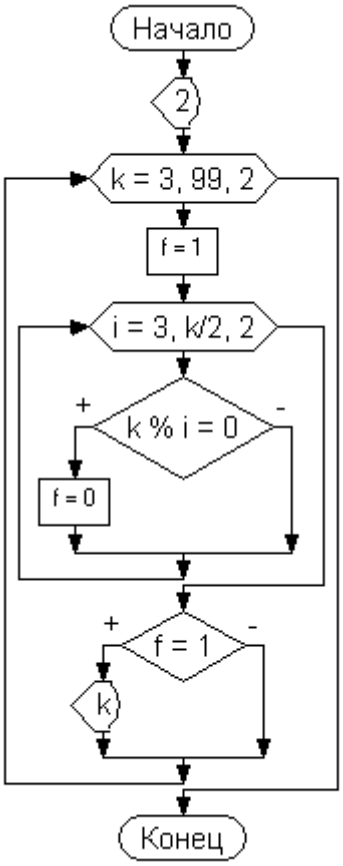


Рис.3.5. Блок-схема решения задачи о простых числах.

Код программы для задачи 4.

```
# include <stdio.h>
void main(void)
{
     printf("2 ");
     for(int k=3; k<100; k+=2)
     {
          int f=1;
          for(int i=3;i<=k/2;i+=2)
          {
             // если найдется делитель числа,
             //  т.е. оно не простое,
             // переменная-флажок f меняет значение
             if(k%i==0)
                 f=0;
          }
          if(f==1)
               printf("%d ",k);
     }
}
```

Задача 5. Распечатать все четырехзначные числа, в записи которых нет повторяющихся цифр.

Первая цифра числа может принимать значение от 1 до 9. Остальные – от 0 до 9. Поэтому в алгоритме создается четыре вложенных друг в друга цикла. Переменные-счетчики каждого цикла соответствуют цифре разряда числа. При получении цифры очередного разряда, проверяется, не встречалась ли она ранее. Когда все цифры искомого числа получены, формируем само число и печатаем его.

Блок-схема показана на Рис.3.6.

Код программы для задачи 5.

```
# include <stdio.h>
void main(void)
{
  for(int i=1; i<=9;i++)
    for(int j=0;j<=9; j++)
      if(i!=j)
        for(int k=0; k<=9; k++)
          if(k!=i && k!=j)
            for(int t=0;t<=9; t++)
              if(t!=i && t!=j && t!=k)
              {
                  int c=i*1000+j*100+k*10+t;
                  printf("%d ",c);
              }
}
```

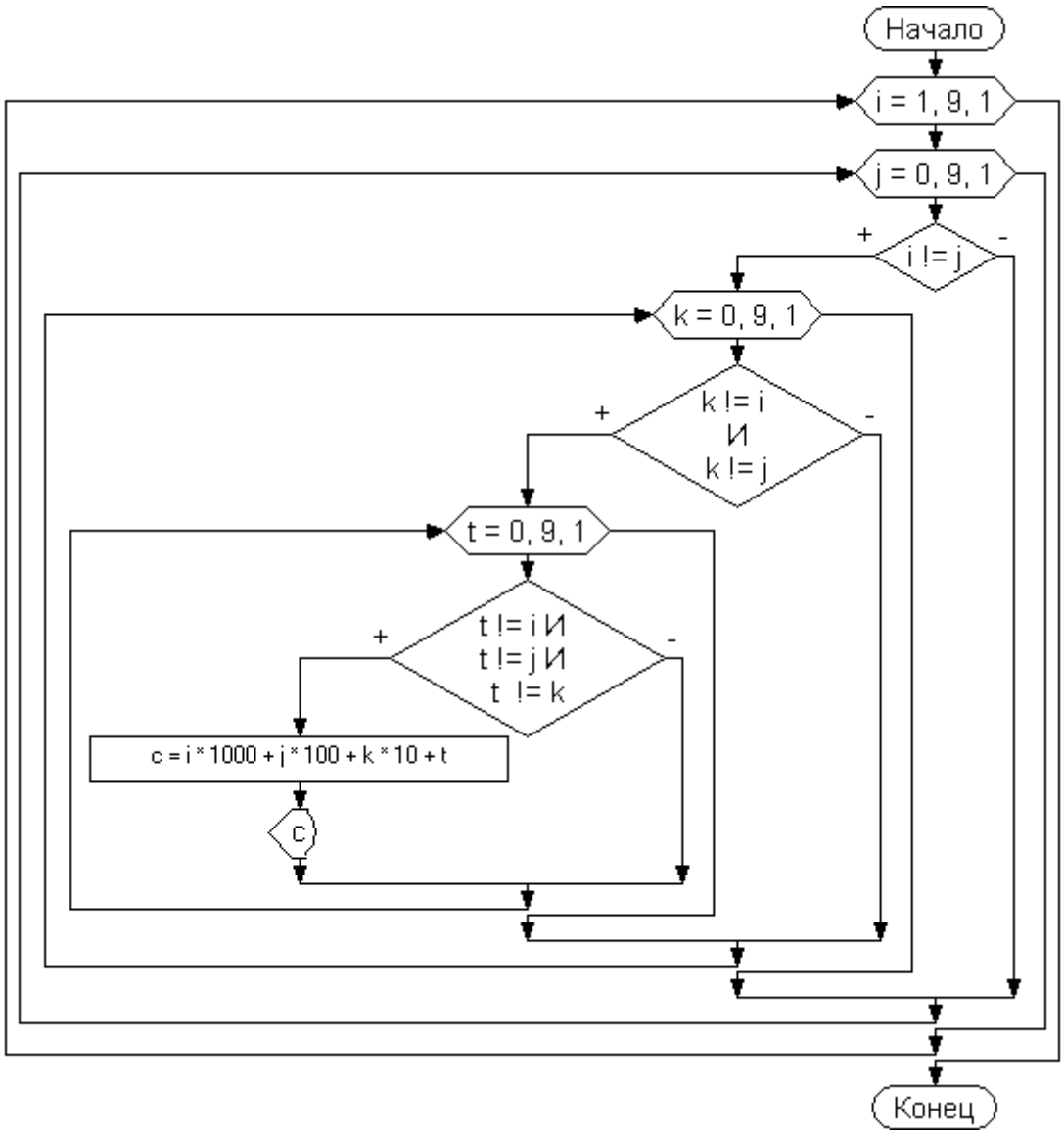


Рис.3.6. Блок-схема решения задачи о числах
с неповторяющимися цифрами.

Задача 6. Разложение числа на простые множители. Известно, что любое целое положительное число можно представить в виде произведения простых чисел. Например, 84=2·2·3·7.

Для решения этой задачи требуется проверить все возможные делители числа, начиная с 2. Пусть **i** – очередной делитель числа **k**. Далее ищем делители числа **k/i** до тех пор, пока делимое не станет равным 1. Каждый делитель может встречаться в произведении несколько раз. Поэтому, не изменяем делитель до тех пор, пока не получим число, которое на него не делится. Заметим, что не требуется проверять делители на "простоту", поскольку составные делители уже вошли в произведение в виде их разложения на простые множители.

Блок-схема показана на Рис.3.7.

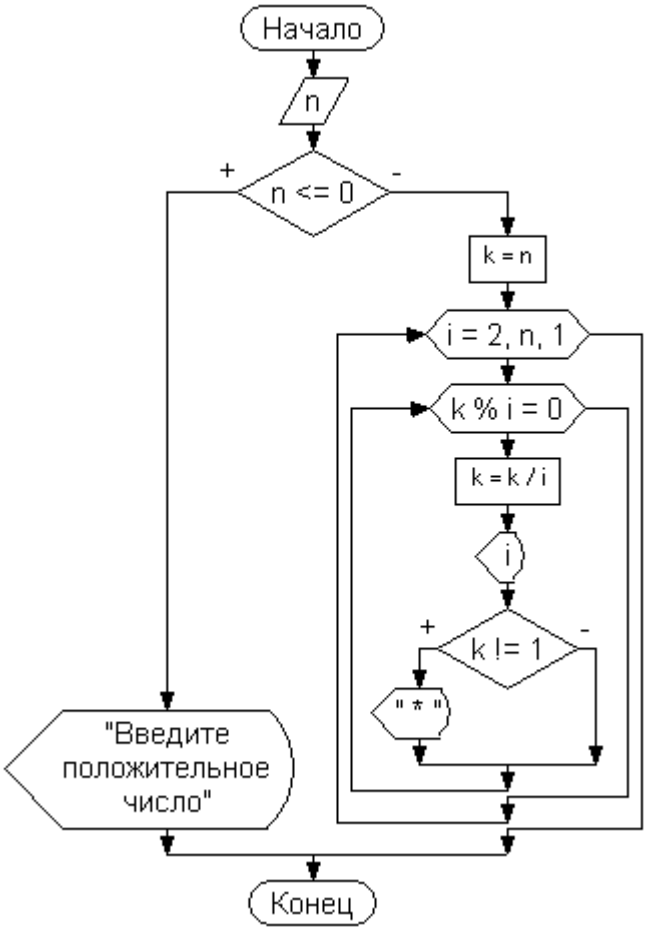


Рис.3.7. Блок-схема решения задачи о разложении числа на простые множители.

Код программы для задачи 6.

```
# include <stdio.h>
void main(void)
{
      int n;
      printf("Введите n:");
      scanf("%d", &n);
      if(n<=0)
      {
            printf("Введите положительное число\n");
            return;
      }
      int k=n;
      for(int i=2; i<=n; i++)
            while(k%i==0)
            {
                  k=k/i;
                  printf("%d",i);
                  if(k!=1) printf("*");
            }
}
```

Домашнее задание

1. Дано натуральное n. Вычислить

$$1 \cdot 2 + 2 \cdot 3 \cdot 4 + \ldots + n \cdot (n+1) \cdot 2n.$$

2. Найти значение $n!!$. Если n – четное, то

$$n!! = 2 \cdot 4 \cdot 6 \ldots n$$

Если n – нечетное, то

$$n!! = 1 \cdot 3 \cdot 5 \ldots n$$

3. Даны два положительных числа a, b. Найти число точек с целыми координатами, попадающих в эллипс

$$\left\{(x,y) : \frac{x^2}{a} + \frac{y^2}{b} \le 1\right\}.$$

.

4. Найти все числа Армстронга, принадлежащие заданному отрезку $[a,b]$. Числом Армстронга является число, сумма кубов цифр которого совпадает с самим числом. Например, $153 = 1^3 + 5^3 + 3^3$.

5. Найти все пары дружественных чисел, принадлежащих отрезку $[a,b]$. Два числа называются дружественными, если они равны сумме делителей друг друга (исключая сами числа). Например, дружественными числами являются числа 220 и 284.

6. Вычислить x^n.

7. Вычислить по схеме Горнера следующую функцию:

$$y = x^{10} + 2x^9 + 3x^8 + \ldots + 10x + 11.$$

Согласно схеме Горнера

$$y = x^{10} + 2x^9 + 3x^8 + \ldots + 10x + 11 = x(x^9 + 2x^8 + \ldots + 10) + 11$$

$$= x(x(x^8 + 2x^7 + \ldots + 9) + 10) + 11 \ldots$$

Глава 2. Основные структуры данных

Раздел 4. Массивы

Под структурой данных понимается организация хранения набора данных. Структура данных определяет также и алгоритмы доступа к ним.

Массивы являются одной из часто используемых структур данных. Их использование позволяет удобным способом размещать в памяти большое количество необходимой информации. Массив – это набор переменных одинакового типа, имеющих одно базовое имя. Обратиться к конкретной переменной массива можно по ее базовому имени и номеру (индексу). Например, к десятому элементу массива **a** обращаемся как к **a[10]**. В памяти компьютера переменные массива расположены друг за другом в одном непрерывном блоке.

С элементами массива работают как с обычными переменными. Поэтому, когда требуется работать с массивом в целом, для перебора элементов используется цикл. Например, ввод и вывод массива происходит по следующей схеме:

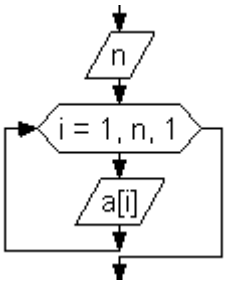


Рис.4.1. Фрагмент блок-схемы ввода массива

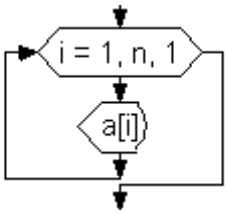


Рис.4.2. Фрагмент блок-схемы вывода массива

В дальнейшем для краткости будем обозначать каждый из этих блоков так:

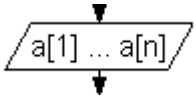


Рис.4.3. Блок ввода массива

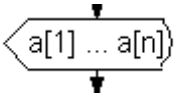


Рис.4.4. Блок вывода массива

Задача 1. Найти сумму отрицательных элементов массива.

Кратко алгоритм решения задачи таков. Просматриваем все элементы массива. Если очередной элемент отрицательный, добавляем его значение к сумме. После завершения цикла определяем, были ли отрицательные элементы в массиве, по значению переменной, хранящей сумму. Если таких элементов не было, сумма остается равной 0.

Блок-схема решения задачи представлена на Рис.4.5.

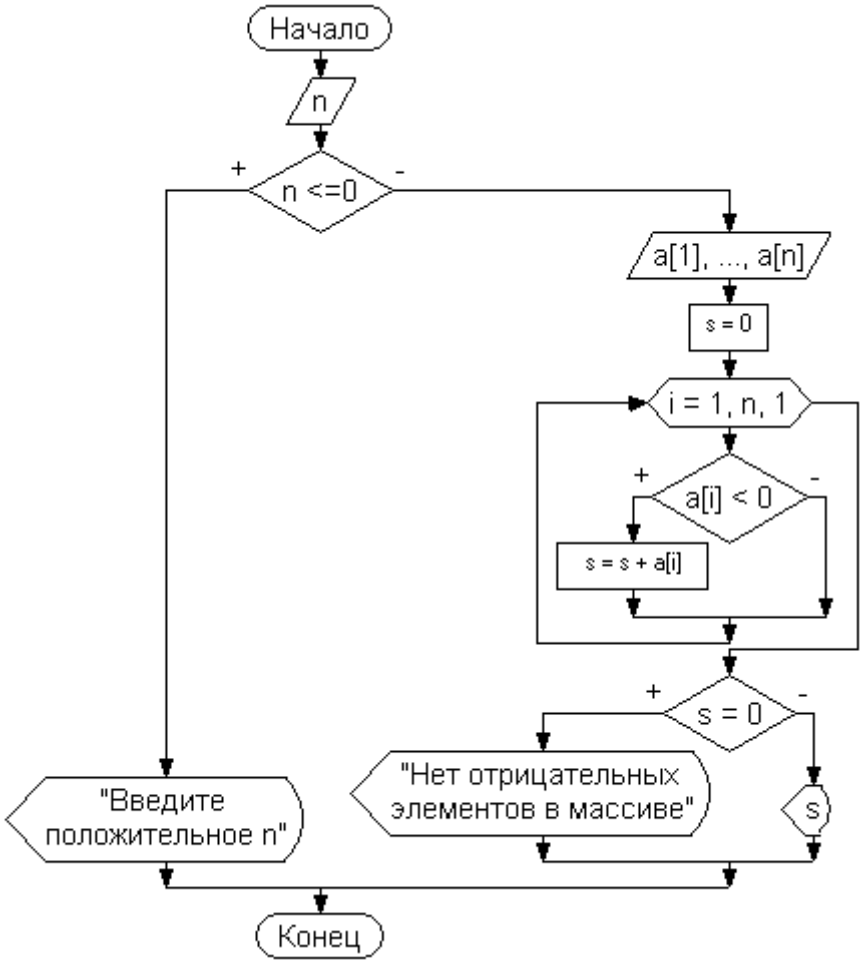


Рис.4.5. Блок-схема решения задачи о сумме отрицательных чисел массива

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
{
     int n;
     int * a;
     printf("Введите количество элементов массива n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительное n\n");
```

```
            return;
        }
        // выделение памяти для массива размера n
        a=new int[n];
        if(a==NULL)
        {
            printf("Запрошено большое количество памяти.\n
                Попробуйте ввести меньшее количество.\n");
            return;
        }
        // нумерация элементов массива в языке C++
        // начинается с 0
        printf("Введите элементы массива:\n");
        for(int i=0;i<n; i++)
            scanf("%d", &a[i]);
        // переменная для хранения суммы
        // отрицательных элементов
        int s=0;
        for(i=0;i<n; i++)
            if(a[i]<0)
                s=s+a[i];
        if(s==0)
            printf("Нет отрицательных элементов в массиве\n");
        else
            printf("s=%d\n",s);
        // освобождение памяти, занимаемой массивом
        delete [] a;
}
```

Задача 2. Найти максимальный элемент массива.

Предположим, что первый элемент массива может быть максимальным. Поэтому запоминаем его в переменную **max**, т.е. принимаем это значение за текущий максимум. Далее просматриваем все элементы массива, начиная со второго, и сравниваем каждый со значением переменной **max** (с текущим максимумом). Если очередной элемент больше текущего максимума, то запоминаем его в качестве максимального. Просмотрев весь массив, в переменной **max** получим максимальное значение.

Блок-схема решения задачи представлена на Рис.4.6.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
    int n;
    int * a;
    printf("Введите количество элементов массива n:");
    scanf("%d",&n);
    if(n<=0)
    {
```

```
            printf("Введите положительное n\n");
            return;
      }
      a=new int[n];
      if(a==NULL)
      {
            printf("Запрошено большое количество памяти.\n
                  Попробуйте ввести меньшее количество.\n");
            return;
      }
      printf("Введите элементы массива:\n");
      for(int i=0;i<n; i++)
            scanf("%d", &a[i]);
      int max=a[0];
      for(i=1; i<n; i++)
            if(a[i]>max)
                  max=a[i];
      printf("max=%d\n",max);
      delete [] a;
}
```

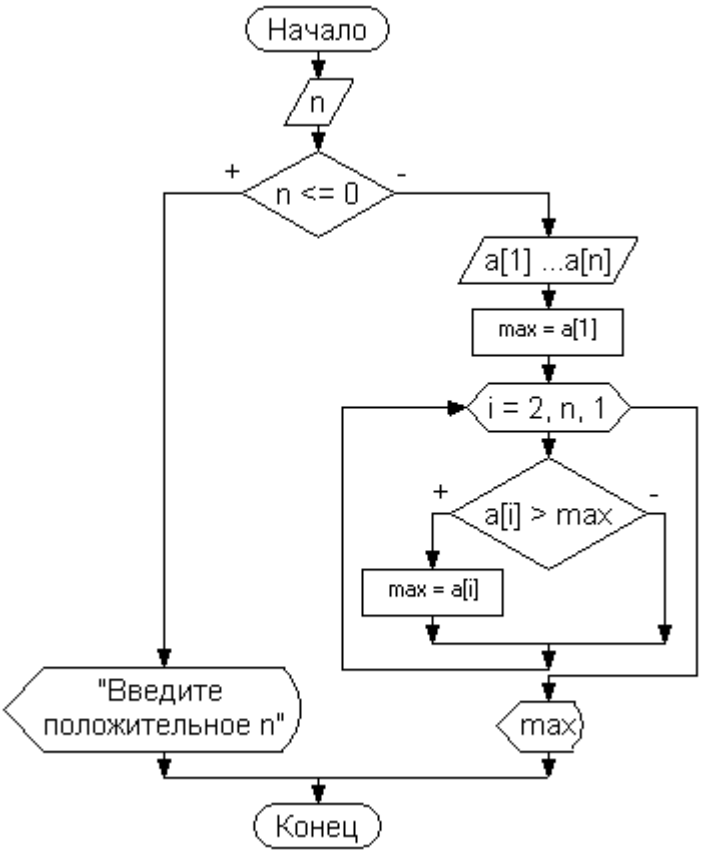


Рис.4.6. Блок-схема решения задачи о максимальном элементе массива

Задача 3. Определить количество минимальных элементов массива.

Рассмотрим два алгоритма решения этой задачи. Первый алгоритм (более понятный) заключается в разбиении исходной задачи на две подзадачи: нахождение минимального элемента и подсчет количества равных ему элементов. Заметим, что при этом приходится дважды просмотреть весь массив.

Второй алгоритм (более эффективный) решает обе подзадачи за один просмотр массива. Так же, как в предыдущей задаче, предполагаем, что первый элемент может являться минимальным. Соответственно, количество минимальных элементов в начальный момент равно 1. Встретив элемент, совпадающий с текущим минимумом, увеличиваем их количество на 1. Если же найден элемент, меньший текущего минимума, то меняем минимум на новый и устанавливаем количество в 1.

Блок-схемы, описывающие эти алгоритмы, представлены на Рис.4.7 и Рис.4.8.

Код программы для задачи 3 (алгоритм 1).

```
# include <stdio.h>
void main(void)
{
     int n;
     int * a;
     printf("Введите количество элементов массива n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительное n\n");
          return;
     }
     a=new int[n];
     if(a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньшее количество.\n");
          return;
     }
     printf("Введите элементы массива:\n");
     for(int i=0;i<n; i++)
          scanf("%d", &a[i]);
     int min=a[0];
     for(i=1; i<n; i++)
          if(min>a[i])
               min=a[i];
     int kol=0;
     for(i=0;i<n; i++)
          if(a[i]==min)
```

```
                kol++;
        printf("Количество=%d\n",kol);
        delete []a;
}
```

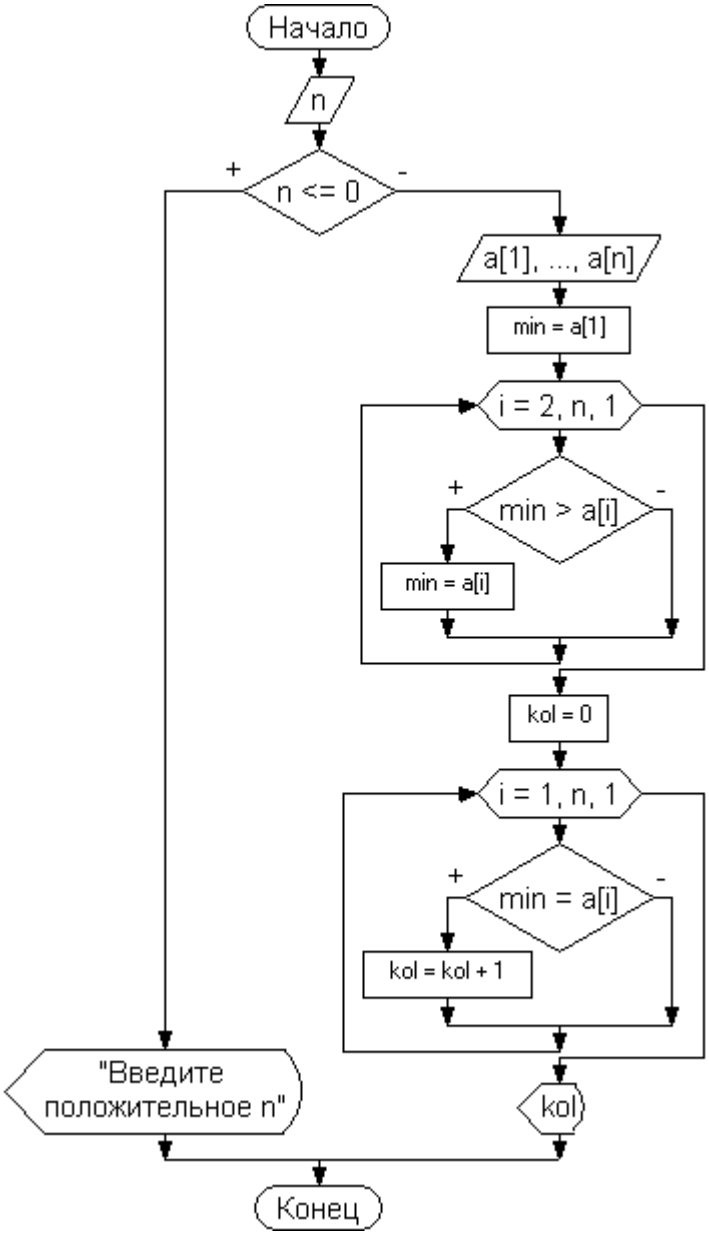


Рис.4.7. Блок-схема решения задачи о количестве минимальных элементов массива (алгоритм 1).

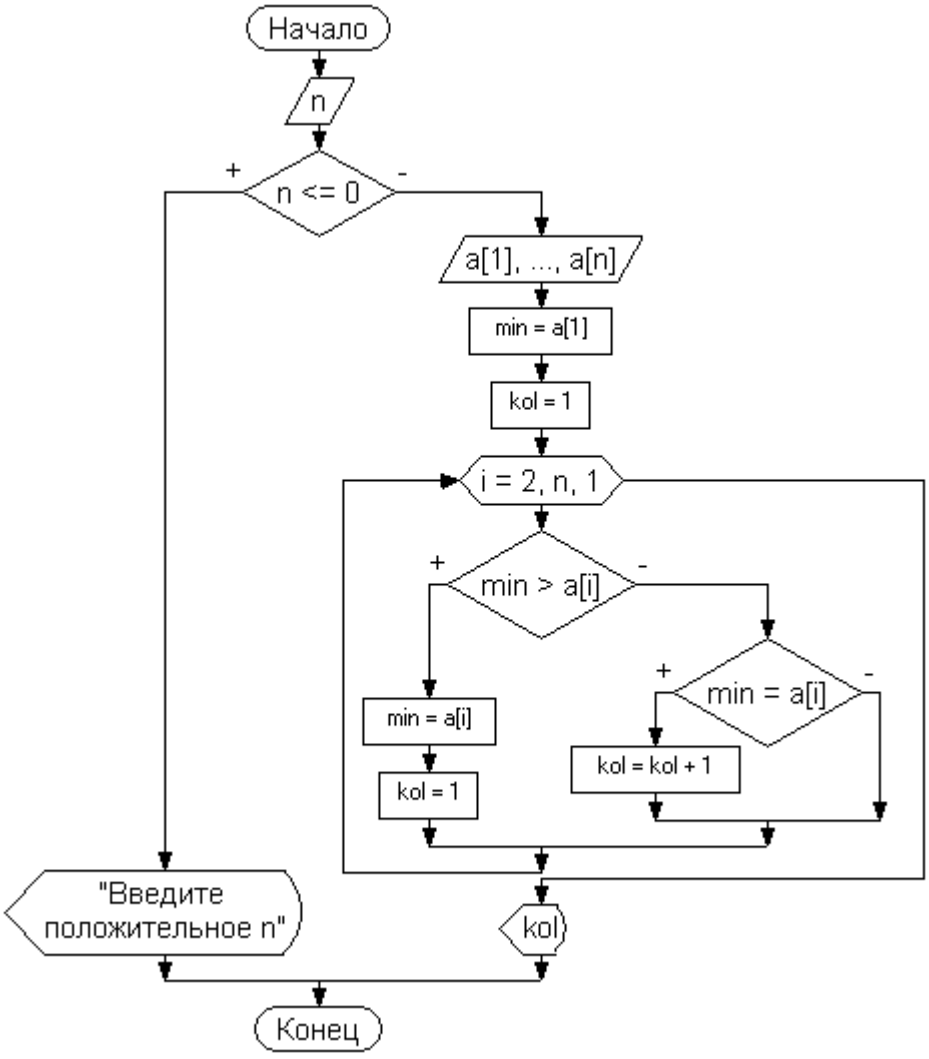


Рис.4.8. Блок-схема решения задачи о количестве минимальных элементов массива (алгоритм 2).

Код программы для задачи 3 (алгоритм 2).

```
# include <stdio.h>
void main(void)
{
     int n;
     int * a;
     printf("Введите количество элементов массива n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительное n\n");
          return;
     }
     a=new int[n];
     if(a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньшее количество.\n");
```

```
            return;
        }
        printf("Введите элементы массива:\n");
        for(int i=0;i<n; i++)
            scanf("%d", &a[i]);
        int min=a[0];
        int kol=1;
        for(i=1; i<n; i++)
            if(min>a[i])
            {
                min=a[i];
                kol=1;
            }
            else if(min==a[i]) kol++;
        printf("Количество=%d\n",kol);
        delete []a;
}
```

Задача 4. Поменять в массиве местами минимальный и максимальный элементы.

Отличие этой задачи от двух предыдущих заключается в том, что для перестановки местами двух элементов массива надо запомнить дополнительно их порядковые номера. Для этого вводятся две переменные **kmin** и **kmax**. Их значения меняются одновременно с изменением текущих минимума и максимума (значений переменных **min** и **max**).

Блок-схема решения задачи 4 показана на Рис.4.9.

Код программы для задачи 4.

```
# include <stdio.h>
void main(void)
{
        int n;
        int *a;
        printf("Введите количество элементов массива n:");
        scanf("%d",&n);
        if(n<=0)
        {
            printf("Введите положительное n\n");
            return;
        }
        a=new int[n];
        if(a==NULL)
        {
            printf("Запрошено большое количество памяти.\n
                Попробуйте ввести меньшее количество.\n");
            return;
        }
```

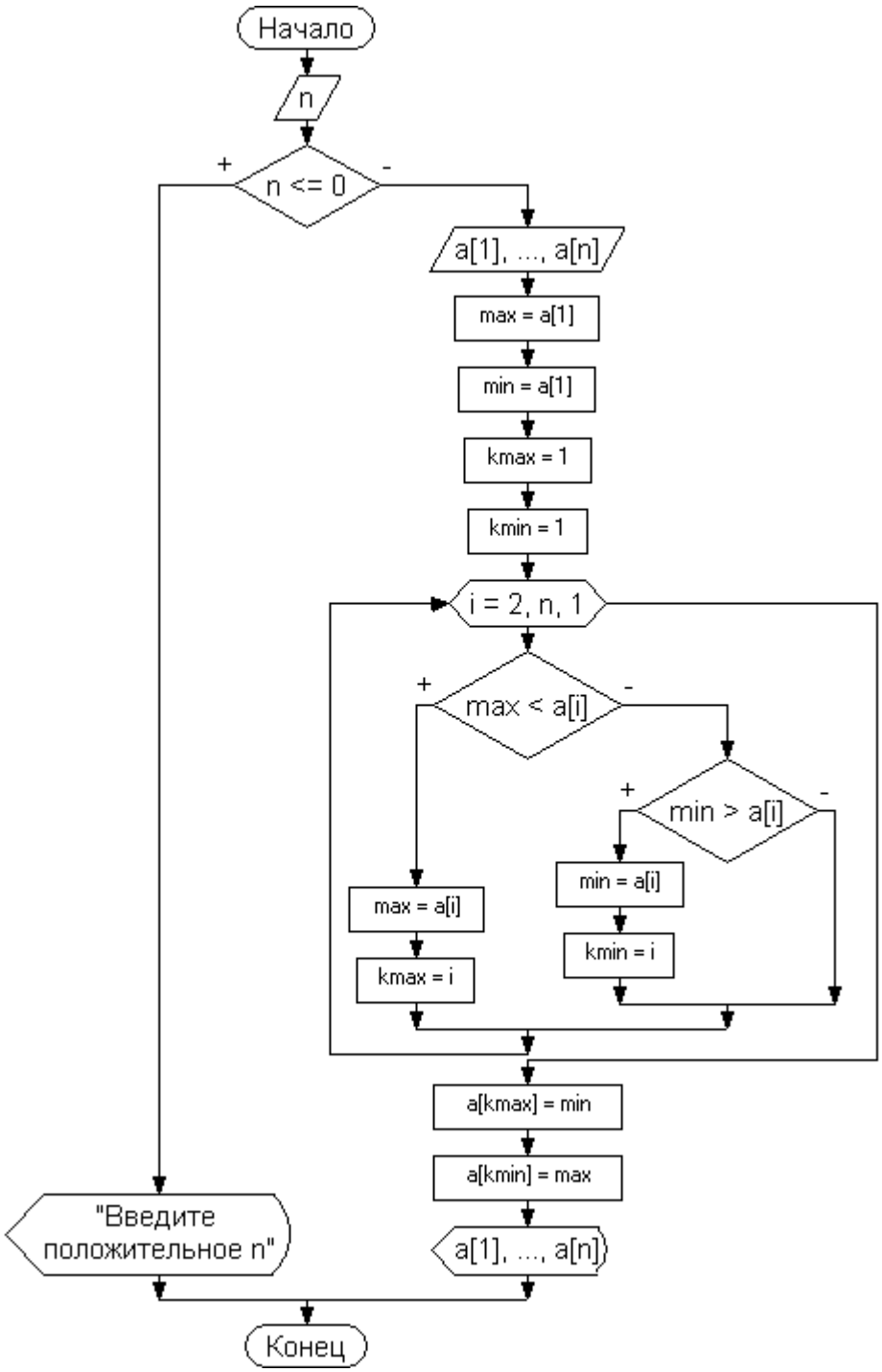


Рис.4.9. Блок-схема решения задачи о перестановке максимального и минимального элементов массива.

```
printf("Введите элементы массива:\n");
for(int i=0;i<n; i++)
     scanf("%d", &a[i]);
int max=a[0], min=a[0];
int kmax=0, kmin=0;
for(i=1; i<n; i++)
     if(max<a[i])
     {
          max=a[i];
```

```
                kmax=i;
            }
            else if(min>a[i])
                {
                    min=a[i];
                    kmin=i;
                }
        a[kmax]=min;
        a[kmin]=max;
        printf("Массив после перестановки элементов\n");
        for(i=0; i<n; i++)
            printf("%d ",a[i]);
        printf("\n");
        delete [] a;
}
```

Задача 5. Дан массив. Отсортировать элементы массива по возрастанию. Существует множество алгоритмов сортировки массивов. Ряд из них приведен в книге Кнута "Искусство программирования", том 3 "Сортировка и поиск". Рассмотрим два таких алгоритма.

5.1. Метод линейного выбора.

Сначала требуется найти минимальный элемент массива. Его нужно поставить на первое место. Поэтому меняем местами первый и минимальный элементы массива. После выполнения этой процедуры первый элемент массива находится на своем месте. Далее следует повторить эти же действия, исключив из рассмотрения первый элемент. В оставшемся массиве ищется минимальный элемент и устанавливается на второе место и т.д. В итоге получим отсортированный массив.

Блок-схема решения задачи 5 методом линейного выбора представлена на Рис.4.10.

Код программы для задачи 5 (алгоритм линейного выбора).

```
# include <stdio.h>
void main(void)
{
    int n;
    int *a;
    printf("Введите количество элементов массива n:");
    scanf("%d",&n);
    if(n<=0)
    {
        printf("Введите положительное n\n");
        return;
    }
```

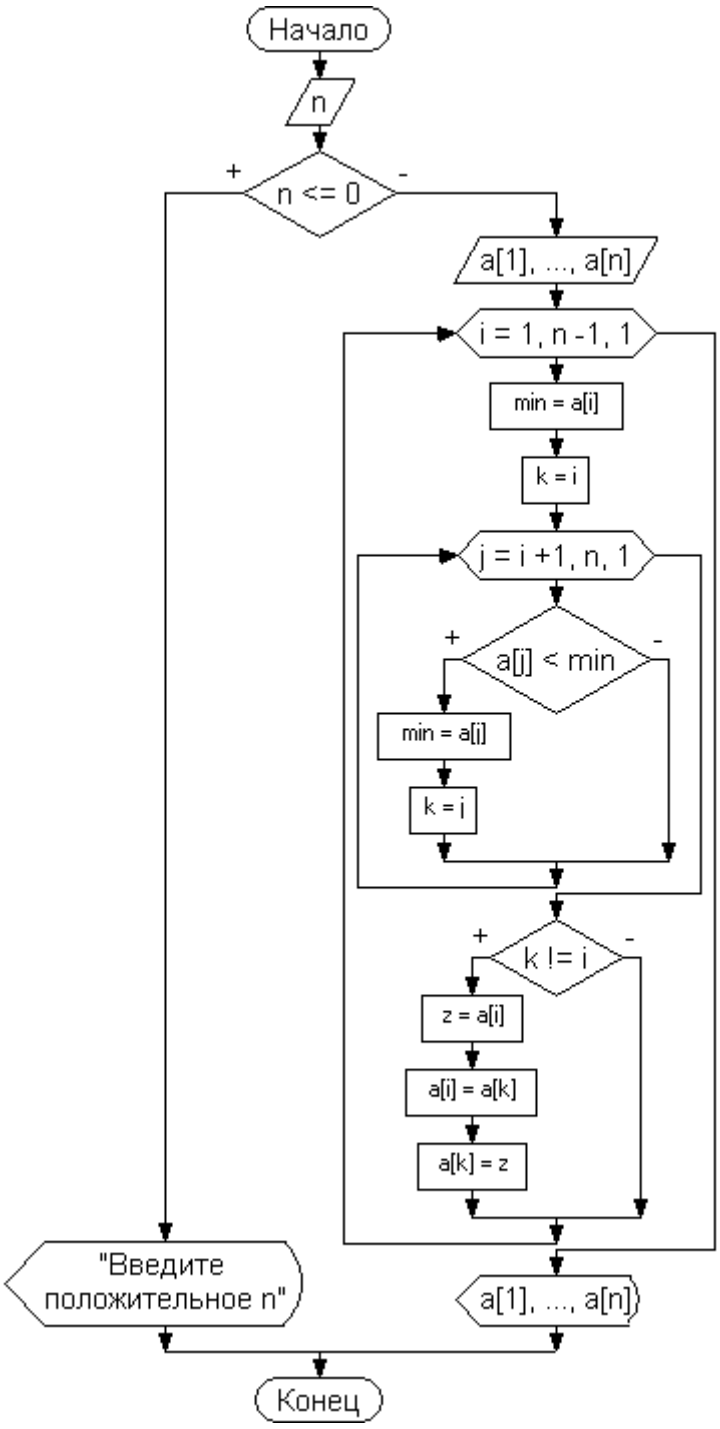


Рис.4.10. Блок-схема решения задачи сортировки массива методом линейного выбора.

```
a=new int[n];
if(a==NULL)
{
     printf("Запрошено большое количество памяти.\n
          Попробуйте ввести меньшее количество.\n");
     return;
}
printf("Введите элементы массива:\n");
for(int i=0;i<n; i++)
```

```
            scanf("%d", &a[i]);
      for(i=0;i<n-1;i++)
      {
            int min=a[i];
            int k=i;
            for(int j=i+1; j<n; j++)
                  if(a[j]<min)
                  {
                        min=a[j]; k=j;
                  }
            if(k!=i)
            {
                  int z=a[i]; a[i]=a[k]; a[k]=z;
            }
      }
      printf("Отсортированный массив:\n");
      for(i=0; i<n; i++)
            printf("%d ",a[i]);
      printf("\n");
      delete [] a;
}
```

5.2. Метод "пузырька".

При сортировке массива методом пузырька сравниваются последовательно пары соседних элементов массива. Если упорядоченность элементов нарушена, меняем их местами. В результате за один проход по массиву максимальный элемент встает на последнее место (всплывает как воздушный пузырек в воде). Поэтому дальше можно рассматривать массив, количество элементов в котором на 1 меньше (исключаем из рассмотрения последний элемент). В худшем случае будет $n - 1$ таких проходов по массиву. Если порядок элементов в сравниваемых парах не был нарушен ни разу, массив является упорядоченным. Для проверки этого создается переменная-флажок с именем **f**, принимающая значение 1 в случае, если массив не упорядочен по возрастанию, и 0 – в противном случае.

Блок-схема решения задачи 5 методом пузырька показана на Рис.4.11.

Код программы для задачи 5 (метод "пузырька").

```
# include <stdio.h>
void main(void)
{
      int n;
      int *a;
      printf("Введите количество элементов массива n:");
      scanf("%d",&n);
```

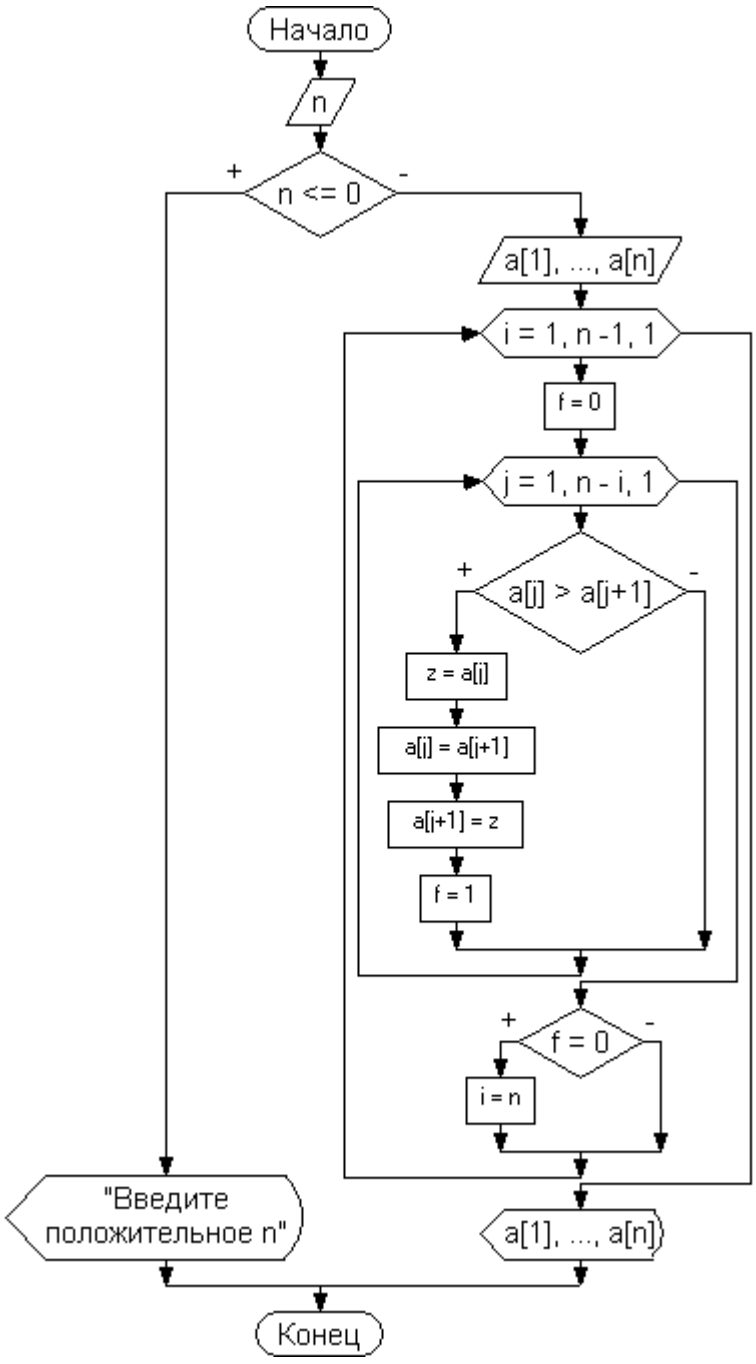


Рис.4.11. Блок-схема решения задачи сортировки массива методом "пузырька".

```
if(n<=0)
{
     printf("Введите положительное n\n");
     return;
}
a=new int[n];
if(a==NULL)
{
     printf("Запрошено большое количество памяти.\n
          Попробуйте ввести меньшее количество.\n");
```

```
            return;
        }
        printf("Введите элементы массива:\n");
        for(int i=0;i<n; i++)
            scanf("%d", &a[i]);
        for(i=0;i<n-1;i++)
        {
            int f=0;
            for(int j=0; j<n-i-1;j++)
                if(a[j]>a[j+1])
                {
                    int z=a[j]; a[j]=a[j+1]; a[j+1]=z; f=1;
                }
            if(f==0)
                break;
        }
        printf("Отсортированный массив:\n");
        for(i=0; i<n; i++)
            printf("%d ",a[i]);
        printf("\n");
        delete [] a;
}
```

Задача 6. Осуществить слияние двух отсортированных по возрастанию массивов.

Поскольку оба исходных массива упорядочены, минимальные элементы в них стоят на первом месте. Выбираем из них наименьший и заносим его во вновь создаваемый массив. Далее следует исключить этот элемент из рассмотрения посредством перехода к следующему элементу того массива, откуда был взят минимальный элемент. Повторяем эти действия со следующей парой элементов (с наименьшими из нерассмотренных элементов исходных массивов) и так до тех пор, пока не закончится один из массивов. Остатки второго массива просто переносим в конец нового массива.

Блок-схема решения задачи 6 показана на Рис.4.12.

Код программы для задачи 6.

```
# include <stdio.h>
void main(void)
{
    int n,m;
    int *a, *b, *c;
    printf("Введите количество элементов
              первого массива:");
    scanf("%d",&n);
    printf("Введите количество элементов
              второго массива:");
    scanf("%d",&m);
```

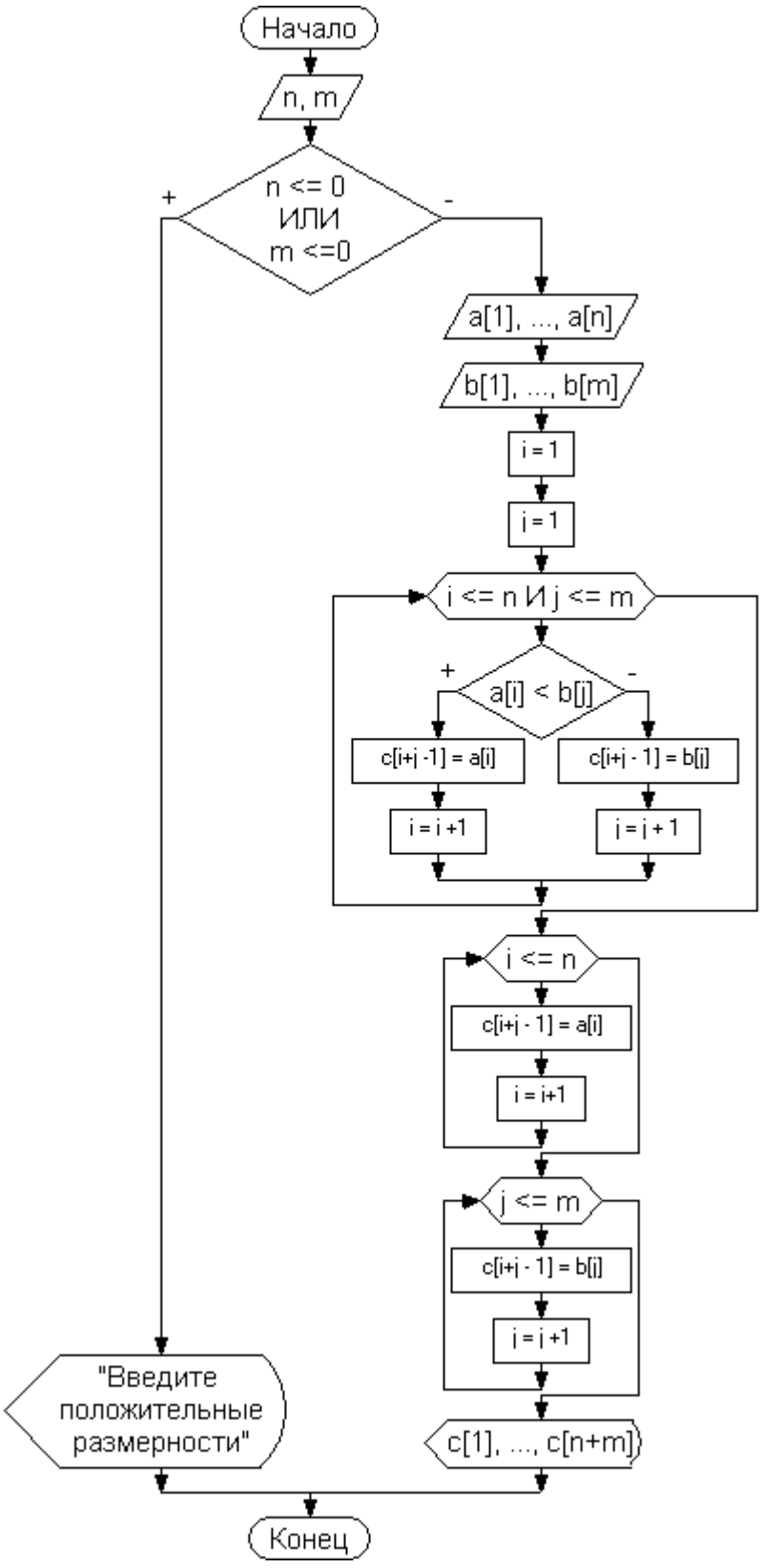


Рис.4.12. Блок-схема решения задачи слияния двух отсортированных массивов

```
if(n<=0 || m<=0)
```

```
        {
             printf("Введите положительные размеры\n");
             return;
        }
        a=new int[n];
        b=new int[m];
        c=new int[n+m];
        if(a==NULL || b==NULL || c==NULL)
        {
             printf("Запрошено большое количество памяти.\n
                  Попробуйте ввести меньшее количество.\n");
             return;
        }
        printf("Введите элементы первого массива:\n");
        for(int i=0;i<n; i++)
             scanf("%d", &a[i]);
        printf("Введите элементы второго массива:\n");
        for(i=0;i<m; i++)
             scanf("%d", &b[i]);
        int k;
        i=0; k=0;
        while(i<n && k<m)
             if(a[i]<b[k])
             {
                  c[i+k]=a[i];
                  i++;
             }
             else
             {
                  c[i+k]=b[k];
                  k=k+1;
             }
        for(;i<n; i++)
             c[i+k]=a[i];
        for(;k<m; k++)
             c[i+k]=b[k];
        printf("Объединенный массив:\n");
        for(i=0;i<n+m; i++)
             printf("%d ",c[i]);
        printf("\n");
        delete [] a;
        delete [] b;
        delete [] c;
}
```

Задача 7. Поиск элемента в отсортированном массиве методом деления пополам (алгоритм бинарного поиска).

В данном алгоритме существенную роль играет факт упорядоченности массива. Итерация алгоритма поиска заключается в следующем.

Выбираем элемент, расположенный в середине массива. Если он

совпадает с искомым, то поиск прекращается. Если же этот элемент больше искомого, то в силу упорядоченности массива продолжать поиск следует в левой его половине, иначе – в правой половине массива.

Алгоритм бинарного поиска является эффективным, поскольку после каждой итерации количество рассматриваемых элементов уменьшается вдвое. Для определения границ сегмента поиска элемента создаются две переменные: **l** – левая граница сегмента поиска, **r** – правая граница сегмента поиска. На каждой из итераций цикла поиска одна из переменных **l** или **r** меняет свое значение. Поиск заканчивается тогда, когда левая и правая границы совпадут, т.е. в сегменте останется только один элемент. Если он равен искомому, выводим его индекс, иначе делаем вывод о том, что такого элемента в массиве нет.

Блок-схема решения задачи 7 показана на Рис.4.13.

Код программы для задачи 7.

```
# include <stdio.h>
void main(void)
{
     int n;
     int *a;
     printf("Введите количество элементов массива n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительное n\n");
          return;
     }
     a=new int[n];
     if(a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньшее количество.\n");
          return;
     }
     printf("Введите элементы массива:\n");
     for(int i=0;i<n; i++)
          scanf("%d", &a[i]);
     int k,l,r;
     printf("Введите искомый элемент:");
     scanf("%d",&k);
```

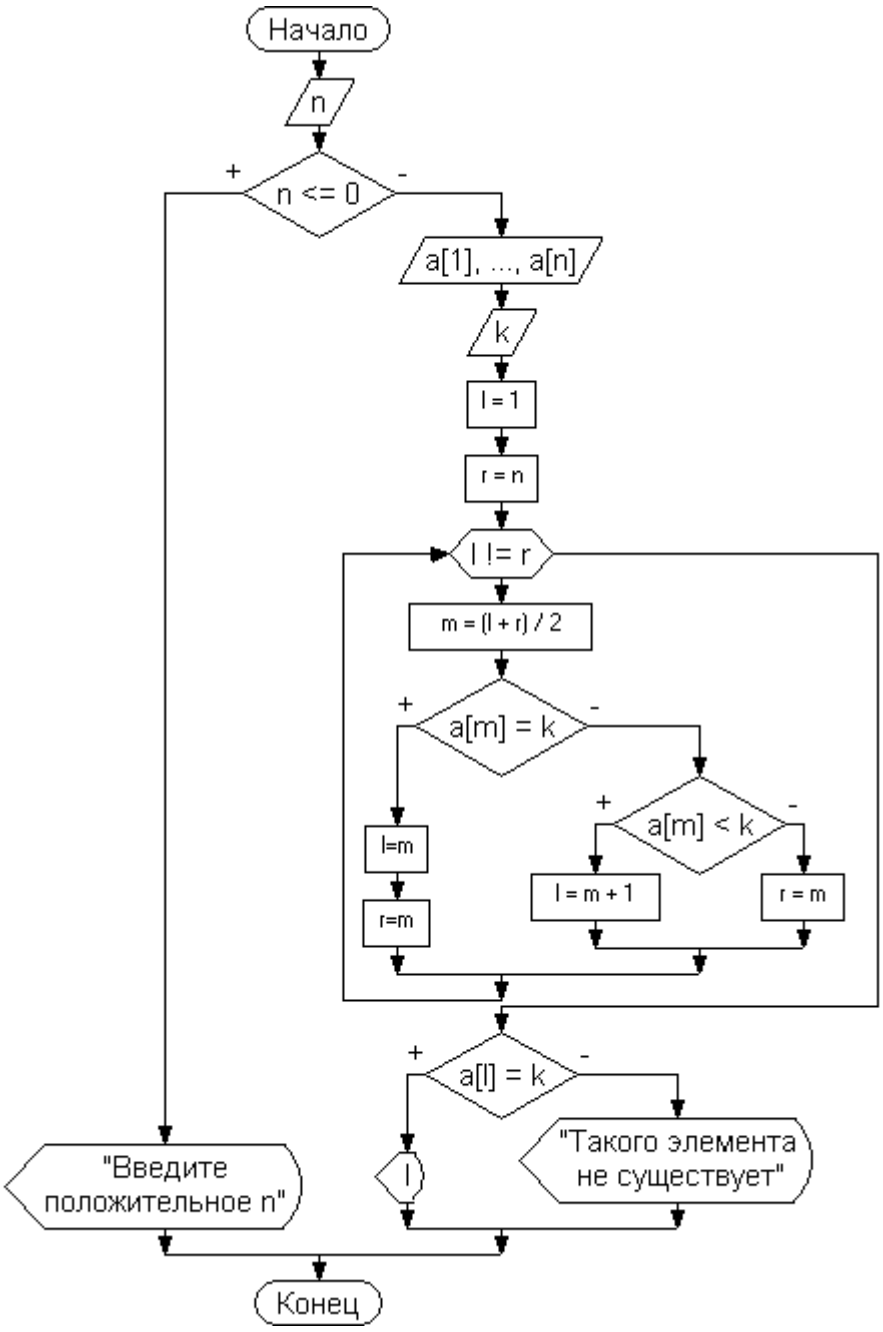


Рис.4.13. Блок-схема решения задачи поиска заданного элемента в отсортированном массиве.

```
l=0; r=n-1;
while(l!=r)
{
      int m=(l+r)/2;
      if(a[m]==k)
      {
            l=m;r=m;
      }
      else
            if(a[m]<k)
                  l=m+1;
```

```
                else
                     r=m;
        }
        if(a[l]==k)
             printf("a[%d]=%d\n",l+1,k);
        else
             printf("Такого элемента не существует\n");
        delete []a;
}
```

Задача 8. Представить заданное целое число в двоичной системе счисления.

Для хранения двоичного представления целого числа будем использовать массив. Алгоритм решения в данном случае можно разбить на три подзадачи – определение количества разрядов числа, нахождение цифр двоичного представления числа и занесение их в массив, и их перестановка, так как двоичные разряды при их вычислении следуют в обратном порядке.

Таким образом, блок-схема решения задачи 8 будет такой (Рис.4.14).

Код программы для задачи 8.

```
# include <stdio.h>
void main(void)
{
     int k,n;
     printf("Введите число k:");
     scanf("%d",&k);
     char z=' ';          // знак числа
     n=k;
     if(k<0)
     {
          k=-k;
          z='-';
     }
     int i=0,        // количество двоичных разрядов
          x=k;
     while(x>0)
     {
          x=x/2;
          i++;
     }
     // массив двоичных разрядов числа
     int* a=new int[i];
     if(a==NULL)
     {
          printf("Для хранения двоичного кода
               числа не хватает памяти\n");
          return;
     }
     i=0;
```

```
        x=k;
        while(k>0)
        {
              a[i]=k%2;
              k=k/2;
              i++;
        }
        for(int i1=0; i1<i/2; i1++)
        {
              x=a[i1];
              a[i1]=a[i-i1-1];
              a[i-i1-1]=x;
        }
        printf("Двоичный код числа %d=%c",n,z);
        for(i1=0;i1<i; i1++)
              printf("%d",a[i1]);
        printf("\n");
        delete [] a;
}
```

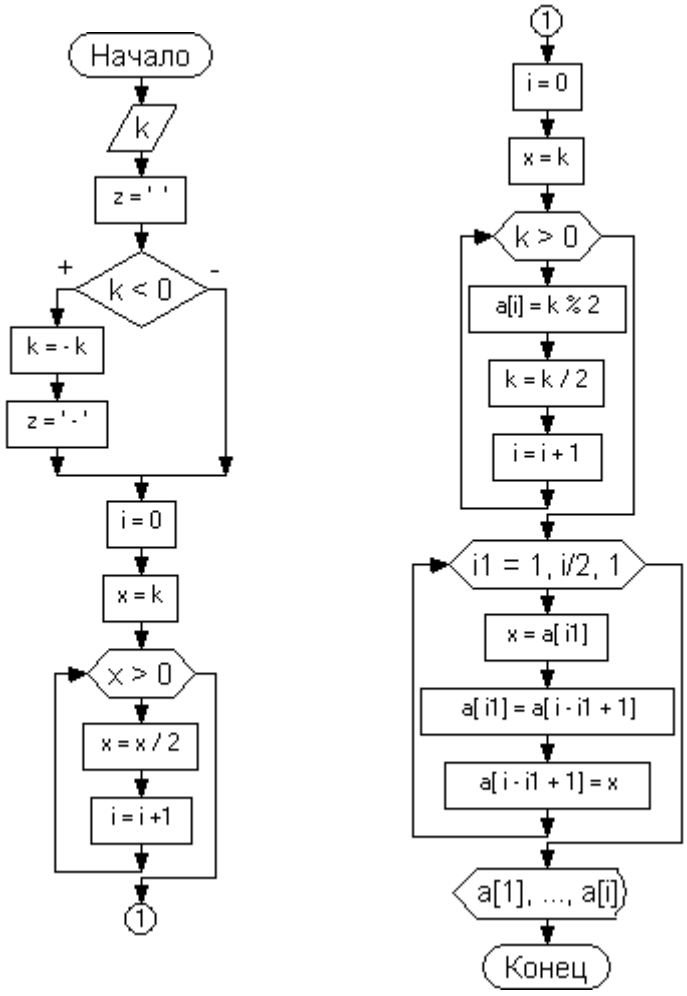


Рис.4.14. Блок-схема решения задачи о двоичном представлении числа.

Домашнее задание

1. Дан массив чисел. Определить, является ли он возрастающей последовательностью.

2. Найти в массиве возрастающую подпоследовательность максимальной длины. Под термином “подпоследовательность” будем понимать некоторый набор соседних элементов массива.

3. Найти в массиве чисел симметричную подпоследовательность максимальной длины.

4. Даны два массива - **a** размера n и **b** размера m ($m > n$). Определить, содержится ли массив **a** в массиве **b**.

5. Дан массив, элементами которого являются результаты забега спортсменов на 100 м. Составить команду из 4 лучших спортсменов.

6. Дан массив. Определить количество различных элементов этого массива.

7. Дан массив. Определить, образуют ли элементы массива множество (все элементы множества должны быть уникальны).

8. Даны два массива, образующие элементы двух множеств. Определить, равны ли множества (для множеств не имеет значения порядок следования элементов).

9. Даны два массива, содержащие элементы двух множеств. Получить массивы, являющиеся

а) объединением,

б) пересечением,

в) разностью исходных множеств.

Раздел 5. Строки

Строки – это специальный вид массивов, элементами которого являются символы. Особенность строк заключается в том, что не все элементы массива могут быть заполнены. Например, для хранения фамилии студента выделяется символьный массив длины, достаточной для хранения любой фамилии (к примеру, 30 символов). Однако естественно, что разные фамилии имеют различную длину в символах, т.е. размер содержательной части символьной строки будет разным. Поэтому используется специальный символ, чтобы задать признак конца символьной строки (в алгоритмах он обозначается как **EOL** – end-of-line). Например, в языке C++ **EOL** – это **'\0'** (ноль-символ).

При работе со строками часто встречаются задачи, в которых требуется анализировать не отдельные символы, а слова. Будем полагать, что слова состоят из любых символов, кроме пробела и символа **'\0'**. Слова в строке могут разделяться любым числом пробелов. Пробелы также могут находиться в начале и в конце символьной строки.

Задача 1. Найти количество символов в символьной строке.

Алгоритм этой задачи прост: подсчитываем все символы до тех пор, пока не встретится символ конца строки (**'\0'**). Блок-схема решения этой задачи приведена на Рис.5.1.

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
{
     char s[100];
     printf("Введите символьную строку:\n");
     gets(s);
     int k=0;
     int i=0;
     while(s[i]!='\0')
     {
          k++; i++;
     }
     printf("Длина строки = %d\n",k);
}
```

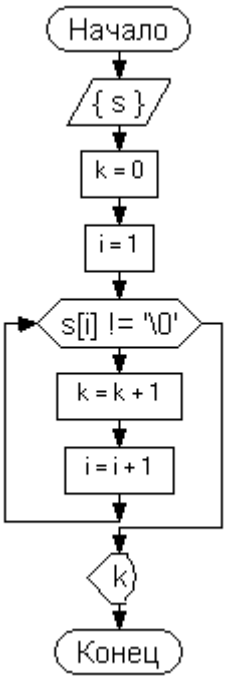


Рис.5.1. Блок-схема решения задачи
о количестве символов в строке.

Задача 2. Подсчитать количество слов в символьной строке.

Для решения этой задачи достаточно определить количество первых символов слов. Слово начинается, если текущий символ – пробел, а следующий – не пробел и не символ конца строки. Это условие не будет выполняться, если первый символ строки – это начало слова. Поэтому требуется рассмотреть этот случай отдельно.

Блок-схема представлена на Рис.5.2.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
     char s[100];
     printf("Введите символьную строку:\n");
     gets(s);
     int k;
     if(s[0]==' ' || s[0]=='\0')
          k=0;
     else k=1;
     int i=0;
     while(s[i]!='\0')
     {
          if(s[i]==' ' && s[i+1]!=' ' && s[i+1]!='\0')
               k++;
          i++;
     }
```

```
        printf("Количество слов в строке = %d\n",k);
}
```

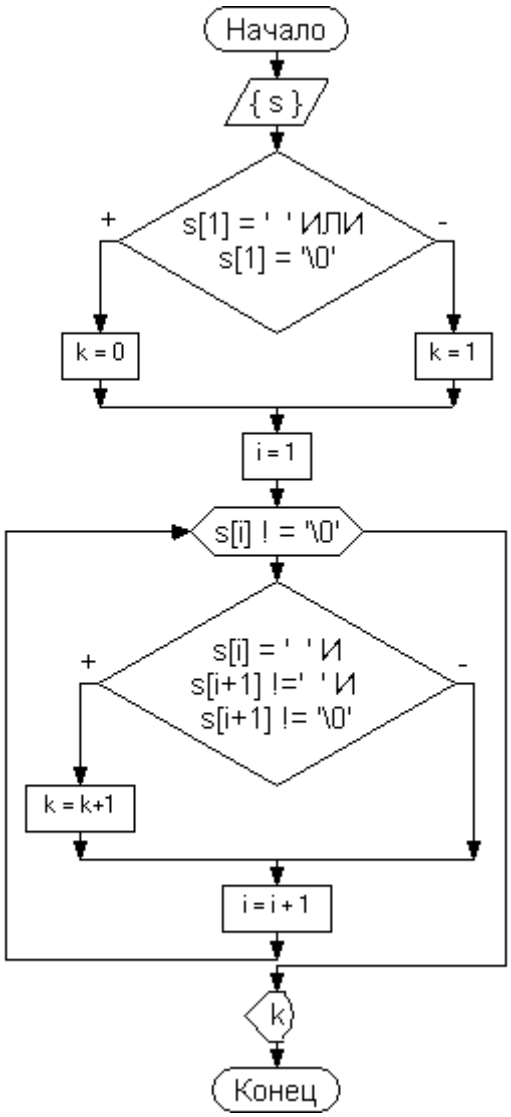


Рис.5.2. Блок-схема решения задачи о количестве слов в строке.

Задача 3. Из заданной символьной строки распечатать слово максимальной длины.

Решаем эту задачу следующим образом. Последовательно в символьной строке выделяем группы “пробелы - слово”. Сначала пропускаем все пробелы, предшествующие текущему слову. Затем подсчитываем количество символов в слове (двигаемся до пробела или до **’\0’**). При этом требуется запомнить позицию, с которой начинается текущее слово. Если количество символов текущего слова больше, чем в достигнутом на данный момент максимуме, запоминаем позицию начала слова и его длину. Далее переходим к рассмотрению следующей группы “пробелы - слово”. Дойдя до конца

строки, печатаем самое длинное слово или же выдаем сообщение о том, что строка не содержит слов.

Блок-схема решения задачи 3 представлена на Рис.5.3.

Код программы для задачи 3.

```
# include <stdio.h>
void main(void)
{
     char s[100];
     printf("Введите строку:\n");
     gets(s);
     int max=0, // длина максимального слова
       imax=0;  //позиция первого символа максимал.слова
     int i=0;
     while(s[i]!='\0')
     {
          while(s[i]==' ')
               i++;
          int k=0;        // длина слова
          while(s[i]!=' ' && s[i]!='\0')
          {
               k++; i++;
          }
          if(k>max)
          {
               max=k; imax=i-max;
          }
     }
     if(max>0)
     {
          for(i=imax; i<imax+max; i++)
               printf("%c",s[i]);
          printf("\n");
     }
     else
          printf("В строке нет слов\n");
}
```

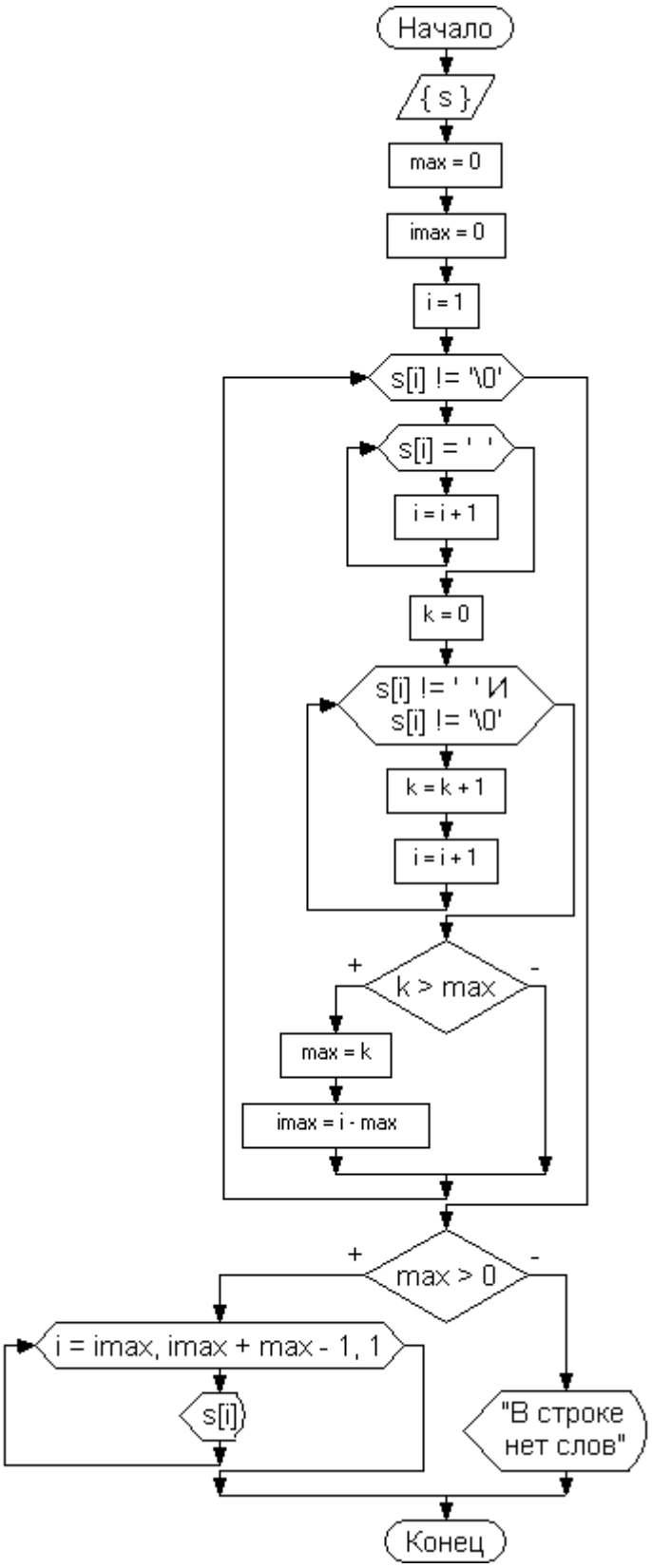


Рис.5.3. Блок-схема решения задачи о слове максимальной длины.

Задача 4. Проверить, входит ли заданное слово в символьную строку.

При решении этой задачи сначала требуется получить количество символов в заданном слове. Далее осуществляется просмотр слов символьной строки (алгоритм выделения слов из строки рассмотрен при решении задачи 3). Каждое выделенное слово побуквенно сравнивается с заданным словом. Требуется, чтобы не только совпали все буквы слова, но и следом за ним в строке стоял символ-разделитель, т.е. пробел или символ конца строки. Например, “сад” и “садовник” - два разных слова. Строка, содержащая слово “садовник”, не содержит слово “сад”.

Блок-схема решения задачи 4 представлена на Рис.5.4.

Код программы для задачи 4.

```
# include <stdio.h>
void main(void)
{
     char s[100];
     printf("Введите символьную строку:\n");
     gets(s);
     char c[20];
     printf("Введите слово:\n");
     scanf("%s", c);
     int k=0;
     while(c[k]!='\0')
          k++;
     int i=0,
         f=0;  // переменная-флажок
               // f=1, когда будет найдено искомое слово
     while(s[i]!='\0' && f==0)
     {
          while(s[i]==' ')
               i++;
          int t=0;
          while(s[i]==c[t] && s[i]!='\0')
          {
               i++;t++;
          }
          if(t==k && (s[i]==' ' || s[i]=='\0'))
               f=1;
          while(s[i]!=' ' && s[i]!='\0') i++;
     }
      if(f==1)
          printf("Слово %s входит в строку\n", c);
     else
          printf("Слово %s не входит в строку\n",c);
}
```

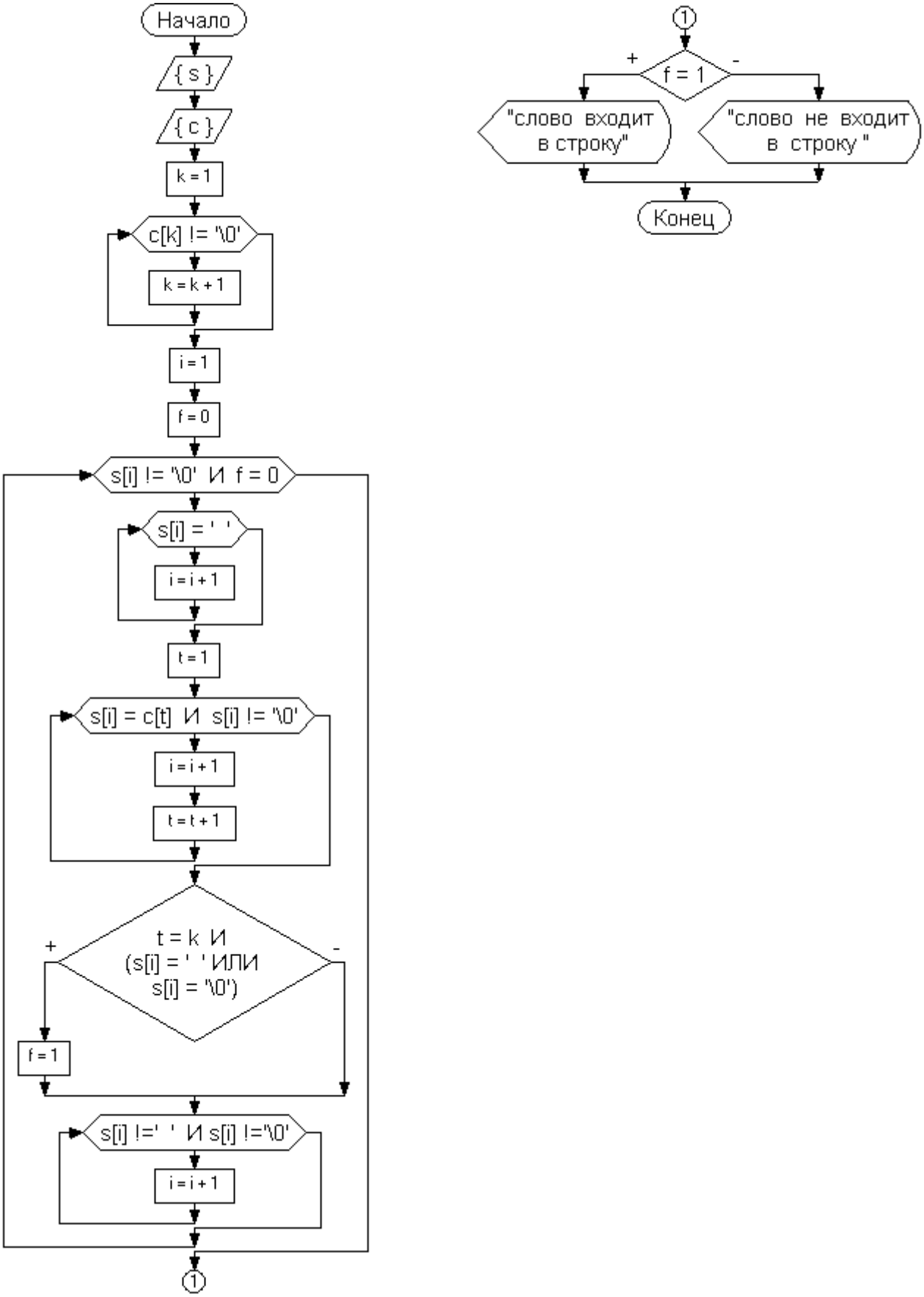


Рис.5.4. Блок-схема решения задачи о вхождении заданного слова в символьную строку.

Задача 5. В заданной символьной строке оставить между словами по

одному пробелу и удалить лишние пробелы в начале и в конце строки.

Алгоритм решения задачи таков. Двигаясь по символьной строке, подсчитываем суммарное количество лишних пробелов между словами (переменная **k**). Дойдя до очередного слова, сдвигаем его на **k** позиций влево, оставив между словами только один пробел. Затем повторяем процедуру до тех пор, пока не закончится строка. Отдельно удаляем пробел после последнего слова, если будут ”лишние” пробелы в конце строки.

Блок-схема решения задачи представлена на Рис.5.5.

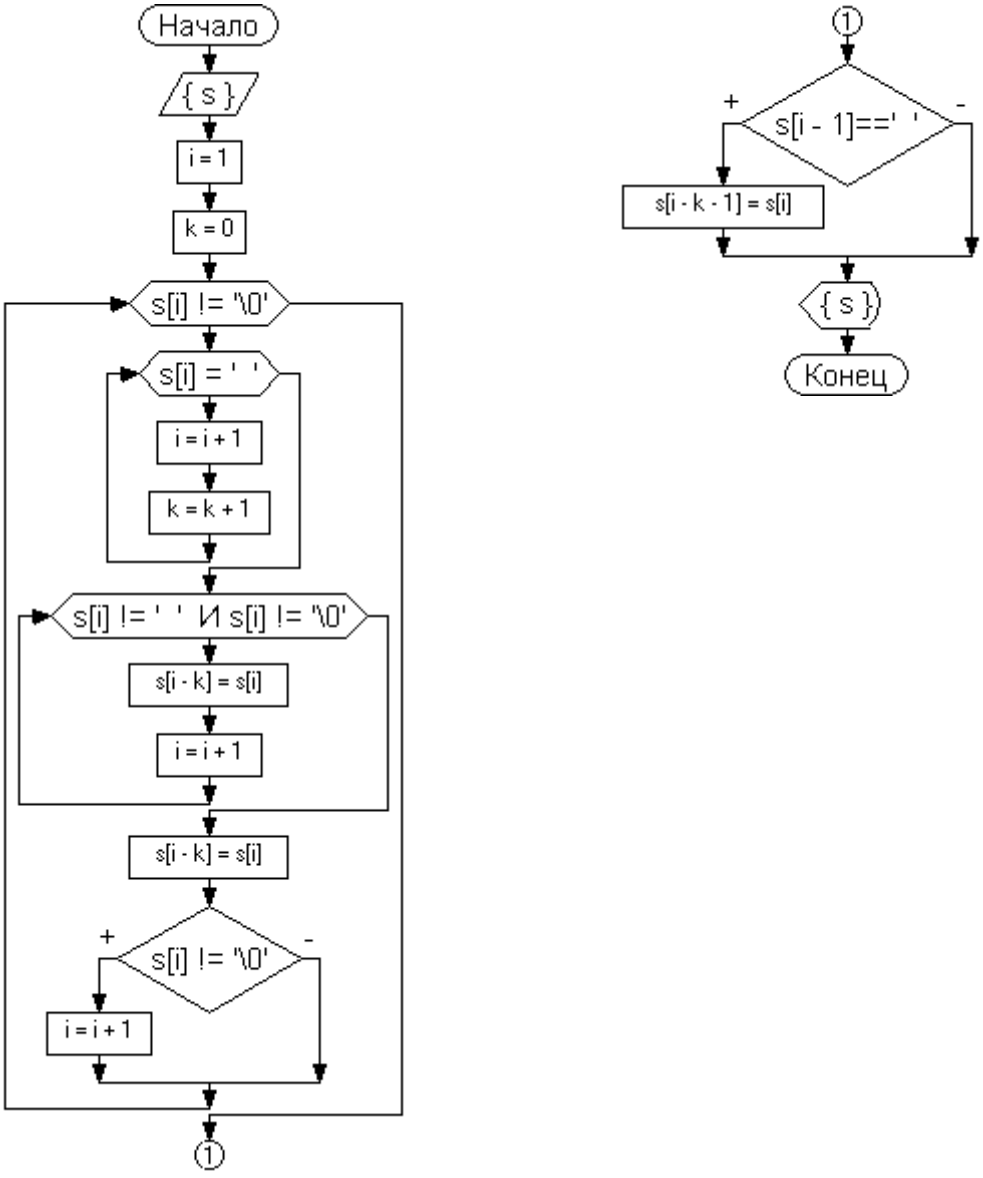


Рис.5.5. Блок-схема решения задачи об удалении “лишних” пробелов.

Код программы для задачи 5.

```
# include <stdio.h>
```

```
void main(void)
{
     char s[100];
     printf("Введите строку:\n");
     gets(s);
     int i=0,
         k=0;  // количество "лишних" пробелов
     while(s[i]!='\0')
     {
          while(s[i]==' ')
          {
               k++; i++;
          }
          while(s[i]!=' ' && s[i]!='\0')
          {
               s[i-k]=s[i];
               i++;
          }
          s[i-k]=s[i];
          if(s[i]!='\0')
               i++;
     }
     printf("Полученная строка:\n");
     puts(s);
}
```

Задача 6. В заданной символьной строке поменять местами слова минимальной и максимальной длины.

Задача разбивается на две части. Первая часть заключается в поиске слов максимальной и минимальной длины (алгоритм рассматривался при решении задачи 3).

Вторая часть заключается собственно в перестановке найденных слов. Эту подзадачу также условно можно разделить на два этапа. Пусть длина минимального слова хранится в переменной **min**, а максимального – в переменной **max**. Сначала меняем местами первые буквы слов в количестве **min** штук. Затем нужно вставить на требуемое место "хвостик" от самого длинного слова. Эту вставку осуществляем по одному символу, смещая ту часть строки, которая находится между словами, на один символ вправо или влево соответственно в зависимости от того, какое из слов – минимальное или максимальное, – встретилось раньше. На освободившееся место устанавливаем очередной символ из “хвостика”. Повторяем такую последовательность действий столько раз, какова разница между длинами слов.

Блок-схема решения этой задачи представлена на Рис.5.6 и 5.7.

Код программы для задачи 6.

```
# include <stdio.h>
void main(void)
{
     char s[100];
     printf("Введите строку:\n");
     gets(s);
     int imax=0,    // позиция слова максимальной длины
          imin=0,   // позиция слова минимальной длины
          max=0,    // длина максимального слова
          min=100,  // длина минимального слова
                    // (нач. значение - длина строки)
     i=0;
     int k;
     while(s[i]!='\0')
     {
          while(s[i]==' ')
               i++;
          k=0;
          while(s[i]!=' ' && s[i]!='\0')
          {
               i++; k++;
          }
          if(k>max)
          {
               max=k; imax=i-k;
          }
          if(k<min)
          {
               min=k;
               imin=i-k;
          }
     }
     // меняем местами первые символы слов в количестве min
     for(k=0;k<min;k++)
     {
          char z=s[imax+k];
          s[imax+k]=s[imin+k];
          s[imin+k]=z;
     }
     // ставим на место "хвостик" максимального слова
     if(imin>imax)
     {
          for(i=0;i<max-min;i++)
          {
               int f=s[imax+min];
```

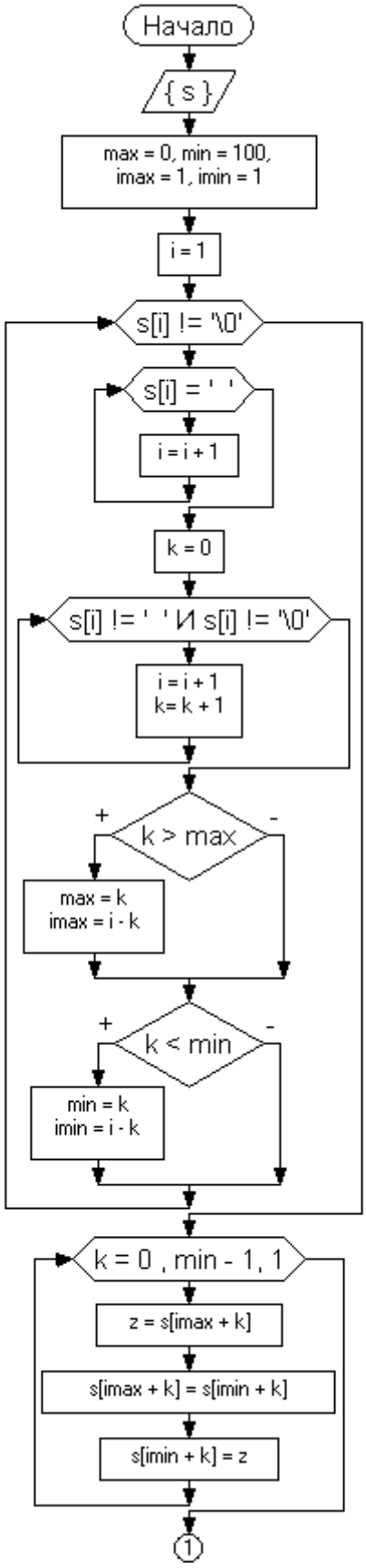


Рис.5.6. Начало блок-схемы решения задачи о перестановке слов максимальной и минимальной длины

```
            for(int j=imax+min+1;j<imin+min;j++)
                        s[j-1]=s[j];
                  s[j-1]=f;
            }
      }
      else
      {
            for(i=0;i<max-min; i++)
            {
                  int f=s[imax+max-1];
                  for(int j=imax+max-1;j>=min+imin+1; j--)
                        s[j]=s[j-1];
                  s[j]=f;
            }
      }
      printf("Полученная строка:\n");
      puts(s);
}
```

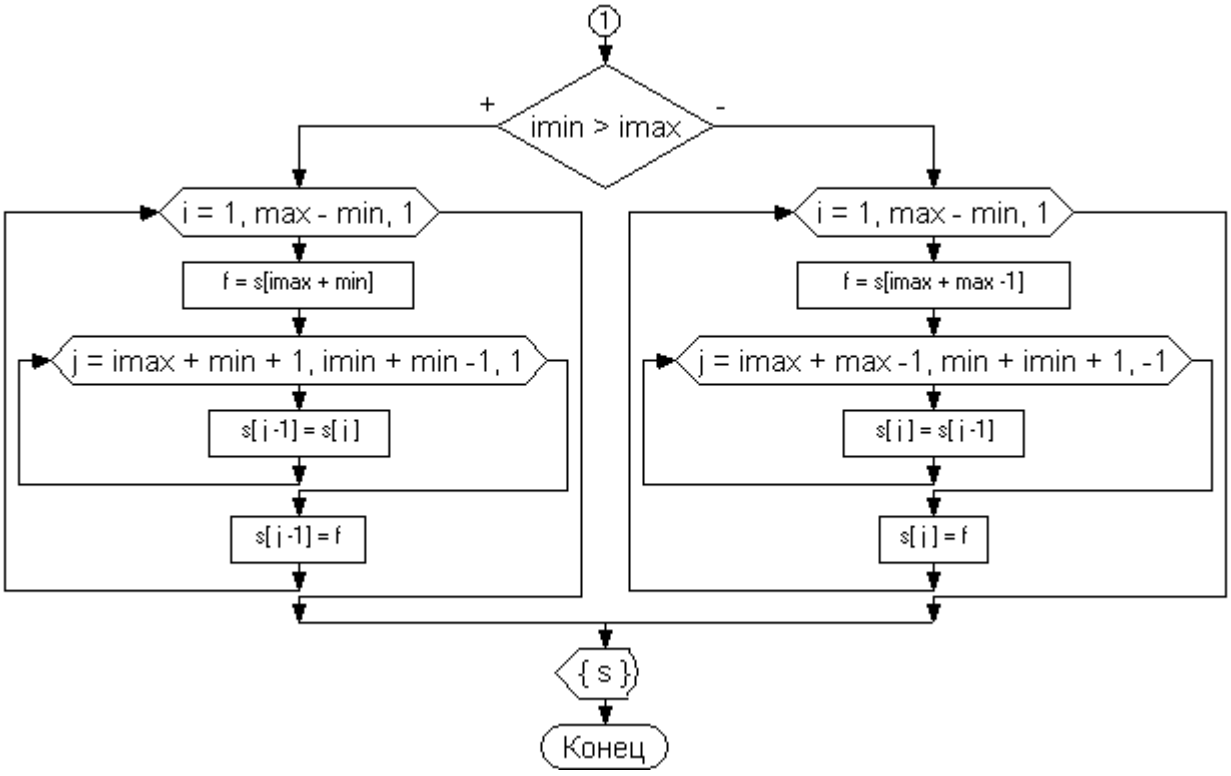


Рис.5.7. Окончание блок-схемы решения задачи о перестановке слов максимальной и минимальной длины

Домашнее задание

1. Дана символьная строка. Распечатать все слова, начинающиеся на заданную букву.

2. Дана символьная строка. Найти в строке симметричное слово (палиндром) максимальной длины.

3. Дана символьная строка. Сформировать строку, слова в которой идут в обратном порядке.

4. Дана символьная строка. Удалить из каждого слова все последующие вхождения его первой буквы.

5. Дана символьная строка. Удалить из строки слова длиной менее трех букв.

6. Даны две символьные строки. Первая строка содержит пары “слово - синоним”. Во второй строке заменить все слова на их синонимы, если такие найдены.

7. Дана символьная строка. Определить слово с максимальной долей гласных букв (процент гласных букв в слове).

Раздел 6. Двумерные массивы (матрицы)

Двумерные массивы (матрицы) являются примером использования табличной структуры данных. Данные здесь упорядочиваются по двум измерениям – строкам и столбцам. Таким образом, для обозначения элемента матрицы используется два индекса – индексы строки и столбца, на пересечении которых находится этот элемент. Например, чтобы обратиться к элементу матрицы **a**, находящемуся в 3-ей строке и 1-ом столбце, надо использовать следующую запись: **a[3][1]**. Для того чтобы просмотреть все элементы двумерного массива, требуется "пробежаться" по всем строкам, а внутри каждой строки – по столбцам. Поэтому для ввода элементов матрицы используют два цикла.

Задача 1. Транспонировать квадратную матрицу (поменять местами элементы так, чтобы строки матрицы стали столбцами, а столбцы – строками).

Для транспонирования матрицы достаточно поменять местами элементы, стоящие симметрично относительно главной диагонали. Элементы главной диагонали расположены так, что индексы строки и столбца у них совпадают.

Будем перебирать те элементы из симметричных пар, которые находятся ниже главной диагонали (у них номер столбца меньше номера строки). Поэтому осуществляем перебор всех строк (переменная **x**) и для каждой строки перебор столбцов (переменная **y**) от 1 до **x**-1. Для каждой пары индексов **(x,y)** меняем местами элементы **a[x][y]** и **a[y][x]**.

При построении блок-схемы решения задачи 1 для ввода и вывода матрицы будем использовать следующие группы блоков:

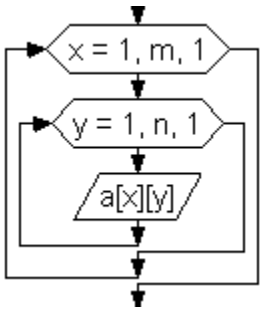


Рис.6.1. Блок ввода матрицы

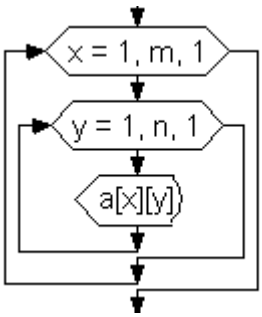


Рис.6.2. Блок печати матрицы

При решении следующих задач заменим их более короткой записью (Рис.6.3 и 6.4):

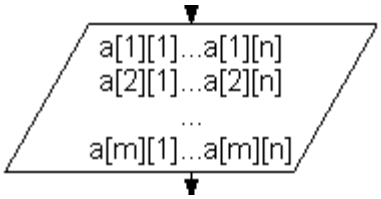


Рис.6.3. Блок ввода матрицы

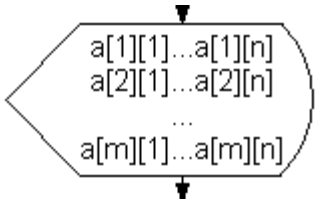


Рис.6.4. Блок печати матрицы

Блок-схема решения задачи представлена на Рис.6.5.

Код программы для задачи 1.

```
# include <stdio.h>
void main(void)
{
     int n;
     printf("Введите размер матрицы n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительный размер\n");
          return;
     }
     // выделение памяти под матрицу
     // a - массив адресов строк матрицы
     int **a=new int*[n];
     if (a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньший размер.\n");
          return;
     }
          for(int x=0;x<n; x++)
          {
               // выделение памяти под x-тую строку матрицы
               a[x]=new int[n];
               if (a[x]==NULL)
               {
                    printf("Запрошено большое количество
                           памяти.\n Попробуйте ввести
                         меньший размер.\n");
                    for (int j=0; j<x; j++)
                    delete [] a[j];
               delete [] a;
               return;
          }
     }
```

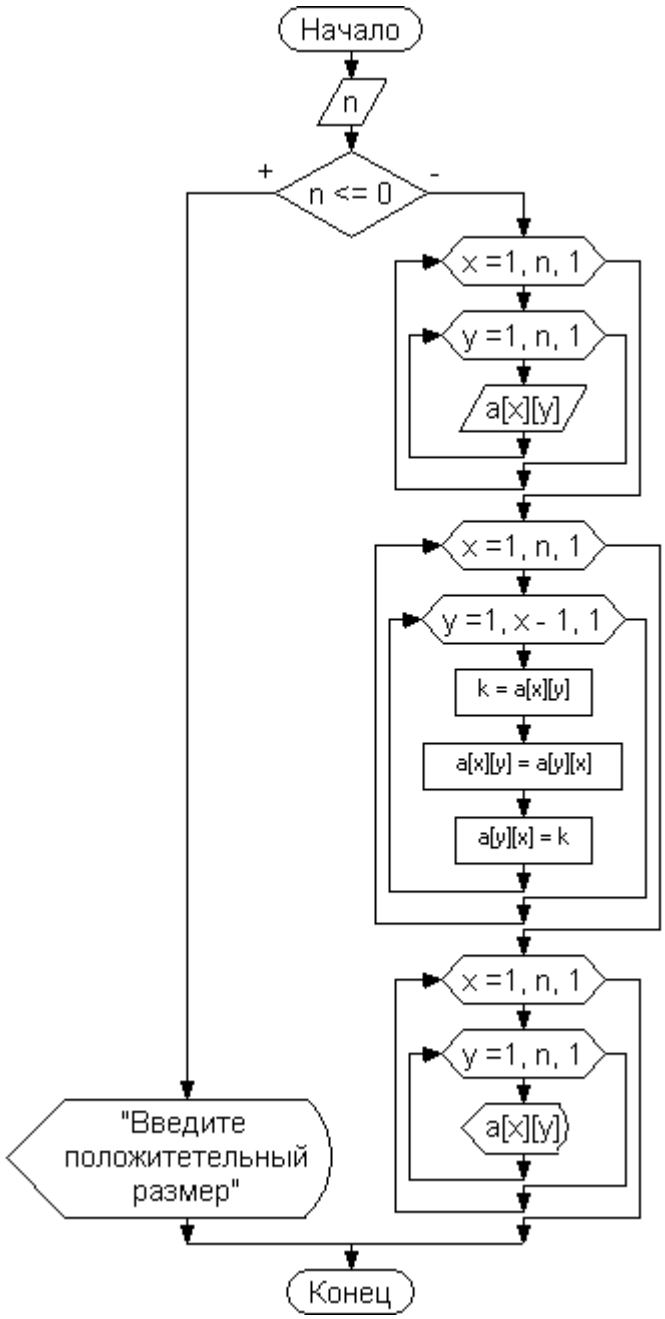


Рис.6.5. Блок-схема решения задачи
о транспонировании квадратной матрицы.

```
printf("Введите элементы матрицы:\n");
for(x=0;x<n;x++)
     for(int y=0;y<n; y++)
          scanf("%d",&a[x][y]);
for(x=0;x<n;x++)
     for(int y=0;y<x; y++)
     {
          int k=a[x][y];
          a[x][y]=a[y][x];
          a[y][x]=k;
```

```
        }
    printf ("Транспонированная матрица:\n");
    for(x=0;x<n;x++)
    {
        for(int y=0;y<n; y++)
            printf("%d\t",a[x][y]);
        printf("\n");
    }
    // освобождение памяти
    for(x=0;x<n;x++)
        delete [] a[x];
    delete [] a;
}
```

Задача 2. Дана прямоугольная матрица. Все максимальные элементы матрицы заменить на 0.

Перебирая все элементы матрицы, находим максимальный элемент (аналогично поиску максимума в одномерном массиве). Далее еще раз обходим всю матрицу и заменяем элемент нулем при его совпадении с максимумом.

Блок-схема представлена на Рис.6.6.

Код программы для задачи 2.

```
# include <stdio.h>
void main(void)
{
    int m,n;
    printf("Введите количество строк матрицы:");
    scanf("%d",&m);
    printf("Введите количество столбцов матрицы:");
    scanf("%d",&n);
        if(n<=0 || m<=0)
    {
        printf("Введите положительные размеры\n");
        return;
    }
    int **a=new int*[m];
    if (a==NULL)
    {
        printf("Запрошено большое количество памяти.\n
            Попробуйте ввести меньший размер.\n");
        return;
    }
```

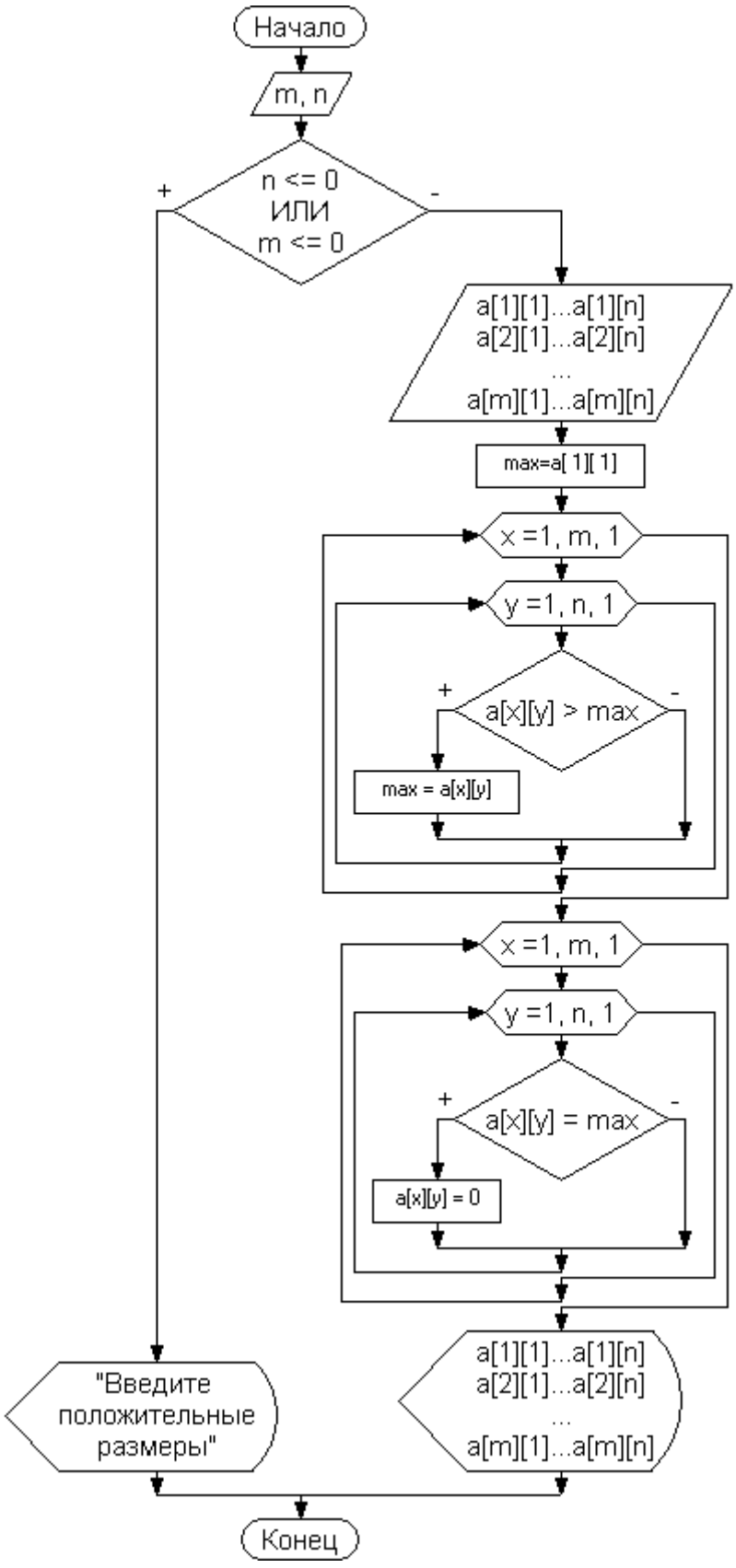


Рис.6.6. Блок-схема решения задачи о замене максимальных элементов.

```
        for(int x=0;x<m; x++)
        {
             a[x]=new int[n];
             if (a[x]==NULL)
             {
                  printf("Запрошено большое количество
                          памяти.\n Попробуйте ввести
                          меньший размер.\n");
                  for (int j=0; j<x; j++)
                       delete [] a[j];
                  delete [] a;
                  return;
             }
        }
        printf("Введите элементы матрицы:\n");
        for(x=0;x<m;x++)
             for(int y=0;y<n; y++)
                  scanf("%d",&a[x][y]);
        int max=a[0][0];
        for(x=0;x<m;x++)
             for(int y=0;y<n;y++)
                  if(a[x][y]>max)
                       max=a[x][y];
        for(x=0;x<m;x++)
             for(int y=0;y<n;y++)
                  if(a[x][y]==max)
                       a[x][y]=0;
        printf("Полученная матрица: \n");
        for(x=0;x<m;x++)
        {
             for(int y=0;y<n; y++)
                  printf("%d\t",a[x][y]);
             printf("\n");
        }
        for(x=0;x<m;x++)
             delete [] a[x];
        delete [] a;
   }
```

Задача 3. Найти произведение двух матриц размеров $m \times n$ и $k \times t$.

По правилам перемножения матриц количество столбцов первой матрицы должно совпадать с количеством строк второй матрицы. Поэтому сначала требуется проверить это условие.

Если размеры были введены корректно, осуществляется перемножение. Каждый элемент матрицы-результата вычисляется как скалярное произведение соответствующей строки первой матрицы на столбец второй матрицы. Поэтому при решении задачи создается три цикла: первые два перебирают элементы результирующей матрицы, а третий используется для вычисления самого элемента.

Блок-схема представлена на Рис.6.7.

Код программы для задачи 3.

```
# include <stdio.h>
void main(void)
{
     int m,n,k,t;
     printf("Введите количество строк первой матрицы m:");
     scanf("%d",&m);
     printf("Введите количество столбцов
               первой матрицы n:");
     scanf("%d",&n);
     printf("Введите количество строк второй матрицы k:");
     scanf("%d",&k);
     printf("Введите количество столбцов
               второй матрицы t:");
     scanf("%d",&t);
     if(m<=0 || n<=0 || k<=0 || t<=0)
     {
          printf("Введите положительные размеры\n");
          return;
     }
     if(n!=k)
     {
          printf("n должно быть равно k\n");
          return;
     }
     int **a=new int*[m];
     int **b=new int*[k];
     int **c=new int*[m];
     if(a==NULL || b==NULL || c==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньший размер.\n");
          return;
     }
     for(int x=0; x<m; x++)
     {
          a[x]=new int[n];
          if (a[x]==NULL)
          {
               printf("Запрошено большое количество
                     памяти.\n Попробуйте ввести
                     меньший размер.\n");
               for (int j=0; j<x; j++)
                    delete [] a[j];
               delete [] a;
               return;
          }
     }
```

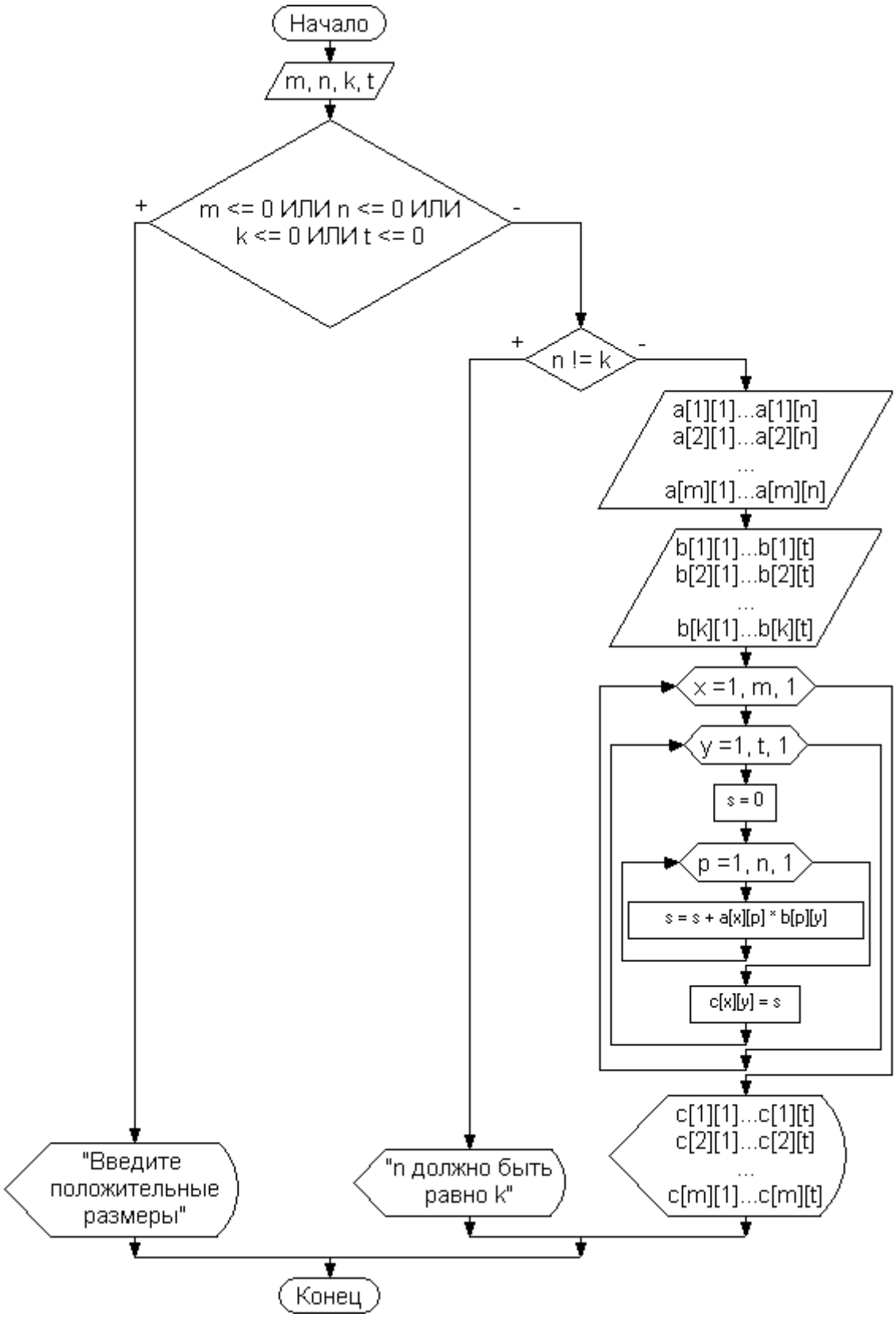


Рис.6.7. Блок-схема решения задачи о перемножении матриц.

```
for(x=0;x<k; x++)
{
     b[x]=new int[t];
     if (b[x]==NULL)
     {
          printf("Запрошено большое количество
                памяти.\n Попробуйте ввести
                меньший размер.\n");
          for (int j=0; j<x; j++)
               delete [] b[j];
          for(j=0; j<m; j++)
               delete [] a[j];
          delete [] a;
          delete [] b;
          return;
     }
}
for(x=0;x<m; x++)
{
     c[x]=new int[t];
     if (c[x]==NULL)
     {
          printf("Запрошено большое количество
                памяти.\n Попробуйте ввести
                меньший размер.\n");
          for (int j=0; j<x; j++)
               delete [] c[j];
          for(j=0; j<m; j++)
               delete [] a[j];
          delete [] a;
          for(j=0; j<k;j++)
               delete [] b[j];
          delete [] b;
          delete [] c;
          return;
     }
}
printf("Введите матрицу A:");
for(x=0;x<m;x++)
     for(int y=0;y<n; y++)
          scanf("%d",&a[x][y]);
printf("Введите матрицу B:");
for(x=0;x<k;x++)
     for(int y=0;y<t; y++)
          scanf("%d",&b[x][y]);
// вычисление произведения матриц
for(x=0;x<m;x++)
     for(int y=0; y<t; y++)
     {
          int s=0;
          for(int p=0;p<n;p++)
               s=s+a[x][p]*b[p][y];
```

```
                c[x][y]=s;
            }
        printf("Полученная матрица:\n");
        for(x=0;x<m;x++)
        {
            for(int y=0;y<t; y++)
                printf("%d\t",c[x][y]);
            printf("\n");
        }
        for(x=0;x<m;x++)
            delete [] a[x];
        for(x=0;x<n;x++)
            delete [] b[x];
        for(x=0;x<m;x++)
            delete [] c[x];
        delete [] a;
        delete [] b;
        delete [] c;
}
```

Задача 4. Дана квадратная матрица. Поменять в ней местами элементы главной и побочной диагоналей.

Здесь сделаем только одно пояснение: элемент, находящийся в **x**-ой строке на главной диагонали, имеет индексы (x,x), а соответствующий элемент побочной диагонали - $(x,n-x-1)$. Поэтому требуется просто перебрать строки матрицы и поменять в них местами элементы, имеющие такие индексы. Блок-схема решения задачи представлена на Рис.6.8.

Код программы для задачи 4.

```
# include <stdio.h>
void main(void)
{
    int n;
    printf("Введите размер матрицы n:");
    scanf("%d",&n);
    if(n<=0)
    {
        printf("Введите положительный размер\n");
        return;
    }
    int **a=new int*[n];
    if (a==NULL)
    {
        printf("Запрошено большое количество памяти.\n
            Попробуйте ввести меньший размер.\n");
        return;
    }
```

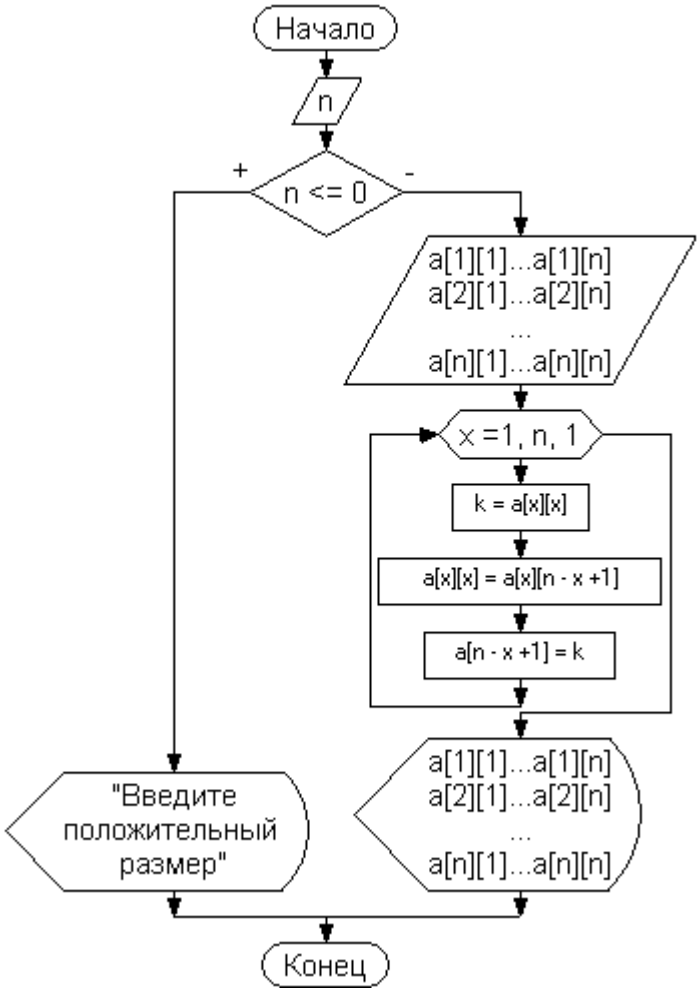


Рис.6.8. Блок-схема решения задачи о перестановке диагоналей квадратной матрицы.

```
for(int x=0;x<n; x++)
{
     a[x]=new int[n];
     if (a[x]==NULL)
     {
          printf("Запрошено большое количество
                памяти.\n Попробуйте ввести
                меньший размер.\n");
          for (int j=0; j<x; j++)
               delete [] a[j];
          delete [] a;
          return;
     }
}
printf("Введите элементы матрицы:\n");
for(x=0;x<n;x++)
     for(int y=0;y<n; y++)
          scanf("%d",&a[x][y]);
for(x=0;x<n;x++)
{
```

```
            int k=a[x][x];
            a[x][x]=a[x][n-x-1];
            a[x][n-x-1]=k;
      }
      printf("Полученная матрица:\n");
      for(x=0;x<n;x++)
      {
            for(int y=0;y<n; y++)
                  printf("%d\t",a[x][y]);
            printf("\n");
      }
      for(x=0;x<n;x++)
            delete [] a[x];
      delete [] a;
}
```

Задача 5. Заполнить двумерный массив размера *nxn* числами $1,2,...,n^2$ по спирали, например, при $n=4$ матрица должна выглядеть следующим образом:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 12 & 13 & 14 & 5 \\ 11 & 16 & 15 & 6 \\ 10 & 9 & 8 & 7 \end{pmatrix}$$

Заполнение одного витка спирали состоит из четырех циклов, определяющих последовательное заполнение элементов в направлениях слева направо, сверху вниз, справа налево и снизу вверх. Границы заполнения в соответствующей строке или столбце определяются элементами, стоящими на главной и побочной диагоналях. В случае нечетного значения **n** требуется заполнить последний элемент отдельно, поскольку он стоит на пересечении главной и побочной диагоналей.

Блок-схема решения этой задачи представлена на Рис.6.9.

Код программы для задачи 5.

```
# include <stdio.h>
void main(void)
{
      int n;
      printf("Введите размер матрицы n:");
      scanf("%d",&n);
      if(n<=0)
      {
            printf("Введите положительный размер\n");
            return;
```

```
}
int **a=new int*[n];
if (a==NULL)
{
     printf("Запрошено большое количество памяти.\n
          Попробуйте ввести меньший размер.\n");
     return;
}
for(int x=0;x<n; x++)
{
     a[x]=new int[n];
     if (a[x]==NULL)
     {
          printf("Запрошено большое количество
                памяти.\n Попробуйте ввести
                меньший размер.\n");
          for (int j=0; j<x; j++)
               delete [] a[j];
          delete [] a;
          return;
     }
}
int y,k; x=1;y=1;k=1;
while(k<n*n)
{
     // направление слева направо
     while(x+y<=n)
     {
          a[x-1][y-1]=k;
          k++;
          y++;
     }
     // направление сверху вниз
     while(x<y)
     {
          a[x-1][y-1]=k;
          k++;
          x++;
     }
     // направление справа налево
     while(x+y>n+1)
     {
          a[x-1][y-1]=k;
          k++;
          y--;
     }
     // направление снизу вверх
     while(x>y)
     {
          a[x-1][y-1]=k;
          k++;
          x--;
```

```
}
```

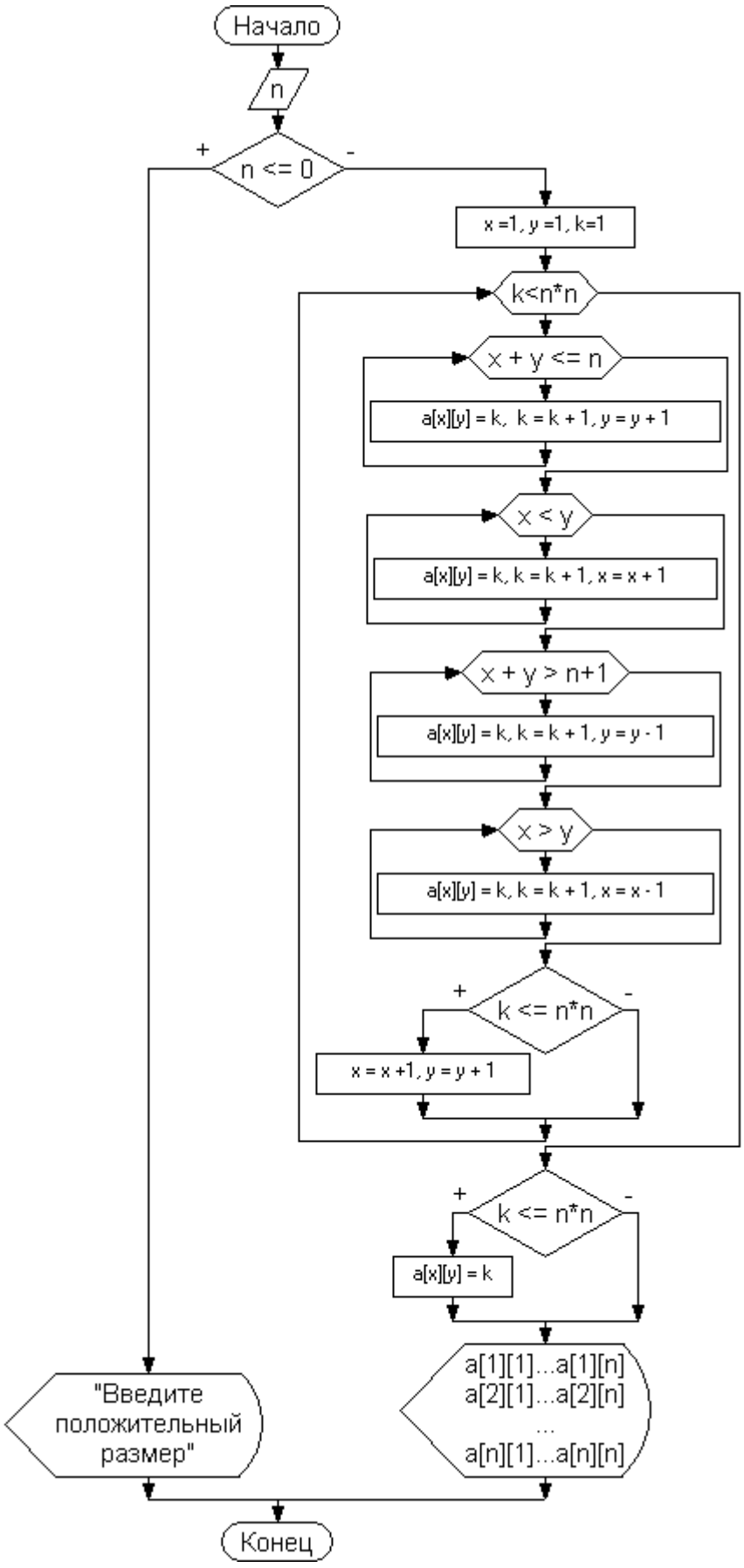


Рис.6.9. Блок-схема решения задачи о заполнении квадратной матрицы элементами по спирали.

```
            if(k<=n*n)
            {
                  x++; y++;
            }
      }
      // ставим последний элемент,
      // если он еще не установлен (n - нечетно)
      if(k<=n*n)
            a[x-1][y-1]=k;
      printf("Полученная матрица:\n");
      for(x=0;x<n;x++)
      {
            for(y=0;y<n;y++)
                  printf("%d\t",a[x][y]);
            printf("\n");
      }
      for(x=0;x<n;x++)
            delete [] a[i];
      delete [] a;
}
```

Задача 6. Вычислить определитель квадратной матрицы, приведя ее к треугольному виду.

Треугольной матрицей является такая квадратная матрица, в которой выше или ниже главной диагонали все элементы равны нулю. Определитель треугольной матрицы равен произведению элементов, стоящих на главной диагонали. Поэтому основная задача состоит в приведении матрицы к треугольному виду.

Будем приводить матрицу к виду верхней треугольной матрицы (элементы ниже главной диагонали равны 0). Последовательно для каждого столбца проводим следующую процедуру. Если элемент главной диагонали в этом столбце равен нулю, ищем ненулевые элементы в этом столбце в строках ниже. Если поиск окончится неудачей, то определитель равен 0. При нахождении ненулевого элемента в рассматриваемом столбце меняем местами две строки так, чтобы ненулевой элемент оказался на главной диагонали. При такой смене строк меняется знак определителя на противоположный.

Найдя ненулевой элемент на главной диагонали, можно исключить элементы, стоящие ниже его в том же столбце (сделать их 0). Для этого последовательно из нижних строк вычитается строка с рассматриваемым элементом главной диагонали, умноженная на коэффициент, гарантирующий,

что все элементы ниже рассматриваемого будут равны 0. При таких преобразованиях значение определителя сохранится.

Блок-схема решения этой задачи представлена на Рис.6.10.

Код программы для задачи 6.

```
# include <stdio.h>
void main(void)
{
     int n;
     printf("Введите размер матрицы n:");
     scanf("%d",&n);
     if(n<=0)
     {
          printf("Введите положительный размер\n");
          return;
     }
     float **a=new float*[n];
     if (a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
                Попробуйте ввести меньший размер.\n");
          return;
     }
     for(int x=0;x<n; x++)
     {
          a[x]=new float[n];
          if (a[x]==NULL)
          {
               printf("Запрошено большое количество
                       памяти.\n Попробуйте ввести
                       меньший размер.\n");
               for (int j=0; j<x; j++)
                    delete [] a[j];
               delete [] a;
               return;
          }
     }
     printf("Введите элементы матрицы:\n");
     for(x=0;x<n;x++)
          for(int y=0;y<n; y++)
               scanf("%f",&a[x][y]);
     float z=1;          // знак определителя
     for(int k=0;k<n-1; k++)
     {
          float d=a[k][k];    // элемент главной диагонали
          // если на главной диагонали стоит 0, то находим
          // ненулевой элемент ниже в том же столбце
          if(d==0)
          {
               x=k+1;
               int f=0;
```

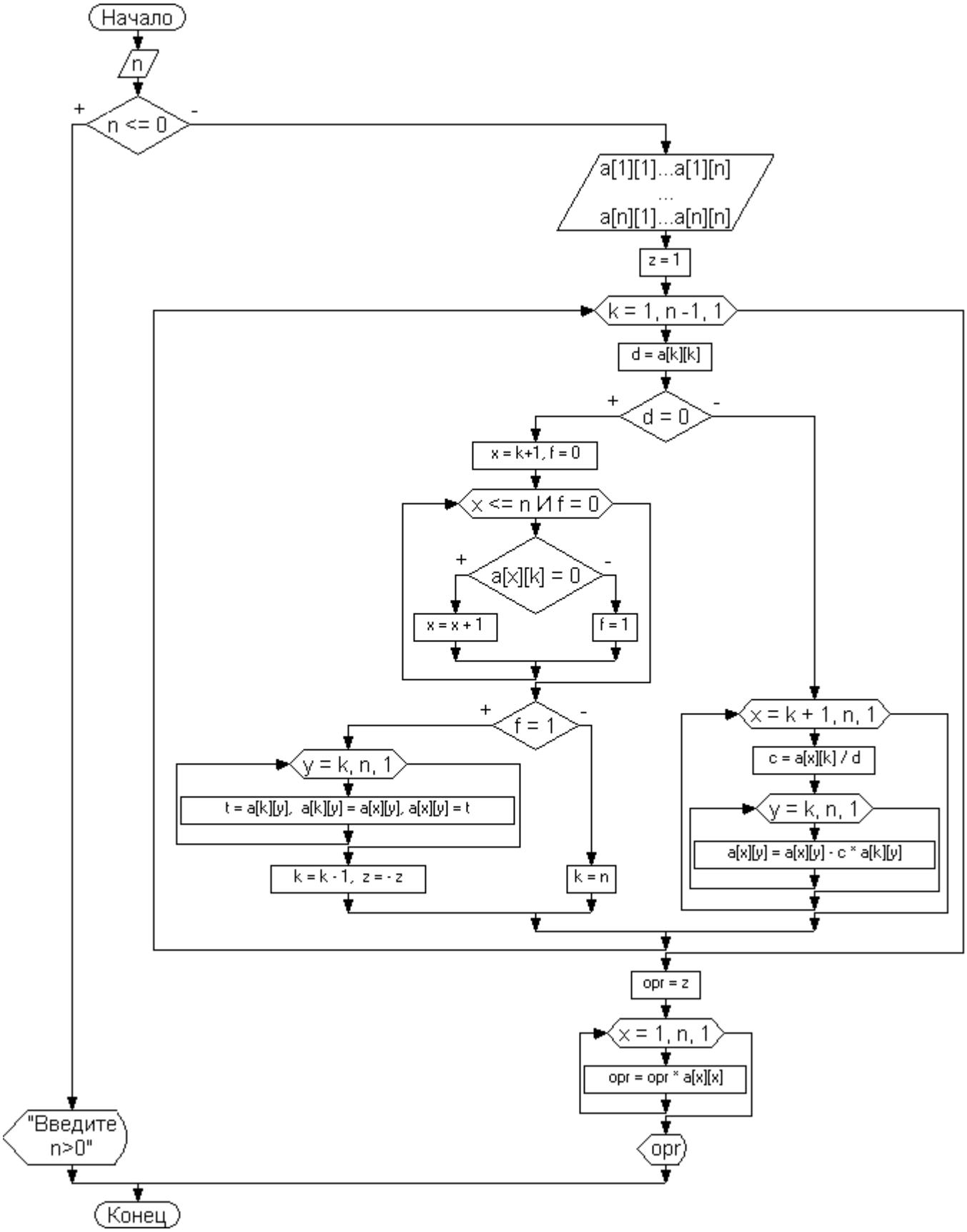


Рис.6.10. Блок-схема решения задачи

о вычислении определителя матрицы.

```
                while(x<n && f==0)
                     if(a[x][k]==0)
                          x++;
                     else
                          f=1;
                // если ненулевой элемент найден,
                // меняем строки местами
                if(f==1)
                {
                     z=-z;
                     for(int y=k;y<n;y++)
                     {
                          float t=a[k][y];
                          a[k][y]=a[x][y];
                          a[x][y]=t;
                     }
                     k--;
                }
                else break;
           }
           else
           {
                // обнуляем все элементы,
                // стоящие ниже в столбце
                for(x=k+1;x<n;x++)
                {
                     float c=a[x][k]/d;
                     for(int y=k;y<n;y++)
                          a[x][y]=a[x][y]-c*a[k][y];
                }
           }
      }
      printf("Матрица после приведения
                к треугольному виду:\n");
      for(x=0;x<n;x++)
      {
           for(int y=0;y<n;y++)
                printf("%f\t",a[x][y]);
           printf("\n");
      }
      // вычисление определителя как подсчет произведения
      // элементов главной диагонали с учетом знака
      float opr=z;
      for(x=0;x<n;x++)
           opr=opr*a[x][x];
      printf("Определитель = %f\n",opr);
      for(x=0;x<n;x++)
           delete [] a[i];
      delete [] a;
}
```

Задача 7. Распечатать все седловые точки прямоугольной матрицы.

Седловая точка - это пара индексов (номер строки, номера столбца), на пересечении которых находится элемент, максимальный в строке и одновременно минимальный в столбце, или наоборот – минимальный в строке и одновременно максимальный в столбце.

Для решения этой задачи в каждой строке матрицы находим максимальный и минимальный элементы. Затем все максимумы и минимумы строки проверяем на выполнение условия для седловых точек по столбцу. Если оно выполняется, то печатаем соответствующие пары индексов.

Блок-схема решения задачи представлена на Рис.6.11 и 6.12.

Код программы для задачи 7.

```
# include <stdio.h>
void main(void)
{
     int m,n;
     printf("Enter m:");
     scanf("%d",&m);
     printf("Enter n:");
     scanf("%d",&n);
     if(m<=0 || n<=0)
     {
          printf("Введите положительный размер\n");
          return;
     }
     int** a=new int*[m];
     if (a==NULL)
     {
          printf("Запрошено большое количество памяти.\n
               Попробуйте ввести меньший размер.\n");
          return;
     }
     for(int x=0;x<m; x++)
     {
          a[x]=new int[n];
          if (a[x]==NULL)
          {
               printf("Запрошено большое количество
                       памяти.\n Попробуйте ввести
                       меньший размер.\n");
               for (int j=0; j<x; j++)
                    delete [] a[j];
               delete [] a;
               return;
          }
     }
```

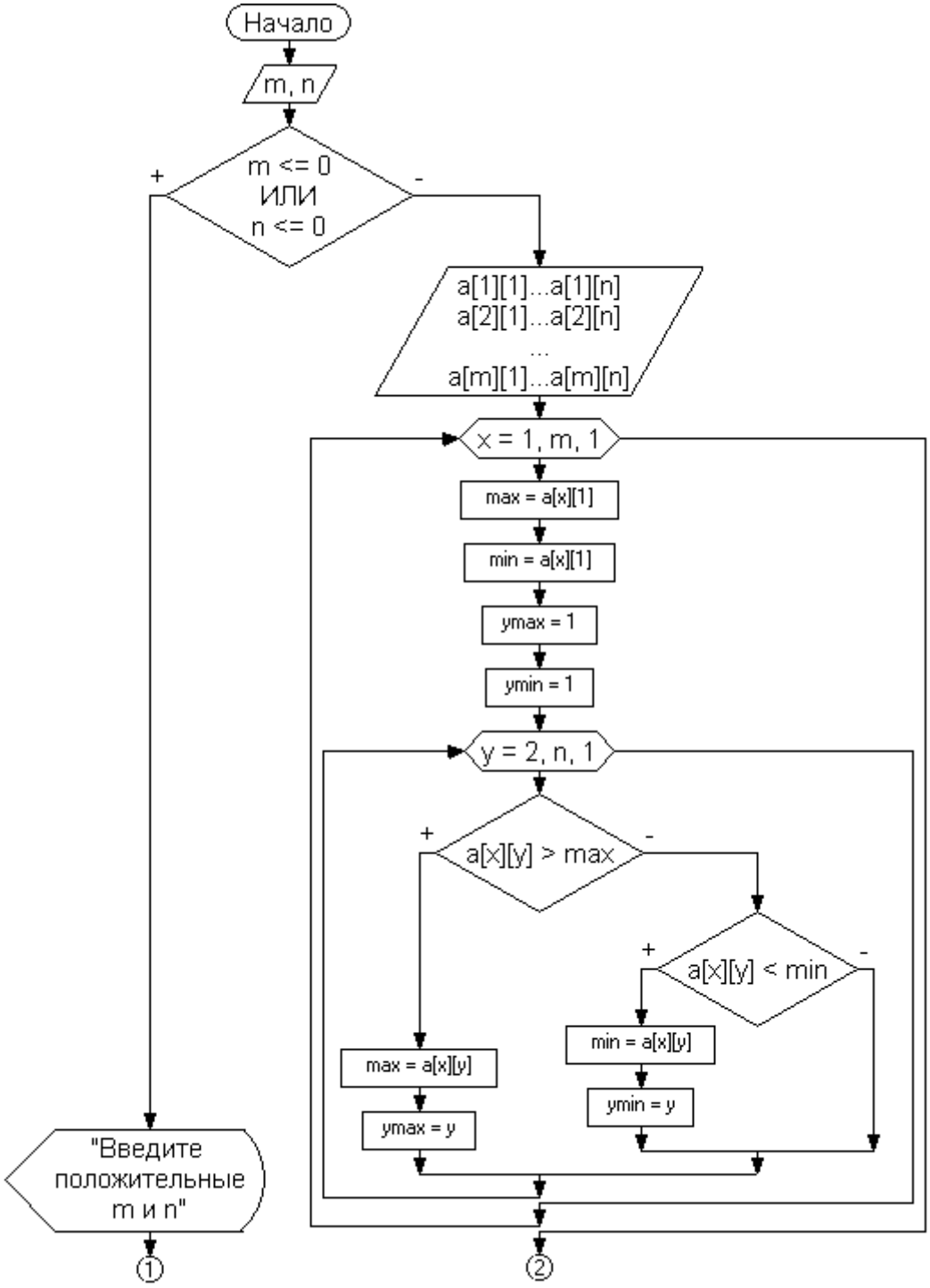


Рис.6.11. Начало блок-схемы решения задачи
об определении седловых точек матрицы.

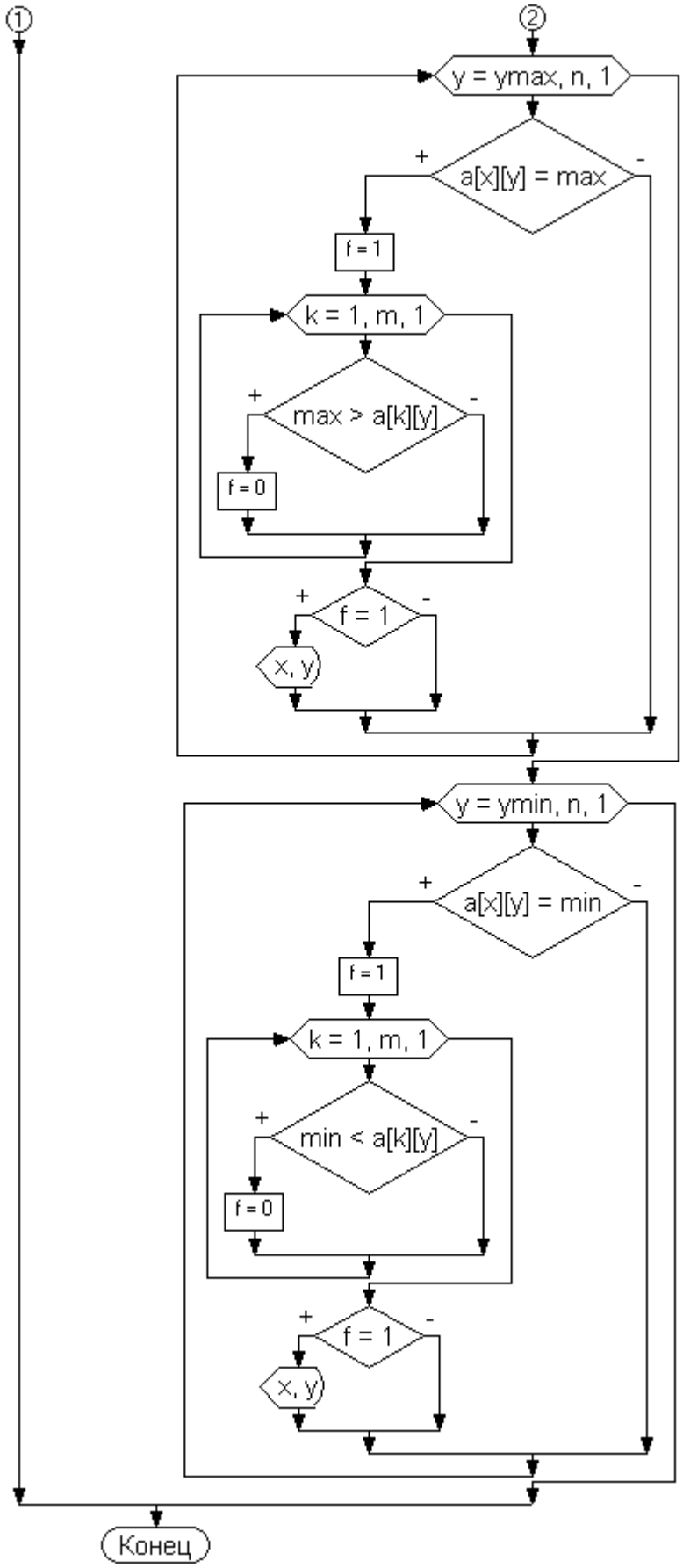


Рис.6.12.
Окончание блок-схемы решения задачи об определении седловых точек матрицы.

```
        printf("Enter A:");
        for(x=0;x<m;x++)
             for(int y=0;y<n;y++)
                  scanf("%d",&a[x][y]);
        for(x=0;x<m;x++)
        {
             // поиск максимального и минимального элемента
             // в x-ой строке
             int max=a[x][0], min=a[x][0];
             int ymax=0, ymin=0;
             for(int y=1;y<n;y++)
                  if(a[x][y]>max)
                  {
                       max=a[x][y];  ymax=y;
                  }
                  else
                       if(a[x][y]<min)
                       {
                            min=a[x][y];  ymin=y;
                       }
             // если максимальных элементов несколько,
             // проверяем их все на выполнение условия
             // для седловой точки в столбце
             for(y=ymax;y<n;y++)
                  if(a[x][y]==max)
                  {
                       int f=1;
                       for(int k=0;k<m;k++)
                            if(max>a[k][y])
                                 f=0;
                       if(f==1)
                            printf("(%d,%d)\t",x+1,y+1);
                  }
             // если минимальных элементов несколько,
             // проверяем их все на выполнение условия
             // для седловой точки в столбце
             for(y=ymin;y<n;y++)
                  if(a[x][y]==min)
                  {
                       int f=1;
                       for(int k=0;k<m;k++)
                            if(min<a[k][y])
                                 f=0;
                       if(f==1)
                            printf("(%d,%d)\t",x+1,y+1);
                  }
        }
        for(x=0;x<m;x++)
             delete [] a[x];
        delete [] a;
}
```

Домашнее задание

1. Найти сумму всех элементов прямоугольной матрицы.
2. Поменять местами максимальный и минимальный элементы прямоугольной матрицы
3. Найти сумму двух прямоугольных матриц.
4. Отсортировать строки прямоугольной матрицы в порядке возрастания их первых элементов.
5. Найти скалярное произведение строки и столбца квадратной матрицы, на пересечении которых находится минимальный элемент.
6. В прямоугольной матрице поменять местами столбцы с максимальной и минимальной суммой элементов.
7. Дана квадратная матрица. Распечатать элементы диагонали, которая параллельна главной диагонали и имеет максимальную сумму элементов.
8. Дана квадратная матрица. Определить, является ли она симметричной относительно главной диагонали.
9. Дана квадратная матрица. Определить, является ли она треугольной. (Треугольной является матрица, у которой выше (ниже) главной диагонали все элементы равны нулю.)

Литература

1. Крячков, А.В. Программирование на C и C++. Практикум [Текст] / А.В.Крячков, И.В.Сухинина, В.К.Томшин. – Москва: Горячая линия-Телеком, 2000. – 344 с.

2. Абрамов, С.А. Задачи по программированию [Текст] / С.А.Абрамов, Г.Г.Гнездилова, Е.Н.Капустина, М.И.Селюн. – Москва: Наука, 1988. – 224 с.

3. Пильщиков, В.Н. Сборник упражнений по языку Паскаль [Текст] / В.Н.Пильщиков. – Москва: Наука, 1989. – 160 с.

4. Павловская, Т.А. C/C++. Структурное программирование: Практикум [Текст] / Т.А. Павловская, Ю.А. Щупак. – СПб: Питер, 2002. – 240 с.

5. Шилдт, Г. Справочник программиста по C/C++ [Текст]: пер. с англ. / Герберт Шилдт. – Москва: Издательский дом "Вильямс", 2001. – 448 с.

6. Меньшиков, Ф.В. Олимпиадные задачи по программированию [Текст] / Ф.В.Меньшиков. – СПб: Питер, 2006. – 315 с.