

6. Массивы

Этот раздел является ключевым в изучении программирования на С. В нём описаны методы построения алгоритмов и программ с использованием статических и динамических массивов. В заключительном параграфе на примерах рассматривается совместное использование указателей, динамических массивов и функций пользователя при решении сложных задач обработки массивов.

6.1 Статические массивы в С

Часто для работы с множеством однотипных данных (целочисленными значениями, строками, датами и т.п.) оказывается удобным использовать массивы. Например, можно создать массив для хранения списка студентов, обучающихся в одной группе. Вместо создания переменных для каждого студента, например Студент1, Студент2 и т.д., достаточно создать один массив, где каждой фамилии из списка будет присвоен порядковый номер. Таким образом, можно дать следующее определение. *Массив* – структурированный тип данных, состоящий из фиксированного числа элементов одного типа.

Массив в табл. 1 имеет 8 элементов, каждый элемент сохраняет число вещественного типа. Элементы в массиве пронумерованы (нумерация массивов начинается с нуля). Такого рода массив, представляющий собой просто список данных одного и того же типа, называют простым или одномерным массивом. Для доступа к данным, хранящимся в определенном элементе массива, необходимо указать имя массива и порядковый номер этого элемента, называемый индексом.

Таблица 1: Одномерный числовой массив

№ элемента массива	0	1	2	3	4	5	6	7
Значение	13.65	-0.95	16.78	8.09	-11.76	9.07	5.13	-25.64

Если возникает необходимость хранения данных в виде матриц, в формате строк и столбцов, то необходимо использовать двумерные массивы. В табл. 2 приведен пример массива, состоящего из четырех строк и пяти столбцов. Это двумерный массив. Строки в нем можно считать первым измерением, а столбцы вторым. Для доступа к данным, хранящимся в этом массиве, необходимо указать имя массива и два индекса, первый должен соответствовать номеру строки, а второй номеру столбца в которых хранится необходимый элемент.

Таблица 2: Двумерный числовой массив

1.5	-0.9	1.8	7.09	-1.76
3.6	0.5	6.7	0.09	-1.33
13.65	-0.95	16.78	8.09	-11.76
7.5	0.95	7.3	8.9	0.11

Если при описании массивов определен его размер, то массивы называются статическими, рассмотрим работу с одномерными статическими массивами в языке С. Двумерные массивы подробно описаны в следующем

разделе.

6.1.1 Описание статических массивов

Описать статический массив в С можно так:

тип имя_переменной [размерность];

размерность – количество элементов в массиве. Например:

```
int x[10]; //Описание массива из 10 целых чисел. Первый
// элемент массива имеет индекс 0, последний 9.
```

```
float a[20]; //Описание массива из 20 вещественных
чисел.
```

```
// Первый элемент массива имеет индекс 0, последний 19.
```

Размерность массива и тип его элементов определяют объем памяти, который необходим для хранения массива. Рассмотрим ещё один пример описания массива:

```
const int n=15; //Определена целая положительная константа.
double B[n]; //Описан массив из 15 вещественных чисел.
```

При описании статического массива в качестве размерности можно использовать целое положительное число или предопределённую константу.

Элементы массива в С нумеруются с нуля. Первый элемент, всегда имеет номер ноль, а номер последнего элемента на единицу меньше заданной при его описании размерности:

```
char C[5]; //Описан массив из 5 символов,
//нумерация от 0 до 4.
```

6.1.2 Основные операции над массивами

Доступ к каждому элементу массива осуществляется с помощью индекса - порядкового номера элемента. Для обращения к элементу массива указывают его имя, а затем в квадратных скобках индекс:

имя_массива [индекс]

Например:

```
const int n=15;
double C[n], S;
//Сумма первого и последнего элементов массива C.
S=C[0]+C[n-1];
```

Массиву, как и любой другой переменной, можно присвоить начальное значение (инициализировать). Для этого значения элементов массива нужно перечислить в фигурных скобках через запятую:

тип имя_переменной[размерность]={элемент_0, элемент_1, ...};

Например:

```
float a[6]={1.2, (float)3/4, 5./6, 6.1};
//Формируется массив из шести вещественных чисел, значения
//элементам присваиваются по порядку, элементы значения,
// которых не указаны (в данном случае a[4], a[5])
// обнуляются:
//a[0]=1.2, a[1]=(float)3/4, a[2]=5./6, a[3]=6.1, a[4]=0,
// a[5]=0, для элементов a[1] и a[2] выполняется
// преобразование типов.
```

Рассмотрим, как хранится массив в памяти на примере массива

```
double x[30];
```

В памяти компьютере выделяется место для хранения 30 элементов типа `double`. При этом адрес этого участка памяти хранится в переменной `x`. Таким образом получается, что к элементу массива с индексом 0. можно обратиться двумя способами:

1. В соответствии с синтаксисом языка C можно записать `x[0]`.
2. Адрес начала массива хранится в переменной `x` (по существу `x` – указатель на `double`), поэтому с помощью операции `*x` мы можем обратиться к значению нулевого элемента массива.

Следовательно, `x[0]` и `*x` являются обращением к нулевому элементу массива.

Если к значению `x` добавить единицу (число 1), то мы сместимся на один элемент типа `double`. Таким образом, `x+1` – адрес элемента массива `x` с индексом 1. К первому элементу массива `x` также можно обратиться двумя способами `x[1]` или `*(x+1)`. Аналогично, к элементу с индексом 2 можно обращаться либо `x[2]`, либо `*(x+2)`. Таким образом, получается, что к элементу с индексом `i` можно обращаться `x[i]` или `*(x+i)`.

При этом при обработке массива (независимо от способа обращения `x[i]` или `*(x+i)`) программист сам должен контролировать существует ли элемент массива `x[i]` (или `*(x+i)`) и не вышла ли программа за границы массива.

Особенностью статических массивов является определение размера статическим образом при написании текста программы. При необходимости увеличить размер массив необходимо изменить текст программы и перекомпилировать её. Для динамического выделения памяти для массивов в C можно указатели и использовать операторы (функции) выделения памяти.

6.2 Динамические массивы в C

Для создания динамического массива необходимо:

- описать указатель (тип `* указатель;`);
- определить размер массива;
- выделить участок памяти для хранения массива и присвоить указателю адрес этого участка памяти.

Для выделения памяти можно воспользоваться функциями языка C – `calloc`, `malloc`, `realloc`. Все функции находятся в библиотеке `stdlib.h`.

6.2.1 Функция `malloc`

Функция `malloc` выделяет непрерывный участок памяти размером `size` байт и возвращает указатель на первый байт этого участка. Обращение к функции имеет вид:

```
void *malloc(size_t size);
```

где `size` – целое беззнаковое значение¹, определяющее размер выделяемого участка памяти в байтах. Если резервирование памяти прошло успешно, то функция возвращает переменную типа `void *`, которую можно пре-

¹ `size_t` – базовый беззнаковый целочисленный тип языка C, который выбирается таким образом, чтобы в него можно было записать максимальный размер теоретически возможного массива любого типа. В 32-битной операционной системе `size_t` является беззнаковым 32-битным числом (максимальное значение $2^{32}-1$), в 64-битной – 64-битным беззнаковым числом (максимальное значение $2^{64}-1$).

образовать к любому необходимому типу указателя. Если выделить память невозможно, то функция вернёт пустой указатель `NULL`.

Например,

```
double *h; //Описываем указатель на double.
int k;
scanf("%d", &k) //Ввод целого числа k.
//Выделение участка памяти для хранения k элементов типа
//double. Адрес этого участка хранится в переменной h.
h=(double *) malloc(k*sizeof(double));
//h - адрес начала участка памяти, h+1, h+2, h+3 и т. д. -
// адреса последующих элементов типа double.
```

6.2.2 Функция `calloc`

Функция `calloc` предназначена для выделения и обнуления памяти.

```
void *calloc (size_t num, size_t size);
```

С помощью функции будет выделен участок памяти, в котором будет храниться `num` элементов по `size` байт каждый. Все элементы выделенного участка обнуляются. Функция возвращает указатель на выделенный участок или `NULL` при невозможности выделить память.

Например,

```
float *h; //Описываем указатель на float.
int k;
scanf("%d", &k) //Ввод целого числа k.
//Выделение участка памяти для хранения k элементов типа
//float. Адрес этого участка хранится в переменной h.
h=(float *) calloc(k, sizeof(float));
//h - адрес начала участка памяти, h+1, h+2, h+3 и т. д. -
// адреса последующих элементов типа float.
```

6.2.3 Функция `realloc`

Функция `realloc` изменяет размер ранее выделенного участка памяти.

Обращаются к функции так:

```
void *realloc(void *p, size_t size);
```

где `p` – указатель на область памяти, размер которой нужно изменить на `size`. Если в результате работы функции меняется адрес области памяти, то новый адрес вернется в качестве результата. Если фактическое значение первого параметра `NULL`, то функция `realloc` работает, так же как и функции `malloc`, то есть выделяет участок памяти размером `size` байт.

6.2.4 Функция `free`

Для освобождения выделенной памяти используется функция `free`.

Обращаются к ней так:

```
void free(void *p);
```

где `p` – указатель на участок памяти, ранее выделенный функциями `malloc`, `calloc` или `realloc`.

6.2.5 Операторы new и delete

В языке C++ есть операторы new для выделения и delete для освобождения участка памяти.

Для выделения памяти для хранения *n* элементов одного типа оператор new имеет вид [5]:

```
x=new type [ n ] ;
```

type – тип элементов, для которых выделяется участок памяти;

n – количество элементов;

x – указатель на тип данных type, в котором будет храниться адрес выделенного участка памяти.

При выделении памяти для одного элемента оператор new имеет вид:

```
x=new type;
```

Например,

```
float *x ; //Указатель на тип данных f l o a t .
```

```
int n ;
```

```
cin>>n ;
```

```
//Ввод n
```

```
x=new float[n];
```

```
//Выделение участка памяти для хранения n элементов типа
```

```
// float. Адрес этого участка хранится
```

```
//в переменной x; x+1, x+2, x+3 и т. д. – адреса
```

```
//последующих элементов типа float.
```

Освобождение выделенного с помощью new участка памяти осуществляется с помощью оператора delete следующей структуры:

```
delete [] p;
```

p – указатель (адрес участка памяти, ранее выделенного с помощью оператора new).

6.3 Отличие статического и динамического массива

В чём же отличие статического и динамического массива?

Если у нас есть статический массив, например.

```
double x[75];
```

Мы имеем участок памяти для хранения 75 элементов типа double (массив из 75 элементов типа double). Адрес начала массива хранится в переменной x. Для обращения к i-му элементу можно использовать конструкции x[i] или *(x+i). Если понадобится обрабатывать массив более, чем из 75 элементов, то придётся изменить описание и перекомпилировать программу. При работе с массивами небольшой размерности, большая часть памяти, выделенная под статический массив будет использоваться вхолостую.

Если имеется динамический массив, например

```
double *x;//Указатель на double
```

```
int k;
```

```
//Вводим размер массива k.
```

```
printf("\n k="); scanf("%d",&k);
```

```
//Выделение памяти для хранения динамического массива из k
```

```
//чисел. Адрес начала массива хранится в переменной x.
```

```
x=(double *)calloc(k,sizeof(double));
```

```
// Память можно будет выделить и так
//x=(double *) malloc(k*sizeof(float));
```

Мы имеем указатель на тип данных `double`, вводим `k` – размер динамического массива, выделяем участок памяти для хранения `k` элементов типа `double` (массив из `k` элементов типа `double`). Адрес начала массива хранится в переменной `x`. Для обращения к `i`-му элементу можно использовать конструкции `x[i]` или `*(x+i)`. В случае динамического массива мы с начала определяем его размер (в простейшем случае просто вводим размер массива с клавиатуры), а потом выделяем память для хранения реального количества элементов. Таким образом основное отличие статического и динамического массивов состоит в том, что в динамическом массиве выделяется столько элементов, сколько необходимо.

Таким образом, при использовании как статического, так и динамического массива, имя массива – адреса начала участка памяти, обращаться к элементам массива можно двумя способами – `x[i]`, `*(x+i)`.

6.4 Основные алгоритмы обработки массивов

Все манипуляции с массивами в С осуществляются поэлементно. Организовывается цикл, в котором происходит последовательное обращение к нулевому, первому, второму и т.д. элементам массива. В общем виде алгоритм обработки массива выглядит, так как показано на рис. 1.

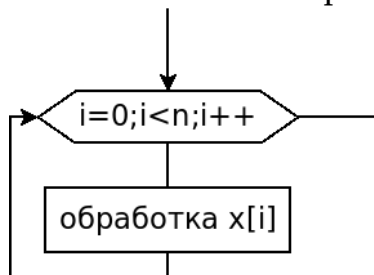


Рис. 1. Алгоритм обработки элементов массива

Алгоритмы, с помощью которых обрабатывают одномерные массивы, похожи на обработку последовательностей (вычисление суммы, произведения, поиск элементов по определенному признаку, выборки и т. д.). Отличие заключается в том, что в массиве одновременно доступны все его компоненты, поэтому становится возможной, например, сортировка его элементов и другие, более сложные преобразования.

6.4.1 Ввод-вывод элементов массива

Ввод и вывод массивов также осуществляется поэлементно. Блок-схемы алгоритмов ввода и вывода элементов массива `X[N]` изображены на рис. 2-3.

Рассмотрим несколько вариантов ввода массива:

```
//Вариант 1. Ввод массива с помощью функции scanf.
//При организации ввода используются специальные символы:
//табуляция – \t и переход на новую строку – \n.
#include <stdio.h>
int main()
```

```

{
float x[10]; int i,n;
//Ввод размерности массива.
printf("\n N="); scanf("%d",&n);
printf("\n Введите элементы массива X \n");
for(i=0;i<n;i++)
//Ввод элементов массива в цикле.
//Обратите внимание!!! Использование x+i.
scanf("%f",x+i);
return 0;
}

```

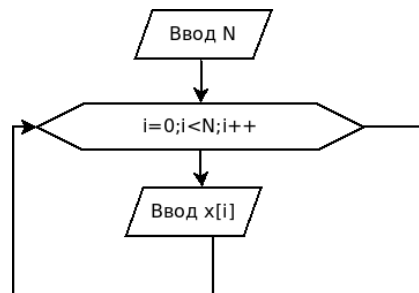


Рис. 2. Алгоритм ввода массива $X[N]$

Результат работы программы:

N=3

Введите элементы массива X

1.2

-3.8

0.49

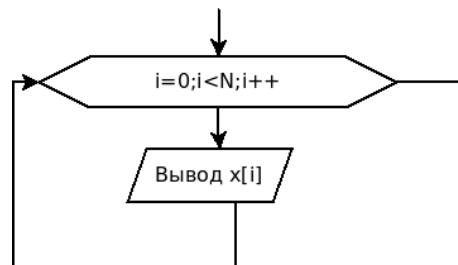


Рис. 3. Алгоритм вывода массива $X[N]$

```

//-----
//Вариант 2. Ввод массива с помощью функции scanf
//и вспомогательной переменной b.
#include <stdio.h>
int main()
{
float x[10],b; int i,n;
//Ввод размерности массива.
printf("\n N="); scanf("%d",&n);
printf("\n Массив X \n");
for(i=0;i<n;i++)
{
//Сообщение о вводе элемента.
printf("\n Элемент %d \t",i);

```

```
scanf("%f",&b);          //Ввод переменной b.
//Присваивание элементу массива значения переменной b.
x[i]=b;
}
return 0;}
```

Результат работы программы:

N=4

Массив X

Элемент 0	8.7
Элемент 1	0.74
Элемент 2	-9
Элемент 3	78

//-----

Вывод статического или динамического массива можно осуществить несколькими способами:

```
//Вариант 1. Вывод массива в виде строки.
for(i=0;i<n;i++) printf("%f \t",X[i]); printf("\n");
//Вариант 2. Вывод массива в виде столбца.
for(i=0;i<n;i++) printf("\n %f ",X[i]);
```

6.4.2 Вычисление суммы элементов массива

Дан массив X, состоящий из N элементов. Найти сумму элементов этого массива. Процесс накапливания суммы элементов массива достаточно прост и практически ничем не отличается от суммирования значений некоторой числовой последовательности. Переменной S присваивается значение равное нулю, затем к переменной S последовательно добавляются элементы массива X. Блок-схема алгоритма расчёта суммы приведена на рис. 4.

Соответствующий алгоритму фрагмент программы будет иметь вид:

```
for (S=i=0;i<N;i++)
    S+=X[i];
cout<<"S="<<S<<"\n";
```

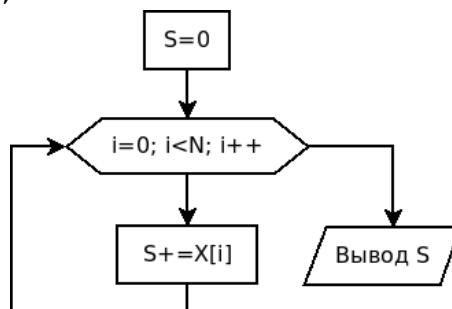


Рис. 4. Алгоритм вычисления суммы элементов массива

6.4.3 Вычисление произведения элементов массива

Дан массив X, состоящий из N элементов. Найти произведение элементов этого массива. Решение этой задачи сводится к тому, что значение переменной P, в которую предварительно была записана единица, последовательно

умножается на значение i -го элемента массива. Блок-схема алгоритма приведена на рис. 5.

Соответствующий фрагмент программы будет иметь вид:

```
for (P=1, i=0; i<N; i++)
    P*=X[i];
cout<<"P="<<P<<"\n";
```

ЗАДАЧА 1. Задан массив целых чисел. Найти сумму простых чисел и произведение отрицательных элементов массива.

Алгоритм решения задачи состоит из следующих этапов.

1. Вводим массив $X[N]$.
2. Для вычисления суммы в переменную S записываем значение 0, для вычисления произведения в переменную P записываем 1.
3. В цикле (i изменяется от 0 до $N-1$ с шагом 1) перебираем все элементы массива X , если очередной элемент массива является простым числом, добавляем его к сумме, а если очередной элемент массива отрицателен, то умножаем его на P .
4. Выводим на экран значение суммы и произведения.

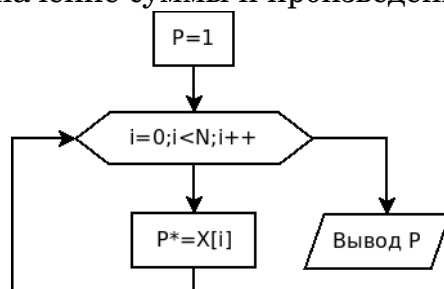


Рис. 5. Вычисление произведения элементов массива

Блок-схема решения задачи представлена на рис. 6. Для решения задачи применим функцию (prostoe) проверки является ли число простым.

Текст программы с подробными комментариями приведён далее.

```
#include <stdio.h>
#include <stdlib.h>
//Текст функции prostoe.
int prostoe (int N)
{
    int i;
    int pr;
    if (N<2) pr=0;
    else
    for(pr=1, i=2; i<=N/2; i++)
        if (N%i==0)
        {
            pr=0;
            break;
        }
    return pr;
}
int main()
{
```

```

    int *X,i,N,S,P;
    //Ввод размерности массива.
    printf("Введите размер массива ");scanf("%d",&N);
    //Выделение памяти для хранения динамического массива X.

    X=(int *)calloc(N, sizeof(int));
    //Ввод массива X.
    printf("Введите массив X\n");
    for(i=0;i<N;i++)
    {printf("X(%d)=",i);scanf("%d",X+i);}
    //В цикле перебираем все элементы массива

```

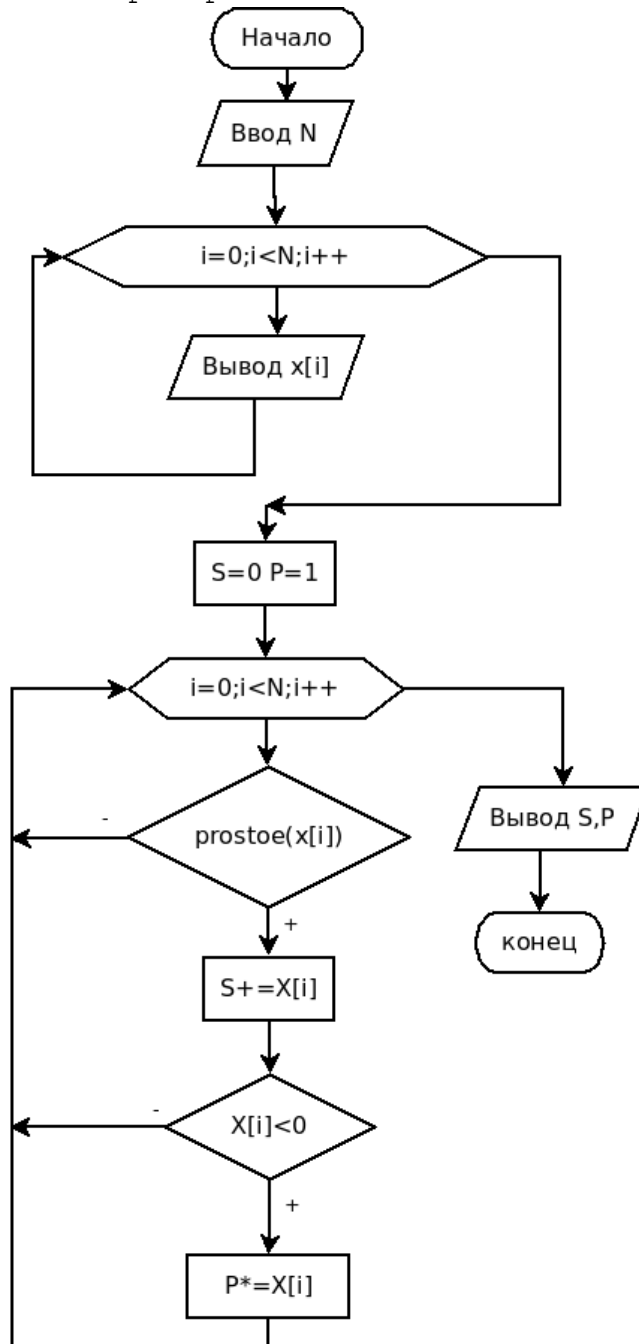


Рис. 6. Блок-схема алгоритма решения задачи 1

```

for (P=1, S=i=0; i<N; i++)
{

```

```

//Если очередной элемент массива - простое число,
// добавляем его к сумме.
    if (prostoe(X[i])) S+=X[i];
//Если очередной элемент массива отрицателен, умножаем его
// на P.
    if (X[i]<0) P*=X[i];
}
//Вывод S и P на экран.
printf("S=%d\tP=%d\n",S,P);
// Освобождение занимаемой массивом X памяти.
free(X);
return 0;
}

```

Результаты работы программы представлены ниже.

Введите размер массива 10

Ведите массив X

X(0)=-7

X(1)=-9

X(2)=5

X(3)=7

X(4)=2

X(5)=4

X(6)=6

X(7)=8

X(8)=10

X(9)=12

S=14 P=63

6.4.4 Поиск максимального элемента в массиве и его номера

Дан массив X, состоящий из n элементов. Найти максимальный элемент массива и номер, под которым он хранится в массиве.

Алгоритм решения задачи следующий. Пусть в переменной с именем Max хранится значение максимального элемента массива, а в переменной с именем Nmax – его номер. Предположим, что нулевой элемент массива является максимальным и запишем его в переменную Max, а в Nmax – его номер (то есть ноль). Затем все элементы, начиная с первого, сравниваем в цикле с максимальным. Если текущий элемент массива оказывается больше максимального, то записываем его в переменную Max, а в переменную Nmax – текущее значение индекса i. Процесс определения максимального элемента в массиве приведен в таблице 3 и изображен при помощи блок-схемы на рис. 7.

Соответствующий фрагмент программы имеет вид:

```

for (Max=X[0],Nmax=0,i=1;i<n;i++)
if (Max<X[i])
{
Max=X[i];
Nmax=i;
}
cout<<"Max="<<Max<<"\n";
cout<<"Nmax="<<Nmax<<"\n";

```

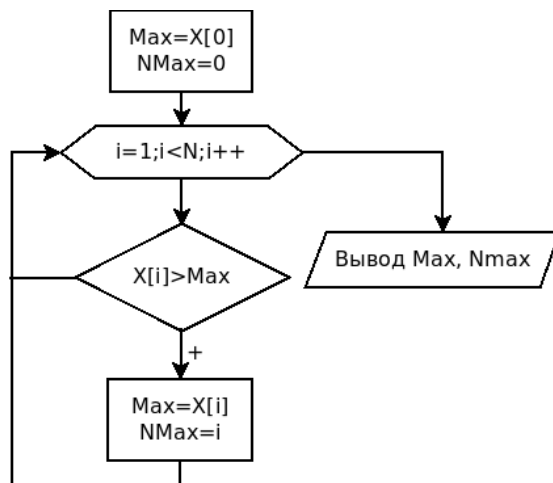


Рис. 7. Поиск максимального элемента и его номера в массиве

Таблица 3: Определение максимального элемента и его номера в массиве

Номера элементов	0	1	2	3	4	5
Исходный массив	4	7	3	8	9	2
Значение переменной Max	4	7	7	8	9	9
Значение переменной Nmax	1	2	2	4	5	5

Текст программы поиска номера максимального элемента:

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    float *X,max;
    int i,N,nmax;
    //Ввод размерности динамического массива
    printf("Введите размер массива ");scanf("%d",&N);
    //Выделения памяти для хранения динамического массива X.
    X=(float *)calloc(N,sizeof(float));
    //Ввод динамического массива X.
    printf("Введите элементы массива X\n");
    for(i=0;i<N;i++)
        scanf("%f",X+i);
    //В переменной nmax будем хранить номер максимального
    // элемента (значение 0). Само значение максимального
    //элемента будем хранить в переменной max.
    for(max=X[0],nmax=0,i=1;i<N;i++)
    //Если очередной элемент X[i] больше максимального,
    // значит max не является максимальным
    //поэтому переписываем x[i] в переменную max,
    //а значение i в переменную nmax.
        if (X[i]>max) {max=X[i]; nmax=i;}
    printf("Максимальный элемент=%6.2f, его номер=%d\n",
    max,nmax);
    free(X);
    return 0;
}
  
```

При поиске максимального элемента и его номера, можно найти номер максимального элемента, а потом по номеру извлечь значение максимального элемента из массива. Текст такой версии программы с комментариями приведен ниже.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float *X;
    int i,N,nom;
    //Ввод размерности динамического массива
    printf("Введите размер массива ");scanf("%d",&N);
    //Выделения памяти для хранения динамического массива X.
    X=(float *)calloc(N,sizeof(float));
    //Ввод динамического массива X.
    printf("Введите элементы массива X\n");
    for(i=0;i<N;i++)
        scanf("%f",X+i);
    //В переменной nom будем хранить номер максимального
    // элемента. Предположим, что максимальным элементом,
    //является элемент с номером 0.
    nom=0;
    for(i=1;i<N;i++)
    //Если очередной элемент больше X[nom], значит nom не
    // является номером максимального элемента, элемент с
    //номером i больше элемента X[nom], поэтому переписываем
    //число i в переменную nom.
        if (X[i]>X[nom]) nom=i;
    printf("Максимальный элемент=%6.2f, его номер=%d\n",
    X[nom],nom);
    free(X);
    return 0;
}
```

Совет. Алгоритм поиска минимального элемента в массиве будет отличаться от приведенного выше лишь тем, что в условном блоке и, соответственно, в конструкции if текста программы знак поменяется с < на >.

Рассмотрим несколько задач.

ЗАДАЧА 2. Найти минимальное простое число в целочисленном массиве $x[N]$.

Эта задача относится к классу задач поиска минимума (максимума) среди элементов, удовлетворяющих условию. Подобные задачи рассматривались в задачах на обработку последовательности чисел. Здесь поступим аналогично. Блок-схема приведена на рис. 8.

Необходимо первое простое число объявить минимумом, а все последующие простые элементы массива сравнивать с минимумом. Будем в цикле обрабатывать последовательно проверять, является ли элемент массива простым числом (функция `prostoe`). Если $X[i]$ является простым числом, то количество простых чисел (k) увеличиваем на 1 ($k++$), далее, проверяем, если k равен 1 (`if (k==1)`), то этот элемент объявляем минимальным (`min=x[i]`;

nom=i;), иначе сравниваем его с минимальным (if (x[i]<min) {min=x[i];nom=i;}).

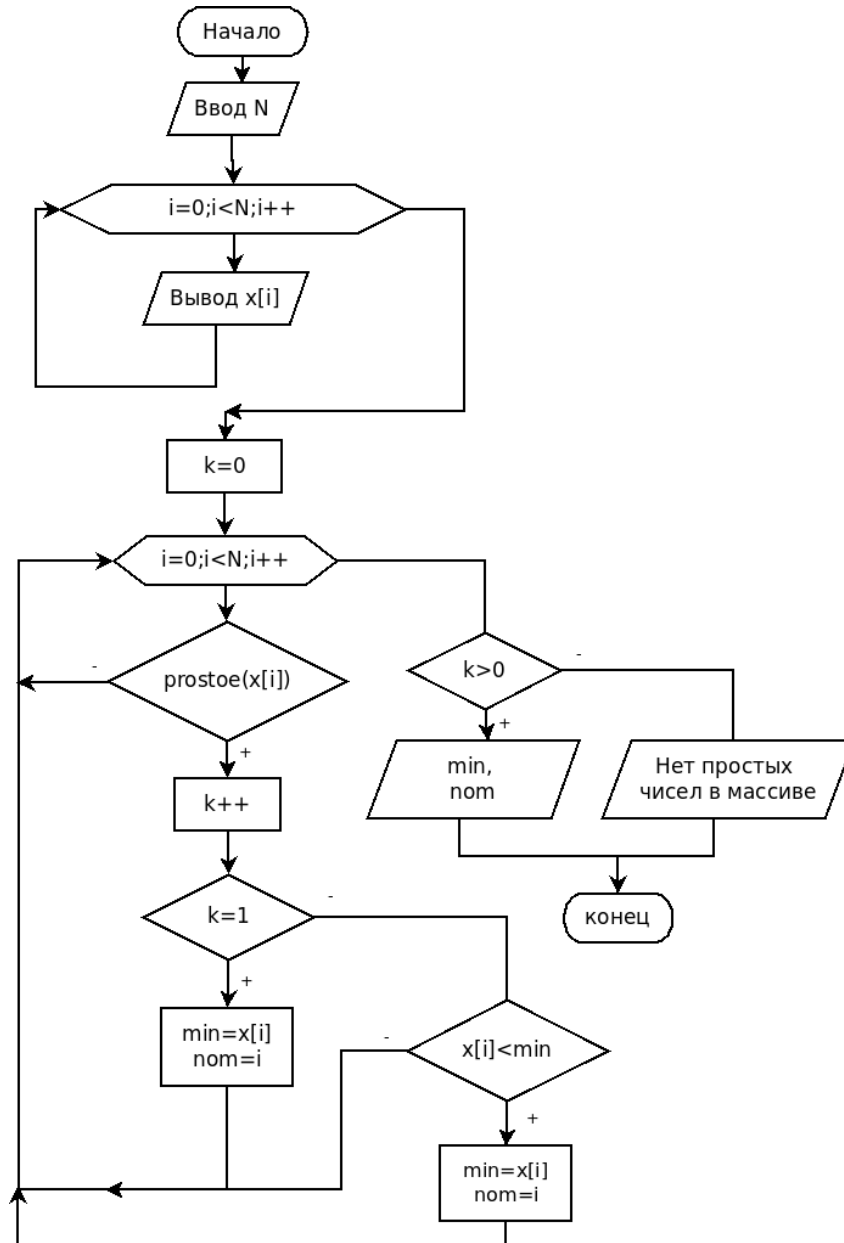


Рис. 8. Блок-схема решения задачи 2

Текст программы:

```

#include <stdio.h>
#include <stdlib.h>
int prostoe (int N)
{ int i; int pr;
  if (N<2) pr=0;
  else
    for(pr=1,i=2;i<=N/2;i++)
      if (N%i==0)
        {pr=0; break;}
    return pr;}
int main(int argc, char **argv)
{

```

```

    int i,k,n,nom,min,*x;
    //Ввод количества элементов в массиве.
    printf("n="); scanf("%d",&n);
    //Выделяем память для динамического массива x.
    x=(int *)calloc(n, sizeof(int));
    //Ввод элементов массива.
    printf("Введите элементы массива X");
    for(i=0;i<n;i++)
        scanf("%d",x+i);
    //С помощью цикла по переменной i, перебираем все элементы
    // в массиве x, k - количество простых чисел в массиве.
    for(i=k=0;i<n;i++)
    //Проверяем, является ли очередной элемент массива
    // простым числом. Если x[i] - простое число.
        if (prostoe(x[i]))
        {
    //Увеличиваем счётчик количества простых чисел в массиве.
            k++;
    //Если текущий элемент является первым простым числом в
    // массиве, объявляем его минимумом, а его номер сохраняем
    // в переменной nom.
            if (k==1) {min=x[i];nom=i;}
            else
    //Все последующие простые числа в массиве сравниваем с
    // минимальным простым числом. Если текущее число меньше
    // min, перезаписываем его
    //в переменную min, а его номер - в переменную nom.
            if (x[i]<min) {min=x[i];nom=i;}
        }
    //Если в массиве были простые числа, выводим значение и
    // номер минимального простого числа.
    if (k>0)
        printf("min=%d\tnom=%d\n",min,nom);
    //Иначе выводим сообщение о том, что в массиве нет простых
    // чисел.
    else printf("Нет простых чисел в массиве\n");
    free(x);
    return 0;
}

```

Аналогичным образом можно написать программу любой задачи поиска минимума (максимума) среди элементов, удовлетворяющих какому-либо условию (минимум среди положительных элементов, среди чётных и т.д.).

ЗАДАЧА 3. Найти k минимальных чисел в вещественном массиве.

Перед решением этой довольно сложной задачи рассмотрим более простую задачу.

Найти два наименьших элемента в массиве X . Фактически надо найти номера (n_{min1}, n_{min2}) двух наименьших элементов массива.

Алгоритм состоит в следующем.

Будут храниться индексы двух наименьших элементов массива $X[n_{min1}] < X[n_{min2}]$. На первом этапе сравниваем $X[0]$ и $X[1]$ заполняем

```

nmin1 и nmin2.
    if (X[0]<X[1])
    { nmin1=0;nmin2=1;}
    else
    { nmin1=1;nmin2=0;}

```

Далее, все элементы, начиная с индекса 2 сравниваем с X[nmin1] и X[nmin2]. Возможны три соотношения между X[i], X[nmin1] и X[nmin2].

1. X[i]>X[nmin2]>X[nmin1], в этом случае значение индексов nmin1, nmin2 не изменяется.
2. X[nmin2]>X[i]>X[nmin1], в этом случае необходимо заменить значение переменной nmin1 на i (nmin1=i).
3. X[nmin2]>X[nmin1]>X[i], в этом случае в nmin1 надо записать в nmin1 значение i, а в nmin2 значение nmin1 (nmin2=nmin1;nmin1=i).

Перебрав значения всех элементов от 2 до последнего, в переменных nmin1 и nmin2 будем хранить индексы двух наименьших элементов массива (X[nmin1]<X[nmin2]).

Текст программы с комментариями приведён ниже.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int i,n,nmin1,nmin2;
    double *X;
    printf("n="); scanf("%d",&n);
    X=(double *)calloc(n,sizeof(double));
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%lf",X+i);
    //Определяем первоначальные значения nmin1 и nmin2, исходя
    //из того, X[nmin1]<X[nmin2].
    if (X[0]<X[1])
    {
        nmin1=0;
        nmin2=1;
    }
    else
    {
        nmin1=1;
        nmin2=0;
    }
    for(i=2;i<n;i++)
    // Если X[nmin2]>X[i]>X[nmin1], необходимо изменить
    // значение переменной nmin1 на i (nmin1=i).
        if ((X[nmin2]>X[i])&& (X[i]>X[nmin1])) nmin1=i;
        else
    // Если X[nmin2]>X[nmin1]>X[i], в nmin1 надо записать
    // значение i, а в nmin2 значение nmin1.
        if ((X[nmin2]>X[nmin1]) && (X[nmin1]>X[i]))
        {nmin2=nmin1;nmin1=i;}
    //Вывод двух минимальных элементов и их индексов.

```



```
printf("nmin1=%d\tmin1=%lf\n",nmin1,X[nmin1]);
printf("nmin2=%d\tmin2=%lf\n",nmin2,X[nmin2]);
free(X);
return 0;}
```

Рассмотрим вторую версию программы. Этот алгоритм осуществляет поиск двух минимальных за два прохода цикла, однако этот алгоритм может быть обобщён для поиска k минимальных элементов массивов.

На первом этапе надо найти номер минимального ($nmin1$) элемента массива. На втором этапе надо искать минимальный элемент, при условии, что его номер не равен $nmin1$. Вторая часть очень похожа на предыдущую задачу (минимум среди элементов, удовлетворяющих условию, в этом случае условие имеет вид $i \neq nmin$).

Решение задачи с комментариями:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int kvo,i,n,nmin1,nmin2;
    double *X;
    printf("n="); scanf("%d",&n);
    X=(double *)calloc(n,sizeof(double));
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%lf",X+i);
    //Стандартный алгоритм поиска номера первого минимального
    //элемента (nmin1).
    for(nmin1=0,i=1;i<n;i++)
        if (X[i]<X[nmin1]) nmin1=i;
    //Второй этап - поиск номера минимального элемента,
    //среди элементов, номер которых не совпадает nmin1.
    //kvo - количество таких элементов.
    for(kvo=i=0;i<n;i++)
    //Если номер текущего элемента не совпадает с nmin1,
        if (i!=nmin1)
        {
            //увеличиваем количество таких элементов на 1.
            kvo++;
            //Номер первого элемента, индекс которого не равен nmin1,
            //объявляем номером второго минимального элемента.
            if (kvo==1) nmin2=i;
            else
            //очередной элемент индекс которого не равен nmin1
            //сравниваем с минимальным, если он меньше, номер
            // перезаписываем в переменную nmin2.
                if (X[i]<X[nmin2]) nmin2=i;}
    //Вывод двух минимальных элементов и их индексов.
    printf("nmin1=%d\tmin1=%lf\n",nmin1,X[nmin1]);
    printf("nmin2=%d\tmin2=%lf\n",nmin2,X[nmin2]);
    free(X);
    return 0;}
```

```
}

```

По образу и подобию этой задачи можно написать задачу поиска трёх минимальных элементов в массиве. Первые два этапа (поиск номеров двух минимальных элементов в массиве) будут полным повторением кода, приведённого выше. На третьем этапе нужен цикл, в котором будем искать номер минимального элемента, при условии, что его номер не равен `nmin1` и `nmin2`. Авторы настоятельно рекомендуют читателям самостоятельно написать подобную программу. Аналогично можно написать программу поиска четырёх минимальных элементов. Однако при этом усложняется и увеличивается код программы. К тому же, рассмотренный приём не позволит решить задачу в общем случае (найти k минимумов).

Для поиска k минимумов в массиве можно поступить следующим образом. Будем формировать массив `nmin`, в котором будут храниться номера минимальных элементов массива `x`. Для его формирования организуем цикл по переменной `j` от 0 до $k-1$. При каждом вхождении в цикл в массиве `nmin` элементов будет $j-1$ элементов и мы будем искать j -й минимум (формировать j -й элемент массива). Алгоритм формирования j -го элемента состоит в следующем: необходимо найти номер минимального элемента в массиве `x`, исключая номера, которые уже хранятся в массиве `nmin`. Внутри цикла по `j` необходимо выполнить такие действия. Для каждого элемента массива `x` (цикл по переменной `i`) проверить содержится ли номер в массиве `nmin`, если не содержится, то количество (переменная `kvo`) таких элементов увеличить на 1. Далее, если `kvo` равно 1, то это первый элемент, который не содержится в массиве `nmin`, его номер объявляем номером минимального элемента массива (`nmin_temp=i`;). Если `kvo>1`, сравниваем текущий элемент `x[i]` с минимальным (`if (x[i]<X[nmin_temp]) nmin_temp=i`;). Блок-схема алгоритма поиска k минимальных элементов массива представлена на рис. 9². Далее приведен текст программы с комментариями.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int p,j,i,n,*nmin,k,kvo,nmin_temp;
    int pr;
    double *x;
    printf("n="); scanf("%d",&n);
    x=(double *)calloc(n,sizeof(double));
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%lf",x+i);
    printf("Введите количество минимумов\n");
    scanf("%d",&k);
    nmin=(int *)calloc(k,sizeof(int));
    //Цикл по переменной j для поиска номера j+1 минимального
    //элемента

```

2 В блок-схеме отсутствует ввод данных и вывод результатов.

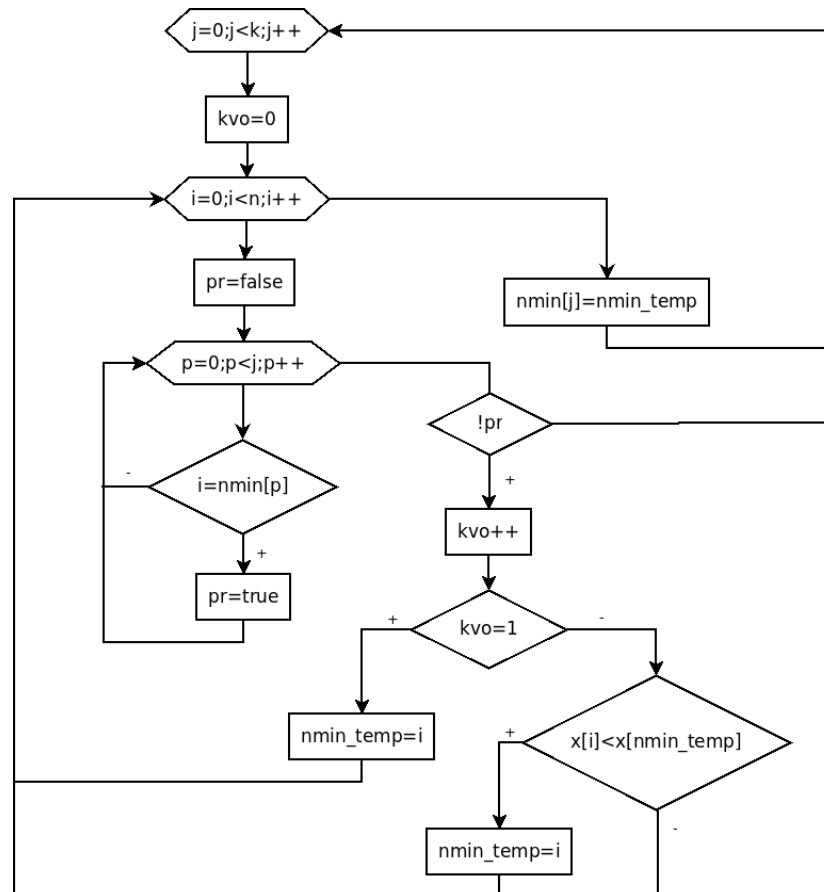


Рис. 9. Блок-схема алгоритма поиска k минимальных элементов в массиве x .

```

for (j=0; j<k; j++)
{
    kvo=0;
    //Перебираем все элементы массива.
    for (i=0; i<n; i++)
    {
        //Цикл по переменной p проверяет содержится ли номер i в
        // массиве nmin.
        pr=0;
        for (p=0; p<j; p++)
            if (i==nmin[p]) pr=1;
        //Если не содержится, то количество элементов увеличить на 1.
        if (!pr)
        {
            kvo++;
            //Если kvo=1, то найден первый элемент, который не
            // содержится в массиве nmin, его номер объявляем номером
            // минимального элемента массива
            if (kvo==1) nmin_temp=i;
            else
            //Если kvo>1, сравниваем текущий элемент x[i] с
            // минимальным.
            if (x[i]<x[nmin_temp]) nmin_temp=i;
        }
    }
}
  
```

```

//Номер очередного минимального элемента записываем в
// массив nmin.
    nmin[j]=nmin_temp;
}
//Вывод номеров и значений k минимальных элементов
// массива.
    for(j=0;j<k;j++)
        printf("nmin1=%d\tmin1=%lf\n",nmin[j],x[nmin[j]]);
    return 0;
}

```

Проверку содержится ли число *i* в массиве *nmin*, можно оформить в виде функции, тогда программа может быть записана следующим образом:

```

#include <stdio.h>
#include <stdlib.h>
using namespace std;
//Функция проверяет содержится ли число i в массиве x из n
// элементов. Функция возвращает true, если содержится
// false, если не содержится.
int proverka(int i, int *x, int n)
{
    bool pr;
    int p;

    pr=0;
    for(p=0;p<n;p++)
        if (i==x[p]) pr=1;
    return pr;
}
int main(int argc, char **argv)
{
    int j,i,n,*nmin,k,kvo,nmin_temp;

    double *x;
    printf("n="); scanf("%d",&n);
    x=(double *)calloc(n,sizeof(double));
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%lf",x+i);
    printf("Введите количество минимумов\n");
    scanf("%d",&k);
    nmin=(int *)calloc(k,sizeof(int));
    //Цикл по переменной j для поиска номера j+1 минимального
    // элемента
    for(j=0;j<k;j++)
    {
        kvo=0;
        //Перебираем все элементы массива.
        for(i=0;i<n;i++)
        {
            //Вызов функции proverka, чтобы проверить содержится ли
            // число i в массиве nmin из j элементов.

```

```

        if (!proverka(i, nmin, j))
        {
            kvo++;
            if (kvo==1) nmin_temp=i;
            else
                if (x[i]<x[nmin_temp])    nmin_temp=i;
        }
        nmin[j]=nmin_temp;
    }
    //Вывод номеров и значений k минимальных элементов
    // массива.
    for(j=0; j<k; j++)
        printf("nmin1=%d\tmin1=%lf\n", nmin[j], x[nmin[j]]);
    free(x);
    return 0;
}

```

Авторы настоятельно рекомендуют читателю разобрать все версии решения задачи 3.

ЗАДАЧА 4. Поменять местами максимальный и минимальный элементы в массиве X.

Алгоритм решения задачи можно разбить на следующие этапы.

1. Ввод массива.
2. Поиск номеров максимального (nmax) и минимального (nmin) элементов массива.
3. Обмен элементов местами. Не получится записать «в лоб» ($X[nmax]=X[nmin]$; $X[nmin]=X[nmax]$;). При таком присваивании мы сразу же теряем максимальный элемент. Поэтому нам понадобится временная (буферная) переменная temp. Обмен элементов местами должен быть таким.

```
temp=X[nmax]; X[nmax]=X[nmin]; X[nmin]=temp;
```

Далее приведён текст программы с комментариями.

```

#include <stdio.h>
int main(int argc, char **argv)
{
    int i, N, nmax, nmin;
    float temp;
    printf("N="); scanf("%d", &N);
    float X[N];
    printf("Введите элементы массива X\n");
    for(i=0; i<N; i++)
        scanf("%f", X+i);
    //Поиск номеров максимального и минимального элементов
    // массива.
    for(nmax=nmin=0, i=1; i<N; i++)
    {
        if (X[i]<X[nmin]) nmin=i;
        if (X[i]>X[nmax]) nmax=i;
    }
    //Обмен максимального и минимального элементов местами.

```

```

        temp=X[nmax];X[nmax]=X[nmin]; X[nmin]=temp;
//Вывод преобразованного массива.
        printf("Преобразованный массив X\n");
        for(i=0;i<N;i++)
            printf("%6.2f\t",X[i]);
        printf("\n");
        return 0;
    }

```

ЗАДАЧА 5. Найти среднее геометрическое среди простых чисел, расположенных между максимальным и минимальным элементами массива.

Среднее геометрическое k элементов (SG) можно вычислить по формуле $SG = \sqrt[k]{P}$, P – произведение k элементов. При решении этой задачи необходимо найти произведение и количество простых чисел, расположенных между максимальным и минимальным элементами.

Алгоритм решения задачи состоит из следующих этапов:

1. Ввод массива.
2. Поиск номеров максимального (nmax) и минимального (nmin) элементов массива.
3. В цикле перебираем все элементы массива, расположенные между максимальным и минимальным элементами. Если текущий элемент является простым числом, то необходимо увеличить количество простых чисел на 1, и умножить P на значение элемента массива.
4. Вычислить $SG = \sqrt[k]{P}$.

При решении этой задачи следуем учитывать, что неизвестно какой элемент расположен раньше максимальный или минимальный.

Текст программы с комментариями приведён ниже.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int prostoe (int N)
{
    int i;
    int pr;
    if (N<2) pr=0;
    else
        for(pr=1,i=2;i<=N/2;i++)
            if (N%i==0)
            {
                pr=0;
                break;
            }
    return pr;
}
int main(int argc, char **argv)
{
    int i,k,n,nmax,nmin, p, *x;
//Ввод количества элементов в массиве.
    printf("n="); scanf("%d",&n);

```

```

//Выделяем память для динамического массива x.
x=(int *)calloc(n,sizeof(int));
//Ввод элементов массива.
printf("Введите элементы массива x\n");
for(i=0;i<n;i++)
scanf("%d",x+i);
//Поиск номеров максимального и минимального элементов в
// массиве.
for(nmax=nmin=i=0;i<n;i++)
{
    if (x[i]<x[nmin]) nmin=i;
    if (x[i]>x[nmax]) nmax=i;
}
if (nmin<nmax)
for(p=1,k=0,i=nmin+1;i<nmax;i++)
//ОБРАТИТЕ ОСОБОЕ ВНИМАНИЕ НА ИСПОЛЬЗОВАНИЕ В СЛЕДУЮЩЕЙ
//СТРОКЕ ФИГУРНОЙ СКОБКИ (СОСТАВНОГО ОПЕРАТОРА) .
//В ЦИКЛЕ ВСЕГО ОДИН ОПЕРАТОР!!!!
//ПРИ ЭТОМ ПРИ ОТСУТСТВИИ СОСТАВНОГО ОПЕРАТОРА ПРОГРАММА
// НАЧИНАЕТ СЧИТАТЬ С ОШИБКАМИ!!!!
{
//Проверяем, является ли очередной элемент массива простым
// числом.
//Если x[i] - простое число.
    if (prostoe(x[i]))
    {
//Домножаем y[i] на p, а также увеличиваем счётчик
//количества простых чисел в массиве.
        k++;p*=x[i];
    }
}
else
for(p=1,k=0,i=nmax+1;i<nmin;i++)
//Проверяем, является ли очередной элемент массива простым
// числом. Если x[i] - простое число.
    if (prostoe(x[i]))
    {
//Домножаем p на x[i], а также увеличиваем счётчик
// количества простых чисел в массиве.
        k++;p*=x[i];
    }
}
//Если в массиве были простые числа, выводим среднее
//геометрическое простых чисел на экран
    if (k>0)
        printf("SG=%8.3f\n",pow(p,1./k));
//Иначе выводим сообщение о том, что в массиве нет простых
// чисел.
    else printf("Нет простых чисел в массиве\n");
    free(x);
    return 0;
}

```

6.4.5 Удаление элемента из массива

Для удаления элемента с индексом m из массива X , состоящего из n элементов нужно записать $(m+1)$ -й элемент на место элемента m , $(m+2)$ -й – на место $(m+1)$ -го и т.д., $(n-1)$ -й – на место $(n-2)$ -го. После удаления количество элементов в массиве уменьшилось на 1 (рис. 10).

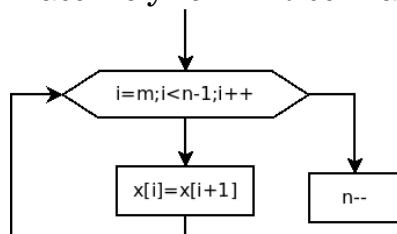


Рис. 10: Алгоритм удаления элемента из массива

Фрагмент программы удаления элемента номер m из массива вещественных чисел:

```

for (i=m; i<n-1; X[i]=X[i+1], i++); //Удаление m-го элемента.
n--;
//Изменение физического размера массива.
X=realloc(float *) (X, n*sizeof(float));
//Вывод измененного массива.
for (i=0; i<n-1; i++) cout<<X[i]<<"\t";

```

При написании программ, в которых удаляются элементы из массива следует учитывать тот факт, что после удаления элемента все элементы, расположенные после удалённого изменяют свои номера (индексы уменьшаются на один). Это особенно важно при удалении нескольких элементов из массива. Рассмотрим несколько задач.

ЗАДАЧА 6. Удалить из массива $x[20]$ все элементы с пятого по десятый.

При решении задач, связанных с удалением подряд идущих элементов, следует понимать, что после удаления очередного элемента следующий переместился на место удалённого. Поэтому далее нужно будет удалять элемент с тем же самым номером. В нашем случае подлежит удалению 6 элементов с пятого по десятый. Однако, реально надо будет 6 раз удалить элемент с номером 5. Блок-схема алгоритма представлена на рис 11, текст программе приведён далее.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int i, j, n=20;
    //Выделяем память для динамического массива x.
    float x[n];
    //Ввод элементов массива.
    printf("Введите элементы массива X\n");
    for(i=0; i<n; i++)
        scanf("%f", x+i);
    //Шесть раз повторяем алгоритм удаления элемента с

```



```

// индексом 5.
    for(j=1;j<=6;j++)
//Удаление элемента с индексом 5.
    for(i=5;i<n-j;i++)
        x[i]=x[i+1];
//Вывод элементов массива.
printf("Преобразованный массив X\n");
    for(i=0;i<n-6;i++)
        printf("%5.1f\t",x[i]);
    printf("\n");
    return 0;
}

```

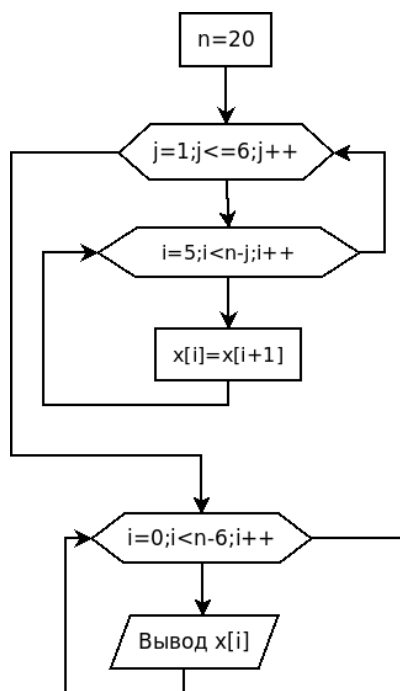


Рис. 11: Алгоритм решения задачи 6

ЗАДАЧА 7. Удалить из массива $X[n]$ все положительные элементы.

При удалении отдельных элементов из массива следует учитывать: при удалении элемента (сдвиге элементов влево и уменьшении n) не надо переходить к следующему, а если элемент не удалялся, то, наоборот, надо переходить к следующему.

Далее приведён текст программы решения задачи 7.

```

#include <stdio.h>
int main(int argc, char **argv)
{
    int i,j,n;
    printf("n="); scanf("%d",&n);
    //Выделяем память для динамического массива x.
    float x[n];
    //Ввод элементов массива.
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)

```

```

scanf("%f", x+i);
for(i=0; i<n; )
//Если текущий элемент положителен,
if (x[i]>0)
//то удаляем элемент с индексом i.
{
    for(j=i; j<n-1; j++)
        x[j]=x[j+1];
    n--;
}
//иначе - переходим к следующему элементу массива.
else i++;

//Вывод элементов массива.
printf("Преобразованный массив X\n");
for(i=0; i<n; i++)
    printf("%f\t", x[i]);
printf("\n");
return 0;
}

```

ЗАДАЧА 8. Удалить из массива все отрицательные элементы, расположенные между максимальным и минимальным элементами массива $X[n]$.

Решение этой задачи можно разделить на следующие этапы:

1. Ввод массива.
2. Поиск номеров максимального (n_{\max}) и минимального (n_{\min}) элементов массива.
3. Определение меньшего (a) и большего (b) из чисел n_{\max} и n_{\min} .
4. Далее, необходимо перебрать все элементы массива, расположенные между числами с номерами a и b . Если число окажется отрицательным, то его необходимо удалить. Однако, на этом этапе нужно учитывать тонкий момент. Если просто организовать цикл от $a+1$ до $b-1$, то при удалении элемента, изменяется количество элементов и номер последнего удаляемого элемента, расположенных между a и b . Это может привести к тому, что не всегда корректно будут удаляться отрицательные элементы, расположенные между a и b . Поэтому этот цикл для удаления организован несколько иначе.

Текст программы:

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int i, j, k, n, nmax, nmin, *x, a, b;
    //Ввод количества элементов в массиве.
    printf("n="); scanf("%d", &n);
    //Выделяем память для динамического массива x.
    x=(int *)calloc(n, sizeof(int));
    //Ввод элементов массива.
    printf("Введите элементы массива X\n");

```

```

    for(i=0;i<n;i++)
        scanf("%d",&x[i]);
//Поиск номеров максимального и минимального элементов в
// массиве.
    for(nmax=nmin=i=0;i<n;i++)
    {
        if (x[i]<x[nmin]) nmin=i;
        if (x[i]>x[nmax]) nmax=i;
    }
//Проверяем, что раньше расположено минимум или максимум
    if (nmin<nmax)
    {
        a=nmin;
        b=nmax;
    }
    else
    {
        a=nmax;
        b=nmin;
    }
//Перебираем все элементы расположенные между максимумом и
// минимумом
    for(i=a+1,k=1;k<=b-a-1;k++)
//Проверяем, является ли очередной элемент массива
// отрицательным.
        if (x[i]<0)
        {
//Если текущий элемент массива является отрицательным
// числом, удаляем его
            for(j=i;j<n-1;j++)
                x[j]=x[j+1];
            n--;
        }
//Если x[i]>=0, переходим к следующему элементу.
        else i++;
    printf("Преобразованный массив X\n");
    for(i=0;i<n;i++)
        printf("%d\t",x[i]);
    printf("\n");
    free(x);

    return 0;
}

```

В качестве тестового можно использовать следующий массив *34 4 -7 -8 -10 7 -100 -200 -300 1*. Здесь, приведенная выше программа работает корректно, а вариант

```

for(i=a+1;i<b;) if (x[i]<0) { for(j=i;j<n-1;j++)
x[j]=x[j+1]; n--;}else i++;

```

приводит к неправильным результатам. Рекомендуем читателю самостоятельно разобраться в особенностях подобных алгоритмов удаления.

ЗАДАЧА 9. Заданы упорядоченные по возрастанию массивы $A[n]$ и $B[k]$. Сформировать из них упорядоченный по возрастанию массив $C[n+k]$.

Рассмотрим алгоритм решения задачи, известный в программировании, как алгоритм слияния.

Пусть начальные значения индексов массивов A , B , C равны 0 ($ia=ib=ic=0$). Выбираем меньшее из значений $A[ia]$, $B[ib]$ и записываем его в массив C под номером ic . После этого увеличиваем значение ic и индекс массива, который был переписан в C . Это действие повторяем, пока $ia < n$ & $ib < k$.

Может возникнуть ситуация, когда один массив закончился, а второй еще нет. Поэтому в программе должен быть организован цикл записи возможного окончания как первого так и второго массива.

Код программы с основными комментариями приведён ниже.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i,n,k,ia,ib,ic;
    printf("n="); scanf("%d",&n);
    int a[n];
    printf("Ввод массива A\n");
    for(i=0;i<n;i++)
        scanf("%d",a+i);
    printf("k="); scanf("%d",&k);
    int b[k];
    printf("Ввод массива B\n");
    for(i=0;i<k;i++)
        scanf("%d",b+i);
    int c[n+k];
    for(ia=ib=ic=0;ia<n & ib<k;)
        //Меньший из текущих элементов массивов A и B (A[ia] или
        // B[ib]) переписываем в массив C. При перезаписи
        // наращивается ic и текущий индекс перезаписанного в C
        // массива. Обратите внимание, как элегантно это
        // реализовано в C с помощью следующего оператора if.
        if (a[ia]<b[ib])
            c[ic++]=a[ia++];
        else
            c[ic++]=b[ib++];
    //Если в массиве A остались элементы переписываем их в C.
    while (ia < n)
        c[ic++]=a[ia++];
    //Если в массиве B остались элементы переписываем их в C.
    while (ib < k)
        c[ic++] = b[ib++];
    printf("Вывод массива C\n");
    for(i=0;i<k+n;i++)
        printf("%d\t",c[i]);
    printf("\n");
}
```

ЗАДАЧА 10. В массиве $X[n]$ найти группу наибольшей длины, которая состоит из знакопередающихся чисел.

Группа подряд идущих чисел внутри массива можно определить любыми двумя из трёх значений:

- nach – номер первого элемента в группе;
- kon – номер последнего элемента в группе;
- k – количество элементов в группе.

Зная любые два из выше перечисленных значений, мы однозначно определим группу внутри массива. Минимальное число элементов в группе 2.

В начале количество элементов в знакопередающейся группе равно 1. Дело в том, что если мы встретим первую пару знакопередающихся элементов, то количество их в группе сразу станет равным 2. Однако все последующие пары элементов будут увеличивать k на 1. И чтобы не решать проблему построения последовательности значений k 0,2,3,4,5,..., первоначальное значение k примем равным 1. Когда будем встречать очередную пару подряд идущих соседних элементов, то k необходимо будет увеличить на 1.

Алгоритм поиска очередной группы состоит в следующем: попарно (x_i, x_{i+1}) перебираем все элементы массива (for (i=0; i<n-1; i++)).

Если произведение соседних элементов $(x_i \cdot x_{i+1})$ отрицательно, то это означает, что они имеют разные знаки и являются элементами группы. В этом случае количество (k) элементов в группе увеличиваем на 1 (k++). Если же произведение соседних элементов $(x_i \cdot x_{i+1})$ положительно, то эти элементы не являются членами группы. В этом случае возможны два варианта:

1. Если $k > 1$, то только что закончилась группа, в этом случае kon=i – номер последнего элемента в группе, k – количество элементов в только что закончившейся группе.
2. Если $k = 1$, то это просто очередная пара незнакомых элементов.

После того как закончилась очередная группа знакопередающихся элементов, необходимо количество групп (kgr) увеличить на 1 (kgr++). Если это первая группа (kgr=1) знакопередающихся элементов, то в переменную max записываем длину этой группы ($\text{max}=k$), а в переменную kon_max номер последнего элемента группы (kon_max=i). Если это не первая группа (kgr>1), то сравниваем max и длину текущей группы (k). Если $k > \text{max}$, то в переменную max записываем длину этой группы ($\text{max}=k$), а в переменную kon_max номер последнего элемента группы (kon_max=i).

После этого в переменную k опять записываем 1 (в группе нет элементов) для формирования новой группы элементов.

По окончании цикла значение k может быть больше 1. Это означает, что в самом конце массива встретилась ещё одна группа. Для неё надо будет провести все те же действия, что и для любой другой группы. Далее приведён текст программы.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    float *x;
    int i, k, n, max, kgr, kon_max;
    //Ввод размера массива.
```

```

    printf("n="); scanf("%d",&n);
//Выделение памяти для массива.
    x=(float *)calloc(n,sizeof(float));
//Ввод элементов массива.
    printf("Введите массив x\n");
    for(i=0;i<n;i++)
        scanf("%f",x+i);
//Попарно перебираем элементы массива.
//Количество знакочередующихся групп в массиве kgr=0,
//Количество элементов в текущей группе - 1.
    for(kgr=i=0,k=1;i<n-1;i++)
//Если соседние элементы имеют разные знаки, то количество
//(k) элементов в группе увеличиваем на 1.
        if (x[i]*x[i+1]<0) k++;
        else
            if (k>1)
            {
//Если k>1, то только что закончилась группа, i - номер
// последнего элемента в группе, k - количество элементов
// в группе. Увеличиваем kgr на 1.
                kgr++;
//Если это первая группа (kgr=1) знакочередующихся
// элементов,
                if (kgr==1)
                {
//то max - длина группы (max=k),
                    max=k;
//kon_max - номер последнего элемента группы.
                    kon_max=i;
                }
            }
            else
//Если это не первая группа (kgr=1),
// то сравниваем max и длину текущей группы.
//Если k>max,
                if (k>max)
                {
//max - длина группы,
                    max=k;
//kon_max - номер последнего элемента группы.
                    kon_max=i;
                }
            }
//В переменную k записываем 1 для формирования
//новой группы элементов.
            k=1;
        }
//Если в конце массива была группа.
    if (k>1)
    {
//Количество групп увеличиваем на 1.
        kgr++;
//Если это первая группа,

```

```

        if (kgr==1)
        {
//то max - длина группы,
            max=k;
//группа закончилась на последнем элементе массива.
            kon_max=n-1;
        }
        else
//Если длина очередной группы больше max.
            if (k>max)
            {
//то в max записываем длину последней группы,
                max=k;
//группа закончилась на последнем элементе массива.
                kon_max=n-1;
            }
        }
//Если знаочередующиеся группы были,
    if (kgr>0)
    {
//то выводим информацию о группе наибольшей длины,
    printf("В массиве %d групп знаочередующихся элементов\n",
kgr);
    printf
("Группа максимальной длины начинается с элемента Номер %d",
kon_max-max+1);
    printf(", её длина %d, номер последнего элемента группы %d.\n",
max, kon_max);
// а также саму группу.
        for(i=kon_max-max+1; i<=kon_max; i++)
            printf("%f\t", x[i]);
        printf("\n");
    }
//Если знаочередующихся групп не было, то выводим
// сообщение об этом.
    else
    printf("В массиве нет групп знаочередующихся элементов\n");
    return 0;
}

```

6.4.6 Сортировка элементов в массиве

Сортировка представляет собой процесс упорядочения элементов в массиве в порядке возрастания или убывания их значений. Например, массив Y из n элементов будет отсортирован в порядке возрастания значений его элементов, если

$$Y[0] < Y[1] < \dots < Y[n-1],$$

и в порядке убывания, если

$$Y[0] > Y[1] > \dots > Y[n-1].$$

Существует большое количество алгоритмов сортировки, но все они бази-

руются на трех основных:

- сортировка обменом;
- сортировка выбором;
- сортировка вставкой.

Представим, что нам необходимо разложить по порядку карты в колоде. Для сортировки карт *обменом* можно разложить карты на столе лицевой стороной вверх и менять местами те карты, которые расположены в неправильном порядке, делая это до тех пор, пока колода карт не станет упорядоченной.

Для *сортировки выбором* из разложенных на столе карт выбирают самую младшую (старшую) карту и держат ее в руках. Затем из оставшихся карт вновь выбирают наименьшую (наибольшую) по значению карту и помещают ее позади той карты, которая была выбрана первой. Этот процесс повторяется до тех пор, пока вся колода не окажется в руках. Поскольку каждый раз выбирается наименьшая (наибольшая) по значению карта из оставшихся на столе карт, по завершению такого процесса карты будут отсортированы по возрастанию (убыванию).

Для *сортировки вставкой* из колоды берут две карты и располагают их в необходимом порядке по отношению друг к другу. Каждая следующая карта, взятая из колоды, должна быть установлена на соответствующее место по отношению к уже упорядоченным картам.

Итак, решим следующую задачу. Задан массив Y из n целых чисел. Расположить элементы массива в порядке возрастания их значений.

6.4.6.1 Сортировка методом «пузырька»

Сортировка пузырьковым методом является наиболее известной. Ее популярность объясняется запоминающимся названием, которое происходит из-за подобия процессу движения пузырьков в резервуаре с водой, когда каждый пузырек находит свой собственный уровень, и простотой алгоритма. Сортировка методом «пузырька» использует метод обменной сортировки и основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Рассмотрим алгоритм пузырьковой сортировки более подробно.

Сравним нулевой элемент массива с первым, если нулевой окажется больше первого, то поменяем их местами. Те же действия выполним для первого и второго, второго и третьего, i -го и $(i+1)$ -го, предпоследнего и последнего элементов. В результате этих действий самый большой элемент станет на последнее $(n-1)$ -е место. Теперь повторим данный алгоритм сначала, но последний $(n-1)$ -й элемент, рассматривать не будем, так как он уже занял свое место. После проведения данной операции самый большой элемент оставшегося массива станет на $(n-2)$ -е место. Так повторяем до тех пор, пока не упорядочим весь массив.

В табл. 4 представлен процесс упорядочивания элементов в массиве.

Таблица 4: Процесс упорядочивания элементов в массиве по возрастанию

Номер элемента	0	1	2	3	4
Исходный массив	7	3	5	4	2
Первый просмотр	3	5	4	2	7

Второй просмотр	3	4	2	5	7
Третий просмотр	3	2	4	5	7
Четвертый просмотр	2	3	4	5	7

Нетрудно заметить, что для преобразования массива, состоящего из n элементов, необходимо просмотреть его $n-1$ раз, каждый раз уменьшая диапазон просмотра на один элемент. Блок-схема описанного алгоритма приведена на рис. 12.

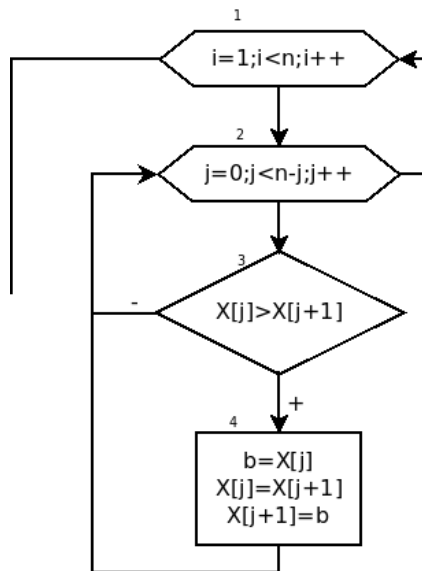


Рис. 12. Сортировка массива
пузырьковым методом

Обратите внимание на то, что для перестановки элементов (рис. 12, блок 4) используется буферная переменная b , в которой временно хранится значение элемента, подлежащего замене. Текст программы, сортирующей элементы в массиве по возрастанию методом «пузырька» приведён далее.

```

#include <stdio.h>
int main()
{
    int n,i,b,j;
    printf("n="); scanf("%d",&n);
    float y[n];
    for (i=0;i<n;i++)          //Ввод массива.
    {
        printf("\n Y[%d]=",i);
        scanf("%f",y+i);
    }
    //Упорядочивание элементов в массиве по возрастанию их
    //значений.
    for(j=1;j<n;j++)
    for(i=0;i<n-j;i++)
    if (y[i]>y[i+1])//Если текущий элемент больше следующего
    {
        b=y[i];      //Сохранить значение текущего элемента
        y[i]=y[i+1]; //Заменить текущий элемент следующим
        y[i+1]=b;    //Заменить следующий элемент текущим
    }
}

```

```

}
//Вывод упорядоченного массива
for (i=0;i<n;i++) printf("%5.1f\t",y[i]);
return 0;
}

```

Для перестановки элементов в массиве по убыванию их значений необходимо в программе и блок-схеме при сравнении элементов массива заменить знак > на <.

Однако, в этом и во всех ниже рассмотренных алгоритмах не учитывается то факт, что на каком-то этапе (или даже в начале) массив уже может оказаться отсортированным. При большом количестве элементов (сотни и даже тысячи чисел) на сортировку «в холостую» массива тратится достаточно много времени. Ниже приведены тексты двух вариантов программы сортировки по убыванию методом пузырька, в которых алгоритм прерывается, если массив уже отсортирован.

Вариант 1.

```

#include <stdio.h>
int main(int argc, char **argv)
{
    int n,i,b,j;
    int pr;
    printf("n="); scanf("%d",&n);
    float y[n];
    for (i=0;i<n;i++)          //Ввод массива.
    {
        printf("\n Y[%d]=",i);
        scanf("%f",y+i);
    }
    //Упорядочивание элементов в массиве по убыванию их
    //значений.
    for (j=1;j<n;j++)
    {
        //Предполагаем, что массив уже отсортирован (pr=0).
        for (pr=i=0;i<n-j;i++)
            if (y[i]<y[i+1])//Если текущий элемент меньше следующего
            {
                b=y[i];      //Сохранить значение текущего элемента
                y[i]=y[i+1];  //Заменить текущий элемент следующим
                y[i+1]=b;     //Заменить следующий элемент на сохраненный в b.
                //Если элемент менялись местами, массив ещё неотсортирован
                // (pr=true);
                pr=1;
            }
        printf("j=%d\n",j);
        //Если на j-м шаге соседние элементы не менялись,
        //то массив уже отсортирован, повторять смысла нет;
        if (!pr) break;
    }
    //Вывод упорядоченного массива
}

```

```

for (i=0;i<n;i++) printf("%5.1f\t",y[i]);
printf("\n");
return 0;
}

```

Вариант 2.

```

#include <stdio.h>
int main(int argc, char **argv)
{
    int n,i,b,j;
    int pr=1;
    printf("n="); scanf("%d",&n);
    float y[n];
    for (i=0;i<n;i++)          //Ввод массива.
    {
        printf("\n Y[%d]=",i);
        scanf("%f",y+i);
    }
    //Упорядочивание элементов в массиве по убыванию их
    //значений.
    //Вход в цикл, если массив не отсортирован (pr=1).
    for(j=1;pr;j++)
    {
        //Предполагаем, что массив уже отсортирован (pr=0).
        for(pr=i=0;i<n-j;i++)
            if (y[i]<y[i+1])//Если текущий элемент меньше следующего
            {
                b=y[i];    //Сохранить значение текущего элемента
                y[i]=y[i+1]; //Заменить текущий элемент следующим
                y[i+1]=b; //Заменить следующий элемент текущим
                //Если элемент менялись местами, массив ещё неотсортирован
                // (pr=1)
                pr=1;
            }
        printf("j=%d\n",j);
    }
    //Вывод упорядоченного массива
    for (i=0;i<n;i++) printf("%5.1f\t",y[i]);
    printf("\n");
    return 0;
}

```

6.4.6.2 Сортировка выбором

Алгоритм сортировки выбором приведён в виде блок-схемы на рис. 13. Идея алгоритма заключается в следующем. В массиве Y состоящем из n элементов ищем самый большой элемент (блоки 2-5) и меняем его местами с последним элементом (блок 7). Повторяем алгоритм поиска максимального элемента, но последний $(n-1)$ -й элемент не рассматриваем, так как он уже занял свою позицию.

Найденный максимум ставим на $(n-2)$ -ю позицию. Описанную выше операцию поиска проводим $n-1$ раз, до полного упорядочивания элементов в

массиве. Фрагмент программы выполняет сортировку массива по возрастанию методом выбора:

```
for (j=1; j<n; b=y[n-j], y[n-j]=y[nom], y[nom]=b, j++)
for (max=y[0], nom=0, i=1; i<=n-j; i++)
if (y[i]>max) {max=y[i]; nom=i;}
```

Для упорядочивания массива по убыванию необходимо менять минимальный элемент с последним элементом.

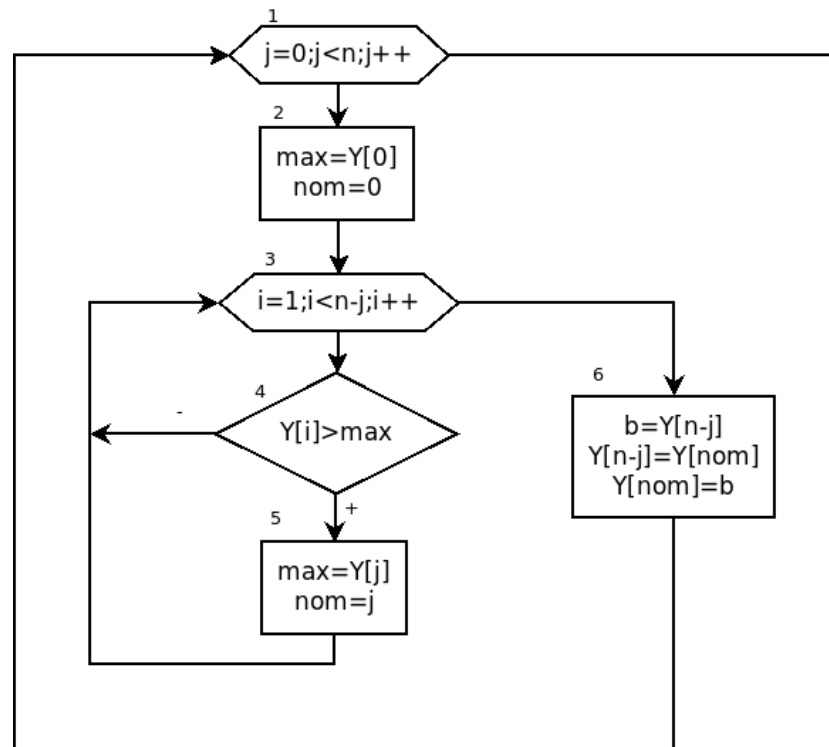


Рис. 13. Сортировка массива выбором наибольшего элемента

6.4.6.3 Сортировка вставкой

Сортировка вставкой заключается в том, что сначала упорядочиваются два элемента массива. Затем делается вставка третьего элемента в соответствующее место по отношению к первым двум элементам. Четвертый элемент помещают в список из уже упорядоченных трех элементов. Этот процесс повторяется до тех пор, пока все элементы не будут упорядочены.

Прежде чем приступить к составлению блок-схемы рассмотрим следующий пример. Пусть известно, что в массиве из десяти элементов первые шесть уже упорядочены (с нулевого по пятый), а шестой элемент нужно вставить между вторым и четвертым. Сохраним шестой элемент во вспомогательной переменной, а на его место запишем пятый. Далее четвертый переместим на место пятого, а третий на место четвертого, тем самым, выполнив сдвиг элементов массива на одну позицию вправо. Записав содержимое вспомогательной переменной в третью позицию, достигнем нужного результата.

Составим блок-схему алгоритма (рис. 14), учитывая, что возможно описанные выше действия придется выполнить неоднократно.

Организуем цикл для просмотра всех элементов массива, начиная с пер-

вого (блок 1). Сохраним значение текущего i -го элемента во вспомогательной переменной b , так как оно может быть потеряно при сдвиге элементов (блок 2) и присвоим переменной j значение индекса предыдущего ($i-1$)-го элемента массива (блок 3). Далее движемся по массиву влево в поисках элемента меньшего, чем текущий и пока он не найден сдвигаем элементы вправо на одну позицию. Для этого организуем цикл (блок 4), который прекратиться, как только будет найден элемент меньше текущего. Если такого элемента в массиве не найдется и переменная j станет равной (-1) , то это будет означать, что достигнута левая граница массива, и текущий элемент необходимо установить в первую позицию. Смещение элементов массива вправо на одну позицию выполняется в блоке 5, а изменение счетчика j в блоке 6. Блок 7 выполняет вставку текущего элемента в соответствующую позицию.

Далее приведен фрагмент программы, реализующей сортировку массива методом вставки.

```
for (i=1; i<n; y[j+1]=b, i++)
```

```
for (b=y[i], j=i-1; (j>-1 && b<y[j]); y[j+1]=y[j], j--);
```

Рассмотрим несколько несложных задач, связанных с упорядочиванием.

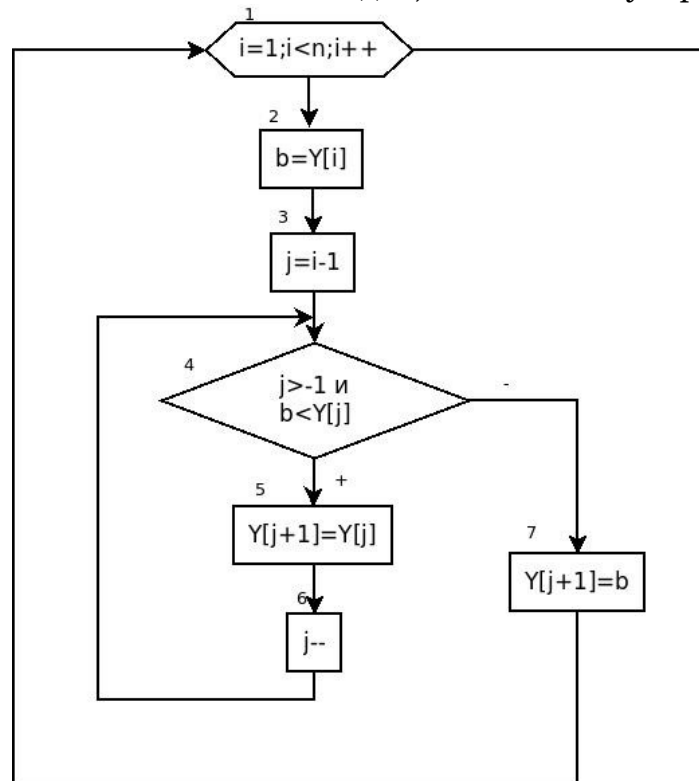


Рис. 14. Сортировка массива вставкой

6.4.6.4 Алгоритм быстрой сортировки. Реализация на C(C++)

Основная идея

Делим массив на 2 части:

1. Массив, состоящий из *больших* элементов, все элементы в нём больше или равны опорному $x_i \geq s$.
2. Массив, состоящий из *маленьких* элементов, все элементы в нём меньше или равны опорному $x_i \leq s$.

Любой элемент из первой части \leq элементу из второй части.

После этого каждую часть опять делим на «маленькие» и большие «элементы».

Процесс деления на две части.

В первой части ищем первый элемент, который больше опорного. Во второй части ищем элемент меньше опорного. Эти элементы находятся не в своей части. Их меняем местами. И так продолжается до тех пор пока в первой части есть *большие*, а во второй – *маленькие* элементы.

Разделили массив на части. Для каждой из них рекурсивно применяем алгоритм деления массива «большие» и «маленькие». Повторяем алгоритм до тех пор, пока не отсортируем (пока в рекурсивном вызове не останется массив из одного элемента).

Количество делений на «большие» и «маленькие» элементы – $\log_2 n$.

Количество сравнений для деления массива на «большие» и «маленькие» элементы – в среднем n .

Всего вычислений – порядка $n \cdot \log_2 n$.

Алгоритм быстрой сортировки носит имя Хоара, по имени автора английского информатика Чарльза Хоара, который разработал его во время его стажировки в МГУ в 1960 году.

Формальное описание алгоритма

1. Задан массив $X[N]$. Вводим два счётчика (f, L), которые будут двигаться навстречу друг другу. Начальные значения счётчиков $f=1, L=N$. Выберем опорный элемент $s = X\left[\frac{f+L}{2}\right]$.
2. Будем увеличивать f на 1, до тех пор, пока $x[f] < s$.
3. Будем уменьшать L на 1, до тех пор, пока $x[L] > s$.
- 2-3. Алгоритм встречного поведения.
4. Если $i < j$, то меняем местами $x[f]$ и $x[L]$. После этого $f++$, $L--$.
5. Повторяем п.2-4, до тех пор пока f не станет больше L .

Элементы правее l больше или равны опорного. Все элементы левее L меньше или равны опорному. Мы как бы раздели массив на две части (левая – маленькие и правая – большие).

6. Теперь рекурсивно запускаем алгоритм 1-5 на массивах $x[f:N]$, и $x[1:f-1]$

https://www.youtube.com/watch?v=Xgaj0Vxz_to

https://www.youtube.com/watch?v=34q6gcWaE_8

Оценка времени быстродействия на C(C++)

```
#include <time.h>
time1=clock();
```

ЗДЕСЬ фрагмент кода

```
time2=clock();
cout<<"\nВремя умножения матриц="<<
(time2-time1)/CLOCKS_PER_SEC<<endl;
```

Функция clock

Прототип функции clock:

```
clock_t clock( void );
```

Заголовочный файл

Название	Язык
time.h	C
ctime	C++

Описание:

Функция возвращает количество временных тактов, прошедших с начала запуска программы. С помощью макроса CLOCKS_PER_SEC функция получает количество пройденных тактов за 1 секунду. Таким образом, зная сколько выполняется тактов в секунду, зная время запуска программы можно посчитать время работы всей программы или отдельного её фрагмента, что и делает данная функция.

Возвращаемое значение

Число тактов прошедшее с момента запуска программы. В случае ошибки, функция возвращает значение -1. Возвращаемое значение функции clock имеет тип данных clock_t, который определен в <ctime>. Тип данных clock_t способен представлять временные такты, а также поддерживает арифметические операции.

Реализация на C(C++)

```
#include <iostream>
#include <ctime>
using namespace std;
int first, last;
//функция сортировки
void quicksort(int *mas, int first, int last)
{
    int mid, count;
    int f=first, L=last;
    mid=mas[(f+L) / 2]; //вычисление опорного элемента
    do
    {
        while (mas[f]<mid) f++;
        while (mas[L]>mid) L--;
        if (f<=L) //перестановка элементов
        {
            count=mas[f];
            mas[f]=mas[L];
            mas[L]=count;
```

```

f++;
L--;
}
} while (f<L);
if (first<L) quicksort(mas, first, L);
if (f<last) quicksort(mas, f, last);
}
const int n=10000;
//главная функция
int main()
{
    setlocale(LC_ALL, "Rus");
    int *A=new int[n], temp;
    clock_t time1,time2;
    srand(time(NULL));
    cout<<"БЫСТРАЯ СОРТИРОВКА!!!!\n";
    cout<<"Исходный массив: ";
    for (int i=0; i<n; i++)
    {
        A[i]=rand()%n;
        if (i==0 || i==n/2 || i== n-1)    cout<<A[i]<<" ";
    }
    first=0; last=n-1;
    time1=clock();
    quicksort(A, first, last);
    time2=clock();
    cout<<endl<<"Результирующий массив: ";
    for (int i=0; i<n; i++)
    {
        A[i]=rand()%n;
        if (i==0 || i==n/2 || i== n-1)    cout<<A[i]<<" ";
        cout<<"\nВремя    быстрой    сортировки="<<(double) (time2-
time1)/CLOCKS_PER_SEC<<endl;
    }
    cout<<"БЫСТРАЯ СОРТИРОВКА!!!!\n";
    cout<<"Исходный массив: ";
    for (int i=0; i<n; i++)
    {
        A[i]=rand()%n;
        if (i==0 || i==n/2 || i== n-1)    cout<<A[i]<<" ";
    }
    time1=clock();
    for (int i=0; i<n-1; i++)
    for (int j=0; j<n-1-i; j++)
    if (A[j]>A[j+1])
    {
        temp=A[j];
        A[j]=A[j+1];
        A[j+1]=temp;
    }

    time2=clock();
    cout<<endl<<"\nРезультирующий массив: ";
    for (int i=0; i<n; i++)

```



```
    if (i==0 || i==n/2 || i== n-1)    cout<<A[i]<<" ";  
    cout<<"\nВремя сортировки методом пузырька="<<(double)  
(time2-time1)/CLOCKS_PER_SEC<<endl;  
  
    delete []A;  
    return 13;  
}
```

БЫСТРАЯ СОРТИРОВКА!!!!

Исходный массив: 5570 5660 5470

Результирующий массив: 3 4997 9999

Время быстрой сортировки=0.001101

БЫСТРАЯ СОРТИРОВКА!!!!

Исходный массив: 8464 7179 1104

Результирующий массив: 0 4874 9999

Время сортировки методом пузырька=0.267147

7. Реализации на Паскале

```

program qsort;
uses crt, dos;
const n=30000;
type
massiv=array[1..n] of integer;

{процедура сортировки}
procedure quicksort(var mas: massiv; first, last:
integer);
var f, l, mid, count: integer;
begin
f:=first;
l:=last;
mid:=mas[(f+l) div 2]; {вычисление опорного элемента}
repeat
while mas[f]<mid do inc(f);
while mas[l]>mid do dec(l);
if f<=l then {перестановка элементов}
begin
count:=mas[f];
mas[f]:=mas[l];
mas[l]:=count;
inc(f);
dec(l);
end;
until f>l;
if first<l then quicksort(mas, first, l);
if f<last then quicksort(mas, f, last);
end;
{основной блок программы}
var A: massiv;
temp,first, last, i,j: integer;
hour,min,sec,hund: word;
t,t1,t2: double;

begin
clrscr;
randomize;
for i:=1 to n do

```

```

A[i]:=random(n);
gettime(hour,min,sec,hund);
t1:=sec + hund*0.01 + min*60 + hour*3600;
first:=1; last:=n;
quicksort(A, first, last);
gettime(hour,min,sec,hund);
t2:=sec + hund*0.01 + min*60 + hour*3600;
writeln('Время быстрой сортировки:',t2-t1:1:8);
for i:=1 to n do
A[i]:=random(n);
gettime(hour,min,sec,hund);
t1:=sec + hund*0.01 + min*60 + hour*3600;
for i:=1 to n do
for j:=1 to n-i do
if A[j]>A[j+1] then
begin
temp:=A[j];
A[j]:=A[j+1];
A[j+1]:=temp;
end;

gettime(hour,min,sec,hund);
t2:=sec + hund*0.01 + min*60 + hour*3600;
writeln('Время сортировки методом пузырька:',t2-t1:1:8);
end.

```

Время быстрой сортировки:0.01000000

Время сортировки методом пузырька:2.43000000

ЗАДАЧА 11. Задан массив $a[n]$ упорядоченный по убыванию, вставить в него некоторое число b , не нарушив упорядоченности массива.

Массив является упорядоченным по убыванию, если каждое последующий элемент массива не больше предыдущего, т. е. при выполнении следующей совокупности неравенств $a_0 \geq a_1 \geq a_2 \geq \dots \geq a_{n-3} \geq a_{n-2} \geq a_{n-1}$.

Для вставки в подобный массив некоторого числа без нарушений упорядоченности, необходимо:

1. Найти номер k первого числа в массиве, которое $a_k \leq b$.
2. Все элементы массива a , начиная от $n-1$ до k -го сдвинуть на один вправо⁴.
3. На освободившееся место с номером k записать число b .

Текст программы с комментариями приведён ниже.

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int i, k, n;
    float b;
    //Ввод размера исходного массива.
    printf("n="); scanf("%d", &n);
    //Выделение памяти с учётом предстоящей вставки одного
    числа в массив.
    float a[n+1];
    //Ввод исходного упорядоченного по убыванию массива.
    printf("Введите массив a\n");
    for(i=0; i<n; i++)
        scanf("%f", a+i);
    //Ввод вставляемого в массив числа b.
    printf("Введите число b="); scanf("%f", &b);
    //Если число b меньше всех элементов массива, записываем b
    // в последний элемент массива.
    if (a[n-1] >= b) a[n] = b;
    else
    //Иначе
    {
        for(i=0; i<n; i++)
        //Ищем первое число, меньшее b.
        if (a[i] <= b)
        {
            //Запоминаем его номер в переменной k.
            k = i;
            break;
        }
        //Все элементы массива от n-1-го до k-го сдвигаем на один
        вправо.
        for(i=n-1; i>=k; i--)
            a[i+1] = a[i];
        //Вставляем число b в массив.
```

⁴ Очень важно, что сдвиг осуществляем от $n-1$ -го до k -го, в противном случае элементы массива оказались бы испорченными.

```

        a[k]=b;
    }
    printf("Преобразованный массив a\n");
    for (i=0;i<=n;i++) printf("%5.1f\t",a[i]);
    printf("\n");
    return 0;
}

```

Обратите внимание, при решении задачи с массивом упорядоченным по возрастанию необходимо во фрагменте

```

    if (a[n-1]>=b) a[n]=b;
    else
    {
        for(i=0;i<n;i++)
            if (a[i]<=b)
            {
                k=i;
                break;
            }
    }

```

заменить все операции отношения на противоположные.

ЗАДАЧА 12. Проверить является ли массив упорядоченным по возрастанию.

Для проверки упорядоченности по возрастанию $a[n]$ ⁵ можно поступить следующим образом. Предположим, что массив упорядочен ($pr=1$). Если хотя бы для одной пары соседних элементов выполняется условие $a_i > a_{i+1}$, то массив не упорядочен по возрастанию ($pr=0$). Текст программы с комментариями приведён ниже. Читателю предлагается преобразовать программу таким образом, чтобы осуществлялась проверка, упорядочен ли массив по убыванию.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    int i,n;
    float *a;
    int pr=1;
    //Ввод размера исходного массива.
    printf("n="); scanf("%d",&n);
    //Выделение памяти для массива.
    a=(float *)calloc(n,sizeof(float));
    //Ввод исходного массива.
    printf("Введите массив a\n");
    for(i=0;i<n;i++)
        scanf("%f",a+i);
    //Предполагаем, что массив упорядочен (pr=1),
    //Перебираем все пары соседних значений (i - номер пары),
    //при i равном n-2 будем сравнивать последнюю пару a[n-2]
    // и a[n-1].
    for(i=0;i<n-1;i++)

```

⁵ Массив является упорядоченным по возрастанию, если выполняются условия $a_0 \leq a_1 \leq a_2 \leq \dots \leq a_{n-3} \leq a_{n-2} \leq a_{n-1}$.

```
//Если для очередной пары соседних элементов выяснилось,  
// что предыдущий элемент больше последующего, то массив  
// неупорядочен по возрастанию (pr=false), остальные пары  
// соседних значений, можно не проверять (оператор break)  
    if (a[i]>a[i+1]) {pr=0;break;}  
    if (pr) printf("Массив упорядочен по возрастанию\n");  
    else printf("Массив не упорядочен по возрастанию\n");  
    free(a);  
    return 0;  
}
```

6.5 Совместное использование динамических массивов, указателей, функций в сложных задачах обработки массивов

Функции в С могут возвращать только одно скалярное значение, однако использование указателей в качестве аргументов функций позволяет обойти это ограничение и писать сложные функции, которые могут возвращать несколько значений.

Если в качестве аргумента в функцию передаётся указатель (адрес), то следует иметь в виду следующее. *При изменении в функции значений, хранящегося по этому адресу, будет происходить глобальное изменение значений, хранящихся по данному адресу в памяти компьютера.* Таким образом, получаем механизм с помощью которого можно возвращать множество значений. Для этого просто надо передавать их, как адреса (указатели). В литературе по программированию подобный механизм зачастую называют *передачей параметров по адресу*. При этом не следует забывать о том, что этот механизм работает без всяких исключений. Любое изменение значений, переданных в функцию по адресу, приводит к глобальному изменению.

Например, в следующей программе, функция `kvadrat` вычисляет корни квадратного уравнения. Обратите внимание на способы передачи параметров.

Задача написать функцию вычисления корней квадратного уравнения.

Версия 1.

```
#include <iostream>
#include <math.h>
using namespace std;
bool kvadrat(float a, float b, float c, float *x1, float
*x2)
{
    float d=b*b-4*a*c;
    if (d<0) return false;
    else
    {
        *x1=(-b+sqrt(d))/2/a;
        *x2=(-b-sqrt(d))/2/a;
        return true;
    }
}

int main(int argc, char **argv)
{
    float a1,b1,c1,x,y;
    cout<<"Введите a1,b1,c1";cin>>a1>>b1>>c1;
    if(kvadrat(a1,b1,c1,&x,&y))
    cout<<" x="<<x<<" y="<<y<<endl;
    else cout<<"Нет действительных корней"<<endl;
    return 0;
}
```


Версия 2а.

```

#include <iostream>
#include <math.h>
using namespace std;
void kvadrat(float a, float b, float c, float *x1, float
*x2, bool * pr)
{
    float d=b*b-4*a*c;
    if (d<0) *pr=false;
    else
    {
        *x1=(-b+sqrt(d))/2/a;
        *x2=(-b-sqrt(d))/2/a;
        *pr=true;
    }
}

int main(int argc, char **argv)
{
    float a1,b1,c1,x,y;
    bool flag;
    cout<<"Введите a1,b1,c1";cin>>a1>>b1>>c1;
    kvadrat(a1,b1,c1,&x,&y,&flag);
    if(flag)
    cout<<" x="<<x<<" y="<<y<<endl;
    else cout<<"Нет действительных корней"<<endl;
    return 0;
}

```

Версия 2b (с ошибкой при работе с указателями).

```
#include <iostream>
#include <math.h>
using namespace std;
void kvadrat(float a, float b, float c, float *x1, float
*x2, bool * pr)
{
    float d=b*b-4*a*c;
    if (d<0) *pr=false;
    else
    {
        *x1=(-b+sqrt(d))/2/a;
        *x2=(-b-sqrt(d))/2/a;
        *pr=true;
    }
}

int main(int argc, char **argv)
{
    float a1,b1,c1,*x,*y;
    bool *flag;
    cout<<"Введите a1,b1,c1";cin>>a1>>b1>>c1;
    kvadrat(a1,b1,c1,x,y,flag);
    if(flag)
        cout<<" x="<<x<<" y="<<y<<endl;
    else cout<<"Нет действительных корней"<<endl;
    return 0;
}
```

Версия 2с (2б исправленная) .

```

#include <iostream>
#include <math.h>
using namespace std;
void kvadrat(float a, float b, float c, float *x1, float
*x2, bool * pr)
{
    float d=b*b-4*a*c;
    if (d<0) *pr=false;
    else
    {
        *x1=(-b+sqrt(d))/2/a;
        *x2=(-b-sqrt(d))/2/a;
        *pr=true;
    }
}

int main(int argc, char **argv)
{
    float a1,b1,c1,*x,*y;
    bool *flag;
    x=new float[1];
    y=new float[1];
    flag=new bool[1];
    cout<<"Введите a1,b1,c1";cin>>a1>>b1>>c1;
    kvadrat(a1,b1,c1,x,y,flag);
    if(flag)
        cout<<" x="<<*x<<" y="<<*y<<endl;
    else cout<<"Нет действительных корней"<<endl;
    return 0;
}

```

Версия 2d (передача данных по ссылке)

```

#include <iostream>
#include <math.h>
using namespace std;
void kvadrat(float a, float b, float c, float &x1, float
&x2, bool &pr)
{
    float d=b*b-4*a*c;
    if (d<0) pr=false;
    else
    {
        x1=(-b+sqrt(d))/2/a;
        x2=(-b-sqrt(d))/2/a;
        pr=true;
    }
}

int main(int argc, char **argv)
{
    float a1,b1,c1,x,y;
    bool flag;
    cout<<"Введите a1,b1,c1";cin>>a1>>b1>>c1;
    kvadrat(a1,b1,c1,x,y,flag);
    if(flag)
        cout<<" x="<<x<<" y="<<y<<endl;
    else cout<<"Нет действительных корней"<<endl;
    return 0;
}

```

В качестве примера рассмотрим задачу *удаления положительных элементов из массива* (см. задачу 7). Пользуясь тем, что задача несложная, напишем несколько вариантов функции удаления элемента с заданным номером из массива.

Для решения задачи удаления положительных элементов из массива понадобится функция удаления элемента из массива.

Назовём функцию `udal`. Её входными параметрами будут:

- массив (`x`),
- его размер (`n`),
- номер удаляемого элемента (`k`).

Функция возвращает:

- модифицированный массив (`x`),
- размер массива после удаления (`n`).

При передаче массива с помощью указателя, исчезает проблема возврата в главную программу модифицированного массива, размер массива будем возвращать с помощью обычного оператора `return`.

Заголовок (прототип) функции `udal` может быть таким:

```
int udal (float *x, int k, int n)
```

Здесь `x` – массив, `k` – номер удаляемого элемента, `n` – размер массива.

Весь текст функции можно записать так

```
int udal(float *x, int k, int n)
{
    int i;
    if (k>n-1) return n;
    else
    {
        for(i=k;i<n-1;i++)
            x[i]=x[i+1];
        n--;
        return n;
    }
}
```

Ниже приведён весь текст программы удаления положительных элементов из массива `x` с использованием функции `udal` и комментарии к нему.

```
#include <stdio.h>
int udal(float *x, int k, int n)
{
    int i;
    //Если номер удаляемого элемента больше номера последнего
    // элемента, то удалять нечего, в этом случае возвращается
    // неизменённый размер массива.
    if (k>n-1) return n;
    else
    {
        //Удаляем элемент с номером k.
        for(i=k;i<n-1;i++)
            x[i]=x[i+1];
        n--;
        //Возвращаем изменённый размер массива.
```

```

        return n;
    }
}

int main(int argc, char **argv)
{
    int i,n;
    printf("n="); scanf("%d",&n);
    //Выделяем память для динамического массива x.
    float x[n];
    //Ввод элементов массива.
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%f",x+i);
    for(i=0;i<n;)
        if (x[i]>0)
            //Если текущий элемент положителен,
            //то удаления элемента с индексом i,
            //Вызываем функцию udal, которая
            //изменяет элементы, хранящиеся по адресу x,
            //и возвращает размер массива.
            n=udal(x,i,n);
    //иначе (x[i]<=0) - переходим к следующему элементу
    массива.
    else i++;
    //Вывод элементов массива после удаления.
    printf("Преобразованный массив X\n");
    for (i=0;i<n;i++) printf("%5.1f\t",x[i]);
    printf("\n");
    return 0;
}

```

Эту функцию можно переписать и по другому, передавая и массив и его размер, как указатели, в этом случае функция будет такой.

```

void udal(float *x, int k, int *n)
{
    int i;
    for(i=k;i<*n-1;i++)
        x[i]=x[i+1];
    if (k<*n) --*n;
}

```

В этом случае изменится и обращение к udal в функции main.

Ниже приведён модифицированный текст программы удаления положительных элементов из массива x с использованием функции udal(float *x, int k, int *n) и комментарии к нему.

```

#include <stdio.h>
void udal(float *x, int k, int *n)
{
    int i;
    //Если номер удаляемого элемента больше номера последнего
    элемента,
    //то удалять нечего, в этом случае возвращается

```

```

неизменённый размер
// массива
//Удаляем элемент с номером k.
    for(i=k;i<*n-1;i++)
        x[i]=x[i+1];
//Уменьшаем на 1 значение, хранящееся по адресу n.
//Обратите внимание, что надо писать именно --*n,
// *n-- - НЕПРАВИЛЬНО!!!!!!!!!!!!!!!!!!!!
    if (k<*n) --*n;
}
int main(int argc, char **argv)
{
    int i,n;
    printf("n="); scanf("%d",&n);
//Выделяем память для динамического массива x.
    float x[n];
//Ввод элементов массива.
    printf("Введите элементы массива X\n");
    for(i=0;i<n;i++)
        scanf("%f",x+i);
    for(i=0;i<n;)
        if (x[i]>0)
//Если текущий элемент положителен,
//то удаления элемента с индексом i,
//Вызываем функцию udal, которая
//изменяет элементы, хранящиеся по адресу x,
// и изменят значение переменной n.
            udal(x,i,&n);
//иначе (x[i]<=0) - переходим к следующему элементу
// массива.
            else i++;
//Вывод элементов массива после удаления.
    printf("Преобразованный массив X\n");
    for (i=0;i<n;i++) printf("%5.1f\t",x[i]);
    printf("\n");
    return 0;
}

```

Рекомендуем разобраться с этими примерами для понимания механизма передачи параметров по адресу.

ЗАДАЧА 13. Из массива целых чисел удалить все простые числа, значение которых меньше среднего арифметического элементов массива. Полученный массив упорядочить по возрастанию.

Алгоритм решения этой задачи без применения функций будет очень громоздким, а текст программы малопонятным. Поэтому, разобьем задачу на подзадачи:

- вычисление среднего арифметического элементов массива;
- определение простого числа;
- удаление элемента из массива;
- упорядочивание массив.

Прототипы функций, которые предназначены для решения подзадач,

могут выглядеть так:

- `float sr_arifm(int *x, int n)` – вычисляет среднее арифметическое массива `x` из `n` элементов;
- `int prostoe(int n)` – проверяет, является ли целое число `n` простым, результат функции – 1, если число простое и 0 – в противном случае;
- `void udal(int *x, int m, int *n)` – удаляет элемент с номером `m` в массиве `x` из `n` элементов;
- `void upor(int *x, int N, int pr)` – сортирует массив `x` из `n` элементов по возрастанию или по убыванию, направление сортировки зависит от значения параметра `pr`, если `pr=1`, то выполняется сортировка по возрастанию, если `pr=0`, то по убыванию.

Текст программы с комментариями:

```
#include <stdio.h>
#include <stdlib.h>
//Функция вычисления среднего значения.
float sr_arifm(int *x, int n)
{
    int i; float s=0;
    for(i=0;i<n;s+=x[i],i++);
    if (n>0) return(s/n);
    else return 0;
}
//Функция для определения простого числа:
int prostoe(int n)
{
    int pr; int i;
    if(n<2) pr=0;else
    for(pr=1,i=2;i<=n/2;i++)
    if(n%i==0) {pr=0;break;}
    return pr;
}
//Функция удаления элемента из массива.
void udal(int *x, int m, int *n)
{
    int i;
    for(i=m;i<*n-1;*(x+i)=*(x+i+1),i++);
    --*n;
    x=(int *)realloc(x,*n*sizeof(int));
}
//Функция сортировки массива.
void upor(int *x, int n, int pr)
{
    int i,j,b;
    if (pr)
    {
        for(j=1;j<=n-1;j++)
        for(i=0;i<=n-1-j;i++)
        if (*(x+i)>*(x+i+1))
        {
            b=*(x+i);
```



```

*(x+i)=*(x+i+1);
*(x+i+1)=b;
}
}
else
for(j=1;j<=n-1;j++)
for(i=0;i<=n-1-j;i++)
if (*(x+i)<*(x+i+1))
{
b=*(x+i);
*(x+i)=*(x+i+1);
*(x+i+1)=b;
}
}
int main()
{
int *a,n,i; float sr;
printf("n="); scanf("%d",&n); //Ввод размерности массива.
a=(int *)calloc(n,sizeof(int)); //Выделение памяти.
printf("Введите массив A\n");
for(i=0;i<n;i++) scanf("%d",&a[i]); //Ввод массива.
sr=sr_arifm(a,n); //Вычисление среднего арифметического.
printf("sr=%f\n",sr); //Вывод среднего арифметического.
for(i=0;i<n;i++)
{
//Если число простое и меньше среднего,
if(prostoe(*(a+i))&& *(a+i)<sr)
udal(a,i,&n); //удалить его из массива,
else i++; //иначе, перейти к следующему элементу.
}
printf("Массив A\n"); //Вывод модифицированного
массива.
for (i=0;i<n;i++) printf("%d\t",*(a+i));
printf("\n");
upor(a, n,1); //Сортировка массива.
//Вывод упорядоченного массива.
printf("Упорядоченный массив A\n");
for (i=0;i<n;i++) printf("%d\t",*(a+i));
printf("\n");
free(a); //Освобождение памяти.
return 0;
}

```

ЗАДАЧА 14. Все положительные элементы целочисленного массива G переписать в массив W . В массиве W упорядочить по убыванию элементы, которые расположены между наибольшим и наименьшим числами палиндромами.

Для создания этой программы напомним следующие функции:

- `int form (int *a, int n, int *b)`, которая из массива целых чисел a формирует массив положительных чисел b , n – количество чисел в массиве a , функция возвращает число элементов в массиве b .

- `int palindrom (int n)`, которая проверяет является ли число n палиндромом.
- `sort (int *x, int n, int k, int p)` которая сортирует по возрастанию элементы массива $x[n]$, расположенные между k -м и p -м элементами массива.

Рекомендуем читателю самостоятельно разобрать текст программы, реализующей решение задачи 14.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int kvo_razryad(int M)
{
    long int k;
    for(k=1;M>9;M/=10,k++);
    return k;
}
int palindrom (int n)
{
    int k=kvo_razryad(n),s,p=n;
    for(s=0;p!=0;p/=10,k--)
        s+=(p%10)*pow(10,k-1);
    if (s==n) return 1; else return 0;
}
int form (int *a, int n, int *b)
{
    int i,k;
    for(i=k=0;i<n;i++)
        if(a[i]>0)
            b[k++]=a[i];
    return k;
}
void sort (int *x, int n, int k, int p)
{
    int i,nom,j;
    int b;
    for(i=k+1;i<p;)
    {
        nom=i;
        for(j=i+1;j<p;j++)
            if (x[j]<x[nom]) nom=j;
        b=x[p-1]; x[p-1]=x[nom]; x[nom]=b;
        p--;
    }
}
int main(int argc, char **argv)
{
    int *G,*W;
    int nmax,nmin,kp,i,N,k;
    printf("N="); scanf("%d",&N);
    G=(int *)calloc(N,sizeof(int));
```

```

W=(int *)calloc(N,sizeof(int));
printf("Ввод массива G\n");
for(i=0;i<N;i++)
scanf("%d",G+i);
k=form(G,N,W);
printf("Вывод массива W\n");
for(i=0;i<k;i++)
printf("%d ",W[i]);
printf("\n");
for(kp=i=0;i<k;i++)
if (palindrom(W[i]))
{
    kp++;
    if (kp==1) {nmax=i;nmin=i;}
    else
    {
        if (W[i]<W[nmin]) nmin=i;
        if (W[i]>W[nmax]) nmax=i;
    }
}
if (nmax<nmin)
sort(W,k,nmax,nmin);
else
sort(W,k,nmin,nmax);
printf("Вывод преобразованного массива W\n");
for(i=0;i<k;i++)
printf("%d ",W[i]);
printf("\n");
return 0;
}

```

Результаты работы программы представлены ниже.

N=17

Ввод массива G

**-5 -6 191 121 12 -13 14 15 -5 100 666 -666 15251 16261
16262 991 -724**

Вывод массива W

191 121 12 14 15 100 666 15251 16261 16262 991

Вывод преобразованного массива W

191 121 15251 666 100 15 14 12 16261 16262 991

6.6 Указатели на функции

При решении некоторых задач возникает необходимость передавать имя функции, как параметр. В этом случае формальным параметром является указатель на передаваемую функцию. В общем виде прототип указателя на функцию можно записать так.

```
type (*name_f)(type1, type2, type3,...)
```

Здесь

`name_f` – имя функции

`type` – тип возвращаемый функцией,

`type1, type2, type3,...` – типы формальных параметров функции.

В качестве примера рассмотрим решение несколько широко известных математических задач.

6.6.1 Решение нелинейных уравнений

Данную проблему рассмотрим на примере решения следующей практической задачи.

ЗАДАЧА 15. Найти корни уравнения $x^2 - \cos(5 \cdot x) = 0$. Для решения задачи использовать:

1. *метод половинного деления,*
2. *метод хорд,*
3. *метод касательных (метод Ньютона),*
4. *метод простой итерации.*

Оценить степень точности предложенных *численных методов*, определив за сколько итераций был найден корень уравнения. Вычисления проводить с точностью $\varepsilon = 10^{-3}$.

Вообще говоря, *аналитическое решение уравнения*

$$f(x) = 0 \quad (6.1.1)$$

можно найти только для узкого класса функций. В большинстве случаев приходится решать уравнение (6.1.1) *численными методами*. Численное решение уравнения (6.1.1) проводят в два этапа: сначала необходимо *отделить корни уравнения*, т.е. найти достаточно тесные промежутки, в которых содержится только один корень, эти промежутки называют *интервалами изоляции корней*; на втором этапе проводят уточнение отделенных корней, т.е. *находят корни с заданной точностью*.

Интервал можно выделить, изобразив график функции, или каким-либо другим способом. Но все способы основаны на следующем свойстве непрерывной функции: если функция $f(x)$ непрерывна на интервале $[a, b]$ и на его концах имеет различные знаки, $f(a) \cdot f(b) < 0$, то между точками имеется хотя бы один корень. Будем считать интервал настолько малым, что в нем находится только один корень. Рассмотрим самый простой способ уточнения корней.

Графическое решение задачи 15 показано на рис. 15. Так как функция $f(x) = x^2 - \cos(5 \cdot x)$ дважды пересекает ось абсцисс, можно сделать вывод о наличии в уравнении $x^2 - \cos(5 \cdot x) = 0$ двух корней. Первый находится на интервале $[-0.4; -0.2]$, второй принадлежит отрезку $[0.2; 0.4]$.

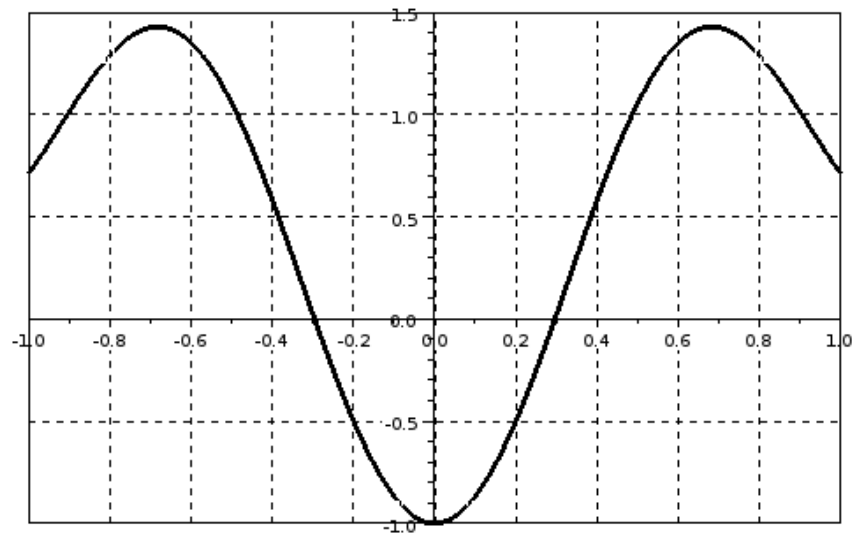


Рисунок 15. Геометрическое решение задачи 15

Рассмотрим, предложенные в задаче численные методы решения нелинейных уравнений.

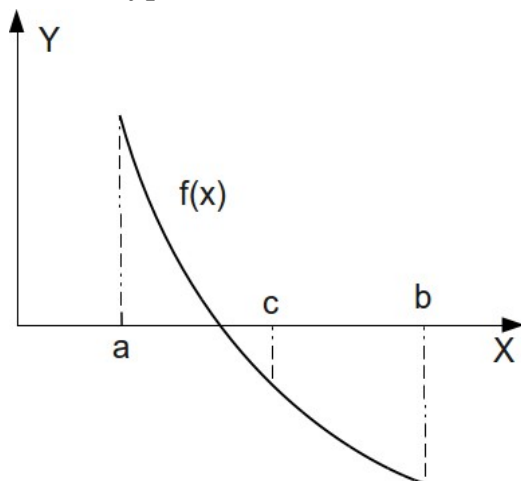


Рисунок 16. Графическая интерпретация метода половинного деления

Метод половинного деления (дихотомии). Пусть был выбран интервал изоляции $[a, b]$ (рис. 16). Примем за первое приближение корня точку c , которая является серединой отрезка $[a, b]$. Далее будем действовать по следующему алгоритму:

1. Находим точку $c = \frac{a+b}{2}$;
2. Находим значение $f(c)$;
3. Если $f(a) \cdot f(c) < 0$, то корень лежит на интервале $[a, c]$, иначе корень лежит на интервале $[c, b]$;
4. Если величина интервала меньше либо равна ε , то найден корень с точностью ε , иначе возвращаемся к п.1.

Итак, для вычисления одного из корней уравнения $x^2 - \cos(5 \cdot x) = 0$ методом половинного деления достаточно знать интервал изоляции корня $a=0.2; b=0.4$ и точность вычисления $\varepsilon=10^{-3}$.

Блок-схема алгоритма решения уравнения методом дихотомии приведена на рис.17. Понятно, что здесь c - корень заданного уравнения.

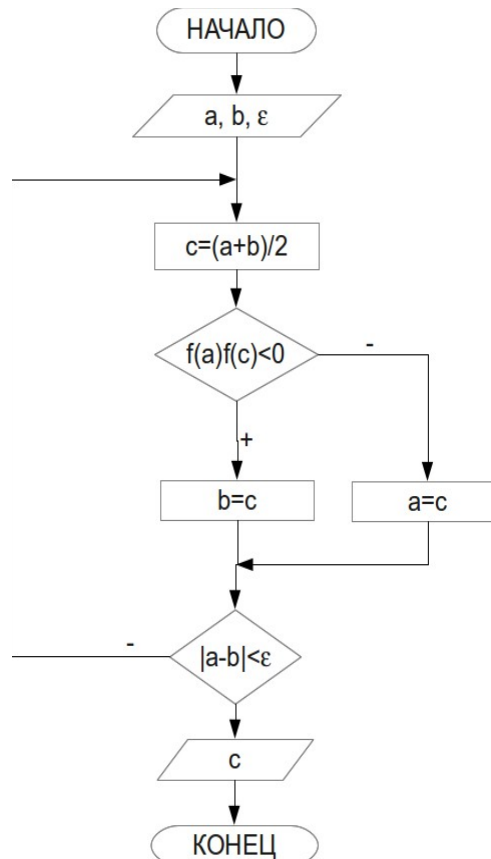


Рисунок 17. Алгоритм
решения уравнения методом
дихотомии

Однако, несмотря на простоту, такое последовательное сужение интервала проводится редко, так как требует слишком большого количества вычислений. Кроме того, этот способ не всегда позволяет найти решение с заданной точностью. Рассмотрим другие способы уточнения корня. При применении этих способов будем требовать, чтобы функция $f(x)$ удовлетворяла следующим условиям на интервале $[a, b]$:

- функция $f(x)$ непрерывна вместе со своими производными первого и второго порядка. Функция $f(x)$ на концах интервала $[a, b]$ имела разные знаки $f(a) \cdot f(b) < 0$;
- первая и вторая производные $f'(x)$ и $f''(x)$ сохраняют определенный знак на всем интервале $[a, b]$.

Метод хорд. Этот метод отличается от метода дихотомии тем, что очередное приближение берем не в середине отрезка, а в точке пересечения с осью X (рис. 18) прямой, соединяющей точки $(a, f(a))$ и $(b, f(b))$.

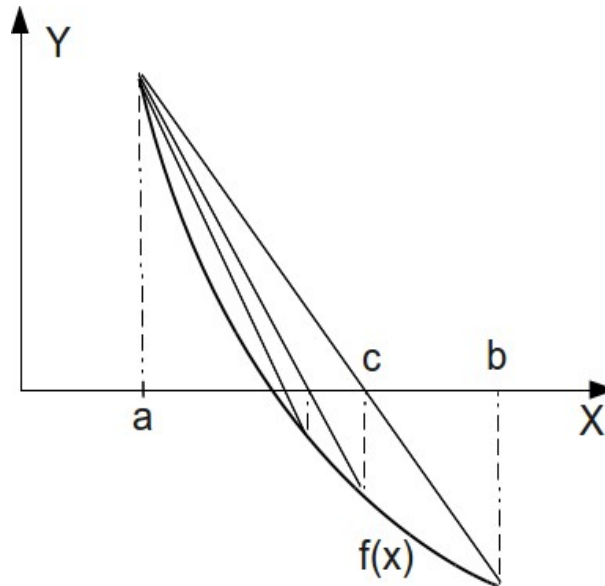


Рисунок 18. Графическая интерпретация метода хорд

Запишем уравнение прямой, проходящей через точки с координатами $(a, f(a))$ и $(b, f(b))$:

$$\frac{y-f(a)}{f(b)-f(a)} = \frac{x-a}{b-a} \quad y = \frac{f(b)-f(a)}{b-a} \cdot (x-a) + f(a) \quad (6.1.2)$$

Прямая, заданная уравнением (6.1.2), пересекает ось X при условии $y=0$. Найдем точку пересечения хорды с осью X :

$$y = \frac{f(b)-f(a)}{b-a} \cdot (x-a) + f(a) \quad , \quad x = a - \frac{f(a) \cdot (b-a)}{f(b)-f(a)} \quad ,$$

итак,
$$c = a - \frac{f(a)}{f(b)-f(a)}(b-a) \quad .$$

Далее необходимо вычислить значение функции в точке c . Это и будет корень уравнения.

Для вычисления одного из корней уравнения $x^2 - \cos(5 \cdot x) = 0$ методом хорд достаточно знать интервал изоляции корня, например, $a=0.2; b=0.4$ и точность вычисления $\varepsilon = 10^{-3}$. Блок-схема метода представлена на рис. 19.

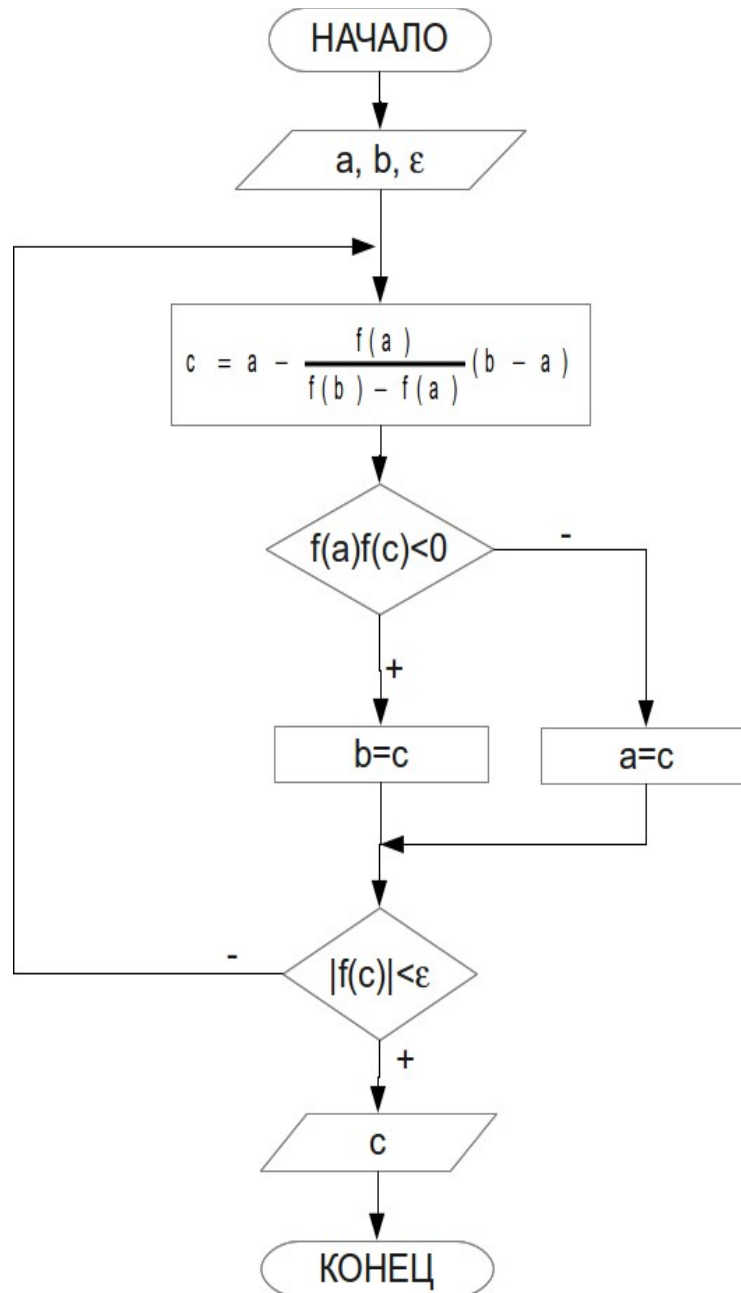


Рисунок 19. Алгоритм метода хорд

Метод касательных (метод Ньютона). В одной из точек интервала $[a; b]$, пусть это будет точка c , проведем касательную (рис. 20). Запишем уравнение этой прямой:

$$y = k \cdot x + m \quad (6.1.3)$$

Так как эта прямая является касательной, и она проходит через точку $(c, f(c))$, то $k = f'(c)$. Следовательно,

$$y = f'(c) \cdot x + m, \quad f(c) = f'(c) \cdot c + m, \quad m = f(c) - c \cdot f'(c),$$

$$y = f'(c) \cdot x + f(c) - c \cdot f'(c), \quad y = f'(c) \cdot (x - c) + f(c).$$

Найдем точку пересечения касательной с осью X :

$$f'(c) \cdot (x - c) + f(c) = 0, \quad x = c - \frac{f(c)}{f'(c)}$$

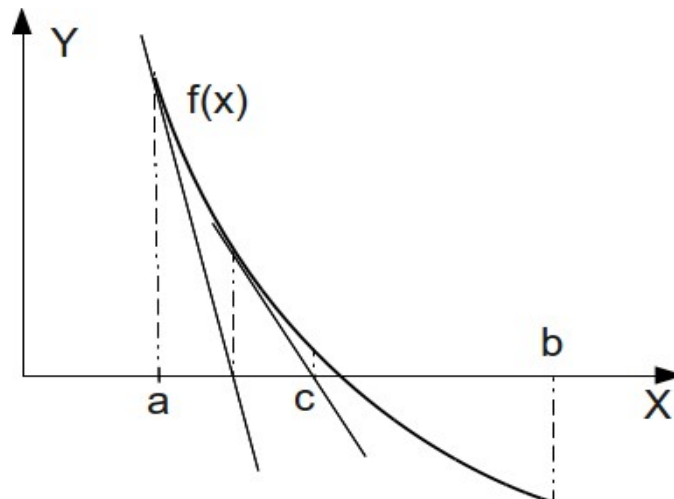


Рисунок 20. Графическая интерпретация метода касательных

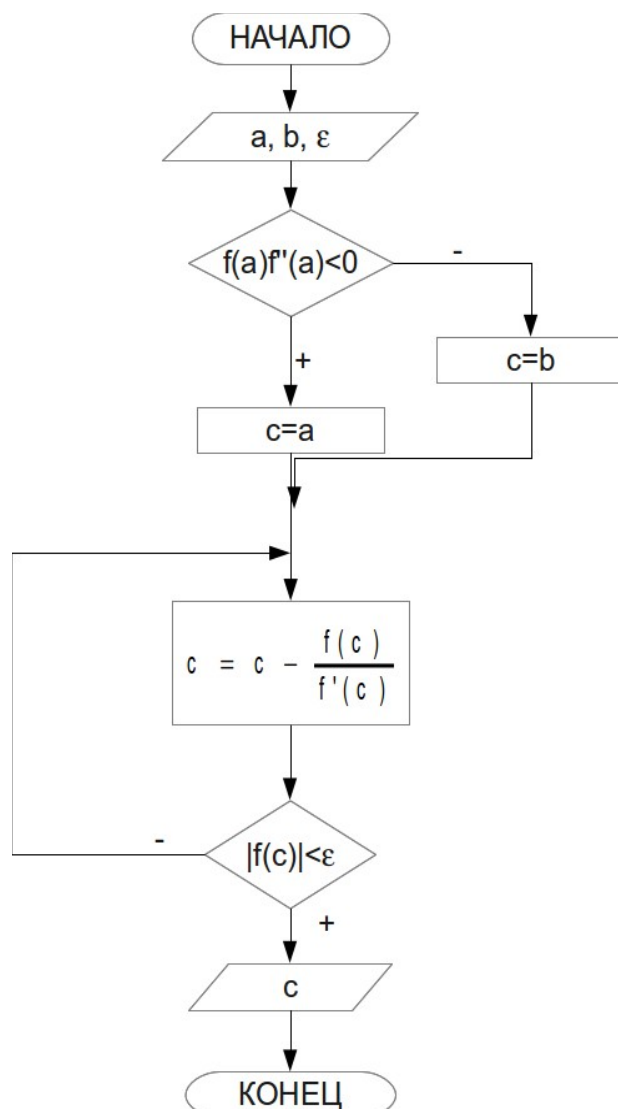


Рисунок 21. Алгоритм метода Ньютона

Если $|f(x)| < \varepsilon$, то точность достигнута, и точка x — решение; иначе необходимо переменной c присвоить значение x и провести касательную через новую точку c ; так продолжать до тех пор, пока $|f(x)|$ не станет меньше ε .

Осталось решить вопрос, что выбрать в качестве точки начального приближения c .

В этой точке должны совпадать знаки функции и ее второй производной. А так как нами было сделано допущение, что вторая и первая производные не меняют знак, то можно проверить условие $f(x) \cdot f'(x) > 0$ на обоих концах интервала и в качестве начального приближения взять ту точку, где это условие выполняется.

Здесь, как и в предыдущих методах, для вычисления одного из корней уравнения $x^2 - \cos(5 \cdot x) = 0$ достаточно знать интервал изоляции корня, например, $a = 0.2$; $b = 0.4$ и точность вычисления $\varepsilon = 10^{-3}$.

Блок-схема метода Ньютона представлена на рис. 21. Понятно, что для реализации этого алгоритма нужно найти первую и вторую производные

функции $f(x) = x^2 - \cos(5 \cdot x)$: $f'(x) = 2 \cdot x + 5 \cdot \sin(5 \cdot x)$,
 $f''(x) = 2 + 25 \cdot \cos(5 \cdot x)$.

Метод простой итерации. Для решения уравнения этим методом необходимо записать уравнение (6.1.1) в виде $x = \phi(x)$, задать начальное приближение $x_0 \in [a; b]$ и организовать следующий итерационный вычислительный процесс

$$x_{k+1} = \phi(x_k), k=0, 1, 2, \dots$$

Вычисление прекратить, если $|x_{k+1} - x_k| < \varepsilon$ (ε - точность).

Если неравенство $|\phi(x)| < 1$ выполняется на всем интервале $[a; b]$, то последовательность $x_0, x_1, x_2, \dots, x_n, \dots$ сводится к решению x^* (т.е. $\lim_{k \rightarrow \infty} x_k = x^*$).

Значение функции $\phi(x)$ должно удовлетворять условию $|\phi'(x)| < 1$ для того, чтобы можно было применить метод простых итераций. Условие $|\phi'(x)| < 1$ является *достаточным условием сходимости* метода простой итерации.

Уравнение (6.1.1) можно привести к виду $x = \phi(x)$ следующим образом. Умножить обе части уравнения $f(x) = 0$ на число λ . К обеим частям уравнения $\lambda \cdot f(x) = 0$ добавить число x . Получим $x = x + \lambda \cdot f(x)$. Это и есть уравнение вида $x = \phi(x)$, где

$$\phi(x) = x + \lambda \cdot f(x) . \quad (6.1.4)$$

Так как, необходимо, чтобы выполнялось неравенство $|\phi'(x)| < 1$ на интервале $[a; b]$, следовательно, $|\phi'(x)| = |1 + \lambda \cdot f'(x)|$ и $|1 + \lambda \cdot f'(x)| < 1$, а значит, с помощью *подбора параметра* λ можно добиться выполнения *условия сходимости*.

Для вычисления корней уравнения $x^2 - \cos(5 \cdot x) = 0$ воспользуемся графическим решением (рис. 15) и определим начальное значение одного из корней. Пусть $x_0 = 0.2$. Подберем значение λ решив неравенство $|1 + \lambda \cdot f'(x)| < 1$:

$$\begin{aligned} f(x) &= x^2 - \cos(5 \cdot x), \\ f'(x) &= 2 \cdot x + 5 \cdot \sin(5 \cdot x), \\ f'(x_0) &= 2 \cdot 0.2 + 5 \cdot \sin(5 \cdot 0.2) \approx 4.6 . \\ |1 + \lambda \cdot 4.6| &< 1 \Rightarrow \\ \Rightarrow 1 + 4.6 \cdot \lambda &< 1 \Rightarrow \lambda < 0 \Rightarrow \\ 1 + 4.6 \cdot \lambda &> -1 \Rightarrow \lambda > -0.4 \Rightarrow \\ \Rightarrow \lambda &\in (-0.4; 0) . \end{aligned}$$

Таким образом, исходными данными для программы будут начальное значение корня уравнения $x_0 = 0.2$, значение параметра λ , пусть $\lambda = -0.2$, и точность вычислений $\varepsilon = 0.001$. Для вычисления второго корня заданного уравнения параметр λ подбирают аналогично.

Блок-схема метода простой итерации приведена на рис. 22.

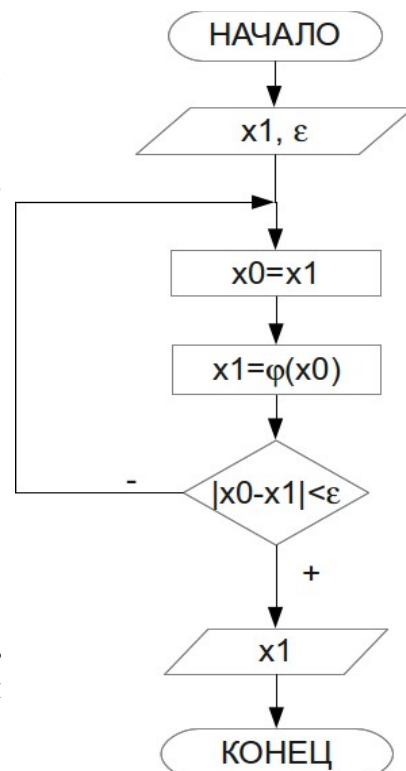


Рисунок 22. Алгоритм метода простой итерации

Далее представлен текст программы, реализующий решение задачи 15.

Результаты работы программы показаны на рис. 23.

```
#include <iostream>
#include <math.h>
using namespace std;
double f(double x)          //Функция, определяющая
левую                        //часть уравнения  $f(x)=0$ .
{
    return(x*x-cos(5*x));
}
//Функция, реализующая метод половинного деления.
int Dichotomy(double a, double b, double *c,
double eps)
{int k=0;
    do
    {
        *c=(a+b)/2;
        if (f(*c)*f(a)<0) b=*c;else a=*c;    k++;
    }
    while (fabs(a-b)>=eps);
return k;
}
//Функция, реализующая метод хорд.
int Chord(double a, double b, double *c, double
eps)
{int k=0;
    do
    {
        *c=a-f(a)/(f(b)-f(a))*(b-a);
        if (f(*c)*f(a)>0) a=*c; else b=*c;
        k++;
    }
    while (fabs(f(*c))>=eps);
return k;
}
double f1(double x)    //Первая производная функции
f(x).
{
    return(2*x+5*sin(5*x));
}
double f2(double x)    //Вторая производная функции
f(x).
{
    return(2+25*cos(5*x));
}
```

```

}
//Функция, реализующая метод касательных.
int Tangent(double a, double b, double *c, double
eps)
{int k=0;
  if (f(a)*f2(a)>0) *c=a;
  else *c=b;
  do
  {
    *c=*c-f(*c)/f1(*c);
    k++;
  }
  while (fabs(f(*c))>=eps);
return k;
}
double fi(double x,double L)          //Функция ,
заданная                             //выражением (1.4).

{
  return(x+L*f(x));
}
//Функция, реализующая метод простой итерации.
int Iteration(double *x, double L, double eps)
{int k=0; double x0;
  do
  {
    x0=*x;
    *x=fi(x0,L);
    k++;
  }
  while (fabs(x0-*x)>=eps);
return k;
}
int main()
{
double A, B, X, P;
double ep=0.001;          //Точность вычислений.
int K;
cout<<"a=";cin>>A;          //Интервал изоляции
корня.
cout<<"b=";cin>>B;
cout<<"Решение уравнения x^2-cos(5*x)=0."<<endl;
cout<<"Метод половинного деления:"<<endl;
K=Dichotomy(A,B,&X,ep);
cout<<"Решение уравнения x="<<X;
cout<<" , количество итераций k="<<K<<endl;

```

```

cout<<"Метод хорд:"<<endl;
K=Chord(A,B,&X,ep);
cout<<"Решение уравнения x="<<X;
cout<<", количество итераций k="<<K<<endl;
cout<<"Метод касательных:"<<endl;
K=Tangent(A,B,&X,ep);
cout<<"Решение уравнения x="<<X;
cout<<", количество итераций k="<<K<<endl;
cout<<"Метод простых итераций:"<<endl;
X=A;
cout<<"L=";cin>>P;
K=Iteration(&X,P,ep);
cout<<"Решение уравнения x="<<X;
cout<<", количество итераций k="<<K<<endl;
return 0;
}

```

```

a=0.2
b=0.4
Решение уравнения x^2-cos(5*x)=0.
Метод половинного деления:
Решение уравнения x=0.296094, количество итераций k=8
Метод хорд:
Решение уравнения x=0.296546, количество итераций k=2
Метод касательных:
Решение уравнения x=0.296556, количество итераций k=2
Метод простых итераций:
L=-0.2
Решение уравнения x=0.296595, количество итераций k=3

-----
(program exited with code: 0)
Press return to continue

```

Рисунок 23. Результат работы программы к задаче 15

Если рассмотреть внимательно представленный код, то можно увидеть, что программа жёстко завязана на функцию $x^2 - \cos(5x)$, её первую и вторую производные, а также функцию $\phi(x) = x + \lambda \cdot f(x)$. При изменении конкретного решаемого уравнения придётся менять текст программы, а не только вводимые величины a , b , λ . Можно написать более универсальную программу, но для этого придётся передавать функцию $f(x)$ и её производные, как параметры.

При решении этой и подобных задач возникает необходимость передавать имя функции, как параметр. В этом случае формальным параметром является указатель на передаваемую функцию. В общем виде прототип указателя на функцию можно записать так.

```
type (*name_f)(type1, type2, type3,...)
```

Здесь

`name_f` – имя функции

`type` – тип возвращаемый функцией,

`type1, type2, type3,...` – типы формальных параметров функции.

```
#include <iostream>
```

```

#include <math.h>
using namespace std;
double f(double x) //Функция, определяющая левую
                    //часть уравнения  $f(x)=0$ .
{
    return(x*x-cos(5*x));
}
double f1(double x) //Первая производная функции  $f(x)$ .
{
    return(2*x+5*sin(5*x));
}
double f2(double x) //Вторая производная функции
 $f(x)$ .
{
    return(2+25*cos(5*x));
}
double fi(double x,double L) //Функция ,
заданная //выражением (4.4).
{
    return(x+L*f(x));
}
//Функция, реализующая метод простой итерации.
int Iteration(double *x, double L, double eps, double
(*fi_)(double,double))
{int k=0; double x0;
    do
    {
        x0=*x;
        *x=fi_(x0,L);
        k++;
    }
    while (fabs(x0-*x)>=eps);
return k;
}
//Функция, реализующая метод половинного деления.
int Dichotomy(double a, double b, double *c, double
eps, double (*f_)(double))
{int k=0;
    do
    {
        *c=(a+b)/2;
        if (f_(*c)*f_(a)<0) b=*c;
        else a=*c;
        k++;
    }
    while (fabs(a-b)>=eps);
}

```

```

return k;
}

//Функция, реализующая метод хорд.
int Chord(double a, double b, double *c, double eps,
double (*f_)(double))
{int k=0;
  do
  {
    *c=a-f_(a)/(f_(b)-f_(a))*(b-a);
    if (f_(a)*f_(c)>0) a=*c;
    else b=*c;
    k++;
  }
  while (fabs(f_(c))>=eps);
return k;
}

```

```

//Функция, реализующая метод касательных.
int Tangent(double a, double b, double *c, double
eps,double(*f_)(double),double(*f1_)(double),double(*f2_)(double))
{int k=0;
  if (f_(a)*f2_(a)>0) *c=a;
  else *c=b;
  do
  {
    *c=*c-f_(c)/f1_(c);
    k++;
  }
  while (fabs(f_(c))>=eps);
return k;
}

```

```

int main()
{
double A, B, X, P;
double ep=0.001;           //Точность вычислений.
int K;
cout<<"a=";<<cin>>A;       //Интервал изоляции корня.
cout<<"b=";<<cin>>B;
cout<<"Решение уравнения x^2-cos(5*x)=0."<<endl;
cout<<"Метод половинного деления:"<<endl;
K=Dichotomy(A,B,&X,ep,f);
cout<<"Решение уравнения x="<<X;
}

```



```

cout<<" , количество итераций k="<<K<<endl;
cout<<"Метод хорд:"<<endl;
K=Chord(A,B,&X,er,f);
cout<<"Решение уравнения x="<<X;
cout<<" , количество итераций k="<<K<<endl;
cout<<"Метод касательных:"<<endl;
K=Tangent(A,B,&X,er,f,f1,f2);
cout<<"Решение уравнения x="<<X;
cout<<" , количество итераций k="<<K<<endl;
cout<<"Метод простых итераций:"<<endl;
X=A;
cout<<"L=";cin>>P;
K=Iteration(&X,P,er,fi);
cout<<"Решение уравнения x="<<X;
cout<<" , количество итераций k="<<K<<endl;
return 0;
}

```

6.6.2 Вычисление интеграла

ЗАДАЧА 16. Вычислить $\int_a^b f(x)dx$ методами Гаусса и Чебышева.

Кратко напомним методы численного интегрирования.

Метод Гаусса состоит в следующем. Определённый интеграл непрерывной функции на интервале от -1 до 1 можно заменить суммой и вычислить по формуле $\int_{-1}^1 f(x)dx = \sum_{i=1}^n A_i f(t_i)$, t_i – точки из интервала [-1,1], A_i – рассчитываемые коэффициенты. Методика определения A_i , t_i представлена в [3]. Для практического использования значения коэффициентов при $n=2,3,4,5,6,7,8$ представлены в табл. 5.

Таблица 5. Значения коэффициентов в квадратурной формуле Гаусса

n	Массив t	Массив A
2	-0.57735027, 0.57735027	1 1
3	-0.77459667, 0, 0.77459667	5/9, 8/9, 5/9
4	-0.86113631, -0.33998104, 0.33998104, 0.86113631	0.34785484, 0.65214516, 0.65214516, 0.34785484
5	-0.90617985, -0.53846931, 0, 0.53846931, 0.90617985	0.23692688, 0.47862868, 0.56888889, 0.47862868, 0.23692688
6	-0.93246951, -0.66120939, -0.23861919, 0.23861919, 0.66120939, 0.93246951	0.17132450, 0.36076158, 0.46791394, 0.46791394, 0.36076158, 0.17132450
7	-0.94910791, -0.74153119, -0.40584515, 0, 0.40584515, 0.74153119, 0.94910791	0.12948496, 0.27970540, 0.38183006, 0.41795918, 0.38183006, 0.27970540, 0.12948496
8	-0.96028986, -0.79666648, -0.52553242, -0.18343464, 0.18343464, 0.52553242, 0.79666648, 0.96028986	0.10122854, 0.22238104, 0.31370664, 0.36268378, 0.36268378, 0.31370664, 0.22238104, 0.10122854

Для вычисления интеграла непрерывной функции на интервале от а до b квадратурная формула Гаусса может быть записана следующим образом $\int_a^b f(x)dx = \frac{b-a}{2} \sum_{i=1}^n A_i f\left(\frac{b+a}{2} + \frac{b-a}{2} t_i\right)$, значения коэффициентов A_i и t_i приведены в табл. 5.

При использовании квадратурной формулы Чебышева определённый интеграл непрерывной функции на интервале от -1 до 1 записывается в виде следующей формулы $\int_{-1}^1 f(x)dx = \frac{2}{n} \sum_{i=1}^n f(t_i)$, t_i – точки из интервала [-1,1]. Формула Чебышева для вычисления интеграла на интервале от а до b может быть записана так $\int_a^b f(x)dx = \frac{b-a}{n} \sum_{i=1}^n f\left(\frac{b+a}{2} + \frac{b-a}{2} t_i\right)$. Рассмотренные формулы имеют смысл при $n=2,3,4,5,6,7,9$, коэффициенты t_i представлены в табл. 6.

Таблица 6. Значения коэффициентов в квадратурной формуле Чебышева

n	Массив t
2	-0.577350, 0.577350
3	-0.707107, 0, -0.707107
4	-0.794654, -0.187592, 0.187592, 0.794654
5	-0.832498, -0.374541, 0, 0.374541, 0.832498
6	-0.866247, -0.422519, -0.266635, 0.266635, 0.422519, 0.866247
7	-0.883862, -0.529657, -0.323912, 0, 0.323912, 0.529657, 0.883862
9	-0.911589, -0.601019, -0.528762, -0.167906, 0, 0.167906, 0.528762, 0.601019, 0.911589

Осталось написать функции вычисления определённого интеграла $\int_a^b f(x)dx$ методами Гаусса и Чебышева. Далее приведены тексты функций и функция main(). В качестве тестовых использовались интегралы $\int_0^2 \sin^4 x dx \approx 0.9701$, $\int_5^{13} \sqrt{2x-1} dx \approx 32.667$.

```
#include <stdio.h>
#include <math.h>
//Функция вычисления определённого интеграла методом
//Чебышева. (a,b) - интервал интегрирования, *fn -
//указатель на функцию типа double f (double).
double int_chebishev(double a, double b, double (*fn)
(double))
{
    int i,n=9;
    double s,
    t[9]={-0.911589, -0.601019, -0.528762, -0.167906, 0,
0.167906, 0.528762, 0.601019, 0.911589};
    for(s=i=0;i<n;i++)
        s+=fn((b+a)/2+(b-a)/2*t[i]);
    s*=(b-a)/n;
    return s;
}
//Функция вычисления определённого интеграла методом
// Гаусса. (a,b) - интервал интегрирования, *fn -
// указатель на функцию типа double f (double)
double int_gauss(double a, double b, double (*fn)(double))
{
    int i,n=8;
    double s,
    t[8]={-0.96028986, -0.79666648, -0.52553242, -
0.18343464, 0.18343464, 0.52553242, 0.79666648,
0.96028986},
    A[8]={0.10122854, 0.22238104, 0.31370664, 0.36268378,
0.36268378, 0.31370664, 0.22238104, 0.10122854};
    for(s=i=0;i<n;i++)
        s+=A[i]*fn((b+a)/2+(b-a)/2*t[i]);
    s*=(b-a)/2;
```

```

        return s;
    }
    //Функции f1 и f2 типа double f (double), указатели на
    // которые будут передаваться в int_gauss и int_chebishev.
    double f1(double y)
    {
        return sin(y)*sin(y)*sin(y)*sin(y);
    }
    double f2(double y)
    {
        return pow(2*y-1,0.5);
    }

    int main(int argc, char **argv)
    {
        double a,b;
        printf("Интеграл sin(x)^4=\n");
        printf("Введите интервал интегрирования\n");
        scanf("%lf%lf",&a,&b);
        //Вызов функции int_gauss(a, b, f1), f1 - имя функции,
        // интеграл от которой надо посчитать.
        printf("Метод Гаусса:%lf\n",int_gauss(a, b, f1));
        //Вызов функции int_chebishev(a, b, f1), f1 - имя функции,
        // интеграл от которой надо посчитать.
        printf("Метод Чебышева:%lf\n",int_chebishev(a,b,f1));
        printf("Интеграл sqrt(2*x-1)=\n");
        printf("Введите интервалы интегрирования\n");
        scanf("%lf%lf",&a,&b);
        //Вызов функции int_gauss(a, b, f2), f2 - имя функции,
        // интеграл от которой надо посчитать.
        printf("Метод Гаусса:%lf\n",int_gauss(a, b, f2));
        //Вызов функции int_chebishev(a, b, f2), f2 - имя функции,
        // интеграл от которой надо посчитать.
        printf("Метод Чебышева:%lf\n",int_chebishev(a,b,f2));
        return 0;
    }

```

Результаты работы программы приведены ниже

Интеграл sin(x)^4=

Введите интервалы интегрирования

0 2

Метод Гаусса:0.970118

Метод Чебышева:0.970082

Интеграл sqrt(2*x-1)=

Введите интервалы интегрирования

5 13

Метод Гаусса:32.6667

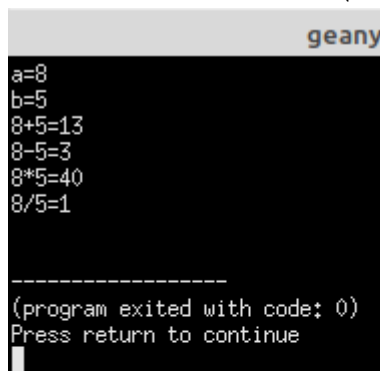
Метод Чебышева:32.6667

6.6.3 Арифметическая задача

Пример. Несложная задача. Имеются функции сложения, вычитания, умножения и деления целых чисел. Вводятся целые числа a,b. С использованием имеющихся функций вычислить сумму, разность, произведения и частное этих чисел.

```
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    int plus(int,int);
    int minus(int,int);
    int mul(int,int);
    int divide(int,int);
    int a,b;
    char s[5]="+-*/";
    int (*y[4])(int,int)={plus,minus,mul,divide};
    cout<<"a=";cin>>a;
    cout<<"b=";cin>>b;
    for(int i=0;i<4;i++)
        cout<<a<<s[i]<<b<<"="<<y[i](a,b)<<endl;
    return 0;
}

int plus(int m,int n){return m+n;};
int minus(int m,int n){return m-n;};
int mul(int m,int n){return m*n;};
int divide(int m,int n){return m/n;};
```



```
geany
a=8
b=5
8+5=13
8-5=3
8*5=40
8/5=1

-----
(program exited with code: 0)
Press return to continue
```

Встроенная функция быстрой сортировки `qsort` (по мотивам <https://calmsen.ru/funkcziya-qsort/>, <https://learn.microsoft.com/ru-ru/cpp/c-runtime-library/reference/qsort?view=msvc-160&viewFallbackFrom=vs-2017>).

Функция определена в заголовочном файле `<cstdlib>`.

Функция `qsort()` в C++ использует функцию сравнения, чтобы решить, какой элемент меньше/больше другого.

Прототип

```
void qsort (void* base, size_t num, size_t size, int
(*compare)(const void*,const void*));
```

Источник: <https://calmsen.ru/funkcziya-qsort/> ;

Параметры

- `base` — указатель на массив;
- `num` — размер массива в элементах.
- `size` — размер элемента массива в байтах.
- `compare` — указатель на пользовательскую функцию, которая сравнивает два элемента массива

```
int compare(const void* a, const void* b);
```

Функция сравнивает элементы и возвращает одно из следующих значений.

Сравнение возвращаемого значения функции	Описание
<code>< 0</code>	<code>elem1</code> меньше <code>elem2</code>
<code>0</code>	<code>elem1</code> эквивалентен <code>elem2</code>
<code>> 0</code>	<code>elem1</code> больше <code>elem2</code>

Пример использования

```
#include <iostream>
#include <cstdlib>
using namespace std;
int compare(const void* a, const void* b)
{   const int* x = (int*) a;
    const int* y = (int*) b;
    if (*x > *y)
        return 1;
    else if (*x < *y)
        return -1;
    return 0;}
int main()
{   const int num = 10; int i;
    int arr[num] = {9,4,19,2,7,9,5,15,23,3};
    cout << "До сортировки" << endl;
    for (i=0; i<num; i++)
        cout << arr[i] << " ";
    qsort(arr,num,sizeof(int),compare);
    cout << endl << endl;
    cout << "После сортировки" << endl;
    for(i=0;i<num;i++)
        cout << arr[i] << " ";
    return 0; }
```

Варианты функции compare

```
int compare(const void* a, const void* b)
{   const int* x = (int*) a;
    const int* y = (int*) b;
    return *x - *y;}
```

```
int compare(const void* a, const void* b)
{   return *(int*) a - *(int*)b;}
```

Если вы хотите сортировать по убыванию, то поменяйте знак у возвращаемого функцией compare значения.

Если тип данных вашего массива не int, замените тип данных везде.