

«Разработка, отладка и испытание алгоритмов и программ с использованием внутренней сортировки и меню пользователя»

Пример программы с 5-ю видами сортировок («пузырьковой», выбором минимального, вставками, Хоара и Шелла):

```

1  #include <iostream>
2  #include <Windows.h>
3  using namespace std;
4
5  void bubbleSort(int*, int); //прототип функции сортировки "пузырьком" по ВОЗРАСТАНИЮ
6  void print(int*, int); //прототип функции печати содержимого элементов одномерного целочисленного массива на консоль
7  void printStr(char*, int); //прототип функции печати содержимого элементов одномерного символьного массива на консоль
8  void swap(int&, int&); //прототип функции обмена значениями между двумя элементами
9  int minimumFrom(int*, int, int); //прототип функции для нахождения индекса (номера) элемента с минимальным значением на участке массива
10 void selectionSort(int*, int); //прототип функции сортировки целочисленного одномерного массива выбором наименьшего элемента
11 void insertSort(char*, int); //прототип функции сортировки по ВОЗРАСТАНИЮ одномерного символьного массива методом вставок
12 void quickSort(int*, int, int); //прототип функции быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
13 void shellSort(int*, int); //прототип функции сортировки методом Шелла
14
15 int main()
16 {
17     SetConsoleOutputCP(1251);
18     SetConsoleCP(1251);
19     const int size = 10;
20     int ar[size] = { 9,0,8,1,7,2,6,3,5,4 };
21     print(ar, size);
22     bubbleSort(ar, size);
23     cout << "Печать отсортированного массива методом ""пузырька"" по ВОЗРАСТАНИЮ:\n";
24     print(ar, size);
25     int mas[size] = { 9,8,7,6,5,4,3,2,1,0 }; //{ 9,0,8,1,7,2,6,3,5,4 };
26     print(mas, size);
27     selectionSort(mas, size);
28     cout << "Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:\n";
29     print(mas, size);
30     char s[size] = { 'j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a' };
31     printStr(s, size);
32     insertSort(s, size);
33     cout << "Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:\n";
34     printStr(s, size);
35     int vector[size] = { 9,0,8,1,7,2,6,3,5,4 };
36     print(vector, size);
37     quickSort(vector, 0, size-1);
38
39     cout << "Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:\n";
40     print(vector, size);
41     int vect[size] = { 9,8,7,6,5,4,3,2,1,0 };
42     print(vect, size);
43     shellSort(vect, size);
44     cout << "Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:\n";
45     print(vect, size);
46     system("pause");
47     return 0;
48 }
49
50 void bubbleSort(int* m, int n) //функция сортировки одномерного целочисленного массива методом "пузырька" по ВОЗРАСТАНИЮ
51 {
52     for (int i = 0; i < n - 1; i++)
53     {
54         for (int j = 0; j < n - 1 - i; j++)
55         {
56             if (m[j] > m[j + 1]) //для сортировки по ВОЗРАСТАНИЮ
57             {
58                 int t = m[j];
59                 m[j] = m[j + 1];
60                 m[j + 1] = t;
61             }
62         }
63     }
64 }
65
66 void print(int* m, int n) //функция печати содержимого элементов одномерного целочисленного массива на консоль
67 {
68     for (int i = 0; i < n; i++)
69     {
70         cout << m[i] << ' ';
71     }
72     cout << endl;
73 }
74
75 void printStr(char* m, int n) //функция печати содержимого элементов одномерного символьного массива на консоль
76 {
77     for (int i = 0; i < n; i++)
78     {
79         cout << m[i] << ' ';
80     }
81     cout << endl;
82 }

```

```

75     cout << m[i] << ' ';
76 }
77 cout << endl;
78 }
79 void swap(int& first, int& second)//функция обмена значениями между двумя элементами. Чтобы обменять значения у ОРИГИНАЛОВ значений в main'e, эта функция принимает ссылки на
80 //элементы, то есть константные указатели на адреса элементов
81     int t = first;
82     first = second;
83     second = t;
84 }
85 int minimumFrom(int* m, int position, int length)//функция для нахождения индекса (номера) элемента с минимальным значением на участке массива, начиная с элемента с индексом
86 //position до элемента с индексом length-1. Эта функция нужна для функции selectionSort()
87     int minIndex = position;
88     for (int i = position + 1; i < length; i++)
89     {
90         if (m[i] < m[minIndex])
91         {
92             minIndex = i;
93         }
94     }
95     return minIndex;
96 }
97 void selectionSort(int* m, int length)//функция сортировки по ВОЗРАСТАНИЮ целочисленного одномерного массива выбором наименьшего элемента
98 {
99     for (int i = 0; i < length - 1; i++)
100     {
101         swap(m[i], m[minimumFrom(m, i, length)]);
102     }
103 }
104 void insertSort(char* m, int n)//функция сортировки по ВОЗРАСТАНИЮ одномерного символьного массива методом вставок. Этот алгоритм сортировки обладает свойством устойчивости, то
105 //есть при совпадении значений в ключевом поле объекты НЕ меняются местами
106     for (int i = 1; i < n; i++)
107     {
108         char t = m[i];
109         for (int j = i - 1; j > -1 & t < m[j]; j--)
110         {
111             m[j + 1] = m[j];
112             m[j] = t;
113         }
114     }
115 }
116 void quickSort(int* m, int left, int right)//функция быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
117 {
118     int leftArrow = left, rightArrow = right, pivot = m[(left + right) / 2];
119     do
120     {
121         while (m[rightArrow] > pivot)
122         {
123             rightArrow--;
124         }
125         while (m[leftArrow] < pivot)
126         {
127             leftArrow++;
128         }
129         if (leftArrow <= rightArrow)
130         {
131             swap(m[leftArrow], m[rightArrow]);
132             leftArrow++;
133             rightArrow--;
134         }
135     }
136     while (rightArrow >= leftArrow);
137     if (left < rightArrow)
138     {
139         quickSort(m, left, rightArrow);
140     }
141     if (leftArrow < right)
142     {
143         quickSort(m, leftArrow, right);
144     }
145 }
146 void shellSort(int* m, int n)//функция сортировки методом Шелла. Это неустойчивая сортировка
147 {
148     for (int i = n / 2; i > 0; i = i / 2)

```

```
149 {
150     for (int j = 0; j < n - i; j++)
151     {
152         for (int k = j; k > -1; k = k - i)
153         {
154             if (m[k] > m[k + i])
155             {
156                 int t = m[k];
157                 m[k] = m[k + i];
158                 m[k + i] = t;
159             }
160             else
161             {
162                 k = 0;
163             }
164         }
165     }
166 }
167 }
```

```
D:\2019\Lab5\Debug\Lab5_4.exe
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом пузырька по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:
a b c d e f g h i j
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
Для продолжения нажмите любую клавишу . . .
```

Алгоритмы поиска и сортировки

При любой сортировке происходит сравнение величин (полей сортировки).

Массив чисел сортируется по принципу сравнения: если надо отсортировать по возрастанию (неубыванию), то сравниваем значения соседних элементов, и если левый элемент по хранимому в нем значению больше правого, то хранимые в них значения надо обменять местами; если массив надо отсортировать по убыванию (невозрастанию), то сравниваем значения соседних элементов, и если леворасположенный элемент хранит в себе значение меньше, чем в праворасположенном элементе, то хранимые в них значения надо обменять местами. При более сложных алгоритмах сортировок сравниваться могут не соседние элементы массива, а имеющие между собой другие «промежуточные» элементы, но все равно эти удаленные друг от друга элементы сравниваются и учитывается, какой элемент расположен левее, а какой – правее. Возможно оперирование и сразу последовательностями элементов (группы расположенных рядом элементов, удовлетворяющих критерию сортировки, переставляются с другими группами элементов).

А если сортируются строки? Что больше – "бао" или "баобаб", "12" или "2"? Ответ: строки сравниваются посимвольно до тех пор, пока не будут найдены разные символы или конец строки (ноль-терминатор). Конец строки считается меньше любого символа. Поэтому: "бао" < "баобаб", "12" < "2".

Кроме того, сравнение может проходить по ограниченному количеству символов (например, только по первым трем). Длина ключа сортировки – 3 символа. Тогда оставшая часть строки в расчет не принимается:

"бао" = "баобаб" "222" = "22299"

Если сортируются структуры, сортировка выполняется по одному или нескольким полям – ключу сортировки.

struct S1

```
{
    int nomer;
    double summa;
    char nominal[10];
};
```

Может быть ключом поле nomer, или summa, или nomer и summa (первый ключ, второй ключ; сначала сортируются значения по ПЕРВОМУ ключу, а в рамках группы элементов с одинаковым значением в поле первого ключа, эти элементы сортируются по второму ключу), или summa и 3 первых символа поля nominal.

Для определения оптимальности того или иного алгоритма сортировки вводятся понятия:

1) **время сортировки** (один и тот же массив сортируем разными методами и определяем наименьшее время), но при разных массивах время может сильно зависеть как от метода, так и от конкретных данных в массиве и его размера. Массив может быть уже отсортирован, а наша сортировка начнет переставлять какие-то элементы, массив может быть частично отсортирован (несколько элементов стоят не на своих местах) – а сортировка работает столь же долго, массив может быть отсортирован в обратном порядке (это самый сложный для сортировки случай, на котором и нужно изначально тестировать любую сортировку, чтобы убедиться, что она дает правильный результат на самых «неудобных» для нее данных) – обычно здесь все алгоритмы показывают примерно одинаковое время.

2) **естественность метода** – если массив уже отсортирован (хотя бы частично), то его сортировка работает быстрее и, стало быть, раньше завершается.

3) **устойчивость метода**: если сортировка идет не по полному ключу (часть строки или структуры) – данные не меняют свой первоначальный порядок:

```
107210 Петров
107210 Якимото
107220 Данилов
107220 Петров
```

Пузырьковая сортировка

Это наиболее простой (но не самый быстрый) способ сортировки.

Он назван так потому, что с каждым циклом самый легкий элемент поднимается вверх списка, подобно пузырьку.

a	
6	a[0]
4	a[1]
8	a[2]
10	a[3]
1	a[4]

В первом проходе сравниваются попарно элементы: первый – второй, второй – третий, третий – четвертый и т.д. до n-2-го и n-1-го элементов. Каждый раз в паре самый легкий перемещается вверх (левее).

В результате одного прохода самый тяжелый элемент опустится на дно:

4 – 6 – 8 – 1 – 10

4 – 6 – 1 – 8 – 10

4 – 1 – 6 – 8 – 10

1 – 4 – 6 – 8 – 10

Самое простое – повторить цикл n-1 раз.

Цикл повторяется n-1 раз поскольку количество пар и, стало быть, количество сравнений равно n-1 (во внутреннем цикле).

Заканчивать сравнения лучше каждый раз на 1 элемент раньше (левее), ведь самый «легкий» элемент занимает свое итоговое место справа массива и при каждом проходе внешнего цикла таких элементов становится на один больше. Сортировка пузырьковым методом, реализованная в функции `bubbleSort(имяМассива, размерМассива)`, которая вызывается для массива `ar` в `main`'е:

```
#include <iostream>
#include <Windows.h>
using namespace std;

void bubbleSort(int*, int); //прототип функции сортировки "пузырьком" по ВОЗРАСТАНИЮ
void print(int*, int); //прототип функции печати содержимого элементов одномерного целочисленного массива на консоль
void printStr(char*, int); //прототип функции печати содержимого элементов одномерного символьного массива на консоль
void swap(int&, int&); //прототип функции обмена значениями между двумя элементами
int minimumFrom(int*, int, int); //прототип функции для нахождения индекса (номера) элемента с минимальным значением на участке массива
void selectionSort(int*, int); //прототип функции сортировки целочисленного одномерного массива выбором наименьшего элемента
void insertSort(char*, int); //прототип функции сортировки по ВОЗРАСТАНИЮ одномерного символьного массива методом вставок
void quickSort(int*, int, int); //прототип функции быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
void shellSort(int*, int); //прототип функции сортировки методом Шелла

int main()
{
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    const int size = 10;
    int ar[size] = { 9,0,8,1,7,2,6,3,5,4 };
    print(ar, size);
    bubbleSort(ar, size);
    cout << "Печать отсортированного массива методом ""пузырька"" по ВОЗРАСТАНИЮ:\n";
    print(ar, size);
}
```

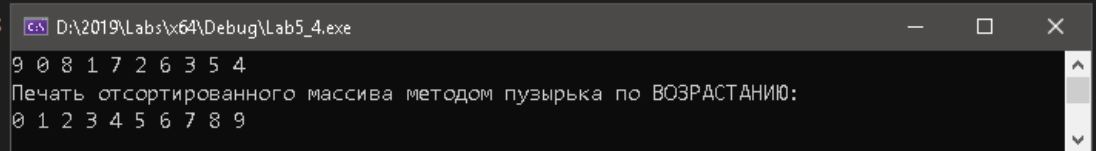
```

system("pause");
return 0;
}

void bubbleSort(int* m, int n)//функция сортировки одномерного целочисленного массива методом "пузырька" по ВОЗРАСТАНИЮ
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - 1 - i; j++)
        {
            if (m[j] > m[j + 1])//для сортировки по ВОЗРАСТАНИЮ
            {
                int t = m[j];
                m[j] = m[j + 1];
                m[j + 1] = t;
            }
        }
    }
}

void print(int* m, int n)//функция печати содержимого элементов одномерного целочисленного массива на консоль
{
    for (int i = 0; i < n; i++)
    {
        cout << m[i] << ' ';
    }
    cout << endl;
}

```



Сортировка выбором наименьшего элемента

Основные действия этого алгоритма для сортировки массива длиной L заключаются в следующем:

Для каждого значения индекса i выполнить два действия:

- 1) Среди элементов с индексами от i до $(L-1)$ найти элемент с минимальным значением.
- 2) Обменять значения i -го и минимального элементов.

Работу этого алгоритма рассмотрим на примере сортировки массива из 5 целых чисел:

int a[5];

Значения элементов неотсортированного массива показаны на рисунке 4.

6	a[0]
4	a[1]
8	a[2]
10	a[3]
1	a[4]

Рис. 4. – Начальное состояние массива до сортировки.

В процессе сортировки методом выбора наименьшего элемента массив будет последовательно переходить в состояния, показанные на рисунке 5 (слева направо). Каждое состояние получается из предыдущего путем перестановки двух элементов, помеченных на рисунке 5 кружками.

6	a[0]	1	a[0]	1	a[0]	1	a[0]	1	a[0]
4	a[1]	4	a[1]	4	a[1]	4	a[1]	4	a[1]
8	a[2]	8	a[2]	8	a[2]	6	a[2]	6	a[2]
10	a[3]	10	a[3]	10	a[3]	10	a[3]	8	a[3]
1	a[4]	6	a[4]	6	a[4]	8	a[4]	10	a[4]

Рис. 5. – Последовательные шаги сортировки массива методом выбора наименьшего элемента.

На Си++ алгоритм сортировки можно реализовать в виде трех функций. Функция высокого уровня будет называться "selectionSort(...)" (у нее два входных параметра – сортируемый массив и его длина). Сначала эта функция вызывает вспомогательную функцию "minimumFrom(array, position, length)", которая возвращает индекс минимального элемента массива "array", расположенного в диапазоне между индексом "position" и концом массива (то есть индексом length-1). Для обмена значений двух элементов массива вызывается функция "swap(адресЭлемента1, адресЭлемента2)".

```

#include <iostream>
#include <Windows.h>
using namespace std;

void bubbleSort(int*, int); //прототип функции сортировки "пузырьком" по ВОЗРАСТАНИЮ
void print(int*, int); //прототип функции печати содержимого элементов одномерного целочисленного массива на консоль
void printStr(char*, int); //прототип функции печати содержимого элементов одномерного символического массива на консоль
void swap(int&, int&); //прототип функции обмена значениями между двумя элементами
int minimumFrom(int*, int, int); //прототип функции для нахождения индекса (номера) элемента с минимальным значением на участке массива
void selectionSort(int*, int); //прототип функции сортировки целочисленного одномерного массива выбором наименьшего элемента
void insertSort(char*, int); //прототип функции сортировки по ВОЗРАСТАНИЮ одномерного символического массива методом вставок
void quickSort(int*, int, int); //прототип функции быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
void shellSort(int*, int); //прототип функции сортировки методом Шелла

int main()
{
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    const int size = 10;

    int mas[size] = { 9,8,7,6,5,4,3,2,1,0 }; // { 9,0,8,1,7,2,6,3,5,4 };
    print(mas, size);
    selectionSort(mas, size);
    cout << "Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:\n";
    print(mas, size);


    system("pause");
    return 0;
}

void swap(int& first, int& second) //функция обмена значениями между двумя элементами. Чтобы обменять значения у ОРИГИНАЛОВ значений в main'e, эта функция принимает ссылки на
//элементы, то есть константные указатели на адреса элементов
{
    int t = first;
    first = second;
    second = t;
}

int minimumFrom(int* m, int position, int length) //функция для нахождения индекса (номера) элемента с минимальным значением на участке массива, начиная с элемента с индексом
//position до элемента с индексом length-1. Эта функция нужна для функции selectionSort()
{
    int minIndex = position;
    for (int i = position + 1; i < length; i++)
    {
        if (m[i] < m[minIndex])
        {
            minIndex = i;
        }
    }
    return minIndex;
}

void selectionSort(int* m, int length) //функция сортировки по ВОЗРАСТАНИЮ целочисленного одномерного массива выбором наименьшего элемента
{
    for (int i = 0; i < length - 1; i++)
    {
        swap(m[i], m[minimumFrom(m, i, length)]);
    }
}

```

 D:\2019\Lab\Debug\Lab5_4.exe

```

9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9

```

Сортировка вставками

Сортировка вставками — третий и последний из простых алгоритмов сортировки. Сначала он сортирует два первых элемента массива. Затем алгоритм вставляет третий элемент в соответствующую порядку позицию по отношению к первым двум элементам. После этого он вставляет четвертый элемент в список из трех элементов. Этот процесс повторяется до тех пор, пока не будут вставлены все элементы на подходящие для них места. Например, при сортировке массива **dcab** каждый проход алгоритма будет выглядеть следующим образом:

Начало **d c a b**

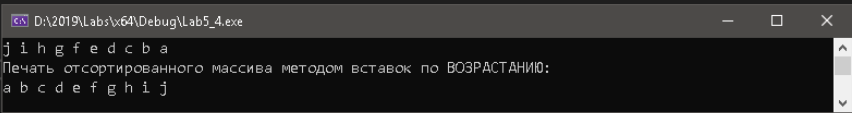
Проход 1 **c d a b**

Проход 2 **a c d b**

Проход 3 **a b c d**

Пример реализации сортировки вставками показан ниже:


```
void insertSort(char* m, int n) //функция сортировки по ВОЗРАСТАНИЮ одномерного символьного массива методом вставок. Этот алгоритм сортировки обладает свойством устойчивости, то
//есть при совпадении значений в ключевом поле объекты НЕ меняются местами
{
    for (int i = 1; i < n; i++)
    {
        char t = m[i];
        for (int j = i - 1; j >= 0 && t < m[j]; j--)
        {
            m[j + 1] = m[j];
            m[j] = t;
        }
    }
}
```



В отличие от пузырьковой сортировки и сортировки посредством выбора, количество сравнений в сортировке вставками зависит от изначальной упорядоченности списка. Если список уже отсортирован, количество сравнений равно $n-1$; в противном случае его производительность является величиной порядка n^2 .

В худших случаях сортировка вставками настолько же плоха, как и пузырьковая сортировка и сортировка посредством выбора, а в среднем она лишь немного лучше. Тем не менее, у сортировки вставками есть два преимущества. Во-первых, ее поведение **естественно**. Другими словами, она работает меньше всего, когда массив уже упорядочен, и больше всего, когда массив отсортирован в обратном порядке. Поэтому сортировка вставками — идеальный алгоритм для почти упорядоченных (значительно отсортированных, отсортированных не на 100%) массивов (списков, коллекций). Второе преимущество заключается в том, что данный алгоритм не меняет порядок одинаковых ключей, то есть он **устойчивый алгоритм** сортировки. Это значит, что если список отсортирован по двум ключам, то после сортировки вставками он останется упорядоченным по обоим.

Несмотря на то, что количество сравнений при определенных наборах данных может быть довольно низким, при каждой вставке элемента на нужное ему место другие элементы в массиве необходимо сдвигать. Вследствие этого количество перемещений может быть значительным.

Рекурсивная реализация алгоритма быстрой сортировки

Рассмотрим известный пример рекурсивного алгоритма — алгоритм быстрой сортировки *quickSort*. Этот рекурсивно определенный алгоритм предназначен для упорядочения значений, хранящихся в массиве, по возрастанию или по убыванию.

Предположим, что 11 элементов массива имеют следующие значения (рис. 6).

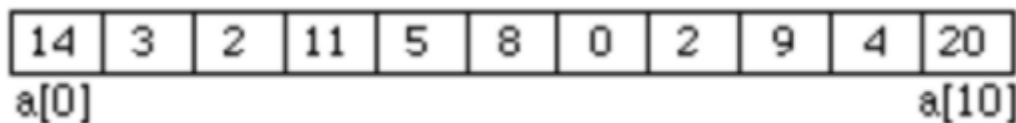


Рис. 6. — Начальное состояние массива.

Идея алгоритма основана на рекурсивном выполнении разбиения массива на две части и переупорядочении элементов в этих частях. Разбиение выполняется путем выбора некоторого элемента, который будем называть *граничным*. После разбиения две части массива обрабатываются так, чтобы в одной части располагались значения, меньшие или равные граничному элементу, а в другой части — только большие или равные.

Например, если выбрать в качестве граничного элемента значение 8, то после переупорядочения массив окажется в состоянии, показанном на рис. 7. Затем тот же самый процесс независимо выполняется для левой и правой частей массива.



Рис. 7. — Массив после разделения на две части и переупорядочения элементов в них.

Процесс разбиения и переупорядочения частей массива можно реализовать рекурсивно. Индекс граничного элемента вычисляется как индекс среднего элемента: $(first + last)/2$ /*деление целочисленное, но тут это не мешает вычислить индекс элемента, который тоже целочисленный согласно свойствам массива*/, где *first* и *last* являются индексами первого и последнего элементов обрабатываемой части массива («куска» массива, отрезка в массиве).

Процедура переупорядочения элементов массива.

Индексы *leftArrow* и *rightArrow* определим как индексы крайних элементов обрабатываемой части массива (которая подвергается разбиению, см. рис. 8).

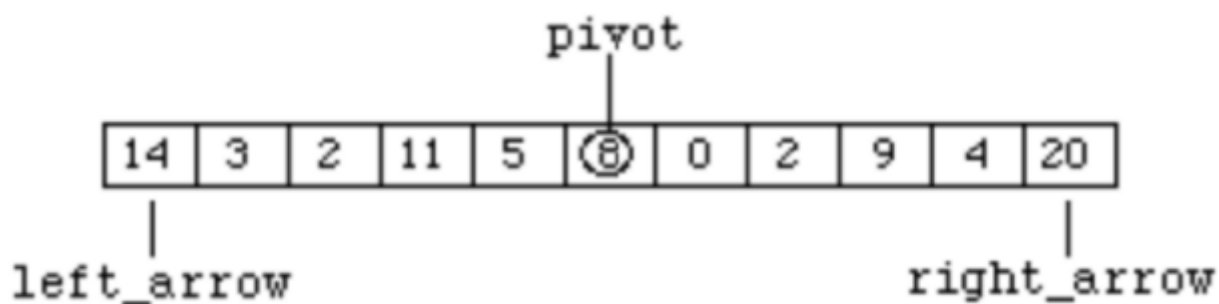


Рис. 8. – Значения индексов leftArrow и rightArrow перед началом процедуры переупорядочения.

В процессе переупорядочения индексы leftArrow и rightArrow смещаются в направлении граничного элемента. Так, rightArrow перемещается влево, пока значение элемента не окажется меньше или равно граничному значению (рис. 9).

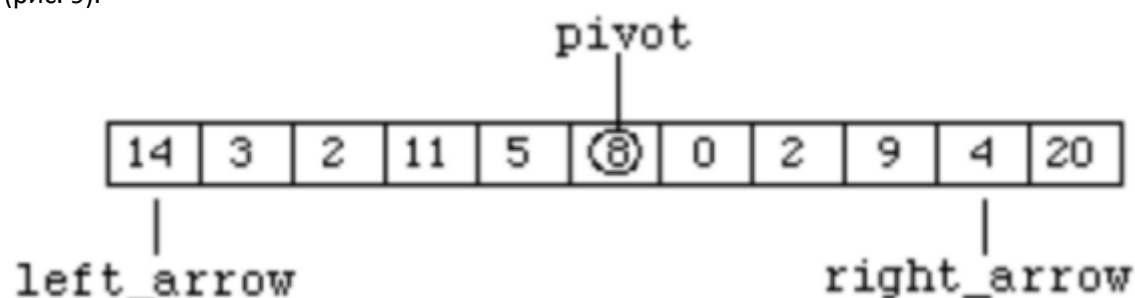


Рис. 9. – Обнаружение левого и правого элементов для обмена.

Аналогично, leftArrow смещается вправо, пока не встретится элемент, больший или равный граничному. В нашем примере этот элемент расположен в начале массива (рис. 9). Теперь значения двух найденных элементов массива меняются местами (рис. 10).

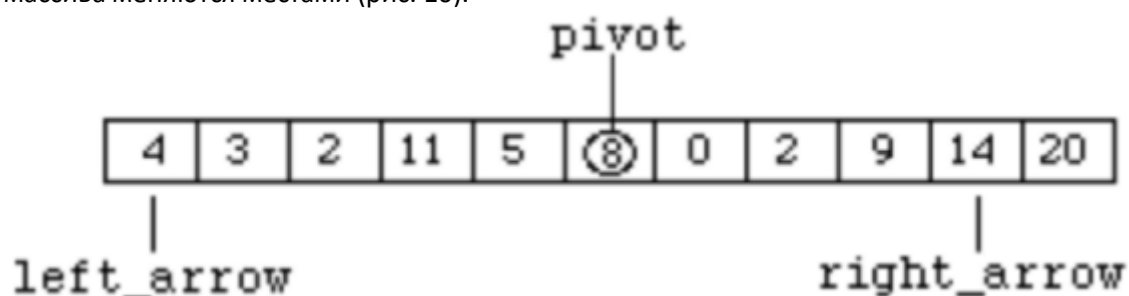


Рис. 10. – Состояние массива после обмена двух элементов.

После обмена двух элементов rightArrow продолжает смещаться влево. Смещение прекращается, когда находится очередной элемент, меньший или равный граничному (рис. 11).

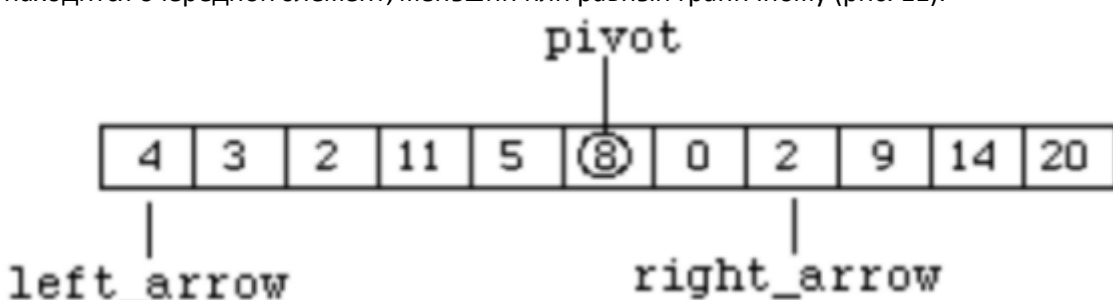


Рис. 11. – Справа обнаружен элемент для обмена.

Индекс leftArrow смещается вправо, пока не встретится элемент, требующий перестановки (рис. 12).

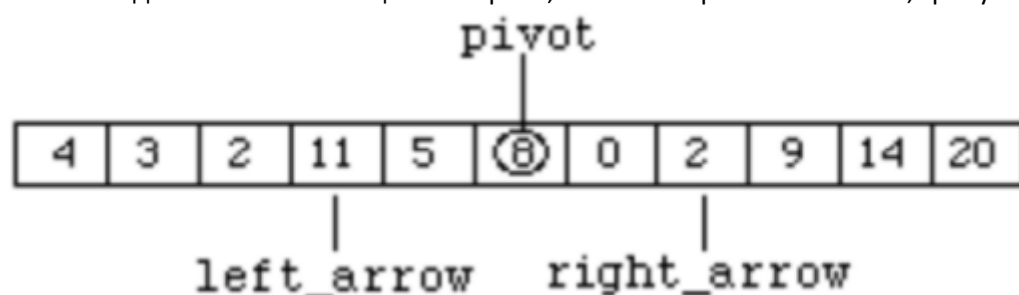


Рис. 12. – Слева обнаружен элемент для обмена.

Значения найденных элементов меняются местами (рис. 13).

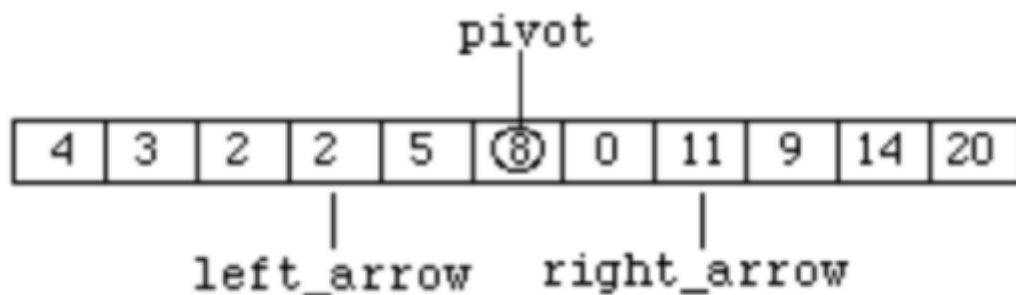


Рис. 13. – Состояние массива после обмена второй пары элементов.

Переупорядочение частей массива прекращается после выполнения условия $\text{leftArrow} > \text{rightArrow}$. В состоянии на рис. 13 это условие ложно, поэтому "rightArrow" продолжает смещаться влево, пока не окажется в положении, показанном на рис. 14.

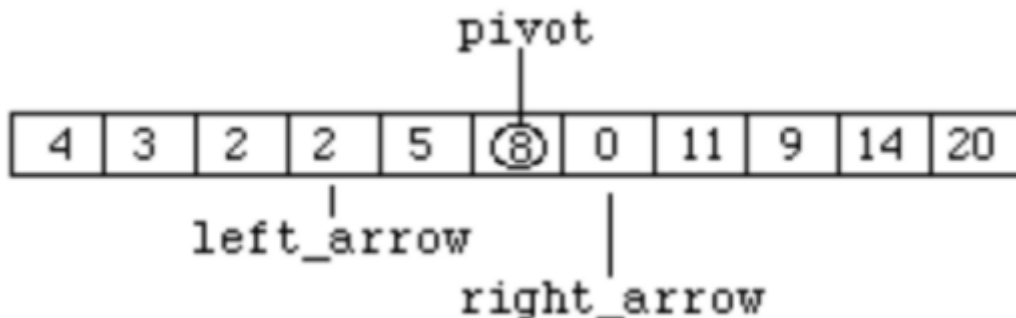


Рис. 14. – Справа обнаружен элемент для обмена.

Перемещение leftArrow приведет к состоянию, показанному на рис. 15. Поскольку при перемещении вправо надо найти элемент, больший или равный pivot, то leftArrow прекращает перемещение после достижения граничного элемента.

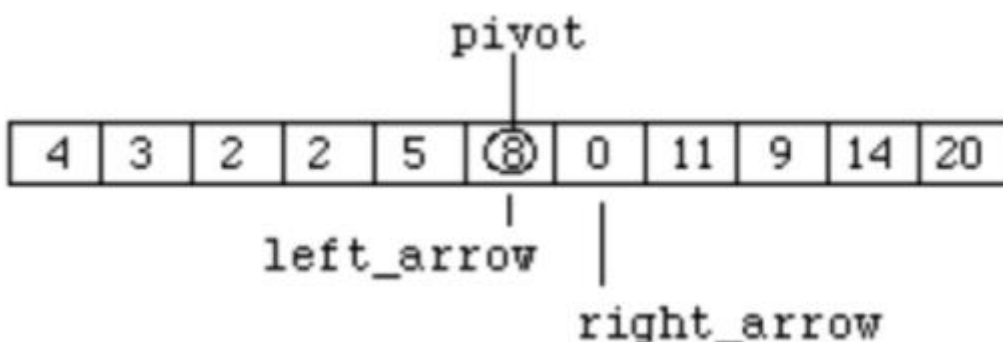


Рис. 15. – Левый элемент для обмена совпадает с граничным.

Теперь выполняется обмен, включающий граничный элемент (такой обмен допустим), и массив переходит в состояние, показанное на рис. 16.

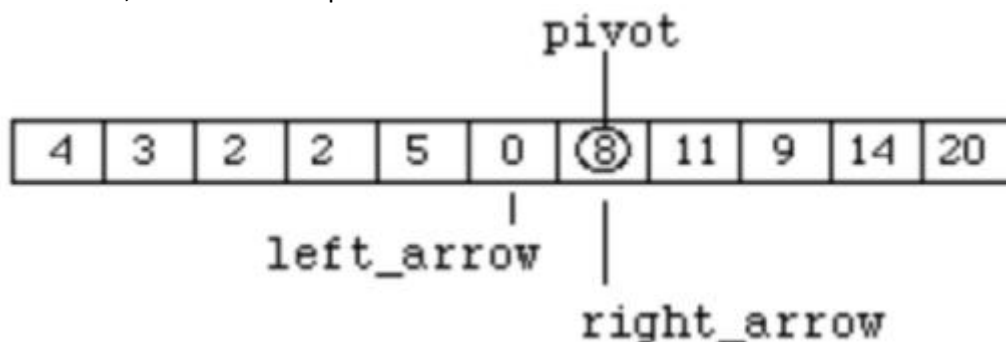


Рис. 16. – Состояние массива после обмена третьей пары элементов.

После обмена элементов rightArrow перемещается влево, а leftArrow – вправо (рис. 17).

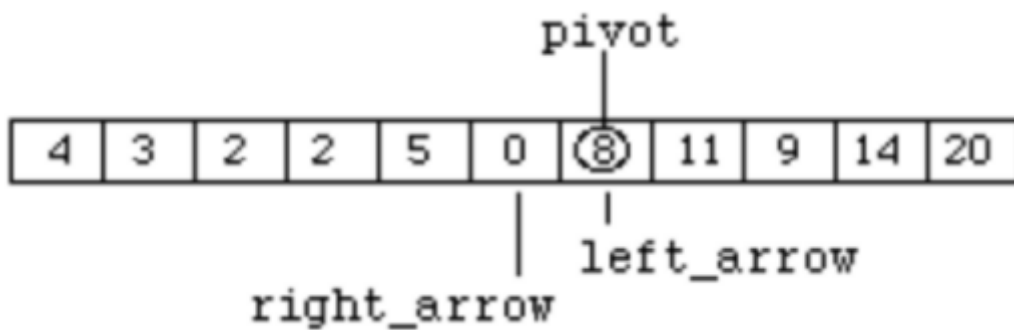
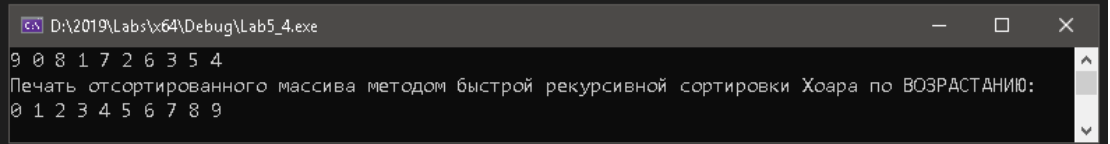


Рис. 17. – Переупорядочение прекращается после прохождения индексами середины массива.

В состоянии, показанном на рис. 17, становится истинным условие завершения процедуры переупорядочения $\text{leftArrow} > \text{rightArrow}$. Поэтому процедуру разбиения и переупорядочения массива можно считать выполненной.

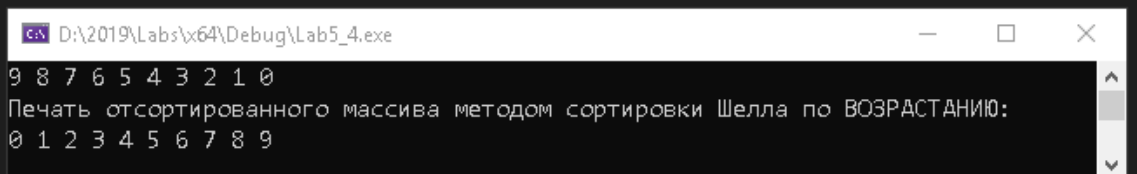
Ниже приведена функция на Си++, в которой рекурсивно реализован алгоритм сортировки `quickSort`:

```
void quickSort(int* m, int left, int right)//функция быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
{
    int leftArrow = left, rightArrow = right, pivot = m[(left + right) / 2];
    do
    {
        while (m[rightArrow] > pivot)
        {
            rightArrow--;
        }
        while (m[leftArrow] < pivot)
        {
            leftArrow++;
        }
        if (leftArrow <= rightArrow)
        {
            swap(m[leftArrow], m[rightArrow]);
            leftArrow++;
            rightArrow--;
        }
    }
    while (rightArrow >= leftArrow);
    if (left < rightArrow)
    {
        quickSort(m, left, rightArrow);
    }
    if (leftArrow < right)
    {
        quickSort(m, leftArrow, right);
    }
}
```



Быстрая сортировка Шелла

```
void shellSort(int* m, int n)//функция сортировки методом Шелла. Это неустойчивая сортировка
{
    for (int i = n / 2; i > 0; i = i / 2)
    {
        for (int j = 0; j < n - i; j++)
        {
            for (int k = j; k > -1; k = k - i)
            {
                if (m[k] > m[k + i])
                {
                    int t = m[k];
                    m[k] = m[k + i];
                    m[k + i] = t;
                }
            }
            else
            {
                k = 0;
            }
        }
    }
}
```



Алгоритм сортировки методом Шелла основывается на прохождении по массиву с определенным шагом, равным половине массива, потом половине половины массива и так далее, что позволяет довольно быстро прийти до

конкретных элементов и поставить их на нужные места.

Бинарный поиск элемента в отсортированном массиве

Если массив НЕ отсортирован ПОЛНОСТЬЮ по конкретному принципу (в примере, только по ВОЗРАСТАНИЮ значений элементов), то бинарный поиск к нему применять не стоит, поскольку правильный результат не гарантирован (но возможен ввиду случайного совпадения). Также бинарный поиск в отсортированном массиве, где есть совпадающие элементы, не гарантирует, что возвратится индекс первого или последнего подошедшего по значению элемента (но возможно совпадение; также алгоритм можно трансформировать для достижения последней цели). Суть бинарного поиска заключается в разбиении массива пополам и предположении, где должен находиться искомый элемент (ключ поиска), поскольку массив отсортирован, а значит элементы в нем располагаются упорядоченно по значениям. Далее эти же действия повторяются в «подходящей» половине массива и так далее, пока не будет достигнуто совпадение либо дробление массива завершится уменьшением подмассива до одно элемента, значение которого не совпадет с ключом поиска и нужно будет вернуть значение, сигнализирующее об отрицательном результате – обычно это -1, поскольку результатом работы функции поиска в одномерном массиве является посылка целочисленного индекса подошедшего элемента, а индекса -1 и других отрицательных индексов у массива быть не может.

Допишем код программы функцией бинарного поиска `binarySearch(одномерныйМассив, размерМассива, искомоеЗначение)`, ее прототипом и вызовом ее в `main'e`:

```
1  #include <iostream>
2  #include <Windows.h>
3  using namespace std;
4
5  void bubbleSort(int*, int); //прототип функции сортировки "пузырьком" по ВОЗРАСТАНИЮ
6  void print(int*, int); //прототип функции печати содержимого элементов одномерного целочисленного массива на консоль
7  void printStr(char*, int); //прототип функции печати содержимого элементов одномерного символьного массива на консоль
8  void swap(int&, int&); //прототип функции обмена значениями между двумя элементами
9  int minimumFrom(int*, int, int); //прототип функции для нахождения индекса (номера) элемента с минимальным значением на участке массива
10 void selectionSort(int*, int); //прототип функции сортировки целочисленного одномерного массива выбором наименьшего элемента
11 void insertSort(char*, int); //прототип функции сортировки по ВОЗРАСТАНИЮ одномерного символьного массива методом вставок
12 void quickSort(int*, int, int); //прототип функции быстрой сортировки (рекурсивной сортировки, сортировки Хоара)
13 void shellSort(int*, int); //прототип функции сортировки методом Шелла
14 int binarySearch(int*, int, int); //прототип функции бинарного поиска значения в одномерном массиве передаваемого размера. Массив ДОЛЖЕН БЫТЬ ОТСОРТИРОВАН по ВОЗРАСТАНИЮ
15
16 int main()
17 {
18     SetConsoleOutputCP(1251);
19     SetConsoleCP(1251);
20     const int size = 10;
21     int ar[size] = { 9,0,8,1,7,2,6,3,5,4 };
22     print(ar, size);
23     bubbleSort(ar, size);
24     cout << "Печать отсортированного массива методом ""пузырька"" по ВОЗРАСТАНИЮ:\n";
25     print(ar, size);
26     int mas[size] = { 9,8,7,6,5,4,3,2,1,0 }; //{ 9,0,8,1,7,2,6,3,5,4 };
27     print(mas, size);
28     selectionSort(mas, size);
29     cout << "Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:\n";
30     print(mas, size);
31     char s[size] = { 'j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a' };
32     printStr(s, size);
33     insertSort(s, size);
34     cout << "Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:\n";
35     printStr(s, size);
36     int vector[size] = { 9,0,8,1,7,2,6,3,5,4 };
37     print(vector, size);
38
39     quickSort(vector, 0, size-1);
40     cout << "Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:\n";
41     print(vector, size);
42     int vect[size] = { 11,23,99,76,5,49,31,92,61,0 };
43     print(vect, size);
44     shellSort(vect, size);
45     cout << "Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:\n";
46     print(vect, size);
47     int k;
48     cout << "Введите значение для поиска в массиве: ";
49     cin >> k;
50     int r = binarySearch(vect, size, k); //вызываем функцию binarySearch(имяМассива, размерМассива, искомоеЗначениеЭлементаВМассиве) и сохраняем возвращенное функцией бинарного
51     if (r < 0) //поиска значение (результат работы функции binarySearch()) в переменную r, которую затем проверяем на отрицательность (можно сравнить == -1). Если r равно -1, то
52     { //есть < 0, то это значит, что бинарным поиском в массиве НЕ удалось найти искомый элемент (ключ, ключевое значение) или массив НЕ отсортирован по ВОЗРАСТАНИЮ
53         cout << "Искомое значение-ключ " << k << " в массиве не найден или массив не отсортирован по возрастанию.\n";
54     }
55     else //если вернуло значение от нуля включительно или больше, то это есть индекс подошедшего по значению элемента массива vect
56     {
57         cout << "Искомое значение-ключ " << k << " найдено в массиве в элементе с индексом [" << r << "].\n";
58     }
59     system("pause");
60     return 0;
61 }
```

```

181 int binarySearch(int* m, int n, int x)//функция поиска значения x в массиве m размера n. Массив ДОЛЖЕН БЫТЬ ОТСОРТИРОВАН по ВОЗРАСТАНИЮ
182 {
183     int low = 0, high = n - 1, middle = (low + high) / 2;
184     while (high >= low)
185     {
186         if (m[middle] == x)//если нашло совпадение, то вернуть индекс подошедшего элемента. НЕГ ГАРАНТИИ, что вернет индекс первого подошедшего элемента, если будет несколько
187         //одинаковых подходящих значений (в отсортированном массиве они располагаются рядом)
188             return middle;
189     }
190     else
191     {
192         if (x > m[middle])//если в отсортированном по возрастанию массиве искомое значение x больше, чем значение в "серединном" элементе массива, то исключим из проверяемого
193         //диапазона элементов в массиве все элементы до текущего "серединного" включительно, для чего "нижнюю" (левую) границу поиска установим на элемент с индексом middle+1
194             low = middle + 1;
195         else//то есть x < m[middle]. Если в отсортированном по возрастанию массиве искомое значение x меньше, чем значение в "серединном" элементе массива, то исключим из
196         //проверяемого диапазона элементов в массиве все элементы, начиная с текущего "серединного" включительно, для чего "верхнюю" (правую) границу поиска установим на
197             high = middle - 1;//элемент с индексом middle-1
198     }
199     middle = (low + high) / 2;//диапазон поиска сузился, поэтому надо пересчитать индекс "серединного" элемента - он находится посередине между крайними левым и правым
200     //элементами оставшегося уточненного уменьшенного диапазона
201 }
202 return -1;//если цикл проработал до конца (крайние левый и правый элементы уточненного диапазона сдвигались и стали указывать на один и тот же элемент, но и он не совпал с
203 //искомым ключевым значением и return не сработал путем отправки индекса совпавшего элемента, то завершаем функцию бинарного поиска возвратом значения -1 как значения,
204 //сигнализирующего, что совпадений нет (поскольку индекса -1 в массиве быть не может). Но если массив НЕ отсортирован по ВОЗРАСТАНИЮ полностью, то бинарный поиск не гарантирует
205 //правильного результата. Можно отредактировать данную функцию, чтобы она корректно осуществляла поиск в отсортированном по УБЫВАНИЮ массиве
206

```

Изменим массив в `vec` и протестируем бинарный поиск на отсортированном по возрастанию одномерном целочисленном массиве:

```

D:\2019\Labs\64\Debug\Lab5_4.exe
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом пузырька по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:
a b c d e f g h i j
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
11 23 99 76 5 49 31 92 61 0
Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:
0 5 11 23 31 49 61 76 92 99
Введите значение для поиска в массиве: 11
Искомое значение-ключ 11 найдено в массиве в элементе с индексом [2].
Для продолжения нажмите любую клавишу . . .

D:\2019\Labs\64\Debug\Lab5_4.exe
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом пузырька по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:
a b c d e f g h i j
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
11 23 99 76 5 49 31 92 61 0
Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:
0 5 11 23 31 49 61 76 92 99
Введите значение для поиска в массиве: 49
Искомое значение-ключ 49 найдено в массиве в элементе с индексом [5].
Для продолжения нажмите любую клавишу . . .

```

```
cs D:\2019\Labs\64\Debug\Lab5_4.exe
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом пузырька по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:
a b c d e f g h i j
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
11 23 99 76 5 49 31 92 61 0
Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:
0 5 11 23 31 49 61 76 92 99
Введите значение для поиска в массиве: -11
Искомое значение-ключ -11 в массиве не найден или массив не отсортирован по возрастанью.
Для продолжения нажмите любую клавишу . . .

cs D:\2019\Labs\64\Debug\Lab5_4.exe
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом пузырька по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
Печать отсортированного массива методом выбора наименьшего значения по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
Печать отсортированного массива методом вставок по ВОЗРАСТАНИЮ:
a b c d e f g h i j
9 0 8 1 7 2 6 3 5 4
Печать отсортированного массива методом быстрой рекурсивной сортировки Хоара по ВОЗРАСТАНИЮ:
0 1 2 3 4 5 6 7 8 9
11 23 99 76 5 49 31 92 61 0
Печать отсортированного массива методом сортировки Шелла по ВОЗРАСТАНИЮ:
0 5 11 23 31 49 61 76 92 99
Введите значение для поиска в массиве: 50
Искомое значение-ключ 50 в массиве не найден или массив не отсортирован по возрастанью.
Для продолжения нажмите любую клавишу . . .
```

Задания по вариантам:

Вариант 1

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Работник (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- по фамилии (по возрастанию);
- по окладу (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка выбором,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.
Оцените скорость, естественность и устойчивость каждого алгоритма.
Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (фамилия, имя, номер телефона, дата рождения, оклад).
Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.
Оптимальную организацию функции продумайте самостоятельно.
Продemonстрируйте работу функции для всех вариантов поиска.

Вариант 2

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры `Маршрут` (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- название конечного пункта маршрута (по возрастанию);
- длина маршрута (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка вставками,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.

Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (название начального пункта маршрута; название конечного пункта маршрута; номер маршрута; длина маршрута).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продemonстрируйте работу функции для всех вариантов поиска.

Вариант 3

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры `Поезд` (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей различным ключам:

- название пункта назначения (по возрастанию);
- номер поезда (по убыванию);

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка выбором,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.
Оцените скорость, естественность и устойчивость каждого алгоритма.
Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (название пункта назначения; номер поезда; время отправления; время прибытия).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 4

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Знак зодиака (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей различным ключам:

- фамилия (по возрастанию);
- дата рождения (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка вставками,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.

Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (фамилия; имя; знак Зодиака; дата рождения).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 5

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Аэрофлот (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей различным ключам:

- название пункта назначения (по возрастанию);
- цена билета (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка выбором,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.

Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (номер рейса; название пункта назначения рейса; тип самолета; цена билета).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 6

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Студент (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- фамилия (по возрастанию);
- средний балл (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка вставками,
- сортировка Шелла,
-

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.

Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (фамилия и инициалы; номер группы; успеваемость (любой из пяти элементов)).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 7

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Работник (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- фамилия (по возрастанию);
- оклад (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка выбором,
- быстрая сортировка.

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.
Оцените скорость, естественность и устойчивость каждого алгоритма.
Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (фамилия и инициалы сотрудника; название занимаемой должности; год поступления на работу, оклад).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 8

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Цена товара (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- номенклатурный номер (по возрастанию);
- стоимость товара (по убыванию);

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка вставками,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.

Оптимальную организацию функций продумайте самостоятельно.

Задание 2.

Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (номенклатурный номер; название товара; стоимость товара; количество на складе.).

Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.

Оптимальную организацию функции продумайте самостоятельно.

Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 9

Задание 1.

В лабораторной работе № 16 в задании № 6 найдите описание структуры Запись в телефонной книге (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)

Выполните сортировку записей по различным ключам:

- по фамилии (по возрастанию);
- по окладу (по убыванию).

Для каждого ключа используйте различные алгоритмы сортировки:

- сортировка выбором,
- сортировка Шелла,

Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.

Для каждого случая подсчитайте количество сравнений и перестановок.

Оцените скорость, естественность и устойчивость каждого алгоритма.
Оптимальную организацию функций продумайте самостоятельно.

Задание 2.
Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (фамилия, имя, номер телефона, дата рождения, оклад).
Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.
Оптимальную организацию функции продумайте самостоятельно.
Продемонстрируйте работу функции для всех вариантов поиска.

Вариант 10

Задание 1.
В лабораторной работе № 16 в задании № 6 найдите описание структуры Аэрофлот (это может быть любой вариант), объявите эту структуру в данной лабораторной работе, дополните ее (если надо для нижеследующих заданий) новыми полями. Создайте динамический массив из экземпляров данной структуры, размер и значения массива определяет пользователь с клавиатуры. (Для удобства тестирования сортировок можно явно проинициализировать массив строками значений.)
Выполните сортировку записей различным ключам:
· название пункта назначения (по возрастанию);
· цена билета (по убыванию).
Для каждого ключа используйте различные алгоритмы сортировки:
· сортировка выбором,
· сортировка Шелла,
Всего сортировка будет выполнена 4 раза, каждый раз сортируется массив в первоначальном состоянии.
Для каждого случая подсчитайте количество сравнений и перестановок.
Оцените скорость, естественность и устойчивость каждого алгоритма.
Оптимальную организацию функций продумайте самостоятельно.

Задание 2.
Напишите функцию(-и) `search()` для поиска записи в массиве из предыдущего задания по определенному ключу. Ключом может являться любое поле структуры (номер рейса; название пункта назначения рейса; тип самолета; цена билета).
Причем функция должна выполнять последовательный поиск, если массив не сортирован, и бинарный поиск, если массив сортирован.
Оптимальную организацию функции продумайте самостоятельно.
Продемонстрируйте работу функции для всех вариантов поиска.

Задание № 3

1	Написать программу, взаимодействующую с пользователем посредством консольного меню (switch-case внутри цикла; каждый case реализует один пункт меню). Кейсы могут содержать подменю.
2	Предложить пользователю ввести размер массива (количество элементов одномерного массива), предложить пользователю выбрать тип элементов массива из двух предложенных (самостоятельно выберите два различных типа из разных семейств типов: double, float, char, bool, short int, long long int) и создать такой массив.
3	Запросить у пользователя размерность двумерного массива и создать такой массив (тип массива определяется вашим номером по журналу группы): 1 – double, 2 – float, 3 – bool, 4 – long int, 5 – short int, 6 – char, 7 – double, и так далее циклически. Пример сортировки двумерного целочисленного массива см. в предыдущих лабораторных работах.
4	Запросить у пользователя размер одномерного массива и создать такой массив типа структуры данных struct по вашему варианту из Лабораторной работы № 15 задания 1.
5	Для каждого из четырех массивов предусмотреть по одной сортировке на ваш выбор:

	<p>пузырьковая (перестановками), выбором, вставками, методом Шелла и быстрая рекурсивная Хоара. Одни сортировки пусть сортируют по возрастанию (2 сортировки), другие – по убыванию (2 сортировки). Элементы типа структуры struct сортируются по какому-либо из своих полей.</p>
6	Предусмотрите код или отдельные пункты меню для печати содержимого массивов на консоль, чтобы видеть содержимое массива до сортировки и после сортировки.
7	Предусмотрите удаление динамических массивов из памяти в виде пунктов меню.
8	Предусмотрите пункт меню для завершения работы вашей программы.