**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

отчет

**по лабораторной работе №2**

**по дисциплине «Программирование»**

Тема: «Одномерные статические массивы»

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2020

**Цель работы.**

Ознакомиться с работой одномерных массивов, базовыми операциями с массивами, различным типам сортировок массивов.

Необходимо написать программу, которая:

1)    Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.

2)    Отсортировать заданный в пункте 1 элементы массива […] сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.

3)    Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.

4)    Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения. Выводит индексы всех элементов, которые равны этому значению, и их количество.

5)    Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.

6)    Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.

7)    Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)

8)     Меняет местами элементы массива, индексы которых вводит пользователь. Выведите скорость обмена, используя библиотеку chrono.

Должна присутствовать возможность запуска каждого пункта многократно.

Таблица – Варианты сортировок

|  |  |
| --- | --- |
| Номер варианта | Название сортировки |
| 1 | Bubble sort (пузырьковая сортировка) |
| 2 | Shaker sort (шейкер-сортировка) |
| 3 | Comb sort (сортировка расчёской) |
| 4 | Insert sort (сортировка вставками) |
| 5 (\*) | Quick sort (быстрая сортировка) |

**Основные теоретические положения.**

* Понятие массива

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнить однообразные действия, им дают одно имя, а различают по порядковому номеру (индексу). Это дает возможность компактно записать множество операций с использованием циклов.

Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение.

Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных.

Аналогом одномерного массива из математики может служить последовательность некоторых элементов с одним индексом: a\_i*ai*​ при  i = 0, 1, 2, … n – одномерный вектор. Каждый элемент такой последовательности представляет собой некоторое значение определенного типа данных. Наглядно одномерный массив можно представить как набор пронумерованных ячеек, в каждой из которых содержится определенное значение:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 3.02 | 1.5 | 7.0 | -2.3 | 12.0 |

Это пример одномерного массива из 5 элементов, каждый из которых представляет собой некоторое вещественное значение и каждое из этих значений имеет индекс от 0 до 4.

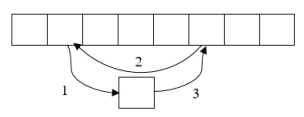
А вот пример одномерного массива из десяти элементов, представляющих собой одиночные символы:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| ‘a’ | ‘b’ | ‘c’ | ‘+’ | ‘1’ | ‘2’ | ‘!’ | ‘#’ | ‘@’ | ‘&’ |

Каждый элемент в этих массивах определяется значением индекса элемента. Например, в последнем массиве элемент с индексом 5 равен символу  ‘2’.

* Обмен местами элементов массива

Обмен элементов массива осуществляется через буферную переменную либо через функцию swap(a, b).

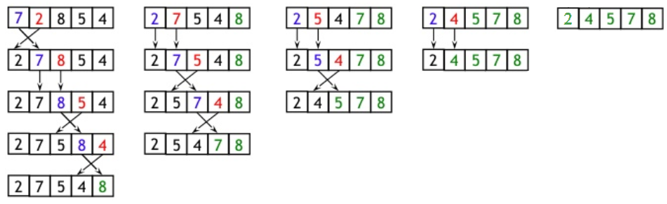


* Пузырьковая сортировка массива (bubble sort)

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива.

Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент.

Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться *N*-1 элемент.



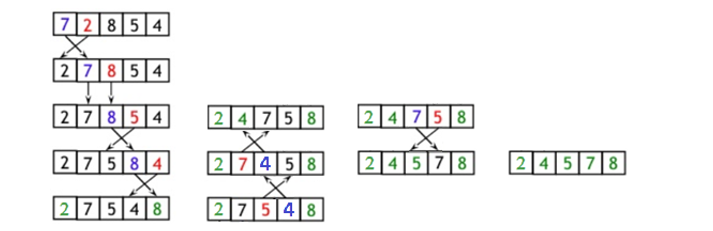
Очевидно, что хуже всего алгоритм будет работать, когда на вход подается массив, отсортированный в обратную сторону (от большего к меньшу). Быстрее же всего алгоритм работает с уже отсортированным массивом.

Но стандартный алгоритм пузырьковой сортировки предполагает полный циклический проход по массиву. Если изначально подается упорядоченная последовательность, то работа алгоритма все равно продолжиться. Исправить это можно, добавив условие проверки: если на текущей итерации ни один элемент не изменил свой индекс, то работа алгоритма прекращается.

* Шейкер-сортировка массива (shaker sort)

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется на порядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.



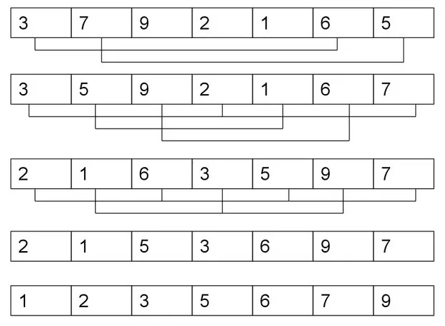
Кажется, что bubble sort теряет свою эффективность по сравнению с shaker sort. Сортировка проходит в массиве в обоих направлениях, а не только от его начала к концу. Но в работе с большими массивами преимущество шейкер-сортировки уменьшается как раз из-за использования двух циклов.

* Сортировка массива расчёской (comb sort)

Очевидный недостаток bubble и shaker sort заключается в том, что элементы переставляются максимум на одну позицию.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива , а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком.



Сортировка расческой работает намного быстрее, чем bubble или shaker sort, в некоторых ситуациях comb sort работает быстрее quick sort. Но данная сортировка обладает одним очевидным минусом – неустойчивость.

* Сортировка массива вставками (insert sort)

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

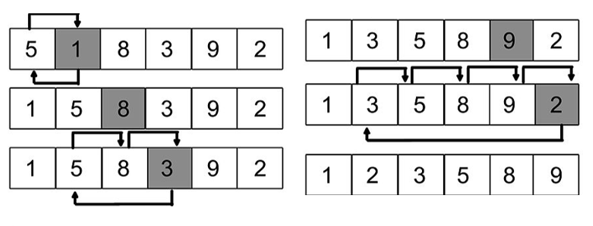
Общая суть сортировки вставками такова:

1)    Перебираются элементы в неотсортированной части массива.

2)    Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делить массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Рассмотрим самый простой способ (рис. 3.5). Необходимо пройти массив слева направо и обработать каждый элемент. Слева будет наращиваться отсортированная часть массива, а справа – уменьшаться неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, что освобождает место в массиве и позволяет сдвинуть элементы и освободить точку вставки.



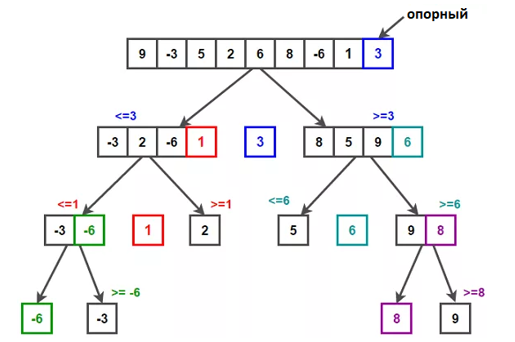
Существует множество модификаций сортировки вставками, некоторые из них затрагивают именно способ вставки элемента в отсортированную часть. Одна из самых лучших модификаций – сортировка простыми вставками с бинарным поиском. Бинарный поиск будет описан позже.

Лучше всего сортировка вставками работает при обработке почти отсортированных массивов. В таком случае insert sort работает быстрее других сортировок.

* Быстрая сортировка массива (quick sort)

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

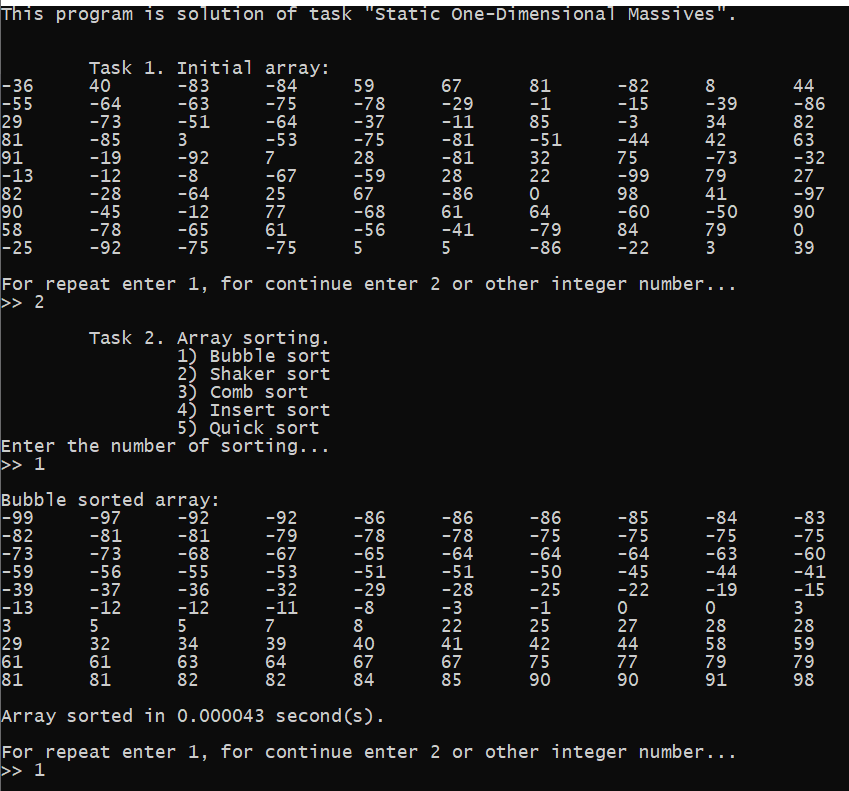
Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

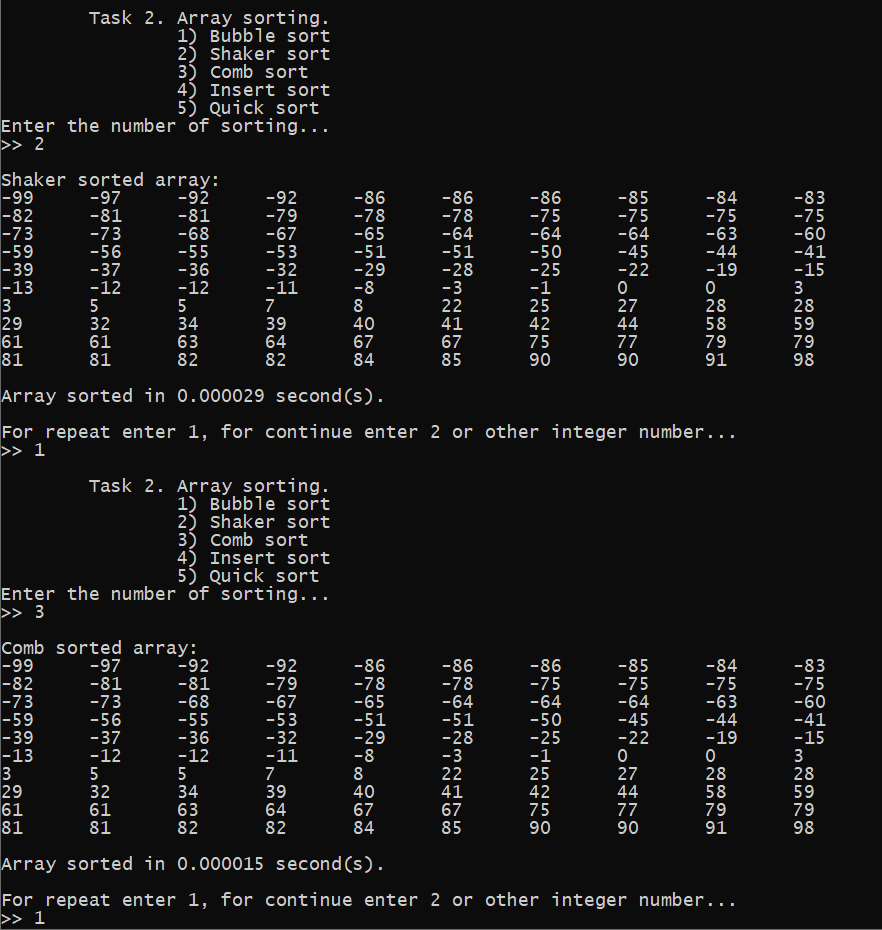


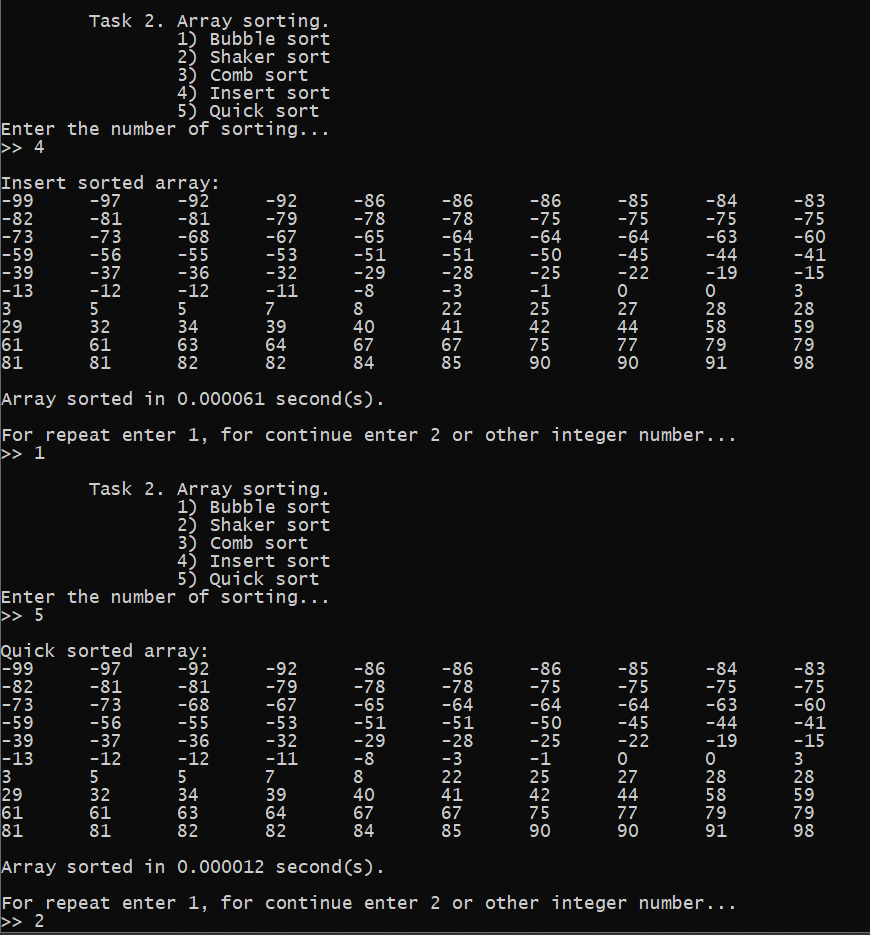
На практике массив обычно делят на две части: «меньше опорного» и «равные и большие» или «меньше опорного или равные» и «большие». Такой поход в общем случае эффективнее, ведь упрощается алгоритм разделения.

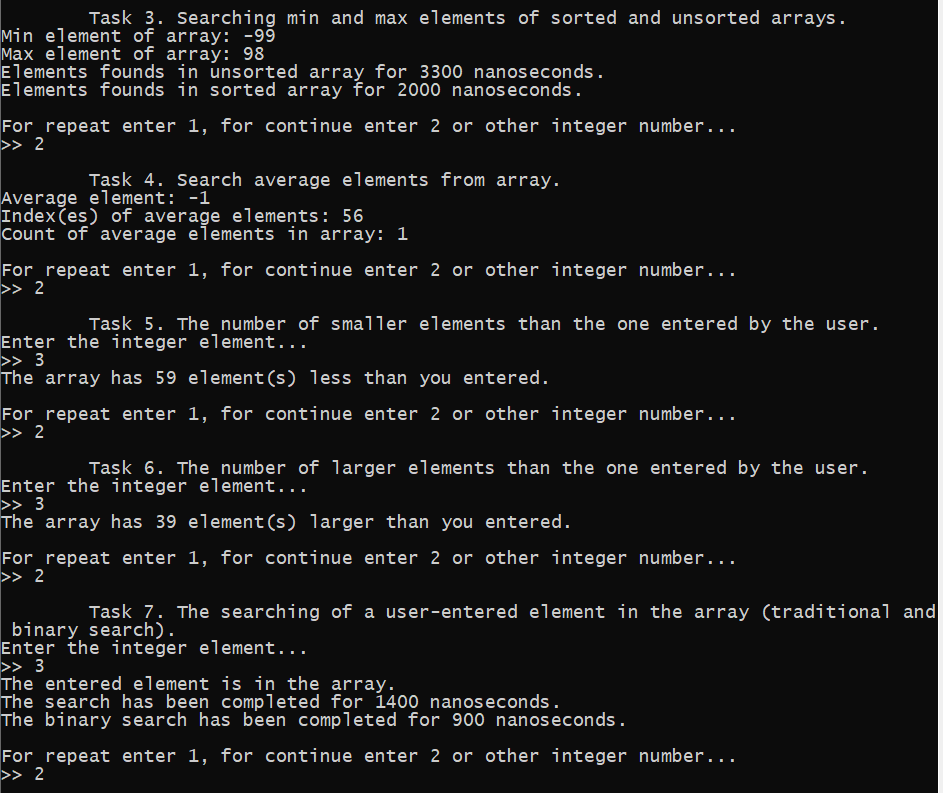
При том, что это один из самых быстродействующих из алгоритмов, данный алгоритм сортировки неустойчив, а прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека.

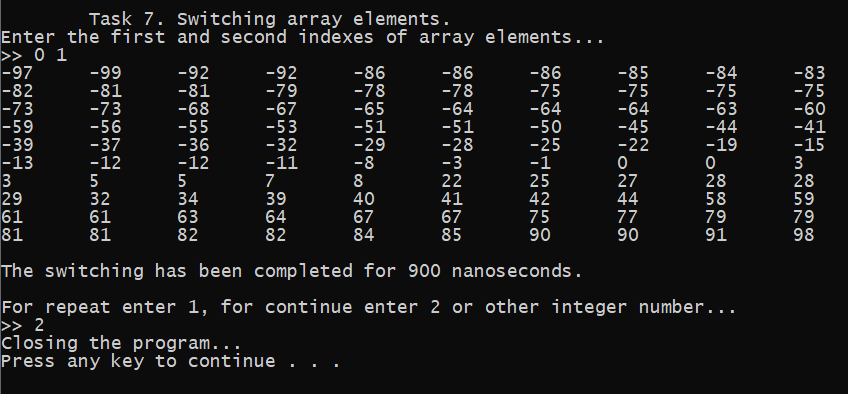
**Экспериментальные результаты.**









  
Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

Приложение а  
полный код программы

#include "pch.h"

#include <iostream>

#include <ctime> // Нужно для рандома

#include <chrono> // Нужно для засечения времени

using namespace std;

using namespace std::chrono; // Нужно для засечения времени

void createAndPrintRandomArray(int[]);

void bubbleSort(int[]);

void shakerSort(int[]);

void combSort(int[]);

void insertSort(int[]);

void quickSort(int[], int, int);

int getMinArrayElement(int[]);

int getMaxArrayElement(int[]);

int binarySearch(int[], int, int, int);

void copyArray(int[], int[]);

void printArray(int[]);

bool choiseNextAction();

float stopSecondsTimer(time\_point<steady\_clock>);

long long stopNanoSecondsTimer(time\_point<steady\_clock>);

const int sizeOfArray = 100;

int main()

{

srand((unsigned)time(NULL));

time\_point<steady\_clock> startTimer = steady\_clock::now();

float stopTimeInSeconds = stopSecondsTimer(startTimer);

long long stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

int count = 0;

cout << "This program is solution of task \"Static One-Dimensional Massives\". \n\n";

// РЕШЕНИЕ ПЕРВОГО ЗАДАНИЯ

int mainArray[sizeOfArray];

do {

cout << "\n\tTask 1. Initial array:" << endl;

createAndPrintRandomArray(mainArray);

} while (choiseNextAction());

do { // РЕШЕНИЕ ВТОРОГО ЗАДАНИЯ

cout << "\n\tTask 2. Array sorting." << endl

<< "\t\t1) Bubble sort" << endl

<< "\t\t2) Shaker sort" << endl

<< "\t\t3) Comb sort" << endl

<< "\t\t4) Insert sort" << endl

<< "\t\t5) Quick sort" << endl

<< "Enter the number of sorting... \n>> ";

int input;

cin >> input;

int copiedArray[sizeOfArray];

copyArray(mainArray, copiedArray); // Копирование массива в другой массив для возможности в дальнейшем на том же массиве проверить другие типы сортировки.

switch (input) {

case 1: // РЕШЕНИЕ 2.1 ЗАДАНИЯ (bubble sort)

startTimer = steady\_clock::now(); // Старт отсчёта времени

bubbleSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nBubble sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 2: // РЕШЕНИЕ 2.2 ЗАДАНИЯ (shaker sort)

startTimer = steady\_clock::now();

shakerSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nShaker sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 3: // РЕШЕНИЕ 2.3 ЗАДАНИЯ (comb sort)

startTimer = steady\_clock::now();

combSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nComb sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 4: // РЕШЕНИЕ 2.4 ЗАДАНИЯ (insert sort)

startTimer = steady\_clock::now();

insertSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nInsert sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 5: // РЕШЕНИЕ 2.5 ЗАДАНИЯ (quick sort)

startTimer = steady\_clock::now();

quickSort(copiedArray, sizeOfArray - 1, 0);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nQuick sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

}

} while (choiseNextAction());

int copiedArray[sizeOfArray]; // Создание отсортированного массива, с которым будет производиться работа в дальнейших заданиях

copyArray(mainArray, copiedArray);

quickSort(copiedArray, sizeOfArray - 1, 0);

do { // РЕШЕНИЕ ТРЕТЬЕГО ЗАДАНИЯ

cout << "\n\tTask 3. Searching min and max elements of sorted and unsorted arrays. \n";

startTimer = steady\_clock::now(); // Поиск в неотсортированном массиве

int max = getMaxArrayElement(mainArray);

int min = getMinArrayElement(mainArray);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

cout << "Min element of array: " << min << endl

<< "Max element of array: " << max << endl

<< "Elements founds in unsorted array for " << stopTimeInNanoSeconds << " nanoseconds. \n";

startTimer = steady\_clock::now(); // Поиск в отсортированном массиве

max = getMaxArrayElement(copiedArray);

min = getMinArrayElement(copiedArray);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

//cout << "Min: " << min << "\tMax: " << max << endl; // Отладка

cout << "Elements founds in sorted array for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ЧЕТВЁРТОГО ЗАДАНИЯ

cout << "\n\tTask 4. Search average elements from array. \n";

int averageElement = int(abs(getMaxArrayElement(copiedArray)) - abs(getMinArrayElement(copiedArray)));

cout << "Average element: " << averageElement << endl;

bool isExists = false; // Проверка на то, существует ли вообще в массиве подходящий средний элемент

for (int i = 0; i < sizeOfArray; i++)

if (copiedArray[i] == averageElement) {

isExists = true;

break;

}

if (isExists) {

cout << "Index(es) of average elements: ";

int count = 0;

for (int i = 0; i < sizeOfArray; i++)

if (copiedArray[i] == averageElement) {

cout << i << " ";

count++;

}

cout << "\nCount of average elements in array: " << count << endl;

}

else

cout << "Element not found in array. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ПЯТОГО ЗАДАНИЯ

cout << "\n\tTask 5. The number of smaller elements than the one entered by the user. \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

int count = 0;

for (int element : copiedArray)

if (element < input)

count++;

cout << "The array has " << count << " element(s) less than you entered. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ШЕСТОГО ЗАДАНИЯ

cout << "\n\tTask 6. The number of larger elements than the one entered by the user. \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

int count = 0;

for (int element : copiedArray)

if (element > input)

count++;

cout << "The array has " << count << " element(s) larger than you entered. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ СЕДЬМОГО ЗАДАНИЯ

cout << "\n\tTask 7. The searching of a user-entered element in the array (traditional and binary search). \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

startTimer = steady\_clock::now();

bool isExists = false;

for (int element : copiedArray) // Поиск обычным перебором

if (element == input) {

isExists = true;

break;

}

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

if (isExists)

cout << "The entered element is in the array. \n";

else

cout << "The entered element is not in the array. \n";

cout << "The search has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

startTimer = steady\_clock::now();

int searchedElement = binarySearch(copiedArray, input, 0, sizeOfArray - 1);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

/\* Отладка

if (searchedElement != -1)

cout << "Searched element (binary search): " << searchedElement << endl;

else cout << "Element not searched. \n"; \*/

cout << "The binary search has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ВОСЬМОГО ЗАДАНИЯ

cout << "\n\tTask 7. Switching array elements. \n"

<< "Enter the first and second indexes of array elements...\n>> ";

int input1, input2;

cin >> input1 >> input2;

startTimer = steady\_clock::now();

//if (((input1 && input2) >= 0) && ((input1 && input2) < sizeOfArray)) { // Ошибка компилятора о небезопасном условии. Разобраться, почему.

if ((input1 >= 0) && (input2 >= 0) && (input1 < sizeOfArray) && (input2 < sizeOfArray)) {

int t = copiedArray[input1];

copiedArray[input1] = copiedArray[input2];

copiedArray[input2] = t;

}

else {

cout << "Wrong input! \n";

continue;

}

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

printArray(copiedArray); // Отладка

cout << "The switching has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

cout << "Closing the program... \n";

system("pause");

return 0;

}

void createAndPrintRandomArray(int inputArray[]) {

for (int i = 0; i < sizeOfArray; i++) {

inputArray[i] = -99 + rand() % 199; // В массив пишутся случайные целые числа диапазона -99..99 включительно. При желании данную функцию можно расширить возможностью выбирать граничные значения.

cout << inputArray[i] << "\t";

if ((i+1) % 10 == 0) // Вывод по 10 элементов для удобочитаемости

cout << endl;

}

}

void bubbleSort(int inputArray[]) {

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++) {

if (inputArray[i] > inputArray[i + 1]) { // Соседние элементы массива меняются местами

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

}

} while (swapped == true); // Если за последний проход цикла никакие элементы не поменялись местами, цикл прерывается.

}

void shakerSort(int inputArray[]) {

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++)

if (inputArray[i] > inputArray[i + 1]) {

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

for (int i = sizeOfArray - 1; i > 0; i--)

if (inputArray[i] < inputArray[i - 1]) {

int t = inputArray[i];

inputArray[i] = inputArray[i - 1];

inputArray[i - 1] = t;

swapped = true;

}

} while (swapped);

}

void combSort(int inputArray[]) {

int swap;

float k = 1.247F, sortingRange = sizeOfArray - 1;

while (sortingRange >= 1)

{

for (int i = 0; i + sortingRange < sizeOfArray; i++)

{

if (inputArray[i] > inputArray[int(i + sortingRange)])

{

swap = inputArray[int(i + sortingRange)];

inputArray[int(i + sortingRange)] = inputArray[i];

inputArray[i] = swap;

}

}

sortingRange /= k;

}

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++) {

if (inputArray[i] > inputArray[i + 1]) { // Соседние элементы массива меняются местами

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

}

} while (swapped == true); // Если за последний проход цикла никакие элементы не поменялись местами, цикл прерывается.

}

void insertSort(int inputArray[]) {

int extractedElement, currentPosition;

for (int i = 1; i < sizeOfArray; i++)

{

extractedElement = inputArray[i];

currentPosition = i;

while ((currentPosition >= 1) && (extractedElement < inputArray[currentPosition - 1]))

{

inputArray[currentPosition] = inputArray[currentPosition - 1];

currentPosition--;

}

inputArray[currentPosition] = extractedElement;

}

}

void quickSort(int inputArray[], int end, int begin) {

int middleElement;

int f = begin;

int l = end;

middleElement = inputArray[(f + l) / 2]; // Определяется опорный элемент

while (f < l) { // Массив сортируется так, что по окончании цикла будет представлять собой две части: элементы меньше либо равные опорному (левая), элементы больше либо равные опорному (правая)

while (inputArray[f] < middleElement) f++;

while (inputArray[l] > middleElement) l--;

if (f <= l)

{

int swap = inputArray[f];

inputArray[f] = inputArray[l];

inputArray[l] = swap;

f++;

l--;

}

}

if (begin < l) quickSort(inputArray, l, begin); // Где l - правая граница левой части разделённого пополам массива

if (f < end) quickSort(inputArray, end, f); // Где f - левая граница правой части разделённого пополам массива

}

int getMinArrayElement(int inputArray[]) {

int minElement = INT\_MAX;

for (int i = 0; i < sizeOfArray; i++)

if (inputArray[i] < minElement)

minElement = inputArray[i];

return minElement;

}

int getMaxArrayElement(int inputArray[]) {

int maxElement = INT\_MIN;

for (int i = 0; i < sizeOfArray; i++)

if (inputArray[i] > maxElement)

maxElement = inputArray[i];

return maxElement;

}

int binarySearch(int inputArray[], int inputElement, int lowPos, int highPos) {

while (lowPos <= highPos) {

int mid = lowPos + (highPos - lowPos) / 2;

if (inputArray[mid] == inputElement)

return mid;

if (inputArray[mid] < inputElement)

lowPos = mid + 1;

else

highPos = mid - 1;

}

return -1;

}

void copyArray(int originalArray[], int resultArray[]) {

for (int i = 0; i < sizeOfArray; i++)

resultArray[i] = originalArray[i];

}

void printArray(int inputArray[]) {

for (int i = 0; i < sizeOfArray; i++) {

cout << inputArray[i] << "\t";

if ((i + 1) % 10 == 0) // Вывод по 10 элементов для удобочитаемости

cout << endl;

}

cout << endl;

}

bool choiseNextAction() {

cout << "\nFor repeat enter 1, for continue enter 2 or other integer number..." << "\n>> ";

int input;

cin >> input;

if (input == 1)

return true;

else return false;

}

float stopSecondsTimer(time\_point<steady\_clock> startTimer) {

using fseconds = duration<float>; // Определение кастомного интервала времени для отображение дробных секунд в таймере.

time\_point<steady\_clock> endTimer = steady\_clock::now(); // Остановка времени

fseconds sortingTime = duration\_cast<fseconds>(endTimer - startTimer); // Вычисление разницы между финальным и стартовым временем

return sortingTime.count();

}

long long stopNanoSecondsTimer(time\_point<steady\_clock> startTimer) {

time\_point<steady\_clock> endTimer = steady\_clock::now();

nanoseconds sortingTime = duration\_cast<nanoseconds>(endTimer - startTimer);

return sortingTime.count();

}