**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №2**

**по дисциплине «Программирование»**

Тема: ОДНОМЕРНЫЕ СТАТИЧЕСКИЕ МАССИВЫ

|  |  |  |
| --- | --- | --- |
| Студент(ка) гр. | Цатинян А.А. |  |
| Преподаватель | Глущенко А. Г. |  |

Санкт-Петербург

2022

**Цель работы.**

Изучение одномерных статических массивов. Изучение алгоритмов сортировки и поиска.

**Основные теоретические положения.**

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.

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

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.

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.

Алгоритм бинарного поиска – классический алгоритм поиска в отсортированном массиве, который использует дробление массива на половины. Если элемент, который необходимо найти, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден.

**Постановка задачи.**

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 массив […] 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.

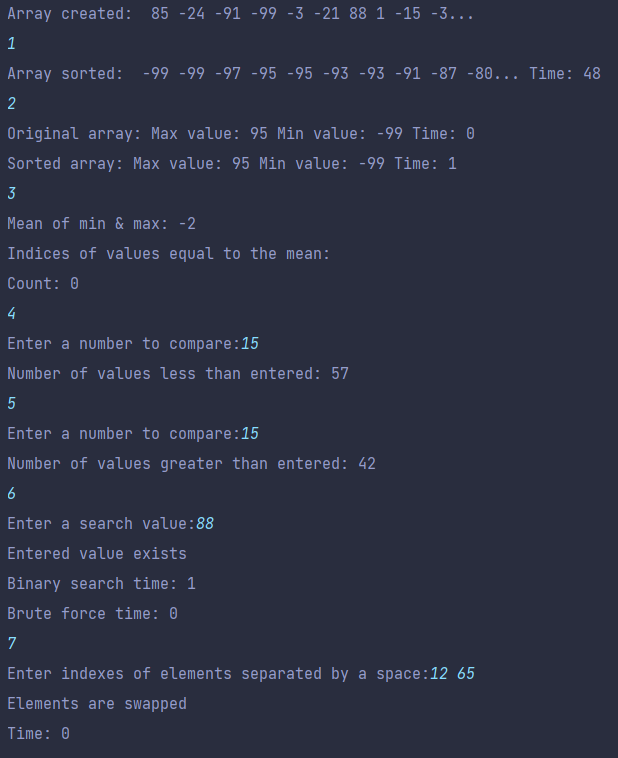
**Выполнение работы.**

Код программы представлен в приложении А.

Блок описания кода и использованных алгоритмов

Каждый пункт из постановки задачи сдвинут на единицу вниз, т.к. первый пункт выполняется единожды при запуске программы. Для выполнения первого пункта реализован алгоритм быстрой сортировки. Для выбора опорного элемента написан алгоритм подсчёта среднего значения и поиска среди значений массива, ближайшего к среднему. Для измерения времени выполнения той или иной функции используются объекты типа time\_point из библиотеки chrono. Для выполнения второго пункта реализован алгоритм поиска наибольшего и наименьшего значений перебором. Для выполнения третьего пункта не реализовывался алгоритм поиска max и min значений и алгоритм из второго не использовался, т.к. функция из второго пункта просто выводила найденные значения в консоль, ничего не возвращая. Поэтому я просто сортирую массив и работаю с первым и последним значением. В пятом и шестом пунктах просто перебором считаем количество элементов больше или меньше введённого значения. В седьмом пункте реализован алгоритм бинарного пункта без рекурсии. В восьмом пункте просто считываем два значения и меняем их местами как в любой сортировке.

Блок скриншотов работы программы



**Выводы.**

Изучен навык работы с одномерными статическими массивами. Изучены базовые алгоритмы сортировок и поиска. Изучен навык бинарного поиска.

Бинарный поиск может быть медленнее на таких множествах значений как у нас (малого размера), но при увеличении размера массива бинарный поиск будет показывать куда большую скорость, чем обычный перебор.

Приложение А

рабочий код

#include <iostream>  
#include <cstdlib>  
#include <cmath>  
#include <chrono>  
  
*using namespace* std;  
  
*int* \*quickSort(*int* \*arr, *const int* SIZE) {  
 *double* mean = 0;  
 *for* (*int* i = 0; i < SIZE; ++i) mean += abs(arr[i]);  
 mean /= SIZE;  
 *int* refNum = 0, refNumIndex;  
 *for* (*int* i = 0; i < SIZE; ++i) {  
 *if* (abs(abs(arr[i]) - abs(mean)) < abs(abs(refNum) - abs(mean))) {  
 refNum = arr[i];  
 refNumIndex = i;  
 }  
 }  
 *int* countOfLower = 0, countOfHigher = 0;  
 *for* (*int* i = 0; i < SIZE; ++i) {  
 *if* (i != refNumIndex) {  
 *if* (arr[i] >= refNum) {  
 countOfHigher++;  
 } *else* countOfLower++;  
 }  
 }  
  
 *int* lowerValues[countOfLower];  
 *int* higherValues[countOfHigher];  
 *int* j = 0, k = 0;  
 *for* (*int* i = 0; i < SIZE; ++i) {  
 *if* (i != refNumIndex) {  
 *if* (arr[i] >= refNum) {  
 higherValues[j] = arr[i];  
 j++;  
 } *else* {  
 lowerValues[k] = arr[i];  
 k++;  
 }  
 }  
 }  
 *if* (countOfLower > 2) {  
 quickSort(lowerValues, countOfLower);  
 } *else if* (countOfLower == 2) {  
 *if* (lowerValues[0] > lowerValues[1]) {  
 *int* t = lowerValues[0];  
 lowerValues[0] = lowerValues[1];  
 lowerValues[1] = t;  
 }  
 }  
 *if* (countOfHigher > 2) {  
 quickSort(higherValues, countOfHigher);  
 } *else if* (countOfHigher == 2) {  
 *if* (higherValues[0] > higherValues[1]) {  
 *int* t = higherValues[0];  
 higherValues[0] = higherValues[1];  
 higherValues[1] = t;  
 }  
 }  
 *for* (*int* i = 0; i < countOfLower; ++i) arr[i] = lowerValues[i];  
 arr[countOfLower] = refNum;  
 *for* (*int* i = countOfLower + 1; i < SIZE; ++i) arr[i] = higherValues[i - countOfLower - 1];  
 *return* arr;  
}  
  
*bool* binarySearch(*int* \*arr, *const int* SIZE, *int* value) {  
 *int* s = *int*(ceil(SIZE / 2.0));  
 *int* middleNum = arr[s - 1];  
 *int* len = SIZE;  
 *while* (len > 2) {  
 *if* (value < middleNum) {  
 s -= *int*(ceil(len / 4.0));  
 middleNum = arr[s - 1];  
 len = *int*(ceil(len / 2.0));  
 } *else if* (value > middleNum) {  
 s += *int*(ceil(len / 4.0));  
 middleNum = arr[s - 1];  
 len -= *int*(ceil(len / 2.0));;  
 } *else return true*;  
 }  
 *for* (*int* i = s - len; i < s + len; i++) {  
 *if* (arr[i] == value) *return true*;  
 }  
 *return false*;  
}  
  
*bool* bruteForce(*int* \*arr, *const int* SIZE, *int* value) {  
 *for* (*int* i = 0; i < SIZE; ++i) *if* (arr[i] == value) *return true*;  
 *return false*;  
}  
  
*void* findMinMax(*int* \*arr, *const int* SIZE) {  
 chrono::time\_point start = std::chrono::high\_resolution\_clock::now();  
 *int* min = 999, max = -999;  
 *for* (*int* i = 0; i < SIZE; ++i) {  
 *if* (arr[i] > max) max = arr[i];  
 *if* (arr[i] < min) min = arr[i];  
 }  
 chrono::time\_point end = std::chrono::high\_resolution\_clock::now();  
 cout << "Max value: " << max << " Min value: " << min << " Time: "  
 << chrono::duration\_cast<chrono::microseconds>(end - start).count() << endl;  
}  
  
*int* main() {  
 srand(time(0));  
 *int* arrValues[100], arrSorted[100];  
 *for* (*int* i = 0; i < 100; ++i) arrValues[i] = rand() % (99 + 99 + 1) - 99;  
 *for* (*int* i = 0; i < 100; ++i) arrSorted[i] = arrValues[i];  
 cout << "Array created: ";  
 *for* (*int* i = 0; i < 10; ++i) cout << " " << arrValues[i];  
 cout << "... " << endl;  
  
 *while* (*true*) {  
 *int* k;  
 cin >> k;  
 *if* (k == 0) {  
 *return* 0;  
 } *else if* (k == 1) {  
 *for* (*int* i = 0; i < 100; ++i) arrSorted[i] = arrValues[i];  
 chrono::time\_point start = std::chrono::high\_resolution\_clock::now();  
 quickSort(arrSorted, 100);  
 chrono::time\_point end = std::chrono::high\_resolution\_clock::now();  
 cout << "Array sorted: ";  
 *for* (*int* i = 0; i < 10; ++i) cout << " " << arrSorted[i];  
 cout << "... ";  
 cout << "Time: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << endl;  
 } *else if* (k == 2) {  
 cout << "Original array: ";  
 findMinMax(arrValues, 100);  
 cout << "Sorted array: ";  
 quickSort(arrSorted, 100);  
 findMinMax(arrSorted, 100);  
 } *else if* (k == 3) {  
 quickSort(arrSorted, 100);  
 *int* mean = round((arrSorted[0] + arrSorted[99]) / 2);  
 cout << "Mean of min & max: " << mean << endl;  
 cout << "Indices of values equal to the mean: ";  
 *int* counter = 0;  
 *for* (*int* i = 0; i < 100; ++i) {  
 *if* (arrValues[i] == mean) {  
 cout << i << " ";  
 counter++;  
 }  
 }  
 cout << endl << "Count: " << counter << endl;  
 } *else if* (k == 4) {  
 *int* a;  
 cout << "Enter a number to compare:";  
 cin >> a;  
 *int* counter = 0;  
 quickSort(arrSorted, 100);  
 *for* (*int* i: arrSorted) *if* (i < a) counter++;  
 cout << "Number of values less than entered: " << counter << endl;  
 } *else if* (k == 5) {  
 *int* b;  
 cout << "Enter a number to compare:";  
 cin >> b;  
 *int* counter = 0;  
 quickSort(arrSorted, 100);  
 *for* (*int* i: arrSorted) *if* (i > b) counter++;  
 cout << "Number of values greater than entered: " << counter << endl;  
 } *else if* (k == 6) {  
 *int* a;  
 cout << "Enter a search value:";  
 cin >> a;  
 quickSort(arrSorted, 100);  
 cout << (binarySearch(arrSorted, 100, a) ? "Entered value exists" : "Entered value does not exist") << endl;  
 chrono::time\_point binarySearchStart = std::chrono::high\_resolution\_clock::now();  
 binarySearch(arrSorted, 100, a);  
 chrono::time\_point binarySearchEnd = std::chrono::high\_resolution\_clock::now();  
 cout << "Binary search time: " <<  
 chrono::duration\_cast<chrono::microseconds>(binarySearchEnd - binarySearchStart).count() << endl;  
 chrono::time\_point bruteForceStart = std::chrono::high\_resolution\_clock::now();  
 bruteForce(arrValues, 100, a);  
 chrono::time\_point bruteForceEnd = std::chrono::high\_resolution\_clock::now();  
 cout << "Brute force time: " <<  
 chrono::duration\_cast<chrono::microseconds>(bruteForceEnd - bruteForceStart).count() << endl;  
 } *else if* (k == 7) {  
 *int* a, b;  
 cout << "Enter indexes of elements separated by a space:";  
 cin >> a >> b;  
 *while* (!(a >= 0 && a < 100 && b >= 0 && b < 100)) {  
 cout << "Wrong index!" << endl << "The index must be between 0 and 99:";  
 cin >> a >> b;  
 }  
 chrono::time\_point start = std::chrono::high\_resolution\_clock::now();  
 *int* t = arrValues[a];  
 arrValues[a] = arrValues[b];  
 arrValues[b] = t;  
 chrono::time\_point end = std::chrono::high\_resolution\_clock::now();  
 cout << "Elements are swapped" << endl << "Time: "  
 << chrono::duration\_cast<chrono::microseconds>(end - start).count() << endl;  
 }  
 }  
}