**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. |  | Адигюзалова А.А. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2024

**Цель работы.**

Изучение одномерных статических массивов, различных типов сортировок и сравнение скорости работы с отсортированным и неотсортированным массивом.

**Основные теоретические положения.**

**Массив** представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Все массивы можно разделить на две группы: одномерные и многомерные. Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных. Объявление в программах одномерных массивов выполняется в соответствии со следующим правилом:

**<Базовый тип элементов> <Идентификатор массива>[<Количество элементов>]**

Так, например:

int ArrInt [10], A1 [20];

Обращение к определенному элементу массива осуществляется с помощью указания значения индекса этого элемента:

A1 [4] = -1200;

cout << A1 [4];

При обращении к конкретному элементу массива этот элемент можно рассматривать как обычную переменную, тип которой соответствует базовому типу элементов массива, и осуществлять со значением этого элемента любые операции, которые характерны для базового типа.

При объявлении массива его можно инициализировать определенными значениями:

short S [5] = {1, 4, 9, 16, 25};

Или:

short S [] = {1, 4, 9, 16, 25};

**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.

**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.

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 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.

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

**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.

**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.

Бинарный поиск работает только в топ случае, если массив отсортирован. Например, если бы искомое минимальное значение стояло не на своем положенном месте, а на месте максимального элемента, то мы бы откинули его на первой же итерации. Сам алгоритм имеет вид:

1. Определение значения в середине массива (или иной структуры данных). Полученное значение сравнивается с ключом (значением, которое необходимо найти).
2. Если ключ меньше значения середины, то необходимо осуществлять поиск в первой половине элементов, иначе – во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет определен элемент, равный значению ключа или не станет пустым интервал для поиска.

**Постановка задачи.**

Необходимо разработать алгоритм и написать программу, которая позволяет:

1. Создать целочисленный массив размерности *N =* 100. Элементы массива принимают случайное значение в диапазоне от -99 до 99.
2. Отсортировать заданный раннее массив одной из доступных сортировок (bubble sort, shaker sort, comb sort, insert sort или quick sort) от меньшего к большему. Определить время, затраченное на сортировку, используя библиотеку chrono.
3. Найти максимальный и минимальный элемент массива. Подсчитать время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.
4. Вывести среднее значение максимального и минимального значения в отсортированном и неотсортированным массиве. Вывести индексы всех элементов, которые равны этому значению и их количество. Подсчитать время поиска.
5. Вывести количество элементов в отсортированном массиве, которые меньше числа *a,* котороеинициализируется пользователем.
6. Вывести количество элементов в отсортированном массиве, которые больше числа *b,* котороеинициализируется пользователем.
7. Вывести информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализовать алгоритм бинарного поиска. Сравнить скорость его работы с обычным перебором.
8. Поменять местами элементы массива, индексы которых вводит пользователь. Вывести скорость обмена, используя библиотеку

**Выполнение работы.**

Код программы представлен в приложении А.

Описание кода и использованных алгоритмов:

При запуске программы перед пользователем появляется окно, в котором на экран выводится

1. Неотсортированный и отсортированный массив элемент, а также время, затраченное на сортировку массива (сортировка quick sort)
2. Максимальный и минимальный элемент массива, а также время затраченное на их поиск в отсортированном и неотсортированном массиве соответсвенно.
3. Среднее арифметическое максимального и минимального элемента массива, есть ли такой элемент в массиве, а также время, затраченное на его поиск в отсортированном и неотсортированном массиве.
4. Интерактивное поле, где пользователь может задать любое число от -99 до 99. После ввода пользователем данных, на экран выводится количество элементов массива, которые меньше введенного пользователем значения.
5. Интерактивное поле, где пользователь может задать любое число от -99 до 99. После ввода пользователем данных, на экран выводится количество элементов массива, которые больше введенного пользователем значения.
6. Интерактивное поле, где пользователь может задать любое число. После ввода пользователем данных, на экран выводится индекс данного элемента в массиве, а также время затраченное для поиска этого элемента с использование бинарного и линейного поиска и разницы между ними.
7. Интерактивное поле, где пользователь может задать любое число от 0 до 99. После ввода пользователем данных, на экран выводится время, затраченное для того, чтобы поменять элементы с соответствующими индексами, а также измененный массив.
8. Интерактивное поле, где пользователь может выбрать, хочет ли он запустить программу еще раз.

Работа алгоритма и вывод на экран:

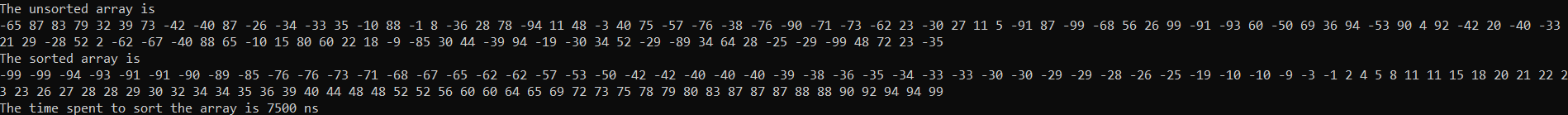


Рис. 1. Вывод на экран неотсортированного и отсортированного массива элементом

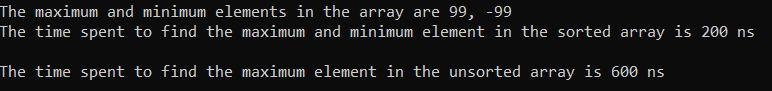


Рис.2 Вывод на экран максимального и минимального элемента массива; времени, затраченного на их поиск в отсортированном и неотсортированном массиве

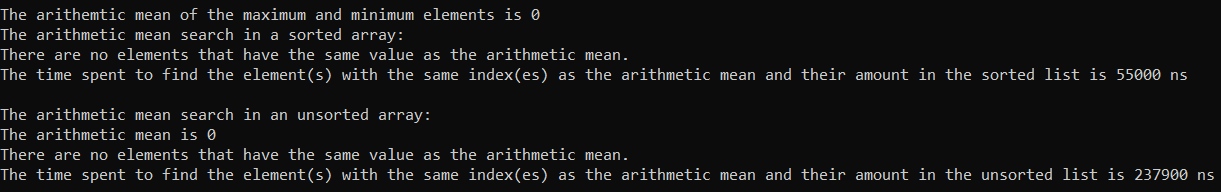


Рис. 3. Вывод на экран среднего арифметического максимального и минимального элементов; времени, затраченного на поиск элементов с таким же значением



Рис. 4. Интерактивное поле для ввода пользователем числа (в данном случае было введено число 8)



Рис. 5. Вывод на экран количества элементом, которые меньше введенного



Рис. 6. Интерактивное поле для ввода пользователем числа (в данном случае было введено число 0)



Рис. 7. Вывод на экран количества элементом, которые больше введенного 

Рис. 8. Интерактивное поле для ввода пользователем числа (в данном случае было введено число 45)

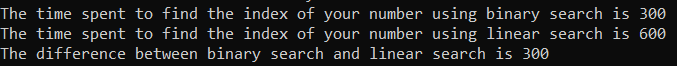


Рис. 9. Вывод на экран времени, затраченного на поиск индекса элемента, используя бинарный и линейный поиск соответственно.



Рис. 10. Интерактивное поле для ввода пользователем двух чисел (в данном случае были введены числа 5 и 7)

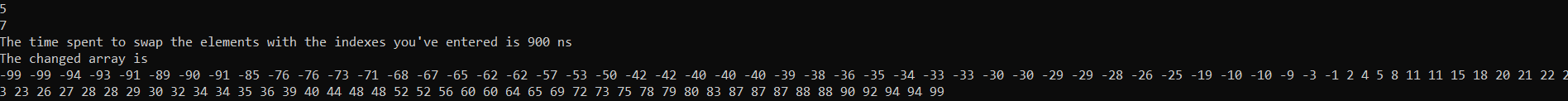


Рис. 11. Вывод на экран времени, затраченного на то, чтобы поменять местами элементы с соответствующими индексами и измененный массив.



Рис. 10. Интерактивное поле для ввода пользователем ответа на вопрос, хотят ли они запустить программу еще раз.

Тестовые данные:

|  |  |  |
| --- | --- | --- |
| Number between -99 and 99 | Number search | Two indexes |
| 8, -39, 68 | 45, 0, -4 | 5, 7, 24, 0 |

**Выводы.**

В ходе практической работы были изучены одномерные статические массивы и получены навыки работы с ними. Были получены знания о различных типах сортировок и их реализации на языке C++. Была разработана программа, которая позволила наглядно увидеть преимущества работы с отсортированными списками.

Приложение А

рабочий код

#include <cstdlib>

#include <iostream>

#include <chrono>

void quicksort(int\* arr, int end, int begin) {

int pivotElement;

int start = begin;

int finish = end;

pivotElement = arr[(start + finish) / 2];

while (start < finish) {

while (arr[start] < pivotElement) start++;

while (arr[finish] > pivotElement) finish--;

if (start <= finish) {

std::swap(arr[start], arr[finish]);

start++;

finish--;

}

}

if (begin < finish) quicksort(arr, finish, begin);

if (start < end) quicksort(arr, end, start);

}

int binarySearch(int arr[], int value, int start, int end) {

if (end >= start) {

int mid = start + (end - start) / 2;

if (arr[mid] == value) {

return mid;

}

if (arr[mid] > value) {

return binarySearch(arr, value, start, mid - 1);

}

else {

return binarySearch(arr, value, mid + 1, end);

}

}

return -1;

}

int sorting(int arr[], int arrayLength, int value) {

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

if (arr[i] == value) {

return i;

}

}

}

void sortedArraySearch(int arr[], int arrayLength) {

int maximum = arr[arrayLength-1];

int minimum = arr[0];

}

void unsortedArraySearch(int arr[], int arrayLength) {

int maximum = 0;

int minimum = 100;

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

if (arr[i] > maximum) {

maximum = arr[i];

}

if (arr[i] < minimum) {

minimum = arr[i];

}

}

}

void arithmeticMeanSearchSortedArray(int arr[], int arrayLength, int arithmeticMean) {

int count = 0;

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

if (arr[i] == arithmeticMean) {

count++;

}

}

if (count != 0) {

std::cout << "The index(es) of the element(s) that have the same value as the arithmetic mean of the maximum and minumum elements is/are ";

for (int j = 0; j < arrayLength; j++) {

if (arr[j] == arithmeticMean){

std::cout << j << " ";

}

}

std::cout << std::endl;

std::cout << "The amount of such elements is " << count << std::endl;

}

else {

std::cout << "There are no elements that have the same value as the arithmetic mean.";

}

std::cout << std::endl;

}

void arithmeticMeanSearchUnsortedArray(int arr[], int arrayLength, int arithmeticMean) {

int maximum = 0;

int minimum = 100;

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

if (arr[i] > maximum) {

maximum = arr[i];

}

if (arr[i] < minimum) {

minimum = arr[i];

}

}

std::cout << "The arithmetic mean is " << arithmeticMean << std::endl;

int count = 0;

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

if (arr[i] == arithmeticMean) {

count++;

}

}

if (count != 0) {

std::cout << "The index(es) of the element(s) that have the same value as the arithmetic mean of the maximum and minumum elements is/are ";

for (int j = 0; j < arrayLength; j++) {

if (arr[j] == arithmeticMean) {

std::cout << j << " ";

}

}

std::cout << std::endl;

std::cout << "The amount of such elements is " << count << std::endl;

}

else {

std::cout << "There are no elements that have the same value as the arithmetic mean.";

}

std::cout << std::endl;

}

void changingElements(int arr[], int index1, int index2) {

std::swap(arr[index1], arr[index2]);

}

int main() {

char userAnswer;

do {

const int arrayLength = 100;

int unsortedArr[arrayLength];

int sortedArr[arrayLength];

srand(time(0));

std::cout << "The unsorted array is " << std::endl;

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

unsortedArr[i] = rand() % 199 - 99;

sortedArr[i] = rand() % 199 - 99;

std::cout << unsortedArr[i] << " ";

}

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

sortedArr[i] = unsortedArr[i];

}

std::cout << std::endl;

int end = arrayLength - 1, begin = 0;

auto start = std::chrono::steady\_clock::now();

quicksort(sortedArr, end, begin);

auto stop = std::chrono::steady\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(stop - start);

std::cout << "The sorted array is " << std::endl;

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

std::cout << sortedArr[i] << " ";

}

std::cout << std::endl << "The time spent to sort the array is " << duration.count() << " ns\n";

std::cout << std::endl;

std::cout << "The maximum and minimum elements in the array are ";

std::cout << sortedArr[arrayLength - 1] << ", " << sortedArr[0] << std::endl;

auto startSortedSearch = std::chrono::steady\_clock::now();

sortedArraySearch(sortedArr, arrayLength);

auto stopSortedSearch = std::chrono::steady\_clock::now();

auto durationSortedSearch = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopSortedSearch - startSortedSearch);

std::cout << "The time spent to find the maximum and minimum element in the sorted array is " << durationSortedSearch.count() << " ns\n";

std::cout << std::endl;

auto startUnsortedSearch = std::chrono::steady\_clock::now();

unsortedArraySearch(unsortedArr, arrayLength);

auto stopUnsortedSearch = std::chrono::steady\_clock::now();

auto durationUnsortedSearch = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopUnsortedSearch - startUnsortedSearch);

std::cout << "The time spent to find the maximum element in the unsorted array is " << durationUnsortedSearch.count() << " ns\n";

std::cout << std::endl;

int arithmeticMean = (sortedArr[arrayLength - 1] + sortedArr[0]) / 2;

std::cout << "The arithemtic mean of the maximum and minimum elements is " << arithmeticMean << std::endl;

std::cout << "The arithmetic mean search in a sorted array: " << std::endl;

auto startArithmeticMeanSearchSorted = std::chrono::steady\_clock::now();

arithmeticMeanSearchSortedArray(sortedArr, arrayLength, arithmeticMean);

auto stopArithmeticMeanSearchSorted = std::chrono::steady\_clock::now();

auto durationArithmeticMeanSearchSorted = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopArithmeticMeanSearchSorted - startArithmeticMeanSearchSorted);

std::cout << "The time spent to find the element(s) with the same index(es) as the arithmetic mean and their amount in the sorted list is " << durationArithmeticMeanSearchSorted.count() << " ns\n";

std::cout << std::endl;

std::cout << "The arithmetic mean search in an unsorted array: " << std::endl;

auto startArithmeticMeanSearchUnsorted = std::chrono::steady\_clock::now();

arithmeticMeanSearchUnsortedArray(sortedArr, arrayLength, arithmeticMean);

auto stopArithmeticMeanSearchUnsorted = std::chrono::steady\_clock::now();

auto durationArithmeticMeanSearchUnsorted = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopArithmeticMeanSearchUnsorted - startArithmeticMeanSearchUnsorted);

std::cout << "The time spent to find the element(s) with the same index(es) as the arithmetic mean and their amount in the unsorted list is " << durationArithmeticMeanSearchUnsorted.count() << " ns\n";

std::cout << std::endl;

std::cout << "Enter any number from -99 to 99: ";

int lowerNumber;

std::cin >> lowerNumber;

int countLowerNumbers = 0;

int indexLowerNumbers = 0;

while (sortedArr[indexLowerNumbers] < lowerNumber) {

countLowerNumbers++;

indexLowerNumbers++;

}

std::cout << "The amount of elements that are lower than your number is " << countLowerNumbers << std::endl;

std::cout << "Enter any number from -99 to 99: ";

int higherNumber;

std::cin >> higherNumber;

int countHigherNumbers = 0;

int indexHigherNumbers = 99;

while (sortedArr[indexHigherNumbers] > higherNumber) {

countHigherNumbers++;

indexHigherNumbers--;

}

std::cout << "The amount of elements that are higher than your number is " << countHigherNumbers << std::endl;

int size = sizeof(sortedArr) / sizeof(sortedArr[0]);

int value;

std::cout << "Please, enter any number: ";

std::cin >> value;

int answer = binarySearch(sortedArr, value, 0, size - 1);

if (answer == -1) {

std::cout << "There's no such number in the array." << std::endl;

}

else {

std::cout << "Your number's index in the array is " << answer << std::endl;

}

auto startComparisonBinarySearch = std::chrono::steady\_clock::now();

binarySearch(sortedArr, value, 0, size - 1);

auto stopComparisonBinarySearch = std::chrono::steady\_clock::now();

auto durationComparisonBinarySearch = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopComparisonBinarySearch - startComparisonBinarySearch);

std::cout << "The time spent to find the index of your number using binary search is " << durationComparisonBinarySearch.count() << std::endl;

auto startComparisonLinearSearch = std::chrono::steady\_clock::now();

sorting(sortedArr, arrayLength, value);

auto stopComparisonLinearSearch = std::chrono::steady\_clock::now();

auto durationComparisonLinearSearch = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopComparisonLinearSearch - startComparisonLinearSearch);

std::cout << "The time spent to find the index of your number using linear search is " << durationComparisonLinearSearch.count() << std::endl;

int durationDifference = durationComparisonBinarySearch.count() - durationComparisonLinearSearch.count();

if (durationDifference < 0) {

durationDifference \*= -1;

}

std::cout << "The difference between binary search and linear search is " << durationDifference << std::endl;

std::cout << "Enter two indexes from 0 to 99 " << std::endl;

int index1, index2;

std::cin >> index1 >> index2;

auto startChangingElements = std::chrono::steady\_clock::now();

changingElements(sortedArr, index1, index2);

auto stopChangingElements = std::chrono::steady\_clock::now();

auto durationChangingElements = std::chrono::duration\_cast<std::chrono::nanoseconds>(stopChangingElements - startChangingElements);

std::cout << "The time spent to swap the elements with the indexes you've entered is " << durationChangingElements.count() << " ns" << std::endl;

std::cout << "The changed array is " << std::endl;

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

std::cout << sortedArr[i] << " ";

}

std::cout << std::endl << "Do you want to run the programm again? (1 - yes, 2 - no) ";

std::cin >> userAnswer;

} while (userAnswer == '1');