**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра CАПР**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Вариант 2**

**Тема: Алгоритмы сортировки и поиска**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Точилин А.Е. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

### Постановка задачи

Реализовать алгоритмы для целочисленного массива: двоичный поиск, быстрая сортировка, сортировка пузырьком, глупая сортировка. Для массива символов реализовать алгоритм сортировки подсчетом.

### Описание реализуемых методов. Оценка временной сложности методов.

Для реализации был набор функций. Список функций представлен в табл. 1.

Таблица 1 – Описание и оценка сложности функций поиска и сортировки.

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Оценка временной сложности |
| int BinarySearch(int\* array, size\_t size, int value); | Функция двоичного поиска | О(log(n)) |
| void QuickSort(int\* array, int left, int right); | Функция быстрой сортировки | O(n\*log(n)) |
| void BogoSort(int\* array, size\_t size); | Функция глупой сортировки | O(n\*n!) |
| void CountingSort(char\* array, size\_t size); | Функция сортировки подсчетом | O(257\*n) |
| void BubbleSort(int\* array, size\_t size); | Функция сортировки пузырьком | O(n^2) |
| bool isSorted(int\* array, size\_t size); | Функция проверки является ли массив отсортированным, используется для глупой сортировки | О(n) |
| void shakeArray(int\* array, size\_t size); | Функция рандомного перемешивания массива, используется для глупой сортировки | О(n) |

Также были написаны функции для замеров скорости работы функция, для массивов данных длины: 10, 100, 1000, 10000, 100000; BubbleSort и QuickSort - measureBubbleSort() и void measureQuickSort() соответственно. На рис. 1 представлен вывод данных функция. Так же на рис.2 и 3 представлены графики замеров. Код всех функций представлен в приложении А.

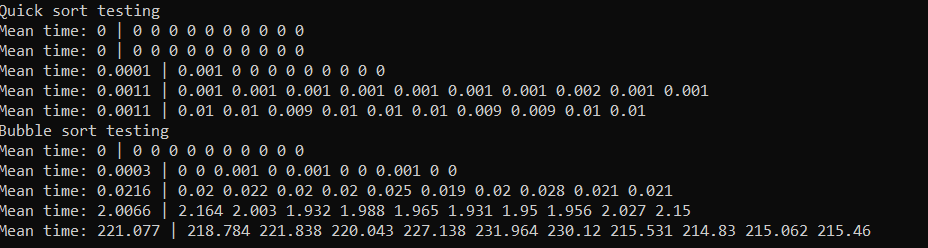


Рисунок 1 – замеры скорости работы функций сортировки

Рисунок 2 – График замера quicksort

Рисунок 3 – График замера bubblesort

**Описание реализованных unit-тестов**

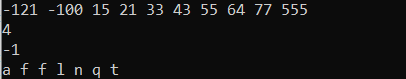
Для проверки работоспособности программы были реализованы unit-тесты. Бля этого был создан новый проект в Visual Studio, в котором написан класс LR2tests, методы класса и их описание представлены в табл. 2, код в приложении В.

Таблица 2 – Описание методов класса LR2tests

|  |  |
| --- | --- |
| Метод | Описание метода |
| TestBinarySearch | Проверяется работа функции бин. поиска, создается отсортированный массив, и с помощью функции BinarySearch, ищется каждый элемент |
| TestQuickSort | Проверяется работоспособность функции QuickSort, для этого создается массив, сортируется и проверяется, что каждый элемент на своем месте. |
| TestBubbleSort | Проверяется работоспособность функции BubbleSort, для этого создается массив, сортируется и проверяется, что каждый элемент на своем месте. |
| TestBogoSort | Проверяется работоспособность функции BogoSort, для этого создается массив, сортируется и проверяется, что каждый элемент на своем месте. |
| TestCountingSort | Проверяется работоспособность функции CountingSort, для этого создается массив, сортируется и проверяется, что каждый элемент на своем месте. |

### Пример работы

|  |  |
| --- | --- |
|  |  |



**Приложение А**

**листинг файла sortings.cpp**

#include <algorithm>

#include <random>

#include <chrono>

#include <iostream>

#include <ctime>

#include "sortings.h"

int BinarySearch(int\* array, size\_t size, int value) {

if (size == 0) {

std::cout << "Array length = 0" << std::endl;

return -1;

}

int first = 0, last = size, middle = 0;

while (first <= last) { //while index of first lower than index of last

middle = (first + last) / 2; //find middle of currnet interval

if (value == array[middle]) { //if find return middle

return middle;

}

else if (value > array[middle]) { //if value bigger than middel search in right interval

first = middle + 1;

}

else { //else search in left interval

last = middle - 1;

}

}

return -1;

}

void QuickSort(int\* array, int left, int right) {

int leftNumber = left, rightNumber = right; //right and left border of array

int pivot = array[(leftNumber + rightNumber) / 2]; //base element

int temp = 0;

while (leftNumber <= rightNumber) { //while borders not equal

while (array[leftNumber] < pivot)

leftNumber++; //change the left border while left border

while (array[rightNumber] > pivot) //is less then pivot

rightNumber--; //change the right border while right border

if (leftNumber <= rightNumber) { //is bigger then pivot

temp = array[leftNumber];

array[leftNumber] = array[rightNumber]; //change the left and the right element

array[rightNumber] = temp;

leftNumber++;

rightNumber--;

}

}

if (left < rightNumber) {

QuickSort(array, left, rightNumber);

}

if (leftNumber < right) {

QuickSort(array, leftNumber, right);

}

}

void measureQuickSort() {

std::cout << "Quick sort testing" << std::endl;

int array\_10[10], array\_100[100], array\_1000[1000], array\_10000[10000], array\_100000[100000];

double array\_10\_time[10], array\_100\_time[10], array\_1000\_time[10], array\_10000\_time[10], array\_100000\_time[10];

clock\_t start;

clock\_t end;

srand(time(0));

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 10; j++) {

array\_10[j] = rand() % 2000 - 1000;

}

start = clock();

QuickSort(array\_10, 0, 10 - 1);

end = clock();

array\_10\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 10 | " << "Mean time: " << meanArray(array\_10\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_10\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 100; j++) {

array\_100[j] = rand() % 2000 - 1000;

}

start = clock();

QuickSort(array\_100, 0, 100 - 1);

end = clock();

array\_100\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 100 | " << "Mean time: " << meanArray(array\_100\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_100\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 1000; j++) {

array\_1000[j] = rand() % 2000 - 1000;

}

start = clock();

QuickSort(array\_1000, 0, 1000 - 1);

end = clock();

array\_1000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 1000 | " << "Mean time: " << meanArray(array\_1000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_1000\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 10000; j++) {

array\_10000[j] = rand() % 2000 - 1000;

}

start = clock();

QuickSort(array\_10000, 0, 10000 - 1);

end = clock();

array\_10000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 10000 | " << "Mean time: " << meanArray(array\_10000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_10000\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 100000; j++) {

array\_100000[j] = rand() % 2000 - 1000;

}

start = clock();

QuickSort(array\_100000, 0, 100000 - 1);

end = clock();

array\_100000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 100000 | " << "Mean time: " << meanArray(array\_100000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_100000\_time[i] << " ";

}

std::cout << std::endl;

}

void BubbleSort(int\* array, size\_t size) {

if (size == 0) {

std::cout << "Array length = 0" << std::endl;

return;

}

for (size\_t i = 0; i < size; i++) {

for (size\_t j = 0; j < size - 1; j++) {

if (array[j] > array[j + 1]) { //swap elements

std::swap(array[j], array[j + 1]);

}

}

}

}

void measureBubbleSort() {

std::cout << "Bubble sort testing" << std::endl;

int array\_10[10], array\_100[100], array\_1000[1000], array\_10000[10000], array\_100000[100000];

double array\_10\_time[10], array\_100\_time[10], array\_1000\_time[10], array\_10000\_time[10], array\_100000\_time[10];

clock\_t start;

clock\_t end;

srand(time(0));

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 10; j++) {

array\_10[j] = rand() % 2000 - 1000;

}

start = clock();

BubbleSort(array\_10, 10);

end = clock();

array\_10\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 10 | " << "Mean time: " << meanArray(array\_10\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_10\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 100; j++) {

array\_100[j] = rand() % 2000 - 1000;

}

start = clock();

BubbleSort(array\_100, 100);

end = clock();

array\_100\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 100 | " << "Mean time: " << meanArray(array\_100\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_100\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 1000; j++) {

array\_1000[j] = rand() % 2000 - 1000;

}

start = clock();

BubbleSort(array\_1000, 1000);

end = clock();

array\_1000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 1000 | " << "Mean time: " << meanArray(array\_1000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_1000\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 10000; j++) {

array\_10000[j] = rand() % 2000 - 1000;

}

start = clock();

BubbleSort(array\_10000, 10000);

end = clock();

array\_10000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 10000 | " << "Mean time: " << meanArray(array\_10000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_10000\_time[i] << " ";

}

std::cout << std::endl;

for (size\_t i = 0; i < 10; i++) {

for (size\_t j = 0; j < 100000; j++) {

array\_100000[j] = rand() % 2000 - 1000;

}

start = clock();

BubbleSort(array\_100000, 100000);

end = clock();

array\_100000\_time[i] = (double)(end - start) / CLOCKS\_PER\_SEC;

}

std::cout << "Array length: 100000 | " << "Mean time: " << meanArray(array\_100000\_time, 10) << " | ";

for (size\_t i = 0; i < 10; i++) {

std::cout << array\_100000\_time[i] << " ";

}

std::cout << std::endl;

}

void BogoSort(int\* array, size\_t size) {

if (size == 0) {

std::cout << "Array length = 0" << std::endl;

return;

}

while (!isSorted(array, size)) { //while array is not sorted shake it

shakeArray(array, size);

}

}

void CountingSort(char\* array, size\_t size) {

if (size == 0) {

std::cout << "Array length = 0" << std::endl;

return;

}

size\_t frequency[256]{ 0 }; //counting array

for (size\_t i = 0; i < size; i++) { //count each symbol quantaty in array

frequency[array[i]]++;

}

size\_t position = 0;

for (size\_t number = 0; number <= 255; number++) { //go throug counting array

for (size\_t i = 0; i < frequency[number]; i++) { //and insert each symbol in array

array[position] = number;

position++;

}

}

}

bool isSorted(int\* array, size\_t size) {

while (size-- > 0) {

if (array[size - 1] > array[size]) { //if each previous symbol higher than next return false

return false;

}

}

return true;

}

void shakeArray(int\* array, size\_t size) {

std::random\_device rd;

std::mt19937 mersenne(rd());

for (size\_t i = 0; i < size; i++) {

std::swap(array[i], array[mersenne() % size]);

}

}

double meanArray(double\* array, size\_t size)

{

double summ = 0;

for (size\_t i = 0; i < size; i++)

{

summ += array[i];

}

return summ / size;

}

**Приложение В**

**листинг файла Lr2\_tests.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include "../lr2/sortings.h"

#include "../lr2/sortings.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lr2tests

{

TEST\_CLASS(Lr2tests)

{

public:

TEST\_METHOD(TestBinarySearch)

{

int numbers[6]{-1, 5, 12, 33, 64, 77};

Assert::AreEqual(BinarySearch(numbers, 6, -1), 0);

Assert::AreEqual(BinarySearch(numbers, 6, 5), 1);

Assert::AreEqual(BinarySearch(numbers, 6, 12), 2);

Assert::AreEqual(BinarySearch(numbers, 6, 33), 3);

Assert::AreEqual(BinarySearch(numbers, 6, 64), 4);

Assert::AreEqual(BinarySearch(numbers, 6, 77), 5);

}

TEST\_METHOD(TestQuickSort)

{

int numbers[6]{ 77, -1, 64, 33, 12, 5 };

QuickSort(numbers, 0, 5);

Assert::AreEqual(numbers[0], -1);

Assert::AreEqual(numbers[1], 5);

Assert::AreEqual(numbers[2], 12);

Assert::AreEqual(numbers[3], 33);

Assert::AreEqual(numbers[4], 64);

Assert::AreEqual(numbers[5], 77);

}

TEST\_METHOD(TestBubbleSort)

{

int numbers[6]{ 77, -1, 64, 33, 12, 5 };

BubbleSort(numbers, 6);

Assert::AreEqual(numbers[0], -1);

Assert::AreEqual(numbers[1], 5);

Assert::AreEqual(numbers[2], 12);

Assert::AreEqual(numbers[3], 33);

Assert::AreEqual(numbers[4], 64);

Assert::AreEqual(numbers[5], 77);

}

TEST\_METHOD(TestBogoSort)

{

int numbers[6]{ 77, -1, 64, 33, 12, 5 };

BogoSort(numbers, 6);

Assert::AreEqual(numbers[0], -1);

Assert::AreEqual(numbers[1], 5);

Assert::AreEqual(numbers[2], 12);

Assert::AreEqual(numbers[3], 33);

Assert::AreEqual(numbers[4], 64);

Assert::AreEqual(numbers[5], 77);

}

TEST\_METHOD(TestCountingSort)

{

char chars[6]{ 'm', 'c', 'a', 'f', 'b', 'd' };

CountingSort(chars, 6);

Assert::AreEqual(chars[0], 'a');

Assert::AreEqual(chars[1], 'b');

Assert::AreEqual(chars[2], 'c');

Assert::AreEqual(chars[3], 'd');

Assert::AreEqual(chars[4], 'f');

Assert::AreEqual(chars[5], 'm');

}

};

}