МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

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	O(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	O(n)
void shakeArray(int* array, size_t size);	Функция рандомного перемешивания массива, используется для глупой сортировки	O(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.

```
Quick sort testing
Mean time: 0 | 0 0 0 0 0 0 0 0 0 0
Mean time: 0 | 0 0 0 0 0 0 0 0 0 0
Mean time: 0.0001 | 0.001 0 0 0 0 0 0 0 0 0
Mean time: 0.0011 | 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.002 0.001 0.001
Mean time: 0.0011 | 0.01 0.01 0.009 0.01 0.01 0.01 0.009 0.009 0.01 0.01
Bubble sort testing
Mean time: 0 | 0 0 0 0 0 0 0 0 0 0
Mean time: 0.0003 | 0 0 0.001 0 0.001 0 0 0.001 0 0
Mean time: 0.0216 | 0.02 0.022 0.02 0.02 0.025 0.019 0.02 0.028 0.021 0.021
Mean time: 2.0066 | 2.164 2.003 1.932 1.988 1.965 1.931 1.95 1.956 2.027 2.15
Mean time: 221.077 | 218.784 221.838 220.043 227.138 231.964 230.12 215.531 214.83 215.062 215.46
```

Рисунок 1 – замеры скорости работы функций сортировки

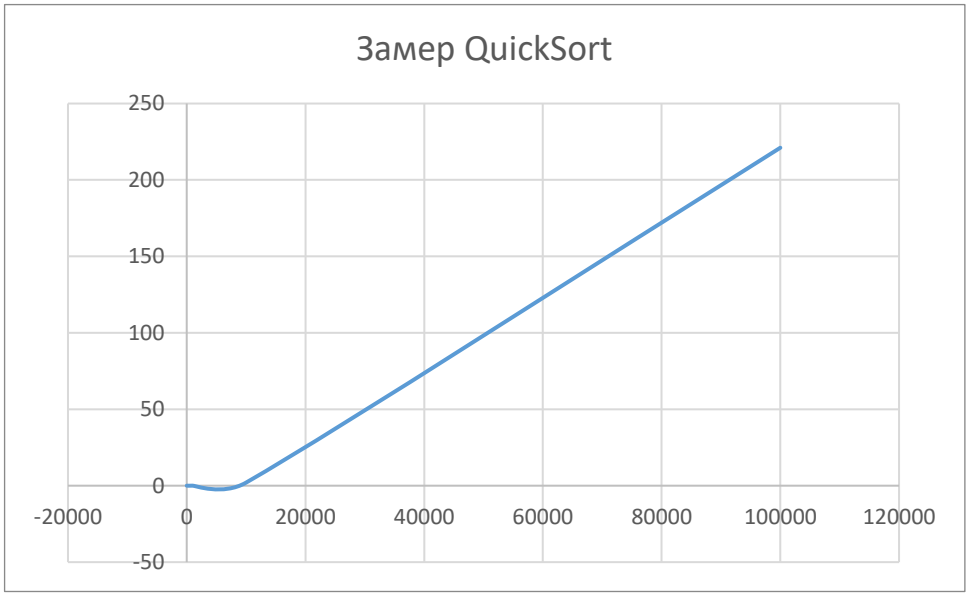

Рисунок 2 – График замера quicksort

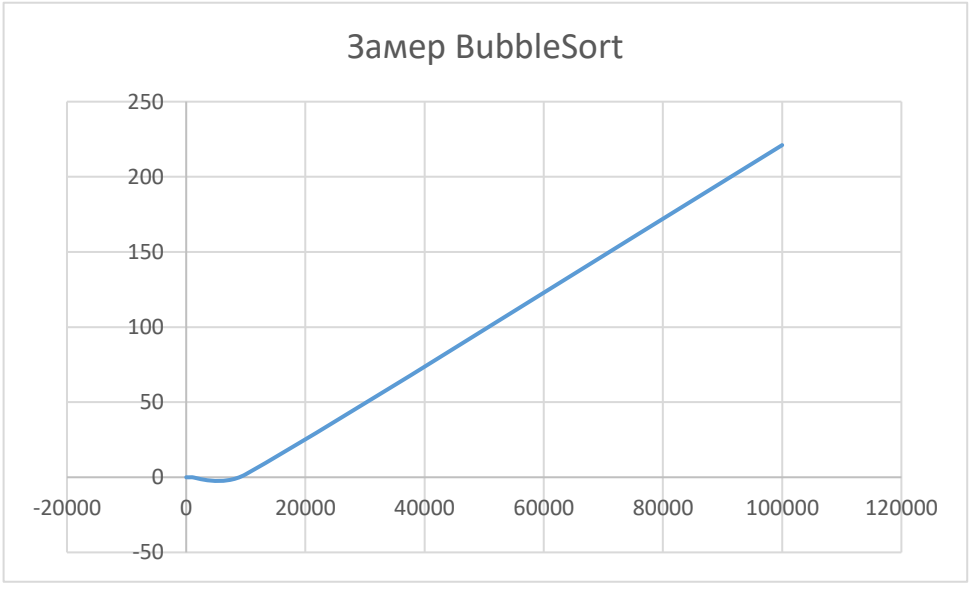

Рисунок 3 – График замера bubblesort

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

```
setlocale(LC_ALL, "rus");
int numbers[10]{ 15, 43, 21, 77, 64, 55, 33, -100, -121, 555 };
QuickSort(numbers, 0, 9);
for (size_t i = 0; i < 10; i++)
{
    std::cout << numbers[i] << " ";
}
std::cout << std::endl;
std::cout << BinarySearch(numbers, 10, 33) << std::endl;
std::cout << BinarySearch(numbers, 10, -33) << std::endl;
char chars[7]{ 'f', 'a', 'q', 't', 'f', 'l', 'n' };
CountingSort(chars, 7);
for (size_t i = 0; i < 7; i++)
{
    std::cout << chars[i] << " ";
}
std::cout << std::endl;
```

```
-121 -100 15 21 33 43 55 64 77 555
4
-1
a f f l n q t
```

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');
                }
        };
}
```