

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

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: Алгоритмы сортировки и поиска**  
**Вариант 2**

Студент гр. 0302

\_\_\_\_\_

Устинов Г.А.

Преподаватель

\_\_\_\_\_

Тутуева А.В.

Санкт-Петербург

2021

## **Цель работы**

Изучить алгоритмы поиска и сортировки, реализовать данные алгоритмы и провести сравнение скорости алгоритмов быстрой и пузырьковой сортировки.

## **Задание**

### **Вариант 2**

Отчет по лабораторной работе должен содержать:

1. Титульный лист с указанием варианта.
2. Постановка задачи.
3. Описание реализуемых алгоритмов.
4. Оценка временной сложности каждого алгоритма.
5. Сравнение временной сложности алгоритмов 2 и 3 (вариант 1) и 2 и 4 (вариант 2) по данным, полученным экспериментально, в виде графиков или таблиц. Замеры времени провести на наборах данных, сгенерированных случайным образом. Размерность данных 10, 100, 1000, 10 000, 100 000. Результирующие значения должны быть средними для 10-ти запусков (для каждого значения размерности). Замер времени автоматизировать.
6. Описание реализованных unit-тестов.
7. Пример работы.
8. Листинг.

Наличие unit-тестов к реализуемым функциям является обязательным требованием.

Все алгоритмы реализуются для целочисленного типа данных `int`, если не указано иное. Сортируем массивы.

Список алгоритмов:

1. Двоичный поиск (`BinarySearch`)
2. Быстрая сортировка (`QuickSort`)

3. Сортировка вставками (InsertionSort)
4. Сортировка пузырьком (BubbleSort)
5. Глупая сортировка (BogoSort)
6. Сортировка подсчётом (CountingSort) для типа char

#### Варианты

№ Варианта	Реализуемые алгоритмы
1	1, 2, 3, 5, 6
2	1, 2, 4, 5, 6

### Выполнение работы

#### Алгоритм двоичного поиска

*int binarySearch(int array[], size\_t n, int value)*

Алгоритм двоичного поиска возвращает индекс, по которому находится элемент с искомым значением в отсортированном массиве. Выбирается центральный элемент, и проверяется равно ли его значение искомому. Если это искомый элемент, то его индекс возвращается и алгоритм завершается. Иначе, массив разбивается на две части. Если искомый элемент меньше центрального, то алгоритм запускается для части с меньшими элементами, если же больше — то для части с большими элементами. Так, рекурсивно находится искомое значение. В реализации для этой работы, если элемент не найден, то функция, реализующая алгоритм возвращает -1.

В худшем случае (если элемент не будет найден, или если элемент будет найден на последней итерации) массив будет поделён  $\log N$  раз, потому что каждый раз массив делится пополам. Поскольку каждый раз рассматривается только центральный элемент рассматриваемого промежутка, то количество сравнений будет равно количеству разделений —  $\log N$ . Поэтому его временная сложность равняется  $\log N$ .

### **Алгоритм быстрой сортировки**

```
void quickSort(int array[], size_t n, size_t low, long long high )
```

Алгоритм быстрой сортировки упорядочивает элементы массива, причём этот алгоритм является одним из самых быстрых известных алгоритмов сортировки. В массиве выбирается опорный элемент (любым удобным способом), после чего все элементы меньше него помещаются в «левую» часть массива, а большие или равные — в «правую». Затем алгоритм рекурсивно применяется к левой и правой частям. В реализации для этой работы используется «Разбиение Хоара» - опорным выбирается средний элемент массива, после чего массив «проходится» с двух сторон одновременно, слева — в поисках элементов не меньше опорного, а справа — не больше. После нахождения таких элементов, если найденные элементы не являются одним и тем же — они меняются местами. Иначе, сохраняется индекс этого элемента. Для рекурсивного вызова массив делится относительно этого элемента «встречи».

В разработанной функции проверяются случаи неверных значений границ рассматриваемого участка массива. Для обработки этих ситуаций используются стандартные исключения *out\_of\_range* для индексов за границами массива и *invalid\_argument* для случая, когда левая граница больше правой.

Скорость работы алгоритма зависит от начальных данных и оценивается как количество разбиений массива на части и, следовательно, глубину рекурсивных вызовов. В худшем случае, когда каждый раз опорным значением выбирается наибольший или наименьший элемент массива, количество таких разбиений будет равно  $N^2$ . В других случаях, когда опорное значение стремится к медианному значению на промежутке, глубина вызовов будет логарифмом с различным основанием. В лучшем случае — когда

массив на каждом промежутке разбивается на два равных промежутка - это основание будет равно 2. На каждой глубине рекурсии необходимо совершить не более  $N$  перестановок и не более  $N$  сравнений. Следовательно сложность данного алгоритма -  $N \log N$

### ***Алгоритм сортировки пузырьком***

```
void bubbleSort(int array[], size_t n)
```

Алгоритм пузырьковой сортировки упорядочивает элементы массива следующим образом — массив проходится от начала до конца, и последовательно сравниваются соседние элементы. Если предыдущий элемент больше последующего, то они меняются местами. Если при каком-либо из проходов не было совершено ни одной перестановки, значит массив уже отсортирован. Этот алгоритм совершает минимум  $N$  проверок, при прохождении уже отсортированного массива. В худшем случае, когда массив упорядочен в обратном порядке, алгоритм совершит  $N^2$  проверок и перестановок, каждый раз «сдвигая» наибольший элемент вправо.

### ***Глупая сортировка***

```
void bogoSort(int array[], size_t n)
```

Алгоритм глупой сортировки действует следующим образом — он перемешивает элементы в случайном порядке, после чего проверяет, отсортирован ли он. Этот алгоритм не имеет худшего времени — теоретически, перемешивание может происходить вечно. В среднем же, его сложность составляет  $N \cdot N!$ .

### ***Сортировка подсчётом***

```
void countingSort(char array[], size_t n)
```

Сортировка подсчётом — алгоритм, который может применяться для сортировки значений, лежащих в пределах достаточно малого диапазона значений. Алгоритм создаёт массив счётчиков, по количеству значений в

диапазоне, после чего проходит по массиву и увеличивает счётчики встреченных значений. Затем совершается ещё один проход по массиву, в котором массив заполняется нужным количеством нужных значений согласно счётчикам. В данной работе всего совершается три прохода по массиву — для поиска максимального (и минимального, на тот случай, если символьный тип является знаковым) значения, после чего создаётся массив счётчиков нужного размера и совершается ещё один проход, для подсчёта значений. Третий проход — запись значений в массив. Время работы алгоритма линейно —  $N$ , однако требуется дополнительная память для хранения счётчиков — в худшем случае —  $N + 1$ .

### ***Экспериментальное сравнение быстрой и пузырьковой сортировок***

Для проведения сравнения была написана программа, которая создаёт массивы случайных значений нужных размерностей и замеряет время работы нужного алгоритма. Замер производится десять раз для каждой размерности, после чего считается среднее арифметическое этих значений для каждой размерности.

Таблица 1. Измерения производительности

Размерность	quickSort	bubbleSort
10	2 мкс	1 мкс
100	28 мкс	54 мкс
1000	388 мкс	4,5 мс
10 000	3,2 мс	562 мс
100 000	20 мс	1 мин 2 с

По результатам замеров можно судить, что функция пузырьковой сортировки выигрывает в скорости у быстрой сортировки при количестве элементов в сортируемых массивах меньше 100.

## Тестирование

Каждая функция сортировки тестируется для случаев пустого и непустого массивов. Также, функция быстрой сортировки тестируется на появление исключений при передаче заведомо неверных аргументов. На непустом массиве проверяется отсутствие исключительных ситуаций, а также упорядоченность элементов. После этого функция вызывается для уже отсортированного массива и также проверяется. Массивы генерируются случайным образом с использованием генератора случайных чисел *std::default\_random\_engine*.

Алгоритм поиска тестируется как нахождением существующих значений в массиве, так и не существующих, как в середине, так и больших и меньших диапазона значений массива.

Для тестирования используется фреймворк googletest.

## Пример работы

Следующий код служит примером работы с разработанными алгоритмами.

```
int *array = new int[data_size];  
fill_data(array, data_size);  
quickSort(array, data_size);  
cout << binarySearch(array, data_size, specific_value);
```

Разработанный программный код см. в приложении А.

## Выводы

Были изучены различные алгоритмы сортировки, а также алгоритм двоичного поиска, принципы их работы, а также написаны собственные реализации.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/sorts.hh**

```
#pragma once
#ifndef SORTS_HH
#define SORTS_HH

#include <cstdint>

namespace sorts {

void quickSort(int array[], size_t n, size_t low = 0, long long high =
-1);

void bubbleSort(int array[], size_t n);

void bogoSort(int array[], size_t n);

void countingSort(char array[], size_t n);

} // namespace sorts

#endif // SORTS_HH
```

Файл **src/sorts.cpp**

```
#include "sorts.hh"

#include <iostream>
#include <random>
#include <stdexcept>
using std::cout;
using std::endl;

namespace sorts {

void quickSort(int array[], size_t n, size_t low, long long high)
{
    if (n == 0)
        return;
    else if (high == -1)
        high = n - 1;
    else if ((size_t)high > n - 1)
        throw std::out_of_range("High is out of range.");
    if ((long long)low > high)
        throw std::invalid_argument("Low is greater than high.");
    if (high - low < 1)
```



```

        return;

    int pivot_index;
    int pivot_val = array[(low + high) / 2];
    size_t i = low;
    size_t j = high;
    while (true) {
        while (array[i] < pivot_val)
            ++i;
        while (array[j] > pivot_val)
            --j;
        if (i >= j) {
            pivot_index = j;
            break;
        }
        int tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
        ++i;
        --j;
    }
    quickSort(array, n, low, pivot_index);
    quickSort(array, n, pivot_index + 1, high);
}

```

```

void bubbleSort(int array[], size_t n)
{
    int buf;
    bool sorted;
    do {
        sorted = true;
        for (size_t i = 1; i < n; ++i) {
            if (array[i - 1] > array[i]) {
                buf = array[i - 1];
                array[i - 1] = array[i];
                array[i] = buf;
                sorted = false;
            }
        }
    } while (!sorted);
}

```

```

bool isSorted(int array[], size_t n)
{
    for (size_t i = 0; i < n - 1; ++i) {
        if (array[i] > array[i + 1])
            return false;
    }
    return true;
}

```

```

void bogoSort(int array[], size_t n)
{
    if (n < 2)
        return;

    std::default_random_engine rng;
    while (!isSorted(array, n)) {
        for (size_t i = 0; i < n; ++i) {
            size_t rand_index = rng() % n;
            int tmp = array[i];
            array[i] = array[rand_index];
            array[rand_index] = tmp;
        }
    }
}

void countingSort(char array[], size_t n)
{
    if (n < 2)
        return;

    // in case char is a signed type
    int max = array[0], min = array[0];
    for (size_t i = 1; i < n; ++i) {
        if ((int)array[i] > max)
            max = array[i];
        if ((int)array[i] < min)
            min = array[i];
    }
    int *counters = new int[max - min + 1];
    for (int i = min; i < max + 1; ++i)
        counters[i - min] = 0;
    for (size_t i = 0; i < n; ++i) {
        ++counters[array[i] - min];
    }
    for (int i = min, j = 0; i < max + 1; ++i) {
        while (counters[i - min]-- > 0)
            array[j++] = (char)i;
    }
}

} //namespace sorts

```

### Файл **src/searches.hh**

```

#pragma once
#ifndef SEARCHES_HH
#define SEARCHES_HH

#include <cstdint>

```

```

namespace searches {

int binarySearch(int array[], size_t n, int value);

} // namespace searches

#endif // SEARCHES_HH

```

#### Файл **src/searches.cpp**

```

#include "searches.hh"

namespace searches {

int binarySearch(int array[], size_t n, int value)
{
    if (array == nullptr || n == 0)
        return -1;
    // left - the lowest index of a slice to work on
    // right - the biggest index of a slice plus 1
    size_t left = 0, right = n;
    size_t mid;
    while (left < right) {
        mid = (left + right) / 2;
        if (value < array[mid])
            right = mid;
        else if (value > array[mid])
            left = mid + 1;
        else
            break;
    }
    return left < right ? mid : -1;
}

} // namespace searches

```

#### Файл **src/measure.cpp**

```

#include <iostream>
#include <chrono>
#include <random>

#include "sorts.hh"

int *generateRandomArray(size_t size)
{
    int *array = new int[size];
    std::default_random_engine rng;
    rng.seed(std::chrono::system_clock::now().time_since_epoch().count(

```

```

));
    for (size_t i = 0; i < size; ++i)
        array[i] = rng();
    return array;
}

int main()
{
    size_t sizes[] = {10, 100, 1000, 10000, 100000};
    std::chrono::steady_clock timer;
    std::cout << "Average quick sorting time on vector of:" <<
std::endl;
    for (size_t s : sizes) {
        std::cout << s << " elements: ";
        decltype(timer.now()) beg, fin;
        auto dur = beg - beg;
        for (int i = 0; i < 10; ++i) {
            int *array = generateRandomArray(s);
            beg = timer.now();
            sorts::quickSort(array, s);
            fin = timer.now();
            dur += fin - beg;
            delete[] array;
        }
        dur /= 10;
        std::cout <<
std::chrono::duration_cast<std::chrono::microseconds>(dur).count() <<
" us" << std::endl;
    }
    std::cout << "Average bubble sorting time on vector of:" <<
std::endl;
    for (size_t s : sizes) {
        std::cout << s << " elements: ";
        decltype(timer.now()) beg, fin;
        auto dur = beg - beg;
        for (int i = 0; i < 10; ++i) {
            int *array = generateRandomArray(s);
            beg = timer.now();
            sorts::bubbleSort(array, s);
            fin = timer.now();
            dur += fin - beg;
            delete[] array;
        }
        dur /= 10;
        std::cout <<
std::chrono::duration_cast<std::chrono::microseconds>(dur).count() <<
" us" << std::endl;
    }
    return 0;
}

```

Файл **test/lab2-sorts-test.cpp**

```
#include "../src/sorts.hh"

#include <iostream>
#include <random>

#include <gtest/gtest.h>

TEST(SortEmpty, QuickSort)
{
    int *a = nullptr;
    ASSERT_NO_THROW(sorts::quickSort(a, 0));
}

TEST(SortEmpty, BubbleSort)
{
    int *a = nullptr;
    ASSERT_NO_THROW(sorts::bubbleSort(a, 0));
}

TEST(SortEmpty, Bogosort)
{
    int *a = nullptr;
    ASSERT_NO_THROW(sorts::bogoSort(a, 0));
}

TEST(SortEmpty, CountingSort)
{
    char *a = nullptr;
    ASSERT_NO_THROW(sorts::countingSort(a, 0));
}

TEST(QuickSort, InvalidArguments)
{
    int a[] = {0};
    // let the high be too big
    ASSERT_THROW(sorts::quickSort(a, 1, 0, 1), std::out_of_range);
    // let low be lesser than high
    ASSERT_THROW(sorts::quickSort(a, 1, 1, 0), std::invalid_argument);
}

class SortTest : public ::testing::Test {
public:
    void SetUp() override
    {
        array_ = new int[100];
    }
};
```

```

        std::default_random_engine rng;
        for (size_t i = 0; i < 100; ++i)
            array_[i] = rng();
    }
    void TearDown() override
    {
        delete[] array_;
    }
protected:
    int *array_;
    size_t n_;
};

TEST_F(SortTest, BubbleSort)
{
    ASSERT_NO_THROW(sorts::bubbleSort(array_, n_));
    for (size_t i = 1; i < n_; ++i)
        ASSERT_TRUE(array_[i - 1] <= array_[i]);

    ASSERT_NO_THROW(sorts::bubbleSort(array_, n_));
    for (size_t i = 1; i < n_; ++i)
        ASSERT_TRUE(array_[i - 1] <= array_[i]);
}

TEST_F(SortTest, QuickSort)
{
    ASSERT_NO_THROW(sorts::quickSort(array_, n_));
    for (size_t i = 1; i < n_; ++i)
        ASSERT_TRUE(array_[i - 1] <= array_[i]);

    ASSERT_NO_THROW(sorts::quickSort(array_, n_));
    for (size_t i = 1; i < n_; ++i)
        ASSERT_TRUE(array_[i - 1] <= array_[i]);
}

TEST(BogoSort, SortTest)
{
    int array[] = {10, 5, 4, 7, 0, 1, 3, 8, 9, 9};
    ASSERT_NO_THROW(sorts::bogoSort(array, 10));
    for (size_t i = 1; i < 10; ++i)
        ASSERT_TRUE(array[i - 1] <= array[i]);
}

class CountingSortTest : public ::testing::Test {
public:
    void SetUp()
    {
        array_ = new char[1000];
        std::default_random_engine rng;
        for (size_t i = 0; i < 1000; ++i)

```

```

        array_[i] = rng() % 256;
        n_ = 1000;
    }
    void TearDown() override
    {
        delete[] array_;
    }
protected:
    char *array_;
    size_t n_;
};

TEST_F(CountingSortTest, SortTest)
{
    ASSERT_NO_THROW(sorts::countingSort(array_, n_));
    for (size_t i = 1; i < n_; ++i) {
        ASSERT_TRUE(array_[i - 1] <= array_[i]);
    }

    ASSERT_NO_THROW(sorts::countingSort(array_, n_));
    for (size_t i = 1; i < n_; ++i) {
        ASSERT_TRUE(array_[i - 1] <= array_[i]);
    }
}

```

### Файл **test/lab2-searches-test.cpp**

```

#include "../src/searches.hh"

#include <gtest/gtest.h>

using searches::binarySearch;

class BinarySearchTest : public ::testing::Test {
public:
    void SetUp() override
    {
        array_ = new int[11];
        n_ = 11;
        array_[0] = {-1};
        array_[1] = {1};
        array_[2] = {2};
        array_[3] = {3};
        array_[4] = {5};
        array_[5] = {7};
        array_[6] = {8};
        array_[7] = {9};
        array_[8] = {10};
        array_[9] = {26};
        array_[10] = {100};
    }
}

```

```

    }

    void TearDown() override
    {
        delete[] array_;
    }
protected:
    int *array_;
    size_t n_;
};

TEST_F(BinarySearchTest, SearchEveryOne)
{
    for (size_t i = 0; i < n_; ++i) {
        ASSERT_EQ(array_[i], array_[binarySearch(array_, n_,
array_[i])]);
    }
}

TEST_F(BinarySearchTest, SearchNonExistent)
{
    ASSERT_EQ(binarySearch(array_, n_, 0), -1);
}

TEST_F(BinarySearchTest, SearchNonExistentLesser)
{
    ASSERT_EQ(binarySearch(array_, n_, -10), -1);
}

TEST_F(BinarySearchTest, SearchNonExistentBigger)
{
    ASSERT_EQ(binarySearch(array_, n_, 120), -1);
}

```