МИНИСТЕРСТВО науки и высшего ОБРАЗОВАНИЯ РОссИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра № 304 «Вычислительные машины, системы и сети»

Программирование

Отчет по лабораторной работе № 2

Алгоритмы поиска

Выполнили студенты группы М3О-211Б-22

Коноплев К.А

Мустафаев Э.А Проверила доцент, к.т.н., Дмитриева Е.А.

Москва 2023 г.

Оглавление

[Задание 3](#_Toc151300276)

[Структурные схемы алгоритмов 4](#_Toc151300277)

[Функция Ordered\_Array\_Search 4](#_Toc151300278)

[Функция Binary\_Search 5](#_Toc151300279)

[Функция main 6](#_Toc151300280)

[Код программы 7](#_Toc151300281)

[Графики зависимости времени от размерности и расположения ключа 11](#_Toc151300282)

[Ключ расположен в начале 11](#_Toc151300283)

[Ключ расположен в середине 13](#_Toc151300284)

[Ключ расположен в конце 15](#_Toc151300285)

[Вывод 17](#_Toc151300286)

# Задание

Для алгоритмов **BLS** и **SLS** в качестве входного массива использовать одну и ту же последовательность значений (функция **rand**( ), можно использовать соответствующую функцию из первой лабораторной работы).

Для алгоритмов **OAS** и **ВS** – значения массива должны быть отсортированы по неубыванию, одна и та же последовательность чисел (можно использовать соответствующую функцию из первой лабораторной работы).

Оценить длительность поиска для различных значений размеров последовательностей (начиная с 10000 до 200000 элементов массива, провести измерения не менее, чем для 10 разных размерностей).

Для каждой размерности рассматриваются случаи нахождения ключа поиска в начале, в середине и в конце массива.

Для алгоритмов **BLS** и **SLS** кроме подсчета **времени,** необходимого для поиска, требуется определить сколько раз выполняются операции **сравнения** (сравнение ключа с элементом массива, а также в **BLS** добавляется подсчет сравнений при анализе индекса элемента массива в цикле… ).

Все результаты оформить в виде таблиц и графиков. На графиках **- только временные характеристики** поиска.

По результатам сделать выводы об эффективности того или иного алгоритма поиска.

# Структурные схемы алгоритмов

## Функция Ordered\_Array\_Search

Поиск элемента в отсортированном в порядке неубывания массиве

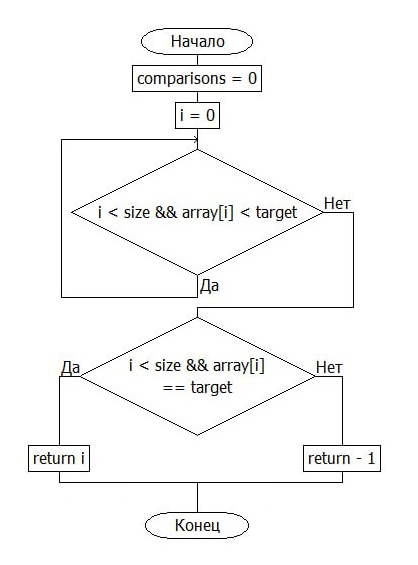


Рисунок 1. Структурная схема алгоритма функции Ordered\_Array\_Search.

## Функция Binary\_Search

Поиск элемента в отсортированном в порядке неубывания массиве с помощью бинарного поиска.

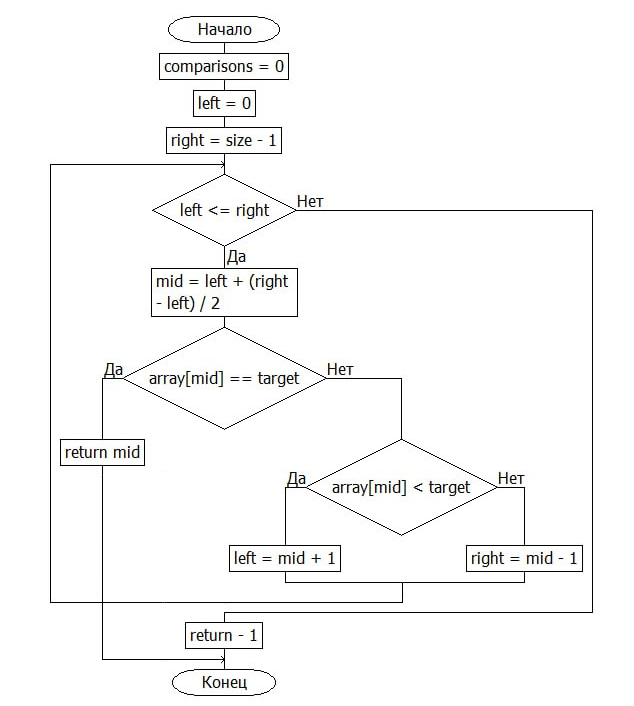


Рисунок 2. Структурная схема алгоритма функции Binary\_Search.

## C:\Users\melis\AppData\Local\Microsoft\Windows\INetCache\Content.Word\lab2 (2).png Функция main

Рисунок 3. Структурная схема алгоритма функции main.

# Код программы

#include <iostream>

#include <random>

#include <algorithm>

#include <chrono>

#include <fstream>

#include <windows.h>

using namespace std;

//функция удалления массва

void deleteArr(int\* arr) {

delete arr;

}

//функция линейного поиска

int Better\_Linear\_Search(const int array[], int size, int target, int& comparisons) {

comparisons = 0;

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

++comparisons; // увеличение счетчика при каждом сравнении ключа с элементом массива

if (array[i] == target) {

return i; // вернуть индекс, если элемент найден

}

}

return -1; // если элемент не найден

}

int Sentinel\_Linear\_Search(const int array[], int size, int target, int& comparisons) {

comparisons = 0;

int lastElement = array[size - 1];

// заменяем последний элемент массива на искомый элемент

int \*arrayWithSentinel = new int[size];

copy(array, array + size, arrayWithSentinel);

arrayWithSentinel[size - 1] = target;

int i = 0;

while (arrayWithSentinel[i] != target) {

++comparisons;

++i;

}

//восстанавливаем последний элемент массива

arrayWithSentinel[size - 1] = lastElement;

if (i < size - 1 || array[size - 1] == target) {

return i; // Вернуть индекс, если элемент найден

}

else {

return -1; // Вернуть -1, если элемент не найден

}

deleteArr(arrayWithSentinel);

}

// Функция линейного поиска в упорядоченном массиве

int Ordered\_Array\_Search(const int array[], int size, int target, int& comparisons) {

comparisons = 0;

int i = 0;

while (i < size && array[i] < target) {

++comparisons;

++i;

}

if (i < size && array[i] == target) {

return i; // Вернуть индекс, если элемент найден

}

else {

return -1; // Вернуть -1, если элемент не найден

}

}

// Функция бинарного поиска в упорядоченном массиве

int Binary\_Search(const int array[], int size, int target, int& comparisons) {

comparisons = 0;

int left = 0;

int right = size - 1;

while (left <= right) {

++comparisons;

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid; // Вернуть индекс, если элемент найден

}

else if (array[mid] < target) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1; // Вернуть -1, если элемент не найден

}

//функция для измерения времени поиска

template<typename Search>

pair<chrono::nanoseconds, int> measureSearchAndCountComparisons(Search searchFunction, const int array[], int size, int target) {

auto startTime = chrono::high\_resolution\_clock::now(); // Запоминаем время начала выполнения функции

int comparisons; // Переменная для хранения количества операций сравнения

int results = searchFunction(array, size, target, comparisons); // // Выполняем поиск и получаем количество операций сравнения

auto endTime = chrono::high\_resolution\_clock::now(); // Запоминаем время окончания выполнения функции

auto passedTime = chrono::duration\_cast<chrono::nanoseconds>(endTime - startTime); // Вычисляем время выполнения в миллисекундах

return{ passedTime, comparisons };//вернуть пару значений: время выполнения и количество операций сравнения

}

int main() {

//инициализация генератора случайных чисел с использованием библиотеки random

random\_device rd;// создание объекта случайных чисел

mt19937 gen(rd()); // создание случайных чисел с хорошим стат свойством и длинным периодом

uniform\_int\_distribution<int> distribution(1, 100);// создание диапазона от 1 до 100

const int maxSize = 200000;

const int step = 10000;

fstream ftime;

ftime.open("test.txt");

ftime << "BLS\tSLS\tOAS\tBS" << endl;

//итерация по различным размерам массива

for (int size = step; size <= maxSize; size += step) {

int \*array = new int[size]; // используем массив для хранения элементов

//Заполнения массива случайными числами и сортировка по неубыванию

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

array[i] = distribution(gen);

}

sort(array, array + size);

// Выбираем случайный элемент для поиска

int target = array[size % 2];

// Измеряем время выполнения и подсчитываем операции сравнения для Better Linear Search

auto linearSearchResult = measureSearchAndCountComparisons(Better\_Linear\_Search, array, size, target);

cout << "Better Linear Search for size " << size << ": " << linearSearchResult.first.count() << " ns," <<

linearSearchResult.second << " comparisons" << std::endl;

ftime << linearSearchResult.first.count() << '\t';

// Измеряем время выполнения и подсчитываем операции сравнения для Sentinel Linear Search

auto sentinalSearchResult = measureSearchAndCountComparisons(Sentinel\_Linear\_Search, array, size, target);

cout << "Sentinel Linear Search for size " << size << ": "

<< sentinalSearchResult.first.count() << " ns, "

<< sentinalSearchResult.second << " comparisons" << endl;

ftime << sentinalSearchResult.first.count() << '\t';

// Измеряем время выполнения и подсчитываем операции сравнения для Binary Search

auto binarySearchResult = measureSearchAndCountComparisons(Binary\_Search, array, size, target);

cout << "Binary Search for size " << size << ": "

<< binarySearchResult.first.count() << " ns, " << binarySearchResult.second << " comparisons" << endl;

ftime << binarySearchResult.first.count() << '\t';

// Измеряем время выполнения и подсчитываем операции сравнения для Ordered Array Search

auto orderedSearchResult = measureSearchAndCountComparisons(Ordered\_Array\_Search, array, size, target);

cout << "Ordered Array Search for size " << size << ": " << orderedSearchResult.first.count() << " ns," <<

orderedSearchResult.second << " comparisons" << endl;

ftime << orderedSearchResult.first.count() << '\t' << '\n';

cout << endl;// Печатаем пустую строку для разделения результатов

deleteArr(array);

}

ftime.close();

return 0;

}

# Графики зависимости времени от размерности и расположения ключа

## Результаты по временным интервалам для массива с ключом расположенном в начале

Для Better\_Linear\_Search:

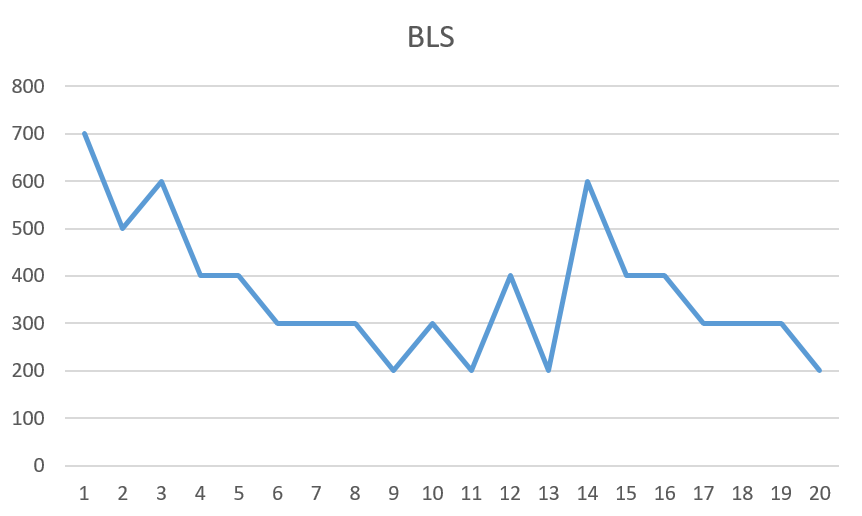


Рисунок 4. График зависимости времени от размерности

Для Sentinel\_Linear\_Search:

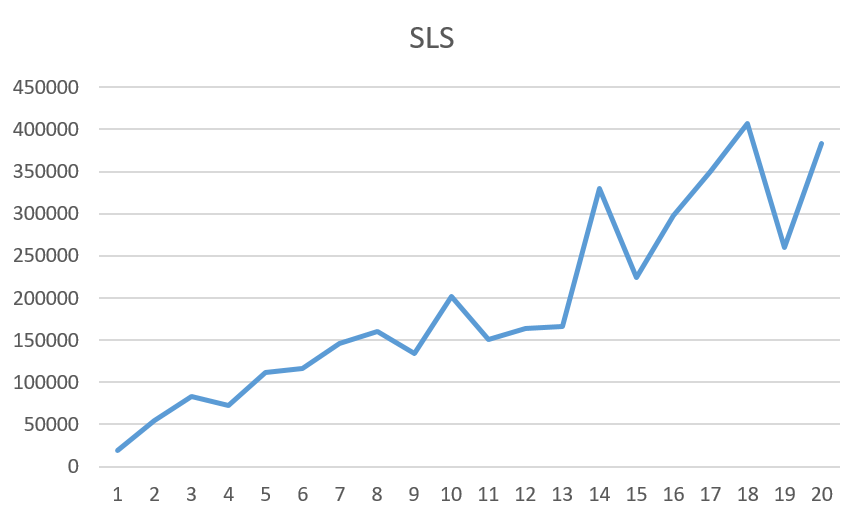


Рисунок 5. График зависимости времени от размерности

Для Ordered\_Array\_Search:

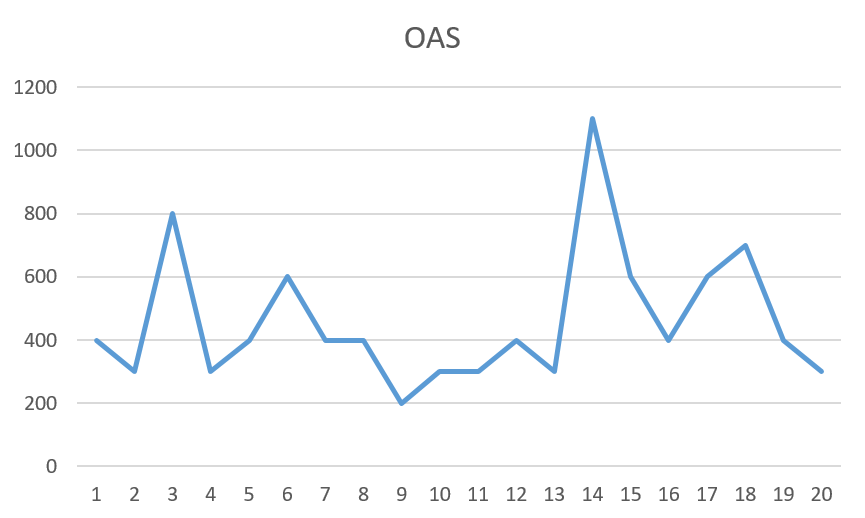


Рисунок 6. График зависимости времени от размерности

Для Binary\_Search:

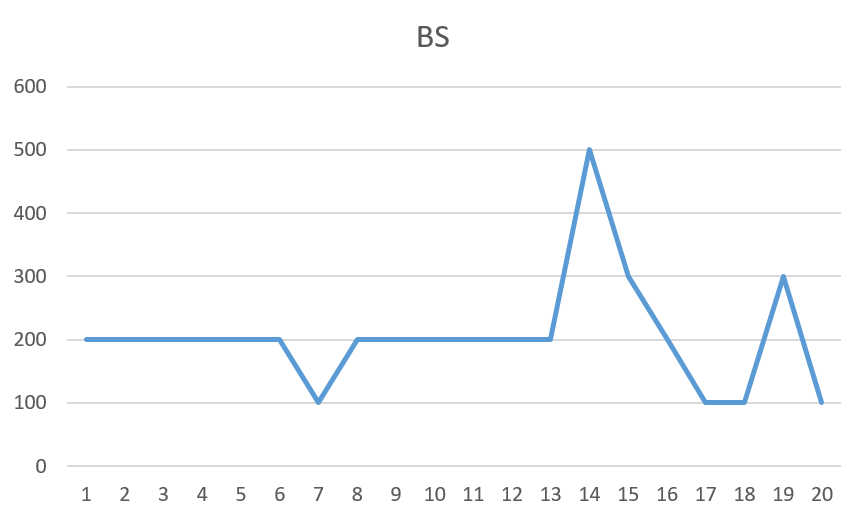


Рисунок 7. График зависимости времени от размерности

## Результаты по временным интервалам для массива с ключом расположенном в середине

Для Better\_Linear\_Search:

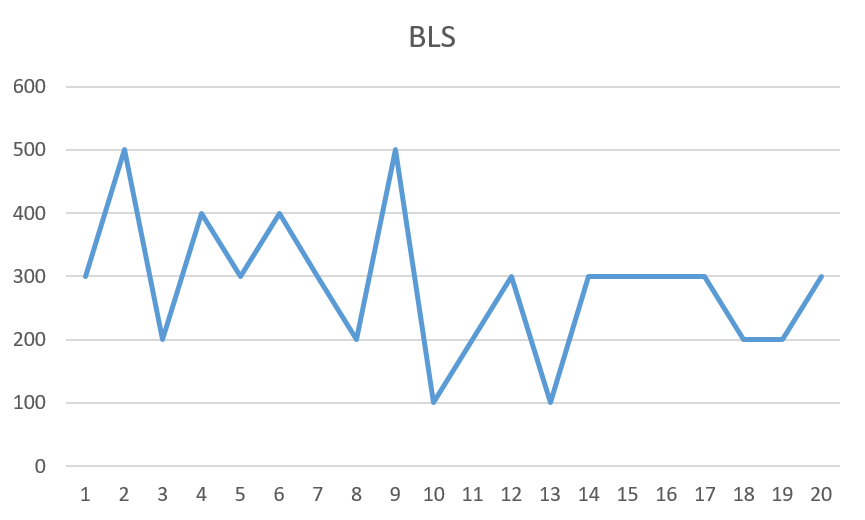


Рисунок 8. График зависимости времени от размерности

Для Sentinel\_Linear\_Search:

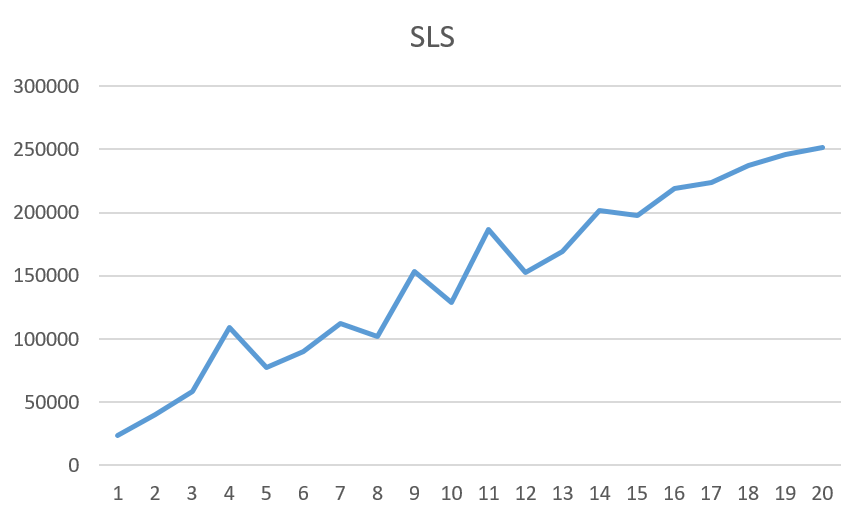


Рисунок 9. График зависимости времени от размерности

Для Ordered\_Array\_Search:

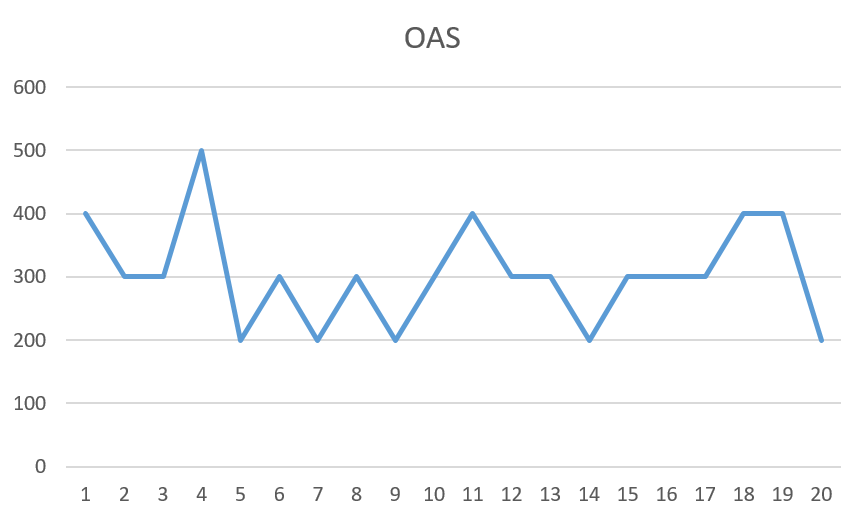


Рисунок 10. График зависимости времени от размерности

Для Binary\_Search:

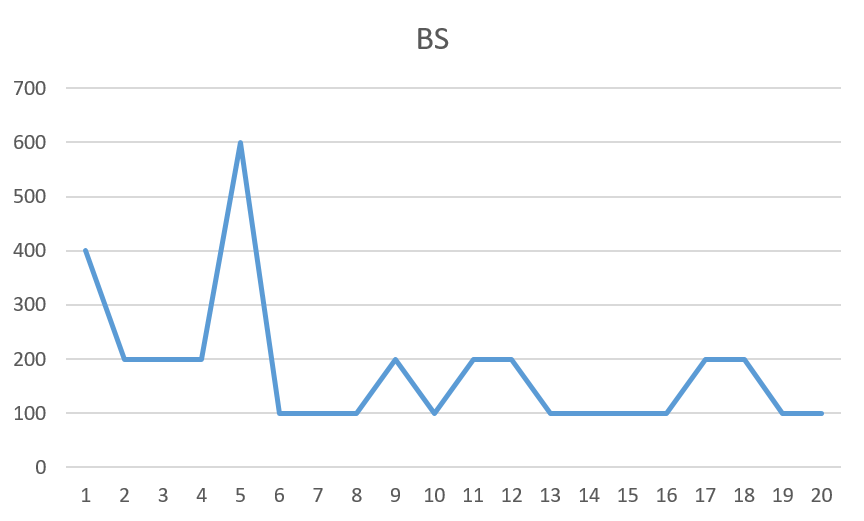


Рисунок 11. График зависимости времени от размерности

## Результаты по временным интервалам для массива с ключом расположенном в конце

Для Better\_Linear\_Search:

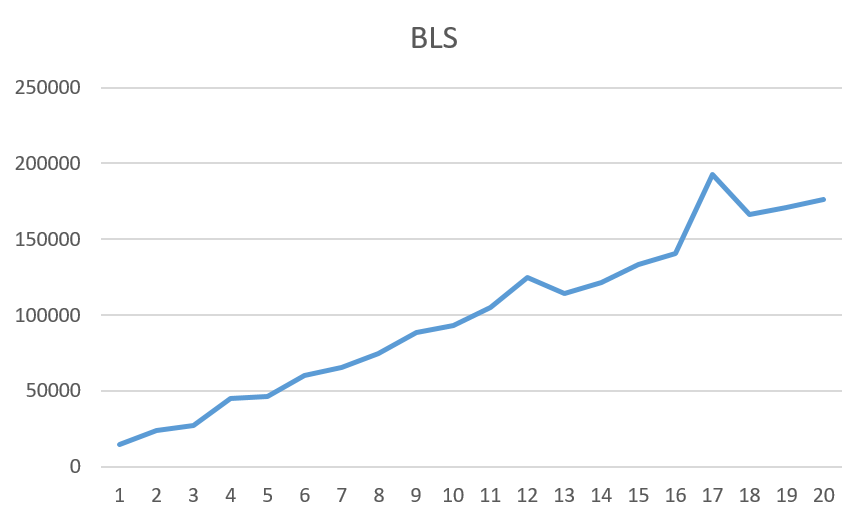


Рисунок 12. График зависимости времени от размерности

Для Sentinel\_Linear\_Search:

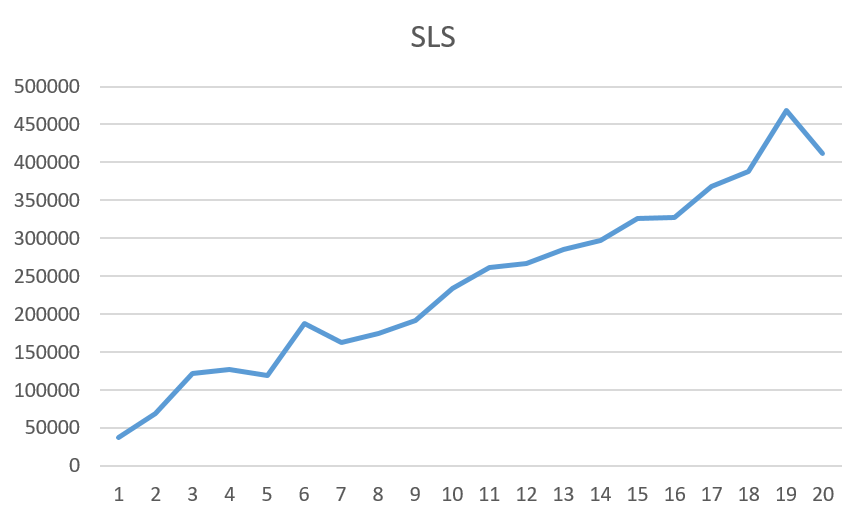


Рисунок 13. График зависимости времени от размерности

Для Ordered\_Array\_Search:

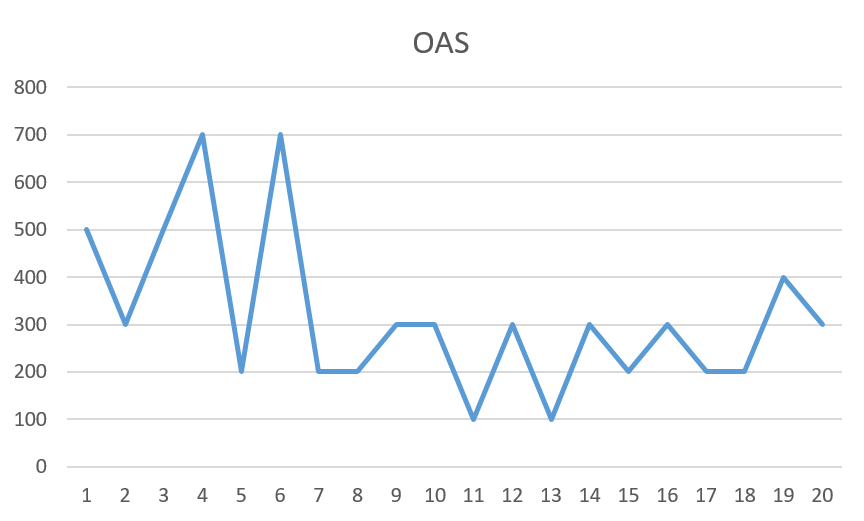


Рисунок 14. График зависимости времени от размерности

Для Binary\_Search:

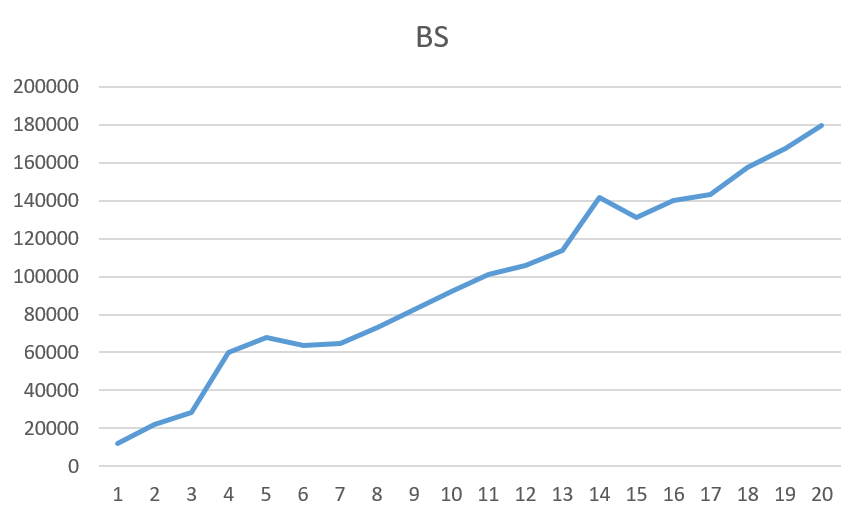


Рисунок 15. График зависимости времени от размерности

# Вывод

Программа работает корректно на основе проделанных тестов. Были освоены принципы формирования монотонных возрастающих и убывающих массивов, а также пилообразных, ступенчатых и синусоидальных массивов. Также были изучены функции, позволяющие производить оценку длительности времени выполнения алгоритмов с учетом вызова функций, а именно функция steady\_clock() из библиотеки chrono. Были изучены и использованы правила оформления функций с использованием указателей на функции в качестве формальных параметров. В ходе проведения лабораторной работы было выяснено:

1. При увеличении размера массива время его заполнения увеличивается пропорционально
2. При увеличении размера массива от 100 до 300 элементов:
   1. Время заполнения линейных функций возросло в 1,2 раза
   2. Время заполнения случайного массива возросло в 9 раз
   3. Время заполнения пилообразного массива возросло в 1,2 раза
   4. Время заполнения ступенчатого массива возросло в 7,1 раз
   5. Время заполнения синусоидального массива возросло в 8 раз