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

**Отчёт по лабораторной работе**  
  
по учебной дисциплине «Программирование»   
на тему «Алгоритмы поиска»

Исполнитель:  
обучающиеся группы М3О-207Б-23  
Ильин А.А.  
Романов Д.И.

Проверили:  
Дмитриева Е.А.

Филимонов И.А.

Москва 2024

Оглавление

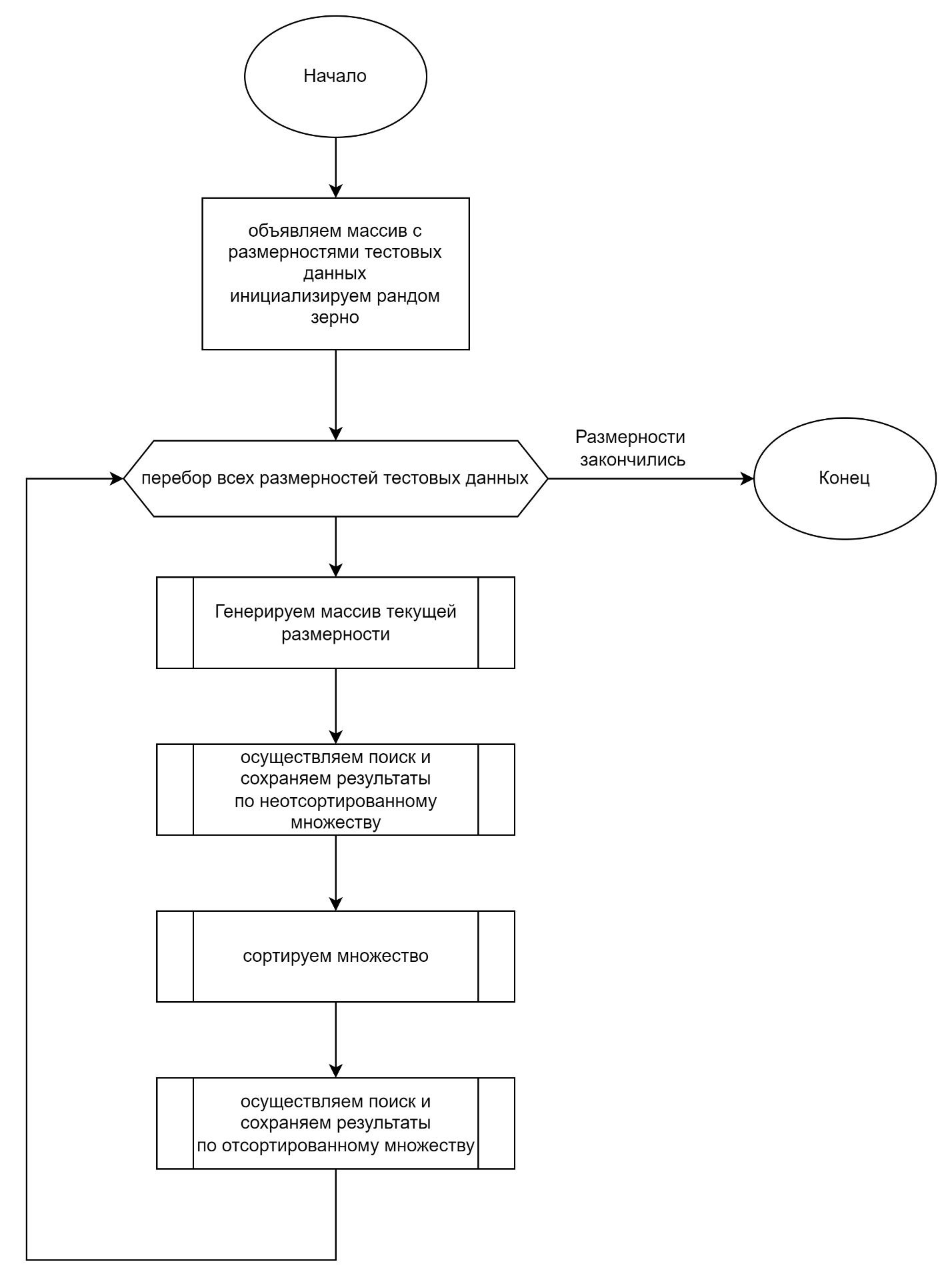
[Задание 3](#_Toc182064832)

## Задание

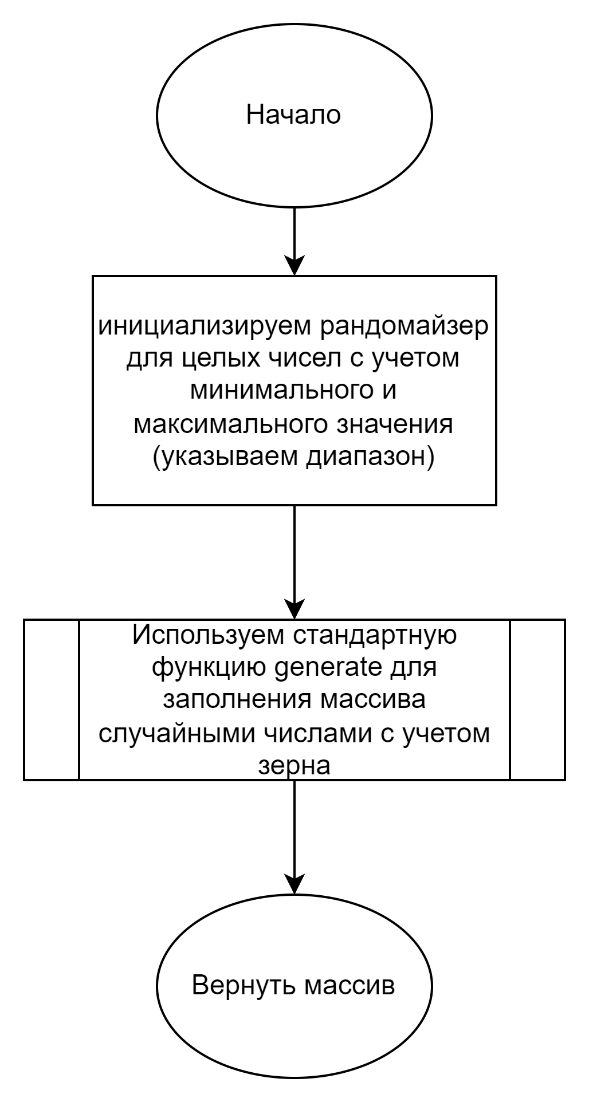
Для алгоритмов BLS и SLS в качестве входного массива использовать одну и ту же последовательность значений (функция rand( ), можно использовать соответствующую функцию из первой лабораторной работы). Для алгоритмов OAS и ВS – значения массива должны быть отсортированы по неубыванию, одна и та же последовательность 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, случай отсутствия ключа в массиве. 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.

## Блок схема

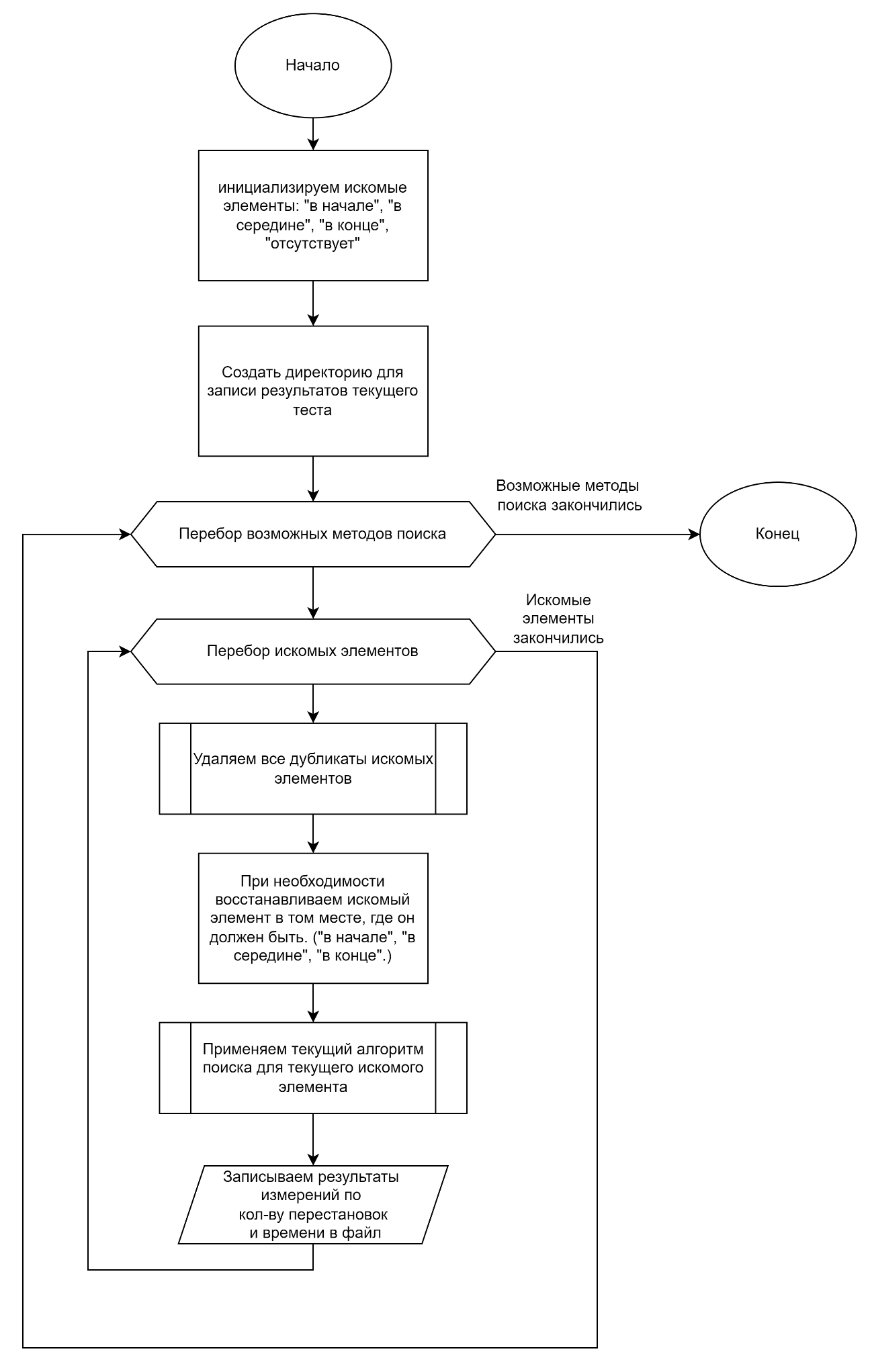
Функция Main



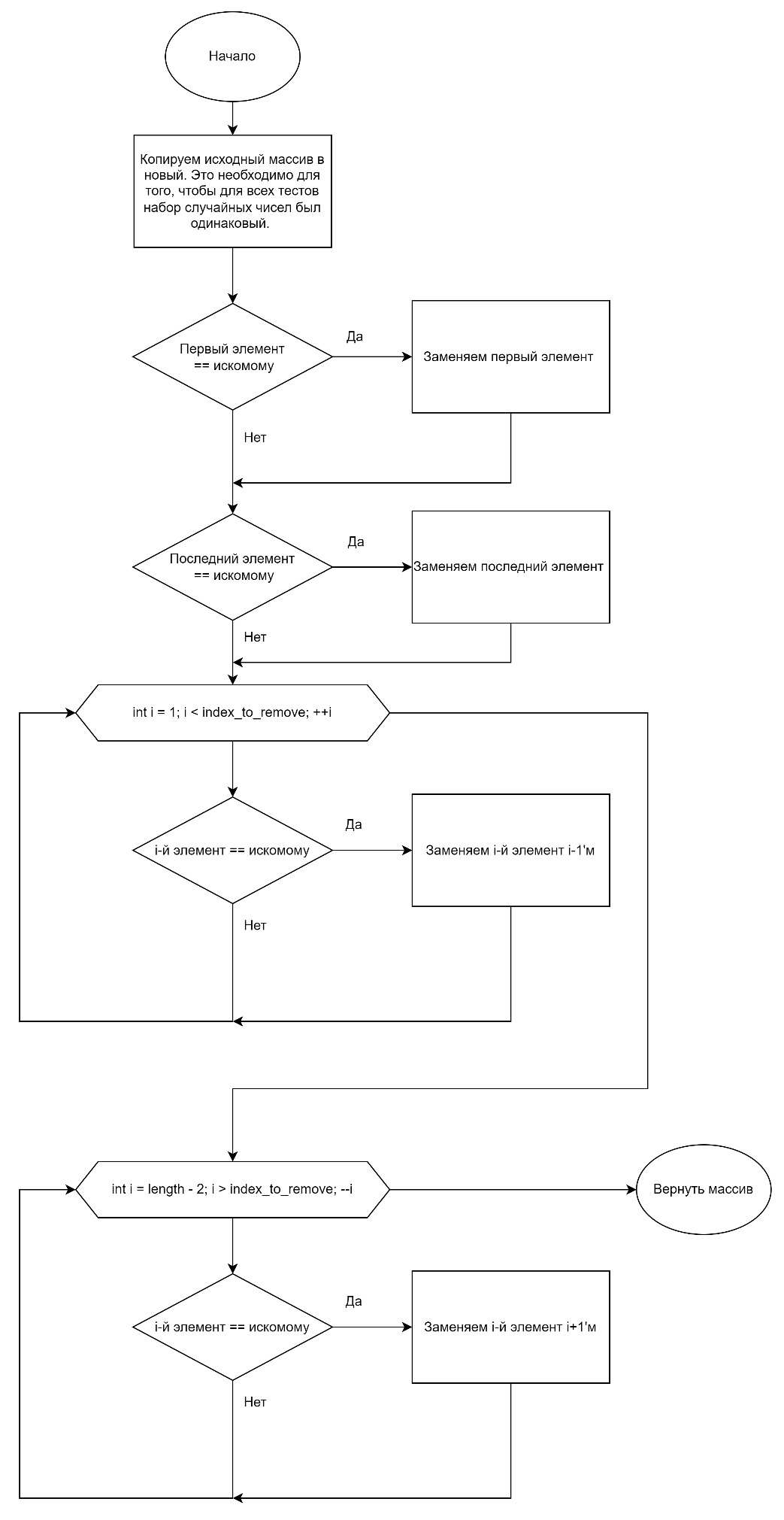
Функция генерации массива случайных чисел



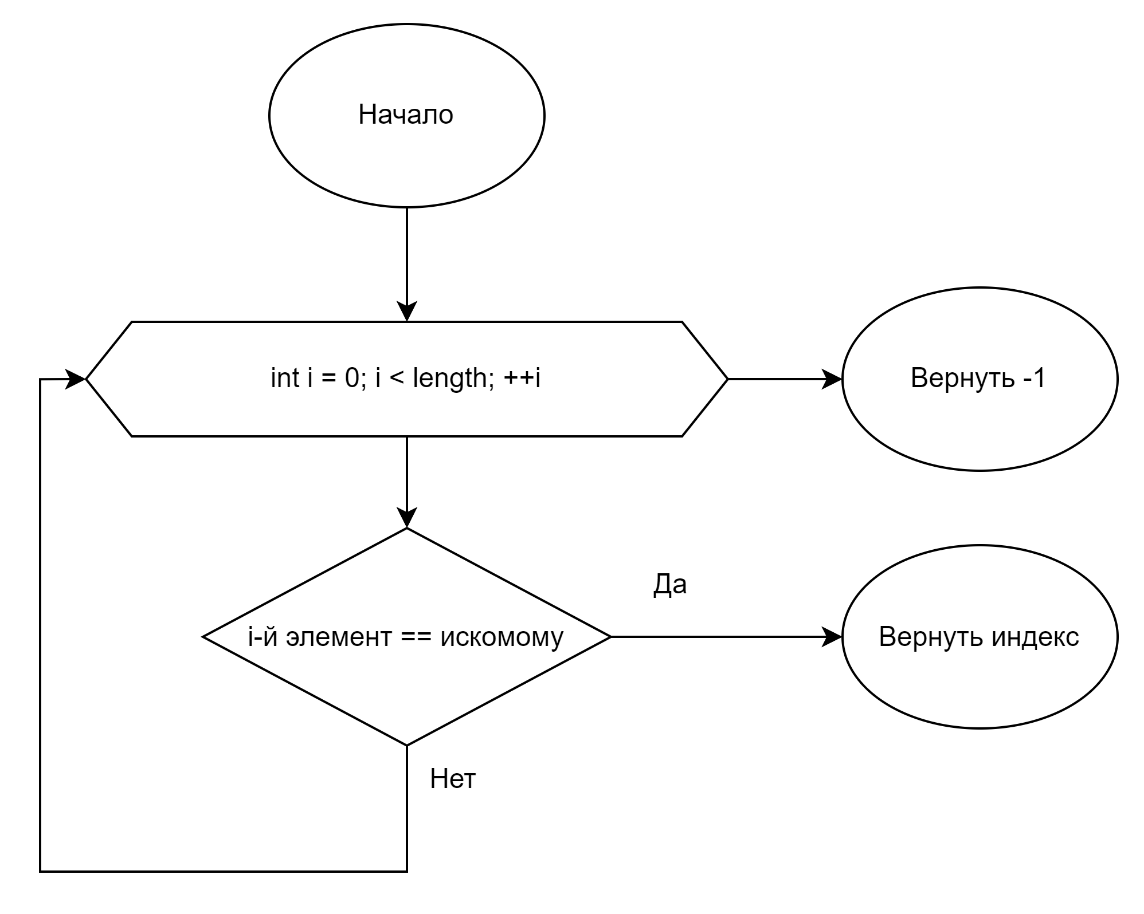
Функция для применения алгоритмов поиска и записи результатов в файл



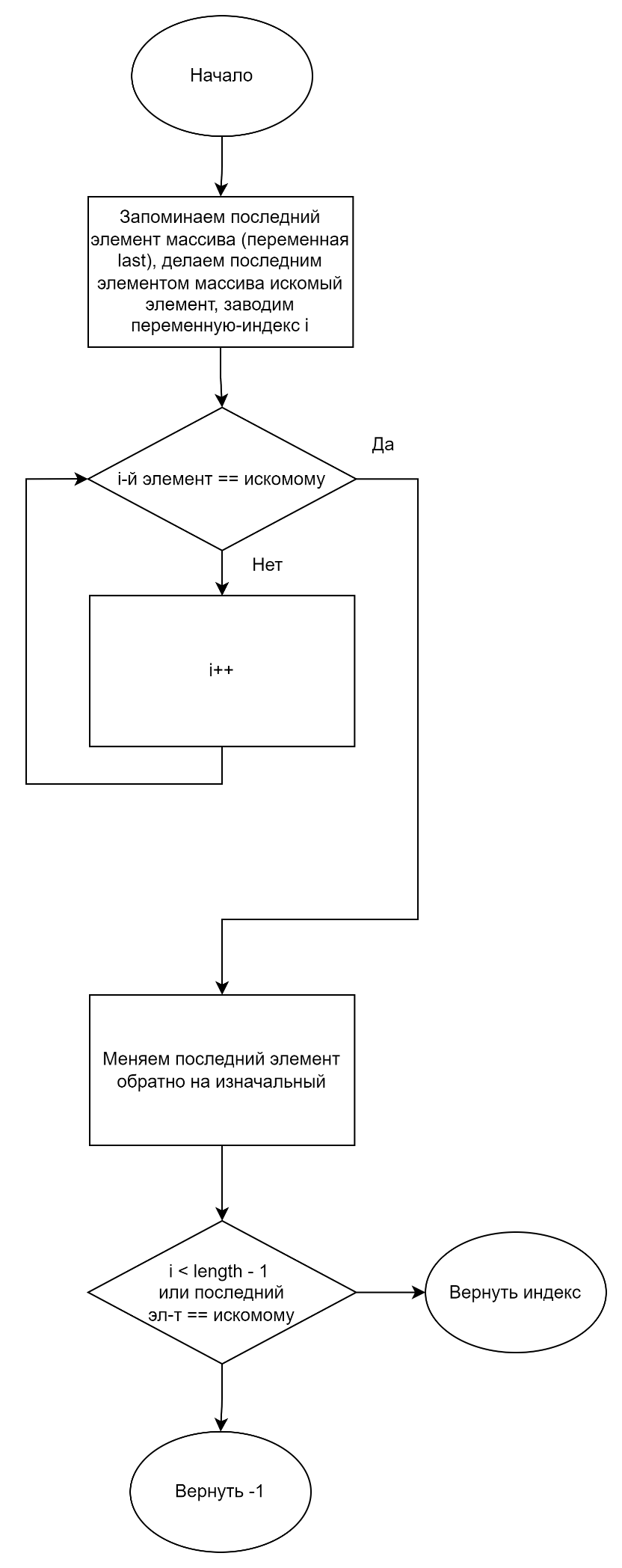
Функция для удаления дубликатов искомого элемента в массиве



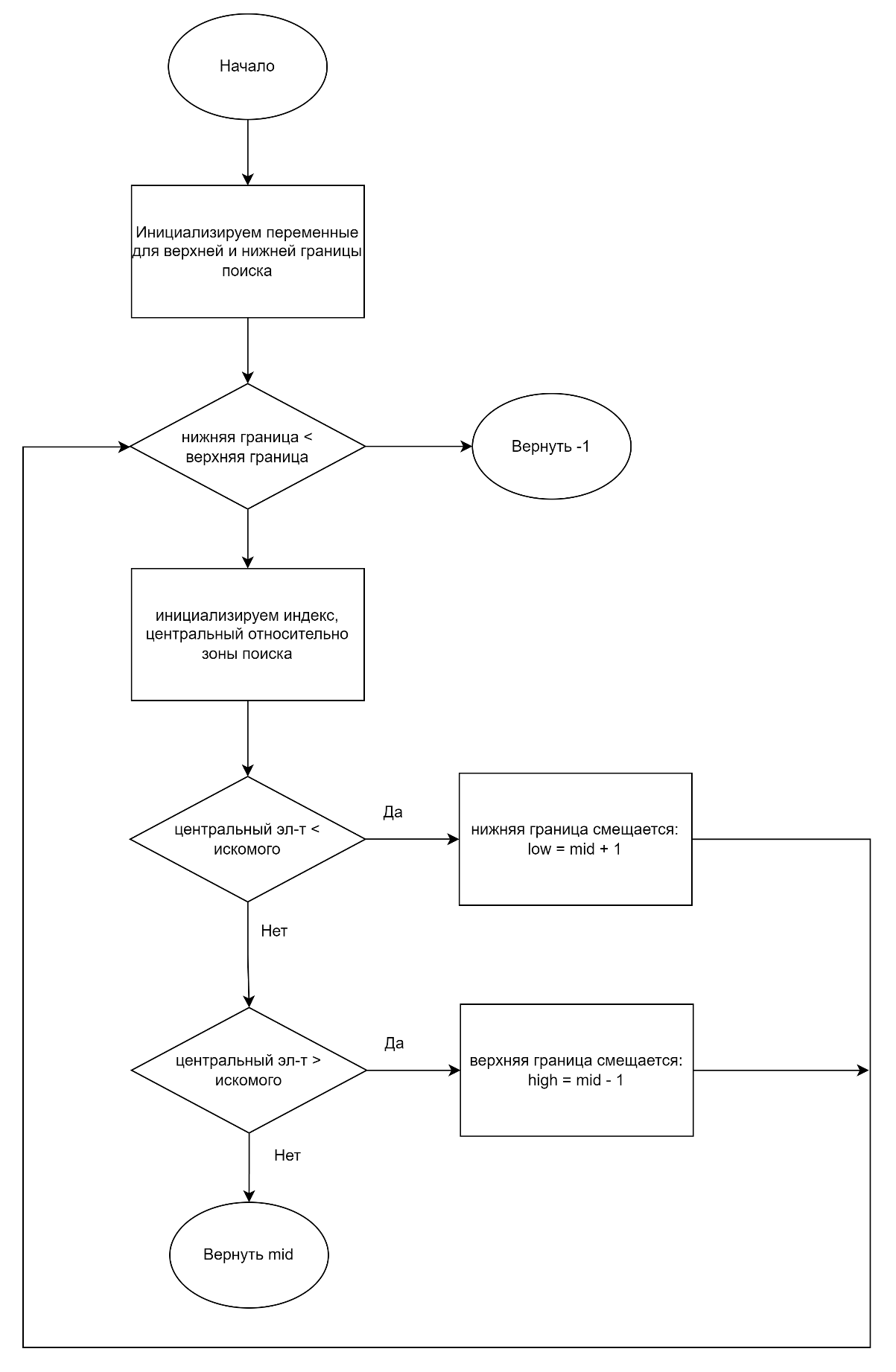
Better linear search



Sentinel linear search



Binary search



## Код программы

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* КАФЕДРА № 304 2 КУРС \*

\*--------------------------------------------------------------------------------------------------------- \*

\* Project Type : Win32 Console Application \*

\* Project Name : Sequences \*

\* File Name : main.cpp \*

\* Language : C/C++ \*

\* Programmer(s) : Романов Д.И., Ильин А.А \*

\* Modified By : \*

\* Created : 20/09/2024 \*

\* Last Revision : 05/10/2024 \*

\* Comment(s) : Формирование массивов экспериментальных данных \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <algorithm>

#include <iostream>

#include <random>

#include <fstream>

#include <filesystem>

#include "types.h"

#include "helpers.h"

#include "searches.h"

const long array\_sizes[] = {

10000,

29000,

48000,

67000,

86000,

105000,

124000,

143000,

162000,

181000,

200000,

};

void

perform\_searches\_and\_save(int const \*const array, long array\_size, const std::vector<search\_with\_name> &search\_pack,

const std::string &pack\_name) {

const std::vector<needle\_def> needles = {

{array[array\_size / 10 + 1984], array\_size / 10l + 1984l, true, "start"},

{array[array\_size \* 4 / 10 + 1984], array\_size \* 4l / 10l + 1984l, true, "middle"},

{array[array\_size \* 9 / 10 - 1984], array\_size \* 9l / 10l - 1984l, true, "end"},

{42, -1, false, "doesn't exist"},

};

auto output\_path = (std::filesystem::path("results") / pack\_name / std::to\_string(array\_size)).replace\_extension(

".txt");

std::filesystem::create\_directories(output\_path.parent\_path());

std::ofstream fout(output\_path);

fout << array\_size;

for (const auto &current\_search: search\_pack) {

fout << '|' << current\_search.name << " timing (ns)" << '|' << current\_search.name << " comparisons";

}

fout << '\n';

std::cout << "##Started " << pack\_name << " sorts with size " << array\_size << '\n';

for (const auto &needle: needles) {

fout << needle.name;

auto fixed\_array = std::make\_unique\_for\_overwrite<int[]>(array\_size);

remove\_number\_from\_array(array, fixed\_array.get(), array\_size, needle.needle, needle.correct\_index);

if (needle.should\_restore\_needle)

fixed\_array[needle.correct\_index] = needle.needle;

for (const auto &current\_search: search\_pack) {

std::cout << "#Started " << current\_search.name << " with the \"" << needle.name << "\" needle" << '\n';

search\_result sort\_result = current\_search.func(fixed\_array.get(), array\_size, needle.needle,

needle.correct\_index);

std::cout << "!Finished " << current\_search.name

<< ". It took "

<< std::chrono::duration\_cast<std::chrono::milliseconds>(sort\_result.time\_taken).count() << "ms"

<< " and " << sort\_result.comparison\_count << " comparisons" << '\n';

fout << '|' << std::chrono::duration\_cast<std::chrono::nanoseconds>(sort\_result.time\_taken).count() << '|'

<< sort\_result.comparison\_count;

}

fout << '\n';

}

std::cout << "!!Finished " << pack\_name << " sorts" << '\n';

}

int main() {

std::random\_device rd;

auto random\_number = rd();

// unsigned random\_number = 150444277;

std::mt19937 gen(random\_number);

#define MAKE\_FUNCTION\_NAME\_PAIR(func) {func, #func}

const std::vector<search\_with\_name> unsorted\_searches = {

MAKE\_FUNCTION\_NAME\_PAIR(better\_linear\_search),

MAKE\_FUNCTION\_NAME\_PAIR(sentinel\_linear\_search),

};

const std::vector<search\_with\_name> sorted\_searches = {

MAKE\_FUNCTION\_NAME\_PAIR(ordered\_array\_search),

MAKE\_FUNCTION\_NAME\_PAIR(binary\_search),

};

#undef MAKE\_FUNCTION\_NAME\_PAIR

std::cout << "Current path is " << std::filesystem::current\_path() << '\n'

<< "Random seed is: " << random\_number << '\n';

// first, we're generating an unsorted array

// then, we're testing it with each unsorted search:

// - with the needle at the start (10th el)

// - with the needle at the end (-10th el)

// - with the needle in the middle

// - with a random needle that's not in the array

// \* every time before searching we're ensuring ONLY ONE NEEDLE is in the haystack (where it should be)

// \*\* fun fact: we're using a "search" function to do that lol

// \* there are two actual search functions: one is for measuring timing, the other one is for measuring comparisons

// \* BUT they're hidden behind a generic one, then the results are combined. ordered functions just have 0 as comparisons

// then, we're sorting this array in asc order

// then, we're testing it with each sorted search:

// \* the same steps as above, but we don't really need to ensure only one needle exists (since they're basically at the same spot)

// \* with the nonexistent needle, we can just copy the previous value over it if the needle exists beforehand

// I'm picking 42 as the "random" needle each time since it's pretty close to the middle (uniform distribution)

for (auto array\_size: array\_sizes) {

int \*array = generate\_array(array\_size, -array\_size, array\_size, gen);

perform\_searches\_and\_save(array, array\_size, unsorted\_searches, "unsorted");

std::sort(array, array + array\_size);

perform\_searches\_and\_save(array, array\_size, sorted\_searches, "sorted");

delete[] array;

}

return 0;

}

struct needle\_def {

int needle; // duh

long correct\_index; // index where this needle was located

bool should\_restore\_needle; // false if the needle should not be put into the array after making only one of them

std::string name;

};

types.h

struct search\_result {

std::chrono::high\_resolution\_clock::duration time\_taken;

long comparison\_count;

};

typedef search\_result (\*int\_search\_func)(int const \*const array, int size, int needle, int correct\_index);

struct search\_with\_name {

int\_search\_func func;

std::string name;

};

Helpers.cpp

#include <random>

#include <algorithm>

#include "helpers.h"

// creates an array of given length filled with random integers from interval

int \*generate\_array(int length, int min, int max, std::mt19937 &gen) {

std::uniform\_int\_distribution<int> dist(min, max);

int \*arr = new int[length];

// use the rng to fill the array with uniformly distributed integers

std::generate(arr, arr + length, [&]() { return dist(gen); });

return arr;

}

// replaces fixed\_array with array with this number replaced with an adjacent one

int \*remove\_number\_from\_array(int const \*const array, int \*fixed\_array, int length, int number\_to\_remove,

int index\_to\_remove) {

std::copy(array, array + length, fixed\_array);

if (fixed\_array[0] == number\_to\_remove)

fixed\_array[0] = number\_to\_remove == -length ? -length + 1 : -length;

if (fixed\_array[length - 1] == number\_to\_remove)

fixed\_array[length - 1] = number\_to\_remove == length ? length - 1 : length;

if (index\_to\_remove == -1)

index\_to\_remove = length - 1;

// remove everything before

for (int i = 1; i < index\_to\_remove; ++i) {

if (fixed\_array[i] == number\_to\_remove) {

fixed\_array[i] = fixed\_array[i - 1];

}

}

// remove everything after

for (int i = length - 2; i > index\_to\_remove; --i) {

if (fixed\_array[i] == number\_to\_remove) {

fixed\_array[i] = fixed\_array[i + 1];

}

}

//remove itself

fixed\_array[index\_to\_remove] = fixed\_array[index\_to\_remove - 1];

return fixed\_array;

}

Better\_linear\_search.cpp

#include <iostream>

#include "../searches.h"

using hrc = std::chrono::high\_resolution\_clock;

int timed\_better\_linear\_search(int const \*const array, int size, int needle, search\_result &res) {

auto start = hrc::now();

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

if (array[i] == needle) {

// found

res.time\_taken = hrc::now() - start;

return i;

}

}

// not found

res.time\_taken = hrc::now() - start;

return -1;

}

int counted\_better\_linear\_search(int const \*const array, int size, int needle, search\_result &res) {

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

++res.comparison\_count; // i < size

++res.comparison\_count; // a\_i == needle

if (array[i] == needle) {

// found

return i;

}

}

// not found

return -1;

}

search\_result better\_linear\_search(int const \*const array, int size, int needle, int correct\_index) {

search\_result res{};

std::cout << "Starting timed BLS. Correct index: " << correct\_index << ". Got: ";

int timed\_index = timed\_better\_linear\_search(array, size, needle, res);

std::cout << timed\_index << ". " << (timed\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted BLS. Correct index: " << correct\_index << ". Got: ";

int counted\_index = counted\_better\_linear\_search(array, size, needle, res);

std::cout << counted\_index << ". " << (counted\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

Binary\_search.cpp

#include <iostream>

#include "../searches.h"

using hrc = std::chrono::high\_resolution\_clock;

int timed\_binary\_search(int const \*const array, int size, int needle, search\_result &res) {

auto start = hrc::now();

int low = 0, high = size - 1;

while (low <= high) {

int mid = ((high - low) >> 1) + low;

if (array[mid] > needle) {

// what we're searching is to the left of mid

high = mid - 1;

} else if (array[mid] < needle) {

// to the right of mid

low = mid + 1;

} else {

// found

res.time\_taken = hrc::now() - start;

return mid;

}

}

// not found

res.time\_taken = hrc::now() - start;

return -1;

}

search\_result binary\_search(int const \*const array, int size, int needle, int correct\_index) {

search\_result res{};

std::cout << "Starting timed BS. Correct index: " << correct\_index << ". Got: ";

int timed\_index = timed\_binary\_search(array, size, needle, res);

std::cout << timed\_index << ". " << (timed\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

ordered\_array\_search.cpp

#include <iostream>

#include "../searches.h"

using hrc = std::chrono::high\_resolution\_clock;

int timed\_ordered\_array\_search(int const \*const array, int size, int needle, search\_result &res) {

// idc the reason behind this being its own "algorithm" but honestly idc

auto start = hrc::now();

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

if (array[i] == needle) {

// found

res.time\_taken = hrc::now() - start;

return i;

}

}

// not found

res.time\_taken = hrc::now() - start;

return -1;

}

search\_result ordered\_array\_search(int const \*const array, int size, int needle, int correct\_index) {

search\_result res{};

std::cout << "Starting timed OAS. Correct index: " << correct\_index << ". Got: ";

int timed\_index = timed\_ordered\_array\_search(array, size, needle, res);

std::cout << timed\_index << ". " << (timed\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

sentinel\_linear\_search.cpp

#include <memory>

#include <iostream>

#include "../searches.h"

using hrc = std::chrono::high\_resolution\_clock;

int timed\_sentinel\_linear\_search(int const \*const array, int size, int needle, search\_result &res) {

// the code above works because we don't modify the array (it's the same each time), so we do this just in case

// it doesn't affect the computation in any way

std::unique\_ptr<int[]> temp\_array(new int[size]); // freed when out of scope

std::copy(array, array + size, temp\_array.get());

auto start = hrc::now();

int last = temp\_array[size - 1];

temp\_array[size - 1] = needle;

int i = 0;

while (temp\_array[i] != needle)

i++;

temp\_array[size - 1] = last;

if ((i < size - 1) || (temp\_array[size - 1] == needle)) {

// found

res.time\_taken = hrc::now() - start;

return i;

}

//not found

res.time\_taken = hrc::now() - start;

return -1;

}

int counted\_sentinel\_linear\_search(int const \*const array, int size, int needle, search\_result &res) {

std::unique\_ptr<int[]> temp\_array(new int[size]);

std::copy(array, array + size, temp\_array.get());

int last = temp\_array[size - 1];

temp\_array[size - 1] = needle;

int i = 0;

while (temp\_array[i] != needle) {

++res.comparison\_count;

++i;

}

temp\_array[size - 1] = last;

++res.comparison\_count; // check if found

if (i < size - 1)

// found

return i;

++res.comparison\_count; // check the last element too

if (temp\_array[size - 1] == needle)

// found

return i;

//not found

return -1;

}

search\_result sentinel\_linear\_search(int const \*const array, int size, int needle, int correct\_index) {

search\_result res{};

std::cout << "Starting timed SLS. Correct index: " << correct\_index << ". Got: ";

int timed\_index = timed\_sentinel\_linear\_search(array, size, needle, res);

std::cout << timed\_index << ". " << (timed\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted SLS. Correct index: " << correct\_index << ". Got: ";

int counted\_index = counted\_sentinel\_linear\_search(array, size, needle, res);

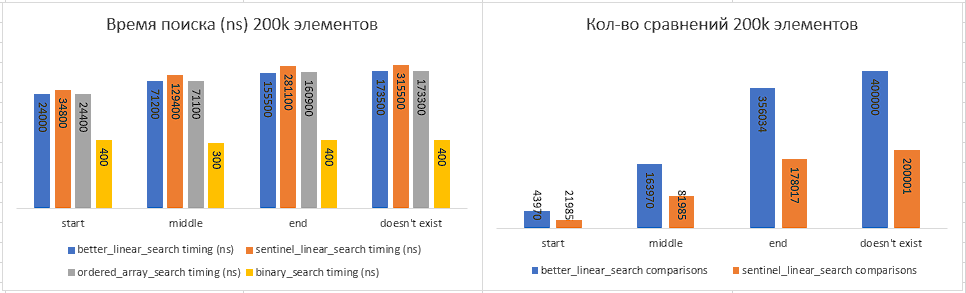
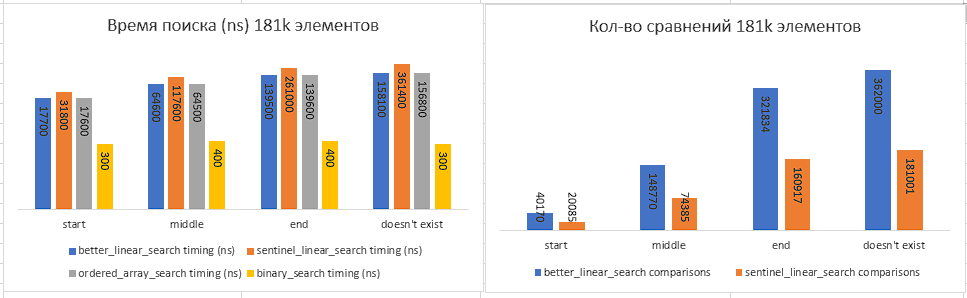
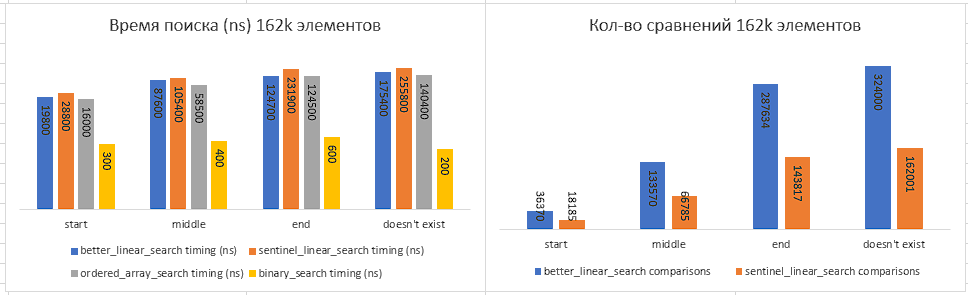
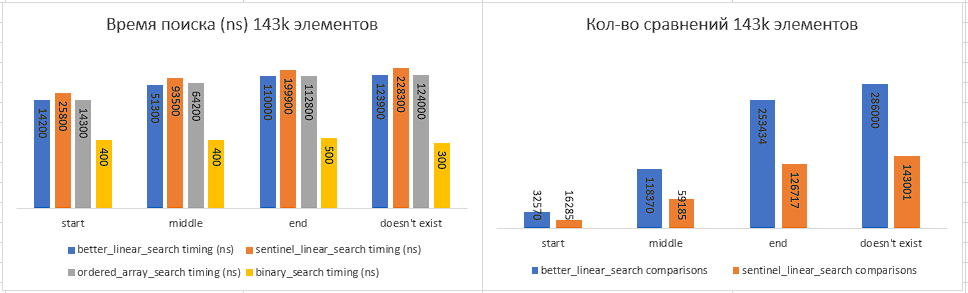
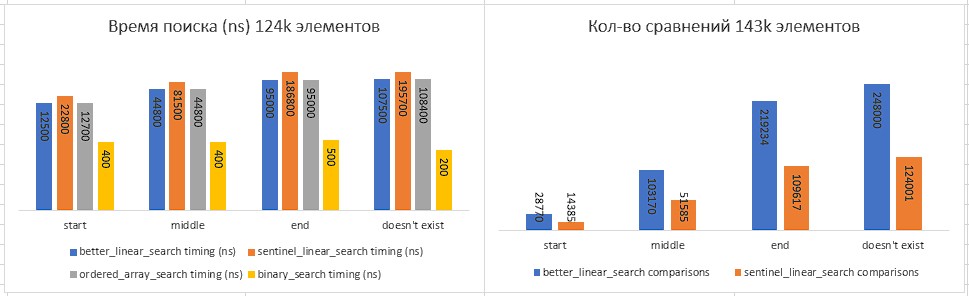
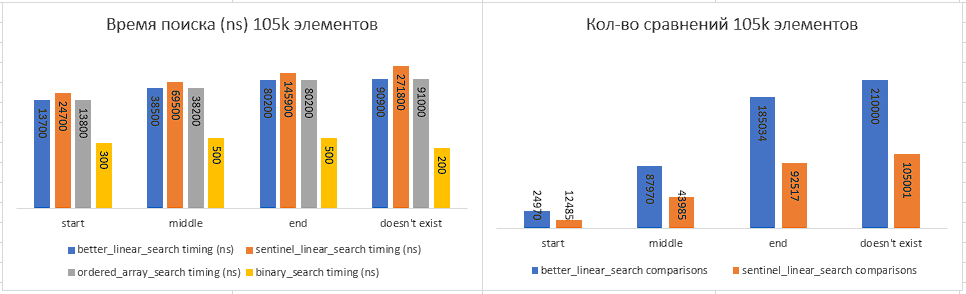
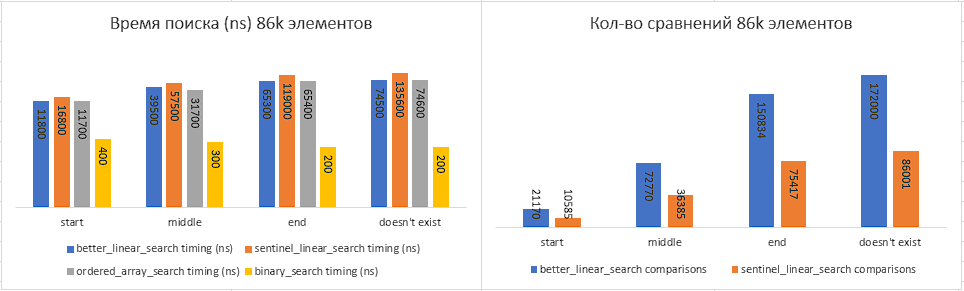
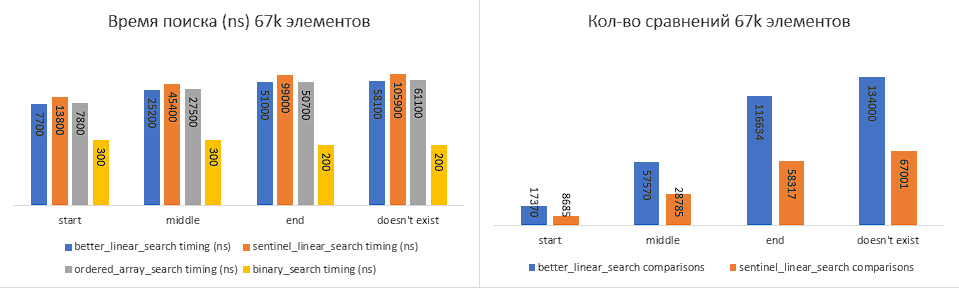
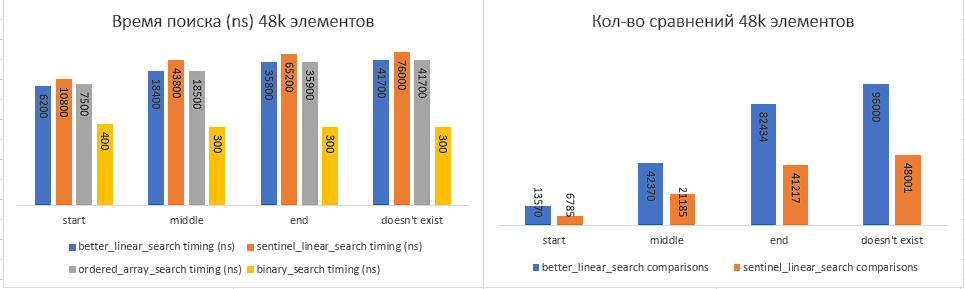
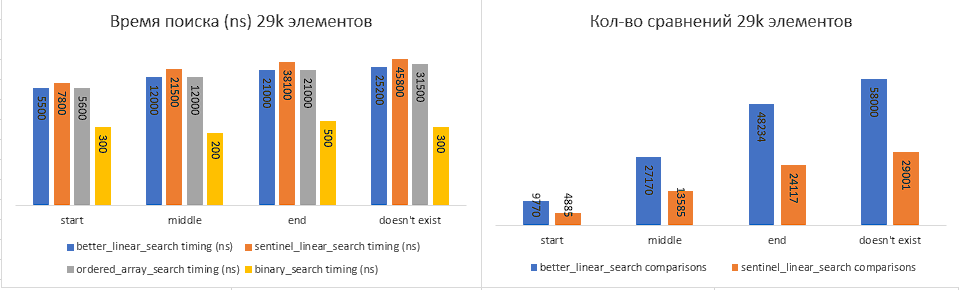
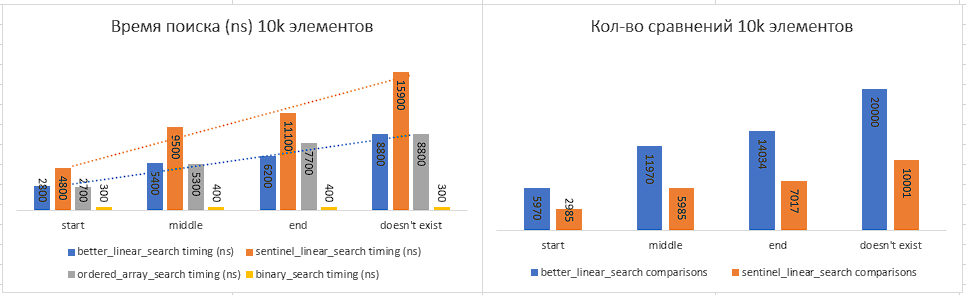
std::cout << counted\_index << ". " << (counted\_index == correct\_index ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

## Результаты



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Bls timing (ns)** | **Bls comparisons** | | **Sls timing (ns)** | | **sls comparisons** | **Oas timing (ns)** | **bs timing (ns)** |
| **200000** | | | | | | | | |
| start | 24000 | | 43970 | | 34800 | 21985 | 24400 | 400 |
| middle | 71200 | | 163970 | | 129400 | 81985 | 71100 | 300 |
| end | 155500 | | 356034 | | 281100 | 178017 | 160900 | 400 |
| doesn't exist | 173500 | | 400000 | | 315500 | 200001 | 173300 | 400 |
| **181000** | | | | | | | | |
| start | 17700 | | 40170 | | 31800 | 20085 | 17600 | 300 |
| middle | 64600 | | 148770 | | 117600 | 74385 | 64500 | 400 |
| end | 139500 | | 321834 | | 261000 | 160917 | 139600 | 400 |
| doesn't exist | 158100 | | 362000 | | 361400 | 181001 | 156800 | 300 |
| **162000** | | | | | | | | |
| start | 19800 | | 36370 | | 28800 | 18185 | 16000 | 300 |
| middle | 87600 | | 133570 | | 105400 | 66785 | 58500 | 400 |
| end | 124700 | | 287634 | | 231900 | 143817 | 124500 | 600 |
| doesn't exist | 175400 | | 324000 | | 255800 | 162001 | 140400 | 200 |
| **143000** | | | | | | | | |
| start | 14200 | | 32570 | | 25800 | 16285 | 14300 | 400 |
| middle | 51300 | | 118370 | | 93500 | 59185 | 64200 | 400 |
| end | 110000 | | 253434 | | 199900 | 126717 | 112800 | 500 |
| doesn't exist | 123900 | | 286000 | | 228300 | 143001 | 124000 | 300 |
| **124000** | | | | | | | | |
| start | 12500 | | 28770 | | 22800 | 14385 | 12700 | 400 |
| middle | 44800 | | 103170 | | 81500 | 51585 | 44800 | 400 |
| end | 95000 | | 219234 | | 186800 | 109617 | 95000 | 500 |
| doesn't exist | 107500 | | 248000 | | 195700 | 124001 | 108400 | 200 |
| **105000** | | | | | | | | |
| start | 13700 | | 24970 | | 24700 | 12485 | 13800 | 300 |
| middle | 38500 | | 87970 | | 69500 | 43985 | 38200 | 500 |
| end | 80200 | | 185034 | | 145900 | 92517 | 80200 | 500 |
| doesn't exist | 90900 | | 210000 | | 271800 | 105001 | 91000 | 200 |
| **86000** | | | | | | | | |
| start | 11800 | | 21170 | | 16800 | 10585 | 11700 | 400 |
| middle | 39500 | | 72770 | | 57500 | 36385 | 31700 | 300 |
| end | 65300 | | 150834 | | 119000 | 75417 | 65400 | 200 |
| doesn't exist | 74500 | | 172000 | | 135600 | 86001 | 74600 | 200 |
| **67000** | | | | | | | | |
| start | 7700 | | 17370 | | 13800 | 8685 | 7800 | 300 |
| middle | 25200 | | 57570 | | 45400 | 28785 | 27500 | 300 |
| end | 51000 | | 116634 | | 99000 | 58317 | 50700 | 200 |
| doesn't exist | 58100 | | 134000 | | 105900 | 67001 | 61100 | 200 |
| **48000** | | | | | | | | |
| start | 6200 | | 13570 | | 10800 | 6785 | 7500 | 400 |
| middle | 18400 | | 42370 | | 43800 | 21185 | 18500 | 300 |
| end | 35800 | | 82434 | | 65200 | 41217 | 35900 | 300 |
| doesn't exist | 41700 | | 96000 | | 76000 | 48001 | 41700 | 300 |
| **29000** | | | | | | | | |
| start | 5500 | | 9770 | | 7800 | 4885 | 5600 | 300 |
| middle | 12000 | | 27170 | | 21500 | 13585 | 12000 | 200 |
| end | 21000 | | 48234 | | 38100 | 24117 | 21000 | 500 |
| doesn't exist | 25200 | | 58000 | | 45800 | 29001 | 31500 | 300 |
| **10000** | | | | | | | | |
| start | 2800 | | 5970 | | 4800 | 2985 | 2700 | 300 |
| middle | 5400 | | 11970 | | 9500 | 5985 | 5300 | 400 |
| end | 6200 | | 14034 | | 11100 | 7017 | 7700 | 400 |
| doesn't exist | 8800 | | 20000 | | 15900 | 10001 | 8800 | 300 |

## Вывод

В данной работе мы на практике реализовали и сравнили несколько алгоритмов поиска элемента в массиве. Для поиска элемента в отсортированном массиве самым оптимальным алгоритмом из тестируемых является Binary Search, в неотсортированном – better linear search, ведь несмотря на большее количество сравнений, по сравнению с sentinel search, время поиска меньше в 100% тестовых случаях.