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 сортировки»

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

Проверили:  
Дмитриева Е.А.

Ивашенцев И. В.

Москва 2024

Оглавление

[Задание 3](#_Toc182143329)

[Блок схема 4](#_Toc182143330)

[Код программы 11](#_Toc182143331)

[Результаты 31](#_Toc182143332)

[Вывод 36](#_Toc182143333)

## Задание

Для массива из ***n*** элементов выполнить сортировку по возрастанию с помощью двух указанных методов (по вариантам) для:

1. заданной произвольным образом последовательности чисел (массив один и тот же для разных сортировок);
2. уже отсортированных последовательностей в возрастающем и убывающем порядке (лучший и худший случаи для выполнения сортировки);
3. для быстрой сортировки выполнить три варианта исследования:
   * опорный элемент – последний элемент в массиве,
   * опорный элемент – медиана,
   * по схеме Хоара.

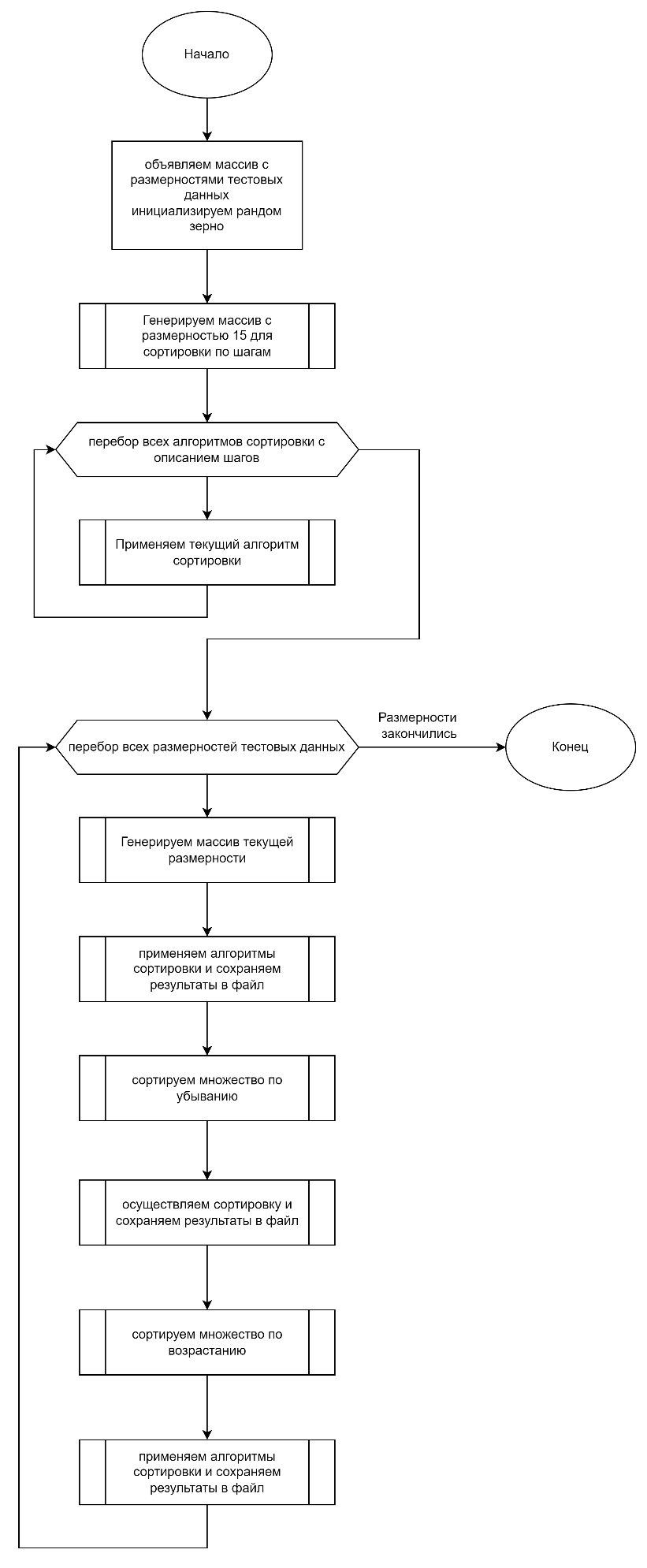
Этапы выполнения ЛР:

1. Промежуточные результаты сортировки представить по каждой итерации для массива размерностью ***n*** = 15 (необходимо выводить на печать весь массив на каждом промежуточном этапе сортировки). Сравнить число **необходимых** **сравнений и число пересылок**.
2. Выполнить сортировку массивов размерности ***n*** = 500, 1000, 10000, 50000. Сравнить **время выполнения алгоритмов, число необходимых сравнений и число пересылок**.

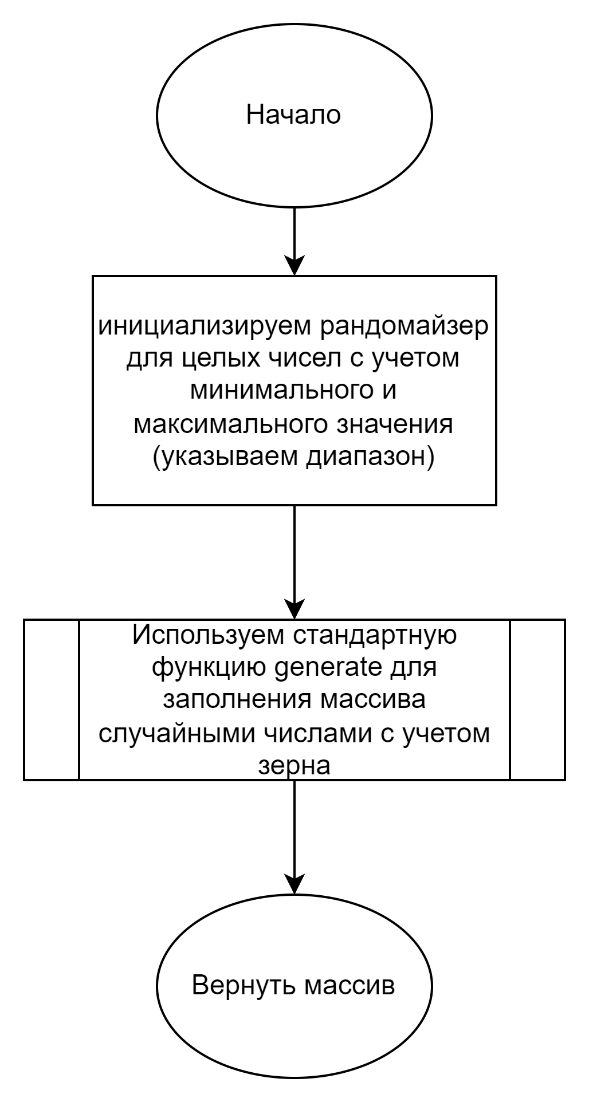
|  |  |
| --- | --- |
| № вар. | метод |
| 1 | выбором, быстрая |

## Блок схема

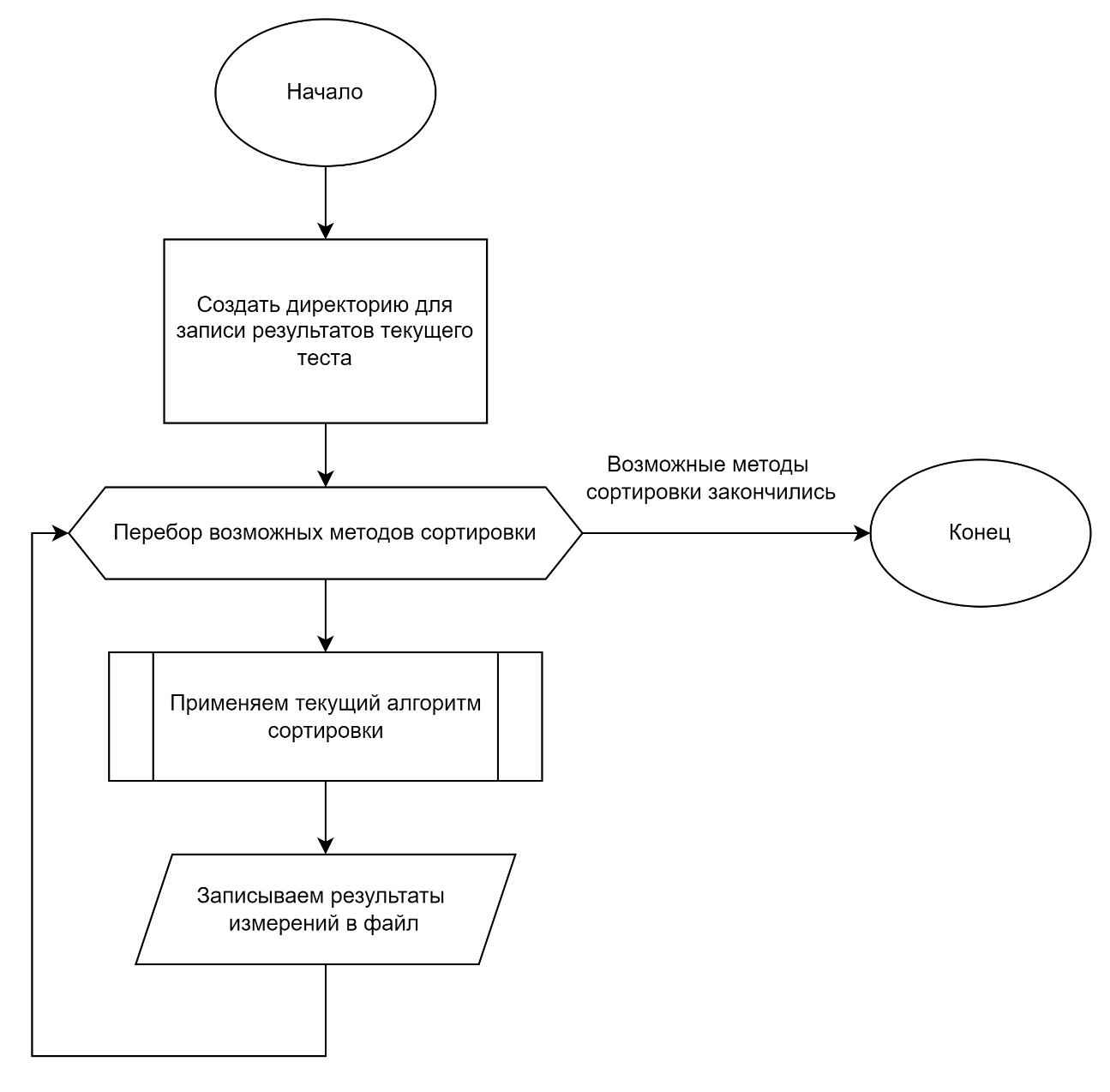
Функция Main



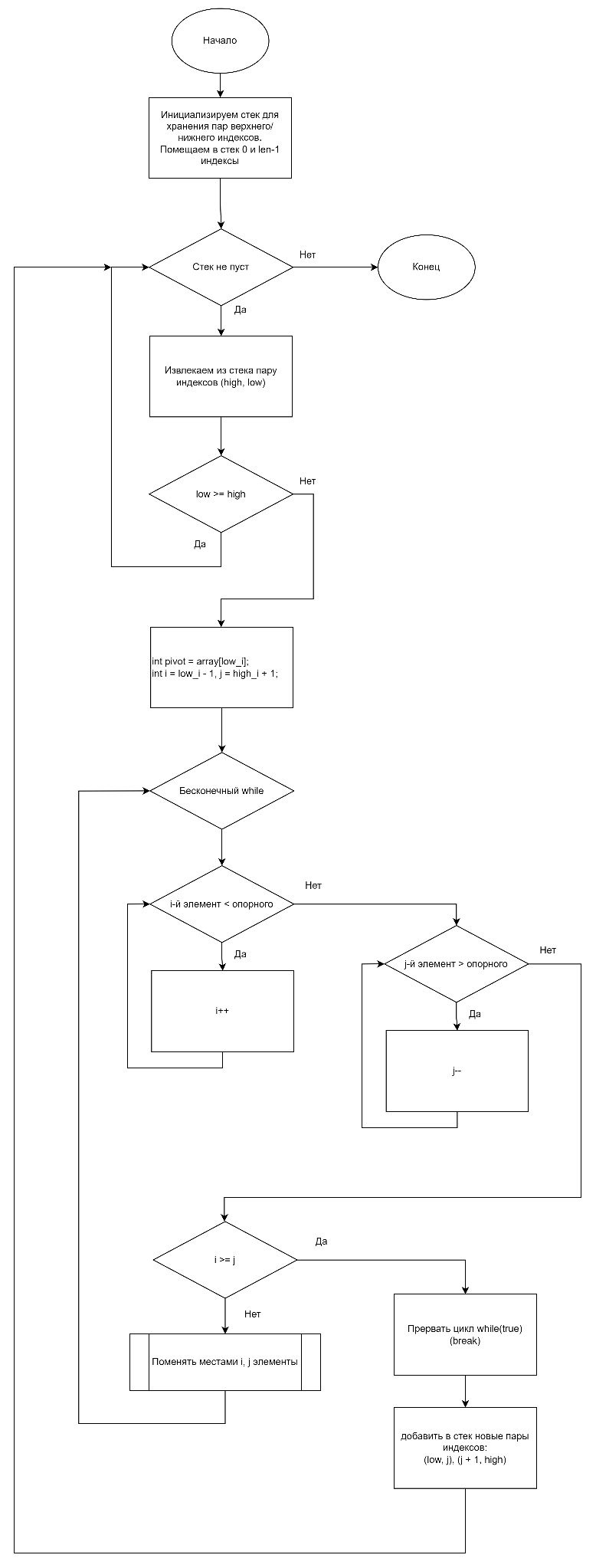
Функция генерации массива случайных чисел



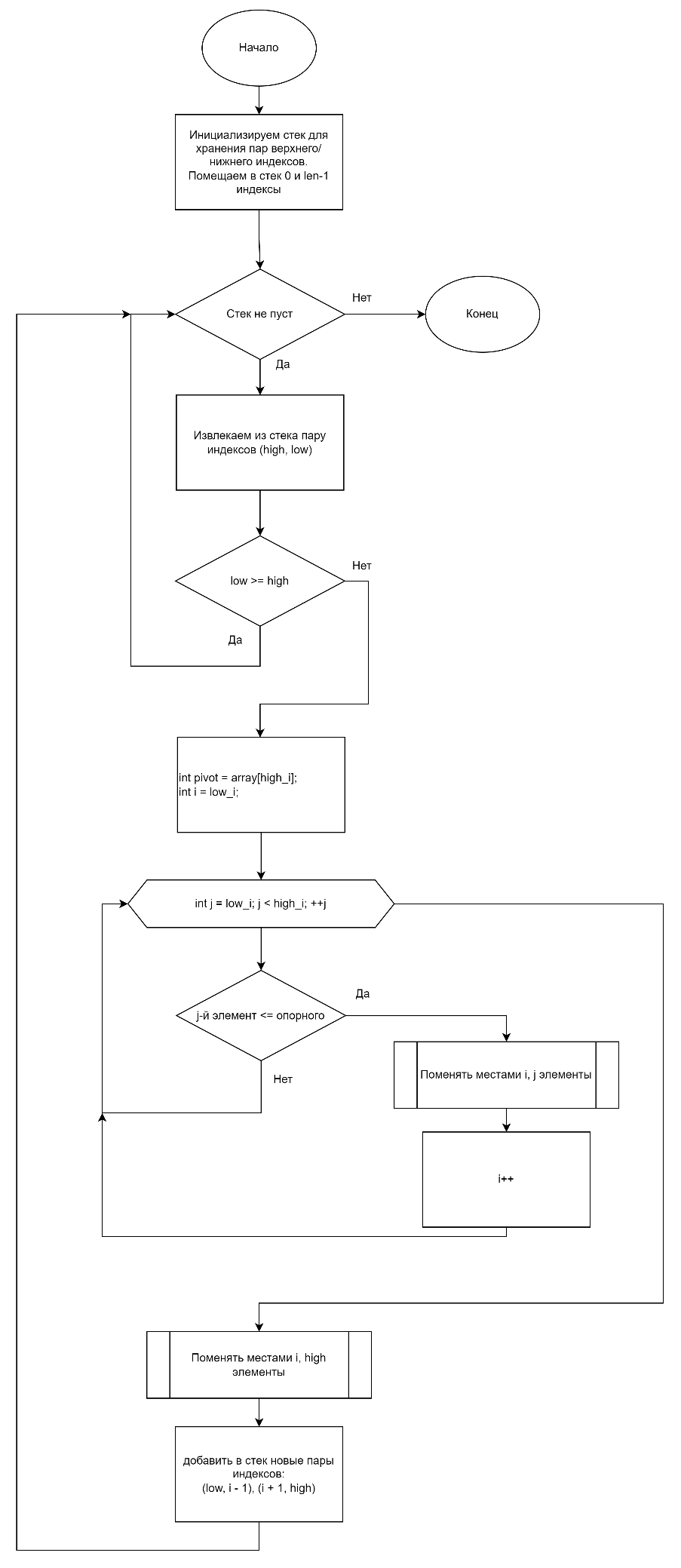
Функция для применения алгоритмов сортировки и записи результатов в файл

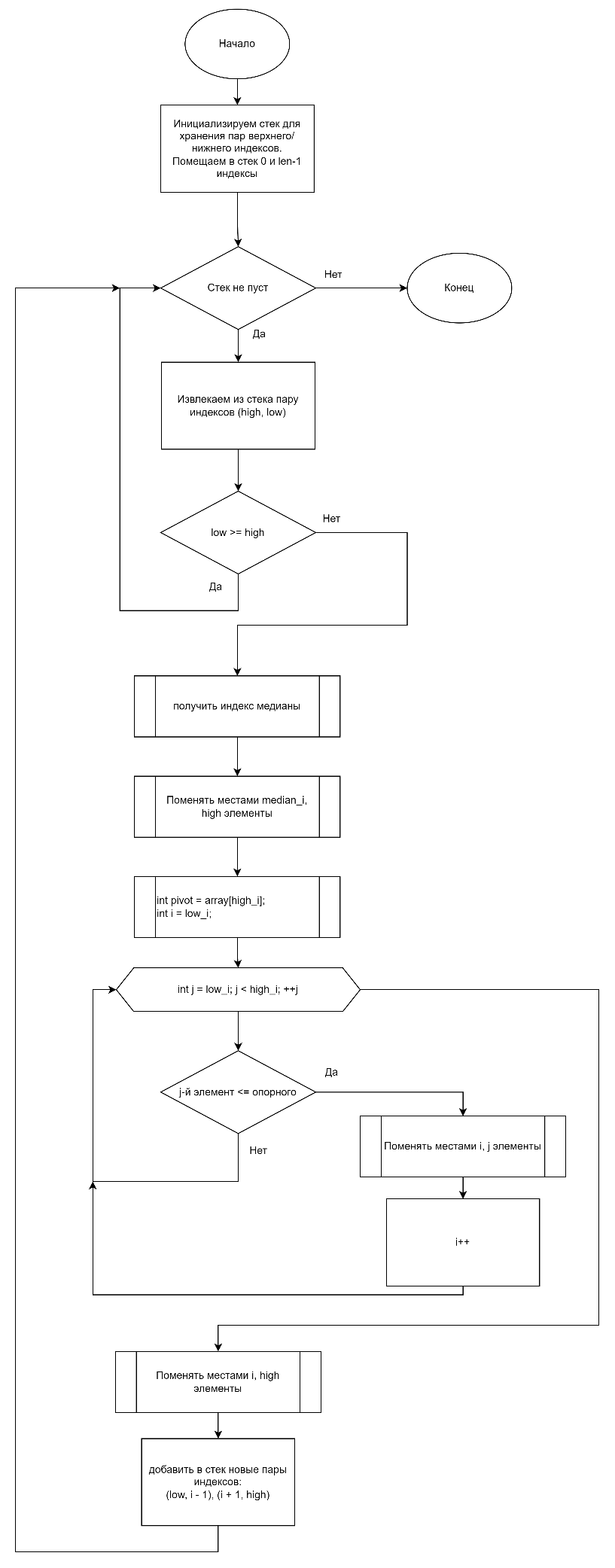


Быстрая сортировка Хоара

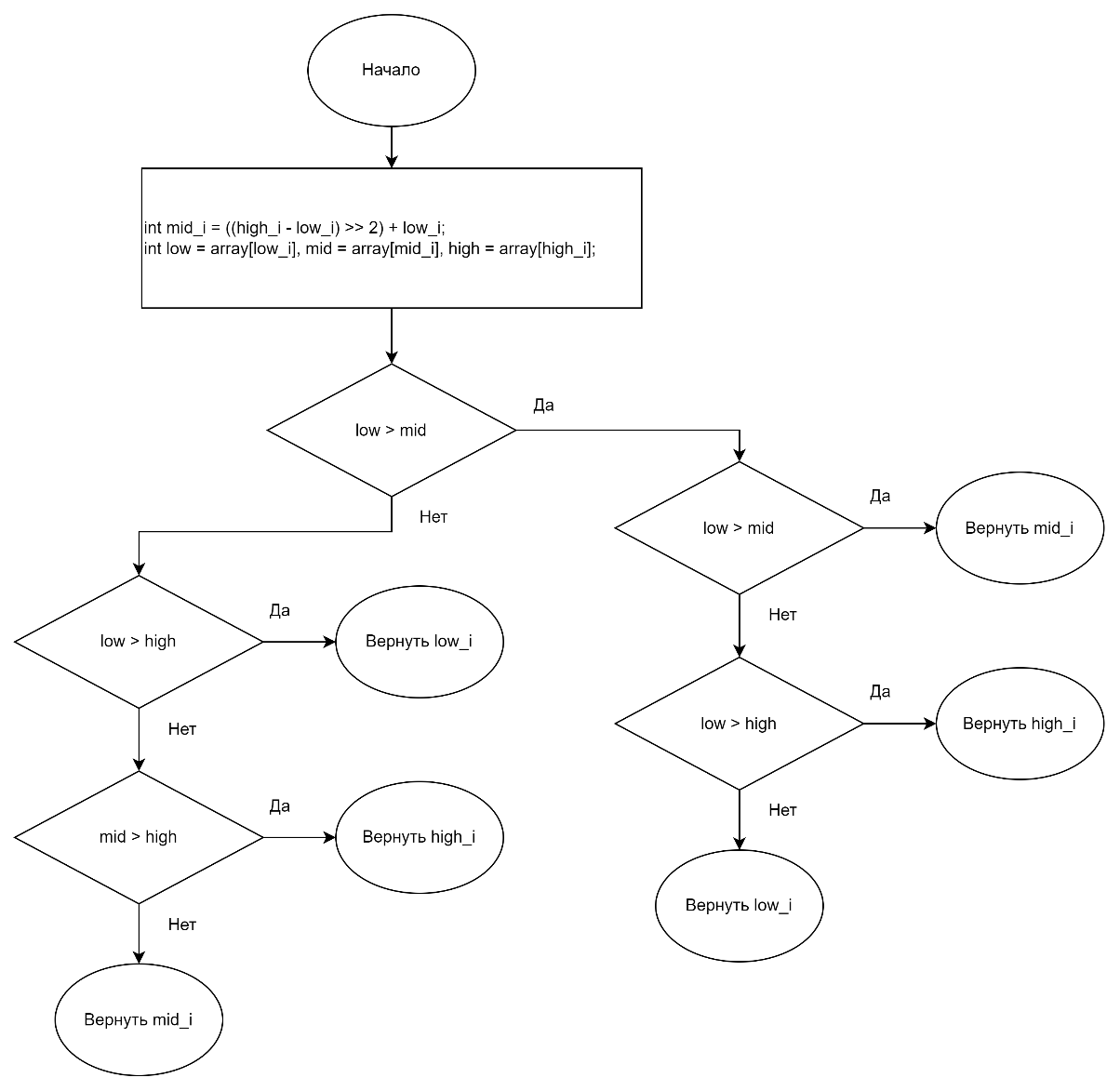


Быстрая сортировка, опорный элемент - последний

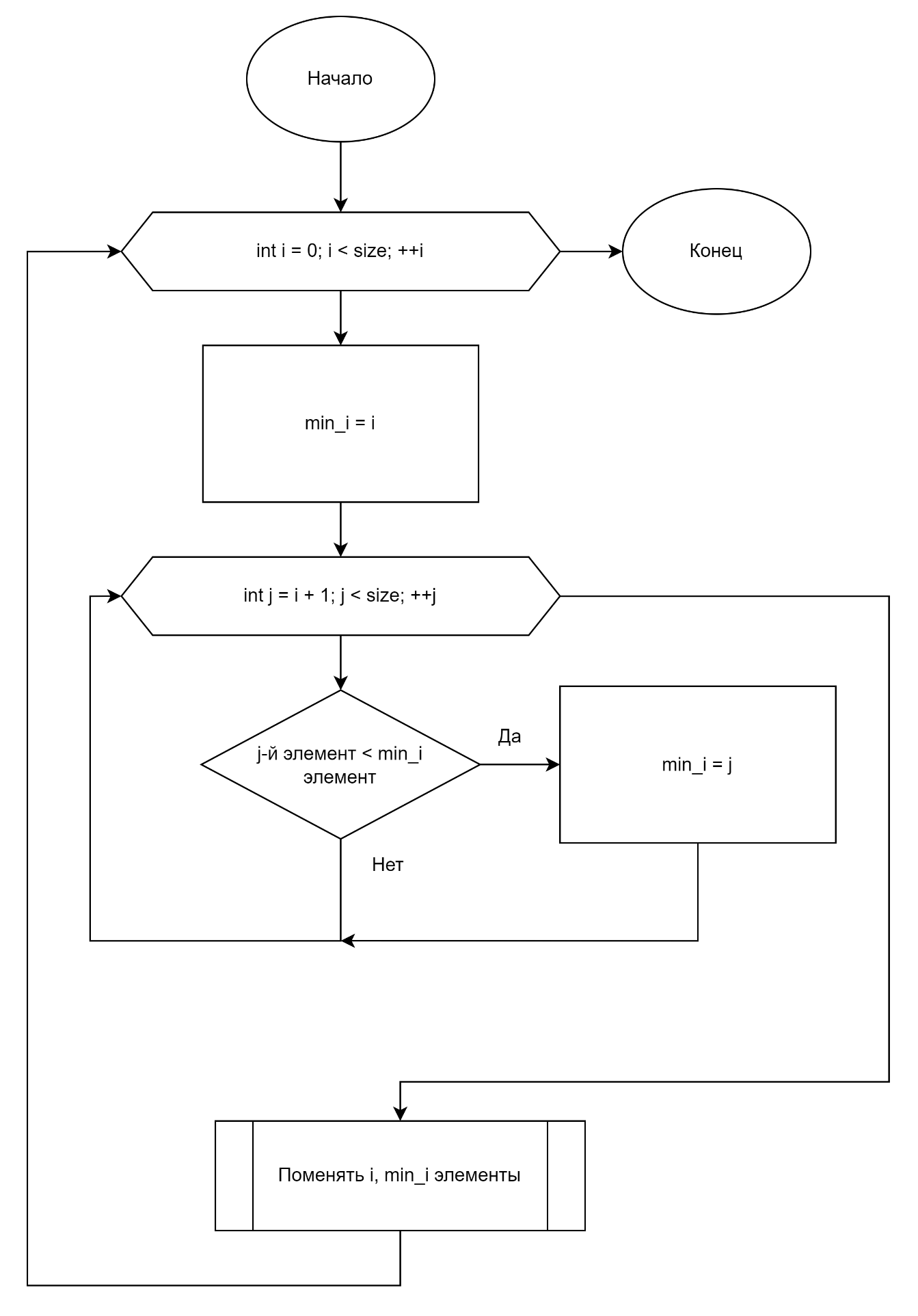


Быстрая сортировка, опорный элемент – медиана 3

Получение индекса медианы



Сортировка выбором



## 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 "sorts.h"

const long array\_sizes[] = {

5,

500,

1000,

10000,

50000,

};

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

const std::string &pack\_name) {

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);

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

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

<< current\_search.name << " swaps";

}

fout << '\n';

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

fout << "results";

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

std::copy(array, array + array\_size, sorted\_array.get());

std::sort(sorted\_array.get(), sorted\_array.get() + array\_size);

for (const auto &current\_sort: sort\_pack) {

std::cout << "#Started " << current\_sort.name << '\n';

sort\_result sort\_result = current\_sort.func(array, array\_size, sorted\_array.get());

std::cout << "!Finished " << pack\_name << ' ' << current\_sort.name

<< ". It took "

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

<< ", " << sort\_result.comparison\_count << " comparisons"

<< " and " << sort\_result.swap\_count << " swaps" << '\n';

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

<< sort\_result.comparison\_count << '|' << sort\_result.swap\_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<sort\_with\_steps\_with\_name> sorts\_with\_steps = {

MAKE\_FUNCTION\_NAME\_PAIR(selection\_sort\_with\_steps),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_last\_with\_steps),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_median\_of\_3\_with\_steps),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_hoare\_with\_steps),

};

const std::vector<sort\_with\_name> sorts = {

MAKE\_FUNCTION\_NAME\_PAIR(selection\_sort),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_last),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_median\_of\_3),

MAKE\_FUNCTION\_NAME\_PAIR(quick\_sort\_hoare),

};

#undef MAKE\_FUNCTION\_NAME\_PAIR

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

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

{

std::cout << "##Started sorts with steps\n";

int array\_size = 15;

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

for (const auto &sort\_with\_steps: sorts\_with\_steps) {

std::cout << "#Started " << sort\_with\_steps.name << '\n';

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

std::copy(array, array + array\_size, array\_to\_sort.get());

sort\_with\_steps.func(array\_to\_sort.get(), array\_size);

std::cout << "!Finished " << sort\_with\_steps.name << '\n';

}

std::cout << "!!Finished sorts with steps\n";

delete[] array;

}

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

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

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

std::sort(array, array + array\_size, std::less{});

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

std::sort(array, array + array\_size, std::greater{});

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

delete[] array;

}

return 0;

}

Helpers.cpp

#include "helpers.h"

#include <random>

#include <algorithm>

#include <iostream>

// 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;

}

bool check\_arrays\_equal(const int \*const arr1, const int \*const arr2, int size) {

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

if (arr1[i] != arr2[i])

return false;

return true;

}

void print\_array(const int \*arr, int size, const std::initializer\_list<int>& underlined\_indices) {

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

std::cout << arr[i] << '\t';

}

std::cout << '\n';

std::string underlined\_string(size\*4, ' ');

for (int underlined\_i : underlined\_indices) {

underlined\_string[underlined\_i \* 4] = '^';

underlined\_string[underlined\_i \* 4 + 1] = '~';

}

std::cout << underlined\_string << '\n';

}

Types.h

#ifndef LW\_TYPES\_H

#define LW\_TYPES\_H

#include <chrono>

#include <string>

struct needle\_def {

int needle; // duh

int 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;

};

struct sort\_result {

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

long long comparison\_count;

long long swap\_count;

};

typedef sort\_result (\*int\_sort\_test)(int const \*const array, int size, int const \*const sorted\_array);

struct sort\_with\_name {

int\_sort\_test func;

std::string name;

};

typedef void (\*int\_sort\_with\_steps)(int \*const array, int size);

struct sort\_with\_steps\_with\_name {

int\_sort\_with\_steps func;

std::string name;

};

#endif //LW\_TYPES\_H

Quick\_sort\_hoare.cpp

#include <chrono>

#include <iostream>

#include <memory>

#include <stack>

#include "../sorts.h"

#include "../helpers.h"

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

void timed\_quick\_sort\_hoare(int \*const array, int size, sort\_result &res) {

auto start = hrc::now();

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

int pivot = array[low\_i];

int i = low\_i - 1, j = high\_i + 1;

while (true) {

do {

i++;

} while (array[i] < pivot);

do {

j--;

} while (array[j] > pivot);

if (i >= j)

break;

std::swap(array[i], array[j]);

}

sort\_stack.emplace(low\_i, j);

sort\_stack.emplace(j + 1, high\_i);

}

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

}

void counted\_quick\_sort\_hoare(int \*const array, int size, sort\_result &res) {

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

++res.comparison\_count;

if (low\_i >= high\_i)

continue;

int pivot = array[low\_i];

int i = low\_i - 1, j = high\_i + 1;

while (true) {

do {

i++;

++res.comparison\_count;

} while (array[i] < pivot);

do {

j--;

++res.comparison\_count;

} while (array[j] > pivot);

++res.comparison\_count;

if (i >= j)

break;

++res.swap\_count;

std::swap(array[i], array[j]);

}

sort\_stack.emplace(low\_i, j);

sort\_stack.emplace(j + 1, high\_i);

}

}

sort\_result quick\_sort\_hoare(int const \*const array, int size, int const \*const sorted\_array) {

auto temp\_array = std::make\_unique\_for\_overwrite<int[]>(size);

sort\_result res{};

std::cout << "Starting timed QUICK SORT HOARE. Got: " << std::flush;

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

timed\_quick\_sort\_hoare(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted QUICK SORT HOARE. Got: " << std::flush;

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

counted\_quick\_sort\_hoare(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

void quick\_sort\_hoare\_with\_steps(int \*const array, int size) {

std::cout << "Исходный массив:\n";

print\_array(array, size);

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

std::cout << "Сортируем подмассив:\n";

print\_array(array, size, {low\_i, high\_i});

std::cout << "Опорным элементом берём первый элемент подмассива.\n";

int pivot = array[low\_i];

int i = low\_i - 1, j = high\_i + 1;

while (true) {

do {

i++;

} while (array[i] < pivot);

do {

j--;

} while (array[j] > pivot);

if (i >= j) {

std::cout << "Индексы сошлись\n";

break;

}

std::cout << "Переставим элемент слева, больший опорного, и справа, меньший опорного:\n";

std::swap(array[i], array[j]);

print\_array(array, size, {i, j});

}

sort\_stack.emplace(low\_i, j);

sort\_stack.emplace(j + 1, high\_i);

}

}

Quick\_search\_last.cpp

#include <chrono>

#include <iostream>

#include <memory>

#include <stack>

#include "../sorts.h"

#include "../helpers.h"

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

void timed\_quick\_sort\_last(int \*const array, int size, sort\_result &res) {

auto start = hrc::now();

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; j < high\_i; ++j) {

if (array[j] <= pivot) {

std::swap(array[i], array[j]);

++i;

}

}

std::swap(array[i], array[high\_i]);

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

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

}

void counted\_quick\_sort\_last(int \*const array, int size, sort\_result &res) {

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

++res.comparison\_count;

if (low\_i >= high\_i)

continue;

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; ++res.comparison\_count, j < high\_i; ++j) {

++res.comparison\_count;

if (array[j] <= pivot) {

++res.swap\_count;

std::swap(array[i], array[j]);

++i;

}

}

++res.swap\_count;

std::swap(array[i], array[high\_i]);

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

}

sort\_result quick\_sort\_last(int const \*const array, int size, int const \*const sorted\_array) {

auto temp\_array = std::make\_unique\_for\_overwrite<int[]>(size);

sort\_result res{};

std::cout << "Starting timed QUICK SORT LAST. Got: " << std::flush;

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

timed\_quick\_sort\_last(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted QUICK SORT LAST. Got: " << std::flush;

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

counted\_quick\_sort\_last(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

void quick\_sort\_last\_with\_steps(int \*array, int size) {

std::cout << "Исходный массив:\n";

print\_array(array, size);

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

std::cout << "Сортируем подмассив:\n";

print\_array(array, size, {low\_i, high\_i});

std::cout << "Опорным элементом берём последний элемент подмассива\n";

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; j < high\_i; ++j) {

std::cout << j << "-й элемент ";

if (array[j] <= pivot) {

std::cout << "меньше опорного. Переставим его с первым элементом больше опорного\n";

std::swap(array[i], array[j]);

print\_array(array, size, {i, j});

++i;

} else {

std::cout << "больше опорного. Оставим его не месте.\n";

}

}

std::cout << "Поставим опорный элемент между большими и меньшими, переставив его с первым большим\n";

std::swap(array[i], array[high\_i]);

print\_array(array, size, {i, high\_i});

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

}

quick\_sort\_median\_of\_3.css

#include <chrono>

#include <iostream>

#include <memory>

#include <stack>

#include "../sorts.h"

#include "../helpers.h"

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

// returns index of median-of-3 between indices of array

inline int median\_of\_3(int const \*const array, int low\_i, int high\_i) {

int mid\_i = ((high\_i - low\_i) >> 2) + low\_i;

int low = array[low\_i], mid = array[mid\_i], high = array[high\_i];

if (low > mid) {

if (mid > high)

return mid\_i;

else if (low > high) // low > mid < high

return high\_i;

else

return low\_i;

} else { // mid > low

if (low > high)

return low\_i;

else if (mid > high) // mid > low < high

return high\_i;

else

return mid\_i;

}

}

void timed\_quick\_sort\_median\_of\_3(int \*const array, int size, sort\_result &res) {

auto start = hrc::now();

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

int median\_i = median\_of\_3(array, low\_i, high\_i);

std::swap(array[median\_i], array[high\_i]);

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; j < high\_i; ++j) {

if (array[j] <= pivot) {

std::swap(array[i], array[j]);

++i;

}

}

std::swap(array[i], array[high\_i]);

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

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

}

inline int median\_of\_3\_counted(int const \*const array, int low\_i, int high\_i, sort\_result &res) {

int mid\_i = ((high\_i - low\_i) >> 2) + low\_i;

int low = array[low\_i], mid = array[mid\_i], high = array[high\_i];

++res.comparison\_count;

if (low > mid) {

++res.comparison\_count;

if (mid > high)

return mid\_i;

else {

++res.comparison\_count;

if (low > high) // low > mid < high

return high\_i;

else

return low\_i;

}

} else { // mid > low

++res.comparison\_count;

if (low > high)

return low\_i;

else {

++res.comparison\_count;

if (mid > high) // mid > low < high

return high\_i;

else

return mid\_i;

}

}

}

void counted\_quick\_sort\_median\_of\_3(int \*const array, int size, sort\_result &res) {

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

++res.comparison\_count;

if (low\_i >= high\_i)

continue;

int median\_i = median\_of\_3\_counted(array, low\_i, high\_i, res);

std::swap(array[median\_i], array[high\_i]);

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; ++res.comparison\_count, j < high\_i; ++j) {

++res.comparison\_count;

if (array[j] <= pivot) {

++res.swap\_count;

std::swap(array[i], array[j]);

++i;

}

}

++res.swap\_count;

std::swap(array[i], array[high\_i]);

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

}

sort\_result quick\_sort\_median\_of\_3(int const \*const array, int size, int const \*const sorted\_array) {

auto temp\_array = std::make\_unique\_for\_overwrite<int[]>(size);

sort\_result res{};

std::cout << "Starting timed QUICK SORT MEDIAN. Got: " << std::flush;

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

timed\_quick\_sort\_median\_of\_3(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted QUICK SORT MEDIAN. Got: " << std::flush;

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

counted\_quick\_sort\_median\_of\_3(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

void quick\_sort\_median\_of\_3\_with\_steps(int \*array, int size) {

std::cout << "Исходный массив:\n";

print\_array(array, size);

std::stack<std::pair<int, int>> sort\_stack{};

sort\_stack.emplace(0, size - 1);

while (!sort\_stack.empty()) {

auto [low\_i, high\_i] = sort\_stack.top();

sort\_stack.pop();

if (low\_i >= high\_i)

continue;

std::cout << "Сортируем подмассив:\n";

print\_array(array, size, {low\_i, high\_i});

std::cout << "Опорным элементом берём медианный среди первого, центрального и последнего, "

"затем переставляем его с последним:\n";

int median\_i = median\_of\_3(array, low\_i, high\_i);

std::swap(array[median\_i], array[high\_i]);

print\_array(array, size, {median\_i, high\_i});

int pivot = array[high\_i];

int i = low\_i;

for (int j = low\_i; j < high\_i; ++j) {

std::cout << j << "-й элемент ";

if (array[j] <= pivot) {

std::cout << "меньше опорного. Переставим его с первым элементом больше опорного\n";

std::swap(array[i], array[j]);

print\_array(array, size, {i, j});

++i;

} else {

std::cout << "больше опорного. Оставим его не месте.\n";

}

}

std::cout << "Поставим опорный элемент между большими и меньшими, переставив его с первым большим\n";

std::swap(array[i], array[high\_i]);

sort\_stack.emplace(low\_i, i - 1);

sort\_stack.emplace(i + 1, high\_i);

}

}

Selection\_srot.cpp

#include <chrono>

#include <iostream>

#include <memory>

#include <stack>

#include "../sorts.h"

#include "../helpers.h"

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

void timed\_selection\_sort(int \*const array, int size, sort\_result &res) {

auto start = hrc::now();

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

int min\_i = i;

for (int j = i + 1; j < size; ++j) {

if (array[j] < array[min\_i])

min\_i = j;

}

std::swap(array[i], array[min\_i]);

}

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

}

void counted\_selection\_sort(int \*const array, int size, sort\_result &res) {

for (int i = 0; ++res.comparison\_count, i < size; ++i) {

int min\_i = i;

for (int j = i + 1; ++res.comparison\_count, j < size; ++j) {

++res.comparison\_count;

if (array[j] < array[min\_i])

min\_i = j;

}

++res.swap\_count;

std::swap(array[i], array[min\_i]);

}

}

sort\_result selection\_sort(int const \*const array, int size, int const \*const sorted\_array) {

auto temp\_array = std::make\_unique\_for\_overwrite<int[]>(size);

sort\_result res{};

std::cout << "Starting timed SELECTION SORT. Got: " << std::flush;

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

timed\_selection\_sort(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

std::cout << "Starting counted SELECTION SORT. Got: " << std::flush;

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

counted\_selection\_sort(temp\_array.get(), size, res);

std::cout << (check\_arrays\_equal(temp\_array.get(), sorted\_array, size) ? "Correct" : "!!!!!!!!INCORRECT!!!!!!!!")

<< '\n';

return res;

};

void selection\_sort\_with\_steps(int \*array, int size) {

std::cout << "Исходный массив:\n";

print\_array(array, size);

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

int min\_i = i;

for (int j = i + 1; j < size; ++j) {

if (array[j] < array[min\_i])

min\_i = j;

}

std::cout << "Самый маленький элемент - " << array[min\_i] << " на " << min\_i << " месте. "

<< "Заменим его с " << i << "-м.\n";

std::swap(array[i], array[min\_i]);

print\_array(array, size, {i, min\_i});

}

}

## Результаты

Отсортировано по возрастанию

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS time (ns) | QSL time (ns) | QSM time (ns) | QSH time (ns) |
| 500 | 132900 | 547400 | 122500 | 324000 |
| 1000 | 531900 | 1811000 | 244700 | 754900 |
| 10000 | 52490000 | 161580800 | 2425700 | 36357100 |
| 50000 | 1301230300 | 4011418500 | 12088300 | 824964500 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS comparations | QSL comparations | QSM comparations | QSH comparations |
| 500 | 250501 | 250998 | 10609 | 123525 |
| 1000 | 1001001 | 1001998 | 23805 | 487491 |
| 10000 | 100010001 | 100019998 | 319339 | 48665458 |
| 50000 | 2500050001 | 2500099998 | 1879353 | 1213578201 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS swaps | QSL swaps | QSM swaps | QSH swaps |
| 500 | 500 | 125249 | 1376 | 115 |
| 1000 | 1000 | 500499 | 3072 | 217 |
| 10000 | 10000 | 50004999 | 40781 | 2167 |
| 50000 | 50000 | 1250024999 | 240088 | 10823 |

Отсортировано по убыванию

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS time (ns) | QSL time (ns) | QSM time (ns) | QSH time (ns) |
| 500 | 159400 | 369000 | 159300 | 108313 |
| 1000 | 602300 | 1129200 | 270100 | 438033 |
| 10000 | 59250500 | 92738100 | 2807400 | 43567321 |
| 50000 | 2175390200 | 2113260900 | 14951700 | 1086147530 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS comparations | QSL comparations | QSM comparations | QSH comparations |
| 500 | 250501 | 205451 | 15452 | 108313 |
| 1000 | 1001001 | 804187 | 38449 | 438033 |
| 10000 | 100010001 | 81052767 | 586152 | 43567321 |
| 50000 | 2500050001 | 2014952581 | 3631916 | 1086147530 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS swaps | QSL swaps | QSM swaps | QSH swaps |
| 500 | 500 | 53751 | 4738 | 368 |
| 1000 | 1000 | 207127 | 11714 | 728 |
| 10000 | 10000 | 20765701 | 183515 | 7217 |
| 50000 | 50000 | 516100013 | 1147773 | 36166 |

Не отсортировано

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS time (ns) | QSL time (ns) | QSM time (ns) | QSH time (ns) |
| 500 | 151200 | 162900 | 136300 | 202600 |
| 1000 | 565400 | 314300 | 272500 | 383600 |
| 10000 | 52966200 | 2893700 | 3259400 | 3907100 |
| 50000 | 1300463300 | 15299900 | 14959100 | 20590300 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS comparations | QSL comparations | QSM comparations | QSH comparations |
| 500 | 250501 | 10835 | 10427 | 9237 |
| 1000 | 1001001 | 24505 | 22223 | 21418 |
| 10000 | 100010001 | 350682 | 302122 | 267973 |
| 50000 | 2500050001 | 1913268 | 1777605 | 1534045 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | SS swaps | QSL swaps | QSM swaps | QSH swaps |
| 500 | 500 | 2686 | 2514 | 1101 |
| 1000 | 1000 | 6916 | 5173 | 2403 |
| 10000 | 10000 | 94544 | 70085 | 31831 |
| 50000 | 50000 | 491200 | 447668 | 187112 |

## Вывод

В данной работе мы на практике реализовали и сравнили несколько алгоритмов сортировки элементов в массиве. Проведя анализ выяснили, что время сортировки может сильно зависеть от исходных данных: при изначально отсортированном массиве данных быстрая сортировка с опорным последним элементом в большинстве случаев работала дольше, чем сортировка выбором.