ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировки»

Выполнил работу

Дубовик Никита

Академическая группа J3110

Принято

Ментор, Владислав Вершинин

Санкт-Петербург

2024

**Введение:**

Цель работы – Реализовать и протестировать 3 алгоритма сортировки.

Задачи работы:

Разобраться в принципах работы данных сортировок;

Реализовать 3 алгоритма сортировок;

Протестировать их работу на различных входных данных.

**Реализация:**

1. Реализация алгоритма сортировки comb sort

void comb\_sort(std::vector<int>& vec)

{

size\_t gap = vec.size();

bool swapped = true;

while (gap != 1 || swapped) // Суммарно(рассчитывая как геометрическую прогрессию) получим константу умноженную на O(n) и по мере уменьшения step в итоге получим O(n) \* O(n) = O(n^2)

{

gap = (gap \* 10) / 13;

if (gap < 1)

gap = 1;

swapped = false;

for (size\_t i = 0; i < vec.size() - gap; i++) // O(n / gap)

{

if (vec[i] > vec[i + gap])

{

std::swap(vec[i], vec[i + gap]);

swapped = true;

}

}

}

// Time Complexity O(n^2)

// Average Time Complexity Ω(n^2/2^p), where p is the number of increments. Или примерно O(3\*log2(n)\*n) (Из интернета)

// Space Complexity O(1)

}

1. Реализация алгоритма сортировки Introsort

void intro\_sort(std::vector<int>& vec) {

std::sort(vec.begin(), vec.end());

// Worst Time Complexity O(n log n)

// Average Time Complexity O(n log n)

// Space Complexity Варьируется

}

1. Реализация алгоритма сортировки pigeon hole sort

void pigeon\_hole\_sort(std::vector<int>& vec) {

auto [min\_it, max\_it] = std::minmax\_element(vec.begin(), vec.end()); // O(n)

const int min\_value = \*min\_it;

const int range = \*max\_it - min\_value + 1;

std::vector<int> holes(range, 0); // O(range) | O(range)

for (const int& x: vec) { // O(n)

holes[x - min\_value]++;

}

size\_t i = 0;

for (int count = 0; count < range; count++) { // Суммарно O(n)

while (holes[count] > 0) {

holes[count]--;

vec[i] = count + min\_value;

i++;

}

}

// Time Complexity O(n + range)

// Space Complexity O(range)

}

**Экспериментальная часть:**

Проведём тестирование времени выполнения алгоритмов. Для подсчёта времени использовалась стандартная библиотека #include <chrono> и std::chrono::high\_resolution\_clock::now() для определения времени до и после выполнения алгоритма

График зависимости времени выполнения алгоритма comb sort от количества входных данных с регрессионным анализом представлен на изображении №1:

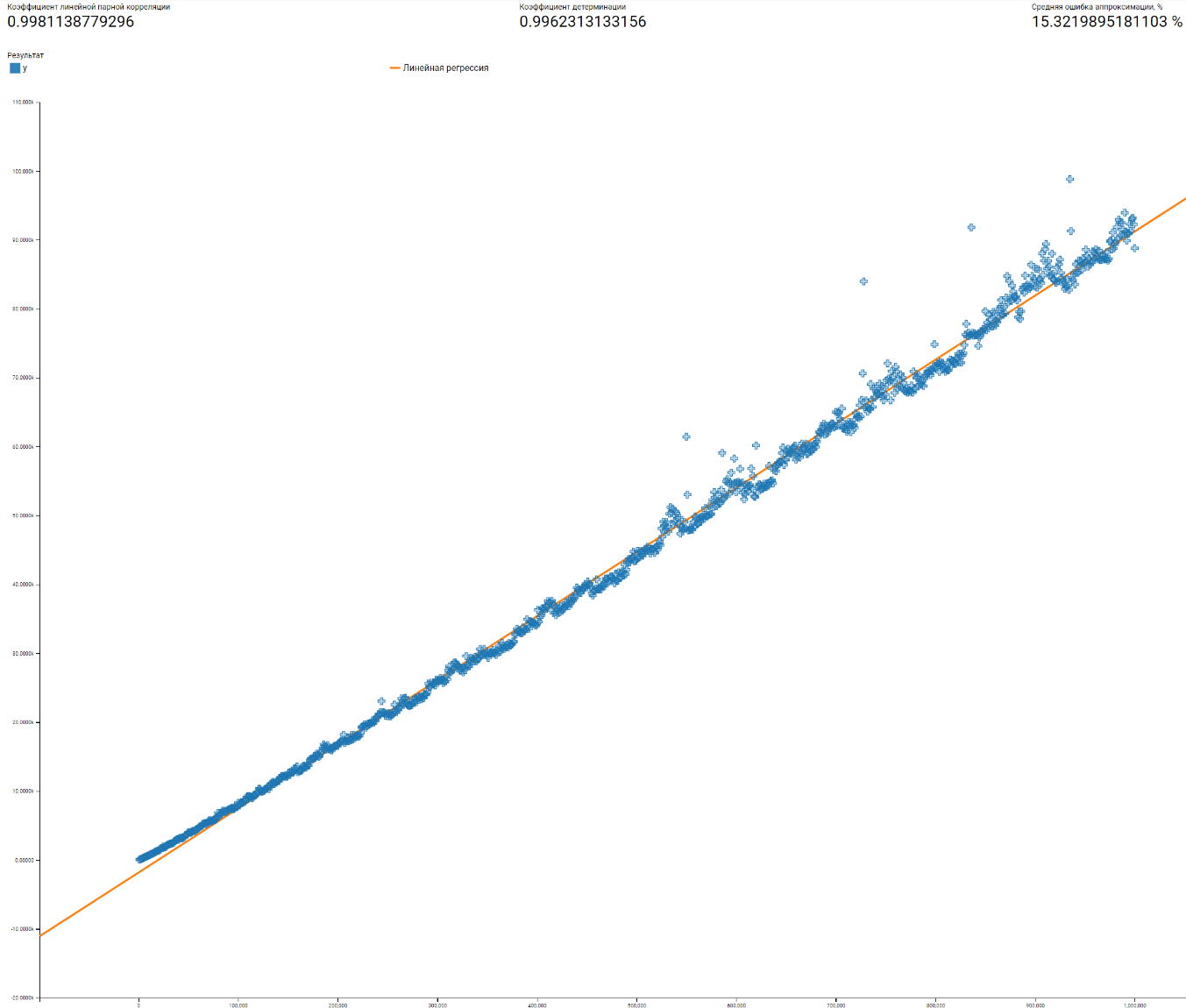


Рисунок 1 – График зависимости времени выполнения алгоритма comb sort от входных данных

График зависимости времени выполнения алгоритма intro sort от количества входных данных с регрессионным анализом представлен на изображении №2:

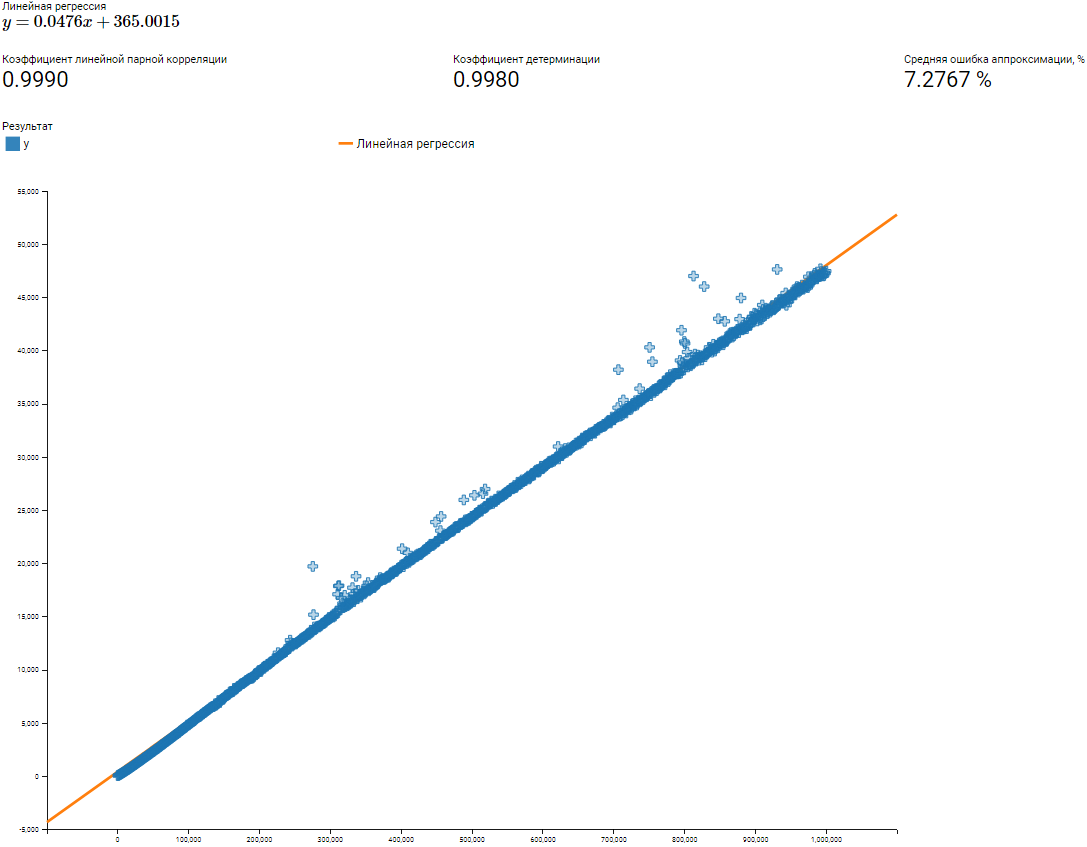


Рисунок 2 – График зависимости времени выполнения алгоритма intro sort от входных данных

График зависимости времени выполнения алгоритма intro sort от количества входных данных с регрессионным анализом представлен на изображении №3:

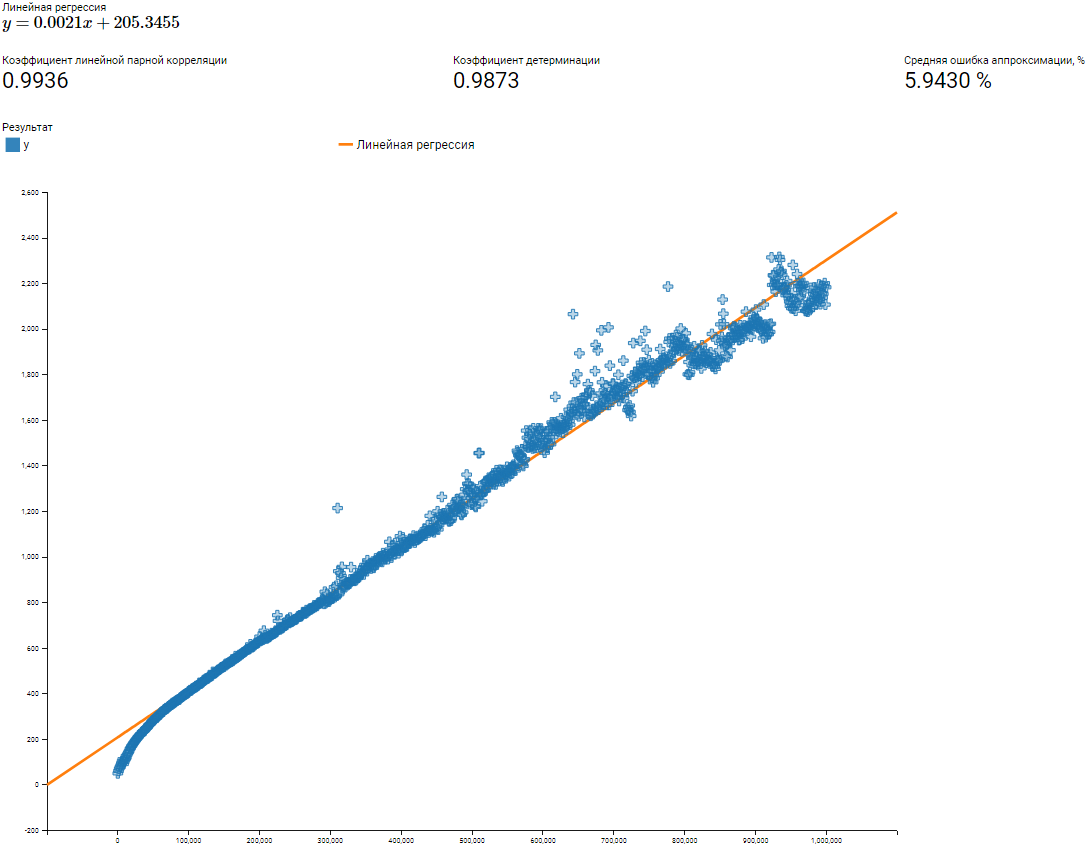


Рисунок 3 – График зависимости времени выполнения алгоритма pigeon hole sort от входных данных

Видно, что быстрее всего работает pigeon hole, но минус данного алгоритма в его не универсальности. Далее по скорости идёт intro sort. Именно этот алгоритм сортировки используется во многих реализациях алгоритма сортировки из стандартной библиотеки. Медленней всего работал comb sort, что не удивительно, так как он является немного улучшенным алгоритмом сортировки пузырьком и этот концепт является не слишком оптимальным.

Приведу box plot выполнения алгоритмов на 1e4 в массиве на изображении №4

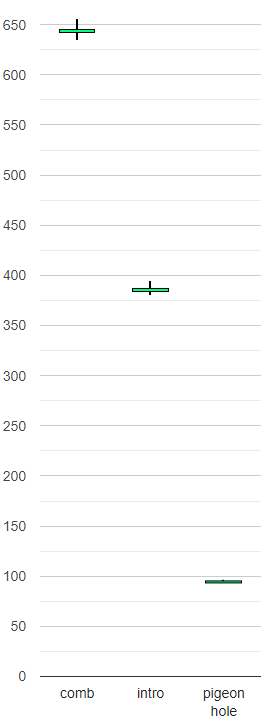


Рисунок 4 – График работы алгоритмов с размером массива 1e4

Приведу box plot выполнения алгоритмов на 1e4 в массиве на изображении №5

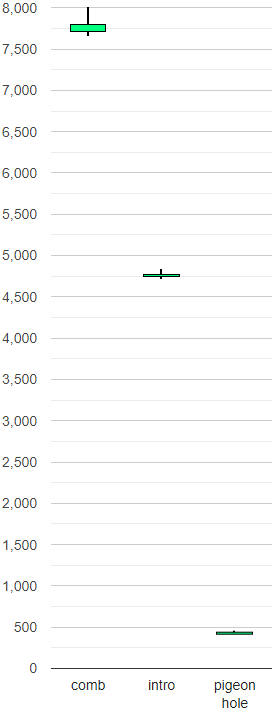


Рисунок 5 – График работы алгоритмов с размером массива 1e5

**Заключение:**

В ходе выполнения работы мною были реализованы и протестированы 3 алгоритма сортировки. Во время тестирования алгоритмов обнаружились некоторые странности. Почему-то везде появлялась линейная зависимость, хотя согласно теоретическим расчётам асимптотика не должна быть линейной.

**Приложения:**

Исходный код

#include <iostream>

#include <vector>

#include <algorithm>

#include <numeric>

#include <cassert>

#include <chrono>

void comb\_sort(std::vector<int>& vec)

{

size\_t gap = vec.size();

bool swapped = true;

while (gap != 1 || swapped) // Суммарно(рассчитывая как геометрическую прогрессию) получим константу умноженную на O(n) и по мере уменьшения step в итоге получим O(n) \* O(n) = O(n^2)

{

gap = (gap \* 10) / 13;

if (gap < 1)

gap = 1;

swapped = false;

for (size\_t i = 0; i < vec.size() - gap; i++) // O(n / gap)

{

if (vec[i] > vec[i + gap])

{

std::swap(vec[i], vec[i + gap]);

swapped = true;

}

}

}

// Time Complexity O(n^2)

// Average Time Complexity Ω(n^2/2^p), where p is the number of increments. Или примерно O(3\*log2(n)\*n) (Из интернета)

// Space Complexity O(1)

}

void intro\_sort(std::vector<int>& vec) {

std::sort(vec.begin(), vec.end());

// Worst Time Complexity O(n log n)

// Average Time Complexity O(n log n)

// Space Complexity Варьируется

}

void pigeon\_hole\_sort(std::vector<int>& vec) {

auto [min\_it, max\_it] = std::minmax\_element(vec.begin(), vec.end()); // O(n)

const int min\_value = \*min\_it;

const int range = \*max\_it - min\_value + 1;

std::vector<int> holes(range, 0); // O(range) | O(range)

for (const int& x: vec) { // O(n)

holes[x - min\_value]++;

}

size\_t i = 0;

for (int count = 0; count < range; count++) { // Суммарно O(n)

while (holes[count] > 0) {

holes[count]--;

vec[i] = count + min\_value;

i++;

}

}

// Time Complexity O(n + range)

// Space Complexity O(range)

}

template<typename T>

bool is\_sorted(const std::vector<T>& vec) {

return std::is\_sorted(vec.begin(), vec.end());

}

std::vector<int> generate\_random\_vector(size\_t size) {

std::vector<int> vec(size);

srand(time(0));

std::generate(vec.begin(), vec.end(), std::rand);

return vec;

}

void run\_tests() {

// comb\_sort

{

// Лучший случай (уже отсортированный массив)

std::vector<int> best\_case(1000);

std::iota(best\_case.begin(), best\_case.end(), 1);

comb\_sort(best\_case);

assert(is\_sorted(best\_case));

std::cout << "comb\_sort best case passed!\n";

// Средний случай (случайный массив)

auto average\_case = generate\_random\_vector(1000);

comb\_sort(average\_case);

assert(is\_sorted(average\_case));

std::cout << "comb\_sort average case passed!\n";

// Худший случай (массив в обратном порядке)

std::vector<int> worst\_case(1000);

std::iota(worst\_case.rbegin(), worst\_case.rend(), 1);

comb\_sort(worst\_case);

assert(is\_sorted(worst\_case));

std::cout << "comb\_sort worst case passed!\n";

}

// introsort

{

// Лучший случай (уже отсортированный массив)

std::vector<int> best\_case(1000);

std::iota(best\_case.begin(), best\_case.end(), 1);

intro\_sort(best\_case);

assert(is\_sorted(best\_case));

std::cout << "introsort best case passed!\n";

// Средний случай (случайный массив)

auto average\_case = generate\_random\_vector(1000);

intro\_sort(average\_case);

assert(is\_sorted(average\_case));

std::cout << "introsort average case passed!\n";

// Худший случай (массив в обратном порядке)

std::vector<int> worst\_case(1000);

std::iota(worst\_case.rbegin(), worst\_case.rend(), 1);

intro\_sort(worst\_case);

assert(is\_sorted(worst\_case));

std::cout << "introsort worst case passed!\n";

}

// pigeon\_hole\_sort

{

// Лучший случай (уже отсортированный массив)

auto best\_case = generate\_random\_vector(1000);

std::sort(best\_case.begin(), best\_case.end());

std::reverse(best\_case.begin(), best\_case.end());

pigeon\_hole\_sort(best\_case);

assert(is\_sorted(best\_case));

std::cout << "pigeon\_hole\_sort best case passed!\n";

// Средний случай (случайный массив)

auto average\_case = generate\_random\_vector(1000);

pigeon\_hole\_sort(average\_case);

assert(is\_sorted(average\_case));

std::cout << "pigeon\_hole\_sort average case passed!\n";

// Худший случай (массив с одинаковыми значениями)

std::vector<int> worst\_case(1000, 5); // Все элементы равны

pigeon\_hole\_sort(worst\_case);

assert(is\_sorted(worst\_case));

std::cout << "pigeon\_hole\_sort worst case passed!\n";

}

}

int main() {

run\_tests();

}