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

ОТЧЕТ

по курсовой работе

дисциплина «Алгоритмы и Структуры данных»

Тема: «Сравнение различных алгоритмов поиска в массиве»

Студент гр. 3351 ____________________________ Фабер К.А.

Преподаватель ____________________________ Пестерев Д.О.

Санкт-Петербург

2024

Цель лабораторной работы:

Реализация различных алгоритмов поиска в массиве, исследование их временной сложности и сравнение друг с другом.

Даны следующие алгоритмы поиска:

1) Линейный поиск
2) Бинарный поиск
3) Тернарный поиск
4) Интерполяционный поиск

Теоретическая часть.

Асимптотика алгоритмов.

Алгоритм	Асимптотическая временная сложность			Асимптотическая пространственная сложность
	Лучший случай	Средний случай	Худший случай	Все сулчаи
Линейный поиск	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$
Бинарный поиск	$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(1)$
Тернарный поиск	$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(1)$
Интерполяционный поиск	$\theta(1)$	$\theta(\log(\log n))$	$\theta(n)$	$\theta(1)$

Линейный поиск.

Линейный поиск – это алгоритм поиска в произвольном массиве, основанный на последовательном сравнении всех элементов массива с заданным ключом.

Временная сложность:

$$T_{LS}^{w}(n) = 1 + 2n + n + 2 = 3n + 3 = \theta(n)$$

$$T_{LS}^{b}(n) = 1 + 1 + 1 + 2 = 5 = \theta(1)$$

$$T_{LS}^{a}(n) = 1 + 2\frac{n}{2} + \frac{n}{2} + 2 = 1 + n + \frac{n}{2} + 2 = 3 + \frac{3}{2}n = \theta(n)$$

Пространственная сложность:

$$S_{LS}(n) = 4 = \theta(1)$$

Бинарный поиск.

Бинарный поиск – это алгоритм поиска в отсортированном массиве, который последовательно делит массив пополам и ищет ключ в одной из частей, в зависимости от значения (если ключ больше чем средний элемент, то поиск продолжается в правой части, иначе – в левой, до тех пор пока средний элемент не будет равен ключу, либо пока не переберется весь массив)

Временная сложность:

Для нахождения функции временной сложности для худшего случая воспользуемся мастер теоремой:

$T_{BS}^{w}(n) = T_{BS}^{w}(n/2) + f(n)$, где $f(n)$ – время, необходимое для разбиения массива на 2 части.
$f(n) = 1 + 1 + 1 + 1 = 4$, тогда $T_{BS}^{w}(n) = T_{BS}^{w}(n/2) + 4$

$T_{BS}^{w}(n) = T_{BS}^{w}(n/4) + 4 + 4 = T_{BS}^{w}(n/8) + 4 + 4 + 4 = T_{BS}^{w}(n/2^{k}) + k * 4$, где k – количество делений массива на 2 части.

Продолжаем до тех пор, пока не дойдем до одного элемента ($T_{BS}^{w}(n/2^{k}) = T_{BS}^{w}(1)$):

$T_{BS}^{w}(n) = T_{BS}^{w}(1) + k * 4;$

$T_{BS}^{w}(1) = 1 + 1 + 1 = 3$

$\frac{n}{2^{k}} = 1$, тогда $k = \log_2 n$

Подставим результаты:

$T_{BS}^{w}(n) = 4 + 4 * \log_2 n = \theta(\log n)$

$T_{BS}^{b}(n) = 1 + 2 + 1 + 1 + 1 = 6 = \theta(1)$

$T_{BS}^{a}(n) = \theta(\log n)$

Пространственная сложность:

$S_{BS}(n) = 4 + 4 + 4 = 12 = \theta(1)$

Тернарный поиск.

Тернарный поиск – это алгоритм поиска в отсортированном массиве, который основан на последовательном делении массива на 3 части. Он сравнивает искомый элемент с двумя третьими элементами массива. Пусть m1 и m2 – индексы, которые разделяют первую и вторую треть и вторую и третью треть части массива соответственно, тогда, если ключ больше m1 и меньше m2, то поиск продолжается в средней части, если же ключ меньше m1, поиск продолжается в левой части, если ключи больше m2 – поиск продолжается в правой части. Массив делится на 3 части пока элемент на индексах m1 или m2 не станет равен ключу, либо пока не переберется весь массив.

Временная сложность:

Рассчитаем функцию временной сложности для худшего случая при помощи мастер теоремы аналогично бинарному поиску:

$$T_{TS}^{w}(n) = T_{TS}^{w}\left(\frac{n}{3}\right) + f(n);$$

$$f(n) = 1 + 1 + 1 + 2 + 2 + 2 = 9;$$

$$T_{TS}^{w}(1) = 1 + 1 = 2;$$

$$T_{TS}^{w}(n) = T_{TS}^{w}\left(\frac{n}{3^k}\right) + k * 9;$$

$$T_{TS}^{w}(n) = 2 + 9\log_3 n = \theta(\log n)$$

$$T_{TS}^{b}(n) = 4 + 1 + 1 = 6 = \theta(1)$$

$$T_{TS}^{a}(n) = \theta(\log n)$$

Пространственная сложность:

$$S_{TS}(n) = 4 * 4 = 16 = \theta(1)$$

Интерполяционный поиск.

Интерполяционный поиск – это алгоритм поиска в отсортированном массиве, который основан на предсказании индекса искомого элемента, по интерполирующей элементы массива функции y = f(i), где i – индекс элемента y

Временная сложность:

При использовании линейной функции для интерполяции элементов худшим случаем является неравномерно распределенные элементы, например, если элементы описывают экспоненциальную функцию: $y = f(i) = ae^{bi} + c$ и искомый ключ находится в точке выпуклости графика $f(i)$, в таком случае мы будем последовательно сдвигать назад крайний индекс, пока не дойдем до искомого элемента, следовательно:

$$T_{IS}^{w}(n) = \theta(n)$$

$T_{IS}^{b}(n) = 1 + 2 + 3 + 1 + 1 + 2 = 10 = \theta(1)$

$T_{IS}^{a}(n) = \theta(\log(\log n))$

Пространственная сложность:

$S_{IS}(n) = 4 * 2 = 8 = \theta(1)$

Практическая часть.

Поиск несуществующего элемента:

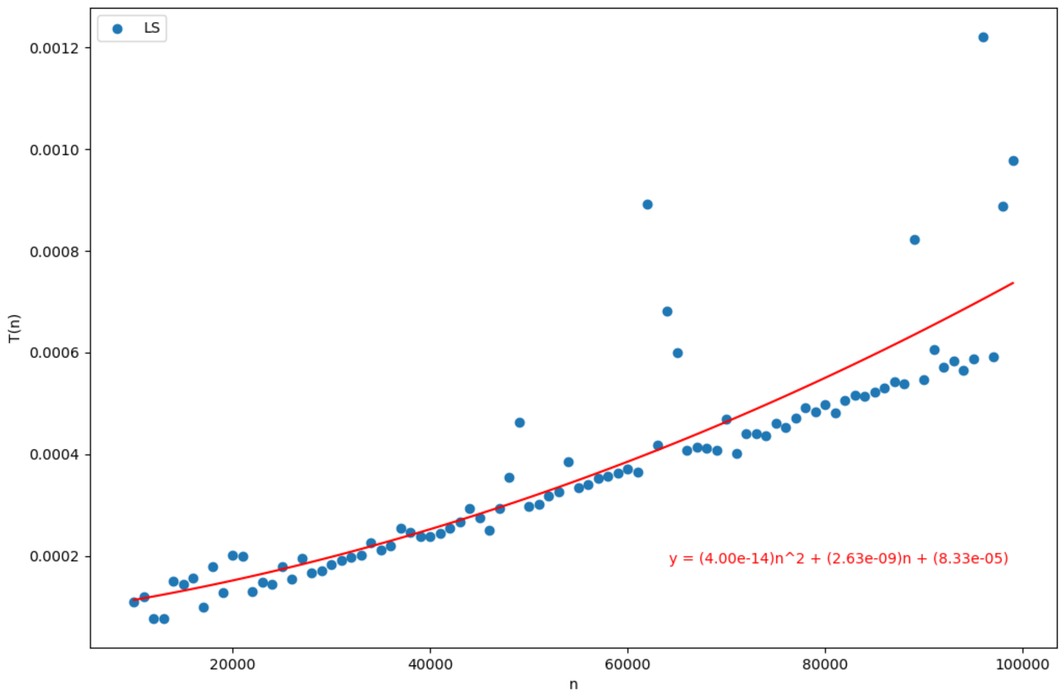

Рисунок 1 – Временная сложность линейного поиска несуществующего элемента в отсортированном массиве

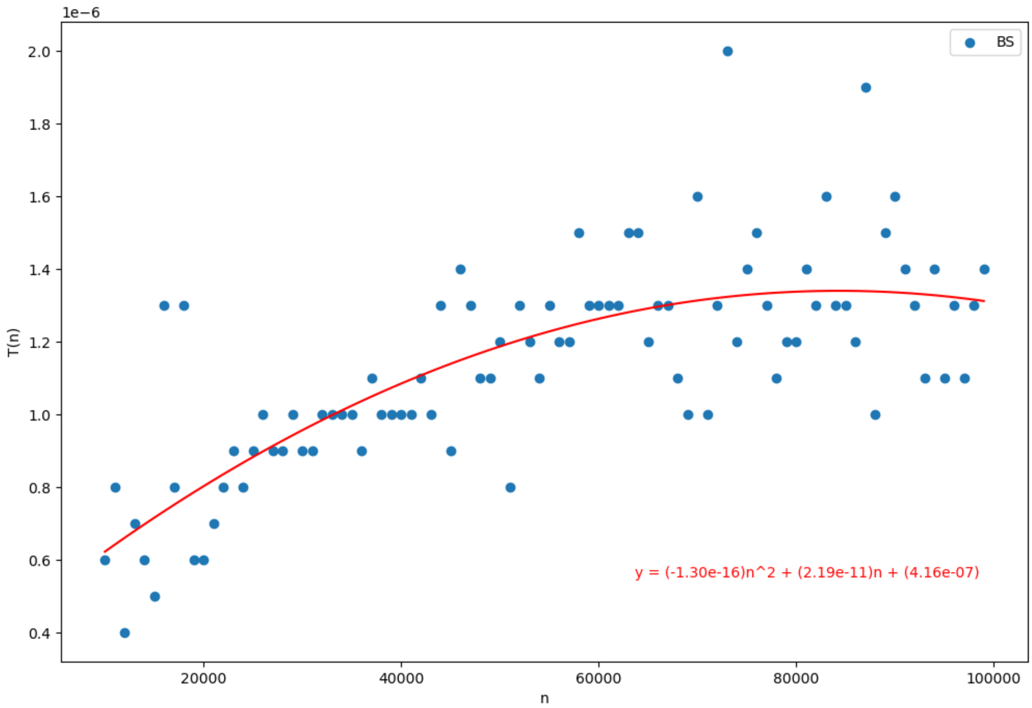

Рисунок 2 – Временная сложность бинарного поиска несуществующего элемента в отсортированном массиве

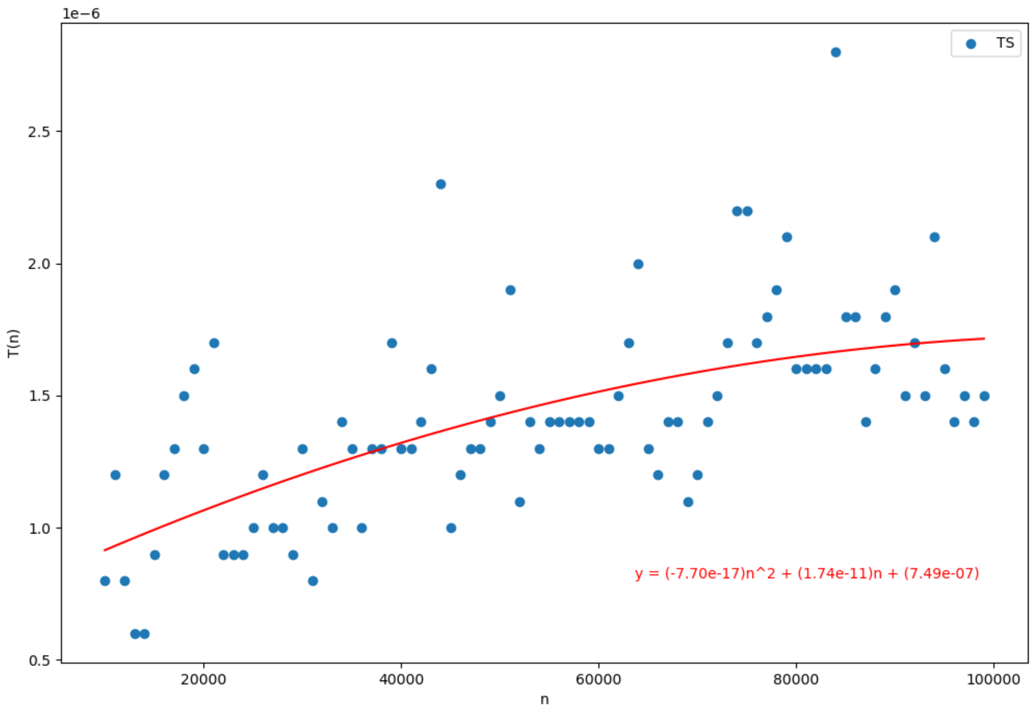

Рисунок 3 – Временная сложность тернарного поиска несуществующего элемента в отсортированном массиве

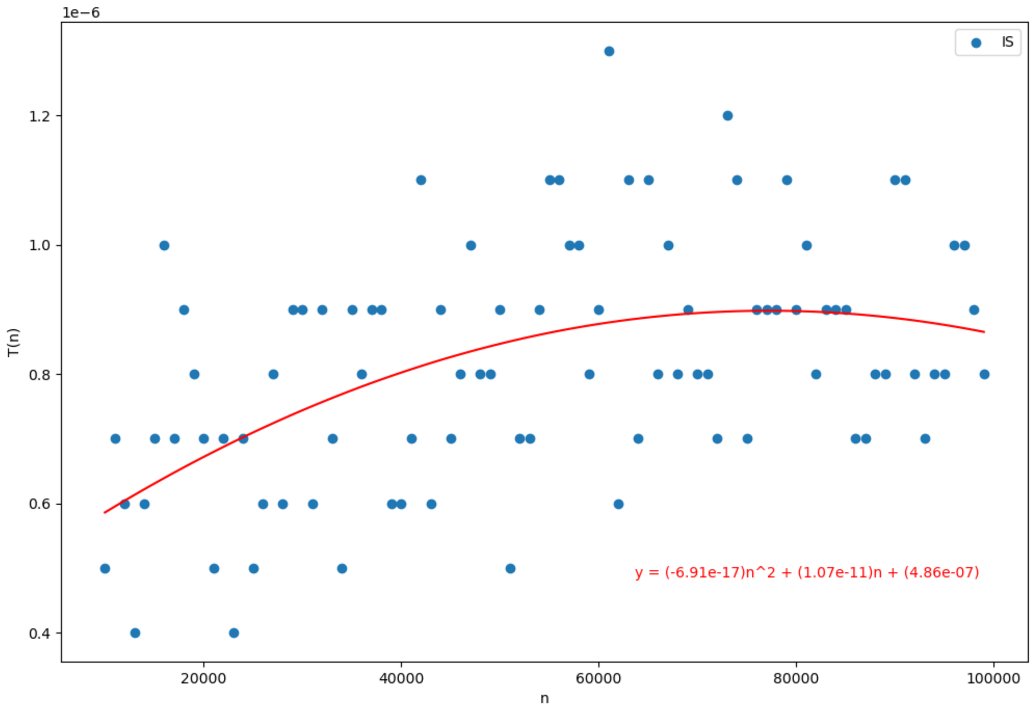

Рисунок 4 – Временная сложность интерполяционного поиска несуществующего элемента в отсортированном массиве

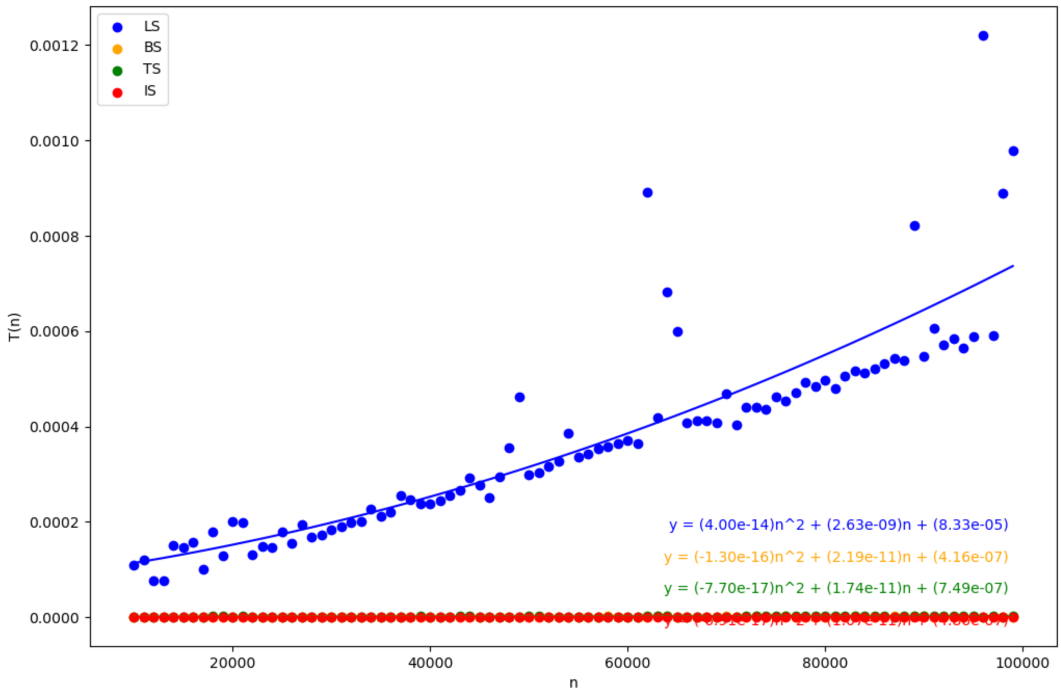

Рисунок 5 – Временная сложность всех алгоритмов поиска несуществующего элемента в отсортированном массиве

Рисунок 6 – Временная сложность всех алгоритмов поиска (кроме линейного) несуществующего элемента в отсортированном массиве

Поиск первого элемента:

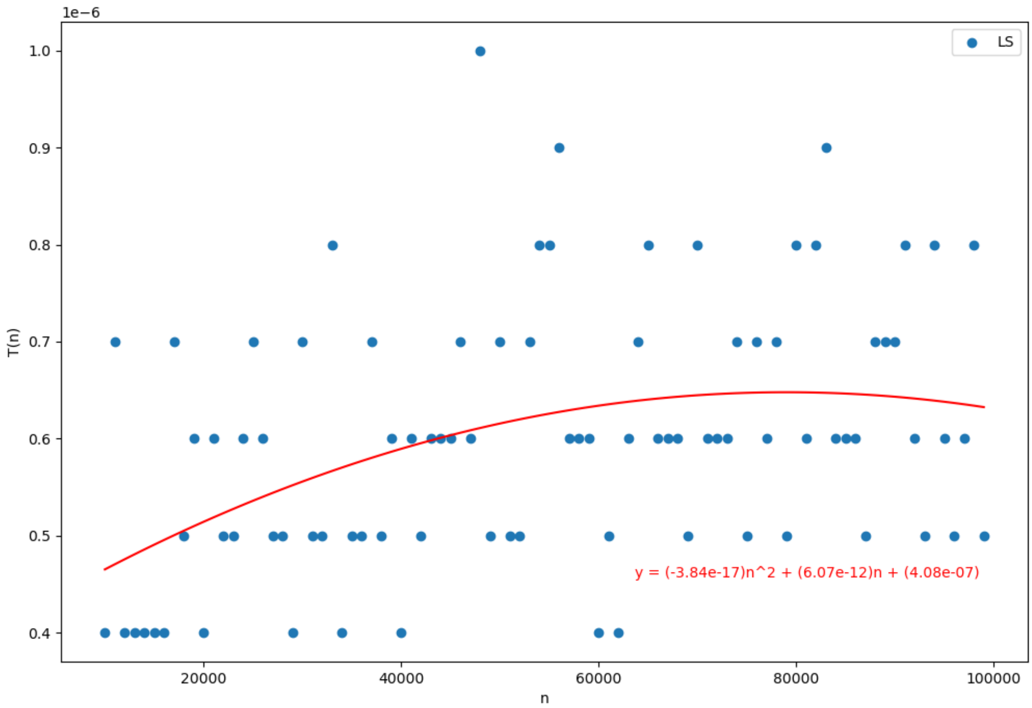

Рисунок 7 – Временная сложность линейного поиска первого элемента в отсортированном массиве

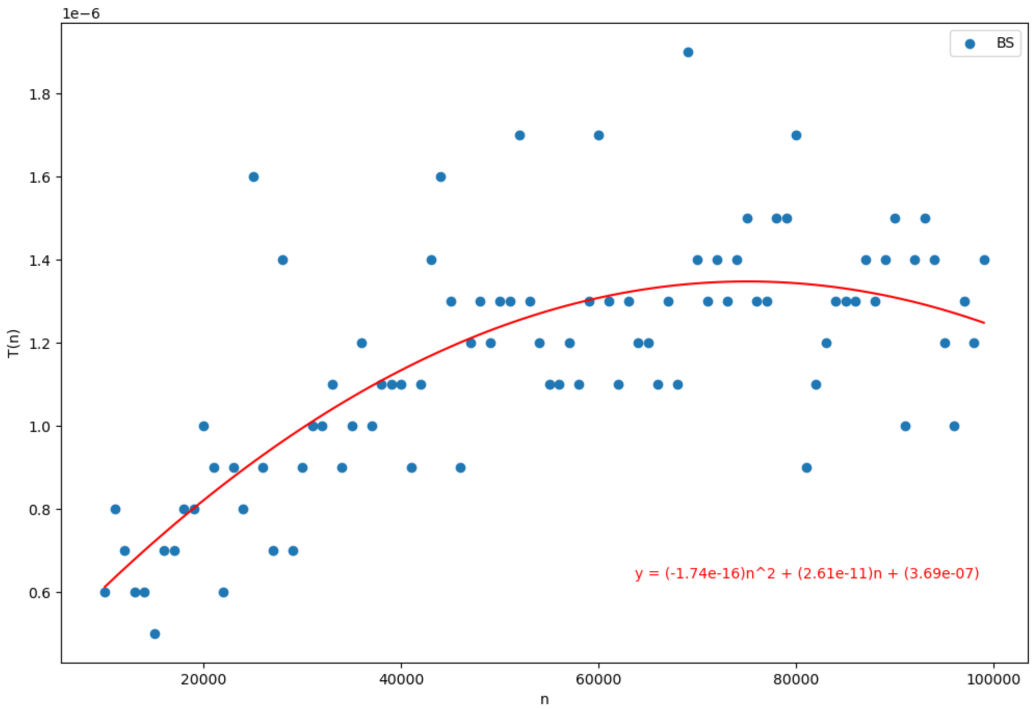

Рисунок 8 – Временная сложность бинарного поиска первого элемента в отсортированном массиве

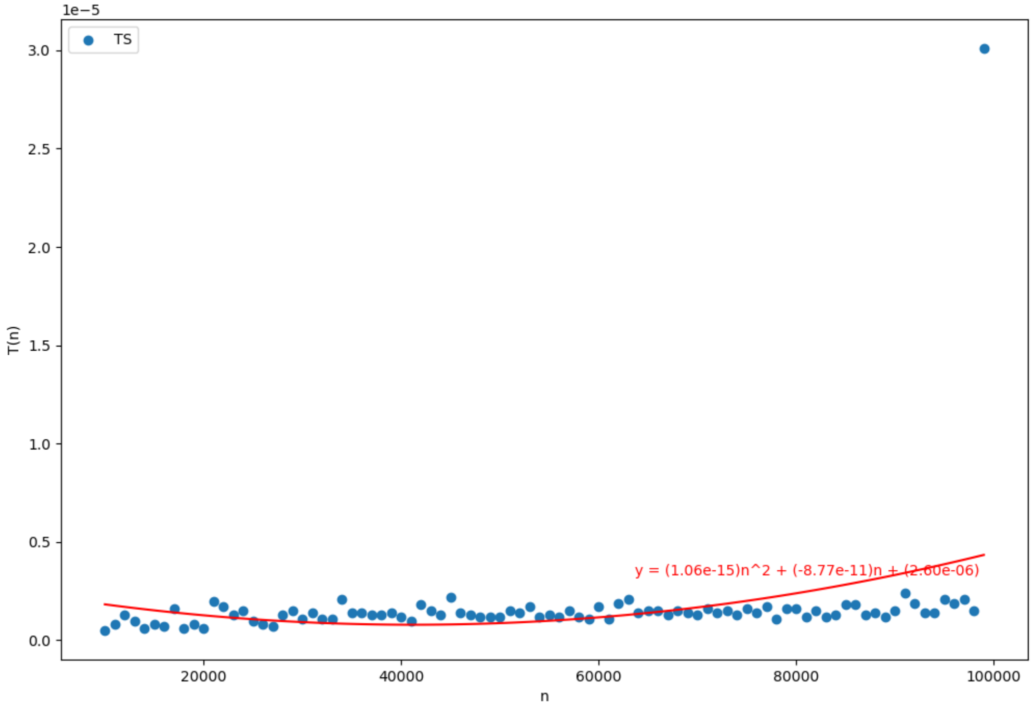

Рисунок 9 – Временная сложность тернарного поиска первого элемента в отсортированном массиве

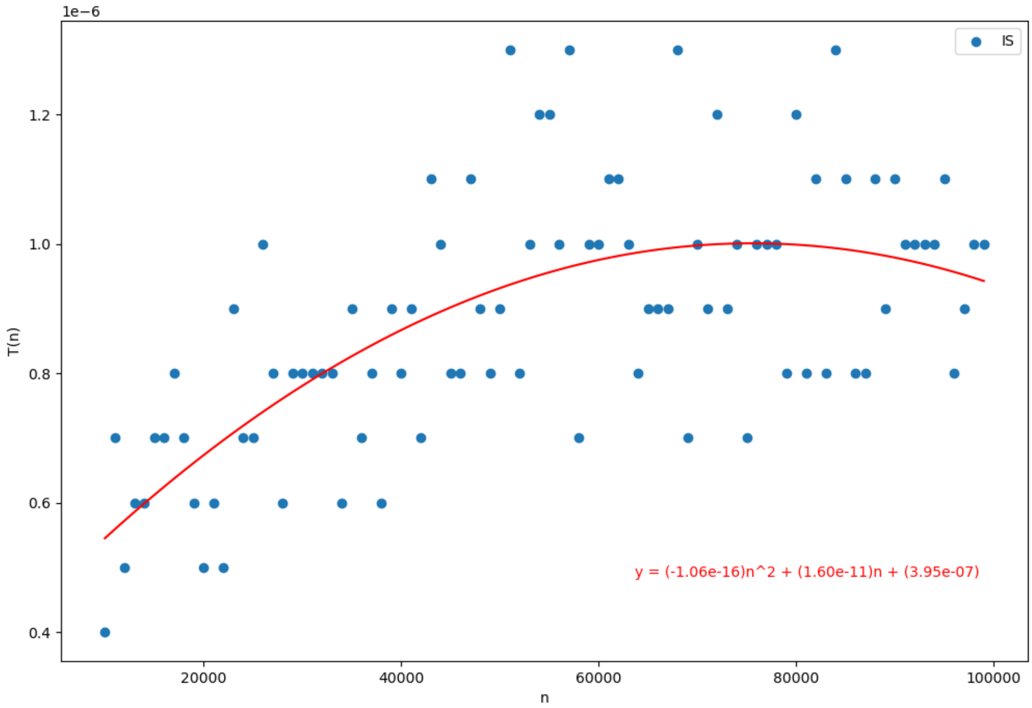

Рисунок 10 – Временная сложность интерполяционного поиска первого элемента в отсортированном массиве

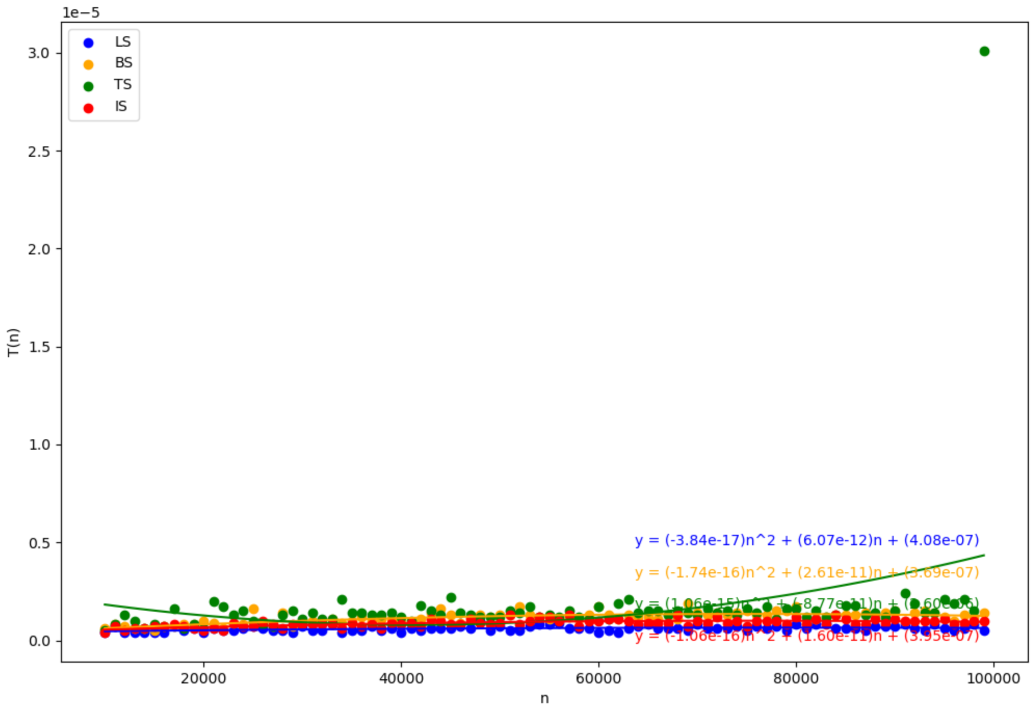

Рисунок 11 – Временная сложность всех алгоритмов поиска первого элемента в отсортированном массиве

Поиск среднего элемента:

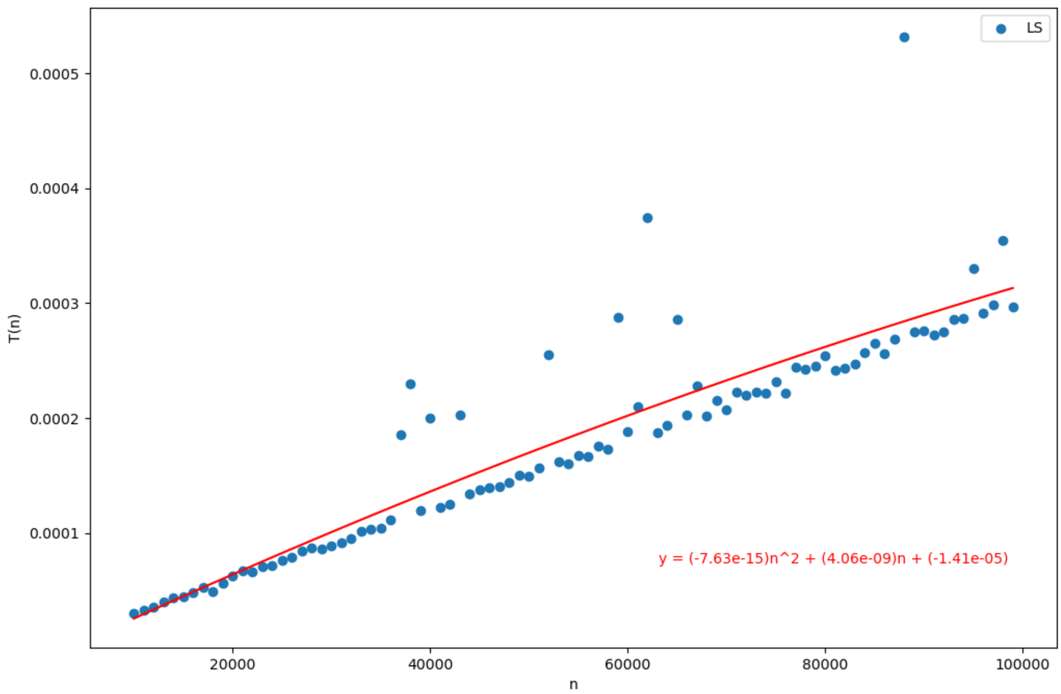

Рисунок 12 – Временная сложность линейного поиска среднего элемента в отсортированном массиве

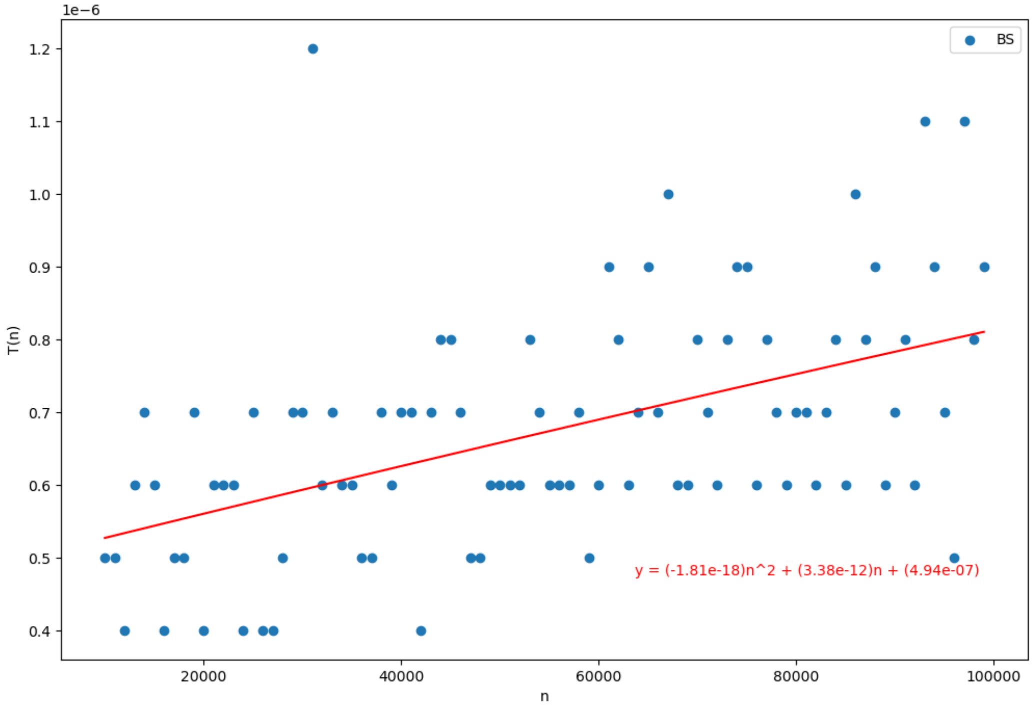

Рисунок 13 – Временная сложность бинарного поиска среднего элемента в отсортированном массиве

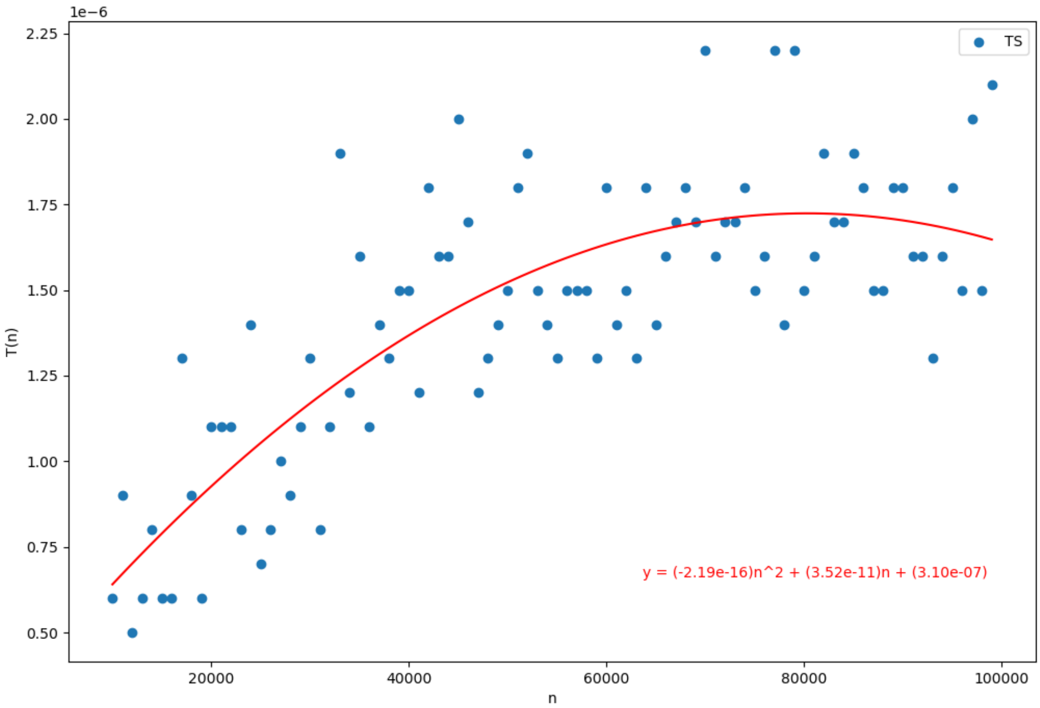

Рисунок 14 – Временная сложность тернарного поиска среднего элемента в отсортированном массиве

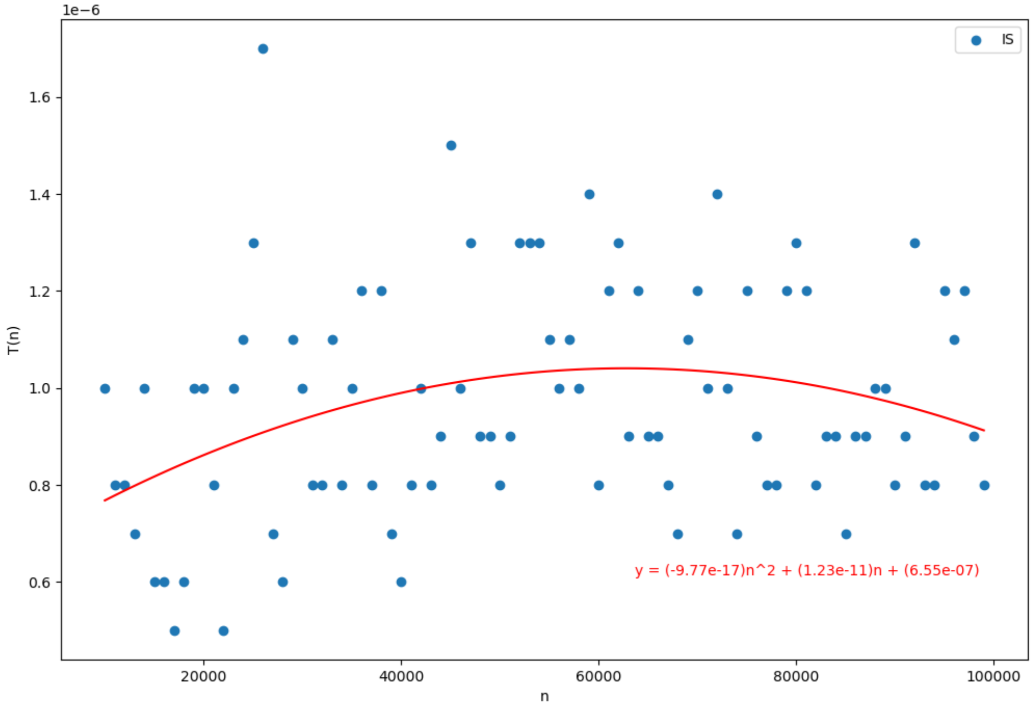

Рисунок 15 – Временная сложность интерполяционного поиска среднего элемента в отсортированном массиве

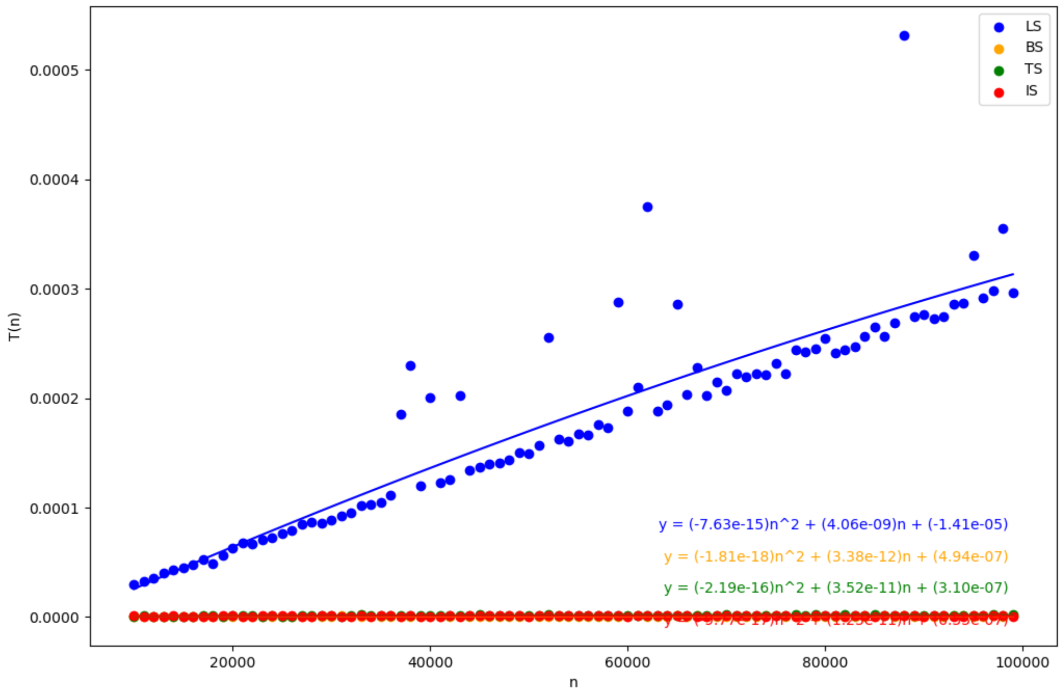

Рисунок 16 – Временная сложность всех алгоритмов поиска среднего элемента в отсортированном массиве

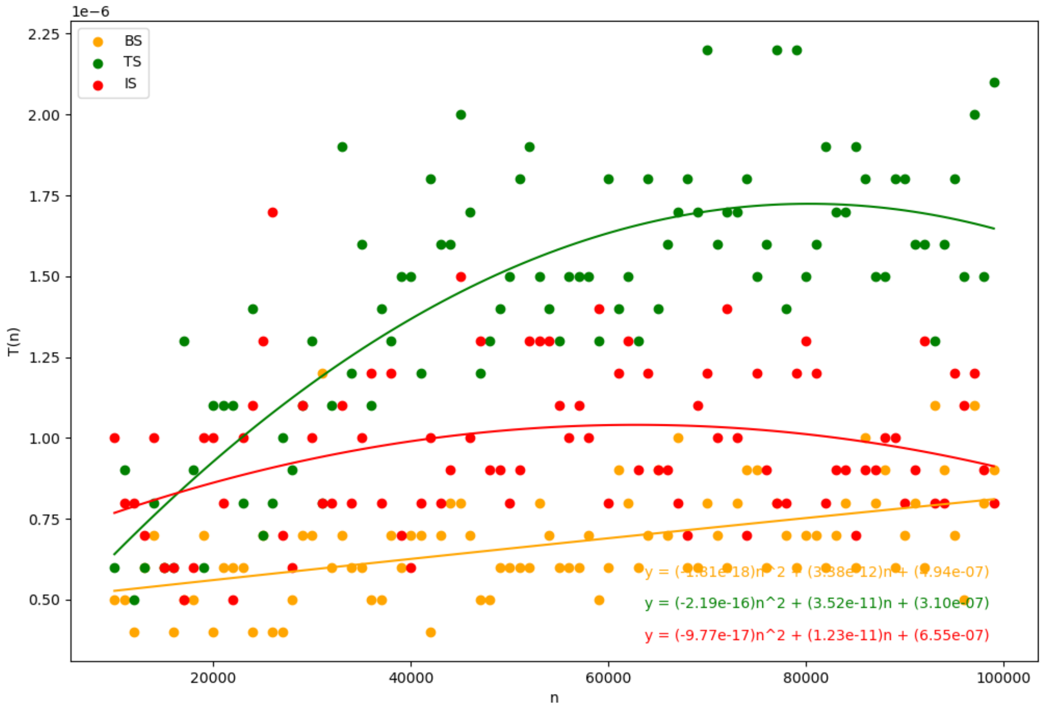

Рисунок 17 – Временная сложность всех алгоритмов поиска (кроме линейного) среднего элемента в отсортированном массиве

Поиск последнего элемента:

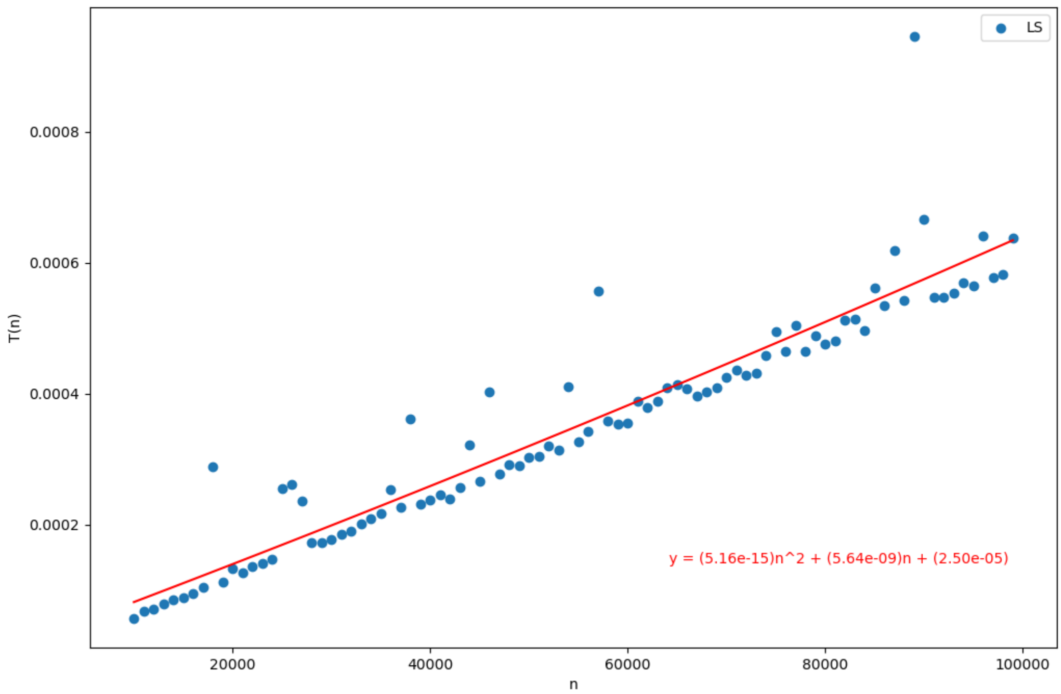

Рисунок 18 – Временная сложность линейного поиска последнего элемента в отсортированном массиве

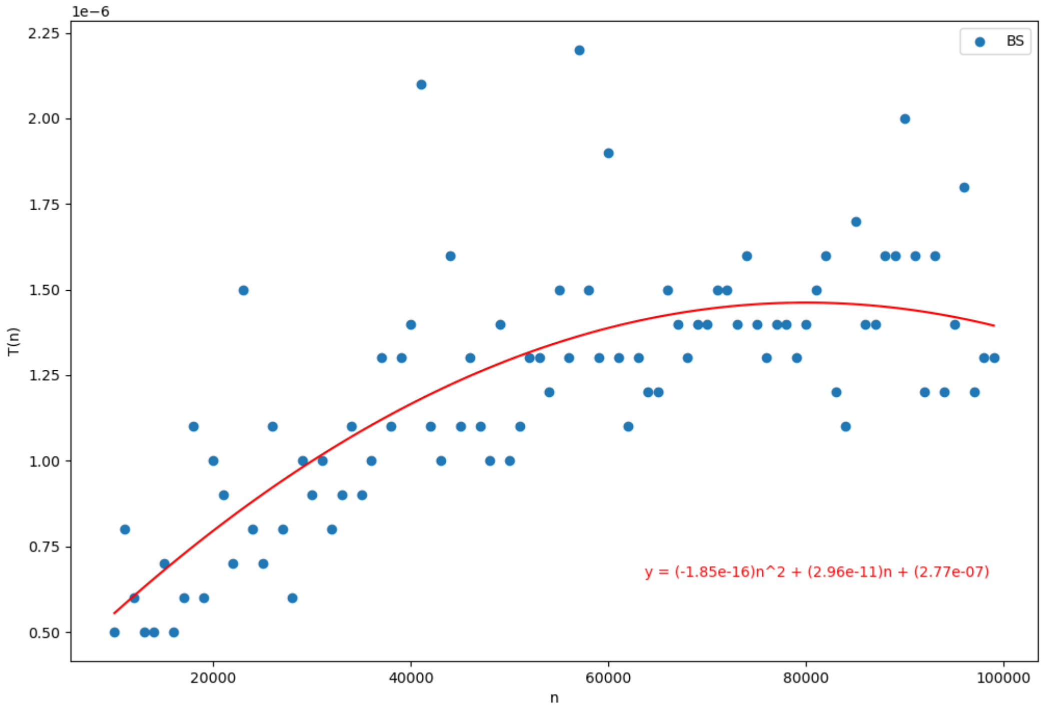

Рисунок 19 – Временная сложность бинарного поиска последнего элемента в отсортированном массиве

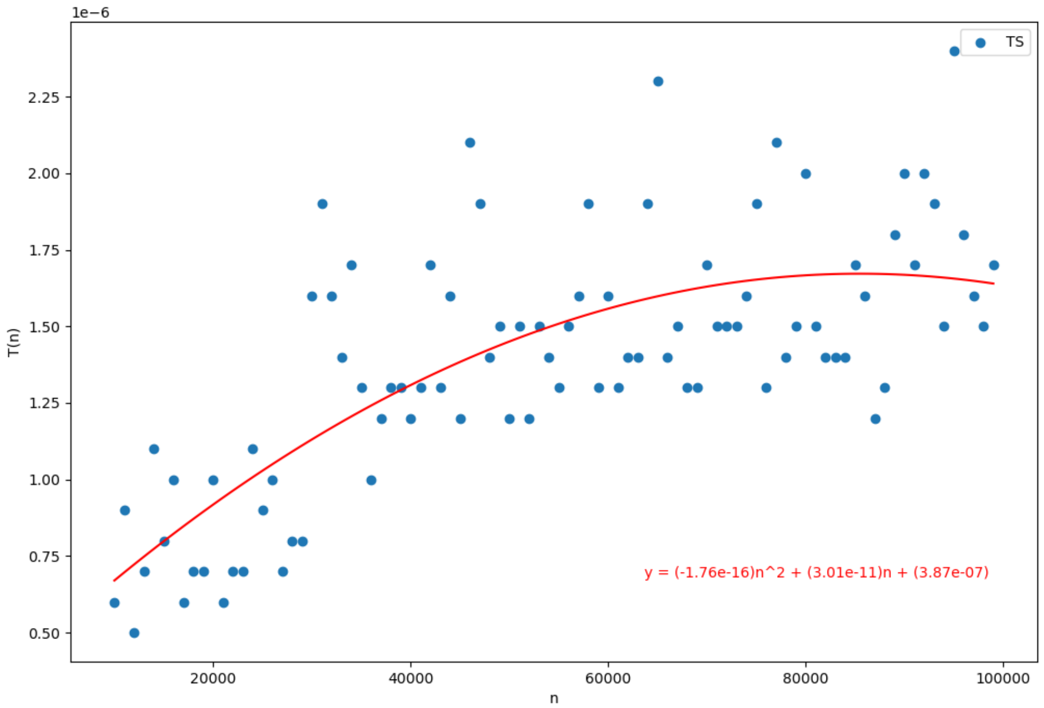

Рисунок 20 – Временная сложность тернарного поиска последнего элемента в отсортированном массиве

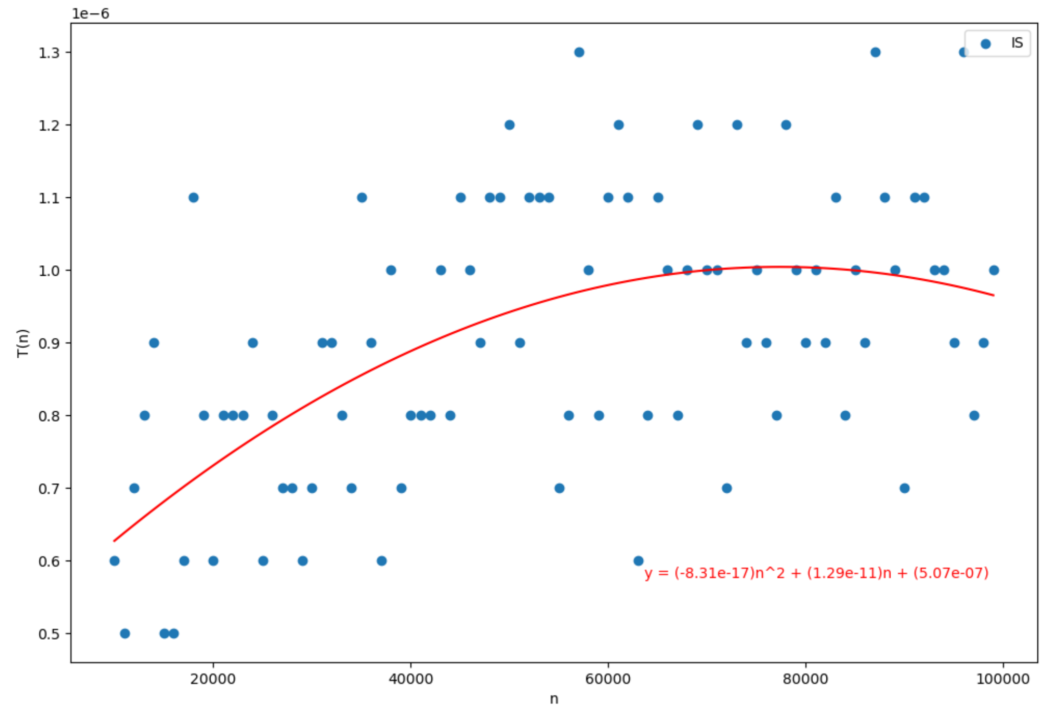

Рисунок 21 – Временная сложность интерполяционного поиска последнего элемента в отсортированном массиве

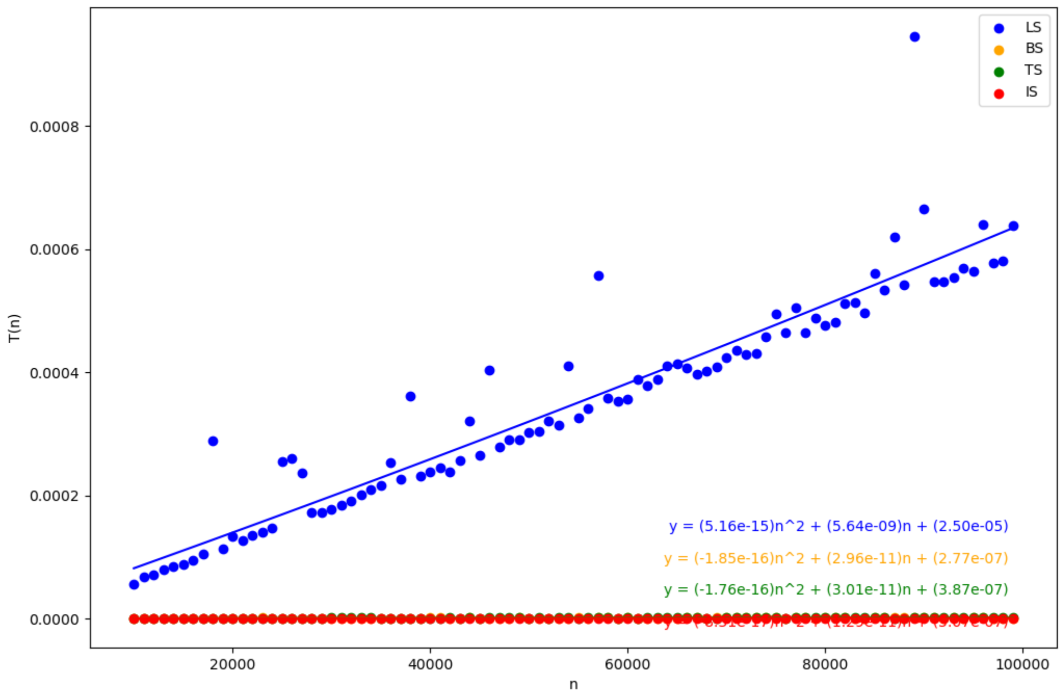

Рисунок 22 – Временная сложность всех алгоритмов поиска последнего элемента в отсортированном массиве

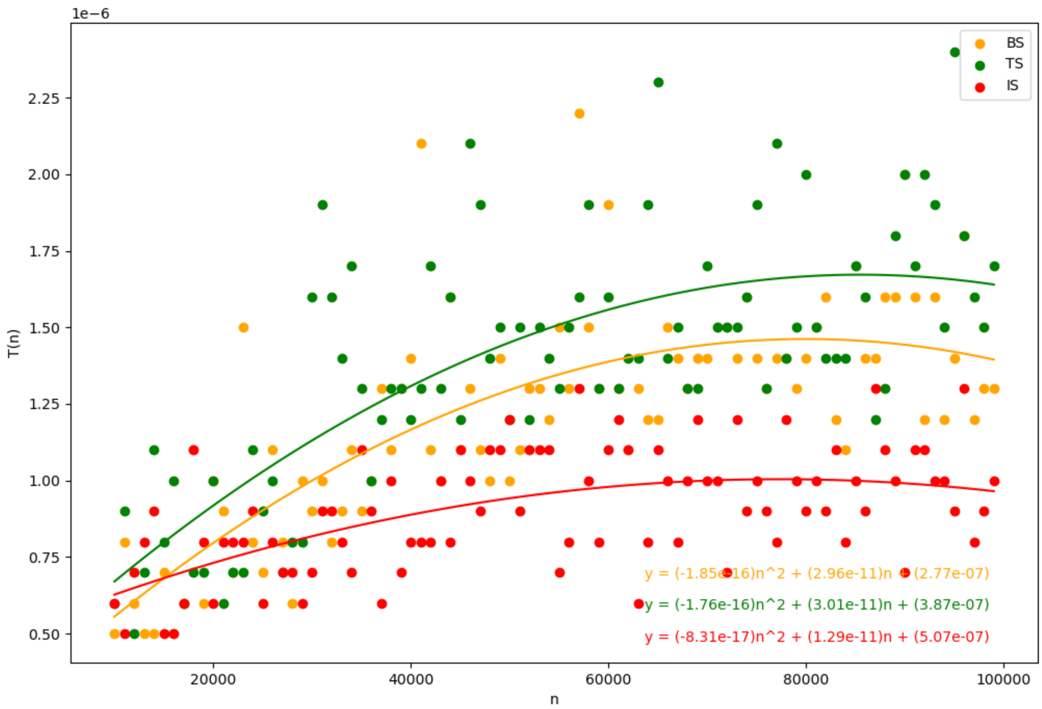

Рисунок 23 – Временная сложность всех алгоритмов поиска (кроме линейного) последнего элемента в отсортированном массиве

Вывод:

Сравнивая практические графики с теорией можно сделать вывод, что в большинстве случаев графики согласуются с теорией и демонстрируют ожидаемую асимптотику, за исключением некоторых отклонений. Также можно сделать вывод, что каждый поиск в зависимости от заполнения массива и в зависимости от местоположения искомого ключа, имеет свои особенности и однозначно сказать какой из них лучше для произвольного случая нельзя. Для неотсортированных массивов наиболее подходящим алгоритмом является линейный поиск, так как остальные алгоритмы требуют предварительной сортировки данных. Бинарный и тернарный поиски особенно эффективны для нахождения средних элементов в отсортированных массивах. Они быстро сужают диапазон поиска, что позволяет находить искомый элемент за логарифмическое время. Если мы знаем что элементы в массиве распределены равномерно, то лучшим решением для поиска будет интерполяционный поиск, т.к. он использует линейную интерполяцию для предсказания индекса.

Ссылки:

Репозиторий GitHub: https://github.com/KIRILLFABER/AicdKurs

Код:

main.cpp

```
#include <iostream>
#include <vector>
#include "Searches.h"
#include "Data.h"
#include <random>

using namespace std;

int main() {
	fillDataFile("DATA.csv");
	return 0;
}
```

Data.cpp

```
#include "Data.h"



const long int FROM = 1e4;
const long int TO = 1e5;
const int STEP = 1e3;
const long int MAX_VALUE = 1e6;

void fillArray(vector<int>& arr, int TO) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, MAX_VALUE);

    std::unordered_set<int> unique_numbers;

    while (unique_numbers.size() < TO) {
        int val = dis(gen);
        unique_numbers.insert(val);
    }

    arr.assign(unique_numbers.begin(), unique_numbers.end());
    std::sort(arr.begin(), arr.end());
}

void fillDataFile(string filename) {
	ofstream data_file(filename);
    if (!data_file.is_open()) {
        cout << "ERROR\n";
        return;
    }

    data_file << "search;case;n;T(n)\n";

    // LS
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);

        auto start = chrono::high_resolution_clock::now();
        linearSearch(arr, MAX_VALUE * 2);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
```

```
        double T = duration.count();
        data_file << "LS;W;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        linearSearch(arr, arr[0]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "LS;F;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        linearSearch(arr, arr[(n - 1)/2]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "LS;M;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        linearSearch(arr, arr[n - 1]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "LS;L;" << n << ";" << T << endl;
    }
    cout << "LINEAR SEARCH: DONE\n";
    // BS
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        binarySearch(arr, MAX_VALUE * 2);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "BS;W;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        binarySearch(arr, arr[0]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "BS;F;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        binarySearch(arr, arr[(n - 1)/2]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "BS;M;" << n << ";" << T << endl;
    }
```

```
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    binarySearch(arr, arr[n - 1]);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "BS;L;" << n << ";" << T << endl;
}
cout << "BINARY SEARCH: DONE\n";
// TS
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    ternarySearch(arr, MAX_VALUE * 2);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "TS;W;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    ternarySearch(arr, arr[0]);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "TS;F;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    ternarySearch(arr, arr[(n - 1) / 2]);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "TS;M;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    ternarySearch(arr, arr[n - 1]);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "TS;L;" << n << ";" << T << endl;
}
cout << "TERNARY SEARCH: DONE\n";
// IS
for (int n = FROM; n < TO; n += STEP) {
    vector<int> arr;
    fillArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    interpolationSearch(arr, MAX_VALUE * 2);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    double T = duration.count();
    data_file << "IS;W;" << n << ";" << T << endl;
}
for (int n = FROM; n < TO; n += STEP) {
```

```
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        interpolationSearch(arr, arr[0]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "IS;F;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        interpolationSearch(arr, arr[(n - 1) / 2]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "IS;M;" << n << ";" << T << endl;
    }
    for (int n = FROM; n < TO; n += STEP) {
        vector<int> arr;
        fillArray(arr, n);
        auto start = chrono::high_resolution_clock::now();
        interpolationSearch(arr, arr[n - 1]);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        double T = duration.count();
        data_file << "IS;L;" << n << ";" << T << endl;
    }
    cout << "INTERPOLATION SEARCH: DONE\n";


    data_file.close();
}
```

Data.h

```
#pragma once
#include <string>
#include <fstream>
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <time.h>
#include <unordered_set>
#include "Searches.h"



using namespace std;

void fillDataFile(string filename);
void fillArray(vector<int>& arr, int size);
```

Searches.cpp

```
#include "Searches.h"


int binarySearch(vector<int>& arr, int key) {
      if (arr.empty()) return -1;
      int left = 0, right = arr.size() - 1;
      while (left <= right) {
            int mid = left + (right - left) / 2;
```

```
		if (arr[mid] == key) {
			return mid;
		}
		if (arr[mid] < key) {
			left = mid + 1;
		}
		else {
			right = mid - 1;
		}
	}
	return -1;


	// Time complexity
	// worst case:
	// best case: 1 + 2 + 1 + 1 + 1 = 6 = O(1)
	// average case:
	// Space complexity
	// 4 + 4 + 4 = O(1)
}

int linearSearch(vector<int>& arr, int key) {
	int i = 0;
	for (; i < arr.size() && arr[i] != key; i++);
	return i < arr.size() && arr[i] == key ? i : -1;
	// Time complexity
	// worst case: 1 + 2n + n + 2 = 3n + 3 = O(n)
	// best case: 1 + 1 + 1  + 2 = 5 = O(1)
	// average case: 1 + 2(n / 2) + n / 2 + 2 = 1 + n + n/2 + 2 = 3 + 3/2n = O(n)
	// Space complexity
	// 4 = O(1)
}

int ternarySearch(vector<int>& arr, int key) {
	int left = 0;
	int right = arr.size() - 1;
	int m1 = left + (right - left) / 3;
	int m2 = right - (right - left) / 3;
	while (left <= right) {
		if (key == arr[m1]) {
			return m1;
		}
		if (key == arr[m2]) {
			return m2;
		}
		if (arr[m1] < key && arr[m2] > key) {
			right = m2 - 1;
			left = m1 + 1;
		}
		if (arr[m1] > key) {
			right = m1 - 1;
		}
		if (arr[m2] < key) {
			left = m2 + 1;
		}
		m1 = left + (right - left) / 3;
		m2 = right - (right - left) / 3;
	}
	return -1;

	// Time complexity
	// worst case:
	// best case: 4 + 1 + 1 = 6
	// average case:
	// Space complexity
```

```
	// 4 * 4 = 16 = O(1)
}
int interpolationSearch(vector<int>& arr, int key) {
	if (arr.empty()) return -1;
	int left = 0;
	int right = arr.size() - 1;

	while (left <= right && key >= arr[left] && key <= arr[right]) {
		if (left == right) {
			if (arr[left] == key) return left;
			return -1;
		}

		int i = left + ((key - arr[left]) * (right - left)) / (arr[right] - 
arr[left]);

		if (i < left || i > right) {
			return -1;
		}

		if (arr[i] == key) {
			return i;
		}

		if (arr[i] < key) {
			left = i + 1;
		}
		else {
			right = i - 1;
		}
	}

	return -1;

	// Time complexity
	// worst case:
	// best case: 1 + 2 + 3 + 1 + 1 + 2 = 10
	// average case:
	// Space complexity
	// 4 * 2 = 8 = O(1)
}
```

Searches.h

```
#pragma once
#include <vector>

using namespace std;

int binarySearch(vector<int>& arr, int key);
int linearSearch(vector<int>& arr, int key);
int ternarySearch(vector<int>& arr, int key);
int interpolationSearch(vector<int>& arr, int key);
```

graphics.py

import csv

import matplotlib.pyplot as plt

import numpy as np

```
filename = "DATA.csv"
X = 12
Y = 8


def readData(n, h, tree):
    with open(filename, "r") as r_file:
        reader = csv.DictReader(r_file, delimiter=";")  # Чтение данных из DATA
        for row in reader:


            if(row["search"] == tree):
                n.append(int(row["n"]))
                h.append(float(row["T(n)"]))


def reg(n, T, col, index):
    # Находим коэффициенты регрессии 2-й степени
    coefficients = np.polyfit(n, T, 2)
    polynomial_regression = np.poly1d(coefficients)

    # Генерация значений для кривой регрессии
    x_reg = np.linspace(min(n), max(n), 100)
    y_reg = polynomial_regression(x_reg)

    # Построение кривой регрессии
    plt.plot(x_reg, y_reg, color=col)

    a, b, c = coefficients



    # Сдвиг уравнения
        plt.text(0.95, 0.2 - 0.05 * index, f"y = ({a:.2e})n^2 + ({b:.2e})n + ({c:.2e})",
transform=plt.gca().transAxes,
```

```
        fontsize=10, color=col, ha='right', va='top')




def readData(search, case, n, T):
    with open(filename, "r") as r_file:
        reader = csv.DictReader(r_file, delimiter=";")  # Чтение данных из DATA
        for row in reader:
            if(row["search"] == search and row["case"] == case):
                n.append(int(row["n"]))
                T.append(float(row["T(n)"]))


def plotGraphic(search, case, n, T, filename, index = 1, color = "red"):
    plt.clf()
    plt.figure(figsize=(X, Y))
    plt.xlabel("n")
    plt.ylabel("T(n)")
    n.clear()
    T.clear()
    # Чтение данных из csv файла
    readData(search, case, n, T)
    # Построение точек
    plt.scatter(n, T, label=search)
    #Построение регрессии
    reg(n, T, "red", index)
    plt.legend()
    # Сохранение файла
    plt.savefig(filename)
    plt.close()
```

```
def plotAllGraphics(searches, case, n, T, filename, without = None):
    plt.clf()
    plt.figure(figsize=(X, Y))
    plt.xlabel("n")
    plt.ylabel("T(n)")
    n.clear()
    T.clear()
    n1, n2, n3, n4 = [], [], [], []
    T1, T2, T3, T4 = [], [], [], []
    n = [n1, n2, n3, n4]
    T = [T1, T2, T3, T4]
    i = 0
    without_index = -1
    for search in searches:
        if (search == without):
            without_index = i
            i += 1
            continue
        readData(search, case, n[i], T[i])
        i += 1
    colors = ["blue", "orange", "green", "red"]
    for i in range(len(n)):
        if (without_index != -1 and i == without_index):
            continue
        plt.scatter(n[i], T[i], label=searches[i], color = colors[i])
        reg(n[i], T[i], colors[i], i)

    plt.legend()
    # Сохранение файла
    plt.savefig(filename)
    plt.close()
```

```
def graphics():
    searches = ["LS", "BS", "TS", "IS"]
    cases = ["W", "F", "M", "L"]
    n = []
    T = []
    all_n = []
    all_T = []
    #Worst case
    plotGraphic(searches[0], cases[0], n, T, "PracGraphics\Worst case\LS.png")
    plotGraphic(searches[1], cases[0], n, T, "PracGraphics\Worst case\BS.png")
    plotGraphic(searches[2], cases[0], n, T, "PracGraphics\Worst case\TS.png")
    plotGraphic(searches[3], cases[0], n, T, "PracGraphics\Worst case\IS.png")
    #for all
    plotAllGraphics(searches, cases[0], all_n, all_T,"PracGraphics\Worst case\ALL.png")
    plotAllGraphics(searches, cases[0], all_n, all_T,"PracGraphics\Worst case\ALLwoLS.png",
without="LS")
    #Search first
    plotGraphic(searches[0], cases[1], n, T, "PracGraphics\First\LS.png")
    plotGraphic(searches[1], cases[1], n, T, "PracGraphics\First\BS.png")
    plotGraphic(searches[2], cases[1], n, T, "PracGraphics\First\TS.png")
    plotGraphic(searches[3], cases[1], n, T, "PracGraphics\First\IS.png")
    #for all
    plotAllGraphics(searches, cases[1], all_n, all_T,"PracGraphics\First\ALL.png")
    #Search middle
    plotGraphic(searches[0], cases[2], n, T, "PracGraphics\Mid\LS.png")
    plotGraphic(searches[1], cases[2], n, T, "PracGraphics\Mid\BS.png")
    plotGraphic(searches[2], cases[2], n, T, "PracGraphics\Mid\TS.png")
    plotGraphic(searches[3], cases[2], n, T, "PracGraphics\Mid\IS.png")
    #for all
    plotAllGraphics(searches, cases[2], all_n, all_T,"PracGraphics\Mid\ALL.png")
    plotAllGraphics(searches, cases[2], all_n, all_T,"PracGraphics\Mid\ALLwoLS.png",
without="LS")
```

```
    #Search last
    plotGraphic(searches[0], cases[3], n, T, "PracGraphics\Last\LS.png")
    plotGraphic(searches[1], cases[3], n, T, "PracGraphics\Last\BS.png")
    plotGraphic(searches[2], cases[3], n, T, "PracGraphics\Last\TS.png")
    plotGraphic(searches[3], cases[3], n, T, "PracGraphics\Last\IS.png")
    #for all
    plotAllGraphics(searches, cases[3], all_n, all_T,"PracGraphics\Last\ALL.png")
    plotAllGraphics(searches, cases[3], all_n, all_T,"PracGraphics\Last\ALLwoLS.png", without="LS")



graphics()
```