Министерство науки и высшего образования Российской Федерации
Российский химико-технологический университет имени Д. И. Менделеева

Факультет информационных технологий и управления
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

«Быстрая сортировка»

Выполнил студент группы КС-30 Колесников Артем Максимович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/AMKolesnikov_30_ALG

Приняли: аспирант кафедры ИКТ Пысин Максим Дмитриевич
аспирант кафедры ИКТ Краснов Дмитрий Олегович

Дата сдачи: 21.03.2022

Москва
2022

Содержание

Описание задачи

1. Необходимо реализовать метод быстрой сортировки.
2. Для реализованного метода сортировки необходимо провести серию тестов для всех значений N из списка (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000), при этом:
   - в каждом тесте необходимо по 20 раз генерировать вектор, состоящий из N элементов
   - каждый элемент массива заполняется случайным числом с плавающей запятой от -1 до 1
3. На основании статьи реализовать проверки негативных случаев и устроить на них серии тестов аналогичные второму пункту:
   - Отсортированный массив
   - Массив с одинаковыми элементами
   - Массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорного
   - Массив с максимальным количеством сравнений при детерминированном выборе опорного элемента
4. При работе сортировки подсчитать количество вызовов рекурсивной функции, и высоту рекурсивного стека. Построить график худшего, лучшего, и среднего случая для каждой серии тестов.
5. Для каждой серии тестов построить график худшего случая.
6. Подобрать такую константу C, чтобы график функции c * n * log(n) находился близко к графику худшего случая, если возможно построить такой график.
7. Проанализировать полученные графики и определить есть ли на них следы деградации метода относительно своей средней сложности.

Описание метода

Быстрая сортировка (англ. *quick sort*, сортировка Хоара) — один из самых известных и широко используемых алгоритмов сортировки. Среднее время работы $O(n \log n)$, что является асимптотически оптимальным временем работы для алгоритма, основанного на сравнении. Хотя время работы алгоритма для массива из n элементов в худшем случае может составить $O(n^2)$, на практике этот алгоритм является одним из самых быстрых.

Алгоритм состоит из трёх шагов:

- Выбрать элемент из массива. Назовём его опорным.
- Разбиение: перераспределение элементов в массиве таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.
- Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Выполнение задачи

Алгоритм быстрой сортировки был реализован на языке C++. Моя программа состоит из 8 функций ~ numGenerator, vectorGenerator, quSort, measurements, sameVector, midBadVector, determVector, main.

Функция «numGenerator» ~ возвращает случайное число с плавающей запятой в диапазоне от Min до Max.

```
double numGenerator(double min, double max) {
	return min + ((double)rand() / RAND_MAX * (max - min));
	 }
```

Функция «vectorGenerator» ~ возвращает вектор размера N заполненный случайными дробными числами от -1 до 1, полученными при помощи функции «numGenerator».

```
vector <double> vectorGenerator(int N) {

	const int MIN = -1;
	const int MAX = 1;

	vector <double> v;
	v.resize(N);

	for (int i = 0; i < N; i++)
	{
		v[i] = numGenerator(MIN, MAX);
	}

	return v;
	 }
```

Функция «quSort» ~ принимает вектор, который нужно отсортировать, индекс начала и конца вектора. Она сортирует полученный вектор методом быстрой сортировки и выполняет подсчет рекурсий.

```
void quSort(vector<double>& vector, int left, int right, int& callCount) {


    callCount++;

    //Указатели в начало и в конец массива
    int i = left;
    int j = right;

    //Центральный элемент массива
    auto mid = vector[(left + right) / 2];

    //Делим массив
    do {
        //Пробегаем элементы, ищем те, которые нужно перекинуть в другую часть
        //В левой части массива пропускаем(оставляем на месте) элементы, которые меньше
центрального
        while (vector[i] < mid) {
```

```
            i++;
        }
        //В правой части пропускаем элементы, которые больше центрального
        while (vector[j] > mid) {
            j--;
        }

        //Меняем элементы местами
        if (i <= j) {

            swap(vector[i], vector[j]);

            i++;
            j--;
        }
    } while (i < j);


    //Рекурсивные вызовы, если осталось, что сортировать
    if (left < j)
        quSort(vector, left, j, callCount);

    if (i < right)
        quSort(vector, i, right, callCount);

}
```

Функция «measurements» ~ возвращает число вызовов рекурсивной функции «quSort» и подсчитывает время сортировки.

```
int measurements(vector<double>& vec, double* time) {
    chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();

    int callCount = 0;

    quSort(vec, 0, vec.size() - 1, callCount);

    chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> milli_diff = end - start;

    *time = milli_diff.count();
    return callCount;
        }
```

Функция «sameVector» ~ возвращает вектор, заполненный одинаковыми элементами.

```
vector <double> sameVector(vector<double> vec) {
    for (int i = 0; i < vec.size(); i++)
        vec[i] = vec[0];
    return vec;
        }
```

Функция «midBadVector» ~ возвращает вектор с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

```
vector <double> midBadVector(vector<double> vec) {
    for (int i = 0; i < vec.size(); i++)
        swap(vec[i], vec[i / 2]);
```

```
    return vec;
        }
```

Функция «determVector» ~ возвращает вектор максимальным количеством сравнений при детерминированном выборе опорного элемента.

```
vector <double> determVector(int lenght) {
    int n = lenght;
    vector <double> determVec;
    determVec.reserve(lenght);

    vector<pair<double, int>> pairVec;
    pairVec.reserve(lenght);

    for (int i = 0; i < lenght; i++) {
        pairVec.emplace_back(0, i + 1);
    }

    int step = 1;
    const int left = 0;
    int decrease = 1;
    while (lenght > 0) {
        int right = pairVec.size() - (decrease++);
        pairVec[(left + right) / 2] = make_pair((n - step + 1), (pairVec[(left + right) /
2]).second);

        pair<double, int> pivot = pairVec[(left + right) / 2];
        swap(pairVec[(left + right) / 2], pairVec[right]);
        lenght--;
        step++;
    }

    sort(pairVec.begin(), pairVec.end(), comparator);

    for (int i = 0; i < n; i++) {
        determVec.emplace_back(pairVec[i].first);
    }
    return determVec;
        }
```

Функция «main» ~ состоит из цикла в котором происходит расчет минимального, максимального, среднего времени, затраченного на сортировку вектора, состоящего из N = { 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 } чисел, каждое из которых рассчитывается 20 раз. Там же и осуществляется проверка всевозможных негативных случаев (Отсортированный массив, Массив с одинаковыми элементами, Массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорног, Массив с максимальным количеством сравнений при детерминированном выборе опорного элемента). На выходе мы получаем файл «Recursions.txt» с расчетом рекурсий и файл «Time.txt» с расчетом времени работы нашего алгоритма.

```
int main()
{
    srand(time(0));
    setlocale(LC_ALL, "Rus");

    const int NUMS_COUNT = 8;
```

```
    const int NUM_TRIES = 20;
    int call[NUM_TRIES] = { 0 };
    double time[NUM_TRIES] = { 0 };
    double time_sort[NUMS_COUNT] = { 0 };
    double time_same[NUMS_COUNT] = { 0 };
    double time_midle[NUMS_COUNT] = { 0 };
    double time_determ[NUMS_COUNT] = { 0 };

    int N[] = { 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 };

    vector <double> sortVec;

    ofstream rout("Recursions.txt");
    ofstream tout("Time.txt");

    if (rout.is_open()) {
        for (int i = 0; i < NUMS_COUNT; i++) {
            double sum_call = 0;
            double sum_time = 0;
            rout << "Кол-во чисел: " << N[i] << endl << endl << "Кол-во вызовов: ";
            tout << "Количество чисел: " << N[i] << endl << "Время (мс): ";

            for (int j = 0; j < NUM_TRIES; j++) {
                sortVec = vectorGenerator(N[i]);
                call[j] = measurements(sortVec, &time[j]);
                rout << call[j] << "  ;  ";
                tout << time[j] << " ; ";

                sum_call += call[j];
                sum_time += time[j];
            }

            rout << endl;
            tout << endl;
            double min_r = call[0];
            double max_r = call[0];

            double min_t = time[0];
            double max_t = time[0];

            for (int j = 0; j < NUM_TRIES; j++) {
                if (max_r < call[j]) max_r = call[j];
                if (min_r > call[j]) min_r = call[j];

                if (max_t < time[j]) max_t = time[j];
                if (min_t > time[j]) min_t = time[j];
            }

            rout << endl << "Минимальное: " << min_r << endl << "Максимальное: " << max_r
<< endl << "Среднее: " << sum_call / NUM_TRIES << endl << endl;
            tout << endl << "Минимальное: " << min_t << endl << "Максимальное: " << max_t
<< endl << "Среднее: " << sum_time / NUM_TRIES << endl << endl;

            rout << "Отсортированный массив: ";
            tout << "Отсортированный массив: ";

            rout << measurements(sortVec, &time_sort[i]) << endl;
            tout << time_sort[i] << endl;

            rout << "Массив с одинаковыми элементами: ";
            tout << "Массив с одинаковыми элементами: ";

            vector <double> sameVec = sameVector(sortVec);
            rout << measurements(sameVec, &time_same[i]) << endl;
            tout << time_same[i] << endl;
```

```
            rout << "Массив с максимальным количеством сравнений при выборе среднего
элемента в качестве опорного: ";
            tout << "Массив с максимальным количеством сравнений при выборе среднего
элемента в качестве опорного: ";

            vector <double> midVec = midBadVector(sortVec);
            rout << measurements(midVec, &time_midle[i]) << endl;
            tout << time_midle[i] << endl;

            rout << "Массив с максимальным количеством сравнений при детерминированном
выборе опорного элемента: ";
            tout << "Массив с максимальным количеством сравнений при детерминированном
выборе опорного элемента: ";

            vector <double> determVec = determVector(N[i]);
            rout << measurements(determVec, &time_determ[i]) << endl;
            tout << time_determ[i] << endl;

            tout << endl << endl;
            rout << endl << endl;


        }

        rout.close();
        tout.close();
    }

}
```

Время (мс)				
N	**мин**	**сред**	**макс**	**c * n * log(n)**
1000	0,14	0,15	0,21	0,12
2000	0,30	0,32	0,36	0,25
4000	0,83	0,88	0,93	0,55
8000	1,33	1,41	1,89	1,20
16000	2,37	2,96	3,19	2,59
32000	5,01	5,46	6,82	5,55
64000	13,36	13,74	14,20	11,84
128000	22,23	22,58	22,91	25,17

Таблица 1 – Максимальное, минимальное, среднее время (в миллисекундах) сортировки векторов, состоящих из N элементов.

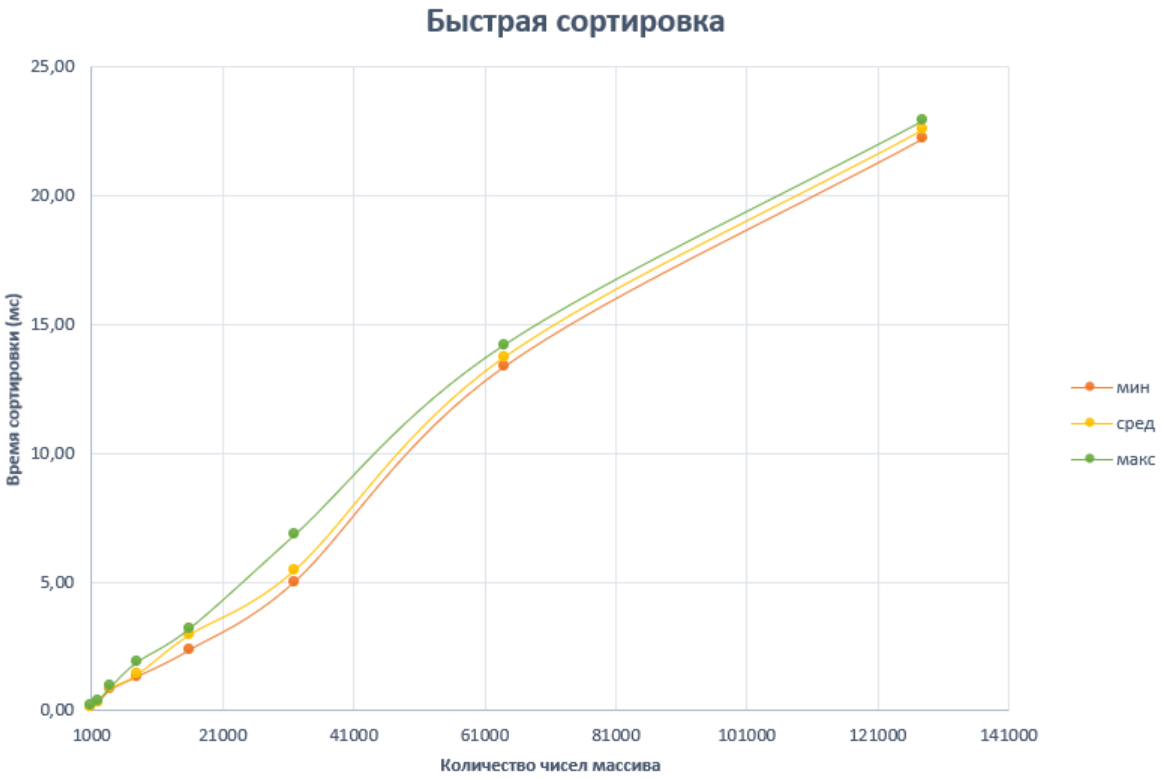

Рисунок 1 – График лучшего (минимальное время для каждого N), худшего (максимальное время для каждого N) и среднего (среднее время для каждого N) случая.

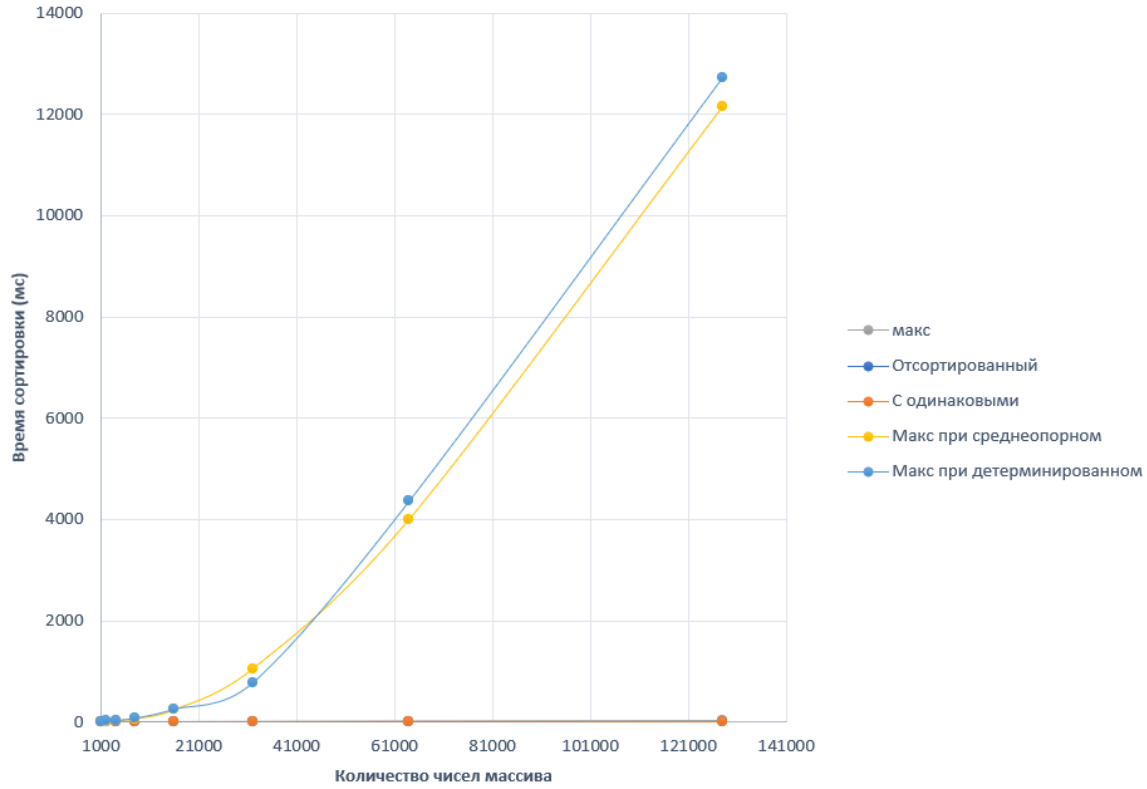

Рисунок 2 – График времени негативных случаев

Я подобрал константу C так, чтобы график функции c * n * log(n) находился близко к графику худшего случая времени.

$$C = 3{,}85 * 10^{-5}$$

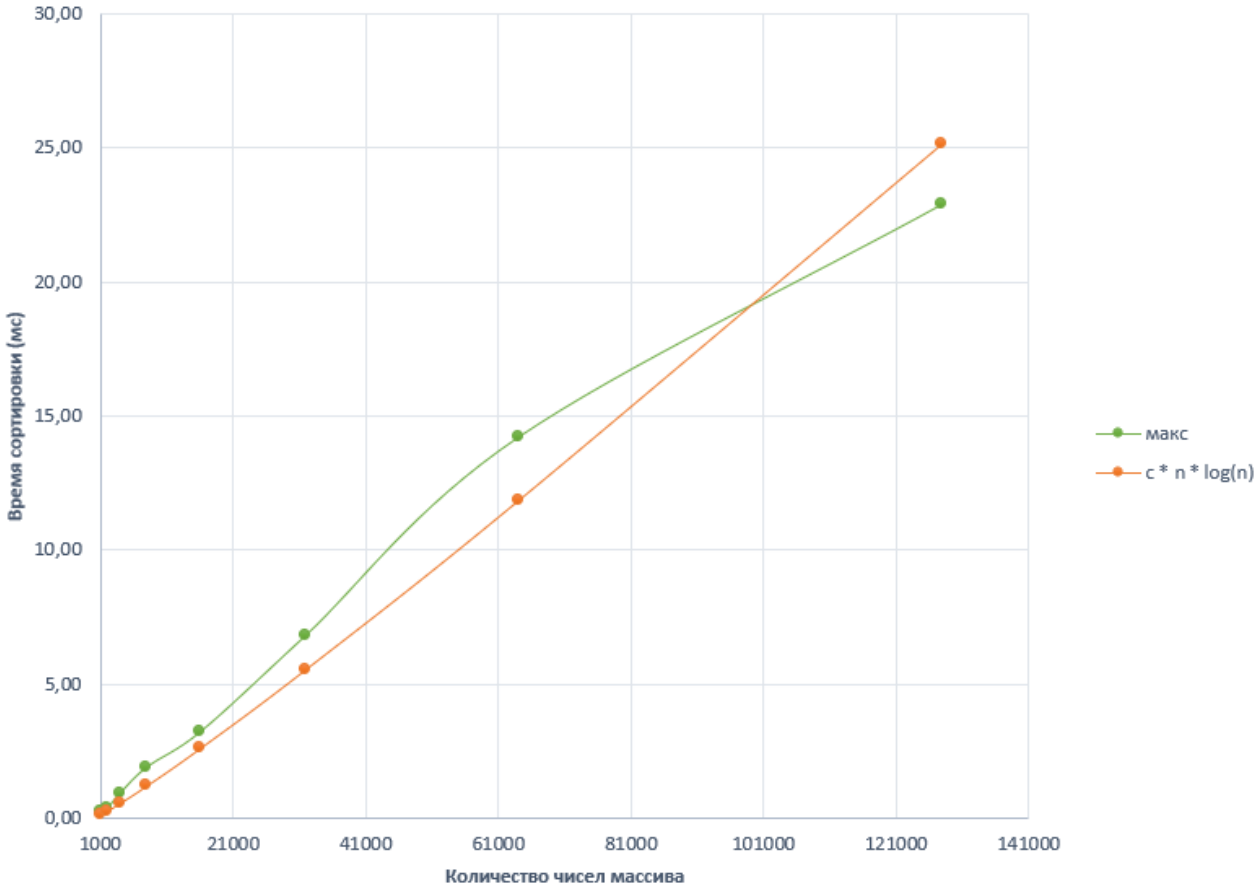

Рисунок 3 – График худшего случая времени и c*n*log(n)

Число рекурсий				
N	мин	сред	макс	c * n * log(n)
1000	800,00	814,40	838,00	480,00
2000	1595,00	1632,10	1670,00	1056,33
4000	3220,00	3257,10	3303,00	2305,32
8000	6461,00	6533,80	6589,00	4995,96
16000	12886,00	13090,90	13196,00	10762,55
32000	26055,00	26157,70	26259,00	23066,37
64000	52171,00	52276,60	52377,00	49215,28
128000	104380,00	104625,00	104904,00	104595,66

Таблица 2 – Максимальное, минимальное, среднее число рекурсий сортировки векторов, состоящих из N элементов.

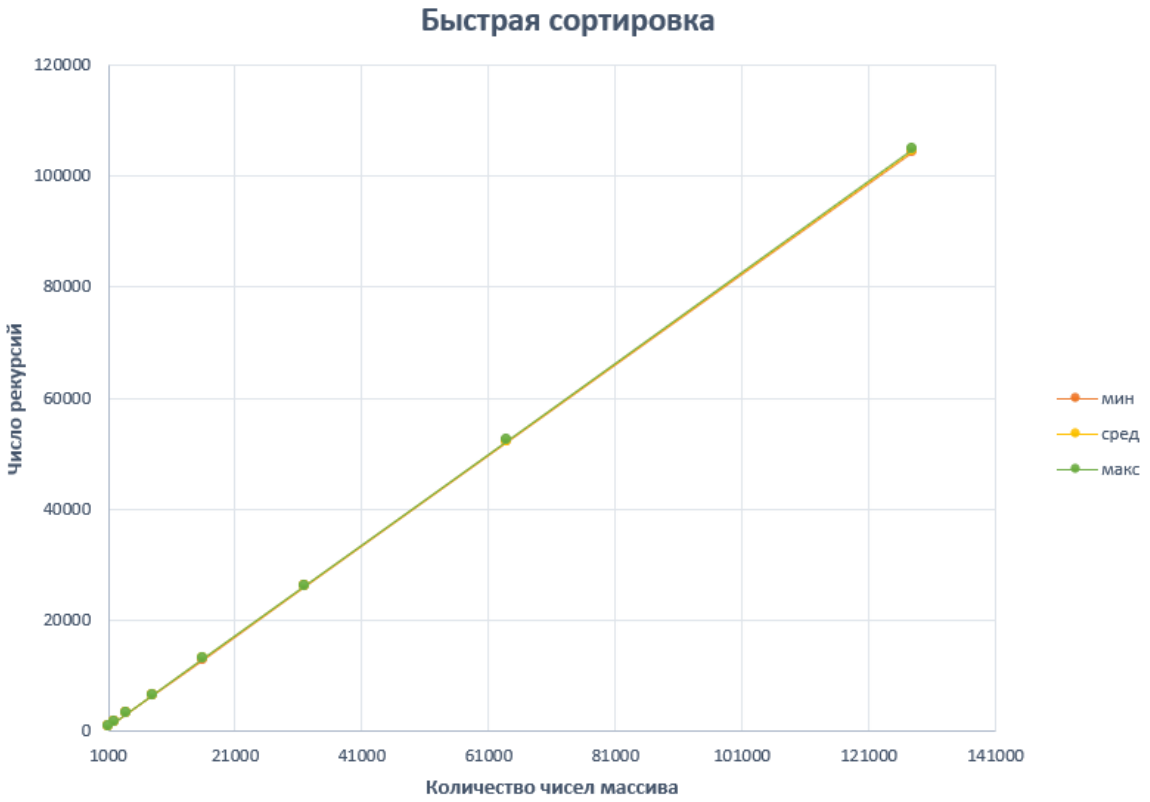

Рисунок 4 – График лучшего (минимальное число рекурсий для каждого N), худшего (максимальное число рекурсий для каждого N) и среднего (среднее число рекурсий для каждого N) случая.

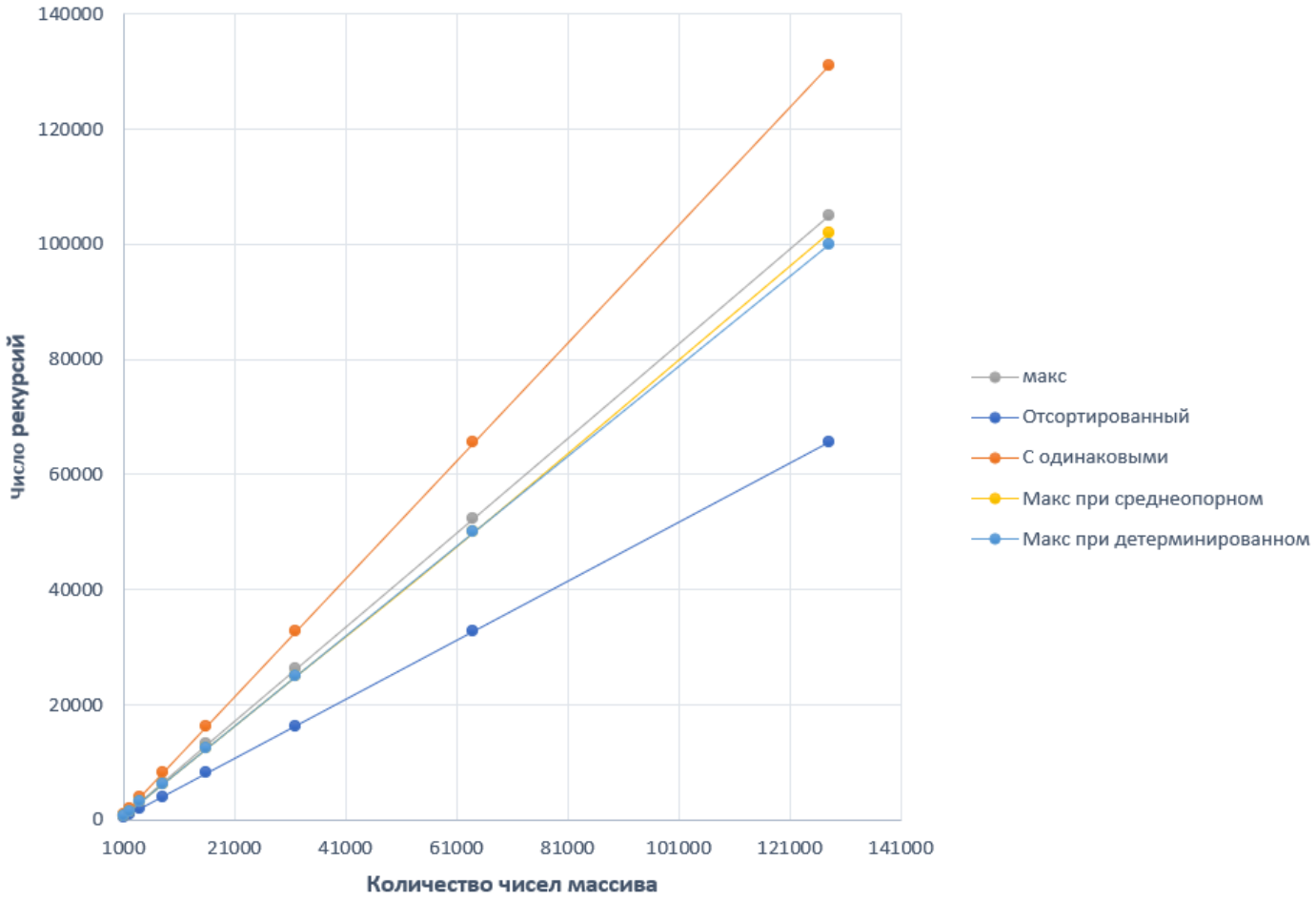

Рисунок 5 – График числа рекурсий негативных случаев

.

Я подобрал константу C так, чтобы график функции c * n * log(n) находился близко к графику худшего случая числа рекурсий.

$$C = 1{,}6 * 10^{-1}$$

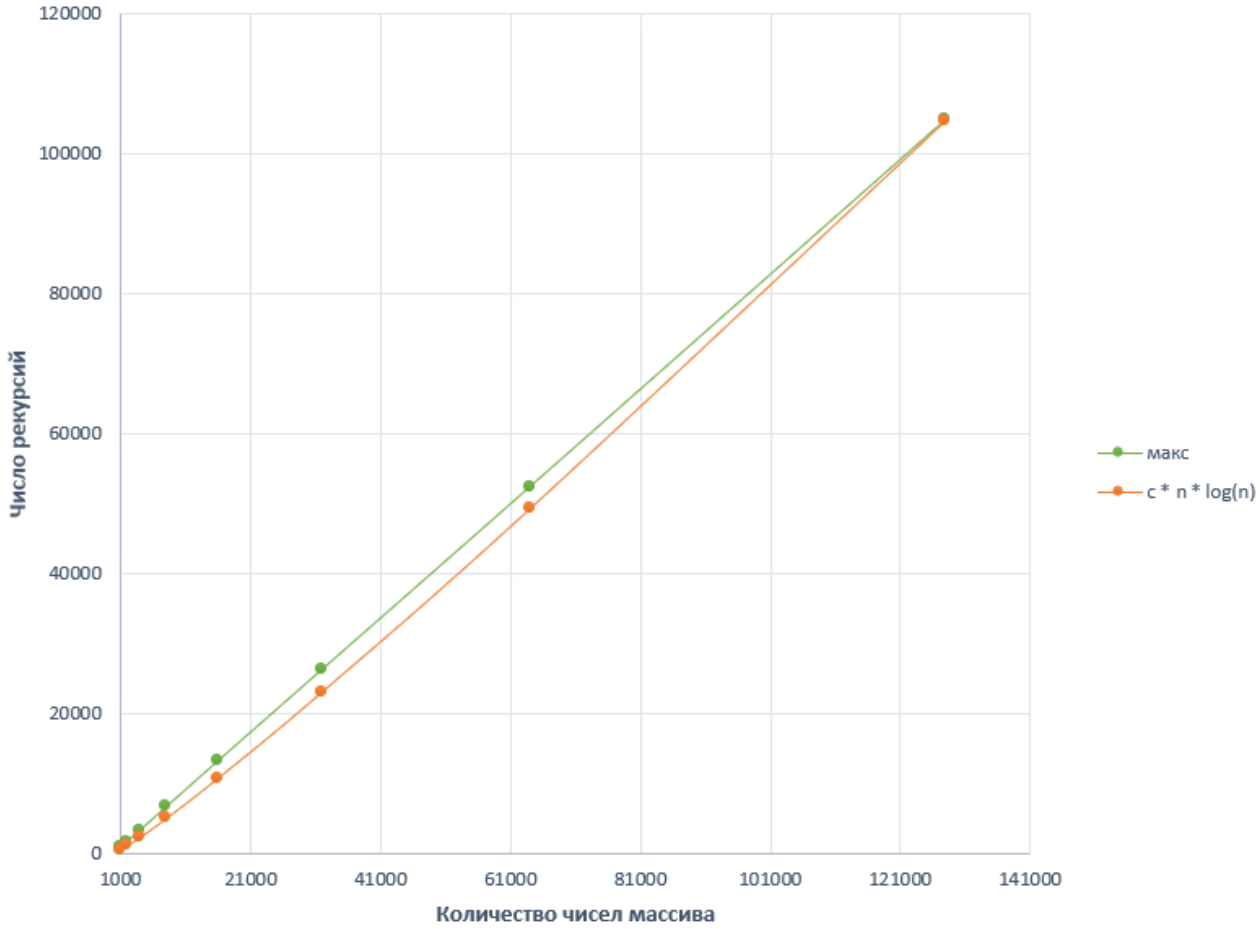

Рисунок 6 – График худшего случая рекурсий и c*n*log(n)

Заключение

Я убедился на практике, что быстрая сортировка по-настоящему быстрый алгоритм сортировки. К тому же он достаточно прост в реализации, наверное, именно поэтому он пользуется такой популярностью. Однако и у него есть свои недостатки: сильно деградирует по скорости (до $O(n^2)$) в худшем или близком к нему случае, что может случиться при неудачных входных данных. Алгоритм является неустойчивым, то есть он меняет порядок одинаковых ключей. Из-за большого числа рекурсий достаточно часто появляется ошибка «переполнение стека».