ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2
БЫСТРАЯ СОРТИРОВКА

Оглавление

Описание задачи.

Необходимо изучить и реализовать быструю сортировку на серии тестов для всех N значений из списка (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000), где в каждом тесте необходимо по 20 раз генерировать вектор из N элементов. Также посмотреть её работу на негативных случаях:

- 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

При работе сортировки подсчитать количество вызовов рекурсивной функции, и высоту рекурсивного стека. Построить график худшего, лучшего, и среднего случая для каждой серии тестов. Для каждой серии тестов построить график худшего случая. Подобрать такую константу c, что бы график функции c * n * log(n) находился близко к графику худшего случая, если возможно построить такой график. Проанализировать полученные графики и определить есть ли на них следы деградации метода относительно своей средней сложности.

Описание метода/модели.

Быстрая сортировка (англ. quick sort, сортировка Хоара) — один из самых известных и широко используемых алгоритмов сортировки. Среднее время работы O(nlogn), 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 Θ(n2), на практике этот алгоритм является одним из самых быстрых.

Алгоритм

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй".

- Массив a[l…r]a[l…r] типа TT разбивается на два (возможно пустых) подмассива a[l…q]a[l…q] и a[q+1…r]a[q+1…r], таких, что каждый элемент a[l…q]a[l…q] меньше или равен a[q]a[q], который в свою очередь, не превышает любой элемент подмассива a[q+1…r]a[q+1…r]. Индекс вычисляется в ходе процедуры разбиения.
- Подмассивы a[l…q]a[l…q] и a[q+1…r]a[q+1…r] сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.
- Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив a[l…r]a[l…r] оказывается отсортированным.

Псевдокод

```
void quicksort(a: T[n], int l, int r)
   if l < r
      int q = partition(a, l, r)
      quicksort(a, l, q)
      quicksort(a, q + 1, r)
```

Разбиение массива

Основной шаг алгоритма сортировки — процедура partitionpartition, которая переставляет элементы массива a[l…r]a[l…r] типа TT нужным образом. Разбиение осуществляется с использованием следующей стратегии. Прежде всего, в качестве разделяющего элемента произвольно выбирается элемент a[(l+r)/2]a[(l+r)/2]. Далее начинается просмотр с левого конца массива, который продолжается до тех пор, пока не будет найден элемент, превосходящий по

значению разделяющий элемент, затем выполняется просмотр, начиная с правого конца массива, который продолжается до тех пор, пока не отыскивается элемент, который по значению меньше разделяющего. Оба элемента, на которых просмотр был прерван, очевидно, находятся не на своих местах в разделенном массиве, и потому они меняются местами. Так продолжаем дальше, пока не убедимся в том, что слева от левого указателя не осталось ни одного элемента, который был бы больше по значению разделяющего, и ни одного элемента справа от правого указателя, которые были бы меньше по значению разделяющего элемента.

Переменная vv сохраняет значение разделяющего элемента a[(l+r)/2]a[(l+r)/2], а ii и jj представляет собой, соответственно, указатели левого и правого просмотра. Цикл разделения увеличивает значение ii и уменьшает значение jj на 11, причем условие, что ни один элемент слева от ii не больше vv и ни один элемент справа от jj не меньше vv, не нарушается. Как только значения указателей пересекаются, процедура разбиения завершается.

```
int partition(a: T[n], int l, int r)
   T v = a[(l + r) / 2]
   int i = l
   int j = r
   while (i ⩽ j)
      while (a[i] < v)
         i++
      while (a[j] > v)
         j--
      if (i ⩾ j)
         break
      swap(a[i++], a[j--])
   return j
```

Худшее время работы

Предположим, что мы разбиваем массив так, что одна часть содержит n−1 элементов, а вторая — 11. Поскольку процедура разбиения занимает время Θ(n), для времени работы T(n) получаем соотношение:

$$T(n)=T(n-1)+\Theta(n)=\sum_{k=1}^{n}\Theta(k)=\Theta\left(\sum_{k=1}^{n}k\right)=\Theta(n^2).$$

В каких случаях возможно получить такую ситуацию:

- Если массив изначально отсортирован.
- Если массив содержит одинаковые элементы.
- Есть способ построить массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорного
- Есть способ построить массив с максимальным количеством сравнений при детерминированном выборе опорного элемента

Преимущества

- Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
- Алгоритм очень короткий: запомнив основные моменты, его легко написать «из головы»
- Требует лишь $O(1)$ дополнительной памяти для своей работы (неулучшенный рекурсивный алгоритм - в худшем случае $O(n)$ памяти).
- Хорошо сочетается с механизмами кэширования и виртуальной памяти.
- Допускает естественное распараллеливание (сортировка выделенных подмассивов в параллельно выполняющихся подпроцессах).
- Работает на связных списках и других структурах с последовательным доступом, допускающих эффективный проход как от начала к концу, так и от конца к началу.

Недостатки

- Сильно деградирует по скорости (до $O(n\text{^}2$)) в худшем или близком к нему случае, что может случиться при неудачных входных данных.
- Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека, так как в худшем случае ей может потребоваться сделать $O(n)$ вложенных рекурсивных вызовов.
- Неустойчив.

Выполнение задачи.

Программа для получения тестовых данных была написана на C++. Графики нарисованы в Excel.

Код:

```
#include <iostream>
#include <string>
#include <ctime>
#include <vector>
#include <chrono>
#include <random>
#include <utility>
#include <fstream>
#include <unordered_map>
#include <map>
using namespace std;
const int VALUES_COUNT = 8;
int TESTS_LIMIT = 20;
const int TEST_VALUES[VALUES_COUNT]{ 1000, 2000, 4000, 8000, 16000, 32000,64000,128000 };

vector<double> arrayWithMaxmMiddleSelection(int i);
vector<double> arrayWithMaxDeterministicSelection(int i, int& recursion_stack, int&
recursion_stack_height, int& recursion_stack_height_max);
int partition(vector<double>& A, int low, int high);
void quickSort(vector<double>& vector_values, int low, int high, int& recursion_stack, int&
recursion_stack_height, int& recursion_stack_height_max);

/// <summary>
/// Програма для тестирования быстрой сортировки
/// </summary>
/// <returns></returns>
int main()
{
    setlocale(LC_ALL, "Russian");
    mt19937 engine(time(0));
    uniform_real_distribution<double> gen(-1.0, 1.0);
    std::ofstream out;
    std::ofstream outr;
    out.open("TestingLog.txt");
    outr.open("Testing2Log.txt");
    if (out.is_open() && outr.is_open())
    {
        for (int array_i = 0; array_i < 5; array_i++) {
            out << "K-" << array_i << endl;
            //cout << "K-" << array_i << endl;
            outr << "K-" << array_i << endl;
            for (int i = 0; i < VALUES_COUNT; i++) {
                /*cout << "N = " << TEST_VALUES[i] << endl;*/
                out << "N = " << TEST_VALUES[i]<<endl;
                outr << "N = " << TEST_VALUES[i] << endl;
                for (int j = 0; j < TESTS_LIMIT; j++) {
                    vector<double> vector_values(TEST_VALUES[i]);
                    int recursion_stack_height = 0;
                    int recursion_stack = 0;
                    int recursion_stack_height_max=0;
                    /*cout << "\nДо Сортировки: " << endl;*/
                    if (array_i == 0 || array_i == 1) {
                        //генерация массива с рандомными значениями от -1 до 1
                        for (int k = 0; k < TEST_VALUES[i]; k++) {
                            vector_values[k] = gen(engine);
                        }
                        //Отсортированный массив
                        if (array_i == 1) {
                            std::sort(vector_values.begin(), vector_values.end());
                        }
                    }
                    //Массив с одинаковыми элементами
```

```
                    else if (array_i == 2) {
                        double rand = gen(engine);
                        for (int k = 0; k < TEST_VALUES[i]; k++) {
                            vector_values[k] = rand;
                        }
                    }
                    //массив с максимальным количеством сравнений при выборе среднего элемента
в качестве опорного
                    else if (array_i == 3 && j < 1 ) {
                        vector_values = arrayWithMaxmMiddleSelection(i);
                    }
                    //массив с максимальным количеством сравнений при детерминированном выборе
опорного элемента
                    else if (array_i == 4 && j<1) {
                        vector_values = arrayWithMaxDeterministicSelection(i, recursion_stack,
recursion_stack_height, recursion_stack_height_max);
                    }

                    //обнуляю счётчики после создания массива с максимальным кол-вом сравнений
при детерминированном выборе опорного элемента
                    recursion_stack_height = 0;
                    recursion_stack = 0;
                    recursion_stack_height_max = 0;

                    //замерка времени и сортировка
                    chrono::high_resolution_clock::time_point start =
chrono::high_resolution_clock::now();
                    quickSort(vector_values, 0, vector_values.size() - 1, recursion_stack,
recursion_stack_height, recursion_stack_height_max);
                    chrono::high_resolution_clock::time_point end =
chrono::high_resolution_clock::now();
                    chrono::duration<double> milli_diff = end - start;
                    out << /*sec_diff.count()*/ "recursion_stack: "<< recursion_stack /*<<
endl*/;
                    out << /*sec_diff.count()*/ " ;recursion_stack_height: " <<
recursion_stack_height_max << endl;
                    outr << milli_diff.count() << endl;

                }
            }
        }
    }
    out.close();
    outr.close();
}
/// <summary>
/// Создание массива с максимальным количеством сравнений при выборе среднего элемента в
качестве опорного
/// </summary>
/// <param name="i">Номер теста</param>
/// <returns>Вектор для тестов</returns>
vector<double> arrayWithMaxmMiddleSelection(int i) {
    vector<double> final_vector(TEST_VALUES[i]);
    //Заполнение массива a длины n элементами от 1 до n,
    for (int k = 0; k < TEST_VALUES[i]; k++) {
        final_vector[k] = k + 1;
    }
    for (int z = 0; z < TEST_VALUES[i]; z++) {
        std::swap(final_vector[z], final_vector[z / 2]);
    }
    return final_vector;
}
/// <summary>
/// Создание массива с максимальным количеством сравнений при детерминированном выборе
опорного элемента
/// </summary>
/// <param name="i">Номер теста</param>
/// <param name="recursion_stack">Подсчет всех вызовов рекурсии</param>
/// <param name="recursion_stack_height">Счетчик высоты рекурсивного стека</param>
```

```
/// <param name="recursion_stack_height_max">Максимальная высота рекурсивного стека</param>
/// <returns>Вектор для тестов</returns>
vector<double> arrayWithMaxDeterministicSelection(int i, int& recursion_stack, int&
recursion_stack_height, int& recursion_stack_height_max) {
    vector<double> final_vector(TEST_VALUES[i]);
    //определяем первый детеменированый опорный элемент
    int p = (0 + TEST_VALUES[i] - 1) / 2;
    //вектор для хранения значений
    vector<double> val(TEST_VALUES[i]);
    //вектор для хранения ключей
    vector<double> key(TEST_VALUES[i]);
    //инициализация этой пары)
    for (int k = 0; k < TEST_VALUES[i]; k++) {
        key[k] = k + 1;
        val[k] = 0;
    }

    for (int k = 0; k < TEST_VALUES[i]; k++) {
        //снова определяем детерминированный опорный элемент по циклу
        p = (TEST_VALUES[i] - 1 - k) / 2;
        //сохраняем значение измененного опорного элемента и сам опорный элемент
        double val_temp = TEST_VALUES[i] - (k + 1) + 1;
        val[p] = val_temp;
        //сортируем массив по значениям val
        quickSort(val, 0, val.size() - 1, recursion_stack, recursion_stack_height,
recursion_stack_height_max);
        auto it = std::find(val.begin(), val.end(), val_temp);
        int temp_index = it - val.begin();
        //меняем положение в key по индексам сортированного val
        std::swap(key[p], key[temp_index]);
    }
    //сохраняем значения индексов массива ключей
    vector<double> temp_vector = key;
    /*cout << "------------final------------" << endl;*/
    quickSort(key, 0, val.size() - 1, recursion_stack, recursion_stack_height,
recursion_stack_height_max);
    //переставляем значения в val по сортированному порядку в key
    for (int z = 0; z < TEST_VALUES[i]; z++) {
        auto it = std::find(temp_vector.begin(), temp_vector.end(), key[z]);
        int index = it - temp_vector.begin();
        final_vector[z] = val[index];
        /*cout << final_vector[z] << ",";*/
    }
   /* cout << endl;
    cout << endl;*/
    return final_vector;
}
/// <summary>
/// Разбиение вектора во время быстрой сортировки
/// </summary>
/// <param name="A">Сортируемый вектор</param>
/// <param name="low">Левая граница</param>
/// <param name="high">Правая граница</param>
/// <returns>Новая правая граница</returns>
int partition(vector<double>& A, int low, int high) {
    double v = A[(low + high) / 2];
    int i = low;
    int j = high;
    while (i <= j) {
        while (A[i] < v) {
            i++;
        }
        while (A[j] > v) {
            j--;
        }
        if (i >= j) {
            break;
        }
        std::swap(A[i++], A[j--]);
```

```
    }
    return j;
}
/// <summary>
/// Быстрая сортировка
/// </summary>
/// <param name="A">Сортируемый вектор</param>
/// <param name="low">Левая граница</param>
/// <param name="high">Правая граница</param>
/// <param name="recursion_stack">Подсчет всех вызовов рекурсии</param>
/// <param name="recursion_stack_height">Счетчик высоты рекурсивного стека</param>
/// <param name="recursion_stack_height_max">Максимальная высота рекурсивного стека</param>
void quickSort(vector<double>& A, int low, int high, int& recursion_stack, int&
recursion_stack_height, int& recursion_stack_height_max) {
    recursion_stack++;
    recursion_stack_height++;
    if (recursion_stack_height_max < recursion_stack_height) recursion_stack_height_max =
recursion_stack_height;
    if (low < high) {

        int p = partition(A, low, high);
        quickSort(A, low, p,recursion_stack, recursion_stack_height,
recursion_stack_height_max);
        quickSort(A, p+1, high, recursion_stack, recursion_stack_height,
recursion_stack_height_max);
    }
    recursion_stack_height--;

}
```

Реализация:

1)Количество значений записано в массиве TEST_VALUES, а количество повторений тестов в TESTS_LIMIT

2) Рандомно сгенерированные, с одинаковыми числами и отсортированные вектора генерируются в функции main. Вектора с максимальным количеством сравнений при выборе среднего элемента в качестве опорного и с максимальным количеством сравнений при детерминированном выборе опорного элемента генерируются в вынесенных отдельно функциях по следующим правилам:

1. Вектор с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

Заполняется сначала массив a длины n элементами от 1 до n, затем применяется следующий алгоритм (нумерация с нуля):

```
void antiQsort(a: T[n])
   for i = 0 to n – 1
      swap(a[i], a[i / 2])
```

2. Вектор с максимальным количеством сравнений при детерминированном выборе опорного элемента.

Создадим массив a длины n, заполненный элементами типа pair. Такой элемент хранит пару значений (val,key), где val — элемент массива, a key — индекс. Изначально a[i] элемент имеет вид (0,i).

Далее, запустим для данного массива алгоритм быстрой сортировки. Сравниваем два элемента типа pair по их значениям val. На каждом шаге будем выполнять следующие действия: при обращении к i-ому элементу в качестве опорного на шаге под номером k, присвоим val=n−k+1 для элемента a[i]. Затем выполним шаг сортировки. После завершения работы алгоритма быстрой сортировки, дополнительно отсортируем получившиеся элементы pair по значениям key. Искомым будет являться массив элементов val в соответствующей последовательности.

3)Там же, внутри обоих циклов (прохождения по всем тестам и повторений этого теста) создается вектор vector_values, который будет хранить значения.

3)Поочередно проверяются рандомно сгенерированный массив, отсортированный, с одинаковыми элементами, с максимальным количеством сравнений при выборе среднего элемента в качестве опорного, с максимальным количеством сравнений при детерминированном выборе опорного элемента.

3)Далее в той же функции main замеряется количество вызовов рекурсии функции и высота рекурсивного стека при сортировке, время старта до сортировки и после, всё это выводится в два файла(TestingLog.txt, который хранит информацию о рекурсиях и TestingLog2.txt, который хранит информацию о времени работы). Сортируется массив в отдельной функции void quickSort(), сортирующей массив быстрой сортировкой

4)В Excel заношу данные из файла и строю сначала графики худшего, лучшего и среднего случая, выбирая в качестве области значений соответственно все худшие, лучшие средние значения тестов.

5)Строю график c * n * log(n), который находится близко к графику худшего случая и анализирую полученные графики.

Результаты:

1)Для более наглядной демонстрации протестировал сам процесс сортировки на меньших значениях TEST_VALUES для всех векторов.

```
До Сортировки:
-0.420326,-0.729867,-0.604725,-0.618152,

После Сортировки:
-0.729867,-0.618152,-0.604725,-0.420326,

До Сортировки:
0.00285318,0.0168481,-0.683237,0.481136,-0.325637,0.189333,-0.333672,0.0704661,

После Сортировки:
-0.683237,-0.333672,-0.325637,0.00285318,0.0168481,0.0704661,0.189333,0.481136,

До Сортировки:
0.318228,-0.301437,0.887874,-0.0929519,0.467648,-0.532498,0.616842,0.385285,

После Сортировки:
-0.532498,-0.301437,-0.0929519,0.318228,0.385285,0.467648,0.616842,0.887874,
```

Рис. 1. Демонстрация работы сортировки на векторе с рандомно сгенерированными значениями

```
До Сортировки:
-0.923216,-0.872575,-0.752436,-0.683459,-0.609985,-0.370356,0.0211589,0.11216,0.116035,0.1374,0.368504,0.421282,0.750371,0.919805,0.924212,0.993462,

После Сортировки:
-0.923216,-0.872575,-0.752436,-0.683459,-0.609985,-0.370356,0.0211589,0.11216,0.116035,0.1374,0.368504,0.421282,0.750371,0.919805,0.924212,0.993462,
N = 32
```

Рис. 2. Демонстрация работы сортировки на отсортированном векторе.

```
N = 8

До Сортировки:
-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,

После Сортировки:
-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,-0.113043,
```

Рис. 3. Демонстрация работы сортировки на векторе с одинаковыми значениями.

```
N = 4

До Сортировки:
2,4,1,3,

После Сортировки:
1,2,3,4,
N = 8

До Сортировки:
2,4,6,8,1,5,3,7,
```

Рис. 4. Демонстрация работы сортировки на векторе с максимальным количеством сравнений при детерминированном выборе опорного элемента и с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

2)Результаты серии тестов сгенерированного из рандомных чисел вектора для всех N на графике.

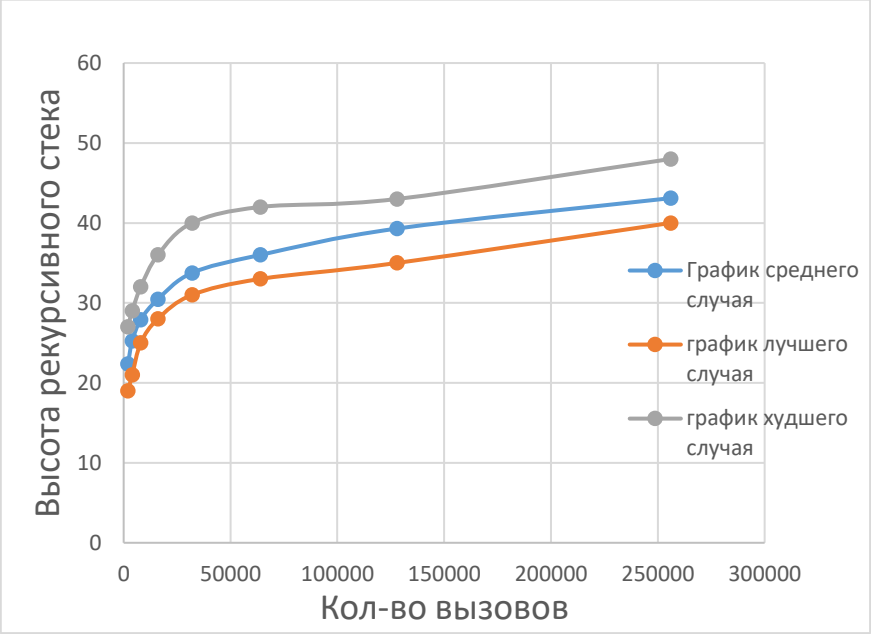

Рис.5 График лучшего, среднего и худшего случаев при тестировании вектора, заполненного рандомными числами.

3)Результаты серии тестов отсортированного вектора для всех N на графике.

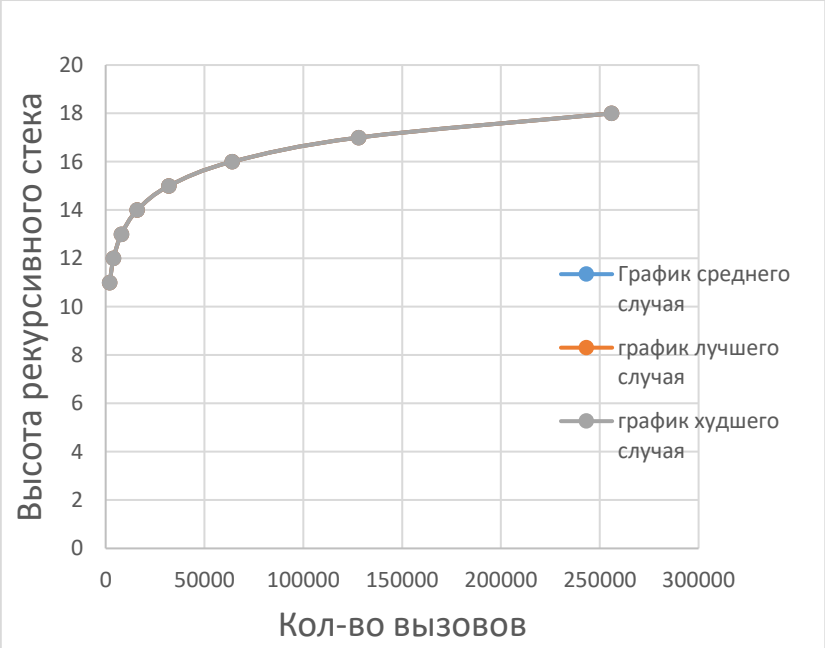

Рис.6 График лучшего, среднего и худшего случаев при тестировании отсортированного вектора.

4)Результаты серии тестов вектора одинаковых чисел для всех N на графике.

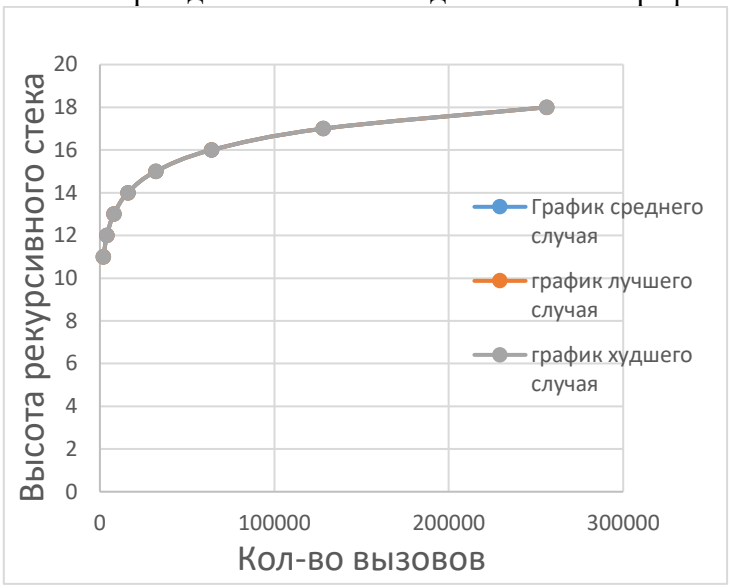

Рис.7 График лучшего, среднего и худшего случаев при тестировании вектора одинаковых чисел.

5)Результаты серии тестов вектора с максимальным количеством сравнений при выборе среднего элемента в качестве опорного для всех N на графике.

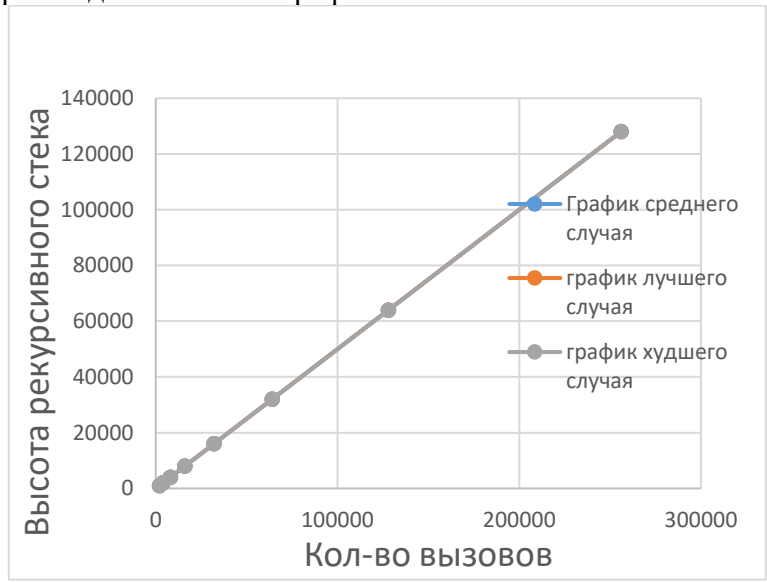

Рис.8 График лучшего, среднего и худшего случаев при тестировании вектора с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

6)Результаты серии тестов вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента.

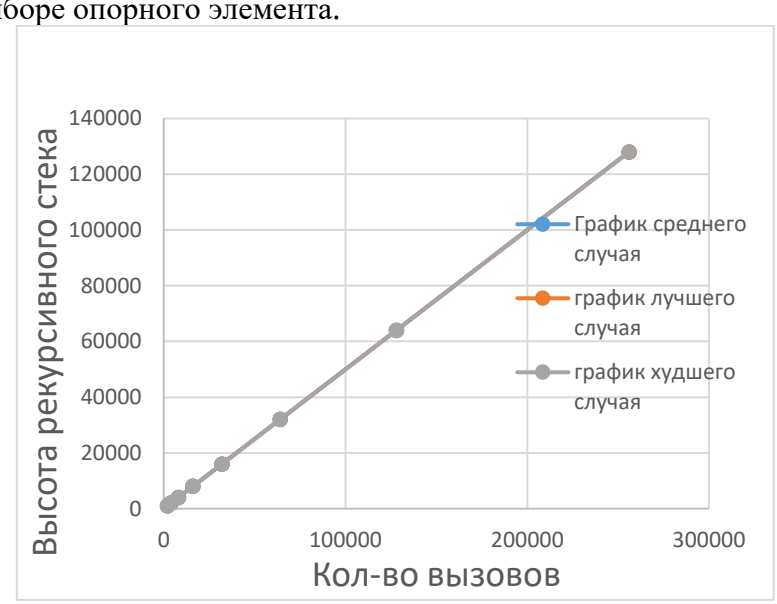

Рис.9 График лучшего, среднего и худшего случаев при тестировании вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента

7)Подобрал такую c, чтобы график функции c*n*log(n) находился близко к графику худшего случая для каждой серии тестов.

- Для первого 0.0004*n*log(n)

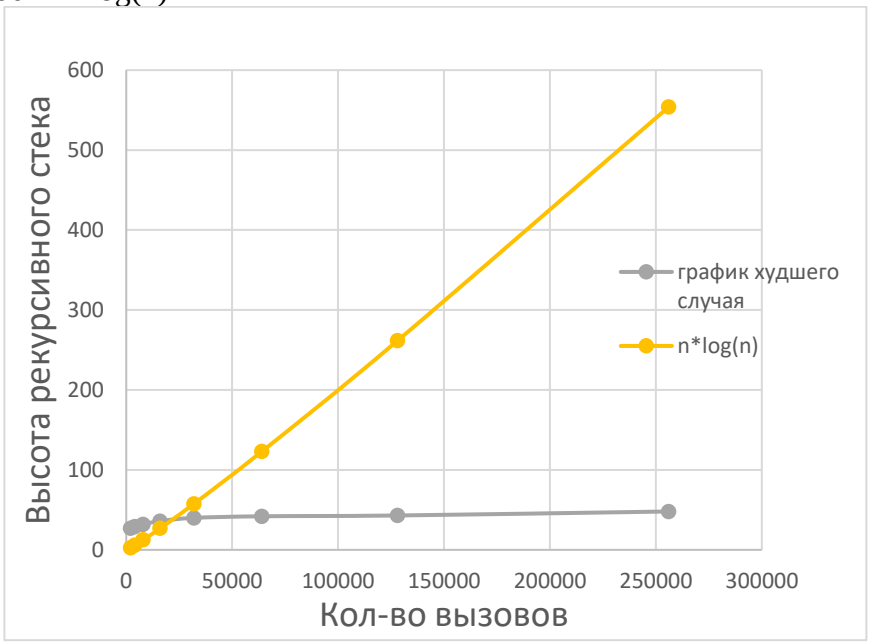

Рис.10 График c*n*log(n) и худшего случаев при тестировании вектора, заполненного случайными элементами.

- Для отсортированного 0.0004*n*log(n)

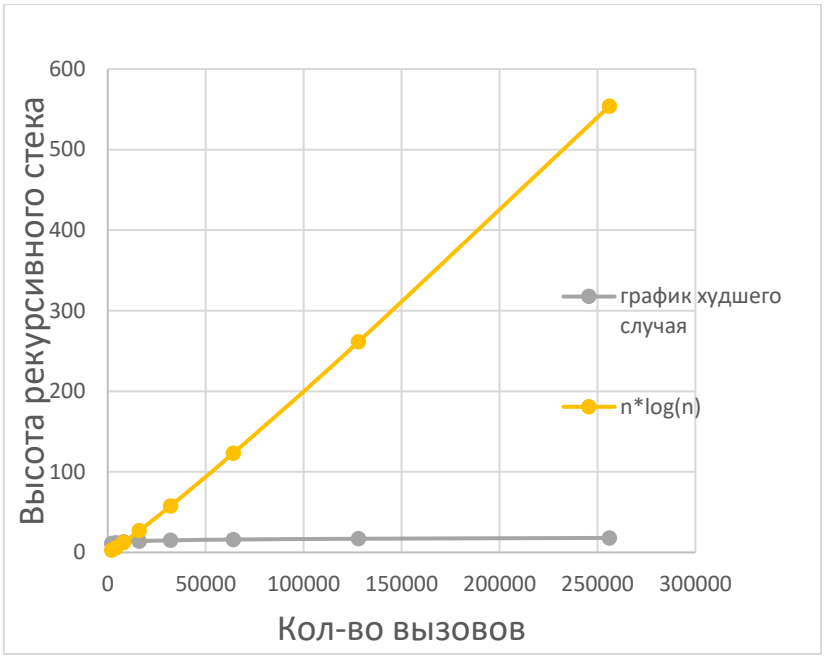

Рис.11 График c*n*log(n) и худшего случаев при тестировании отсортированного вектора.

- Для массива с одинаковыми элементами 0.0004*n*log(n)

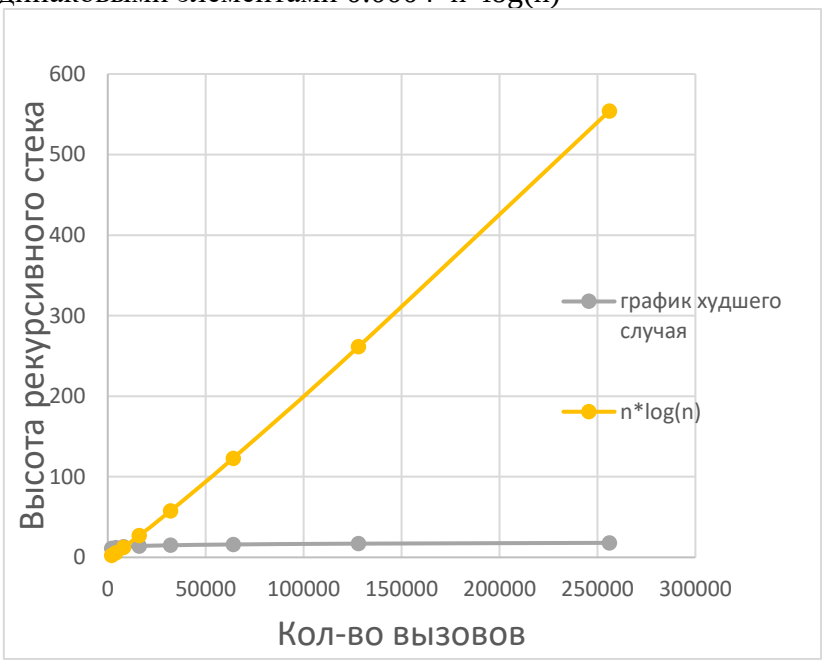

Рис.12 График c*n*log(n) и худшего случаев при тестировании вектора, заполненного одинаковыми элементами.

- Для вектора с максимальным количеством сравнений при выборе среднего элемента в качестве опорного для всех N 0.009*n*log(n)

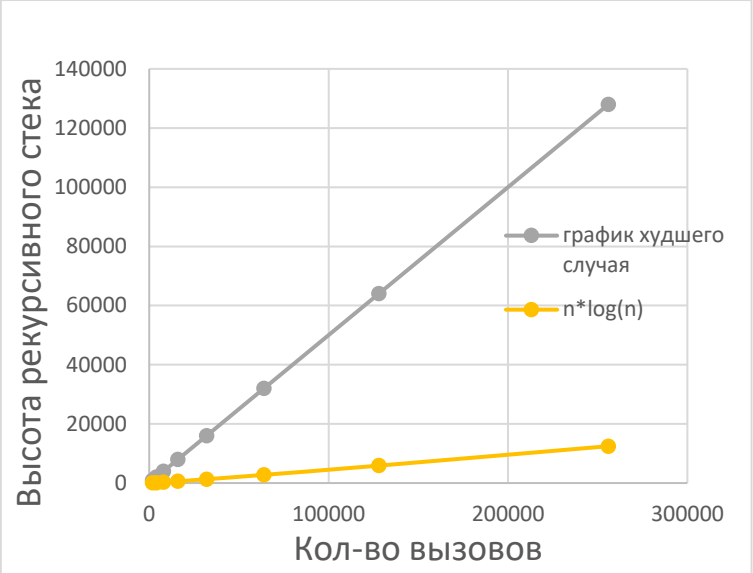

Рис.13 График c*n*log(n) и худшего случаев при тестировании вектора с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

- Для вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента 0.009*n*log(n)

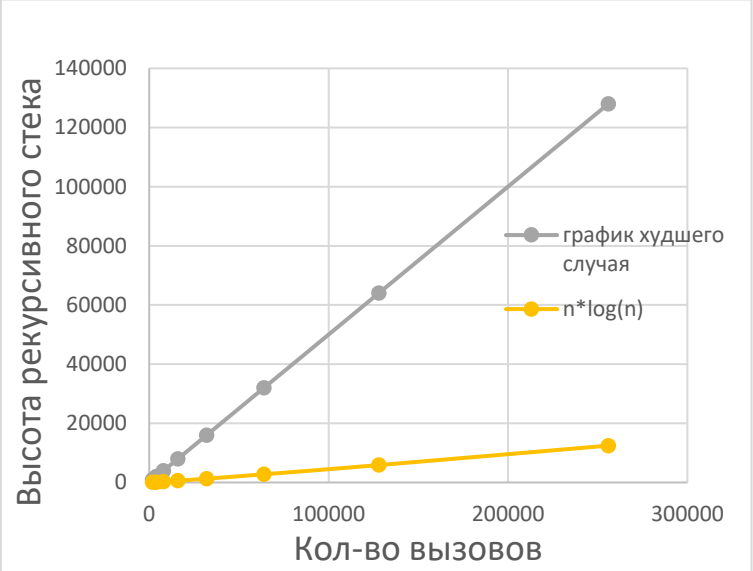

Рис.14 График c*n*log(n) и худшего случаев при тестировании вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента.

8) По графикам видно, что метод быстрой сортировки сильно деградирует в сравнении со средней сложностью на массиве с максимальным количеством сравнений при выборе среднего элемента в качестве опорного и массиве с максимальным количеством сравнений при детерминированном выборе опорного элемента, где высота рекурсивного стека растет линейно.

9)Графики лучшего, среднего и худшего случаев для всех тестов по времени и N

- Для первого

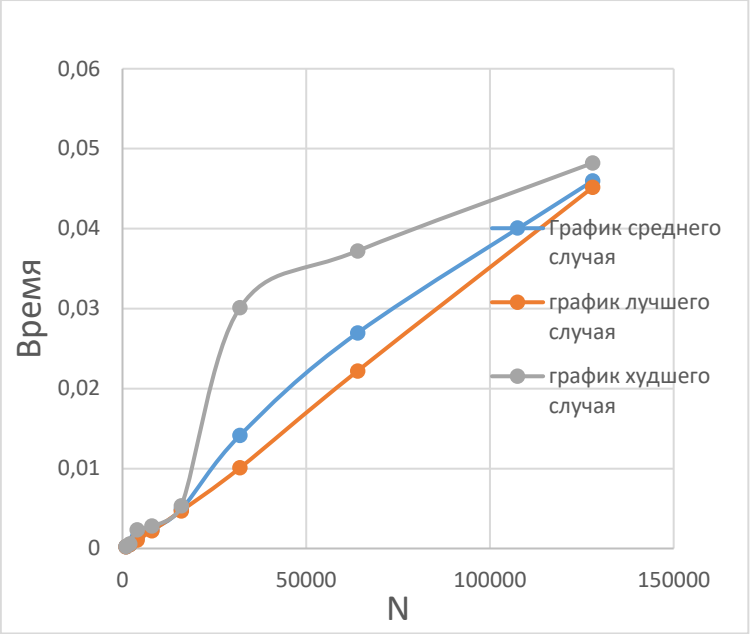

Рис.15 График лучшего, среднего и худшего случаев при тестировании вектора, заполненного рандомными числами.

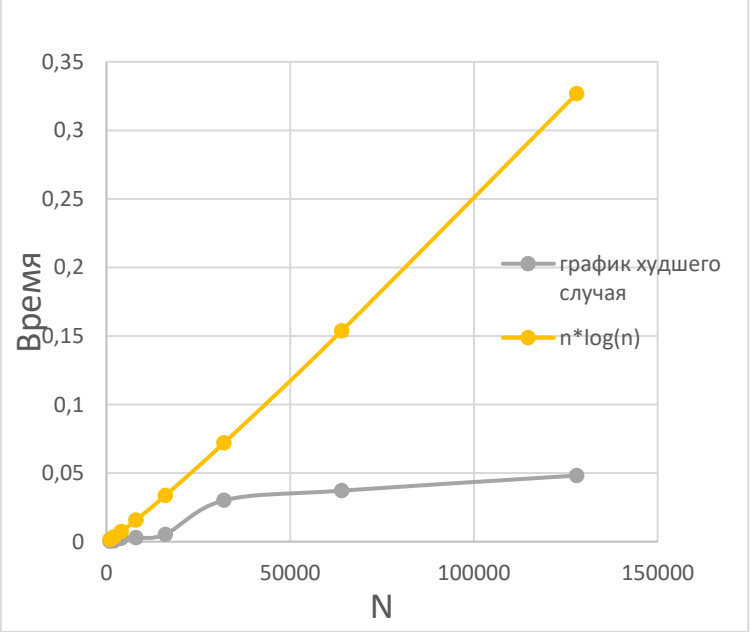

Рис.16 График 0.0000005*n*log(n) и худшего случаев при тестировании вектора, заполненного рандомными числами.

- Для отсортированного.

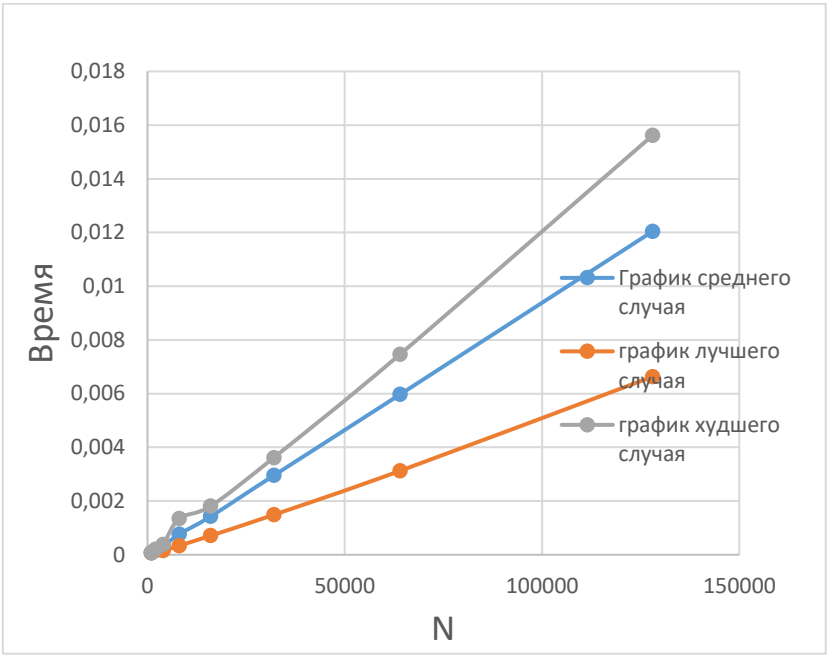

Рис.17 График лучшего, среднего и худшего случаев при тестировании отсортированного.

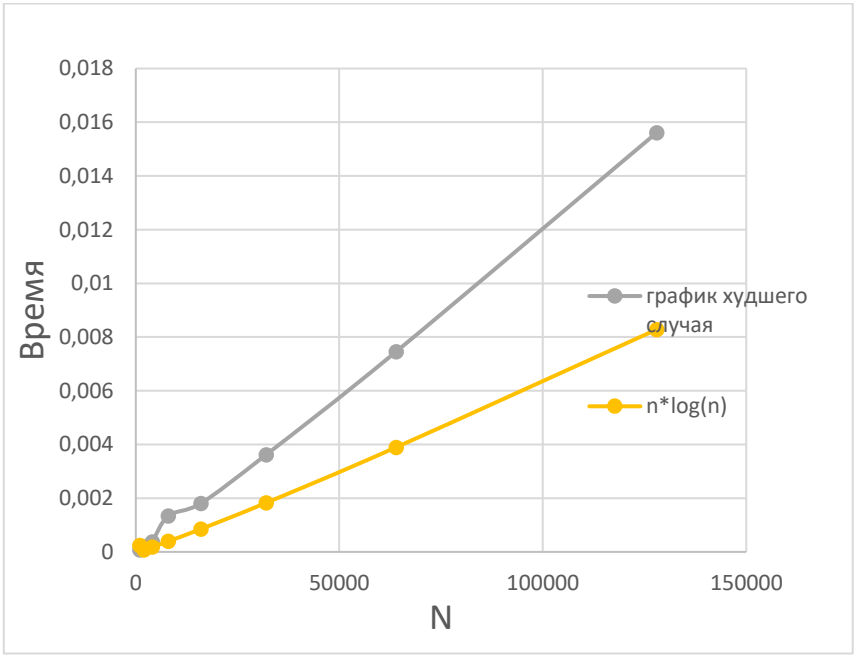

Рис.18 График 0.0000005*n*log(n) и худшего случаев при тестировании отсортированного вектора.

- Для вектора одинаковыми элементами.

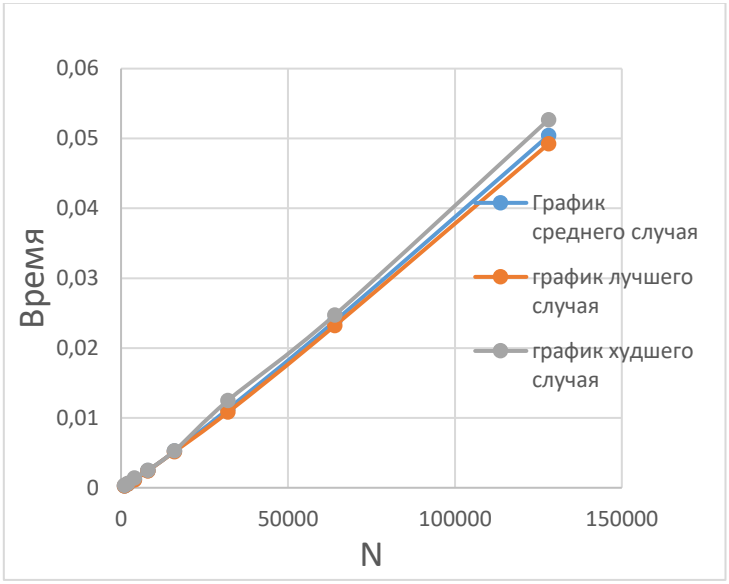

Рис.19 График лучшего, среднего и худшего случаев при тестировании вектора с одинаковыми элементами.

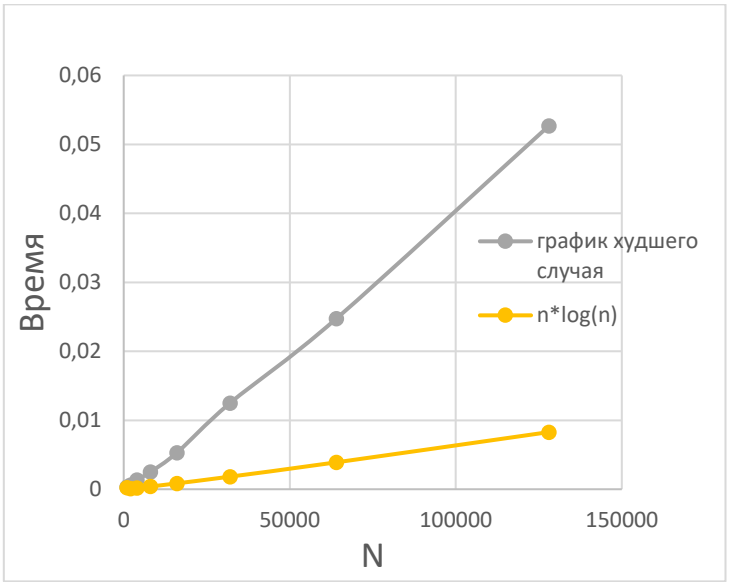

Рис.20 График =0.00001267*n*log(n) и худшего случаев при тестировании вектора из одинаковых элементов.

- Для вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента и с максимальным количеством сравнений при выборе среднего элемента в качестве опорного для всех N.

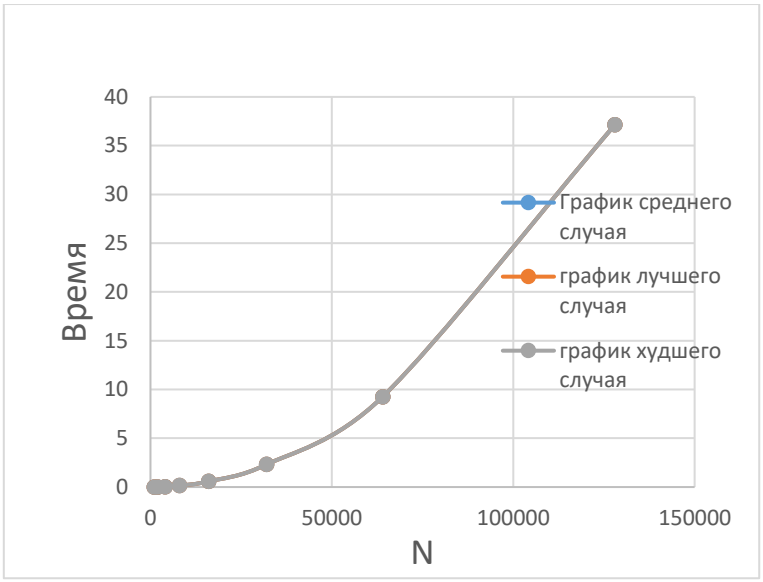

Рис.21 График лучшего, среднего и худшего случаев при тестировании вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента и с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

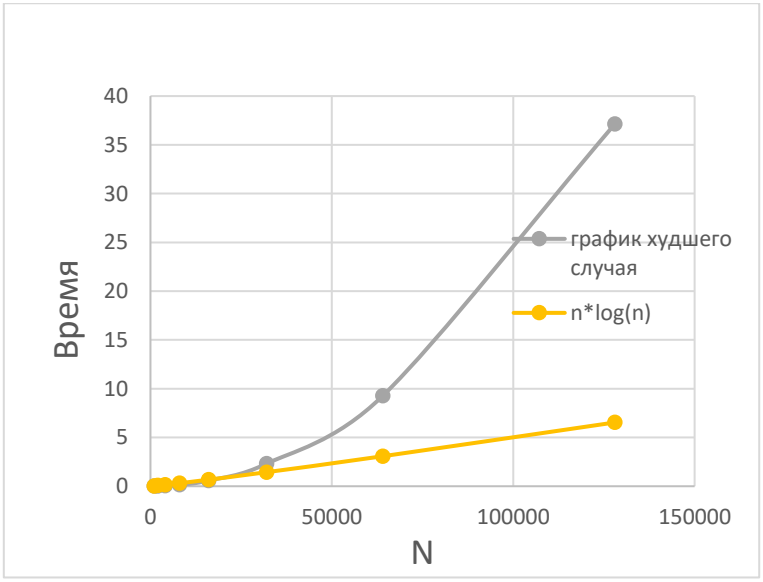

Рис.22 График 0.00001*n*log(n) и худшего случаев при тестировании вектора с максимальным количеством сравнений при детерминированном выборе опорного элемента и с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

10) По графикам видно, что при быстрой сортировке сильно деградирует время выполнения. На массивах, проверяющих негативные случаи, сложность функции повышается до n в отсортированном и массиве, заполненном одинаковыми элементами и до n^2 в векторе с максимальным количеством сравнений при детерминированном выборе опорного элемента и с максимальным количеством сравнений при выборе среднего элемента в качестве опорного.

Заключение.

Метод быстрой сортировки оказался достаточно прост в реализации и быстр в большинстве обычных случаях, но прогонка его по специфичным негативным случаям показала, что его скорость может сильно деградировать на неудачных входных данных.