Федеральное государственное автономное образовательное учреждение высшего образования

«Уральский федеральный университет имени первого Президента России Б.Н.Ельцина» Институт радиоэлектроники и информационных технологий – РТФ

Анализ сложности алгоритмов сортировки

Отчёт по лабораторной работе

по дисциплине «Алгоритмы, структуры данных и анализ сложности» Вариант 6

Выполнил: Комаров

Максим

Сергеевич

Группа:

АТ-02

Преподаватель: доцент, к.ф.-м.н. Трофимов С.П.

2024

**Оглавление**

[Задание 3](#_bookmark0)

[Теоретическая часть 4](#_bookmark1)

[Инструкция пользователя 6](#_bookmark2)

[Инструкция программиста 7](#_bookmark3)

[Тестирование 9](#_bookmark4)

Анализ сложности 10

[Выводы 17](#_bookmark5)

[Литература 1](#_bookmark6)8

[Приложение 1](#_bookmark7)9

# Задание

Реализовать один из алгоритмов сортировки строк:

Пирамидальная сортировка HeapTree

Выбор алгоритма выбирается по согласованию с преподавателем.

Для алгоритма определить сложность относительно наиболее характерной операции (сравнение, перестановка и др.). Вид функции сложности F(n) подобрать в соответствии с теорией. Например, для оптимальных алгоритмов F(n) = C\*n\*log2(n). Найти также коэффициент пропорциональности C. Для аппроксимации можно использовать метод наименьших квадратов и сервис «Поиск решения».

План проведения эксперимента с алгоритмом называется массовой задачей. Представьте план в виде xml-файла.

Результаты решения массовой задачи записать в текстовый файл в 2 столбика: длина массива, количество операций. Файл импортировать в Excel.

В «шапке» листа указать параметры тренда, вычислить квадратичные невязки и минимизировать их сумму с помощью надстройки «Поиск решения»

# Теоретическая часть

Пирамидальная сортировка (англ. Heapsort, «Сортировка кучей») - работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за n)  операций[[1]](#_Литература" \o "1.Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс», 2011.— 1296 с.) при сортировке n элементов, была предложена Дж. Уильямсом в 1963 году.

Сортировка пирамидой использует бинарное сортирующее дерево.

Сортирующее дерево — это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо d-1, либо d, где d — максимальная глубина дерева.
2. Значение в любой вершине не больше значения её двух потомков.

Бинарное сортирующее дерево легко можно представить в виде массива Array, а представление на основе массива является эффективным решением с точки зрения расхода памяти.

Массив Array такой, что, Array[0] — корневой элемент, а потомки элемента каждого родительского элемента с индексом i, Array[i] являются Array[2i+1] и Array[2i+2], где i = 0,…,n/2-1.

Алгоритм сортировки будет состоять из двух основных шагов:

1. Выстраиваем элементы массива в виде бинарного сортирующего дерева:

Array[i] ≥ Array[2i+1],

Array [i] ≥ Array[2i+2] при 

Этот шаг требует O(n) операций, где n - количество элементов в массиве.

1. Благодаря тому, как мы построили дерево, максимальный элемент должен быть в его корне, поэтому возьмём этот максимальный элемент и заменим на минимальный, перестроив дерево, с учётом изменений.

Причём преобразование дерева - это шаг требующий O(log n)

Таким образом на первом шаге мы меняем местами Array[0] и Array[n-1], преобразовываем Array[0], Array[1], … , Array[n-2] в новое сортирующее дерево.

Повторяем вышеуказанные шаги, пока размер кучи больше 1.

Тогда получим Array[0], Array[1], … , Array[n-1] — упорядоченная последовательность.

Всего алгоритм требует порядка O(n\*log n) операций.

Особенности алгоритма:

1. Работает в худшем, в среднем и в лучшем случае (то есть гарантированно) за Θ(n log n) операций при сортировке n элементов.
2. Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).
3. Из-за сложности алгоритма выигрыш получается только на больших n
4. На почти отсортированных массивах работает столь же долго, как и на хаотических данных.
5. Не [распараллеливается](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F" \o "Параллельные вычисления).
6. Неустойчив

# Инструкция пользователя

При запуске программы открывается консоль, где пользователю предлагается провести эксперимент и оценить производительность алгоритма. Если он выразит согласие, то в файл ExperimentResults.txt запишутся результаты для анализа алгаритма в два столбца: 1 столбeц length, второй switchCount. После окончания процедуры записи программы выведет “Запись в файл закончена”.

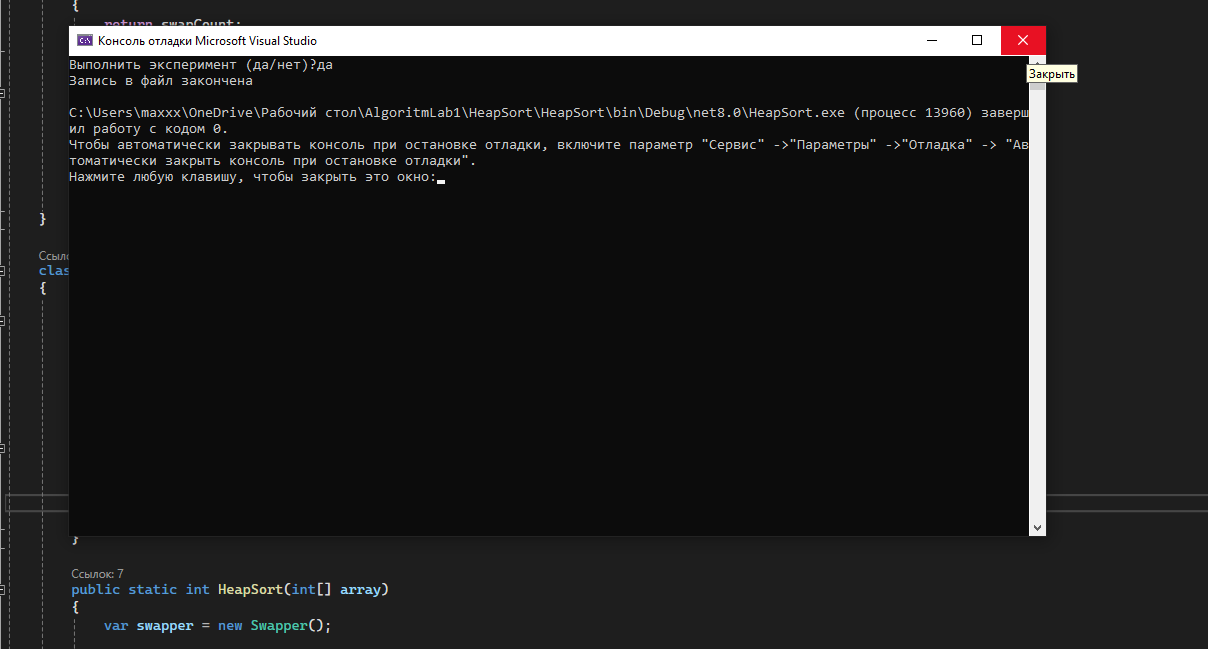


Рис. Проведение эксперимента

# Инструкция программиста

В программе, написанной на языке C#, алгоритм пирамидальной сортировки представлен в классе Program. Для его реализации была написана функция HeapSort:

public static int HeapSort(int[] array)

Принимающая массив чисел сортирующая его алгоритмом HeapSort. Для возможности оценивать производительность сортировки функция возвращает значение счётчика основной операции - swap.

Для удобства подсчёта количества операции Swap, создан класс Swapper ,который включает в себя публичный метод public void Swap<T>(ref T first, ref T second), меняющий местами два элемента, а также закрытое поле private int swapCount; ,в котором подситывается количество исползований метода swap.

Получить swapCount можно с помощью метода public int GetSwapCount()

Помимо этих основных функций, в программе реализованы вспомогательные. Это функция:

private static void GetResultsFromXML()

,выполняющая экспериментальные вычисления и записывающая их в файл ExperimentResults.txt, и

private static int GenerateArrAndGetCount(int minElement, int maxElement, int repeatsCount, int arrayLength)

,принимающая параметры для создания массивов чисел(их длину, диапазон значений элементов, количество массивов); она формирует массивы, запускает функцию сортировки для них и возвращает счётчик самой основной операции.

Также функции:

private static int[] GenerateSortedArray(int minElement, int arrayLength)

private static int[] GenerateReverseSortedArray(int maxElement, int arrayLength)

генерирующие отсортированные и отсортированные в обратном порядке массивы, которые нужны для:

private static int GenerateGoodDataAndGetCount(int maxElement, int arrayLength)

private static int GenerateBadDataAndGetCount(int minElement, int arrayLength), которые по аналогии с GenerateArrAndGetCount, генерируют массивы с помощью функций выше и выврдят количество использованных основных функций swap.

Тем не менее основная функция программы - проведение пирамидальной сортировкой на больших массивах данных. Для этого в xml-файле Experiments был создан план эксперимента. Он включает в себя несколько элементов nodes, каждый из которых описывает свою часть эксперимента: какое количество массивов будет сгенерировано, какой длины, с какими значениями и по какому принципу эти массивы будут изменяться (здесь реализовано создание средних данных - создание массивов с длинами полученными из арифметической прогрессии, а также “хорошие” данные - полученные из обратно отсортированных массивов и “плохие” данные - полученные из отсортированных массивов).

Данный документ обрабатывается в функции GetResultsFromXML: значения атрибутов каждого элемента эксперимента будут получены и использованы для генерации массивов (все значения их элементов выбираются случайным образом из заданного диапазона) и последующей сортировки.

# Тестирование

Кроме основных и вспомогательных, программа также содержит функцию TestHeapSort.

Она содержит тесты, проверяющий корректность работы алгоритма сортировки и выполняется в момент запуска программы. Если один или несколько тестов завершились неудачно, на консоль будут выведены соответствующие сообщения, и программа завершит работу.

Часть кода тестов представлена ниже, с полным кодом можно ознакомиться в Приложении.

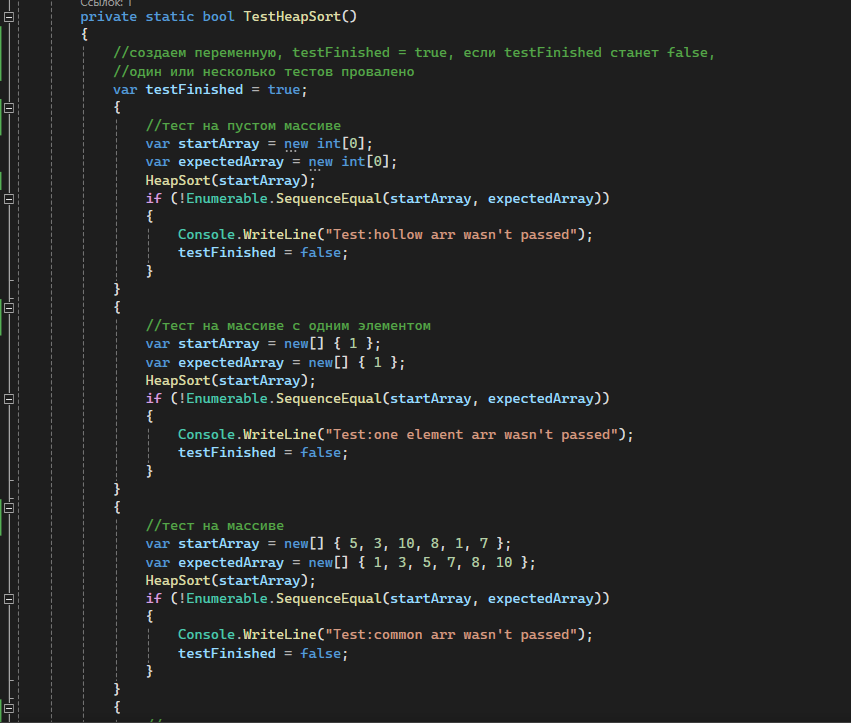


Рис. Часть кода тестирующей функции TestHeapSort

**Анализ сложности алгоритма**

Анализ сложности алгоритма, я проводил в excel, применяя результаты из ExperimentResults.txt, полученные с помощью созданной программы.

Стоит остановиться на выбранных данных.

Средние данные, я получал, создавая массивы с длинами рассчитанными по арифметической прогрессии и заполняемыми случайными числами в заданном диапазоне, все эти параметры задаются в xml файле Experiments.xml.

Однако, что насчёт “хороших” и “плохих” данных. Под “хорошими” данными я подразумеваю данные, при которых алгоритм сортировки использует меньшее количество основных операций - swap.

Под “плохими” соответственно, данные, при которых алгоритм сортировки использует большее количество основных операций.

Изначально идея, о том, что из себя представляют эти данные была чисто теоретической, основанной на моём понимании работы алгоритма, но позднее это подтвердилось, сравнив результаты в ExperimentResults.txt.

Итак, из теории мы знаем, что для HeapSort, мы строим сортировочное бинарное дерево с максимальным элементом в корне.

Поэтому теоретически, в массиве отсортированным в обратном порядке, т.е. с максимальным элементом в начале, мы используем гораздо меньше основных операций, для построения начального сортировочного бинарного дерева.

Ещё меньше операций было бы, если бы мы подавали массивы в алгоритм сортировки прямо в нужной конфигурации, но я посчитал это - излишним.

Соответственно я поступил с “плохими” данными, ведь если изначальный массив будет отсортирован в порядке возрастания, то HeapSort использует большее количество swap операций.

Как видно из полученных результатов, эти выводы были верны.

“Средние” данные: при длинне массива 1099, операций - 10078

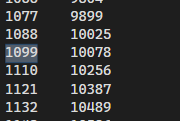


Рис. Часть “средних” данных из результата

“Плохие” данные: при длине массива 1099, операций - 10845

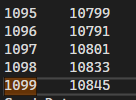


Рис. Часть “плохих” данных из результата

“Хорошие” данные: при длинне массива 1099, операций - 9316

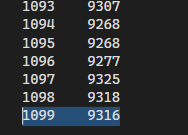


Рис. Часть “хороших” данных из результата

Далее перейдём к непосредственно анализу данных.

Для их сближения данных и воспользуемся методом наименьших квадратов: рассчитываем для каждого случая квадратичные невязки, а затем минимизируя сумму квадратов невязок, получим коэффициенты для функции сложности с помощью сервиса Excel «Поиск решения».

Чтобы получить оценку сверху и снизу, необходимо также рассчитать тренд и в «Поиске решений» добавить ограничение, на то, что столбец трендов **больше** столбца с количеством swap для оценки сверху, и **меньше** для оценки снизу.

Из теории мы знаем, что в среднем HeapSort должен работать n\*log(n), в худшем случае он будет работать порядка n2, тем не менее, он не должен быть быстрее, чем n\*log(n).

Поэтому я нашёл оценку сверху, при “плохих” данных, оценку снизу, при “хороших” данных, а также среднюю для средних данных.

Для оценки сверху я использовал тренд a\*n\*n + b\*n + c

Вот, что получилось:

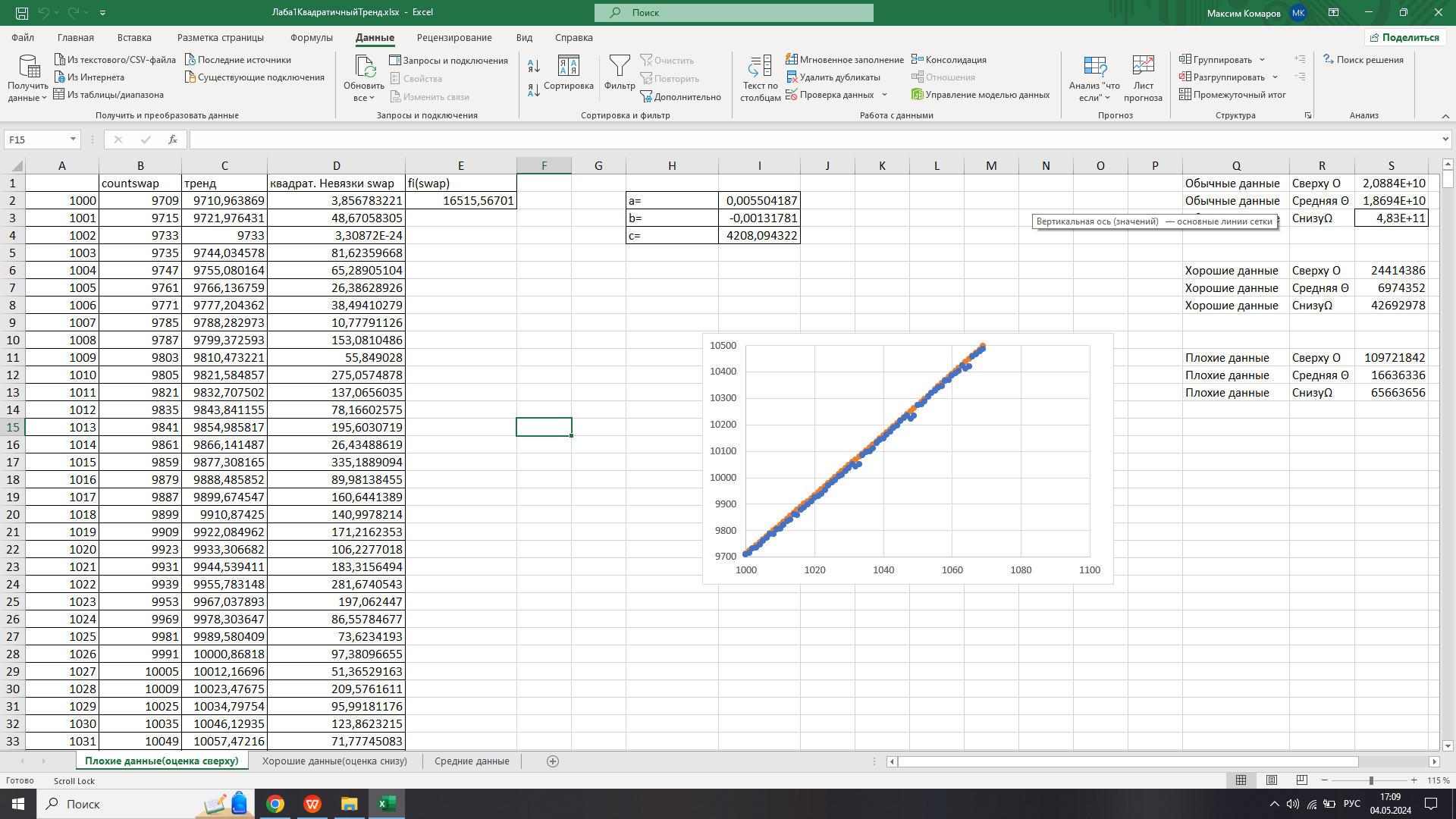


Рис. Ассимтотическая оценка сверху функции сложности алгоритма HeapSort,

тренд a\*n\*n + b\*n + c

Для сравнения оценка сверху по тренду a\*n\*logn + b\*n + c

Видно, что сумма квадратов невязок, в этом случае больше, а значит подходит нам меньше

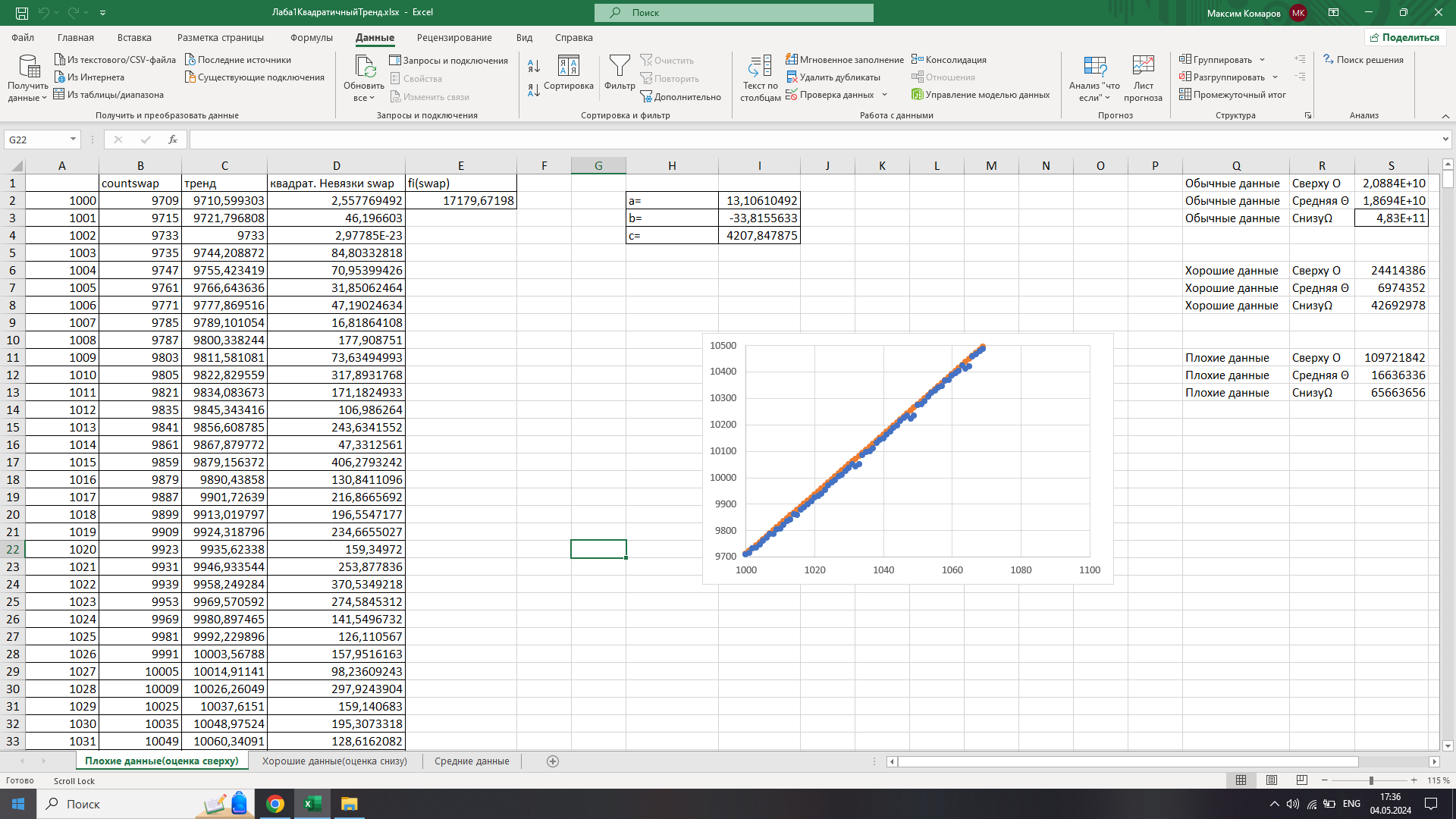


Рис. Ассимтотическая оценка сверху функции сложности алгоритма HeapSort,

тренд a\*n\*logn + b\*n + c

Для средней оценки я решил попробовать оба тренда, чтобы проверить, действительно ли средняя сложность порядка n\*logn

И как видно из графиков и данных ниже, это действительно так!

Сумма квадратов невязок для тренда a\*n\*logn + b\*n + c = 98872

Тогда, как для тренда a\*n2 + b\*n + c = 1743327

По графикам тоже чётко видно, что подходит больше для средней оценки.

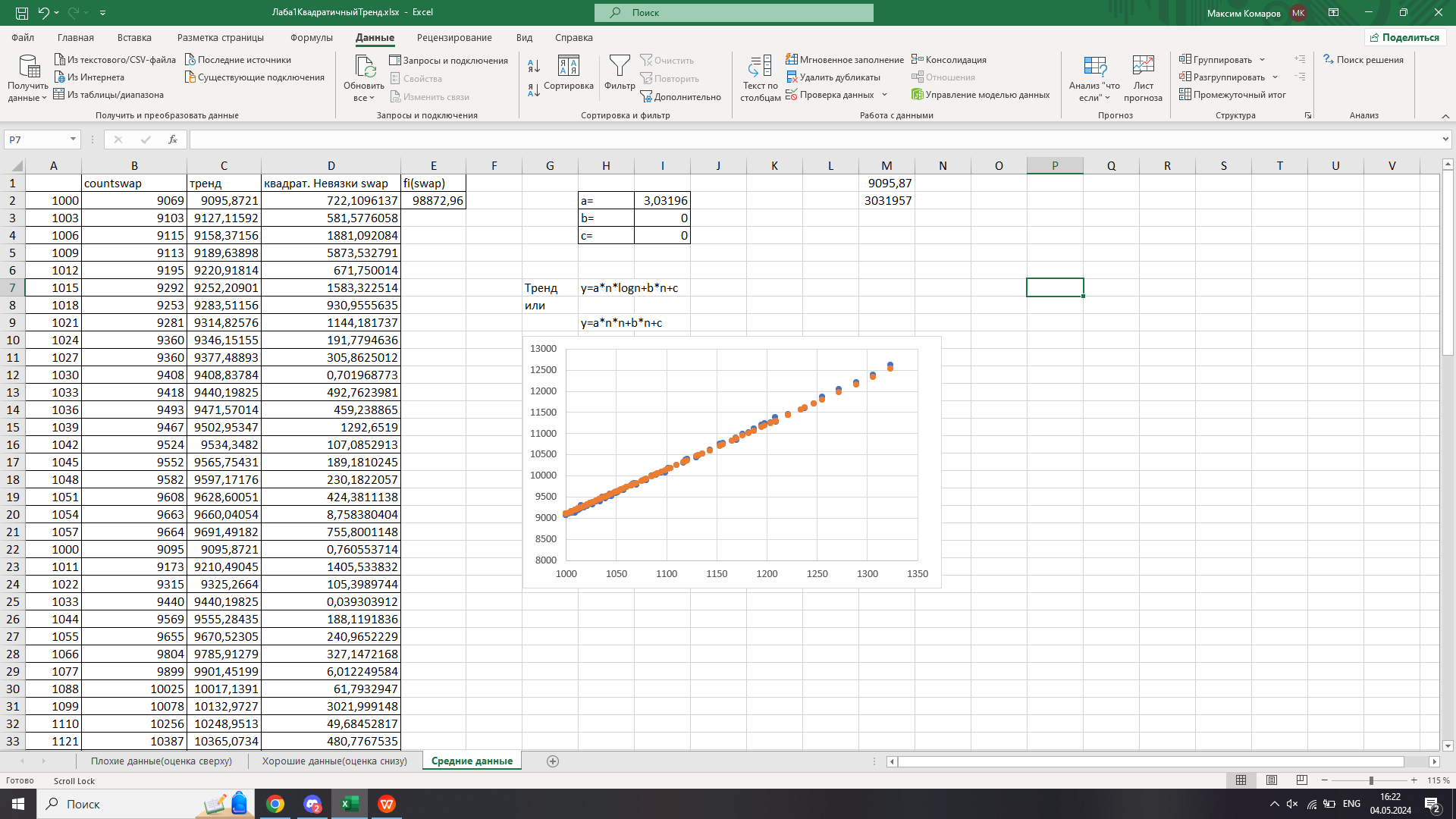


Рис. Ассимтотическая средняя оценка функции сложности алгоритма HeapSort,

тренд a\*n\*logn + b\*n + c

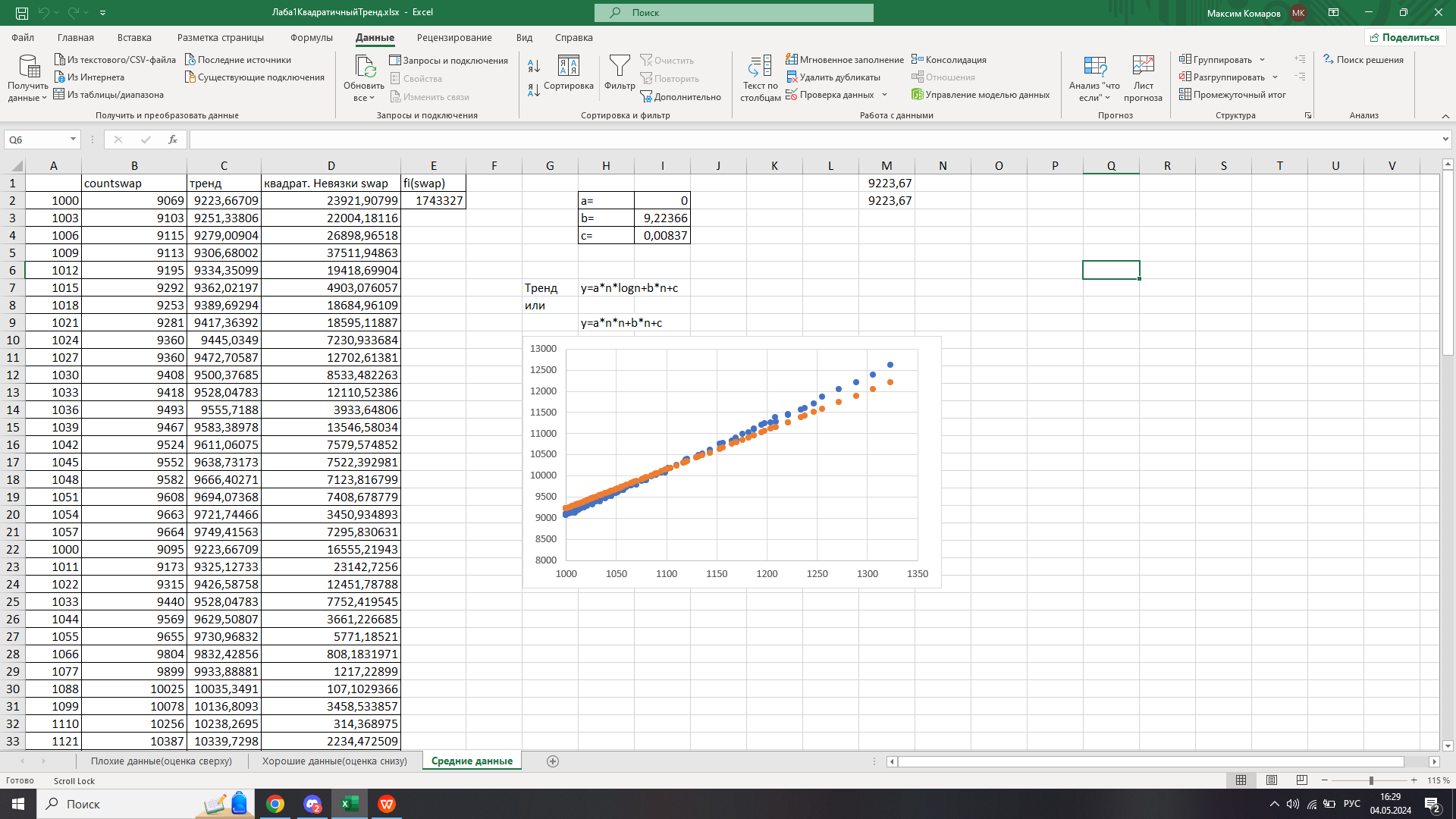


Рис. Ассимтотическая средняя оценка функции сложности алгоритма HeapSort,

тренд a\*n\*n + b\*n + c

С оценкой снизу всё получилось несколько сложнее, для начала я оценил работу, используя тренд a\*n\*logn + b\*n + c

Вот, что получилось:

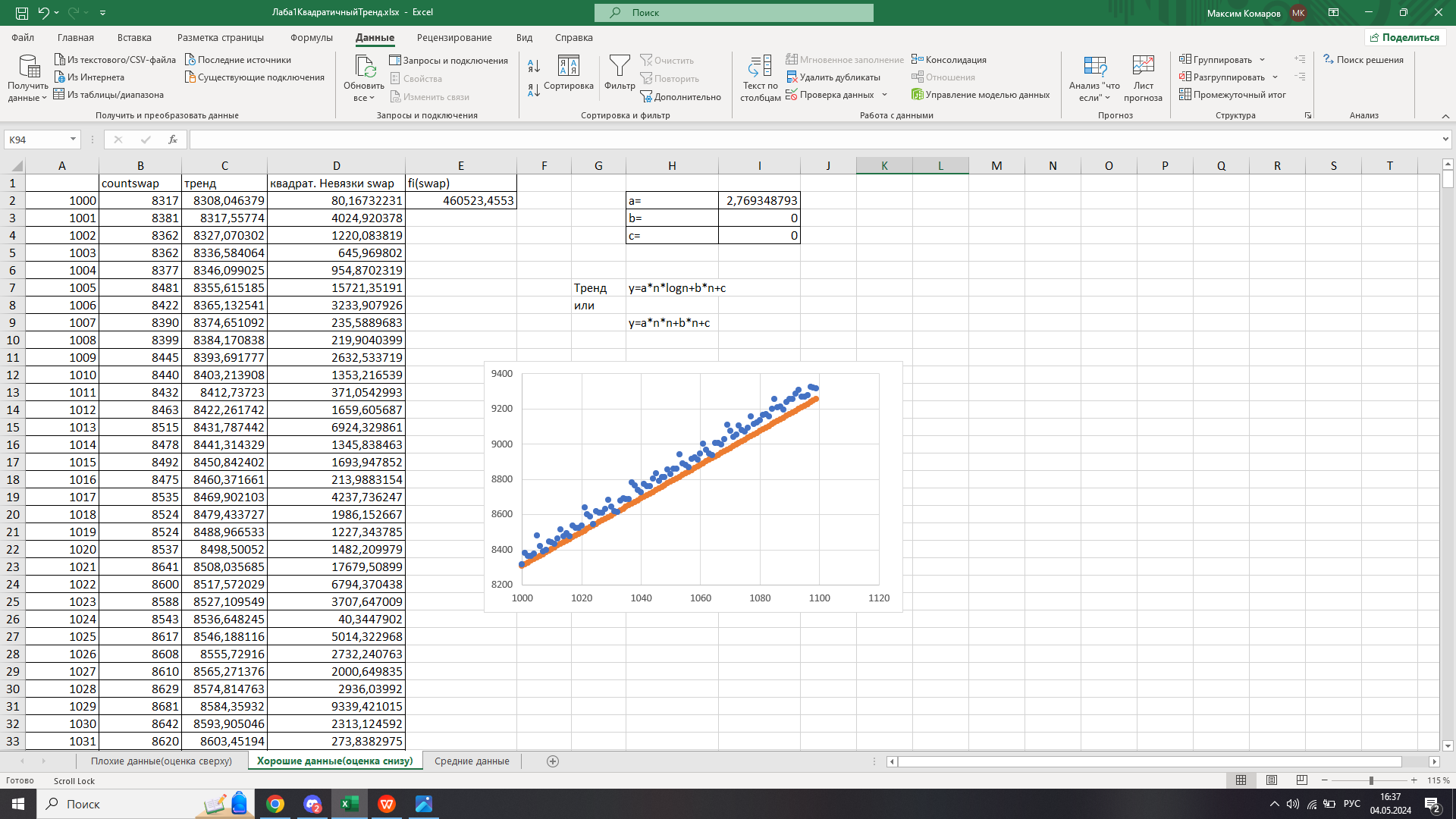


Рис. Ассимтотическая оценка снизу функции сложности алгоритма HeapSort,

тренд a\*n\*logn + b\*n + c

Однако проверив тренд a\*n\*n + b\*n + c, оказалось, что он подходит больше, ведь сумма квадратов невязок во втором случае 335133, а в первом 460523

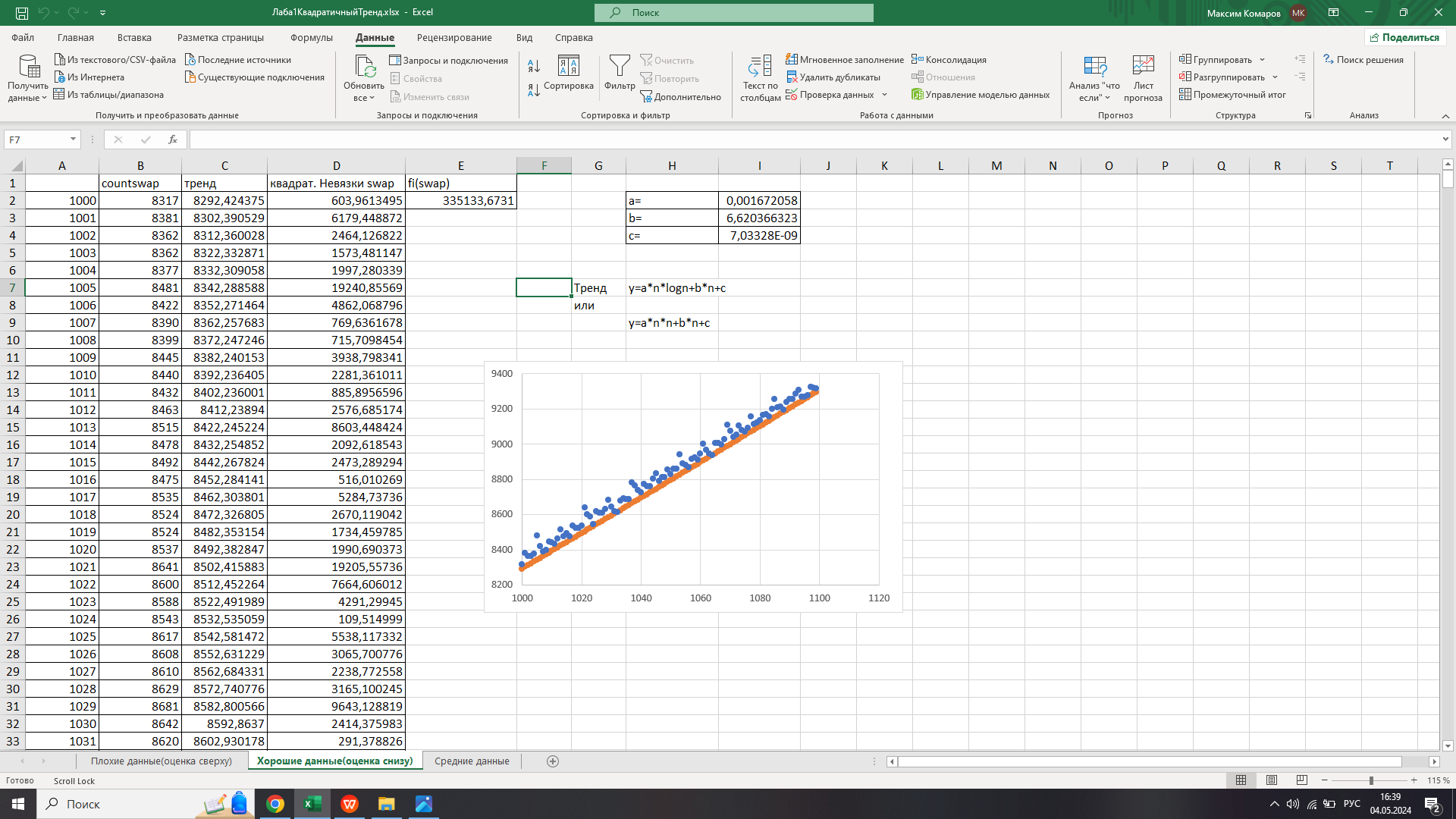


Рис. Ассимтотическая оценка снизу функции сложности алгоритма HeapSort,

тренд a\*n\*logn + b\*n + c

Казалось бы, почему так? Казалось бы для оценки снизу лучше должен был подойти другой тренд. Так как я знаю, что многие алгоритмы лучше работают на больших n, а на малых менее эффективны. Я предположил, что мы имеем, именно, с этим случаем.

Изначально я проверял всё на длинах массивов начиная от 1000, поэтому я решил увеличить это число.

На 10000, результаты не изменились, сумма квадратов невязок, у квадратичного тренда была меньше.

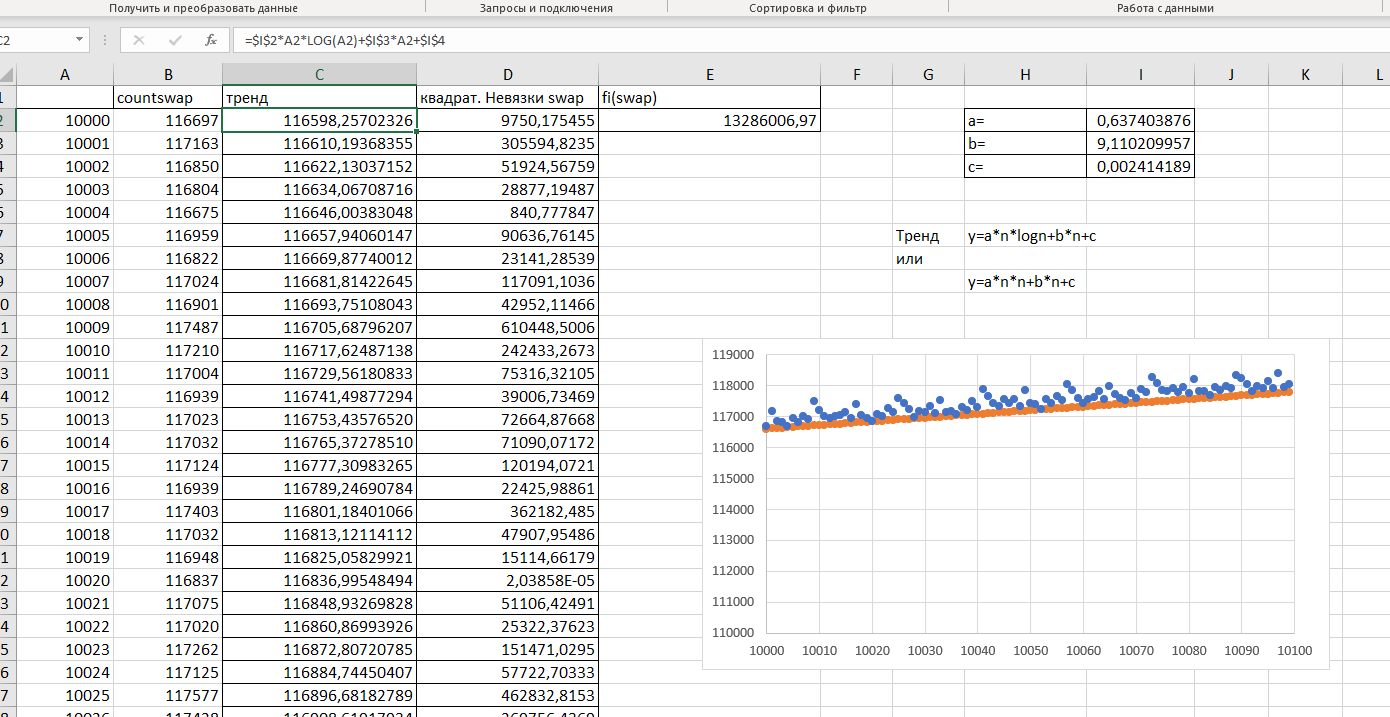


Рис. Ассимтотическая оценка снизу функции сложности алгоритма HeapSort,

тренд a\*n\*logn + b\*n + c, при длинах массивов 10000+

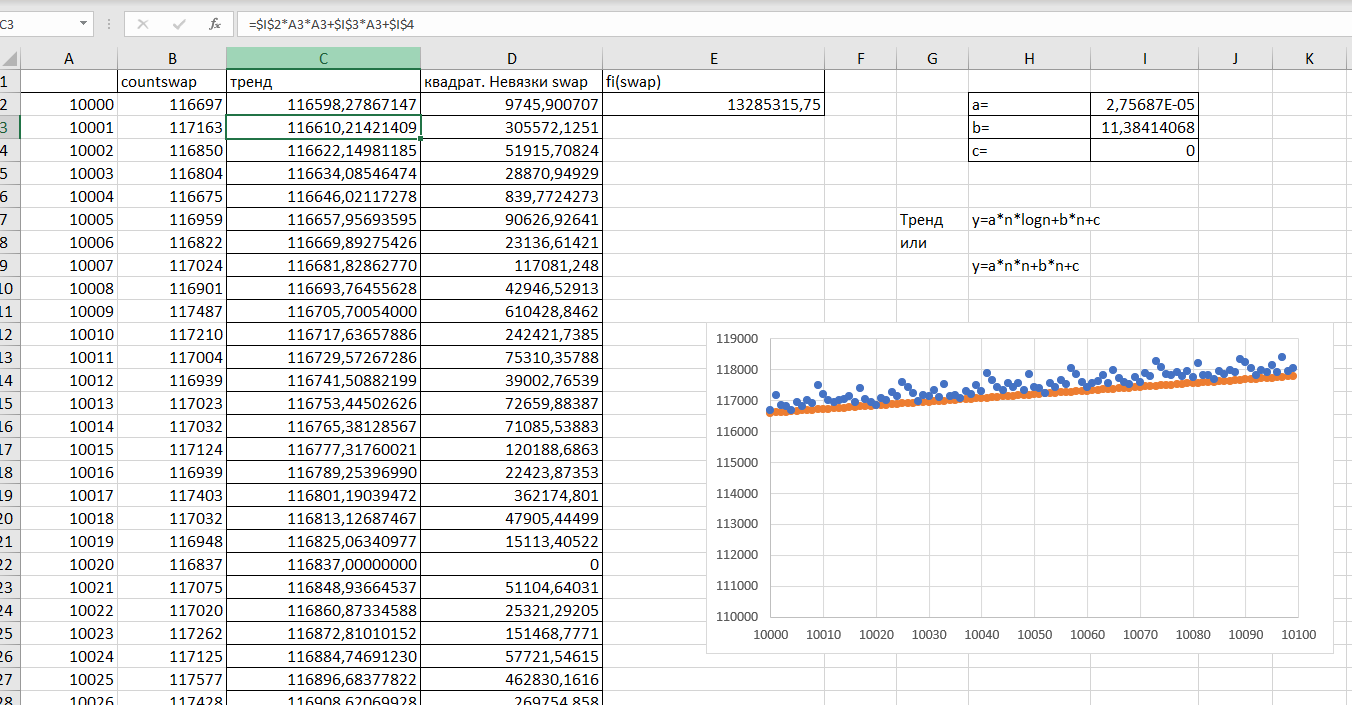


Рис. Ассимтотическая оценка снизу функции сложности алгоритма HeapSort,

тренд a\*n\*n + b\*n + c, при длинах массивов 10000+

Однако на 100000, эта теория подтвердилась:

Сумма квадратов невязок, при квадратичном тренде = 2368468315

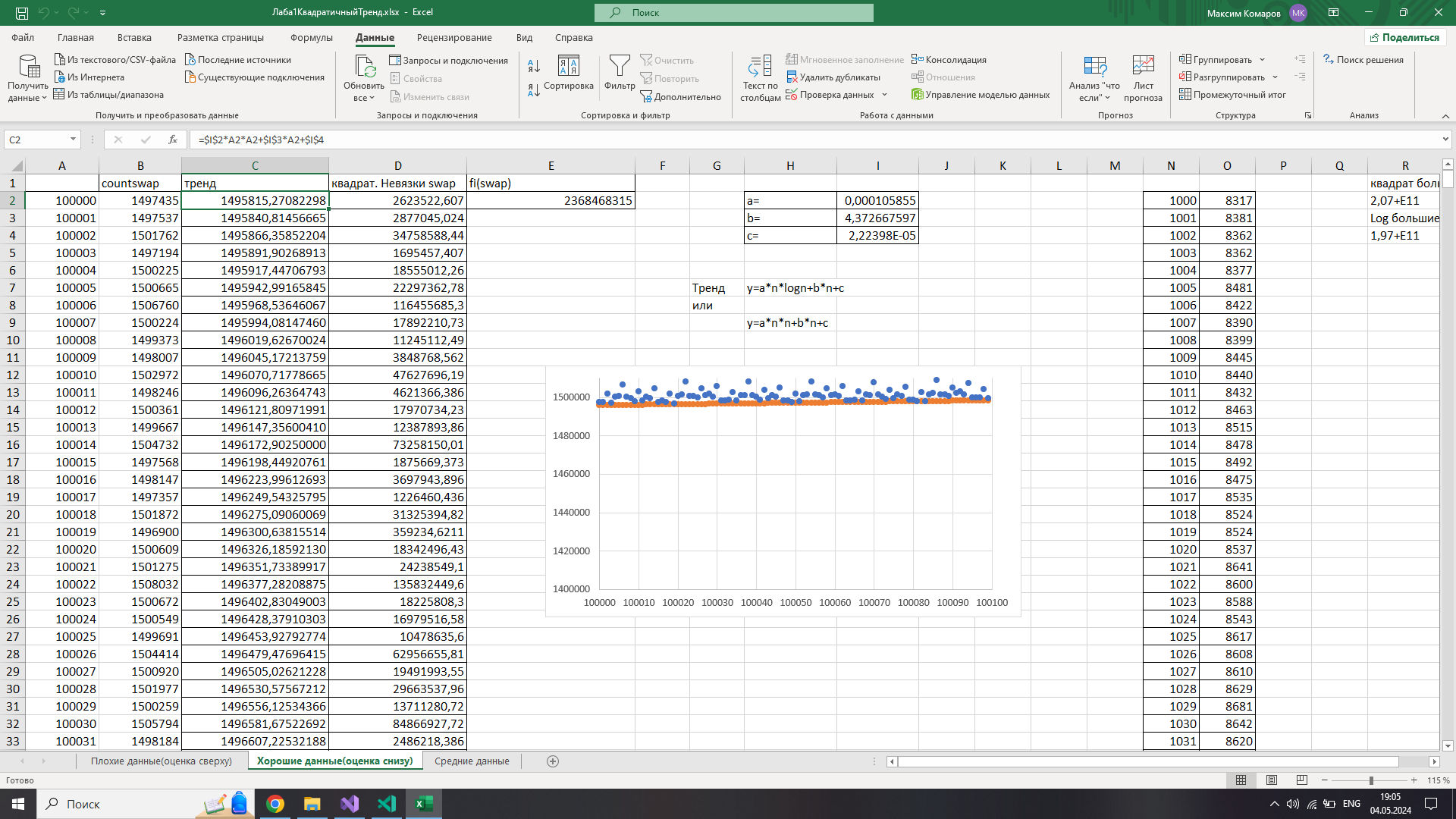


Рис. Ассимтотическая оценка снизу функции сложности алгоритма HeapSort,

тренд a\*n\*n + b\*n + c, при длинах массивов 100000+

Сумма квадратов невязок, при квадратичном тренде = 2128816859

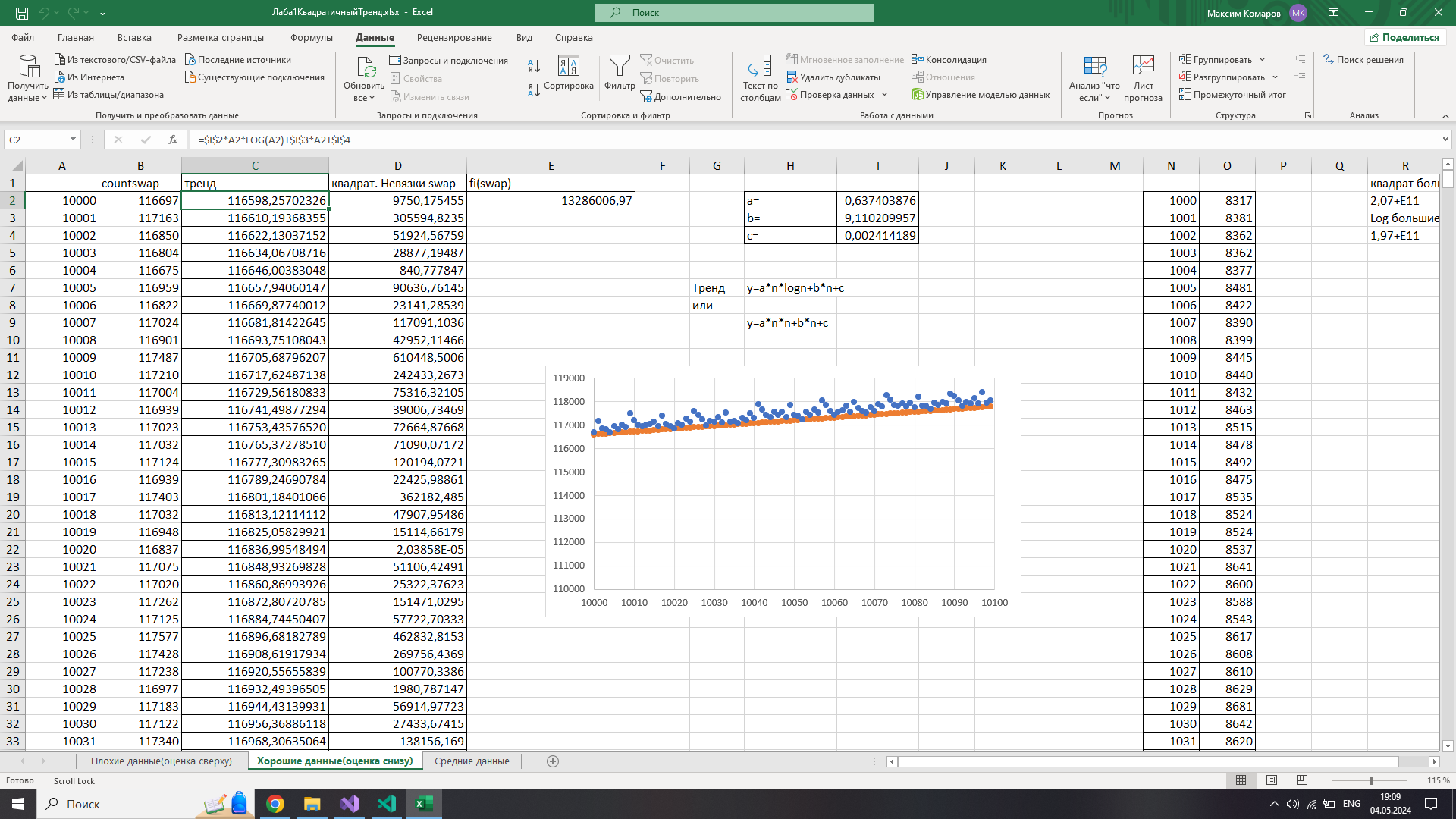


Рис. Ассимтотическая оценка снизу функции сложности

HeapSort, тренд a\*n\*n + b\*n + c, при длинах массивов 100000

Таким образом оценки функции сложности выглядят примерно так:

При n≈1000

Оценка сверху O(n) ≈ 0,005\*n2 - 0,001\*n + 4308

Оценка снизу Ω(n) ≈ 3\*n\*log(n)

Средняя оценка Θ(n)≈ 0,0017\*n2 + 6,62\*n + 7\*10-9

**Вывод**

При написании данной работы, я более подпробно познакомился с одним из алгоритмов сортировки - HeapSort, т.к. мне необходимо было написать программу для сортировки массивов чисел с его использованием.

Также познакомился с xml, и попробовал поработать с этой технологией при задании массовой задачи для эксперимента, а также провел эксперимент для оценки сложности данного алгоритма.

Проанализировал различные данные с помощью «Поиска решений», нашел среднюю оценку, оценку сверху и снизу, а также нашел соответствующие коэффициенты.

Полученная программа работает исправно и позволяет достаточно быстро сортировать массивы из тысяч строк.

Это же подтверждается и результатами эксперимента, ведь алгоритм в среднем работает 3\*n\*logn сложностью, что говорит проведённый анализ.

К тому же, требует константное значение дополнительной памяти, т. е. не зависит от размера входных данных.[[2]](#_Литература" \o "https://ru.wikipedia.org/wiki/Пирамидальная_сортировка)

Всё это говорит о том, что пирамидальная сортировка выигрыш в скорости получается только на достаточно больших данных, что опять же подтверждает анализ и эксперимент.

Этот вывод подтверждают слова из статьи Пирамидальная сортировка (HeapSort):

“Пирамидальная сортировка — это вполне годный алгоритм. Его типичная реализация не стабильна, но может быть таковой сделана…Алгоритм пирамидальной сортировки имеет ограниченное применение, потому что Quicksort (Быстрая сортировка) и Mergesort (Сортировка слиянием) на практике лучше.”[[3]](#_Литература" \o "https://habr.com/ru/companies/otus/articles/460087/)

# Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс», 2011.— 1296 с.
2. <https://habr.com/ru/company/otus/blog/460087/>
3. [https://ru.wikipedia.org/wiki/](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%BC%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)Пирамидальная\_сортировка

# Приложение

using System;

using System.Xml;

namespace Лаб2\_Сортировка\_чисел

{

public class Swapper

{

public int branchesCount;

private int swapCount;

public int GetSwapCount()

{

return swapCount;

}

public void Swap<T>(ref T first, ref T second)

{

var temp = first;

first = second;

second = temp;

swapCount++;

}

}

class Program

{

static void Main(string[] args)

{

//Тестирование, если хоть один тест провален, программа завершается

if (!TestHeapSort())

return;

//Выполнение эксперимента

Console.Write("Выполнить эксперимент (да/нет)?");

var answer = Console.ReadLine();

if (answer == "да")

{

GetResultsFromXML();

Console.WriteLine("Запись в файл закончена");

}

}

//Метод сортировки

public static int HeapSort(int[] array)

{

//Создаём объект класса Swapper, для получения доступа к методу Swap() и подсчёту кол-во swap

var swapper = new Swapper();

var heapSize = array.Length;

//Тут происъодит начальное построение "дерева"

for (var i = heapSize / 2 - 1; i >= 0; i--)

Heapify(array, heapSize, i, swapper);

for (var i = heapSize - 1; i >= 0; i--)

{

//выносим макс элемент в ту часть массива,

//куда складываем результирующий массив(отсортированная часть)

swapper.Swap(ref array[0], ref array[i]);

//После того, как перенесли макс элемент, стабилизируем дерево вновь

Heapify(array, i, 0, swapper);

}

//в конце возвращаем кол-во swap

return swapper.GetSwapCount();

}

//Метод перестройки "дерева"

public static void Heapify(int[] arr, int length, int i, Swapper swapper)

{

//инициализируем largest - корень, left, right - потомки

var largest = i;

var left = 2 \* i + 1;

var right = 2 \* i + 2;

// Если левый потомок больше корня, то он - должен стать корнем

if (left < length && arr[left] > arr[largest])

largest = left;

// Если правый потомок больше корня, то он - должен стать корнем

if (right < length && arr[right] > arr[largest])

largest = right;

//Если корень поменялся

if (largest != i)

{

//То меняем его положение в дереве

swapper.Swap(ref arr[i], ref arr[largest]);

//Повторяем процедуру перестройки дерева, пока не построим

Heapify(arr, length, largest, swapper);

}

swapper.branchesCount += 3;

}

//Метод работы с XML и печати результатов в файл

private static void GetResultsFromXML()

{

//создаем оюъкт класса xmlDocument

var xmlDoc = new XmlDocument();

//Загружаем в него наш xml

xmlDoc.Load(@"C:\Users\maxxx\OneDrive\Рабочий стол\AlgoritmLab1\Experiments.xml");

//Получаем из него корневой элемент experiment, в котором описана массовая задача

var xmlRootNode = xmlDoc.DocumentElement;

var experiment = xmlRootNode.SelectSingleNode("experiment");

//Создаем writer,чтобы записывать результаты в файл, который будет находиться в projectDirectory

var projectDirectory = Directory.GetParent(Environment.CurrentDirectory).Parent.Parent.FullName;

var writer = new StreamWriter(Path.Combine(projectDirectory, "ExperimentResults.txt"));

using (writer)

{

//проходимся по nodes

foreach (XmlNode node in experiment.ChildNodes)

{

//Парсим некоторые общие данные из nodes

var startLength = int.Parse(node.SelectSingleNode("@startLength").Value);

var repeat = int.Parse(node.SelectSingleNode("@repeat").Value);

var name = node.SelectSingleNode("@name").Value;

//Далее в зависимости от name nodes, получаем доп данные

if (name == "Arithmetic Progression")

{

var minElement = int.Parse(node.SelectSingleNode("@minElement").Value);

var maxElement = int.Parse(node.SelectSingleNode("@maxElement").Value);

var diff = int.Parse(node.SelectSingleNode("@diff").Value);

//var maxElement = startLength + repeat \* diff;

var length = startLength;

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

{

length = startLength + i \* diff;

//генерируем массивы и получаем из них swap count, сортируя их

var operationsCount = GenerateArrAndGetCount(minElement, maxElement, length);

//записываем, что нужно в файл

writer.WriteLine(length + "\t" + operationsCount);

}

}

//Тут всё то же самое, что и выше

if (name == "Bad Data")

{

writer.WriteLine("Bad Data");

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

{

var minElement = int.Parse(node.SelectSingleNode("@minElement").Value);

var length = startLength + i;

//только метод создания массивов и получения из них swapCount другой

var operationsCount = GenerateBadDataAndGetCount(minElement, length);

writer.WriteLine(length + "\t" + operationsCount);

}

}

//Тут всё то же самое, что и выше

if (name == "Good Data")

{

writer.WriteLine("Good Data");

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

{

var maxElement = int.Parse(node.SelectSingleNode("@maxElement").Value);

var length = startLength + i;

//только метод создания массивов и получения из них swapCount другой

var operationsCount = GenerateGoodDataAndGetCount(maxElement, length);

writer.WriteLine(length + "\t" + operationsCount);

}

}

}

}

}

//Метод генерации массивов и получения swapCount после их сортировки

private static int GenerateArrAndGetCount(int minElement, int maxElement, int arrayLength)

{

//создаём массив

var array = new int[arrayLength];

//заполняем его рандомными элементам в промежутке от minElement до maxElement

var random = new Random();

for (var j = 0; j < array.Length; j++)

array[j] = random.Next(minElement, maxElement);

//сортируем массив и возвращаем swapCount

return HeapSort(array);

}

//Метод генерации массивов и получения swapCount после их сортировки

private static int GenerateBadDataAndGetCount(int minElement, int arrayLength)

{

//Получаем уже отсортированный массив

var array = GenerateSortedArray(minElement, arrayLength);

//сортируем массив и возвращаем swapCount

return HeapSort(array);

}

//Метод генерации массивов и получения swapCount после их сортировки

private static int GenerateGoodDataAndGetCount(int maxElement, int arrayLength)

{

//Получаем массив отсортированный в обратном порядке

var array = GenerateReverseSortedArray(maxElement, arrayLength);

//сортируем массив и возвращаем swapCount

return HeapSort(array); ;

}

//Создаем массив

private static int[] GenerateSortedArray(int minElement, int arrayLength)

{

var array = new int[arrayLength];

for (var i = 0; i < arrayLength; i++)

{

//каждый следующий элемент > предыдущего

array[i] = minElement + i;

}

//отдаем отсортированный массив

return array;

}

//Создаем массив

private static int[] GenerateReverseSortedArray(int maxElement, int arrayLength)

{

var array = new int[arrayLength];

for (var i = 0; i < arrayLength; i++)

{

//каждый следующий элемент < предыдущего

array[i] = maxElement - i;

}

//отдаем отсортированный в обратном порядке массив

return array;

}

//метод тестирования

private static bool TestHeapSort()

{

//создаем переменную, testFinished = true, если testFinished станет false,

//один или несколько тестов провалено

var testFinished = true;

{

//тест на пустом массиве

var startArray = new int[0];

var expectedArray = new int[0];

HeapSort(startArray);

if (!Enumerable.SequenceEqual(startArray, expectedArray))

{

Console.WriteLine("Test:hollow arr wasn't passed");

testFinished = false;

}

}

{

//тест на массиве с одним элементом

var startArray = new[] { 1 };

var expectedArray = new[] { 1 };

HeapSort(startArray);

if (!Enumerable.SequenceEqual(startArray, expectedArray))

{

Console.WriteLine("Test:one element arr wasn't passed");

testFinished = false;

}

}

{

//тест на массиве

var startArray = new[] { 5, 3, 10, 8, 1, 7 };

var expectedArray = new[] { 1, 3, 5, 7, 8, 10 };

HeapSort(startArray);

if (!Enumerable.SequenceEqual(startArray, expectedArray))

{

Console.WriteLine("Test:common arr wasn't passed");

testFinished = false;

}

}

{

//тест на массиве, в котором есть повторяющиеся элементы

var startArray = new[] { 1, 0, 0, 1 };

var expectedArray = new[] { 0, 0, 1, 1 };

HeapSort(startArray);

if (!Enumerable.SequenceEqual(startArray, expectedArray))

{

Console.WriteLine("Test:repeated elements arr wasn't passed");

testFinished = false;

}

}

//возвращаем прошли ли все тесты

return testFinished;

}

}

}

Содержимое xml-файла Experiments:

<?xml version="1.0" encoding="utf-8" ?>

    <experiments>

        <experiment name= "HeapTree">

            <nodes name = "Arithmetic Progression" minElement = "1" maxElement = "800" diff = "3" startLength="1000"  repeat = "20">

            </nodes>

            <nodes name = "Arithmetic Progression" minElement = "1" maxElement = "800" diff = "11" startLength="1000"  repeat = "20">

            </nodes>

            <nodes name = "Arithmetic Progression" minElement = "1" maxElement = "800" diff = "17" startLength="1000"  repeat = "20">

            </nodes>

            <nodes name = "Arithmetic Progression" minElement = "1" maxElement = "800" diff = "5" startLength="1000"  repeat = "20">

            </nodes>

            <nodes name = "Arithmetic Progression" minElement = "1" maxElement = "800" diff = "13" startLength="1000"  repeat = "20">

            </nodes>

            <nodes name = "Bad Data" minElement = "1" startLength="1000" repeat = "100">

            </nodes>

            <nodes name = "Good Data"  maxElement = "10000" startLength="10000" repeat = "100">

            </nodes>

        </experiment>

    </experiments>