Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Сортировки»**

**Выполнил**:

студентка группы 3821Б1ПМ2

Храмова А.А.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Результаты экспериментов 8](#_Toc26962567)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Задачей данной лабораторной работы является реализация программы, которая выполняет сортировку элементов типа double массива по возрастанию. Для выполнения этого задания используется 4 алгоритма: сортировка вставками, сортировка Шелла, сортировка слиянием и поразрядная сортировка. В ходе исполнения каждого алгоритма необходимо сосчитать число обменов и сравнений, выполненных при сортировке и, опираясь на собранные данные, провести тесты, показывающие теоретическую сложность алгоритма.

# Метод решения

**Сортировка вставками.**

Данная сортировка проста в понимании и реализации, но работает медленно. Входной массив делится на 2 части: отсортированный массив и неотсортированный массив. Сначала отсортированная часть массива пустая. Во время исполнения алгоритма каждый элемент добавляется в отсортированную часть на свою позицию, самый маленький элемент из конца постепенно продвигается на свое место. Из неотсортированной части берется каждый элемент и сравнивается с соседними значениями, таким образом, меняясь местами с элементами большими, чем выбранный, элемент продвигается справа налево на свою позицию поэлементно. Как только неотсортированная часть станет пустой – сортировка кончается.

На первом шаге второй элемент сравнивается с первым: если первый больше второго, то они меняются местами; на втором шаге третий элемент сравнивается сначала со вторым, потом с первым и т. д. Среди уже отсортированных элементов вставляется новый элемент без нарушения порядка.

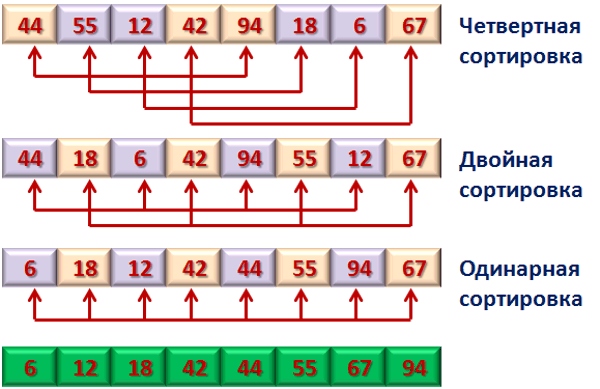


**Сортировка Шелла.**

Является модификацией сортировки вставками – меняется расстояние между сравниваемыми элементами. Если в простом аналоге сортировались соседние элементы, то идея сортировки Шелла в том, чтобы по такому же принципу сравнивать элементы, но с некоторым шагом – на расстоянии.

Входной массив разбивается на несколько подмассивов. Элементы, отстоящие друг от друга на некоторое расстояние d равное шагу, входят в состав одной и той же группы. В каждом подмассиве элементы сортируются вставками, т. е. после выполнения алгоритма с шагом d, рассматривая каждую группу отдельно, будет видно, что она отсортирована должным образом, но только среди отдельных подмассивов, которые составляют целый массив. Далее меняем шаг d и проходим по массиву уже с новым шагом, меньшим, чем был до этого, следовательно массив разбивается на новые подгруппы, которые снова сортируются вставками. Перед тем как шаг станет равным 1, мы получим уже более-менее отсортированный массив, в котором большие по значению элементы стоят в правой части массива, а маленькие соответственно наоборот – вначале, поэтому элементы будут меняться минимальным образом. Когда d = 1 происходит обычная сортировка вставками, но так как массив уже практически отсортирован, элементы быстрее встают на свои места – за меньшее число перестановок.

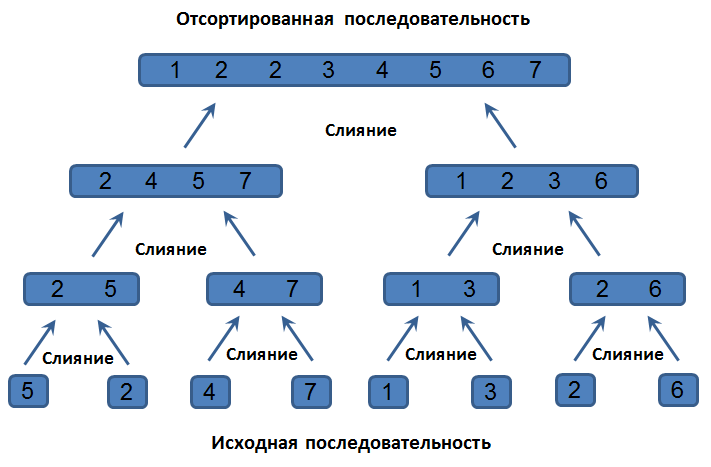
На примере массив разбивается на 4 группы элементов: [44, 94], [55,18], [12,6], [42,67]; шаг равен 4. В каждой из групп элементы упорядочиваются сортировкой вставками и после этого шаг сокращается в 2 раза, а групп становится также 2. Все элементы в двух группах сортируются, после чего шаг = 1 и осуществляется переход к обычной сортировке вставками. В итоге массив полностью отсортирован.



**Сортировка слиянием.**

Алгоритм основан на принципе «разделяй и властвуй», так как основная задача разделяется на маленькие, которые решаются по отдельности, а потом объединяются вместе. Идея состоит в объединении двух отсортированных массивов. Массив разбивается на две части примерно одинакового размера, каждая из которых далее сортируется отдельно той же сортировкой. Затем каждая половина продолжает рекурсивно делиться пополам, пока размер каждого массива не станет равным 1, а такой массив считается отсортированным. Дойдя до конца (когда массив уже невозможно разделить на два) мы сравниваем два этих элемента длины 1 и меньший из них записываем в финальный результат. Так поступаем со всеми подмассивами длины 1. После склейки элементов на предыдущем этапе у нас получаются отсортированные подмассивы длины 2. У двух подмассивов сравниваем первые элементы – меньший записываем в результирующий массив, то же самое проделываем со вторыми элементами. На последнем этапе мы также сравниваем i-тые элементы двух образовавшихся подмассивов и сливаем их в один окончательный. В конце, если один из массивов короче другого, мы просто записываем в конец оставшиеся элементы.

На примере имеем исходный массив = {5,2,4,7,1,3,2,6}. Разбиваем его пополам: {5,2,4,7} и {1,3,2,6}. Каждый делим еще на два, получаем: {5,2} и {4,7}, {1,3} и {2,6}, которые в свою очередь разбиваем на одноэлементные массивы. Сортируем их, получаем последовательности: {2,5} и {4,7}, {1,3} и {2,6}. На следующем шаге сливаем в два массива: {2,4,5,7} и {1,2,3,6}. Уже получившиеся две последовательности сливаем в окончательную отсортированную последовательность чисел: {1,2,2,3,4,5,6,7}.



**Поразрядная сортировка.**

Этот алгоритм достигает большей производительности, чем другие. Особенность его в том, что элементы не сравниваются непосредственно между собой. Сортировка основана на устойчивой версии алгоритма сортировки подсчетом. Для сортировки чисел в любой системе счисления достаточно отсортировать числа, применяя сортировку подсчетом для каждого разряда. У каждого k-значного числа цифра находится в диапазоне от 0 до m-1 (например, в трехзначном десятичном числе k = 3, m = 10). У нас m = 256. Сортировка производится справа налево. Сначала все цифры сортируются по значению крайнего правого разряда. Затем сортируется соседний с ним разряд по тому же принципу, числа упорядочиваются по этому разряду, сохраняя порядок, который был достигнут на предыдущем шаге сортировки. Так сортируются все разряды.

# Руководство пользователя

При запуске программы перед пользователем появляется исходный массив, заданный случайными числами и окно ввода, которое просит выбрать сортировку, с помощью которой пользователь хочет отсортировать массив. Нужно ввести число от 1 до 4.

Изображение выглядит как текст

Автоматически созданное описание

После этого массив будет отсортирован выбранным способом и выведен на консоль. Также будет сделан вывод о корректности выполнения самой сортировки и подсчитано количество перестановок и сравнений, произведенных в ходе работы алгоритма. Все это будет выведено на экран:

Изображение выглядит как текст

Автоматически созданное описание

Чтобы убрать окно необходимо нажать на любую клавишу.

# Описание программной реализации

В проекте содержится 5 файлов: insertion\_sort, mergesort, Shell\_sort, Поразрядная сортировка и all\_sorts. В каждом файле находится сортировка, соответствующая названию своей папки. Внутри этих папок располагаются .cpp файлы, в которых и находится сам код.

**Общие функции:**

void check\_array(double array[], int size) – функция, проводящая проверку массива. На вход принимает сам массив и его размер. Проходит по массиву и сравнивает соседние элементы. Если предыдущий элемент больше, чем следующий за ним, то выводит сообщение о том, что сортировка работает неправильно, в обратном случае – сообщение о том, что сортировка корректна.

**Сортировка вставками.**

В самом начале сегмента объявлены две глобальные переменные comp и swap, при помощи которых будет происходить подсчет сравнений и перестановок соответственно.

Функция void InsertionSort(double array[], int size)

Принимает на вход массив чисел типа double и размер этого массива. Проверяются предыдущий и последующий за ним элементы, то есть сравниваются второй и первый, на следующем шаге третий и второй, второй и первый. На следующих шагах происходит то же самое, пока не заканчивается массив. Если предыдущий элемент больше последующего, то они меняются местами через переменную tmp, которая временно хранит элемент.

**Сортировка Шелла.**

Функция void Shell\_sort(double array[], int size)

Принимает на вход массив и его размер. Переменная step – шаг, меняется от размера массива пополам до минимально возможного, т. е. равного 1. Например, если step = 15, то переменная будет меняться так: 7,3,1 и всего три прохода по циклу. С каждым шагом элементы делятся на подгруппы, когда step = 7, то подгрупп тоже 7, и в каждой из 7 подгрупп элементы сортируются внутри своей подгруппы. Шаг – это расстояние между элементами, когда шаг = 7, то 7 – это расстояние между элементами одной подгруппы. Изобразим на примере, size = 15, step = 7 в отдельные подгруппы будут входить элементы с 0 и 7 индексами (1-ая подгруппа), с 2 и 8 индексами (2-ая подгруппа), со 2 и 9, 3 и 10, 4 и 11, 5 и 12, 6 и 13, 7 и 14. В каждой из этих 7 подгрупп элементы встают на свои места. Далее, step = 3, тогда сортируемые элементы имеют индексы: 0,3,6,9,12 (1-я подгруппа); 1,4,7,10,13 (2-я подгруппа); 2,5,8,11,14 (3-я подгруппа). И так далее, пока массив не будет полностью отсортирован. При шаге =1 будет выполнена обычная сортировка вставками, но алгоритм уже будет работать быстрее, так как большие элементы будут в конце массива, а маленькие вначале, то есть массив более-менее рассортирован и элементы на последнем шаге будут меняться минимальным образом.

**Сортировка слиянием.**

Функция void merge(double a[], int low, int middle, int high)

Принимает исходный массив, нижнюю границу массива (0) и верхнюю (размер массива – 1), а также элемент из середины массива. Внутри функции объявляется дополнительная память – массив b для того, чтобы записывать в него результирующий отсортированный массив, так как того требует алгоритм. Эта функция осуществляет упорядочивание двух частей массива. Пока индекс первого массива не дошел от начала до середины, а индекс второго массива не дошел от середины до конца массива мы сравниваем i-тые элементы первого и второго массива – тот, что меньше записываем в массив b. Если один из индексов дошел до конечной точки раньше другого, значит, что в другом массиве могли остаться элементы, поэтому мы записываем их в конец результирующего массива (так как эти остаточные элементы то же отсортированы).

Функция void Mergesort(double a[], int l, int r)

Принимает сам массив, номер крайнего левого и крайнего правого элементов.

Эта рекурсивная функция разбивает массивы на подмассивы, сначала на 2, потом на 4 и так далее пока не останется по одному элементу. Затем вызывается функция слияния и части сливаются в один результирующий массив.

**Поразрядная сортировка.**

Функция void createCounters(double\* data, long int\* counters, long int N)

Принимает указатель на исходный массив, указатель на массив counters – вспомогательный массив и размер исходного массива.

В этой функции мы создаем вспомогательный массив, который подсчитывает сколько встречается одинаковых значений в массиве, и по индексу этого значения прибавляет 1. Например, массив = {5,3,4,5,5,7,1,4}. Тогда вспомогательный массив будет равен = {0,1,0,1,2,3,0,1,0,0} – мы записываем на 1 место сколько раз в массиве встретился 0, на 2 место – сколько раз встретилась 1 и так далее. В нашем случае числа будут не 10-тичные, 256-ричные.

Функция void radixPass(short int offset, long int N, double\* source, double\* dest, long int\* count)

Принимает номер позиции (offset), число элементов в массиве, исходный массив (source), массив (dest) куда будут записываться числа, отсортированные по байту offset и массив счетчиков, соответствующий текущему проходу.

Находим вспомогательный массив, в котором будет отражаться количество цифр, стоящих перед данной или, другими словами, с какой позиции начинаются числа с этой цифрой. Далее происходит сама сортировка, мы кладем числа по нужной позиции, вычисленной на предыдущем шаге.

Функция void SignedRadixSort(short int offset, long int N, double\* source, double\* dest, long int\* count)

Принимает номер позиции, количество элементов в массиве, массив (dest) куда будут записываться числа, отсортированные по байту offset и массив счетчиков, соответствующий текущему проходу.

Сначала мы подсчитываем количество отрицательных чисел через переменную NumNeg. Первые 128 чисел относятся к положительным числам, мы создаем для них вспомогательный массив, в котором будет отражаться количество цифр, стоящих перед данной. Следующие 128 чисел – отрицательные, для них проделываем то же самое. После этого мы выписываем сами элементы массива на их позиции.

Функция void RadixSort(double\* in, double\* out, long int\* counters, long int N)

Принимает указатель на исходный массив, указатель на вспомогательный массив, длину массива

Эта функция осуществляет проходы от младшего байта к старшему, выполняя окончательную сортировку.

Функция void callradixsort(double\* in, long int N)

Принимает указатель на исходный массив и его длину. Эта функция выделяет память на массивы, с которыми будет работать сортировка и вызывает саму сортировку. Чтобы выполнить сортировку мы вызываем именно эту функцию.

# Подтверждение корректности

Проверкой корректности занимается функция void check\_array(double array[], int size

В отсортированном массиве сравниваются предыдущий и следующий элемент, если предыдущий больше следующего элемента, то сортировка выполнена неправильно и об этом возвращается сообщение. В противном случае – все работает верно, сортировка корректна и об этом также выводится сообщение.

# Результаты экспериментов

Проведем эксперименты для того, чтобы установить сложность алгоритмов.

**Сортировка вставками.**

Сортировка вставками имеет сложность n2, количество сравнений вычисляется по формуле n\*(n-1)/2

Худший случай – исходный массив, отсортированный по убыванию.

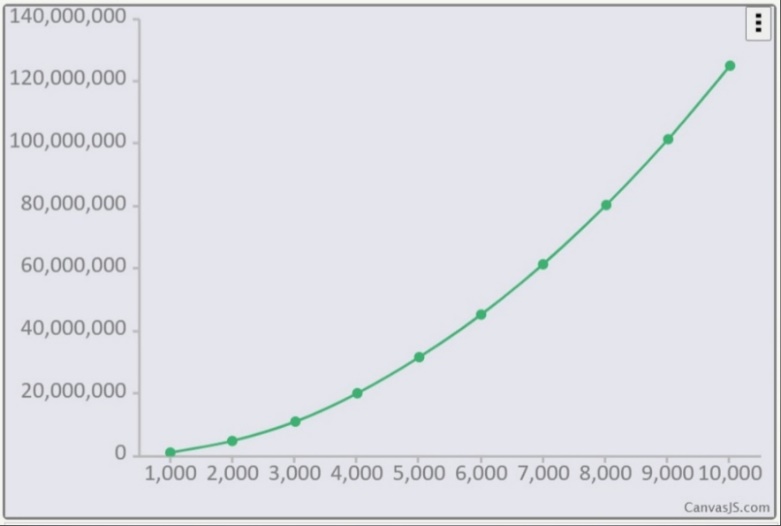
Лучший случай – уже отсортированный должным образом массив.

Пусть массив упорядочен по убыванию, тогда на первом проходе мы поменяем местами элементы 1 раз, на втором проходе мы поменяем элементы местами 2 раза (3-й со 2-ым, 2-ой с 1-ым), на третьем проходе поменяем местами элементы 3 раза (4-й со 3-м, 3-й со 2-м, 2-ой с 1-м) и так далее. Всего нам придется совершить перестановок:

Следовательно, сортировка вставками – квадратичный алгоритм, время его работы пропорционально квадрату от размера массива.

O(n2) сравнений и столько же перестановок.

Построим график сложности:



В 1 столбике то, что выбиралось в качестве длины массива, во втором столбике то, что соответствует этому значению после выполнения сортировки:

1000 1237184

2000 5089699

3000 11308554

4000 20270634

5000 31638734

6000 45471449

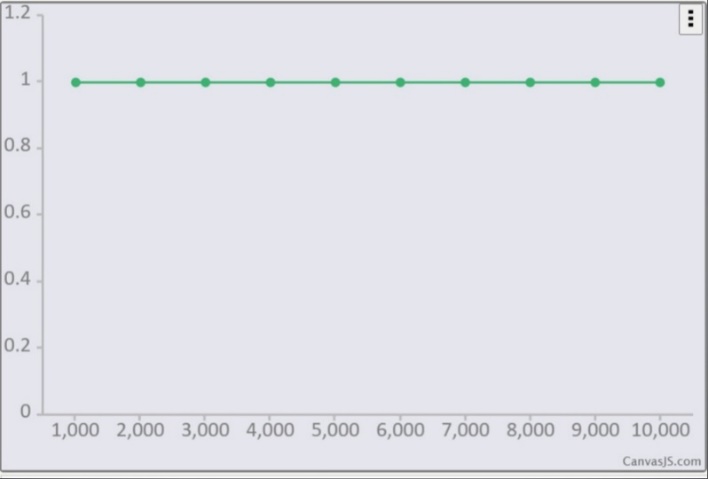
7000 61607604

8000 80634979

9000 101661139

10000 125246974

При делении каждого числа из второго столбика на квадрат длины массива, например 1237184 соответствующего n = 1000 получаем 1,237184



После деления каждого результата второго столбика на n2 получаем константное значение для всех n, что отражено на графике. Так как получаем везде константу, график – прямая линия.

1000 1,237184

2000 1,27242475

3000 1,256506

4000 1,266914625

5000 1,26554936

6000 1,2630958

7000 1,257298

8000 1,259921546875

9000 1,255007579

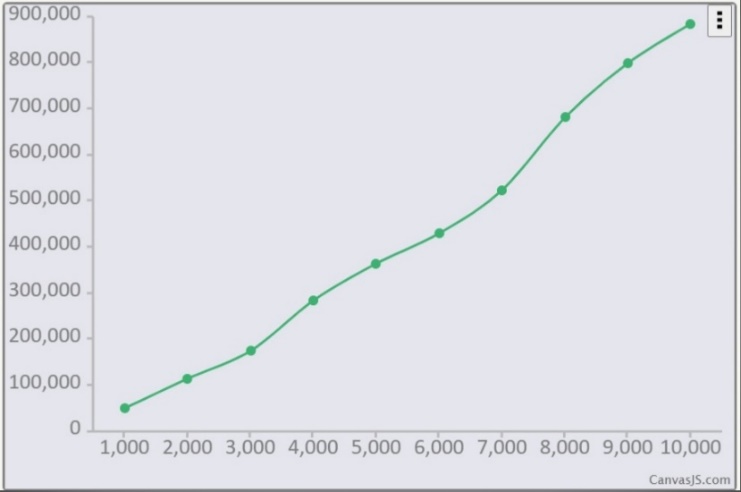
10000 1,25246947

**Сортировка Шелла.**

Сложность алгоритма зависит от выбора шага, который каждый раз уменьшается в два раза. Общее число операций равно:

Худший случай – исходный массив, отсортированный по убыванию.

Лучший случай – уже отсортированный должным образом массив.



В 1 столбике длина массива, во втором - число операций:

1000 49636

2000 112976

3000 176312

4000 283111

5000 364990

6000 429102

7000 522817

8000 681981

9000 800135

10000 884490

При делении каждого числа из второго столбика на получаем всюду константу приблизительно равную 0.

Изображение выглядит как текст

Автоматически созданное описание

Такие значения получаем после деления на :

1000 0,449778

2000 0,46976249

3000 0,4404947796

4000 0,49432793394

5000 0,4834695253

6000 0,45401517

7000 0,45778017

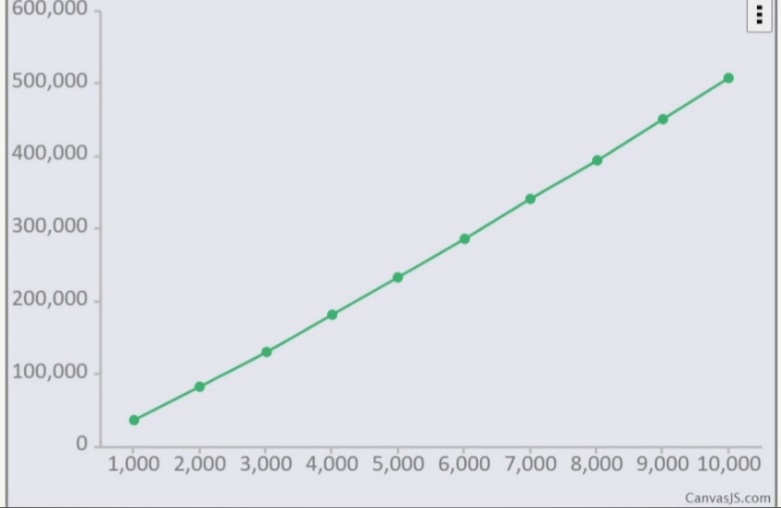
8000 0,50709054

9000 0,515245473

10000 0,5009486698798

**Сортировка слиянием.**

Массив содержит n элементов, после деления пополам каждая из этих двух частей имеет размер n/2, потом мы сливаем их вместе, за 𝑂(𝑛) шагов каждую из половинок можно поделить на две части размером n/4 и затем после сортировки слить их вместе. Аналогично, четыре части размером n/4 за суммарное 𝑂(𝑛) шагов делятся на части размером n/8 и сливаются вместе. Этот процесс продолжается пока размер части не станет равен 1, то есть 𝑙𝑜𝑔2𝑛. Итого, общая сложность этого алгоритма равна 𝑂().



В 1 столбике длина массива, во втором - число операций:

1000 37382

2000 82790

3000 131694

4000 181546

5000 234116

6000 287262

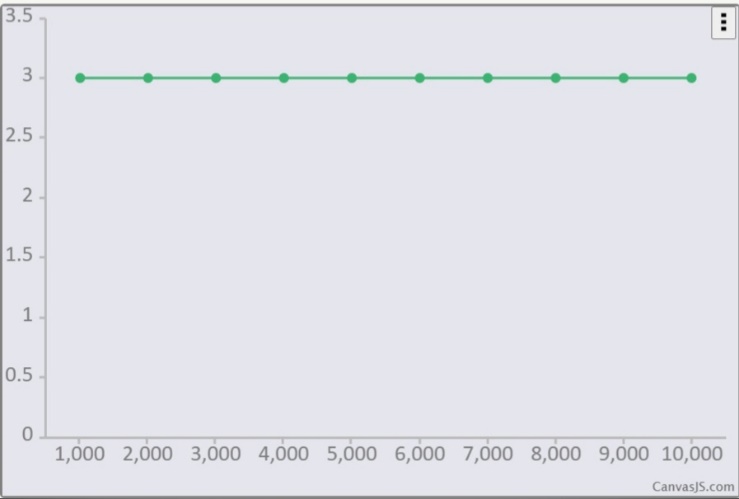
7000 340988

8000 395002

9000 451058

10000 508218

При делении каждого числа из второго столбика на получаем всюду константу приблизительно равную 3.



Точки, полученные после деления на

1000 3,7513329

2000 3,7750216588

3000 3,800473277

4000 3,79303428

5000 3,81060581399

6000 3,814697758

7000 3,81371419

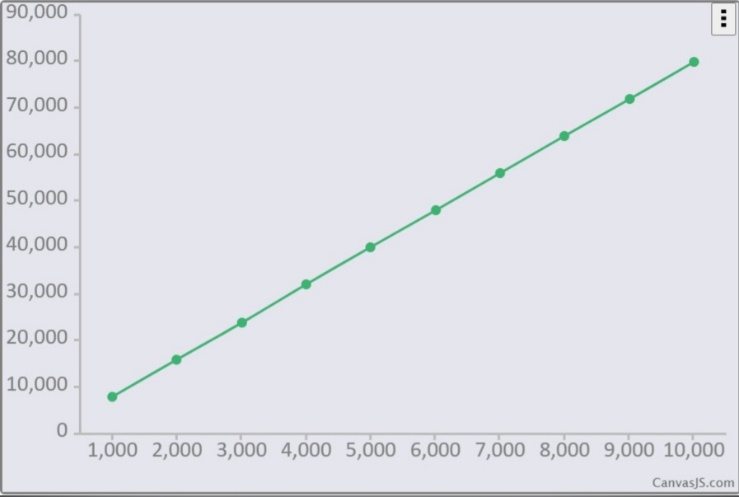
8000 3,8081291093

9000 3,81537967

10000 3,824725121

**Поразрядная сортировка.**

Сложность данного алгоритма O(K\*(N+M)), где K – количество цифр в числе, или число разрядов, M – количество цифр в разряде. Также, по дополнительной памяти алгоритм имеет сложность O(N+M)



По результатам графика можно заметить, что количество операций пропорционально числу элементов в массиве, при n = 1000 количество операций = 8000 и так далее, т. е. сложность может быть получена как O(8\*N) = O(N)

В 1 столбике длина массива, во втором число операций:

1000 8000

2000 16000

3000 24000

4000 32000

5000 40000

6000 48000

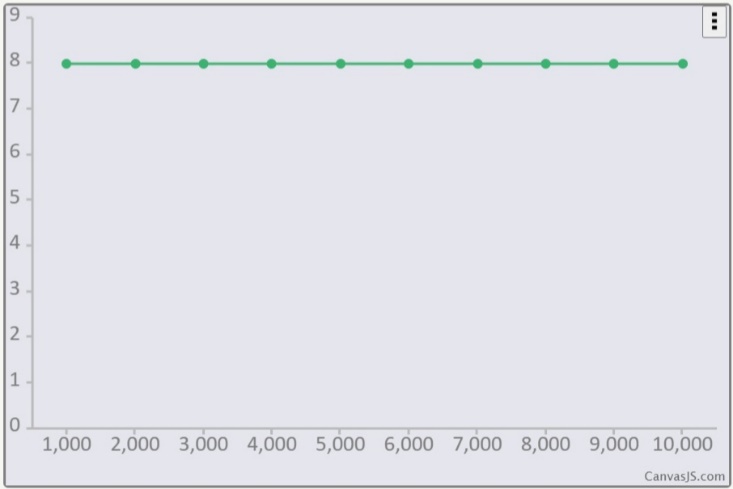
7000 56000

8000 64000

9000 72000

10000 80000

При делении на теоретически заданную сложность = О(N) получаем:



Всем значениям количества элементов в массиве будет поставлено число 8.

# Заключение

Я реализовала четыре алгоритма сортировки чисел, при этом убедилась в том, что каждый из алгоритмов имеет свои особенности и поэтому будет целесообразно использовать различные алгоритмы для решения различных проблем в зависимости от входных данных, ограничений по памяти. Также был проведен анализ сложности каждого алгоритма, что позволило наглядно рассмотреть асимптотическое поведение каждой сортировки и убедиться, что оно совпадает с теоретическим с разницей в константу.

# Приложение

**Сортировка вставками.**

Изображение выглядит как текст

Автоматически созданное описание

**Сортировка Шелла.**

Изображение выглядит как текст

Автоматически созданное описание

**Сортировка слиянием.**

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Поразрядная сортировка.**

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание