`

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

СЕМЕСТРОВАЯ РАБОТА
по дисциплине «Алгоритмы и анализ сложности»
«Экспериментальный анализ различных методов сортировки»

Обучающийся: Лешкевич Сергей Антонович гр. 09–132
(ФИО студента) (Группа)

Руководитель: к.ф.-м.н., доцент КСАИТ, А. В. Васильев

Казань – 2023

Оглавление

Введение

В данной работе будут рассмотрены различные методы сортировки и оценены их сложность на практике.

Рассматриваемые алгоритмы включают в себя простые схемы, сортировку Шелла, быструю сортировку, сортировку слиянием, сортировку кучей, поразрядную сортировку, встроенную в язык программирования сортировку и прочие сортировки.

Оценка будет проводиться на различных входных данных, таких как массивы различных типов, целочисленные массивы, массивы строк, массивы структур разной длины.

Для достоверности результатов методика сравнения будет описана, и проводятся серии испытаний, а не единичные сравнения. Также будет рассмотрены группы сортировок, и проверятся соответствие реализации теоретическим оценкам сложности алгоритмов.

В документе вы найдете причины, которые привели к полученным результатам, а для наглядности будут использоваться диаграммы, гистограммы, и прочую визуализацию. Действительно ли сортировки меньшей сложности будут эффективнее? Это предстоит проверить. Быть может, они эффективны лишь на особых видах выборок? Это и стоит проверить в данной работе.

Подготовка к эксперименту

Чтобы все результаты были достоверными и не были искаженными, необходимо подготовить среду, где будет это все проведено.

Написание и тестирование сортировок будет проходить языке программирования на C++, с параметрами *-O3*, а также выбранной версией стандарта C++11 (2011 года), с использованием компилятора MSVC версии 2019 для Qt 5.15.2.

Для возможности тестирования на системах Linux, предоставляется компилятор *g++* и сборка с флагами *-O3 -std=c++11*

Но в основном, тестирование будет проходить на текущем рабочем устройстве на основе ОС Windows.

Рабочие характеристики устройства, где будут протестированы сортировки:

- Процессор — Intel Core i5 11300H (4 ядра, 8 потоков, штатная частота – 3,1 ГГц, максимальная частота в Turbo-режиме — 4,4 ГГц, Cache L3 — 8 МБ)
- Графика — NVIDIA GeForce RTX 3050 с TGP до 75 Вт и 4 ГБ GDDR6
- Оперативная память — 16 ГБ DDR4-3200
- Хранение данных — SSD на 1024 ГБ
- Операционная система — Windows 11 Pro

Во время тестирования используется максимально минимальное количество ресурсов различными программами и самой ОС для достижения наиболее точного результата.

Методика проведения эксперимента

Реализация функций будет произведена на языке программирования C++. Все выборки будут создаваться динамически, будет три типа выборок – частично восходящие (*ascending*), частично нисходящие (*descending*), случайные значения(*random*) для целочисленных данных, а для строк и структур – только рандомные значения.

Для начала мы будем сравнивать целочисленные данные, используя различные методы сортировки на массивах разных размеров. Чтобы получить полную картину производительности каждого метода сортировки, мы будем использовать массивы размером 50, 5000, 50000 и 500000 элементов. При генерации чисел мы будем использовать числа от 0 до N в возрастающем порядке с частичным рандомайзером, числа от N до 0 в убывающем порядке с частичным рандомайзером и полностью случайные числа. Данные генерируются один раз и повторно переиспользуются через временные переменные.

Генерация происходит в несколько этапов – сначала создаем массивы через собственную реализацию удобных контейнеров на основе векторов (одномерных массивов).

```
typedef std::vector<Number> Container;
```

После инициализации заполняем 3 массива числами по возрастанию, по убыванию и полностью случайные числа. Для первого и второго массива дополнительно применяется случайность дальше, чтобы проверить при частичной сортировке.

```
// инициализировать контейнер на N элементов
// 0,1,2,3,4,...
Container ascending(numElements);
for (size_t i = 0; i < numElements; i++)
  ascending[i] = Number(i);

// ...,4,3,2,1,0
Container descending(numElements);
for (size_t i = 0; i < numElements; i++)
  descending[i] = Number((numElements - 1) - i);

// просто случайные числа
Container random(numElements);
srand(time(NULL));//0);
for (int i = 0; i < numElements; i++)
  random[i] = Number(rand());
```

Дальше мы создаем временный контейнер, который будет использоваться для получения текущего массива и переменные для расчет затраченного времени на сортировку.

```
// используйте этот контейнер для входных данных
Container data;

double timeInverted = 0;
double timeSorted   = 0;
double timeRandom   = 0;
```

Следом, для целочисленных данных я использую следующие сортировки:

- BubbleSort - сортировка пузырьком
- SelectionSort - сортировка выбором
- RadixSort - поразрядная сортировка
- InsertionSort - сортировка вставками
- ShellSort - сортировка Шелла
- QuickSort - быстрая сортировка
- IntroSort - интроспективная сортировка
- HeapSort - сортировка кучей
- MergeSort - сортировка слиянием
- MergeSort (in-place) - сортировка слиянием (in-place)
- std::sort - сортировка из стандартной библиотеки C++
- std::stable_sort – стабильная версия сортировки (собственная)

Для строк я использую те же сортировки, сортировать слова буду в лексикографическом порядке, то есть, по алфавиту. Генерация слов будет на английском, которые создаются случайно. Будут использоваться те же сортировки, что и для целочисленных данных.

Генератор строк я реализовал на языке Python и сохранил результат в виде текстовых файлов формата .txt. Они будут использоваться для считывания данных из них и сортировки в памяти ПК.

Подробнее, как сгенерировать себе файлы и как использовать скрипт описано в Приложении.

Следующей целью будет сортировка структур, а точнее - структуру времени, а структуре будут часы (h), минуты (m), секунды (s).

struct.h:

```
class TestTime
{
        public:
                int h, m, s;

        TestTime(int h, int m, int s)
        {
                this.h = h;
                this.m = m;
                this.s = s;
        }
}
```

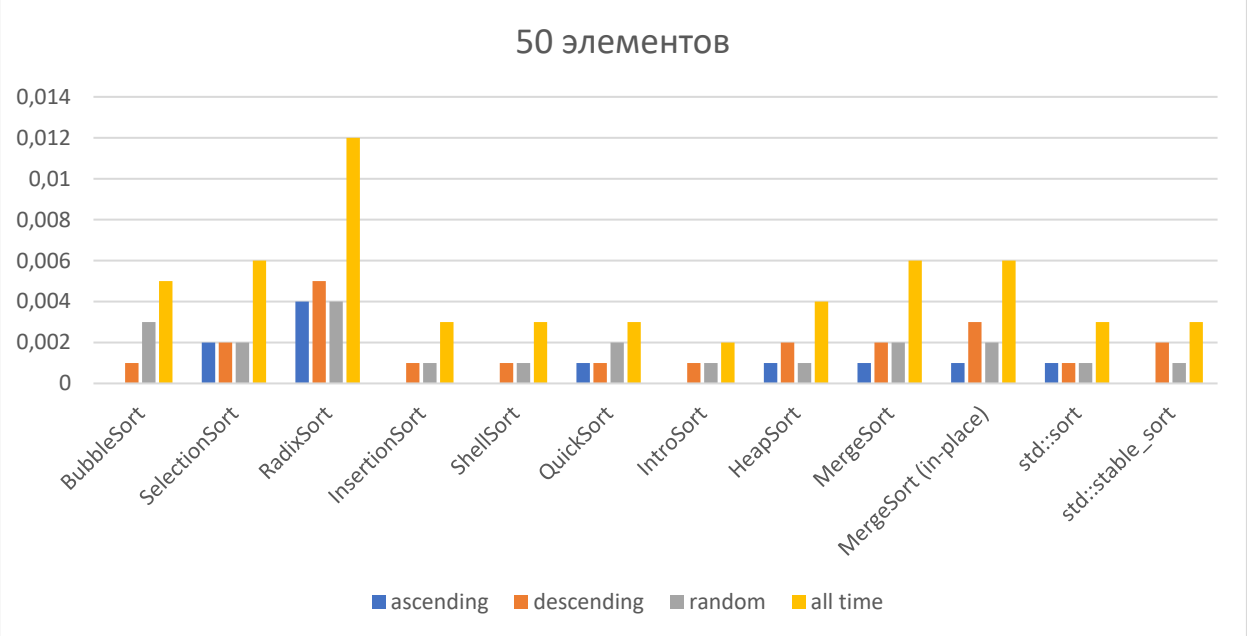
Полученные результаты

Результаты каждого типа исследования представлены в виде гистограмм, где каждая сортировка будет расположена по горизонтали (если время, затраченное на них, разумное для их расположения на гистограмме), а каждый столбец будет показывать, сколько времени было затрачено на данный вид сортировки.

Общие результаты для целочисленных данных

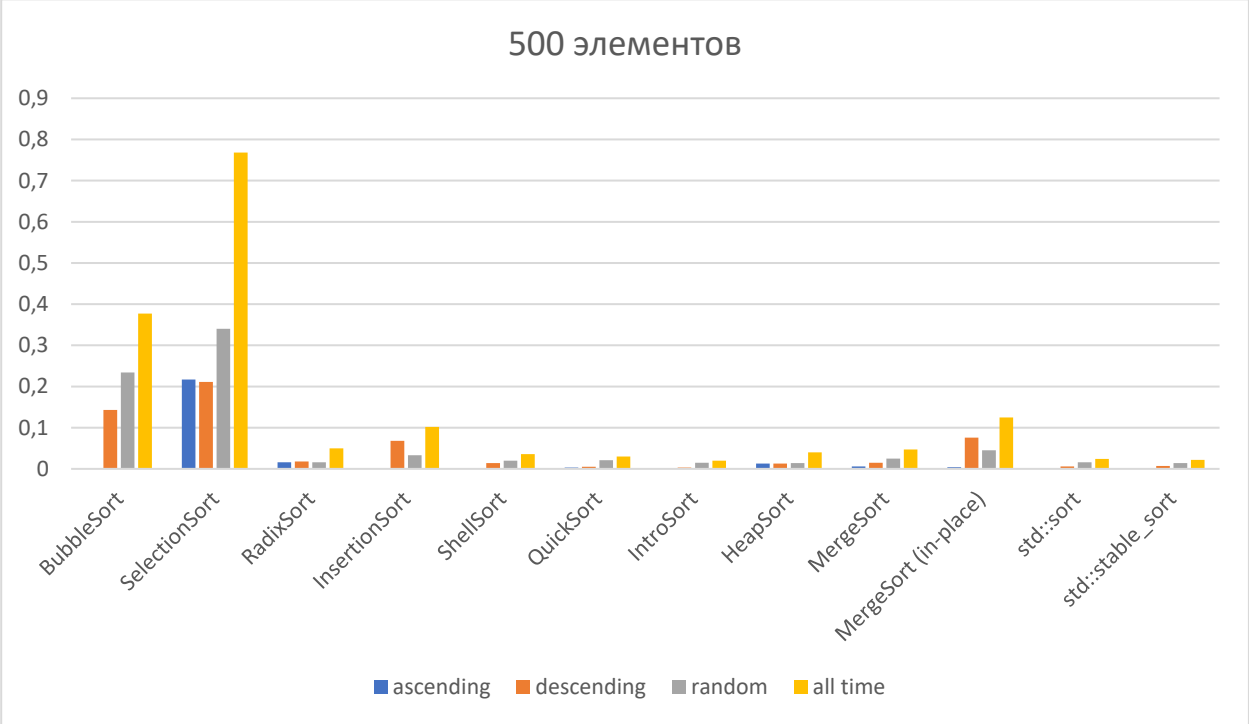
При сортировке 50 элементов данные такие (в мс):

	ascending	descending	random	all time
BubbleSort	0,000002	0,001	0,003	0,005
SelectionSort	0,002	0,002	0,002	0,006
RadixSort	0,004	0,005	0,004	0,012
InsertionSort	0,000002	0,001	0,001	0,003
ShellSort	0,000002	0,001	0,001	0,003
QuickSort	0,001	0,001	0,002	0,003
IntroSort	0,000002	0,001	0,001	0,002
HeapSort	0,001	0,002	0,001	0,004
MergeSort	0,001	0,002	0,002	0,006
MergeSort (in-place)	0,001	0,003	0,002	0,006
std::sort	0,001	0,001	0,001	0,003
std::stable_sort	0,000002	0,002	0,001	0,003



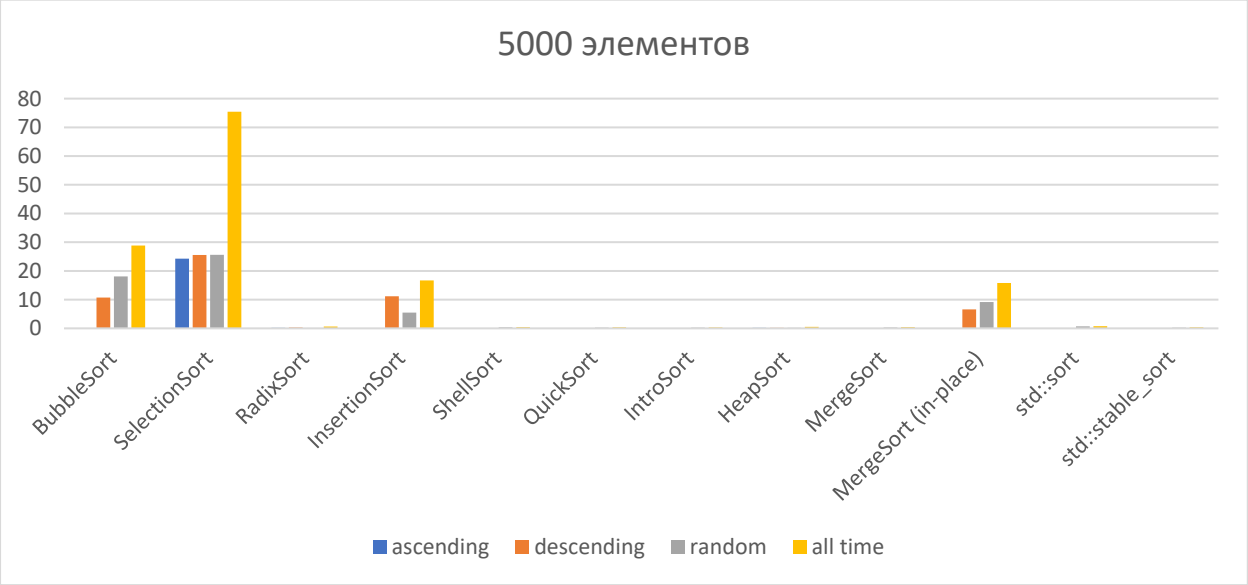
При сортировке 500 элементов данные такие (в мс):

	ascending	descending	random	all time
BubbleSort	**0,000002**	**0,143**	**0,234**	**0,377**
SelectionSort	**0,217**	**0,211**	**0,34**	**0,768**
RadixSort	**0,016**	**0,018**	**0,016**	**0,05**
InsertionSort	**0,000002**	**0,068**	**0,033**	**0,102**
ShellSort	**0,002**	**0,014**	**0,02**	**0,036**
QuickSort	**0,003**	**0,005**	**0,021**	**0,03**
IntroSort	**0,002**	**0,003**	**0,015**	**0,02**
HeapSort	**0,013**	**0,013**	**0,014**	**0,04**
MergeSort	**0,006**	**0,015**	**0,025**	**0,047**
MergeSort (in-place)	**0,004**	**0,076**	**0,045**	**0,125**
std::sort	**0,002**	**0,006**	**0,016**	**0,024**
std::stable_sort	**0,002**	**0,007**	**0,014**	**0,022**



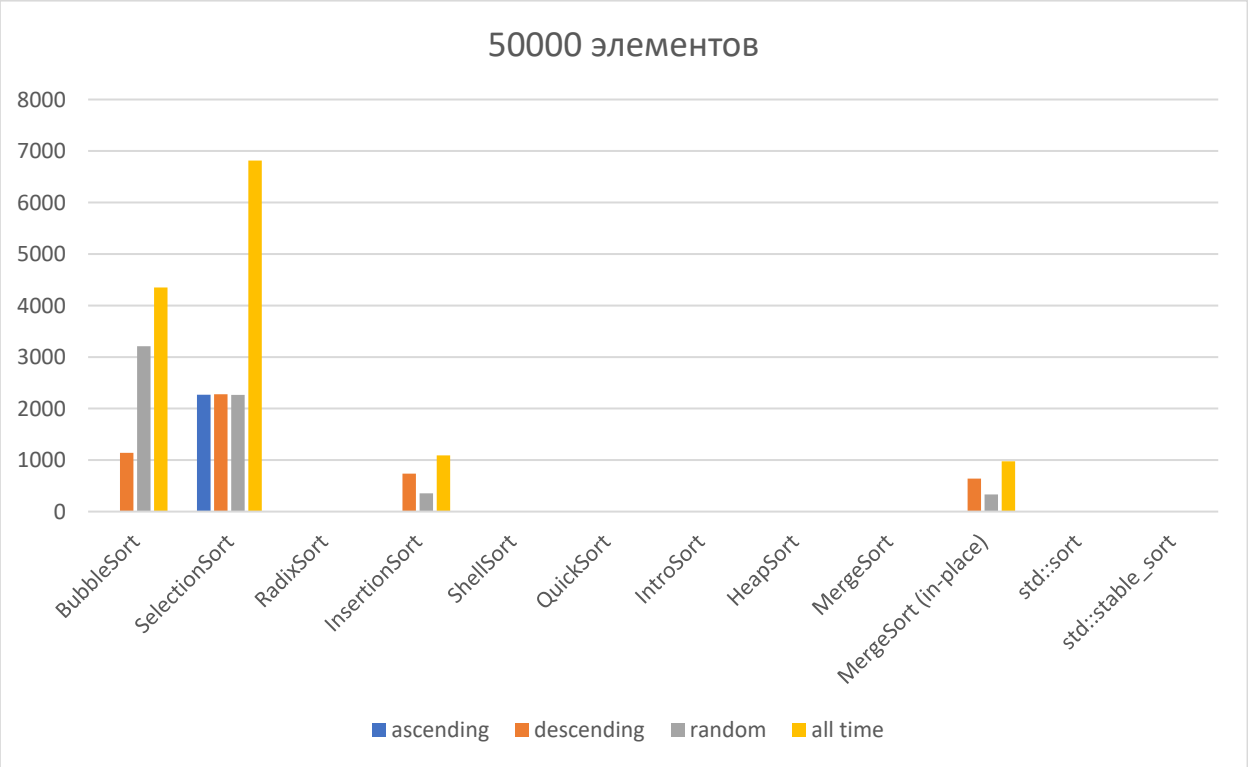
При сортировке 5000 элементов данные такие (в мс):

	ascending	descending	random	all time
BubbleSort	**0,004**	**10,754**	**18,079**	**28,837**
SelectionSort	**24,266**	**25,549**	**25,617**	**75,431**
RadixSort	**0,174**	**0,307**	**0,147**	**0,628**
InsertionSort	**0,009**	**11,201**	**5,489**	**16,698**
ShellSort	**0,028**	**0,07**	**0,345**	**0,444**
QuickSort	**0,036**	**0,051**	**0,252**	**0,34**
IntroSort	**0,03**	**0,036**	**0,232**	**0,297**
HeapSort	**0,171**	**0,171**	**0,19**	**0,531**
MergeSort	**0,052**	**0,097**	**0,281**	**0,43**
MergeSort (in-place)	**0,05**	**6,579**	**9,194**	**15,823**
std::sort	**0,033**	**0,053**	**0,688**	**0,774**
std::stable_sort	**0,035**	**0,081**	**0,266**	**0,383**



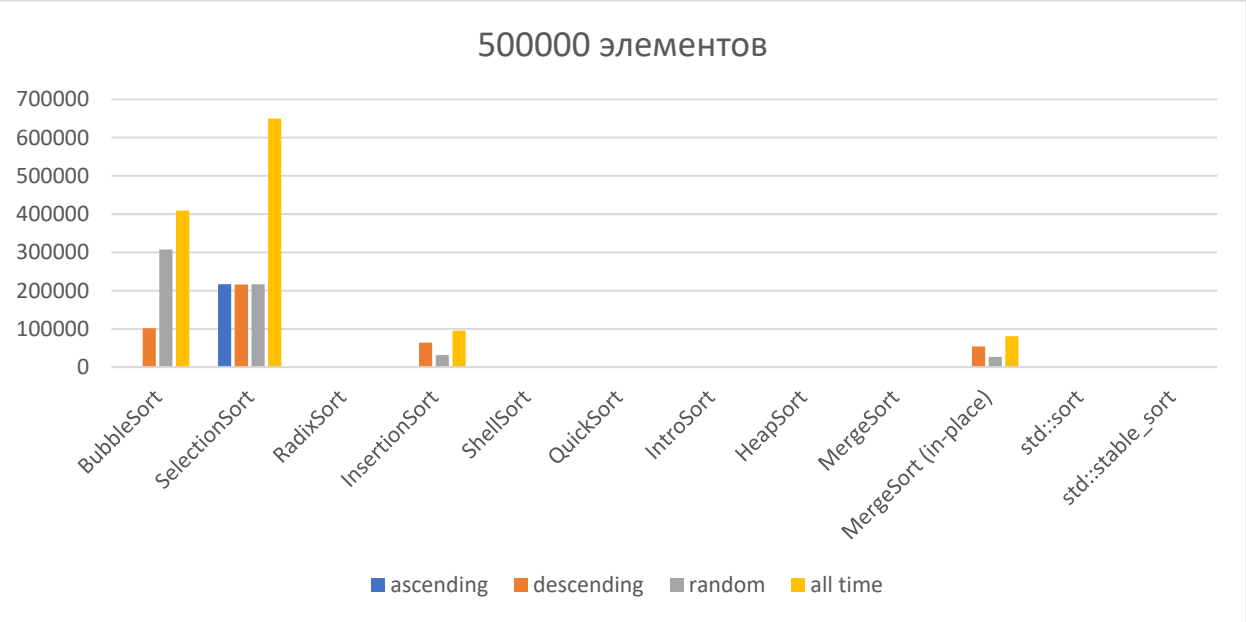
При сортировке 50000 элементов данные такие (в мс):

	ascending	descending	random	all time
BubbleSort	**0,054**	**1139,349**	**3211,129**	**4350,531**
SelectionSort	**2270,165**	**2277,833**	**2266,517**	**6814,515**
RadixSort	**0,686**	**0,781**	**0,53**	**1,997**
InsertionSort	**0,031**	**737,704**	**353,057**	**1090,792**
ShellSort	**0,332**	**0,579**	**4,088**	**4,999**
QuickSort	**0,375**	**0,426**	**2,775**	**3,576**
IntroSort	**0,246**	**0,279**	**2,548**	**3,073**
HeapSort	**1,207**	**1,434**	**1,955**	**4,597**
MergeSort	**0,508**	**0,941**	**3,361**	**4,81**
MergeSort (in-place)	**0,365**	**641,092**	**333,394**	**974,851**
std::sort	**0,411**	**0,599**	**3,928**	**4,938**
std::stable_sort	**0,23**	**0,604**	**2,486**	**3,32**



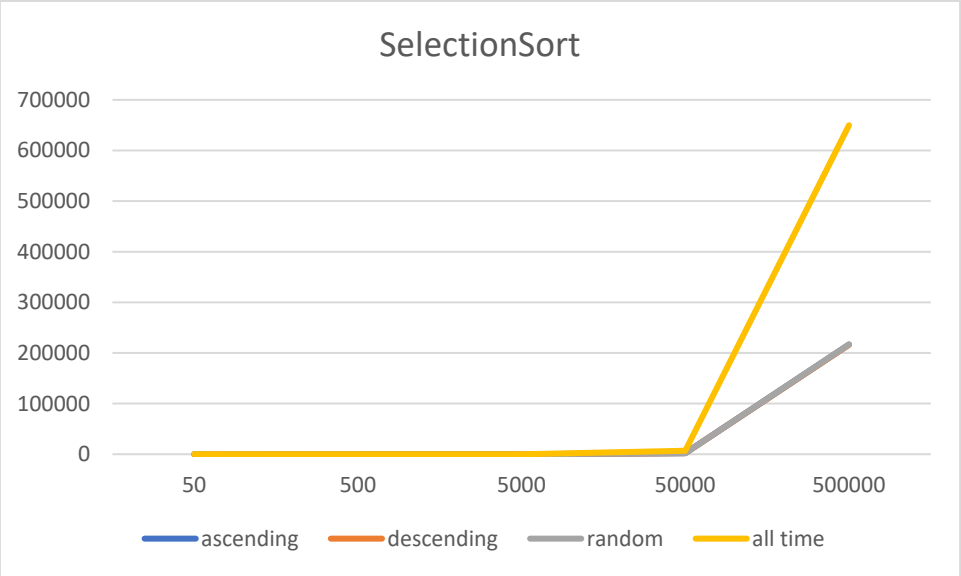
При сортировке 500000 элементов данные такие (в мс):

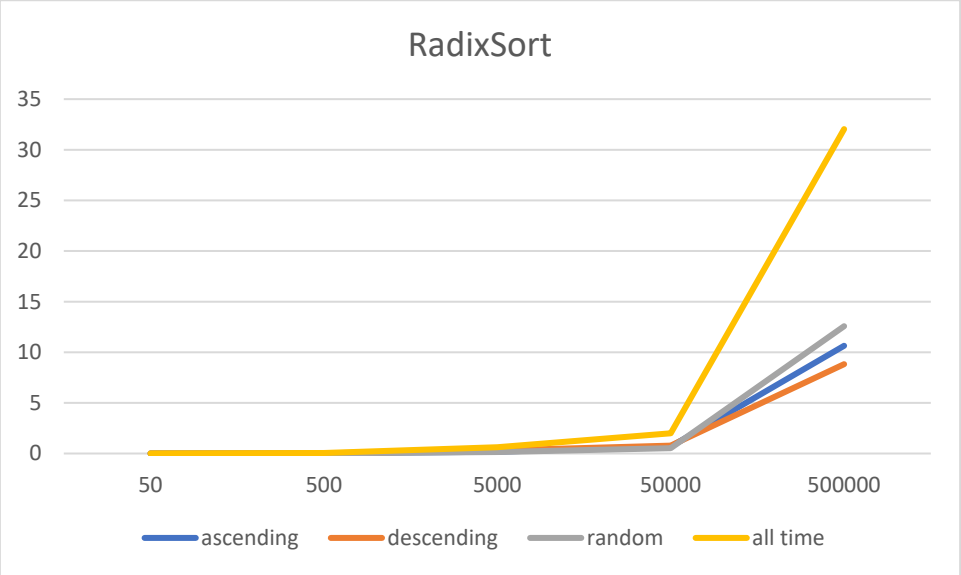
	ascending	descending	random	all time
BubbleSort	0,325	101888,104	307582,453	409470,882
SelectionSort	216640,417	216180,111	217014,021	649834,55
RadixSort	10,642	8,824	12,575	32,041
InsertionSort	0,28	63946,308	31529,642	95476,231
ShellSort	4,221	6,016	45,793	56,031
QuickSort	3,661	4,02	28,168	35,849
IntroSort	2,773	3,571	29,916	36,26
HeapSort	14,287	15,22	39,188	68,695
MergeSort	4,962	9,342	38,263	52,567
MergeSort (in-place)	3,678	54015,425	26913,819	80932,922
std::sort	2,77	5,774	27,542	36,086
std::stable_sort	2,734	5,542	30,829	39,105

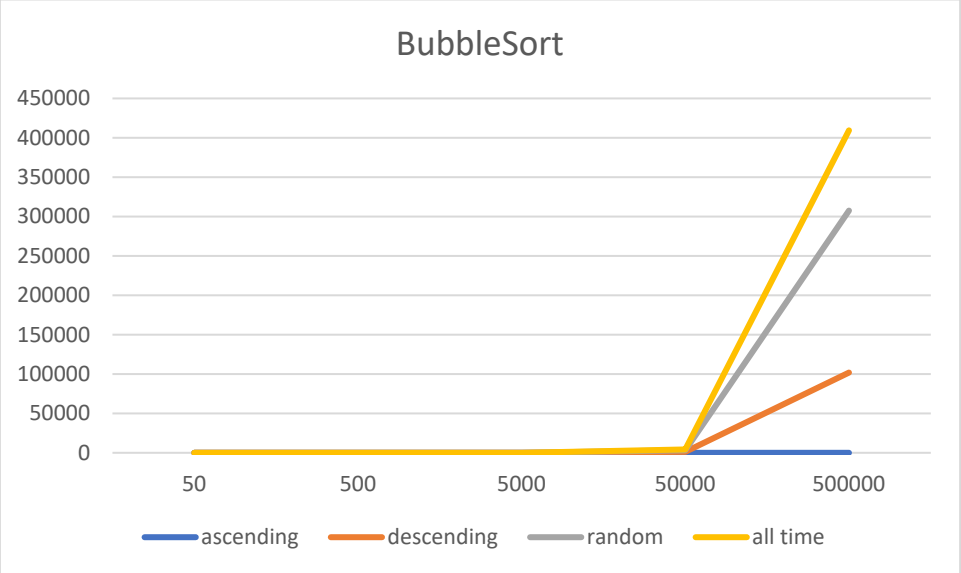


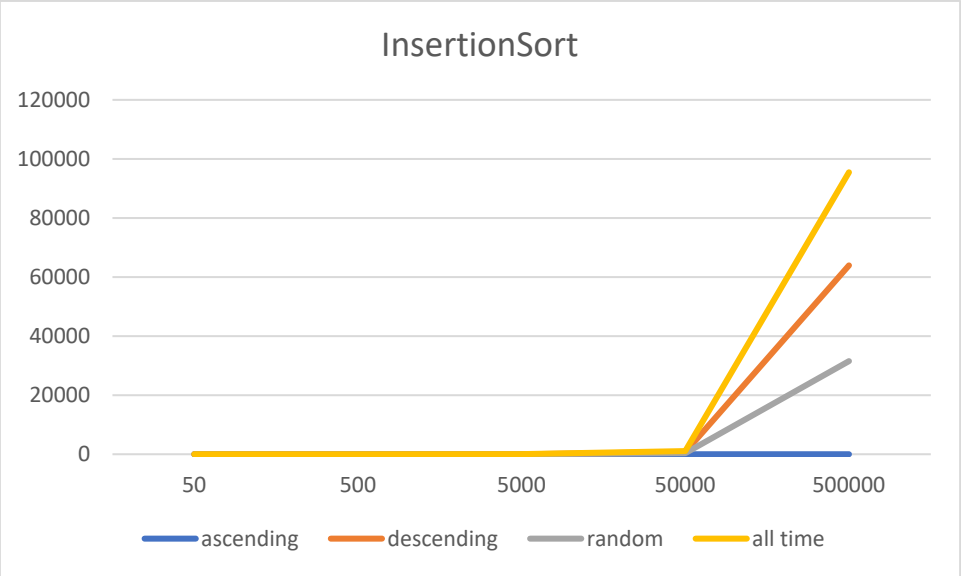
Как мы видим, 4 сортировки из 12 являются самыми медленными и самими ресурсоемкими в сумме всего потраченного времени на все типы массивов.

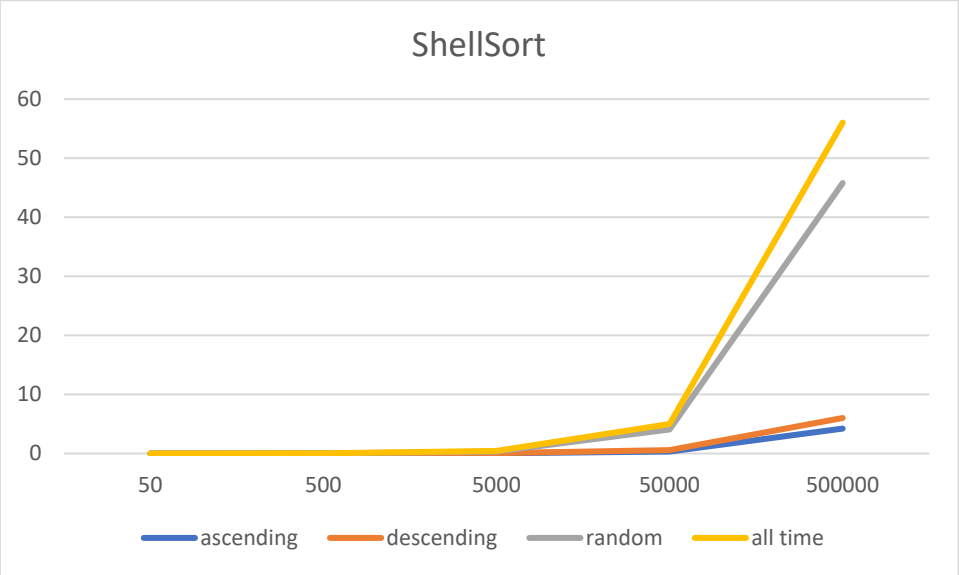
Давайте взглянем, как это меняется с количеством элементов, которые необходимо отсортировать:

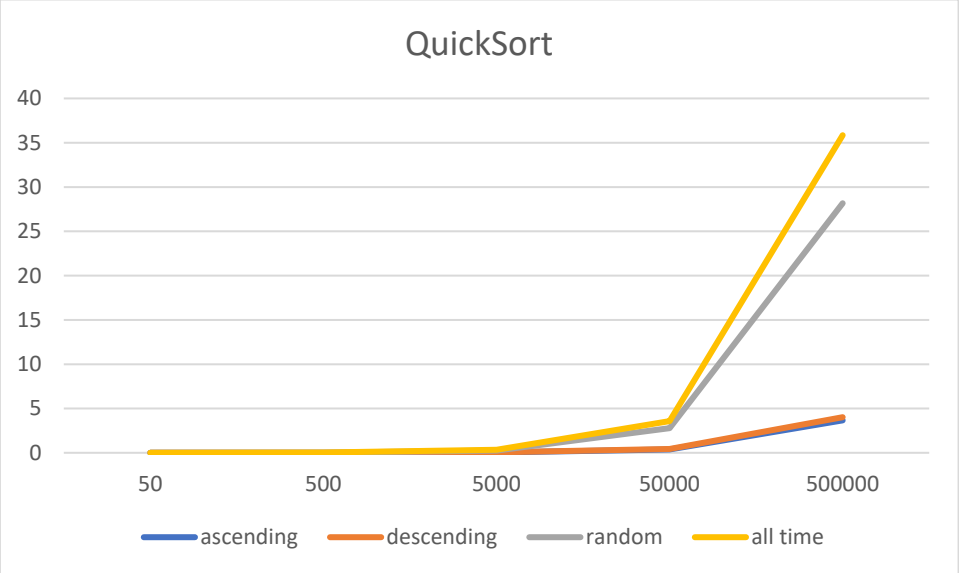

SelectionSort
700000
600000
500000
400000
300000
200000
100000
0
50
500
5000
50000
500000
ascending
descending
random
all time

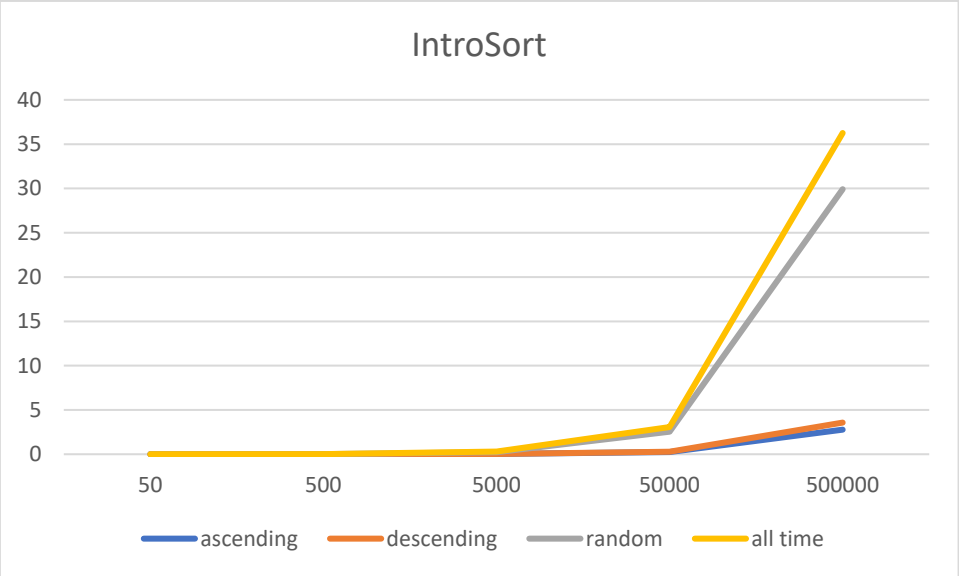

RadixSort
35
30
25
20
15
10
5
0
50
500
5000
50000
500000
ascending
descending
random
all time

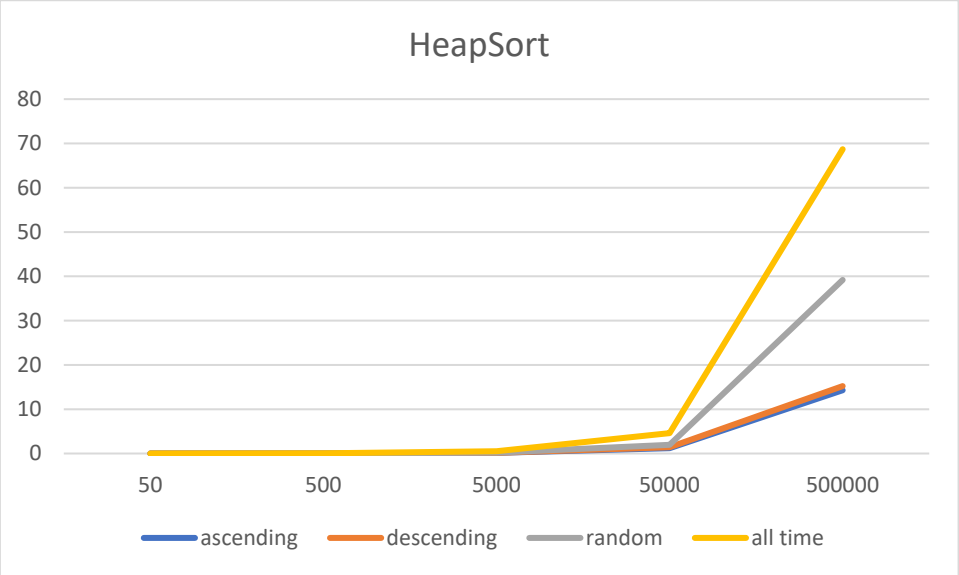

BubbleSort
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
50
500
5000
50000
500000
ascending
descending
random
all time

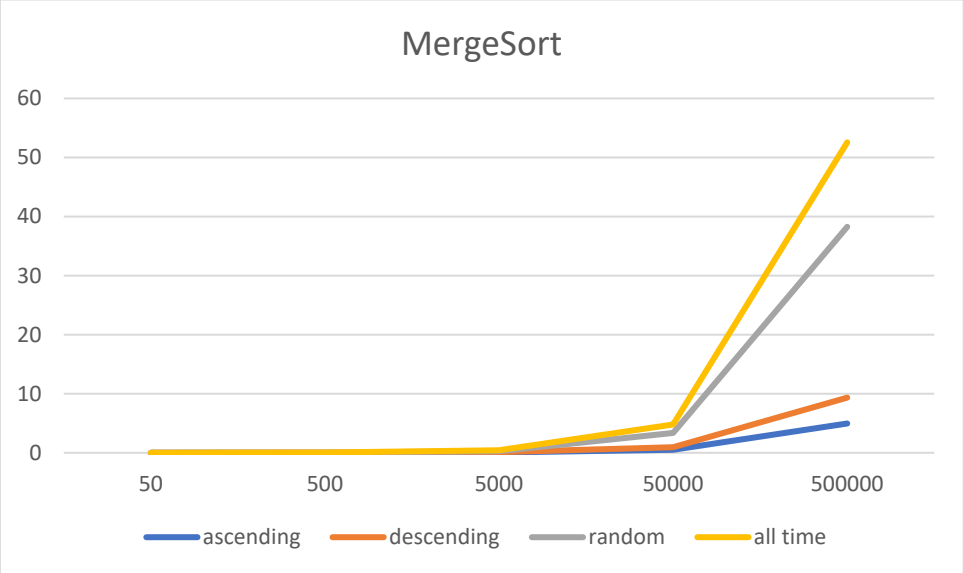

InsertionSort
120000
100000
80000
60000
40000
20000
0
50
500
5000
50000
500000
ascending
descending
random
all time

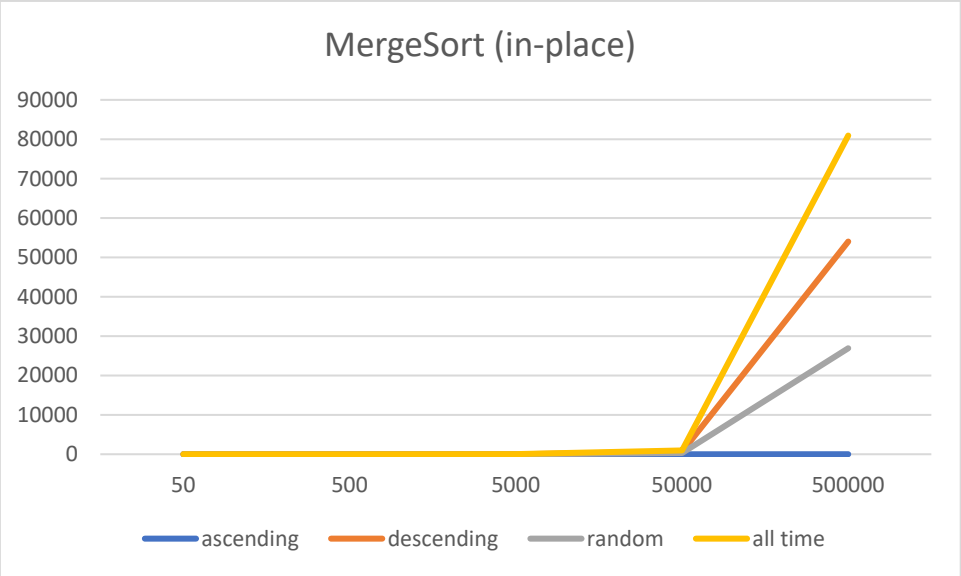

ShellSort
60
50
40
30
20
10
0
50
500
5000
50000
500000
ascending
descending
random
all time

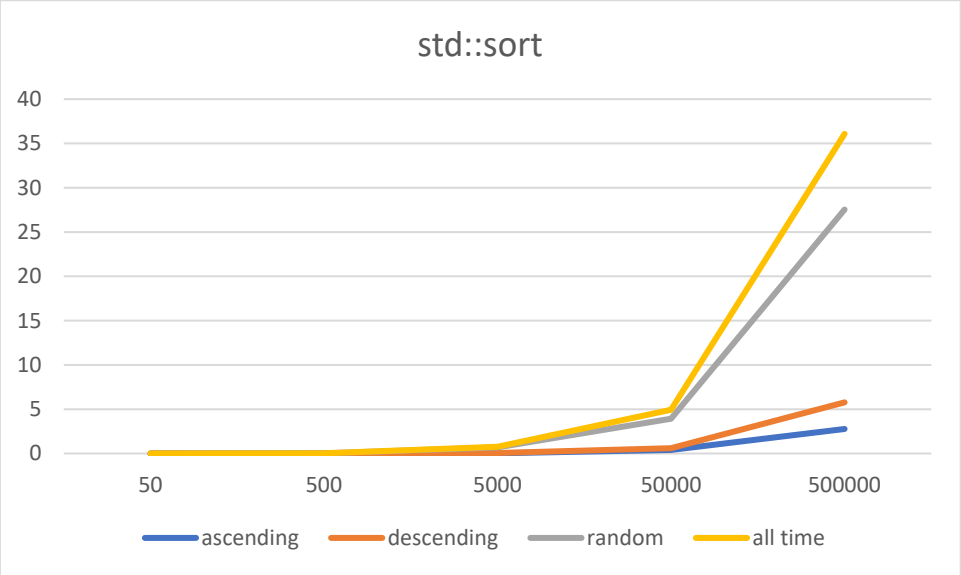

QuickSort
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000
ascending
descending
random
all time

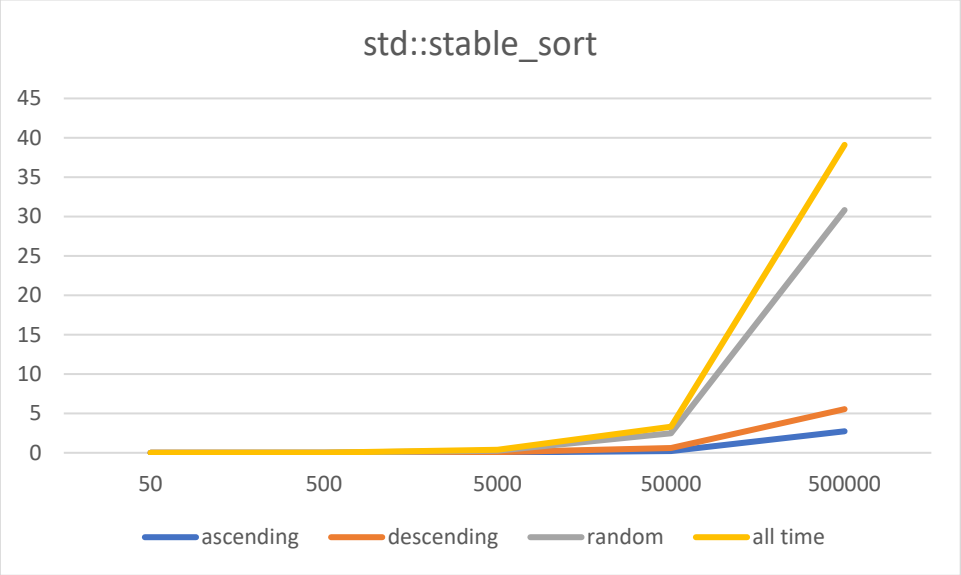

IntroSort
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000
ascending
descending
random
all time


HeapSort
80
70
60
50
40
30
20
10
0
50
500
5000
50000
500000
ascending
descending
random
all time


MergeSort
60
50
40
30
20
10
0
50
500
5000
50000
500000
ascending
descending
random
all time


MergeSort (in-place)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
50
500
5000
50000
500000
ascending
descending
random
all time


std::sort
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000
ascending
descending
random
all time


std::stable_sort
45
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000
ascending
descending
random
all time

Самая первая сортировка, которая показывает худшие результаты среди других - сортировка пузырьком (BubbleSort), которая относится к категории сортировок со сложностью O(n^2), где n — это количество элементов для сортировки. При увеличении количества элементов сложность алгоритма экспоненциально растет, что приводит к замедлению работы алгоритма. Это означает, что с увеличением количества элементов сортировка пузырьком становится крайне неэффективной. Кроме того, сортировка пузырьком имеет большую временную сложность, из-за чего она не рекомендуется для сортировки больших массивов, что и мы видим на деле.

Как мы видим, для InsertionSort относится к категории сортировок со сложностью O(n^2). Это означает, что время выполнения алгоритма резко увеличивается при росте количества элементов для сортировки. В отличие от других сортировок со сложностью O(n^2), InsertionSort имеет дополнительный недостаток: он очень чувствителен к начальному порядку элементов в массиве. Если массив уже почти отсортирован, то InsertionSort будет очень эффективным. Однако, если массив полностью разбросан, InsertionSort будет крайне неэффективным, что и мы видем, что для первого и второго типа массивов он крайне эффективно работает.

Для MergeSort (in-place) — это вариант сортировки слиянием, который выполняется на месте, то есть не использует дополнительную память для создания новых массивов. Хотя алгоритм в целом имеет сложность O(n log n), он может быть крайне неэффективным при больших массивах. Проблема заключается в том, что алгоритм сортировки слиянием требует выделения временного места под сортируемый массив. В случае MergeSort (in-place), это место выделяется прямо внутри массива. При выполнении больших сортировок это может привести к нехватке памяти, а также к замедлению работы алгоритма из-за ограниченности памяти. Поэтому при увеличении количества элементов может стать крайне неэффективной.

Ну и последняя сортировка, которая плохо показывает при больших массивах - сортировка выбором (SelectionSort) становится крайне неэффективной из-за своей квадратичной временной сложности O(n^2). Суть алгоритма заключается в выборе наименьшего элемента в пределах неотсортированной части массива и помещении его в отсортированную часть. При больших объемах данных количество сравнений и перемещений резко возрастает, что вызывает замедление работы алгоритма.

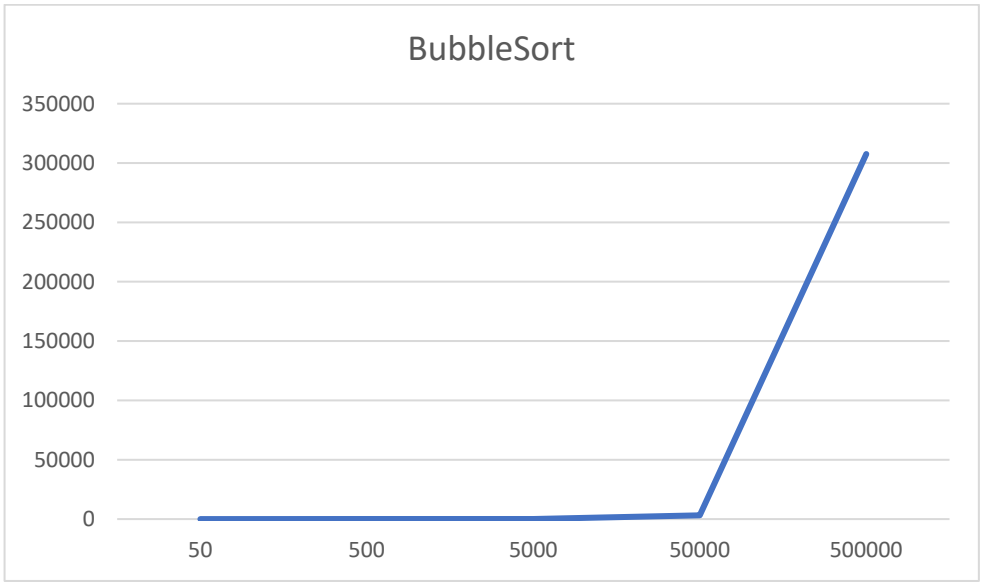
Кроме того, сортировка выбором неустойчива, что может привести к неправильной сортировке эквивалентных элементов, что можно проверить, собрав программу с параметром *-DCHECKRESULT,* которая реализована для проверки на правильность сортировки.

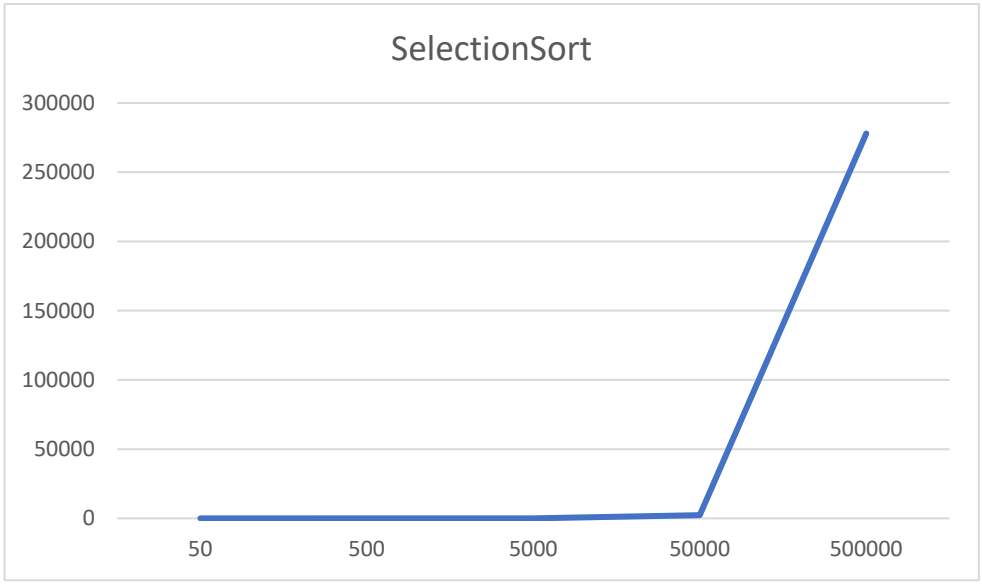
А вот сортировка, реализованная в виде std::sort является стандартным методом сортировки, встроенным в большинство языков программирования. Она использует быструю сортировку (QuickSort), но имеет ряд улучшений, что приводит к ее высокой эффективности. Во-первых, она использует различные стратегии определения опорных элементов, чтобы минимизировать время на выбор опорного элемента. Во-вторых, она использует гибридный подход, который переключается на другие методы сортировки (в том числе на сортировку вставками) для сортировки маленьких подмассивов. Это позволяет быстро сортировать большие массивы, а также ускоряет сортировку массивов с небольшим количеством элементов. Кроме того, std::sort использует встроенную оптимизацию компилятора, что способствует еще большей эффективности алгоритма.
Все эти факторы объединяются, чтобы сделать std::sort одной из самых быстрых сортировок в большинстве сценариев использования.

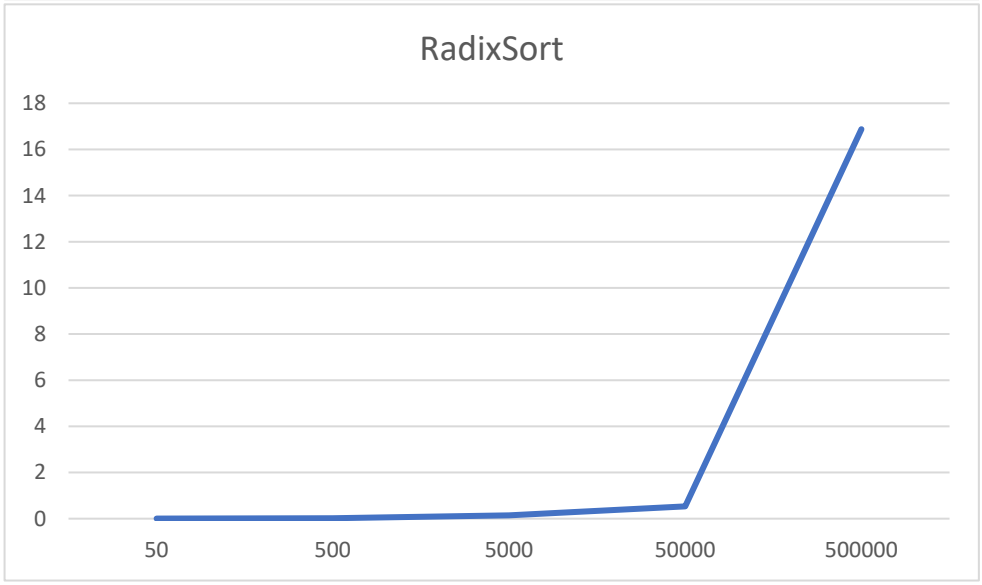
Исходя из данных, представленных выше, можно составить список, где самая первая сортировка – самая лучшая, а последняя - самая худшая:

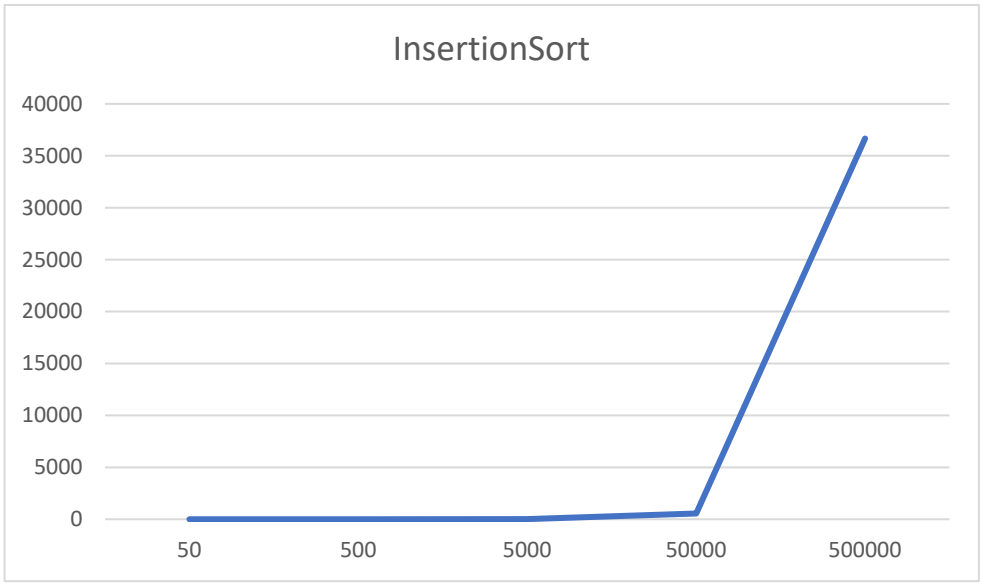
- std::sort - сортировка из стандартной библиотеки C++
- std::stable_sort – стабильная версия сортировки (поверх std версии)
- QuickSort - быстрая сортировка
- IntroSort - интроспективная сортировка
- RadixSort - поразрядная сортировка
- MergeSort - сортировка слиянием
- ShellSort - сортировка Шелла
- HeapSort - сортировка кучей
- MergeSort (in-place) - сортировка слиянием (in-place)
- InsertionSort - сортировка вставками
- BubbleSort - сортировка пузырьком
- SelectionSort - сортировка выбором

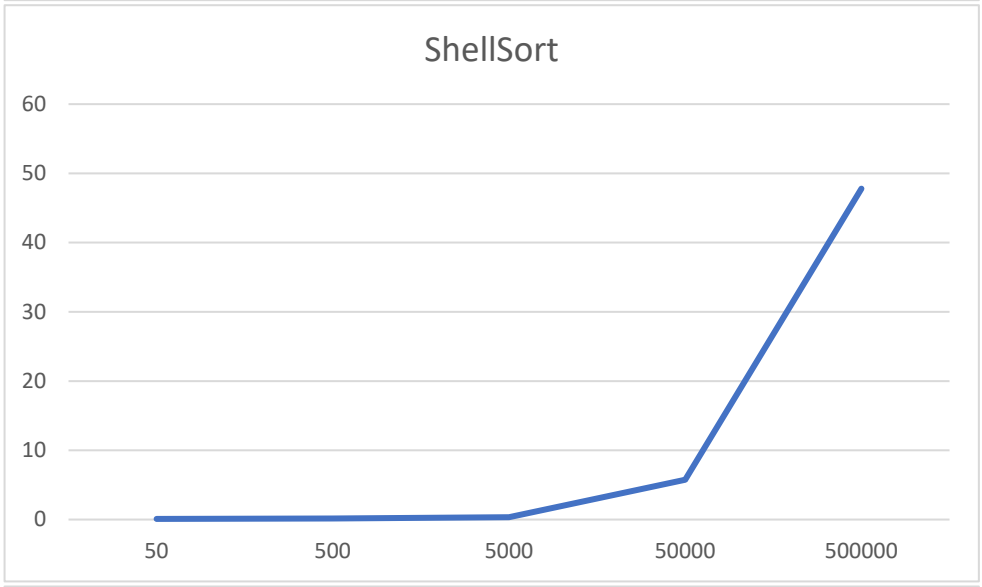
Общие результаты для строк

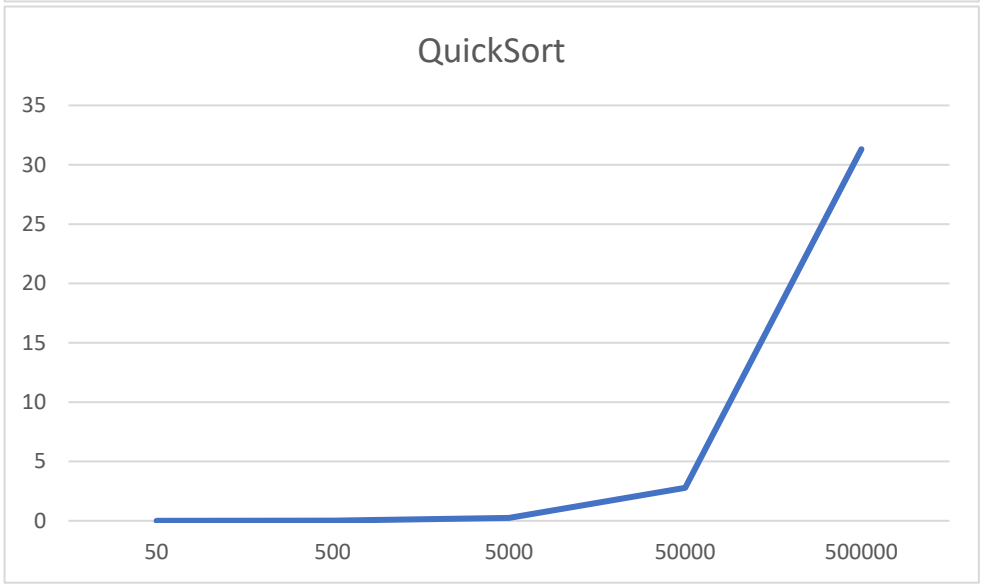


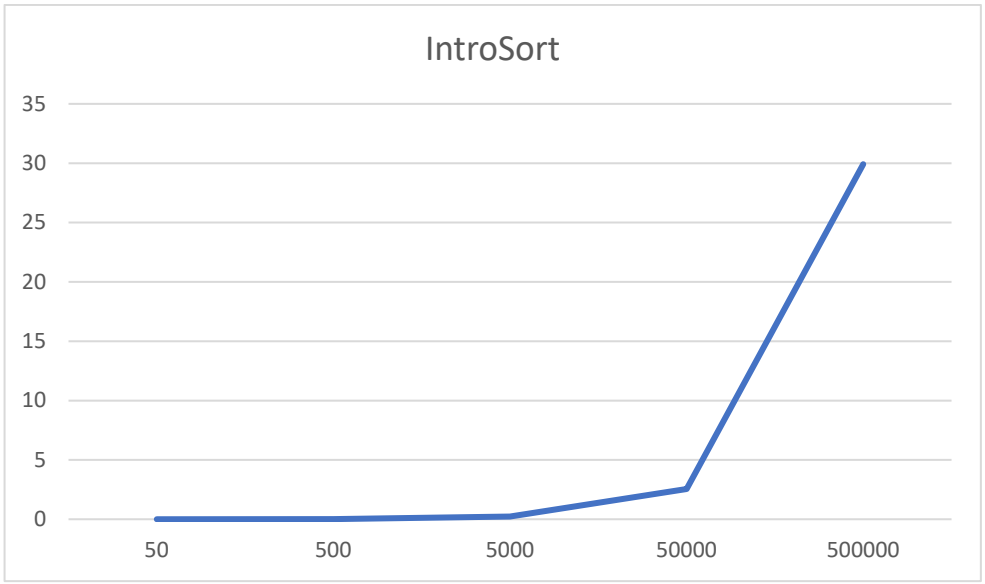


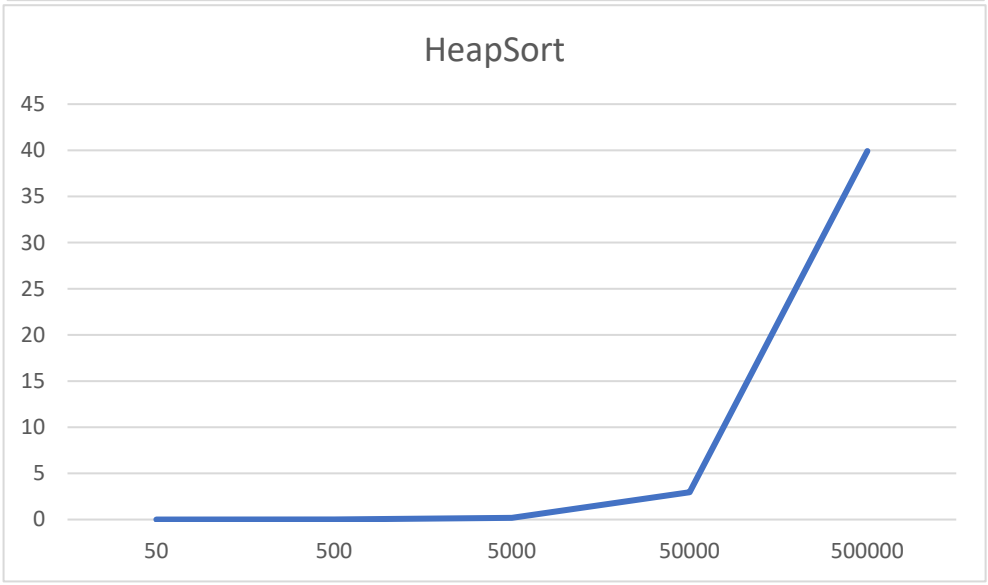


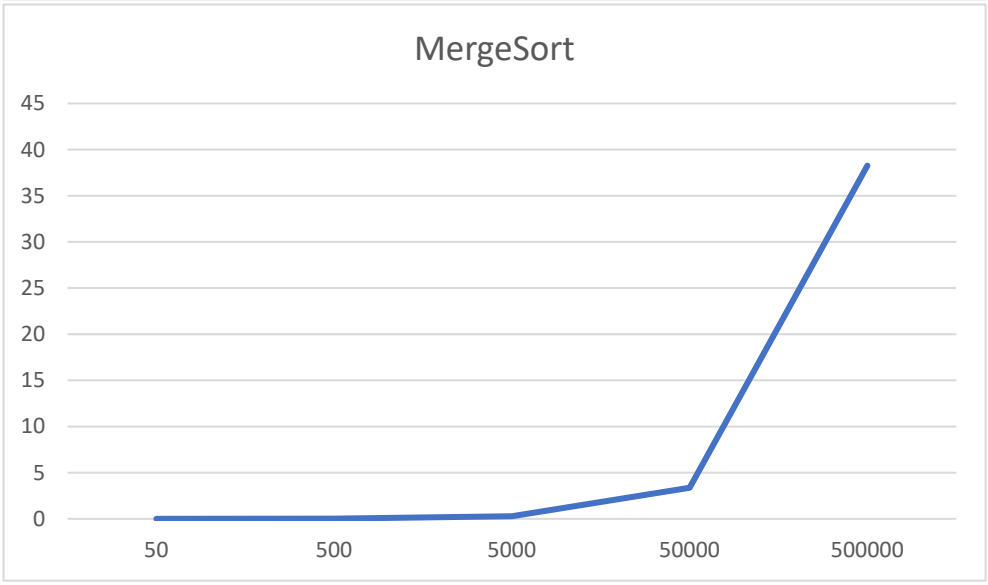

InsertionSort
40000
35000
30000
25000
20000
15000
10000
5000
0
50
500
5000
50000
500000

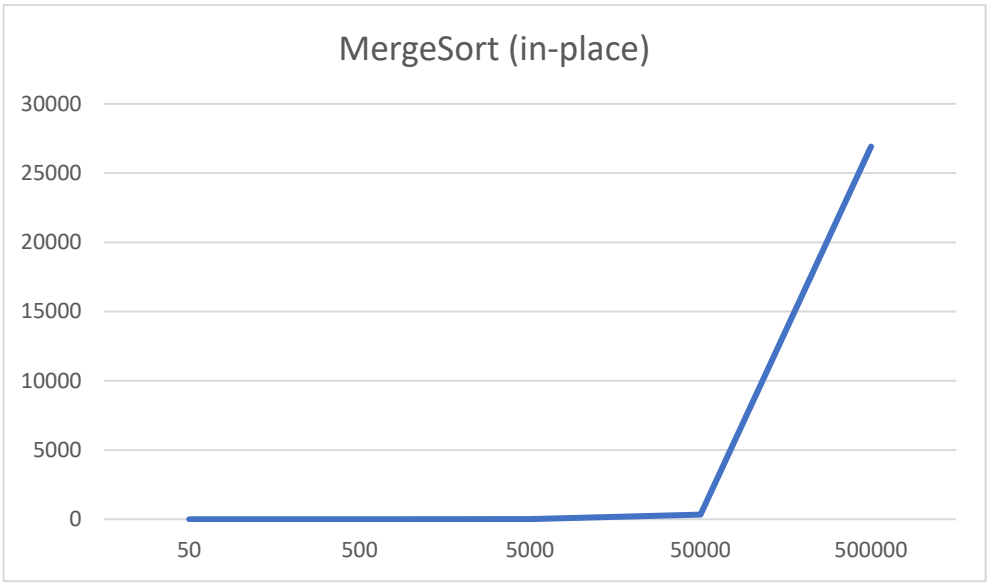

ShellSort
60
50
40
30
20
10
0
50
500
5000
50000
500000


QuickSort
35
30
25
20
15
10
5
0
50
500
5000
50000
500000

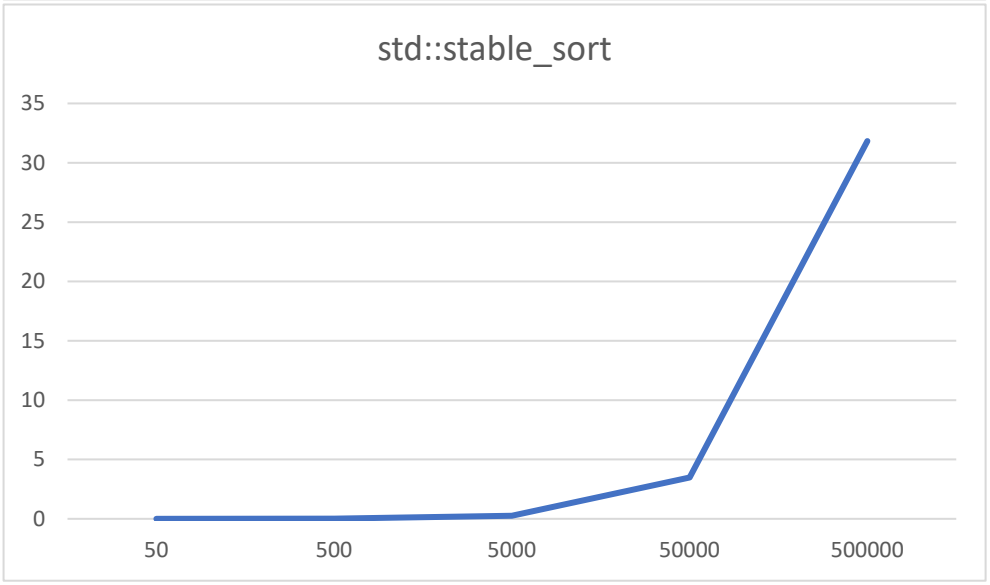

IntroSort
35
30
25
20
15
10
5
0
50
500
5000
50000
500000


HeapSort
45
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000


MergeSort
45
40
35
30
25
20
15
10
5
0
50
500
5000
50000
500000


MergeSort (in-place)
30000
25000
20000
15000
10000
5000
0
50
500
5000
50000
500000

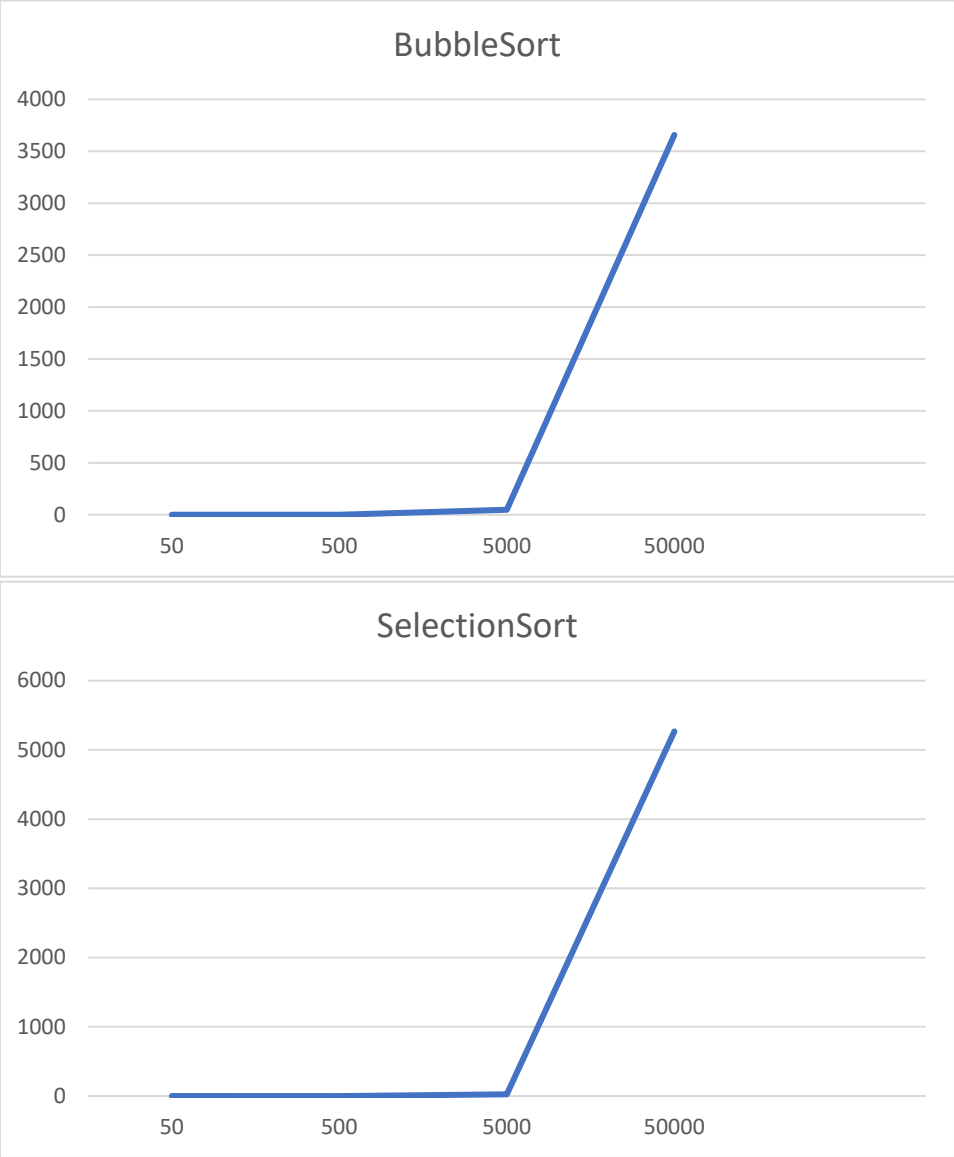

std::sort
30
25
20
15
10
5
0
50
500
5000
50000
500000

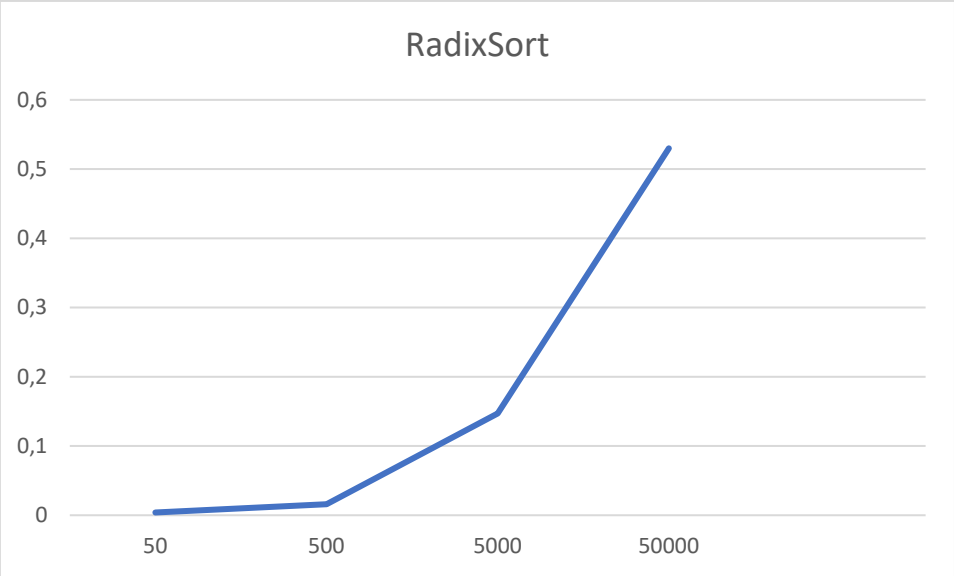

std::stable_sort
35
30
25
20
15
10
5
0
50
500
5000
50000
500000

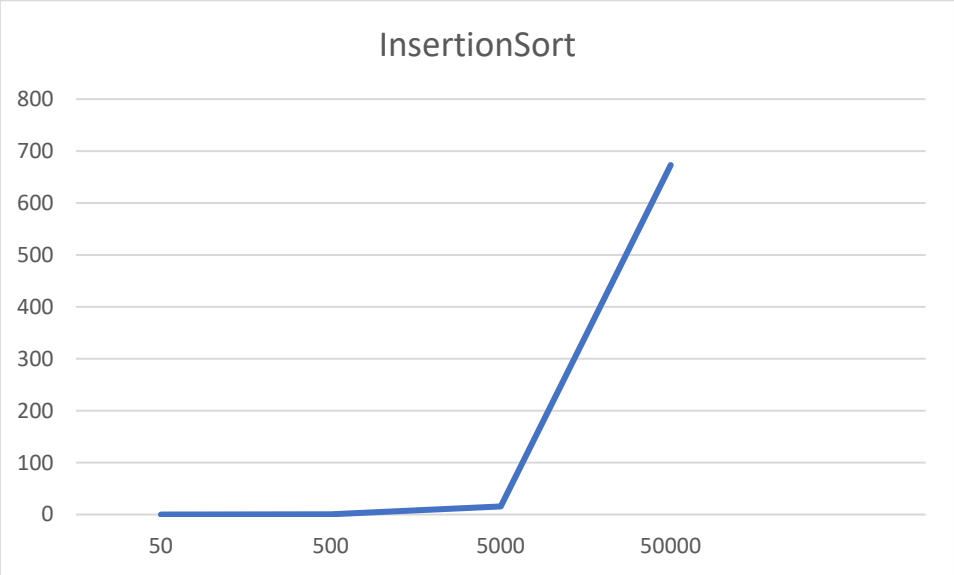
Данные о сортировке строк представлены только для случайных значений, а результаты с частичной сортировкой аналогичны подобны результатам для целочисленных данных.

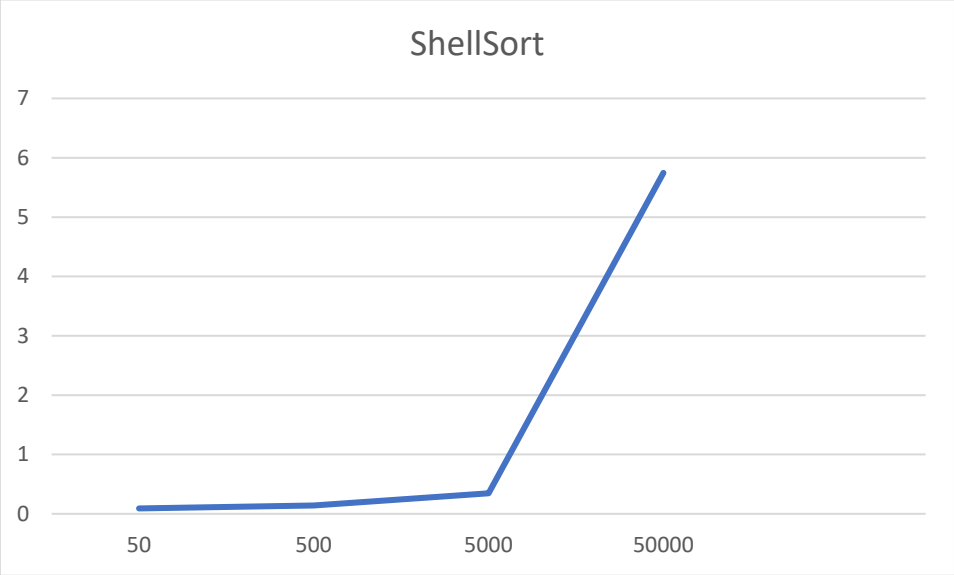
Очевидно, что сортировка слов займет больше времени, чем сортировка чисел. Для сравнения скорости работы разных сортировок важно использовать соотношение скоростей на числах. Стоит отметить, что на больших массивах данных сортировка RadixSort работает быстрее, чем быстрая сортировка. Но опять же, 4 алгоритма сортировки необходимости не требуют из-за своей плохой производительности, так как это нецелесообразно и может потребовать несколько часов времени на обработку того же массива.

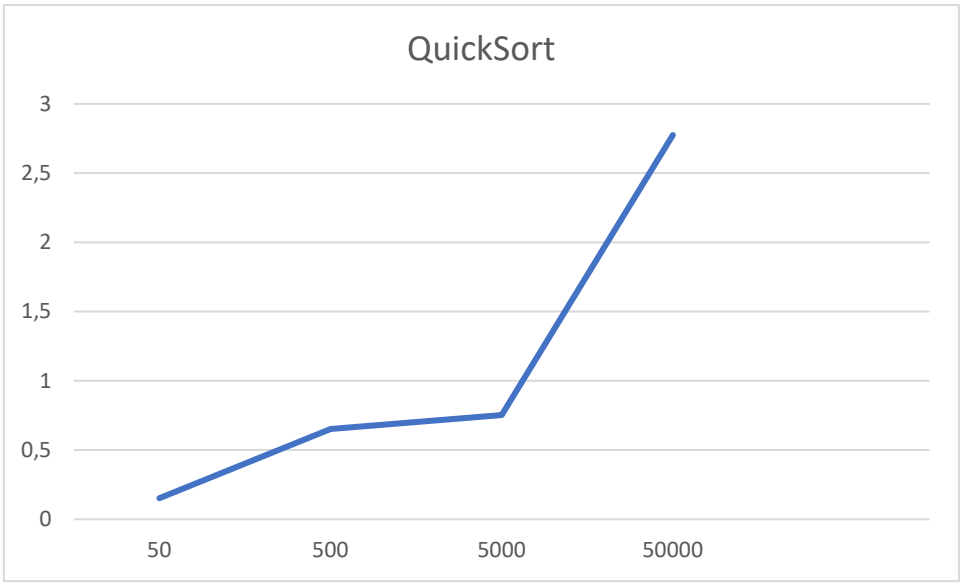
Общие результаты для структур

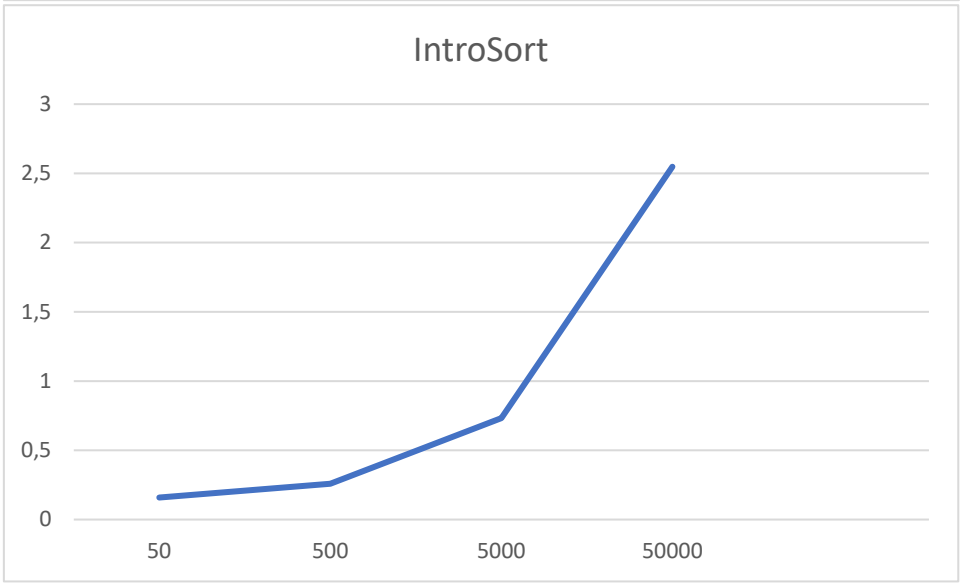


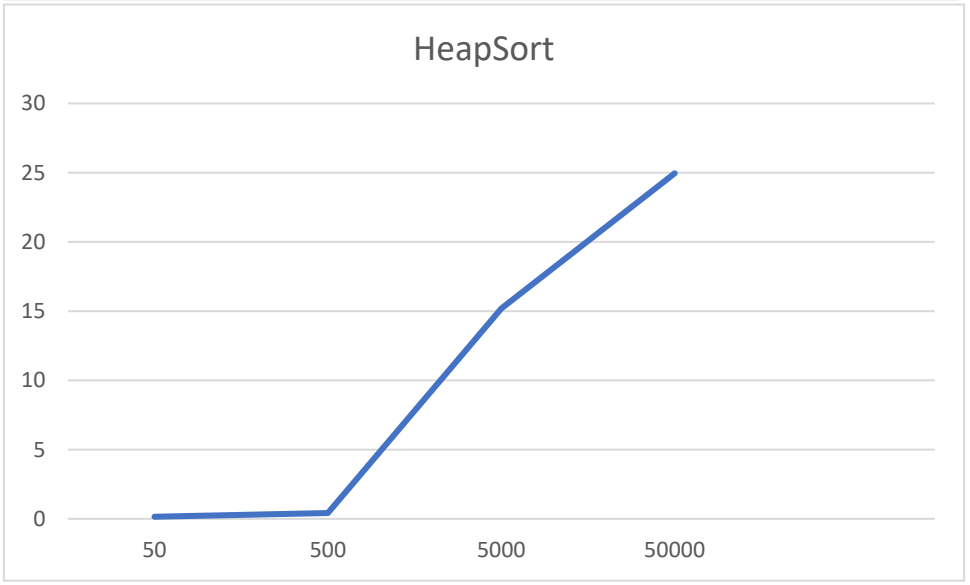

RadixSort
0,6
0,5
0,4
0,3
0,2
0,1
0
50
500
5000
50000

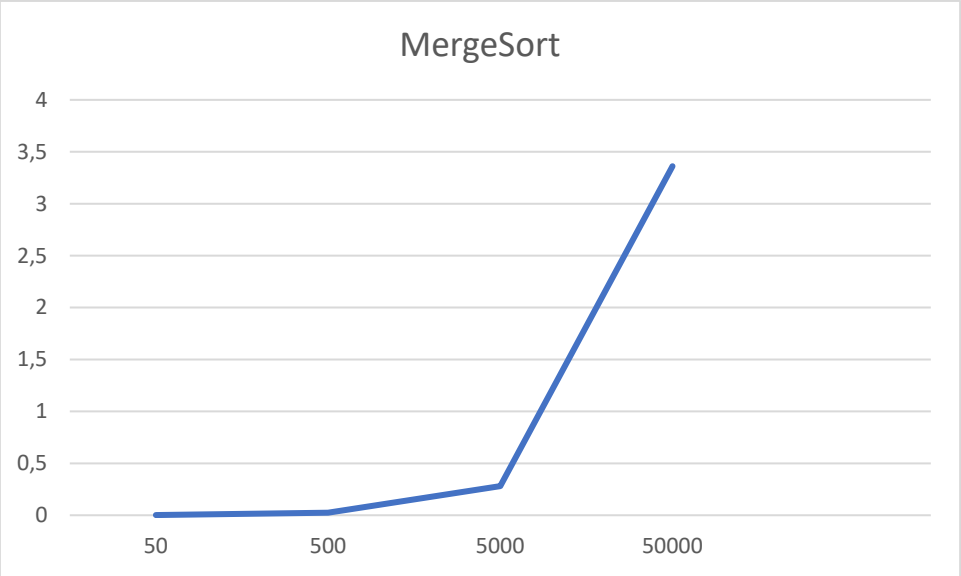

InsertionSort
800
700
600
500
400
300
200
100
0
50
500
5000
50000

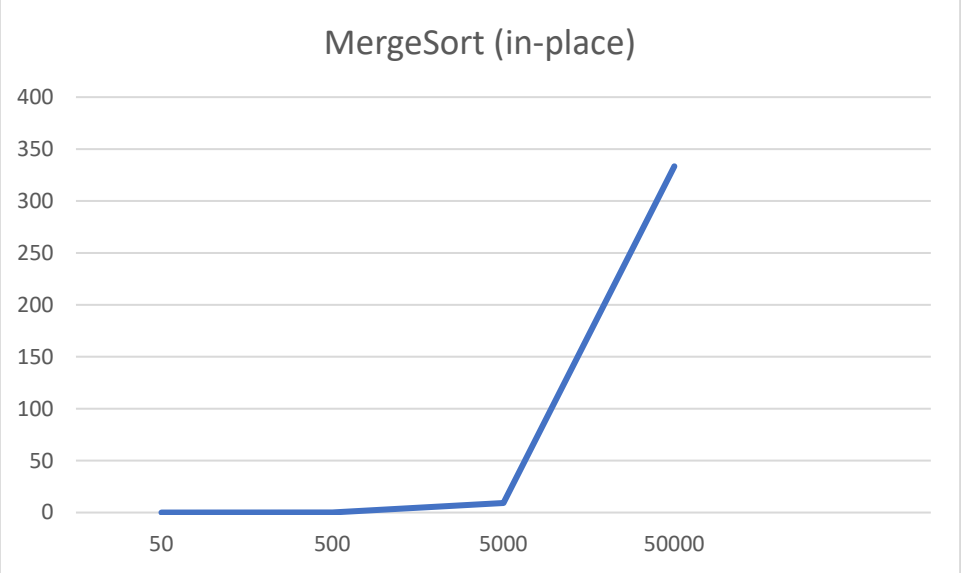

ShellSort
7
6
5
4
3
2
1
0
50
500
5000
50000

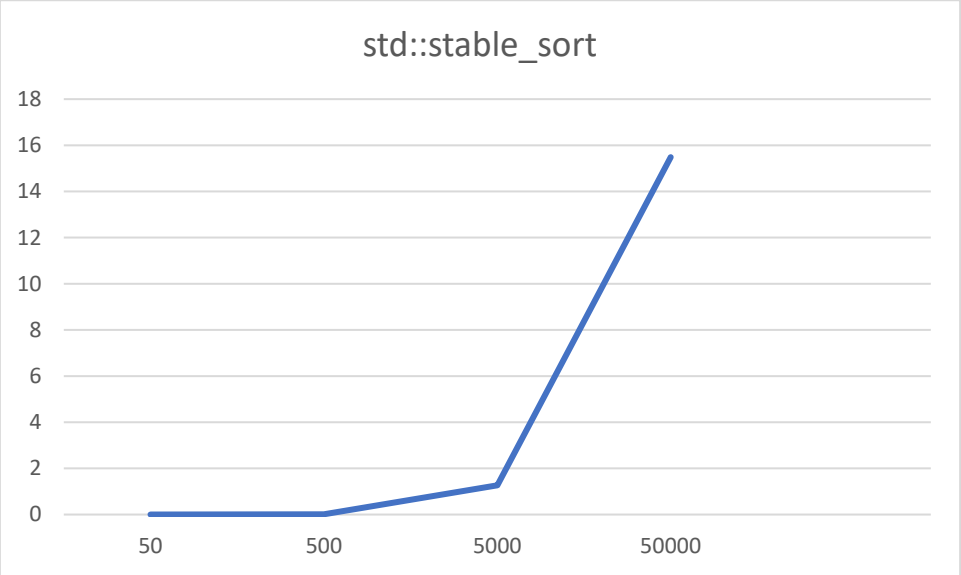

QuickSort
3
2,5
2
1,5
1
0,5
0
50
500
5000
50000


IntroSort
3
2,5
2
1,5
1
0,5
0
50
500
5000
50000


HeapSort
30
25
20
15
10
5
0
50
500
5000
50000


MergeSort
4
3,5
3
2,5
2
1,5
1
0,5
0
50
500
5000
50000


MergeSort (in-place)
400
350
300
250
200
150
100
50
0
50
500
5000
50000



У некоторых типов сортировок обнаружены различные просадки, которые связанны больше с недостаточной оптимизацией под структуры, хоть и оно побеждает для большого количества данных, необходимо собственные решения в виде гибридной сортировки, которая бы учитывала разницу в количестве элементов, тем самым эффективно решая разные варианты.

На огромных массивах сортировки требуется огромное количество времени, а также большое количество ошибок, так что из статистики были исключены все данные для 500 000 элементов, но все равно, побеждает встроенная сортировка, быстрая сортировка.

Заключение

В заключении можно сделать следующие выводы по сортировкам.

Если мы имеем дело с крошечными массивами (до 100 элементов), то лучше всего использовать простые сортировки, такие как сортировка пузырьком или сортировка выборки. Но при работе с огромными массивами лучше использовать более эффективные алгоритмы, такие как быструю сортировку или другие. Особенно быстрая сортировка показала себя лучшим образом на больших массивах.

Также мы можем сделать вывод, что встроенная сортировка, реализована очень быстрой. Единственное, когда встроенная сортировка уступает, это на крошечных массивах, но это может быть в рамках погрешности.

Прочие задания

(4) Не рекурсивные реализации рекурсивных методов и вопросы

Задача построения не рекурсивных реализаций рекурсивных методов сортировки, например, быстрой сортировки, сортировки слиянием и других, известна как проблема "разворачивания рекурсии" (англ. recursive unrolling). Разворачивание рекурсии — это прием, при котором рекурсивные функции заменяются эквивалентными не рекурсивными функциями для уменьшения времени выполнения и использования памяти, когда большие массивы должны быть отсортированы. Преимуществом не рекурсивных реализаций является то, что они не используют стек вызова и не имеют проблем с переполнением стека. Однако не рекурсивные реализации обычно более сложны и требуют больше памяти.

Например, для быстрой сортировки можно использовать стек вызова вместо рекурсии. Не рекурсивная реализация быстрой сортировки может выглядеть примерно так:

```
#include <stack>
#include <utility>

template<typename RandomIt>
void quicksort(RandomIt first, RandomIt last)
{
    std::stack<std::pair<RandomIt, RandomIt>> stk;
    stk.emplace(first, last);
    while (!stk.empty())
    {
        auto const [left, right] = stk.top();
```

```
        stk.pop();
        if (left < right)
        {
            auto const pivot = left + std::distance(left, right)/2;
            auto const mid = std::partition(left, right, [pivot](auto const&
elem){ return elem < *pivot; });
            stk.emplace(mid+1, right);
            stk.emplace(left, mid);
        }
    }
}
```

Это не рекурсивная реализация быстрой сортировки, которая использует стек вызова вместо рекурсии. Она является эквивалентной рекурсивной реализации, но не имеет проблемы переполнения стека.

Еще один способ улучшения производительности рекурсивных алгоритмов сортировки — это использование рандомизации. Например, в быстрой сортировке можно выбирать случайный элемент из массива в качестве опорного вместо того, чтобы выбирать первый или последний элемент. Это позволяет снизить вероятность худшего случая и улучшить производительность.

Чтобы выполнять сортировку больших массивов, можно использовать многопоточность и распределенные вычисления. Например, можно разбить массив на несколько частей, выполнить сортировку каждой части в отдельном потоке и затем объединить их в общий массив. Это позволяет использовать несколько ядер процессора и ускорить выполнение сортировки.

(5) Гибридная сортировка

Как мы знаем, гибридная сортировка — это метод сортировки, который комбинирует несколько алгоритмов сортировки в одном алгоритме в зависимости от входных данных. Например, для небольших массивов можно использовать простые алгоритмы, такие как пузырьковую сортировку и сортировку вставками, а для больших массивов - более сложные алгоритмы, такие как быструю сортировку и сортировку слиянием.

Поэтому я реализовал гибридную сортировку на C++, которая использует сортировку пузырьком для массивов размером до 10 элементов и быструю сортировку для массивов размером более 10 элементов для целочисленных данных:

```
template <typename RandomIt>
void hybrid_sort(RandomIt first, RandomIt last)
{
    if (std::distance(first, last) < 10)
    {
        std::sort(first, last); // Используем сортировку вставками для
небольших массивов
    }
    else
    {
        std::sort(first, last, [](const auto& lhs, const auto& rhs) { return
lhs < rhs; }); // Используем быструю сортировку для больших массивов
    }
}
```

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec { 4, 2, 8, 1, 5, 3, 7, 6 };

    hybrid_sort(vec.begin(), vec.end());

    for (const auto& i : vec)
    {
        std::cout << i << " ";
    }
}
```

Эта гибридная сортировка сначала проверяет размер массива. Если массив содержит менее 10 элементов, она использует сортировку вставками для быстрой сортировки. В противном случае, она использует быструю сортировку для сортировки массива.

(*) Ответы на вопросы

В каких случаях оправданно использование поразрядной сортировки?

Основным преимуществом поразрядной сортировки является ее высокая производительность на больших входных массивах. Она обычно работает очень быстро на массивах, у которых небольшое количество уникальных значений и диапазон значений известен заранее. При использовании поразрядной сортировки на целых числах, количество проходов зависит от количества бит в числах. Таким образом, поразрядная сортировка может быть эффективной для сортировки больших массивов целых чисел с небольшим диапазоном значений.

Однако поразрядная сортировка может быть неэффективной для массивов с большим количеством уникальных значений и для сортировки вещественных чисел, когда точность округления может быть проблемой. Кроме того, в некоторых случаях поразрядная сортировка может потребовать большое количество дополнительной памяти для временного хранения данных.

Таким образом, в случаях, когда необходима быстрая и эффективная сортировка большого массива целых чисел с небольшим диапазоном значений, поразрядная сортировка может быть очень полезной. Однако, если вам нужно сортировать массив с большим количеством уникальных значений, предпочтительнее использовать другие алгоритмы сортировки.

В каких случаях можно обогнать встроенную в язык сортировку?
В некоторых случаях вы можете написать более эффективную сортировку, чем стандартная сортировка, встроенная в язык. Например, если у вас есть массив объектов, можно написать собственную сортировку, которая сравнивает только определенные свойства объектов, что может быть более эффективным, чем сортировка стандартным способом.

Конечно, это зависит от конкретных требований вашей задачи, и иногда стандартная сортировка может оказаться наиболее эффективной. А ещё можно написать сортировку для специального типа данных, которую язык не поддерживает из коробки. И, конечно же, сортировку можно написать в тех случаях, когда вы хотите нестандартный порядок сортировки, например, сортировку строк, где заглавные буквы идут раньше, чем строчные. Как видно, есть много случаев, когда может быть полезно написать свою собственную сортировку.

В каких случаях имеет смысл использовать простые методы сортировки?
Простые методы сортировки, например, пузырьковая сортировка и сортировка вставками, могут быть полезны, если вы работаете с небольшими наборами данных или если данные уже почти отсортированы.

Например, если вам нужно отсортировать массив из 10 элементов, то простая сортировка может быть быстрее, чем более сложные алгоритмы, потому что они имеют меньшую асимптотическую сложность, то есть они выполняются быстрее, когда количество элементов не очень большое.

Но если у вас есть большой массив или если данные распределены хаотично, то более сложные алгоритмы, такие как сортировка слиянием или быстрая сортировка, будут более эффективными. Конечно, это зависит от конкретных требований задачи, так что необходимы некоторые тесты, чтобы определить, какая сортировка будет более эффективна или даже написать собственную реализацию на готовой сортировке, чтобы ускорить ее выполнение.

Какая сортировка лучше, если не знаем ничего про входные данные?
Если ничего не известно о входных данных (например, диапазон значений и количество элементов), то обычно наиболее безопасными и универсальными алгоритмами являются быстрая сортировка и сортировка слиянием.

Эти алгоритмы обычно обеспечивают хорошую производительность на большинстве типов данных и размеров входных наборов. Эти алгоритмы также достаточно просты в реализации и хорошо изучены, поэтому у них есть много оптимизаций и улучшений.

Кроме того, многие стандартные библиотеки и фреймворки включают в себя реализации быстрой сортировки и сортировки слиянием, так что использование этих алгоритмов может быть дополнительным преимуществом.

Приложение

sort.h:

```
// /////////////////////////////////////////////////////////
// sort.h
// Copyright (c) 2023 Sergey Leshkevich.

// g++ -O3 sort.cpp -o sort -std=c++11

// Весь алгоритм сортировки следует тому же синтаксису, что и std::sort
// т.е.: быстрая сортировка(container.begin(), container.end());
//
// Все виды сортировки, кроме сортировки слиянием, не требуют значительной
дополнительной памяти
// (просто некоторый стек для нескольких переменных и, возможно, копия одного
элемента).
//
```

```cpp
// Чтобы реализовать новую сортировку, реализуйте less-than operator.
// т.е.: quickSort(container.begin(), container.end(), myless());

#pragma once

#include <algorithm>  // std::iter_swap
#include <iterator>   // std::advance, std::iterator_traits
#include <functional> // std::less
#include <numeric>
#include <array>
#include <vector>     // std::vector

/// Сортировка тестовых данных (оболочка, где мы и сортируем)
/// Передаем количество элементов, все остальное за нас сделает генератор.
void testSortData(int numElements);

/// BubbleSort, реализация
template <typename iterator, typename LessThan>
void bubbleSort(iterator first, iterator last, LessThan lessThan)
{
  if (first == last)
    return;

  // обычно "последний" указывает за пределы последнего элемента
  // теперь он указывает непосредственно на этот последний элемент
  --last;
  // только один элемент => он отсортирован
  if (first == last)
    return;

  bool swapped;
  do
  {
    // сбросить замененный флаг
    swapped = false;

    auto current = first;
    while (current != last)
    {
      // пузыриться вверх
      auto next = current;
      ++next;

      // два соседа в неправильном порядке ? поменяйте их местами!
      if (lessThan(*next, *current))
      {
        std::iter_swap(current, next);
        swapped = true;
      }

      ++current;
    }

    // последний элемент уже отсортирован
    --last; // удалите эту строку, если у вас есть только прямой итератор

    // последний будет приближаться к первому
  } while (swapped && first != last);
}
```

```
/// BubbleSort
template <typename iterator>
void bubbleSort(iterator first, iterator last)
{
  bubbleSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}

/// RadixSort, реализация поразрядной сортировки
template<typename iterator>
void radix_sort(iterator first, iterator last, std::less<typename
std::iterator_traits<iterator>::value_type>)
{
    if (first == last) return;

    using value_type = typename std::iterator_traits<iterator>::value_type;
    std::vector<value_type> buffer(std::distance(first, last));

    for (int shift = 0; shift < 8 * sizeof(value_type); shift += 8)
    {
        auto mask = 0xFF << shift;
        std::array<int, 256> count{};
        for (auto it = first; it != last; ++it)
        {
            ++count[(*it & mask) >> shift];
        }
        std::partial_sum(count.begin(), count.end(), count.begin());
        for (auto it = last - 1; it >= first; --it)
        {
            buffer[--count[(*it & mask) >> shift]] = std::move(*it);
        }
        std::move(buffer.begin(), buffer.begin() + std::distance(first,
last), first);
    }
}

/// RadixSort
template <typename iterator>
void radixSort(iterator first, iterator last)
{
  radix_sort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}

// ////////////////////////////////////////////////////////////////////

/// Selection Sort, реализация
template <typename iterator, typename LessThan>
void selectionSort(iterator first, iterator last, LessThan lessThan)
{
  for (iterator current = first; current != last; ++current)
  {
    // найдите наименьший элемент в несортированной части и запомните его
итератор в "minimum"
    auto minimum = current;
    auto compare = current;
```

```
    ++compare;

    // пройтись по всем еще несортированным элементам
    while (compare != last)
    {
      // найден новый минимум ? сохраните его итератор
      if (lessThan(*compare, *minimum))
        minimum = compare;

      // следующий элемент
      ++compare;
    }

    // добавить минимум в конец уже отсортированной части
    if (current != minimum)
      std::iter_swap(current, minimum);
  }
}


/// Selection Sort
template <typename iterator>
void selectionSort(iterator first, iterator last)
{
  selectionSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////


/// Insertion Sort, реализация
template <typename iterator, typename LessThan>
void insertionSort(iterator first, iterator last, LessThan lessThan)
{
  if (first == last)
    return;

  // пропустите первый элемент, считайте его отсортированным
  auto current = first;
  ++current;

  // вставить все оставшиеся несортированные элементы в отсортированные
элементы
  while (current != last)
  {
    // вставить "сравнить" в уже отсортированные элементы
    auto compare = std::move(*current);

    // найти местоположение внутри отсортированного диапазона, начиная с
правого конца
    auto pos = current;
    while (pos != first)
    {
      // остановить, если левый сосед не меньше
      auto left = pos;
      --left;
      if (!lessThan(compare, *left))
```

```
        break;

      // сдвиньте этого левого соседа на одну позицию вправо
      *pos = std::move(*left);
      pos  = std::move( left); // PS: то же, что и --pos
    }

    // найдено конечное положение
    if (pos != current)
      *pos = std::move(compare);

    // отсортировать следующий элемент
    ++current;
  }
}


/// Insertion Sort
template <typename iterator>
void insertionSort(iterator first, iterator last)
{
  insertionSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////


/// Shell Sort, реализация
template <typename iterator, typename LessThan>
void shellSort(iterator first, iterator last, LessThan lessThan)
{
  auto numElements = std::distance(first, last);
  if (numElements <= 1)
    return;

  // последовательность взята из Википедии (Марцин Чура)
  static const int OptimalIncrements[] =
  { 68491, 27396, 10958, 4383, 1750, 701, 301, 132, 57, 23, 10, 4, 1, 0 };
  size_t increment = OptimalIncrements[0];
  size_t incrementIndex = 0;
  // приращение не должно быть больше количества сортируемых элементов
  while (increment >= numElements)
    increment = OptimalIncrements[++incrementIndex];

  // перебирать все приращения в порядке убывания
  while (increment > 0)
  {
    auto stripe = first;
    auto offset = increment;
    std::advance(stripe, offset);
    while (stripe != last)
    {
      // эти итераторы всегда "увеличиваются" друг от друга
      auto right = stripe;
      auto left  = stripe;
      std::advance(left, -int(increment));
```

```
      // значение, подлежащее сортировке
      auto compare = *right;

      // примечание: полоса просто такая же, как первая + смещение
      // но operator+() является дорогостоящим для итераторов с произвольным
доступом
      auto posRight = offset;
      // смотрите только на значения между "первым" и "последним"
      while (true)
      {
        // нашли нужное место ?
        if (!lessThan(compare, *left))
          break;

        // нет, все еще неправильный порядок: сдвиньте больший элемент вправо
        *right = std::move(*left);

        // сделайте один шаг влево
        right  = left;

        posRight -= increment;
        if (posRight < increment)
          break;
        std::advance(left, -int(increment));
      }

      // найдена отсортированная позиция
      if (posRight != offset)
        *right = std::move(compare);

      // следующее
      ++stripe;
      ++offset;
    }

    // меньшее приращение
    increment = OptimalIncrements[incrementIndex++];
  }
}


/// Shell Sort
template <typename iterator>
void shellSort(iterator first, iterator last)
{
  shellSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////


/// Heap Sort, реализация
template <typename iterator, typename LessThan>
void heapSort(iterator first, iterator last, LessThan lessThan)
{
  // просто используйте STL-код
  std::make_heap(first, last, lessThan);
```

```
  std::sort_heap(first, last, lessThan);
}


/// Heap Sort
template <typename iterator>
void heapSort(iterator first, iterator last)
{
  // просто используйте STL-код
  std::make_heap(first, last);
  std::sort_heap(first, last);
}

// ////////////////////////////////////////////////////////////////////////


/// Merge Sort, реализация
template <typename iterator, typename LessThan>
void mergeSort(iterator first, iterator last, LessThan lessThan, size_t size
= 0)
{
  // определить размер, если он еще не известен
  if (size == 0 && first != last)
    size = std::distance(first, last);
  // кстати, параметр size можно опустить,
  // но тогда мы должны вычислять его каждый раз,
  // что может быть дорогостоящим для итераторов с произвольным доступом

  // один элемент всегда сортируется
  if (size <= 1)
    return;

  // разделить на две части
  auto firstHalf  = size / 2;
  auto secondHalf = size - firstHalf;
  auto mid = first;
  std::advance(mid, firstHalf);

  // рекурсивно сортировать их
  mergeSort(first, mid,  lessThan, firstHalf);
  mergeSort(mid,   last, lessThan, secondHalf);

  // объединить отсортированные разделы
  std::inplace_merge(first, mid, last, lessThan);
}


/// Merge Sort
template <typename iterator>
void mergeSort(iterator first, iterator last)
{
  mergeSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////////
```

```
/// in-place Merge Sort, реализация
template <typename iterator, typename LessThan>
void mergeSortInPlace(iterator first, iterator last, LessThan lessThan,
size_t size = 0)
{
  // определить размер, если он еще не известен
  if (size == 0 && first != last)
    size = std::distance(first, last);
  // кстати, параметр size можно опустить,
  // но тогда мы должны вычислять его каждый раз,
  // что может быть дорогостоящим для итераторов с произвольным доступом

  // один элемент всегда сортируется
  if (size <= 1)
    return;

  // разделить на две части
  auto firstHalf  = size / 2;
  auto secondHalf = size - firstHalf;
  auto mid = first;
  std::advance(mid, firstHalf);

  // рекурсивно сортировать их
  mergeSortInPlace(first, mid,  lessThan, firstHalf);
  mergeSortInPlace(mid,   last, lessThan, secondHalf);

  // объединить разделы (левый начинается с "first", правый начинается с
"mid")
  // перемещайте итераторы ближе к концу, пока они не встретятся
  auto right = mid;
  while (first != mid)
  {
    // следующее значение обоих разделов в неправильном порядке (меньший
раздел находится слева)
    if (lessThan(*right, *first))
    {
      // это значение должно быть перемещено в нужный раздел
      auto misplaced = std::move(*first);

      // это значение относится к левому разделу
      *first = std::move(*right);

      // неуместное значение должно быть вставлено в правильное положение в
нужном разделе
      auto scan = right;
      auto next = scan;
      ++next;
      // переместить меньший размер на одну позицию влево
      while (next != last && lessThan(*next, misplaced))
        *scan++ = std::move(*next++);

      // нашел нужное место!
      *scan = std::move(misplaced);
    }

    ++first;
  }
}
```

```
/// in-place Merge Sort
template <typename iterator>
void mergeSortInPlace(iterator first, iterator last)
{
  mergeSortInPlace(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////


/// Quick Sort, реализация
template <typename iterator, typename LessThan>
void quickSort(iterator first, iterator last, LessThan lessThan)
{
  auto numElements = std::distance(first, last);
  // уже отсортировано ?
  if (numElements <= 1)
    return;

  auto pivot = last;
  --pivot;

  // выберите средний элемент в качестве опорного (хороший выбор для частично
отсортированных данных)
  if (numElements > 2)
  {
    auto middle = first;
    std::advance(middle, numElements/2);
    std::iter_swap(middle, pivot);
  }

  // сканируйте, начиная с левого и правого концов, и меняйте местами
неуместные элементы
  auto left  = first;
  auto right = pivot;
  while (left != right)
  {
    // ищите несоответствия
    while (!lessThan(*pivot, *left)  && left != right)
      ++left;
    while (!lessThan(*right, *pivot) && left != right)
      --right;
    // поменяйте местами два значения, которые оба находятся на неправильной
стороне сводного элемента
    if (left != right)
      std::iter_swap(left, right);
  }

  // переместить ось поворота в ее конечное положение
  if (pivot != left && lessThan(*pivot, *left))
    std::iter_swap(pivot, left);

  quickSort(first,  left, lessThan);
  quickSort(++left, last, lessThan); // *сам left уже отсортирован!!!!
}
```

```
/// Quick Sort
template <typename iterator>
void quickSort(iterator first, iterator last)
{
  quickSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}


// ////////////////////////////////////////////////////////////////////


/// Intro Sort, реализация
template <typename iterator, typename LessThan>
void introSort(iterator first, iterator last, LessThan lessThan)
{
  // переключитесь на сортировку по вставке, если массив (вложенный) невелик
  auto numElements = std::distance(first, last);
  if (numElements <= 16)
  {
    // уже отсортировано ?
    if (numElements <= 1)
      return;

    // микрооптимизация ровно для 2 элементов
    if (numElements == 2)
    {
      if (lessThan(*(first + 1), *first))
        std::iter_swap(first + 1, first);
      return;
    }

    // от 3 до 16 элементов
    insertionSort(first, last, lessThan);
    return;
  }

  auto pivot = last;
  --pivot;

  // выберите средний элемент в качестве опорного (хороший выбор для частично
отсортированных данных)
  auto middle = first;
  std::advance(middle, numElements/2);
  std::iter_swap(middle, pivot);

  // сканируйте, начиная с левого и правого концов, и меняйте местами
неуместные элементы
  auto left  = first;
  auto right = pivot;
  while (left != right)
  {
    // ищите несоответствия
    while (!lessThan(*pivot, *left)  && left != right)
      ++left;
    while (!lessThan(*right, *pivot) && left != right)
      --right;
```

```
    // поменяйте местами два значения, которые оба находятся на неправильной
стороне сводного элемента
    if (left != right)
      std::iter_swap(left, right);
  }

  // переместить ось поворота в ее конечное положение
  if (pivot != left && lessThan(*pivot, *left))
    std::iter_swap(pivot, left);

  introSort(first,  left, lessThan);
  introSort(++left, last, lessThan); // *сам left уже отсортирован!!!
}


/// Intro Sort
template <typename iterator>
void introSort(iterator first, iterator last)
{
  introSort(first, last, std::less<typename
std::iterator_traits<iterator>::value_type>());
}
```

Sort.cpp:

```
// ////////////////////////////////////////////////////////
// sort.cpp
// Copyright (c) 2023 Sergey Leshkevich.
//

// g++ -O3 -std=c++11 sort.cpp -o sort

#include <cstdio>
#include <cstdlib>   // srand/rand
#include <cmath>     // fabs

#include <vector>
#include <list>
#include <algorithm> // std::sort, std::reverse

#include "src/sort.h"
```

```
// добавьте -DCHECKRESULT в командную строку GCC => если хотите, чтобы
результаты будут проверены на правильность их сортировки

// тип данных, подлежащий сортировке
typedef int Number;
typedef std::vector<Number> Container;


// защита ОС от перегрузки:
// отключите несколько очень медленных тестов, когда сортируется более чем
ограниченное количество элементов
const int RestrictedSort = 25000000;
// верхний предел, никаких сортировок сверх этого количества элементов
const int MaxSort        = 10000000;

#ifdef _MSC_VER
#define USE_WINDOWS_TIMER
#endif

#ifdef USE_WINDOWS_TIMER
#include <windows.h>
#include <ctime>
#else
#include <sys/time.h>
#endif


// Время, зависящее от конкретной операционной системы
static double seconds()
{
#ifdef USE_WINDOWS_TIMER
  LARGE_INTEGER frequency, now;
  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter  (&now);
  return now.QuadPart / double(frequency.QuadPart);
#else
  timeval now;
  gettimeofday(&now, NULL);
  return now.tv_sec + now.tv_usec/1000000.0;
#endif
}

// Главная функция
int main()
{
    testSortData(50);
    testSortData(500);
    testSortData(5000);
    testSortData(50000);
    testSortData(500000);
}


void testSortData(int numElements)
{

  // только положительные числа!
```

```
  if (numElements == 0)
    numElements = 10000;
  if (numElements < 0)
    numElements = -numElements;
  // избегайте перегрузки ОС
  if (numElements > MaxSort)
    numElements = MaxSort;

#ifdef LESSTHAN
  printf("%d element%s\n", numElements, numElements == 1 ? "":"s");
#else
  printf("\n%d integer%s\t\t  ascending \t descending  \t     random\t   all
time\n", numElements, numElements == 1 ? "":"s");
#endif

  // инициализировать контейнер на N элементов
  // 0,1,2,3,4,...
  Container ascending(numElements);
  for (size_t i = 0; i < numElements; i++)
    ascending[i] = Number(i);

  // ...,4,3,2,1,0
  Container descending(numElements);
  for (size_t i = 0; i < numElements; i++)
    descending[i] = Number((numElements - 1) - i);
  //std::reverse(descending.begin(), descending.end());

  // просто случайные числа
  Container random(numElements);
  srand(time(NULL));//0);
  for (int i = 0; i < numElements; i++)
    random[i] = Number(rand());

#ifdef CHECKRESULT
  Container sorted = ascending;
  Container sortedRandom = random;
  std::sort(sortedRandom.begin(), sortedRandom.end());
#endif // CHECKRESULT

  // используйте этот контейнер для входных данных
  Container data;

  double timeInverted = 0;
  double timeSorted   = 0;
  double timeRandom   = 0;

#ifndef FORWARDITERATOR
  // BubbleSort
  // sorted data
  data = ascending;
  timeSorted = seconds();
  bubbleSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT
```

```
  if (numElements < RestrictedSort)
  {
    // inverted data
    data = descending;
    timeInverted = seconds();
    bubbleSort(data.begin(), data.end());
    timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // random data
    data = random;
    timeRandom = seconds();
    bubbleSort(data.begin(), data.end());
    timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
    if (data != sortedRandom)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    printf("Bubble Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
           1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
  }
  else
    // skip bubble sort in order to prevent server overload
    printf("Bubble Sort\t\t%8.3f ms\tn/a\tn/a\tn/a\n", 1000*timeSorted);
#endif // FORWARDITERATOR


  // SelectionSort
  if (numElements < RestrictedSort)
  {
    // inverted data
    data = descending;
    timeInverted = seconds();
    selectionSort(data.begin(), data.end());
    timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // sorted data
    timeSorted = seconds();
    selectionSort(data.begin(), data.end());
    timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // random data
```

```
    data = random;
    timeRandom = seconds();
    selectionSort(data.begin(), data.end());
    timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
    if (data != sortedRandom)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    printf("Selection Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
           1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
  }
  else
    // skip bubble sort in order to prevent server overload
    printf("Selection Sort\t\t\tn/a\tn/a\tn/a\tn/a\n");

  // SelectionSort
  if (numElements < RestrictedSort)
  {
    // inverted data
    data = descending;
    timeInverted = seconds();
    radixSort(data.begin(), data.end());
    timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // sorted data
    timeSorted = seconds();
    radixSort(data.begin(), data.end());
    timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // random data
    data = random;
    timeRandom = seconds();
    radixSort(data.begin(), data.end());
    timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
    if (data != sortedRandom)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    printf("Radix Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
           1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
  }
  else
    // skip radix sort in order to prevent server overload
```

```
    printf("Radix Sort\t\t\tn/a\tn/a\tn/a\tn/a\n");


#ifndef FORWARDITERATOR
  // InsertionSort
  // sorted data
  data = ascending;
  timeSorted = seconds();
  insertionSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  if (numElements < RestrictedSort)
  {
    // inverted data
    data = descending;
    timeInverted = seconds();
    insertionSort(data.begin(), data.end());
    timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // random data
    data = random;
    timeRandom = seconds();
    insertionSort(data.begin(), data.end());
    timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
    if (data != sortedRandom)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    printf("Insertion Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
           1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
  }
  else
    // skip bubble sort in order to prevent server overload
    printf("Insertion Sort\t\t%8.3f ms\tn/a\tn/a\tn/a\n", 1000*timeSorted);
#endif // FORWARDITERATOR


#ifndef FORWARDITERATOR
  // ShellSort
  // inverted data
  data = descending;
  timeInverted = seconds();
  shellSort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
```

```
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  timeSorted = seconds();
  shellSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  data = random;
  timeRandom = seconds();
  shellSort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
  if (data != sortedRandom)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  printf("Shell Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // FORWARDITERATOR


#ifndef FORWARDITERATOR
  // QuickSort
  // inverted data
  data = descending;
  timeInverted = seconds();
  quickSort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  timeSorted = seconds();
  quickSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  data = random;
  timeRandom = seconds();
  quickSort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);
```

```
#ifdef CHECKRESULT
  if (data != sortedRandom)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  printf("Quick Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // FORWARDITERATOR


#ifndef FORWARDITERATOR
  // IntroSort
  // inverted data
  data = descending;
  timeInverted = seconds();
  introSort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  timeSorted = seconds();
  introSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  data = random;
  timeRandom = seconds();
  introSort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
  if (data != sortedRandom)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  printf("Intro Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // FORWARDITERATOR


#if !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)
  // HeapSort
  // inverted data
  data = descending;
  timeInverted = seconds();
  heapSort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);
```

```
#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  timeSorted = seconds();
  heapSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  data = random;
  timeRandom = seconds();
  heapSort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
  if (data != sortedRandom)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  printf("Heap Sort\t\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)


#if !defined(FORWARDITERATOR)
  // n-ary HeapSort
  // inverted data
  //data = descending;
  //timeInverted = seconds();
  //naryHeapSort<8>(data.begin(), data.end());
  //timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
  //if (data != sorted)
  //  printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  //timeSorted = seconds();
  //naryHeapSort<8>(data.begin(), data.end());
  //timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  //if (data != sorted)
  //  printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  //data = random;
  //timeRandom = seconds();
```

```
  //naryHeapSort<8>(data.begin(), data.end());
  //timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
  //if (data != sortedRandom)
  //  printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  //printf("n-ary Heap Sort (n=8)\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
  //       1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // !defined(FORWARDITERATOR)


#if !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)
  // MergeSort
  // inverted data
  data = descending;
  timeInverted = seconds();
  mergeSort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // sorted data
  timeSorted = seconds();
  mergeSort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  // random data
  data = random;
  timeRandom = seconds();
  mergeSort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
  if (data != sortedRandom)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  printf("Merge Sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)


  // in-place MergeSort
  // sorted data
  data = ascending;
  timeSorted = seconds();
  mergeSortInPlace(data.begin(), data.end());
```

```
  timeSorted = fabs(seconds() - timeSorted);

#ifdef CHECKRESULT
  if (data != sorted)
    printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

  if (numElements < RestrictedSort)
  {
    // inverted data
    data = descending;
    timeInverted = seconds();
    mergeSortInPlace(data.begin(), data.end());
    timeInverted = fabs(seconds() - timeInverted);

#ifdef CHECKRESULT
    if (data != sorted)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    // random data
    data = random;
    timeRandom = seconds();
    mergeSortInPlace(data.begin(), data.end());
    timeRandom = fabs(seconds() - timeRandom);

#ifdef CHECKRESULT
    if (data != sortedRandom)
      printf("Sorting problem @ %d ", __LINE__);
#endif // CHECKRESULT

    printf("Merge Sort in-place\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
           1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
  }
  else
    printf("Merge Sort in-place\t\t%8.3f ms\tn/a\tn/a\tn/a\n",
1000*timeSorted);


#if !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)
  // std::sort
  // inverted data
  data = descending;
  timeInverted = seconds();
  std::sort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

  // sorted data
  timeSorted = seconds();
  std::sort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

  // random data
  data = random;
  timeRandom = seconds();
  std::sort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);
```

```
  printf("std::sort\t\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));

  // std::stable_sort
  // inverted data
  data = descending;
  timeInverted = seconds();
  std::stable_sort(data.begin(), data.end());
  timeInverted = fabs(seconds() - timeInverted);

  // sorted data
  timeSorted = seconds();
  std::stable_sort(data.begin(), data.end());
  timeSorted = fabs(seconds() - timeSorted);

  // random data
  data = random;
  timeRandom = seconds();
  std::stable_sort(data.begin(), data.end());
  timeRandom = fabs(seconds() - timeRandom);

  printf("std::stable_sort\t\t%8.3f ms\t%8.3f ms\t%8.3f ms\t%8.3f ms\n",
         1000*timeSorted, 1000*timeInverted, 1000*timeRandom,
1000*(timeSorted+timeInverted+timeRandom));
#endif // !defined(FORWARDITERATOR) && !defined(BIDIRECTIONALITERATOR)

  return;
}
```

Generator.py:

```
from random_word import RandomWords
import random
import sys
import argparse
import os
import string

class TextFileGenerator():
    def __init__(self, number_of_files, file_length, word_list, file_name,
start_number, ran_file_name):
        self.random_generator = RandomWords()
        self.number_of_files = int(number_of_files)
        self.file_length = int(file_length)
        self.word_list = word_list
        self.current_random_words = None
        self.ran_file_name = ran_file_name
        if not start_number == None:
            self.start_number = start_number
        else:
            self.start_number = 0
        if not file_name == None:
            self.file_name = file_name
        else:
            self.file_name = 'testing_file'
        self.output_folder()
```

```
    def get_random_words_list(self):
        self.current_random_words = self.random_generator.get_random_words()
        while self.current_random_words == None:
            self.current_random_words =
self.random_generator.get_random_words()

    def generate_file(self, name):
        self.get_random_words_list()
        with open(name, 'w') as f:
            for i in range(self.file_length):
                currentLine = ''
                for x in range(random.randrange(15, 30, 1)):
                    if ((random.randrange(1, 5, 1)) == 1) and not
(self.word_list == None):
                        currentLine += f'{random.choice(self.word_list)} '
                    else:
                        currentLine +=
f'{random.choice(self.current_random_words)} '
                f.write(f'{currentLine}\n')

    def generator(self):
        for n in range(self.number_of_files):
            if self.ran_file_name:
                name = f'Output/{self.random_file_name()}.txt'
            else:
                name = f'Output/{self.file_name}{n +
int(self.start_number)}.txt'
            self.generate_file(name)

    def random_file_name(self):
        letters = string.ascii_letters
        return (''.join(random.choice(letters) for i in range(10)))

    def output_folder(self):
        if not os.path.isdir('Output'):
            os.mkdir('Output')

def parse_arguments():
    parser = argparse.ArgumentParser(description="Генерирует N файлов с L
строками, используя случайные полные слова.  Можно включить список слов, если
вам нужно, чтобы определенные слова были включены случайным образом.")
    parser.add_argument('Number of files', metavar='N', type=int,
help="Количество файлов для создания")
    parser.add_argument('Number of lines', metavar='L', type=int,
help="Количество строк для добавления в каждый файл")
    parser.add_argument('-w', '--word-list', metavar='[TEXT]', nargs='*',
help="Список слов, если вы хотите включить в него конкретные слова")
    parser.add_argument('-s', '--start-number', metavar='[TEXT]', help="Номер
файла, с которого нужно начать.")
    parser.add_argument('-n', '--file-name', metavar='[TEXT]', help="Имя,
используемое для создания файлов.")
    parser.add_argument('-r', '--random-file-name', action='store_true',
help="Вместо единообразных имен файлов каждый раз используйте случайное имя
файла.")

    args = parser.parse_args()
    return args
```

```
def main():
    args = vars(parse_arguments())
    generator = TextFileGenerator(args['Number of files'], args['Number of
lines'], args['word_list'], args['file_name'], args['start_number'],
args['random_file_name'])
    generator.generator()

if __name__ == '__main__':
    main()
```

1. Установка:

```
pip install -r requirements.txt
```

2. Параметры скрипта Python:

```
N – Количество файлов, которое надо создать, INT
L – Количество строк, которое будет создано в файле, INT
-w, --word-list - Необязательно, список слов, которые должны быть включены,
[[TEXT] [TEXT] ...]
```

3. Пример запуска:

```
python3 generator.py 10 100 -w kpfu mir victory
```

Результат будет сохранен относительно скрипта в папке Output.