Set-3

A1.

Результаты всех замеров можно найти по ссылкам на ВМ, там можно тыкать на легенду справа и отключать часть графиков, в отчёте будет картинки, но лишь, чтобы обратить внимание на конкретные моменты, рекомендуется посмотреть графики по ссылкам, так как они интерактивные:

http://84.201.168.143:8080 (Время сортировки для разных типов массивов)

http://84.201.168.143:8090 (Количество посимвольных сравнений для разных типов массивов)

http://84.201.168.143:9080 (Время сортировки для разных типов массивов с общим префиксом длины минимум 500)

http://84.201.168.143:9090 (Количество посимвольных сравнений для разных типов массивов с общим префиксом длины минимум 500)

ID посылок:

321221642 - String Merge Sort

321221797 - String Quick Sort

321221962 - MSD Radix Sort

321222455 - MSD Radix Sort + Quick Sort

Ссылка на github репо с результатами измерений и реализацией (будет запушена после дедлайна):

https://github.com/OlegSfn/HSE-Algorithms/tree/master/SET9

Дополнительно приложен zip архив с решениями.

Структура проекта:

Results - результаты измерений алгоритмов сортировки.

NoPrefix - при замерах использовались только строки из случайных символов в соответствии с условием.

WithPrefix - при замерах использовались только строки с общим префиксом из 500 "a оставшаяся часть строк сгенерирована в соответствии с условием.

В каждой из этих папок есть ещё 2

Time - результаты сравнения времени работы алгоритмов и скрипт для их отрисовки. (У стандартных сортировок есть ещё с компаратором без специальных оптимизаций для строк, об этом дальше)

Comparasings - результаты сравнения количества посимвольных сравнений и скрипт для их отрисовки.

Src - реализация, используемая для замеров и сами алгоритмы.

basic - реализации стандартных алгоритмов сортировки с возможностью использовать произвольный компаратор над std::string.

common - реализация, используемых компараторов.

SET - копия посылок с Codeforces.

stringAdapted - реализация новых алгоритмов сортировки в двух версиях, с подсчётом количества сравнений и без.

tester - программы для тестирования и замеров алгоритмов

Basic - результаты замеров стандартных алгоритмов

Comparasings - результаты сравнения количества посимвольных сравнений.

Time - результаты сравнения времени работы алгоритмов (у стандартных алгоритмов со стандартных компаратором (operator< == std::less) и кастомным без оптимизаций).

Аналогично StringAdapted те же папки Comparasings, Time, но уже только с одним видом компаратора, потому что char == int

visual - скрипты для отрисовки результатов замеров

tester.cpp - скрипт для замеров времени и количества сравнений, внутри есть требуемые классы StringGenerator и StringSortTester.

testCorrect.cpp - скрипт для проверки корректности алгоритма сортировки, результаты тестируемых алгоритмов сравниваются с std::sort.

Этап 1:

```
class StringGenerator {
public:
    StringGenerator() : gen_(std::random_device{}()), alphabetDist_(0, alphabet_.size()) {}

    std::vector<std::string> RandomArray(size_t arraySize, size_t minStringLength, size_t
        maxStringLength) {
        std::vector<std::string> result(arraySize);

        for (auto& s : result) {
            std::uniform_int_distribution<size_t> stringSizeDist(minStringLength,
                maxStringLength);
            s.resize(stringSizeDist(gen_));

            for (auto& c : s) {
                c = alphabet_[alphabetDist_(gen_)];
            }
        }

        return result;
    }

    std::vector<std::string> ReversedArray(size_t arraySize, size_t minStringLength, size_t
        maxStringLength) {
        std::vector<std::string> result = AlmostSortedArray(arraySize, minStringLength,
            maxStringLength, 0);
        std::reverse(result.begin(), result.end());

        return result;
    }

    std::vector<std::string> AlmostSortedArray(size_t arraySize, size_t minStringLength,
        size_t maxStringLength, float unsortedCoeff) {
        std::vector<std::string> result = RandomArray(arraySize, minStringLength,
            maxStringLength);
        std::sort(result.begin(), result.end());

        std::uniform_int_distribution<size_t> indexDist(0, result.size() - 1);
        for (size_t i = 0; i < result.size() * unsortedCoeff; ++i) {
```

```
33              std::swap(result[i], result[indexDist(gen_)]);
34          }
35
36          return result;
37      }
38
39      std::vector<std::string> RandomArrayWithPrefix(size_t arraySize, size_t minStringLength,
            size_t maxStringLength) {
40          std::vector<std::string> result = RandomArray(arraySize, minStringLength,
                maxStringLength);
41          std::string prefix(500, 'a');
42          for (auto& s : result) {
43              s = prefix + s;
44          }
45
46          return result;
47      }
48
49      std::vector<std::string> ReversedArrayWithPrefix(size_t arraySize, size_t minStringLength,
            size_t maxStringLength) {
50          std::vector<std::string> result = AlmostSortedArrayWithPrefix(arraySize,
                minStringLength, maxStringLength, 0);
51          std::reverse(result.begin(), result.end());
52
53          return result;
54      }
55
56      std::vector<std::string> AlmostSortedArrayWithPrefix(size_t arraySize, size_t
            minStringLength, size_t maxStringLength, float unsortedCoeff) {
57          std::vector<std::string> result = RandomArrayWithPrefix(arraySize, minStringLength,
                maxStringLength);
58          std::sort(result.begin(), result.end());
59
60          std::uniform_int_distribution<size_t> indexDist(0, result.size() - 1);
61          for (size_t i = 0; i < result.size() * unsortedCoeff; ++i) {
62              std::swap(result[i], result[indexDist(gen_)]);
63          }
64
65          return result;
66      }
67
68  private:
69      std::mt19937 gen_;
70      std::string alphabet_ =
            "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#%:;^&*()-";
71      std::uniform_int_distribution<size_t> alphabetDist_;
72  };
```

Этап 2-3:

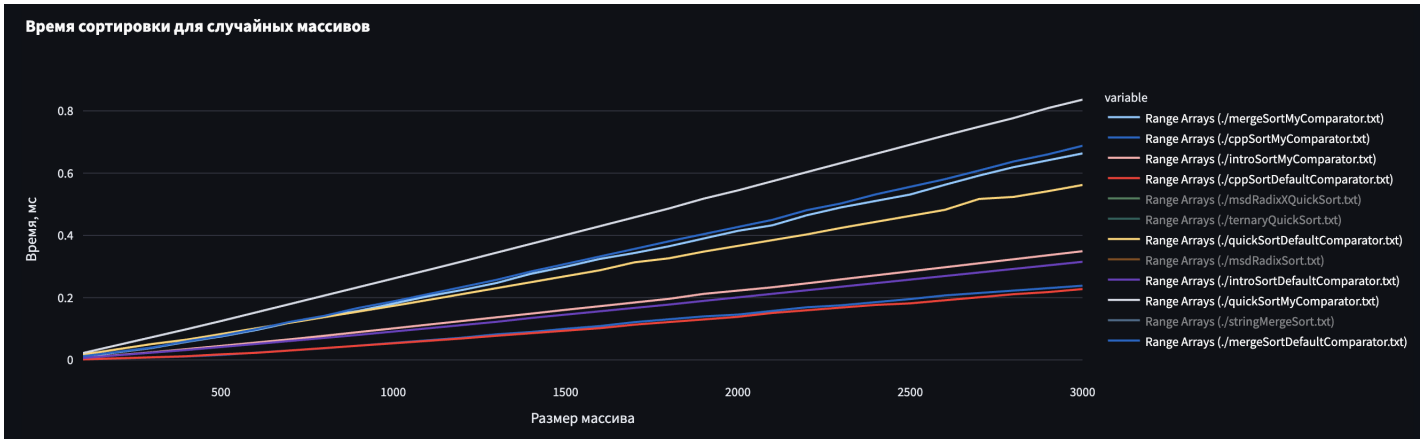
Результаты представлены по ссылкам, описанные выше, на графиках можно выбрать интересующие линии, нажав на легенде на них.

Быстрая сортировка для выбора pivot в обоих случаях использует следующую конструкцию

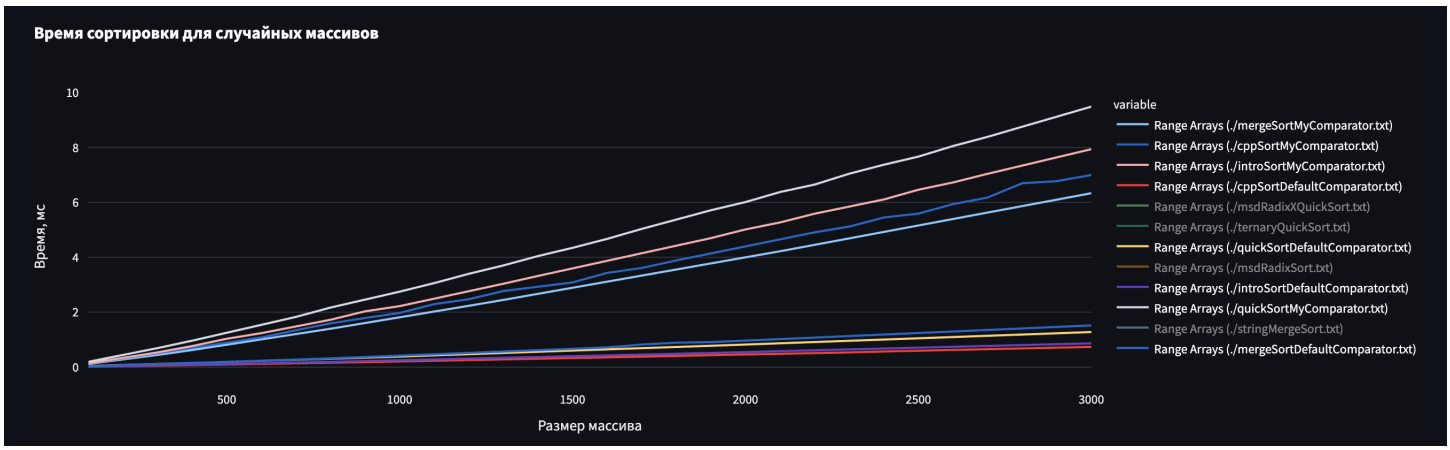
```
static std::mt19937 rng(std::random_device{}());
std::uniform_int_distribution<size_t> dist(begin, end-1);
std::string pivot = a[dist(rng)];
```

Что при достаточно большом количестве замеров делает их идентичными.

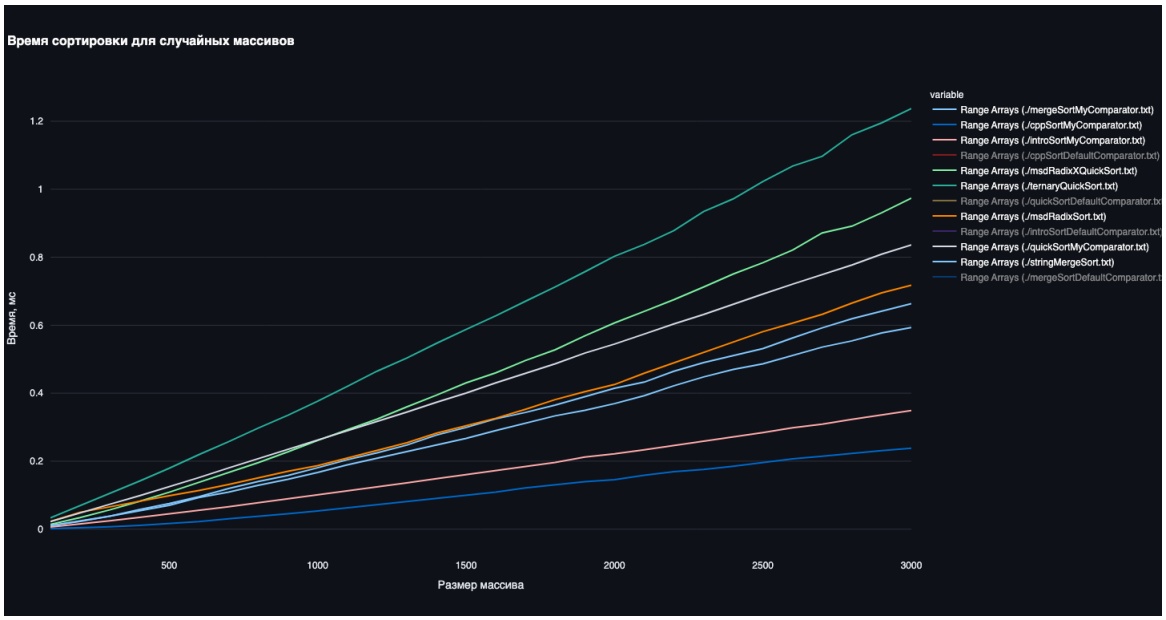
Как оказалось operator< на строках в плюсах оптимизирован на посимвольные сравнение, этот факт подтверждает то, что при замене на "аналогичный"компаратор без оптимизаций время на сортировку сильно возрастает, поэтому для более честных сравнений необходимо смотреть только на стандартные алгоритмы с суффиксом MyComparator.



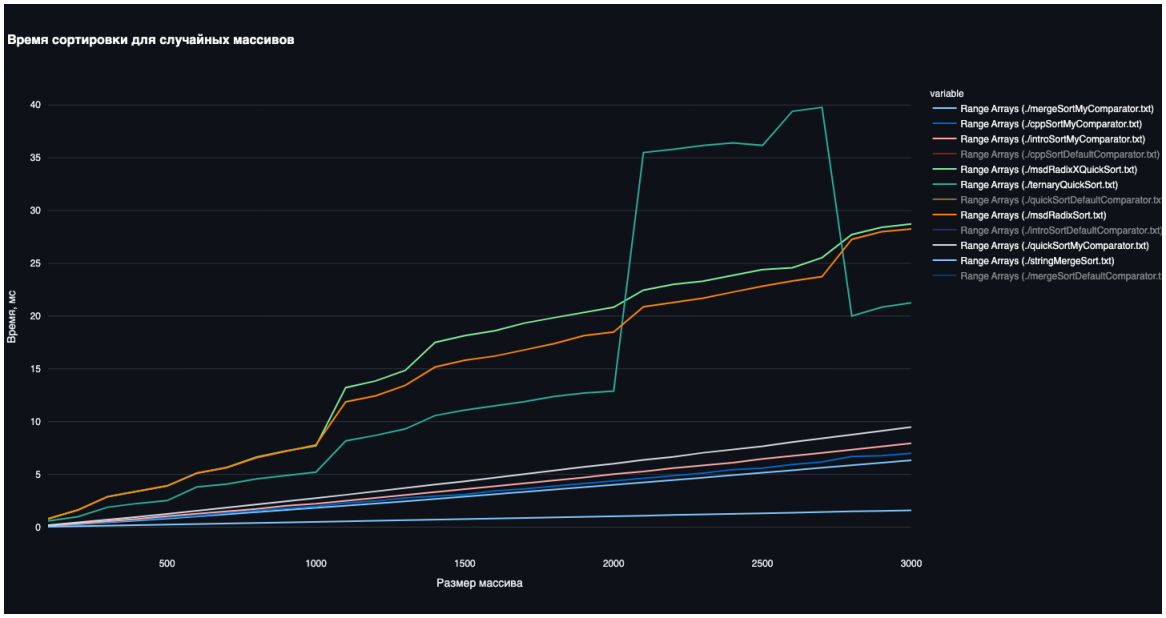
И ещё большие расхождения во времени на строках с общим префиксом из 500 элементов



Сравнивая стандартные алгоритмы (MyComparator) с эффективными алгоритмом по времени остаётся introSort == cppSort (std::sort), неплохие результаты показывают msdRadixSort, stringMergeSort.



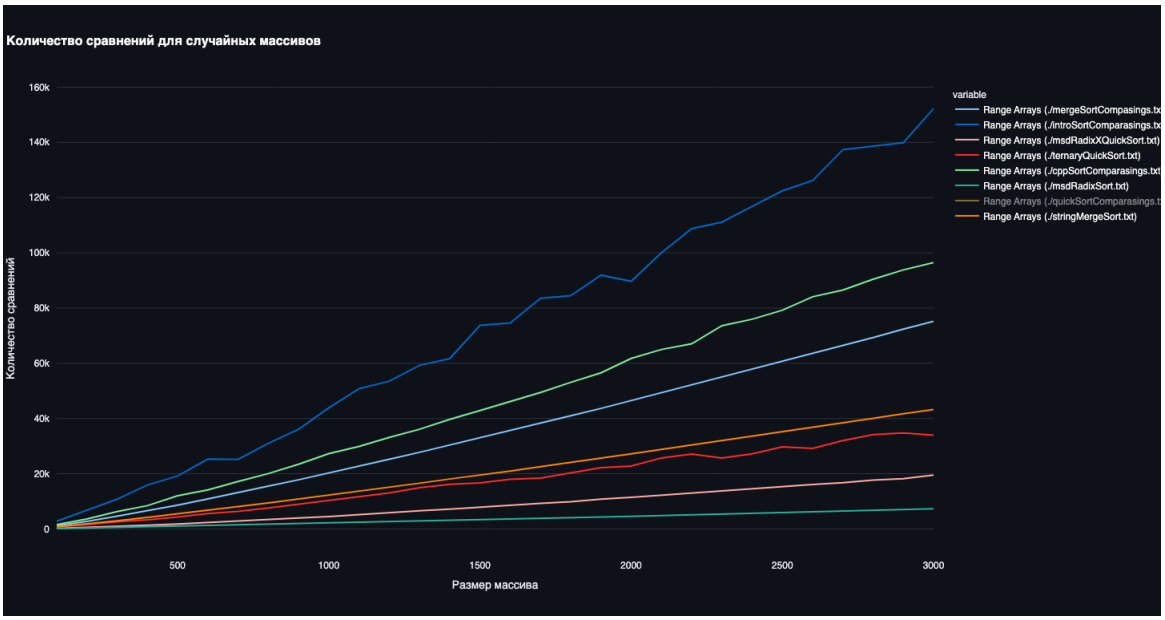
И на строках с общим префиксом, здесь stringMergeSort даже сильно вырвался вперёд. (Что это за поднятие с quickSort я точно понять не смог, перф тест сказал, что частенько вызывается $\text{slow}_p athem place_b ack, ,)$



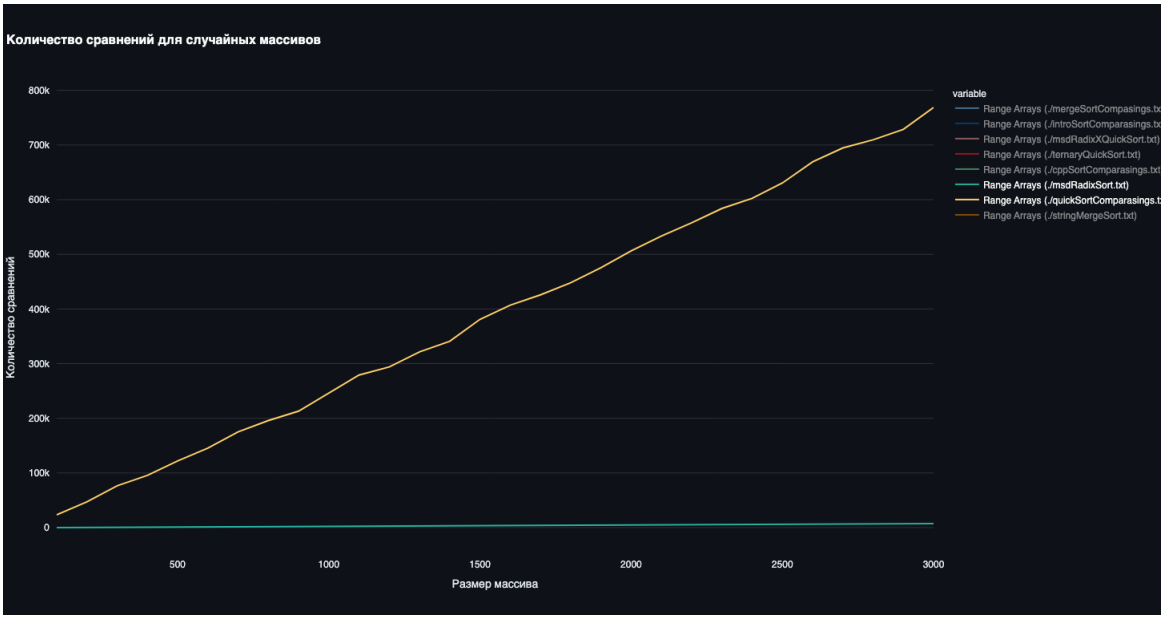
Ситуация между типами разными массивов (обратными, почти отсортированными) почти нет, единственное cppSort как-то очень оптимизирован под обратные массивы, introSort таким не обладает.

Что же насчёт посимвольных сравнений?

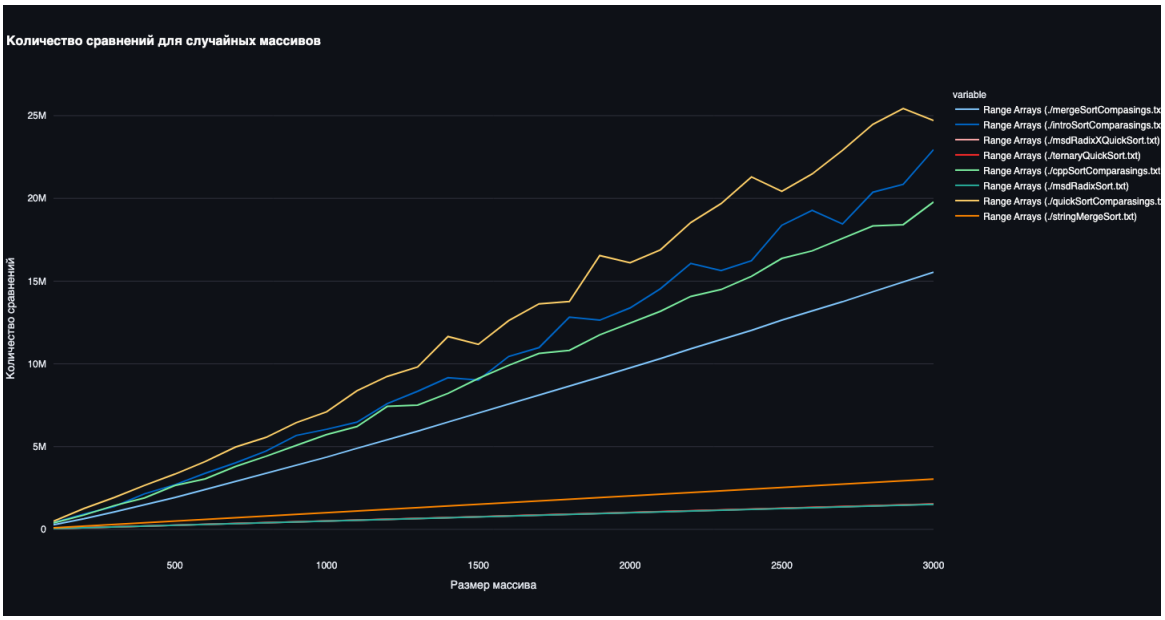
На случайных строках разрвыв (без учёта quickSort - земля пухом и легендарным msdRadixSort примерно в 2.5 раза. quickSort / msdRadixSort = 100)



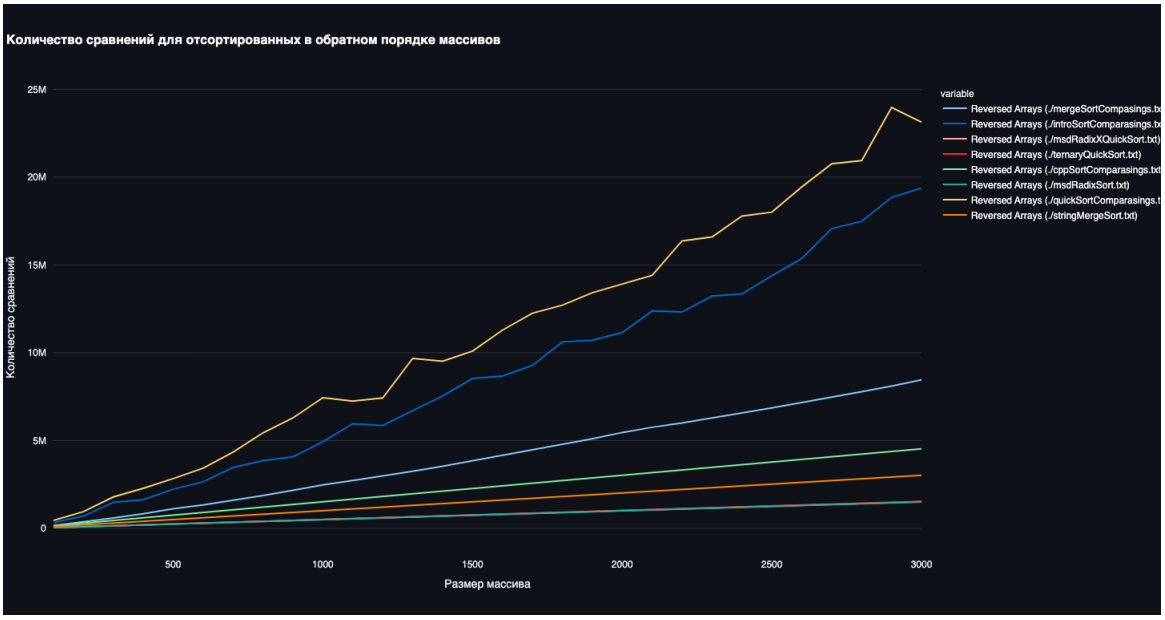
msdRadixSort vs quickSort:



На строках с префиксом ситуация ещё хуже для обычных, можно чёт сказать, где все обычные, а где специализированные:



Интересно, что на обратных массивав cppSort вырывается вперёд, но всё ещё хуже всех специализированных:



По результатам можно сказать, что на маленьких строках и массивах затраты на специализированные алгоритмы излишни, хоть по сравнениям они и выигрывают сильно, скорее всего на больших строках и массивах результаты по времени работы будут не в пользу обычных алгоритмов.

В завершение посчитаем теоретическое кол-во посимвольных сравнений для сортировок на примере строк с общим префиксом в 500 символов, в лучше случае оно:

$$501 \cdot 3000 + 3000 \cdot \log(3000) \approx 1.5\text{млн}$$

, по графику видно, что у нас примерно это значение для всех алгоритмов, кроме stringMergeSort, у него получилось около 3 млн, скорее всего мы всегда попадали в else if, а значит нам нужно было 2 сравнения вместо 1, это всё ещё $O(\sum LCP(R) + nlog(n))$