Посылки на cf

A1m - 321287580
A1q - 321289013
A1r - 321291090
A1rq - 321291740

Ссылка на репозиторий

Ссылка: гитхаб

Отчет

Думаю, не стоит грузит отчет строчками кода, поэтому все файлы с результатами анализа и кодом для его проведения вы найдете в репозитории.
Хочется сказать в общем, что для проведения анализа использовалось три версии массивов перечисленных в задании, в котор ых хранились 3000 рандомных, сгенерированных строк каждая длинной в 200 символов. Для кастомного StringQuickSort'a пивот выбирался рандомно. Замеры времени проводились в микросекундах, для подсчета сравнений сортировки из контеста были немного переделаны, буквально, чтобы считать количество этих самых сравнений. Все замеры усреднялись, все сортировки проводились по 1000 раз, в надежде на нормальное усреднение. Данные, записанные в файлах уже усредненные. Также к сведению были приняты и выполнены все замечания, а именно:
Замечания

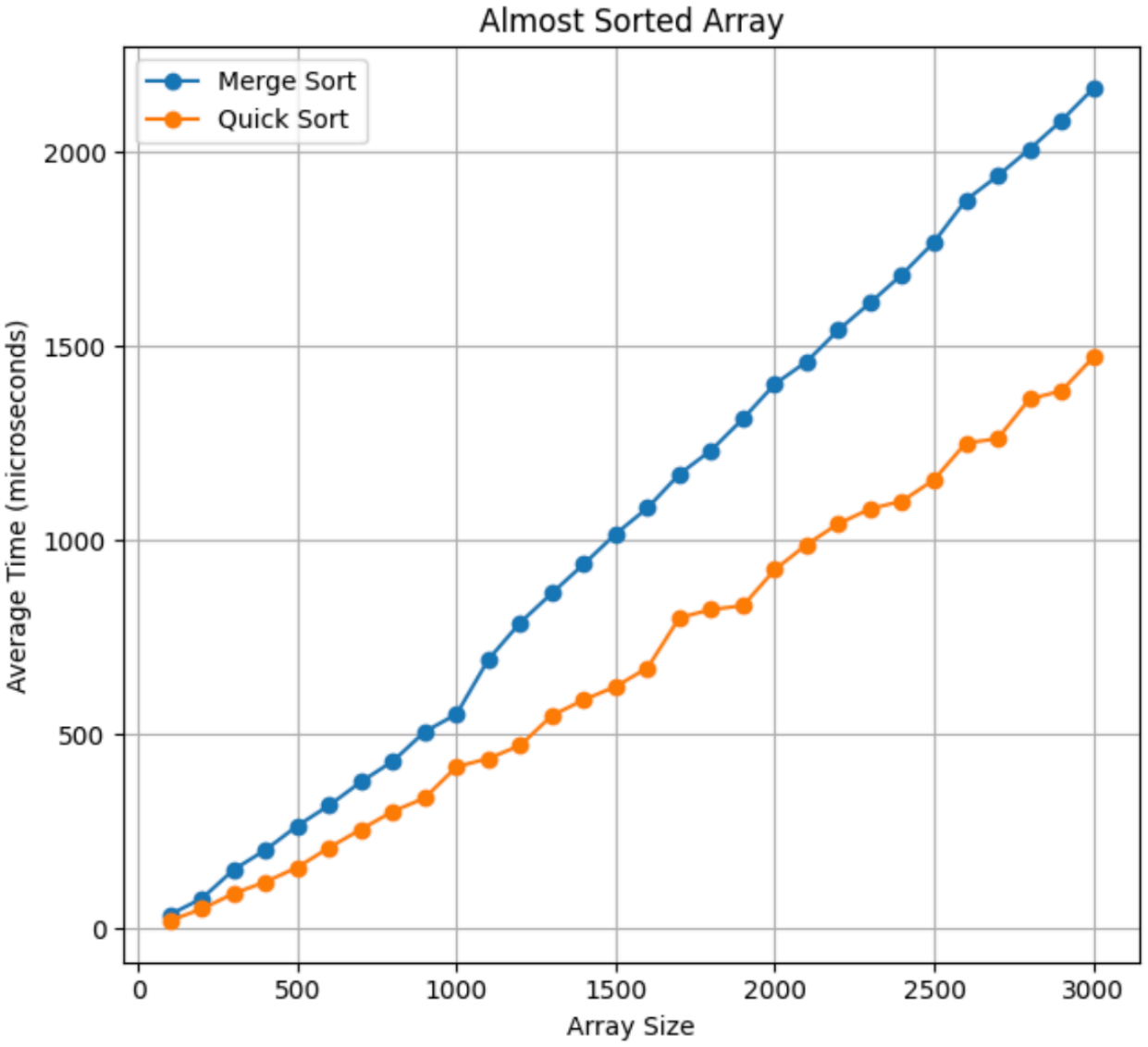
1. Правила изменения времени работы:
    - Обеспечьте минимальное влияние фоновых процессов и сетевого подключения на результаты измерений.
    - Выполните многократные замеры времени работы алгоритмов и рассчитайте среднее значение (конкретное количество замеров и метод усреднения выбираете самостоятельно).
    - Правильно подберите единицы измерения времени — слишком крупные единицы (например, секунды) могут дать трудно интерпретируемые результаты.
2. Создайте отдельный класс **StringSortTester** для реализации функций измерения производительности всех тестируемых алгоритмов сортировки

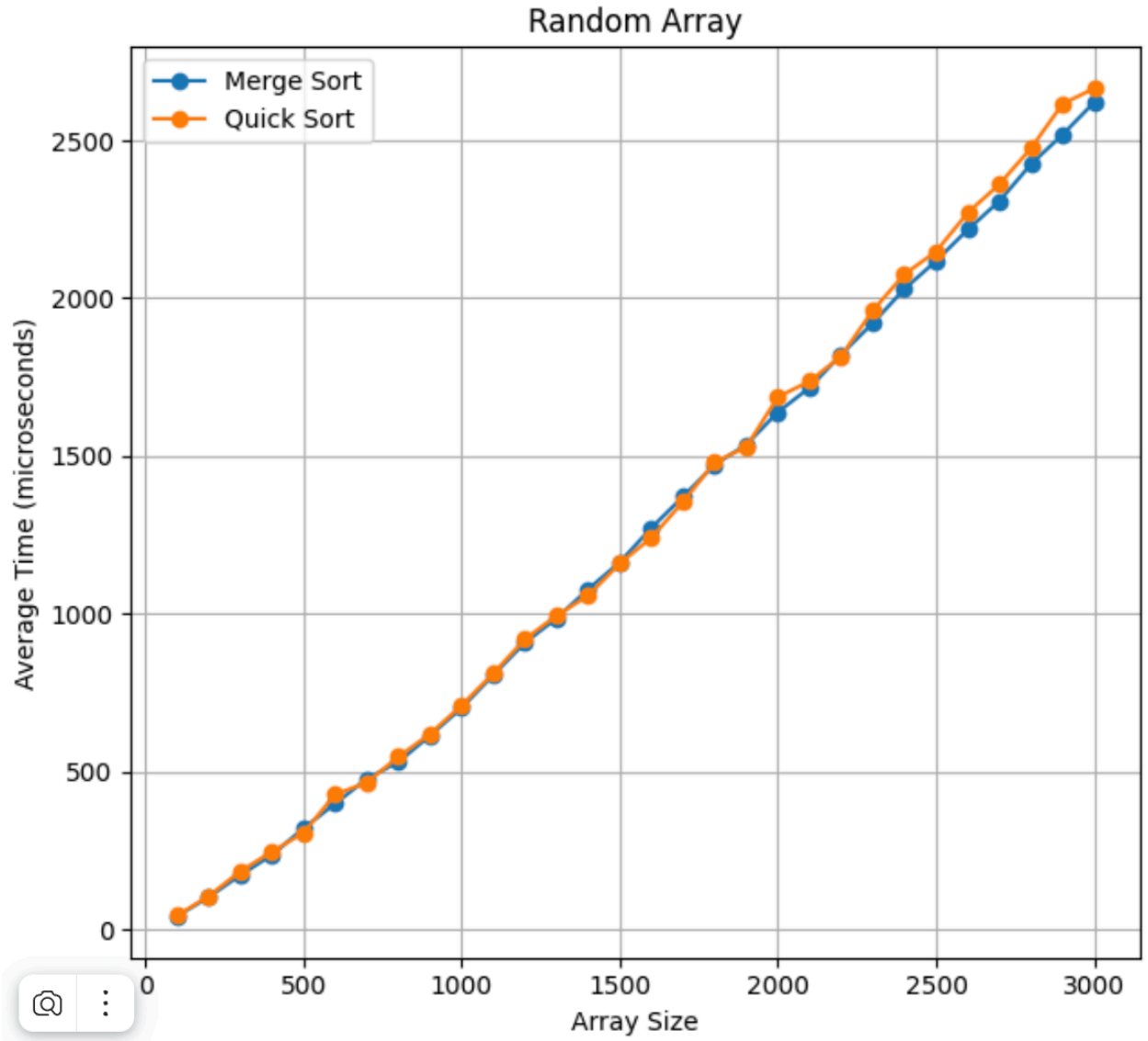
строк.

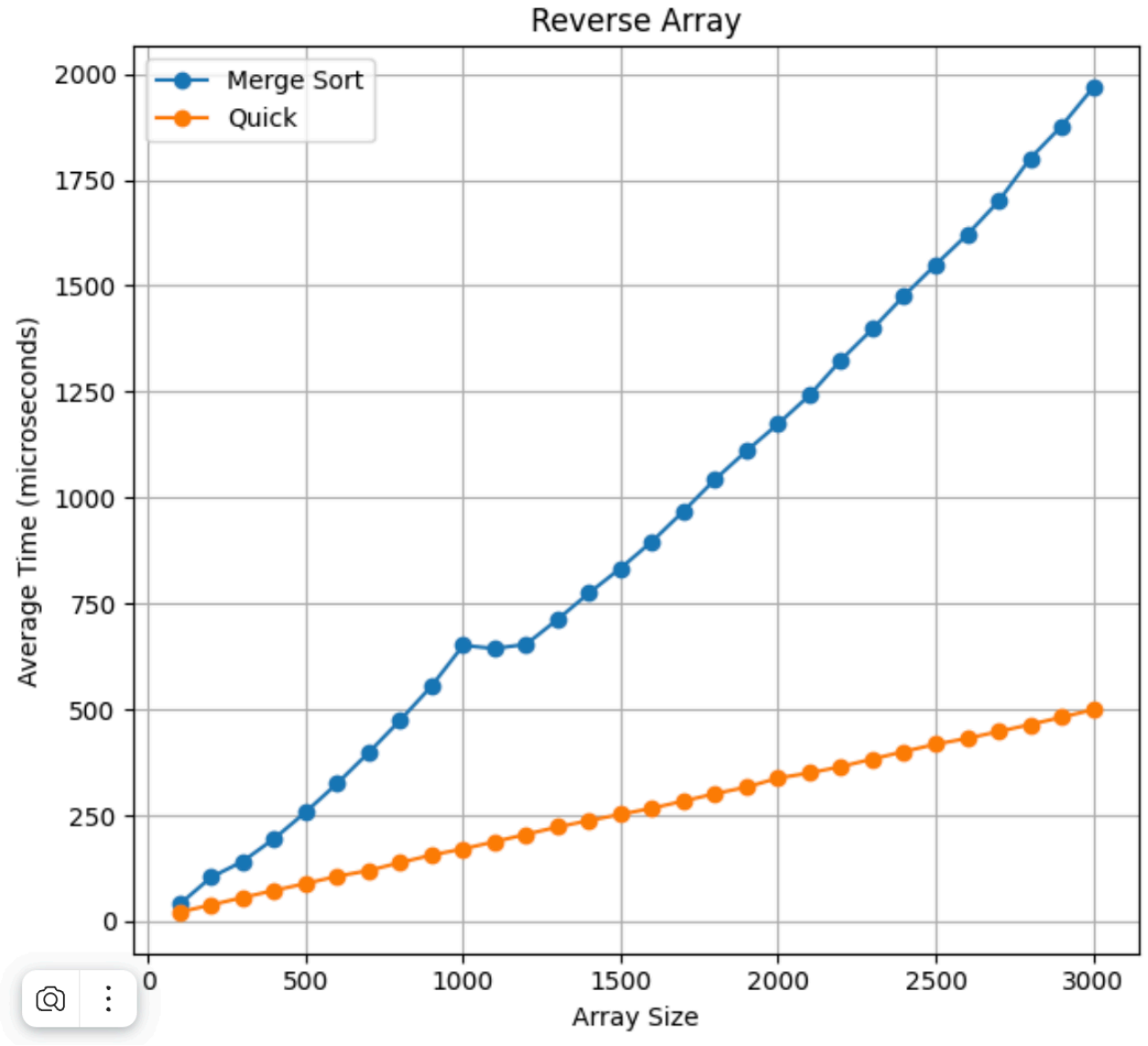
3. Для реализации алгоритма **STRING MERGESORT** используйте материалы конспекта из раздела **Неделя 14**.
   Для визуализации данных, использовался гугл колаб, и все данные записывались в .csv файлы, они тоже есть в репозитории.
   Далее посмотрим на графики.

Визуализация анализа

Сравним сначала стандартные сортировки

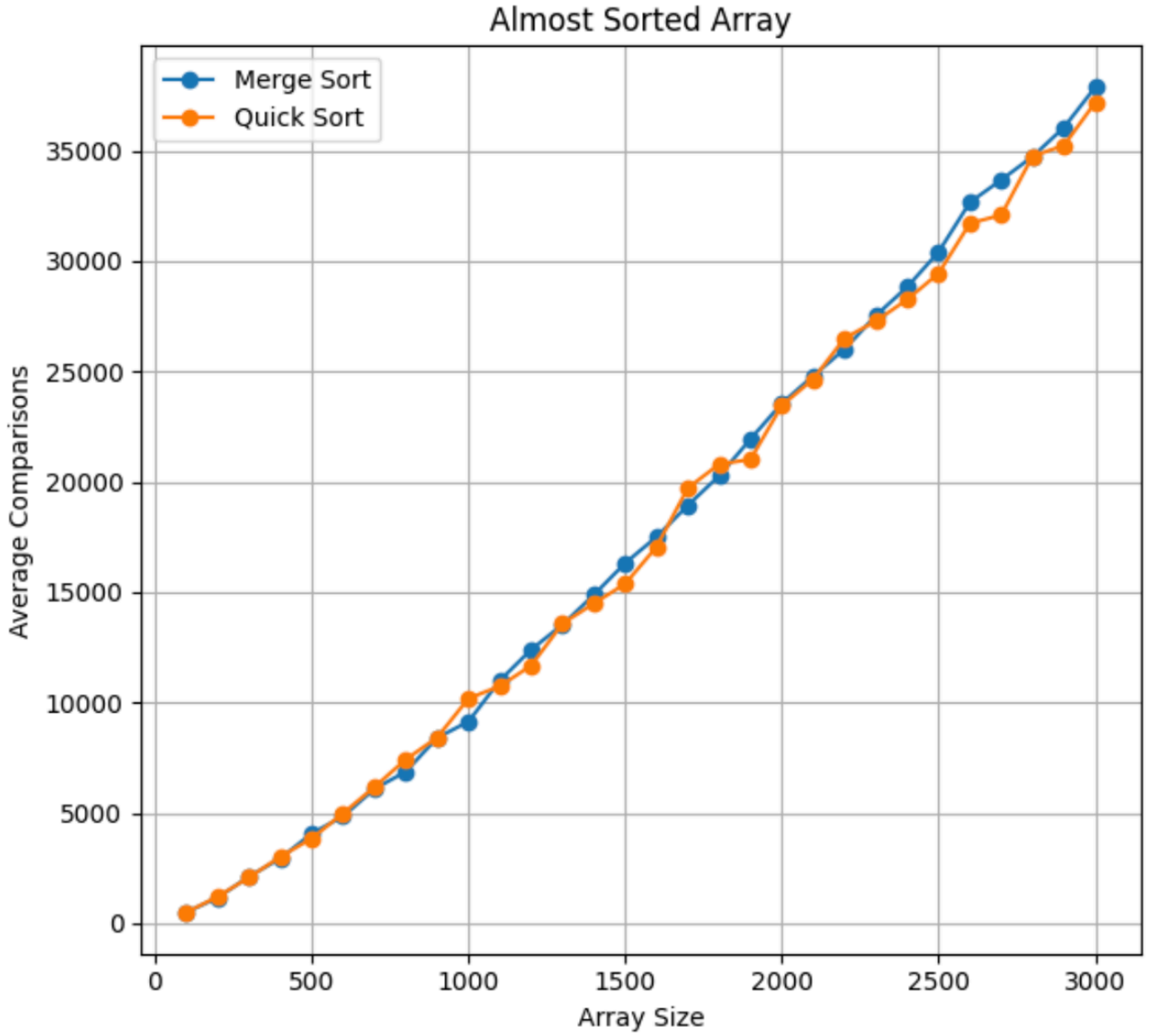


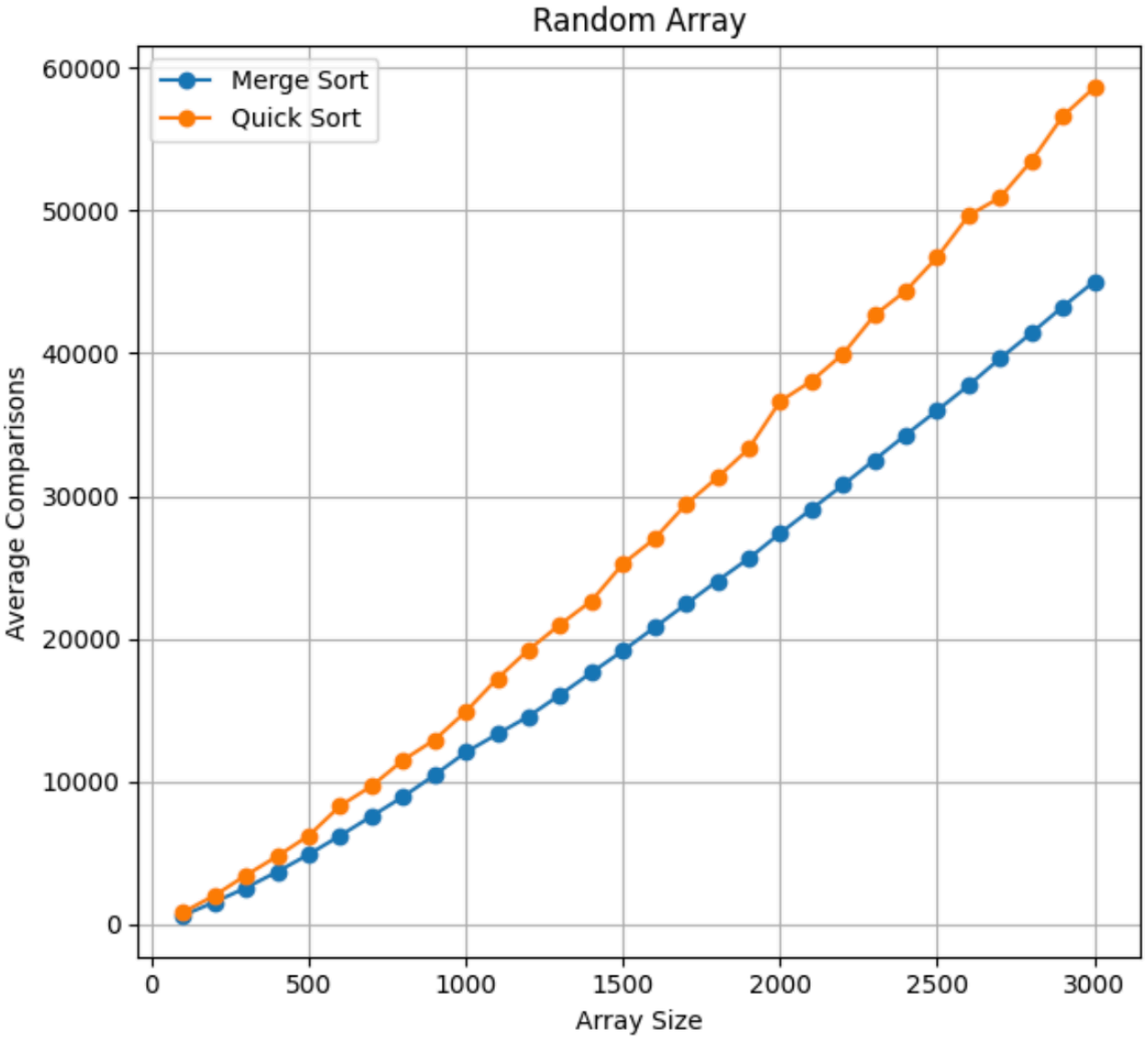

Random Array
Merge Sort
Quick Sort
Average Time (microseconds)
Array Size
0
500
1000
1500
2000
2500
3000

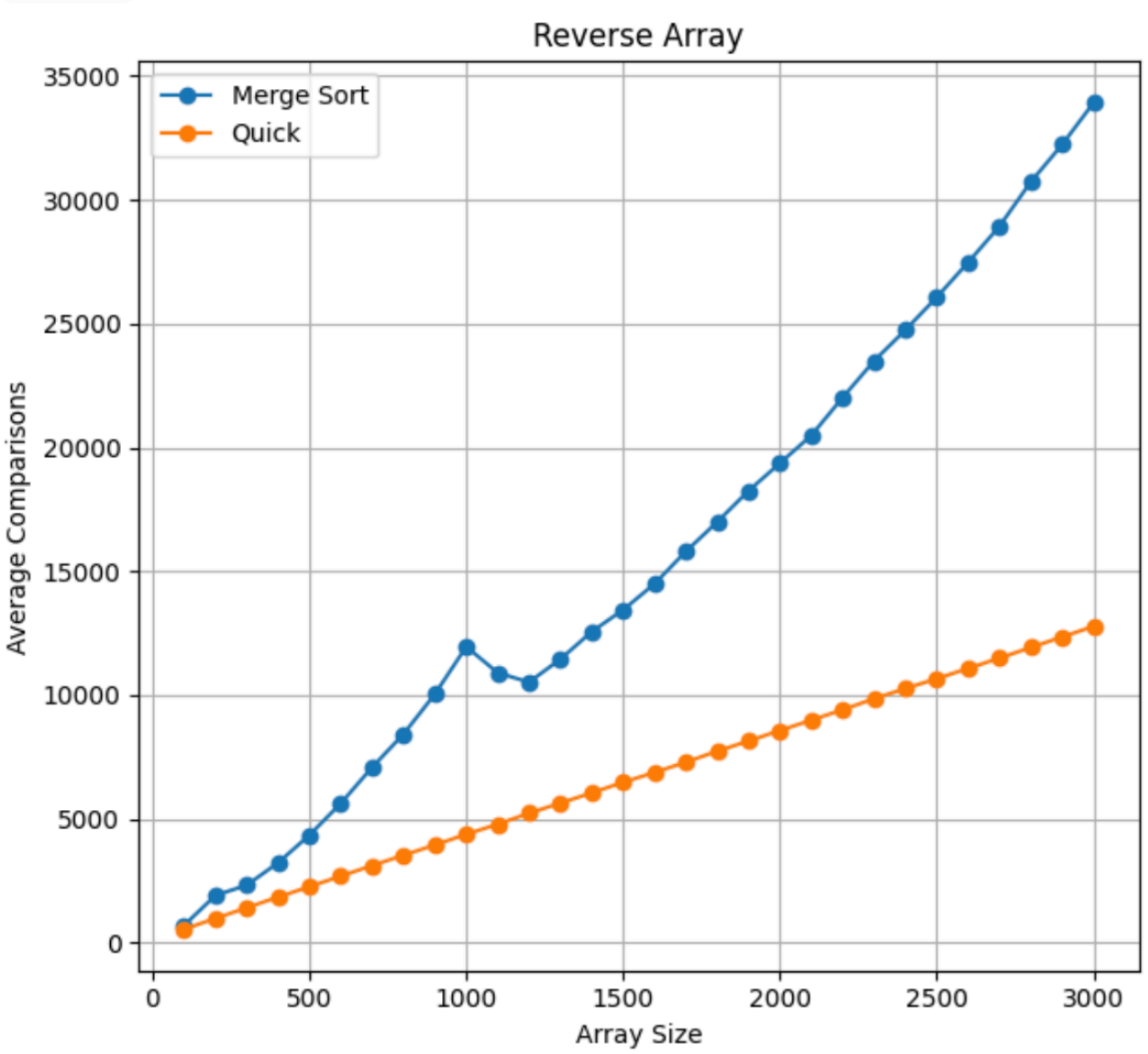


На данных скриншотах можно заметить что `QuickSort` лучше себя показал на почти отсортированных и реверснутых данных, на рандомных данных сортировки равны, наверное потому что в среднем обе имеют сложность $O(n \cdot \log n)$.

Рассмотри количество посимвольных сравнений (для их подсчета настандартных сортировках использовался кастомный компаратор)

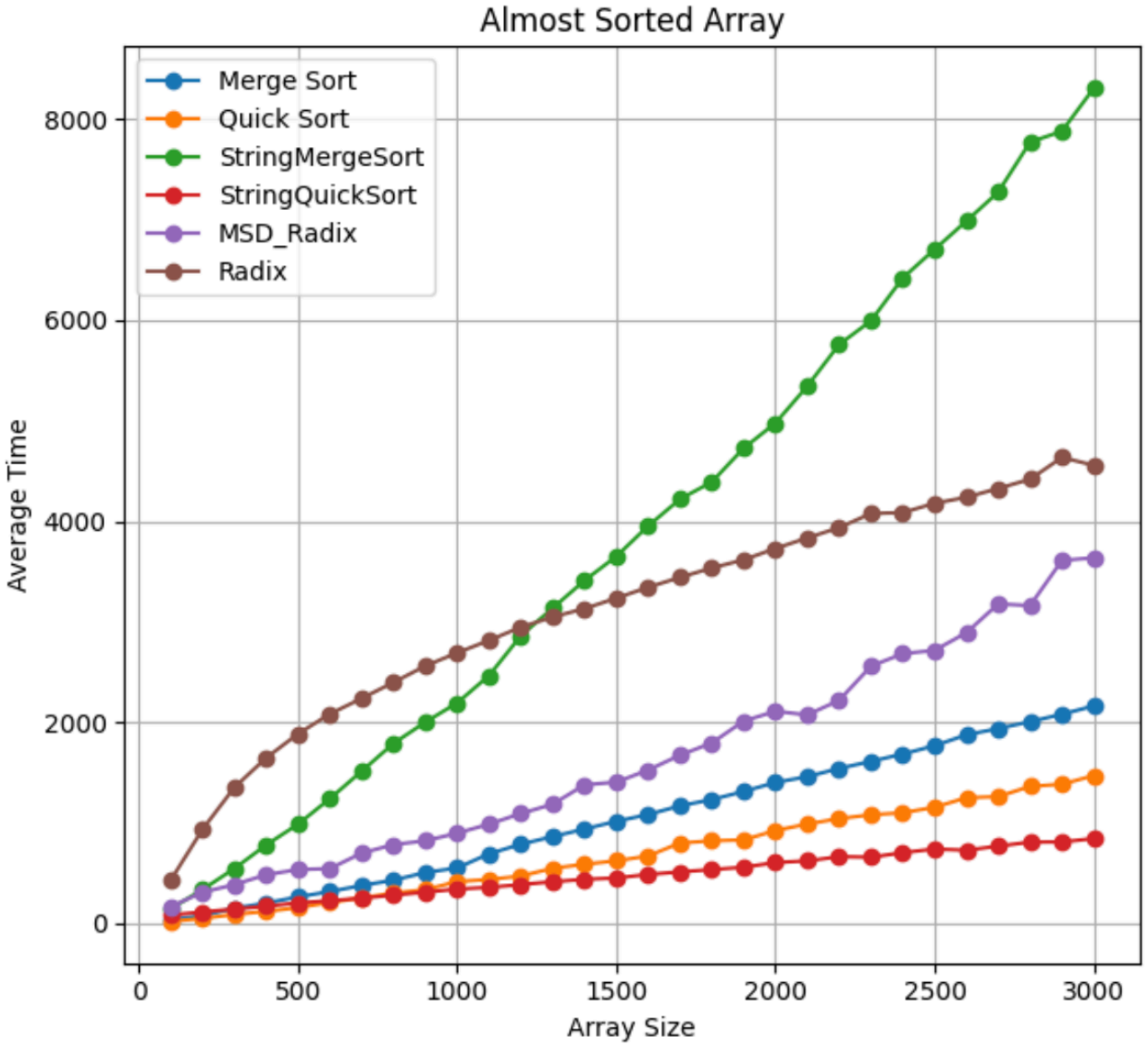

Almost Sorted Array
Merge Sort
Quick Sort
Average Comparisons
Array Size
0
5000
10000
15000
20000
25000
30000
35000
0
500
1000
1500
2000
2500
3000

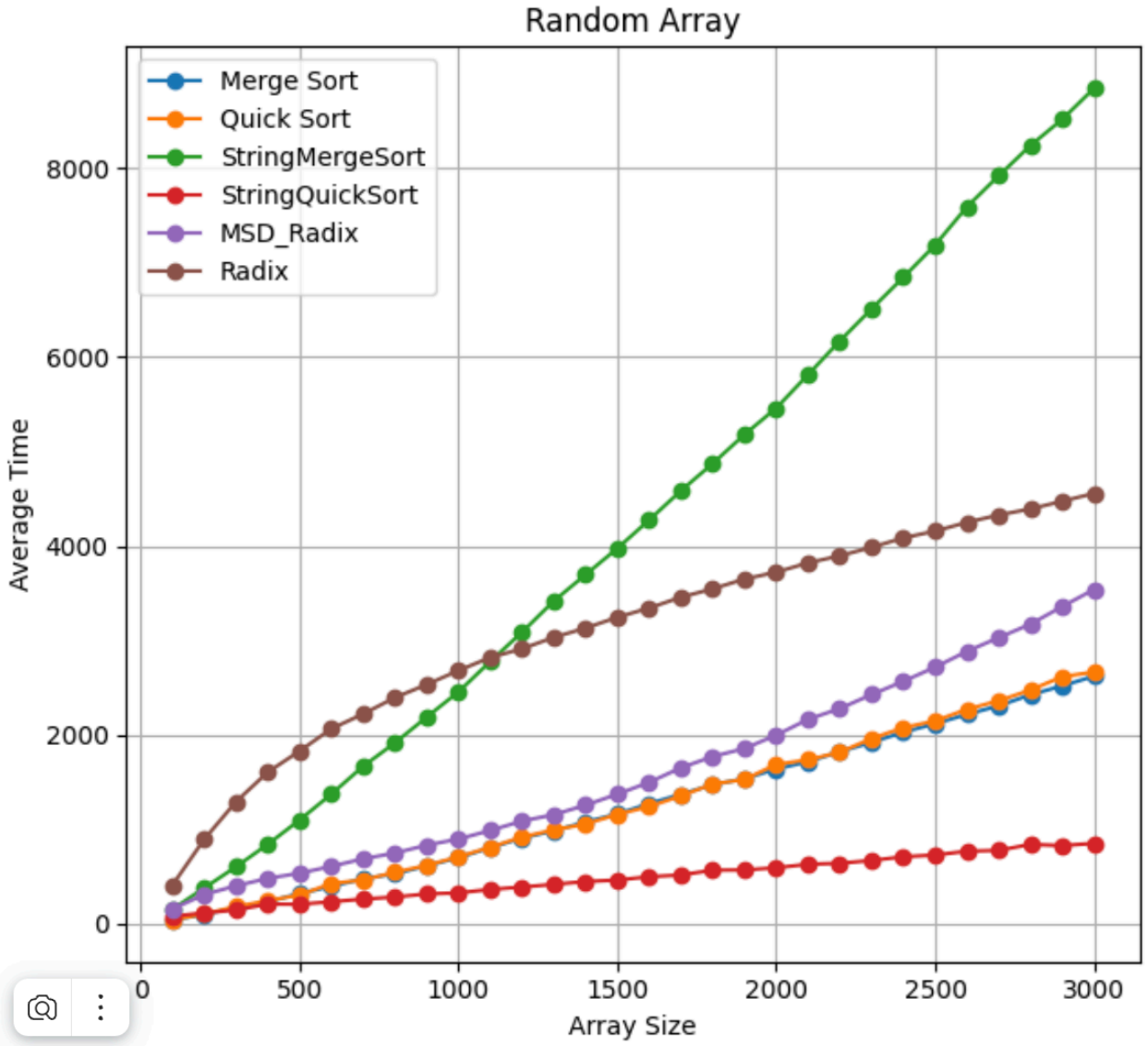

Random Array
Merge Sort
Quick Sort
Average Comparisons
Array Size
0
10000
20000
30000
40000
50000
60000
0
500
1000
1500
2000
2500
3000

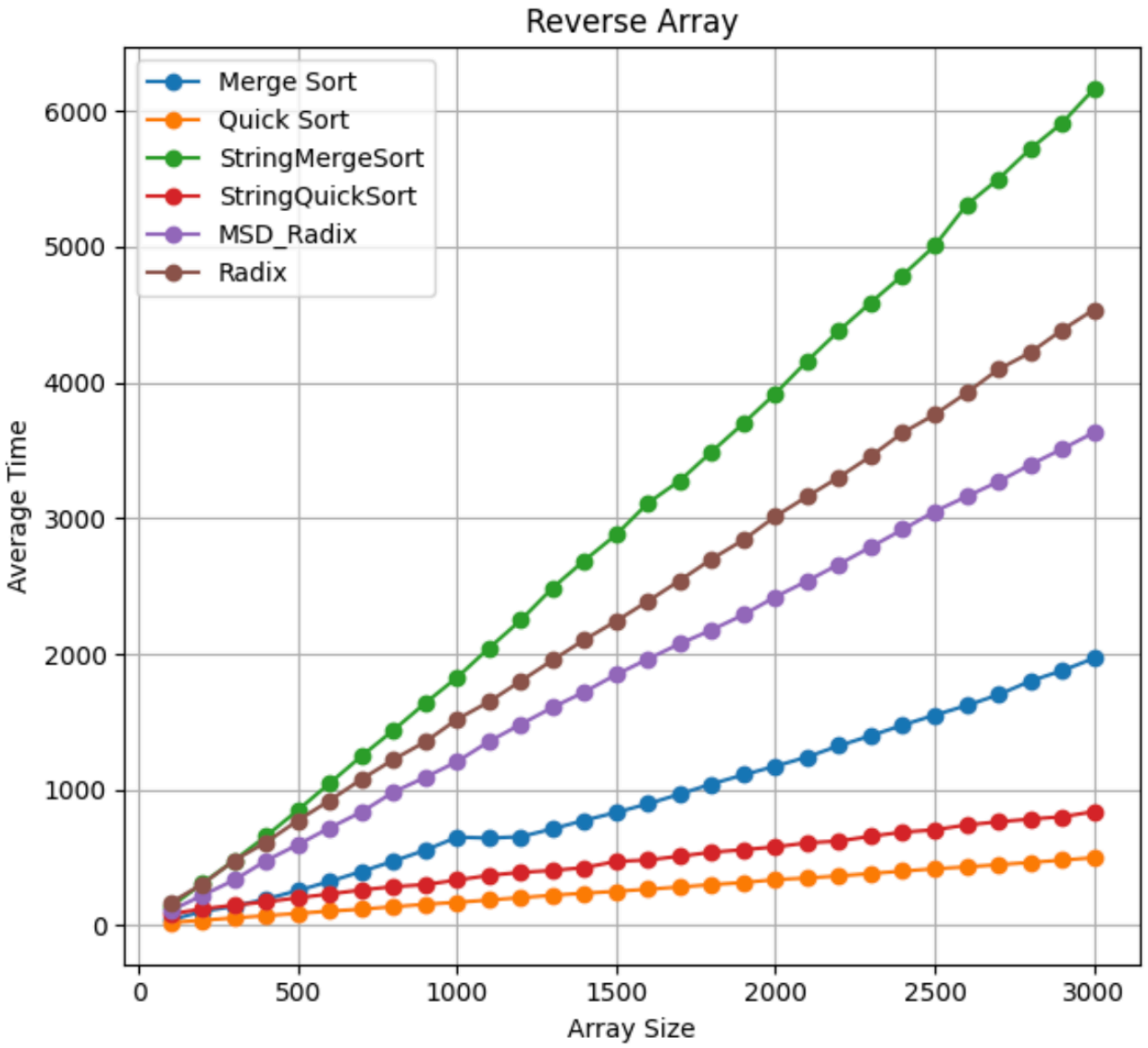


По графикам видно, что на почти отсортированных данных у сортировок количество сравнений почти не отличается. На рандомных данных `MergeSort` выигрывает, а на реверснутых наоборот.

Сравним теперь с кастомными сортировками

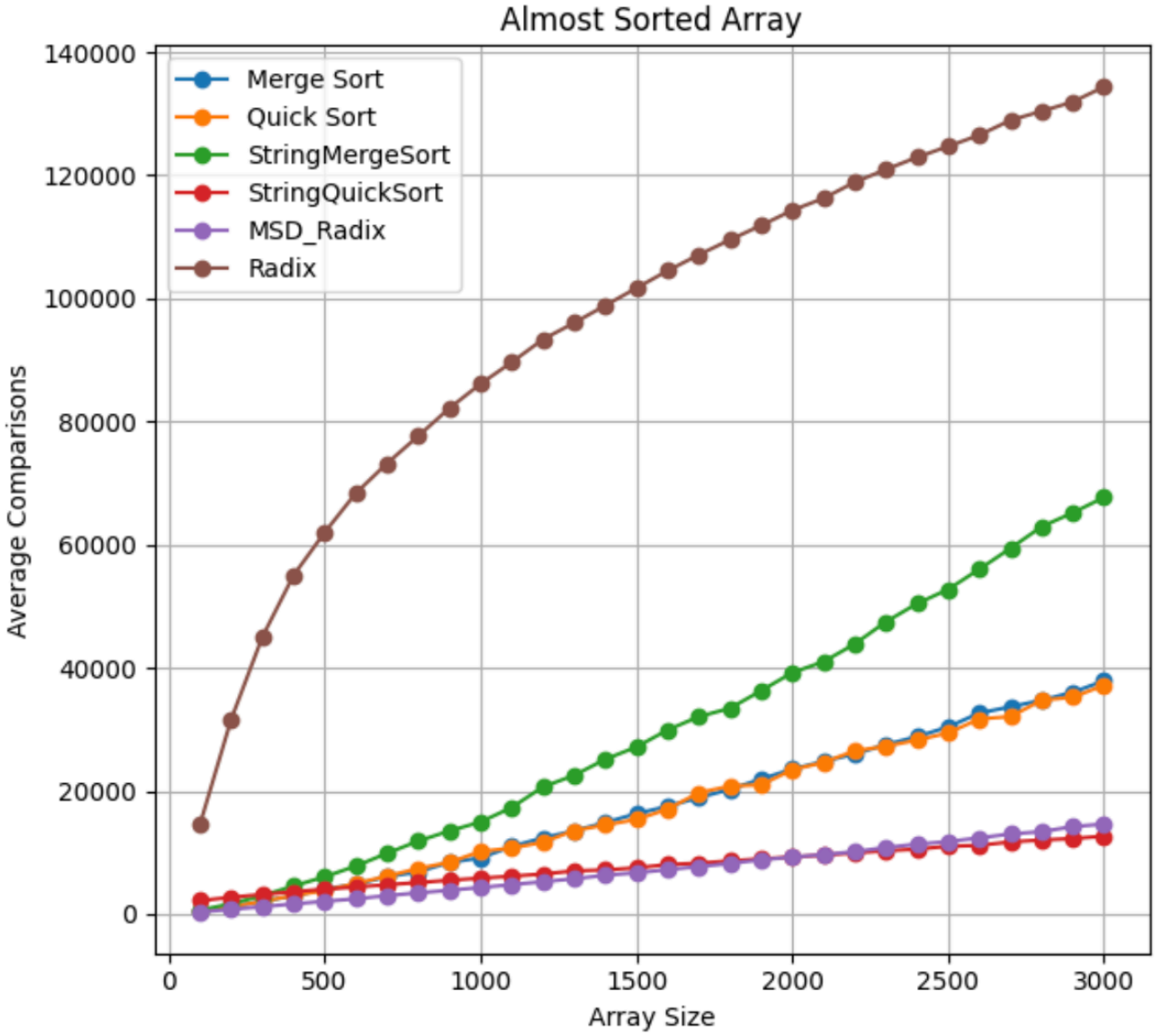

Almost Sorted Array
Merge Sort
Quick Sort
StringMergeSort
StringQuickSort
MSD_Radix
Radix
Average Time
Array Size
0
2000
4000
6000
8000
0
500
1000
1500
2000
2500
3000

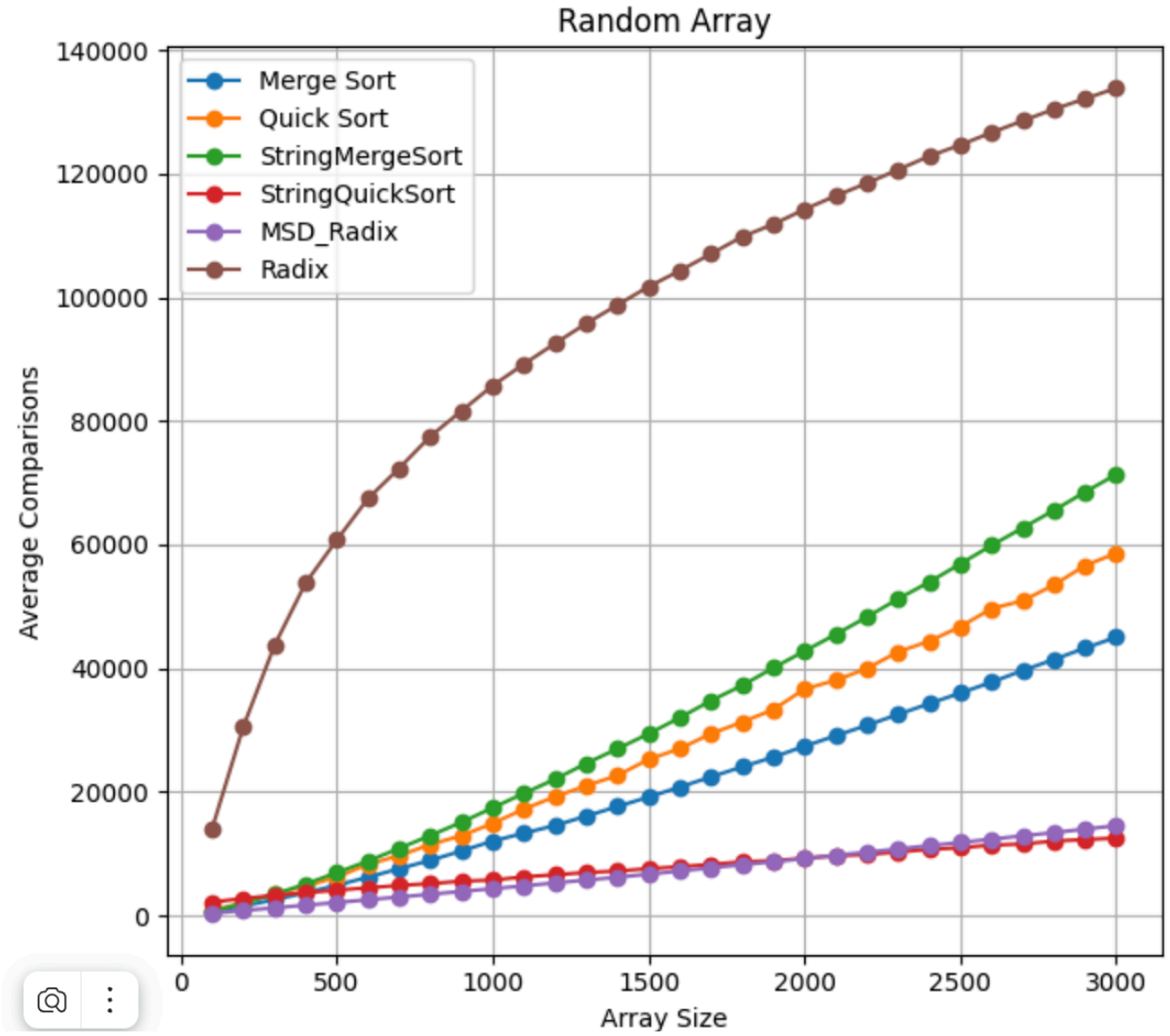

Random Array
Merge Sort
Quick Sort
StringMergeSort
StringQuickSort
MSD_Radix
Radix
Average Time
Array Size
0
2000
4000
6000
8000
0
500
1000
1500
2000
2500
3000

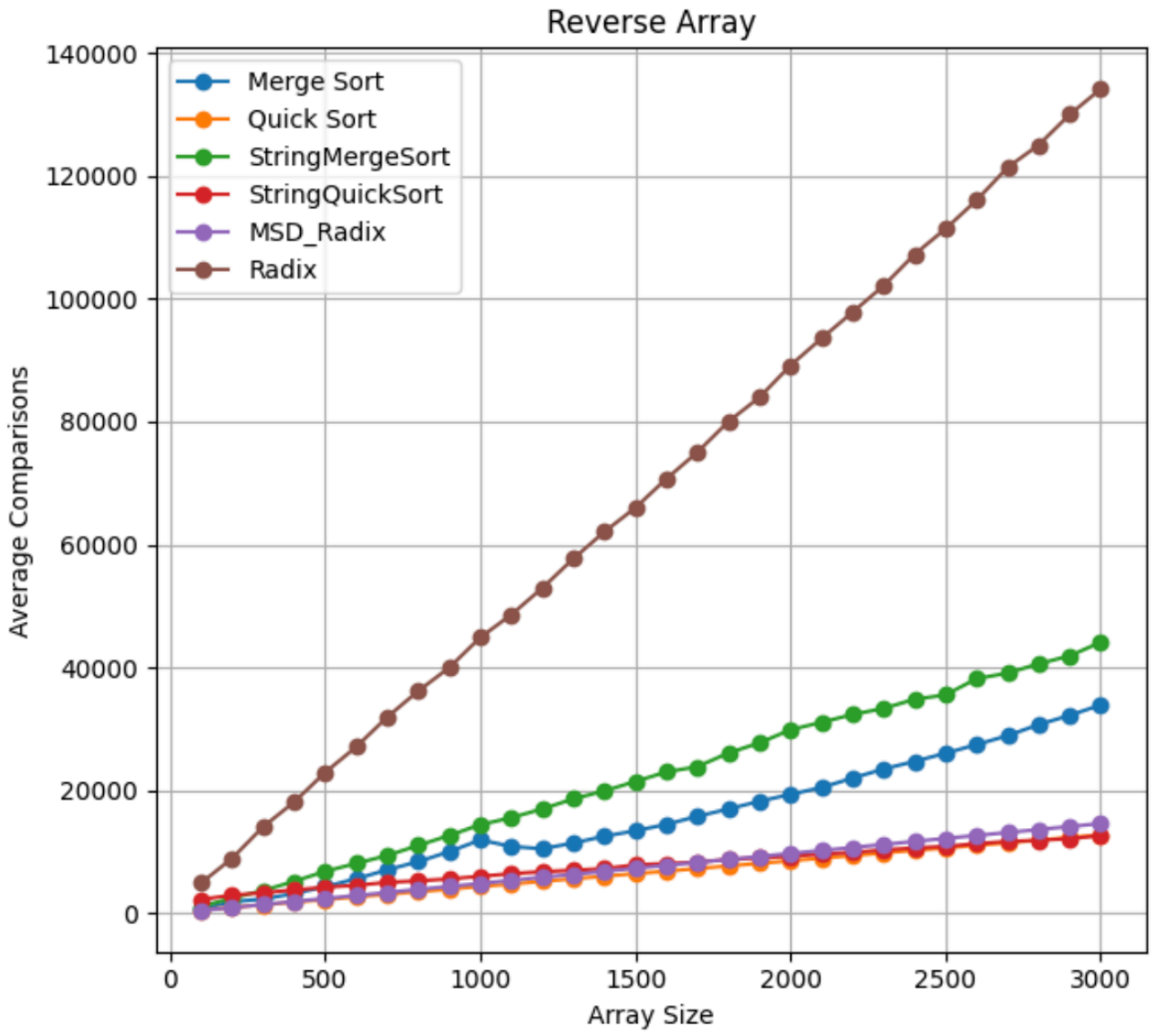


Можем заметить, что кастомный `StringMergeSort` в среднем на любых видах данных получился хуже по времени, чем все остальные сортировка. А вот `StringQuickSort` и `QuickSort` показали лучшие результаты на всех данных. `StringQuickSort` проигрывает по времени обычному, только на реверснутых данных, на остальных же сам берет верх. Остальные же сортировки не так оптимальны.

Посмотрим теперь на посимвольные сравнения




Random Array
Merge Sort
Quick Sort
StringMergeSort
StringQuickSort
MSD_Radix
Radix
Average Comparisons
Array Size
0
20000
40000
60000
80000
100000
120000
140000
0
500
1000
1500
2000
2500
3000



По графикам видно, что `RadixSort` имеет больше всего сравнений на всех видах и объемах данных, причем на почти отсортированных и рандомных график имеет форму похожую, чем-то на функцию корня. а `StringQuicSort` и `MSD_RadixSort` наоборот.

В итоге можно сказать, что в ходе данного ЭЭЭЭЭксперемента, стринговый куик сорт оказался не на реверснутых данных лучше своего "классичесского" аналога, а стринговый мердж сорт, как-то наоборот во всем оказался хуже...
`MSD_RadixSort` и `RadixSort` все же оказались по времени хуже сортировок из STL.

Как-то так...

> “ Мы не можем не совершать ошибки. Они — часть нашей жизни. А самое главное — они помогают нам найти правильный путь. ” – Максим Лавров.