Бершицкий Дмитрий, БПИ234

A1. Анализ строковых сортировок

1. Подготовка тестовых данных

(Все файлы и данные доступны по ссылке https://github.com/brshtsk/A1_9_AiSD)

При вызове конструктора класса **StringGenerator**, создаются массивы строк таких видов:

```
enum GenerationMode {
  RANDOM,
  REVERSED,
  ALMOST_SORTED,
  MATCHING_PREFIX
};
```

1) Random – массив неупорядоченных случайных строк
2) Reversed – строки отсортированы в обратном лексикографическом порядке
3) Almost sorted – строки были отсортированы, затем 10% пар строк перемешано (при этом если s_i и s_j поменяли местами, то $|i - j| \leq 30$)
4) Matching prefix – у строк встречаются общие префиксы. Всего доступно 4 префикса, которые создаются заранее

2. Модуль тестирования

Тестирование алгоритмов сортировки происходит в классе **StringSortTester**. Каждый алгоритм тестируется на всех видах выборок и на всех размерах массивов в диапазоне $[100; 3000]$, с шагом 100.

Для получения усредненных результатов, в **main.cpp** тесты вызываются по 5 раз. В дальнейшем в **graphs.ipynb** рассматриваются средние показатели по каждому из 5 прогонов для каждого теста.

3. StandardQuickSort

Производит простое посимвольное сравнение строк.

Опорный элемент (pivot) выбирается **случайным образом** на отрезке $[l; r]$:

```
// Генератор случайных чисел для выбора опорного элемента
std::uniform_int_distribution<int> dist(l, r);
int pivotIndex = dist(gen);
int newPivotIndex = Partition(strings, l, r, pivotIndex, comparisonCount);
```

4. StandardMergeSort

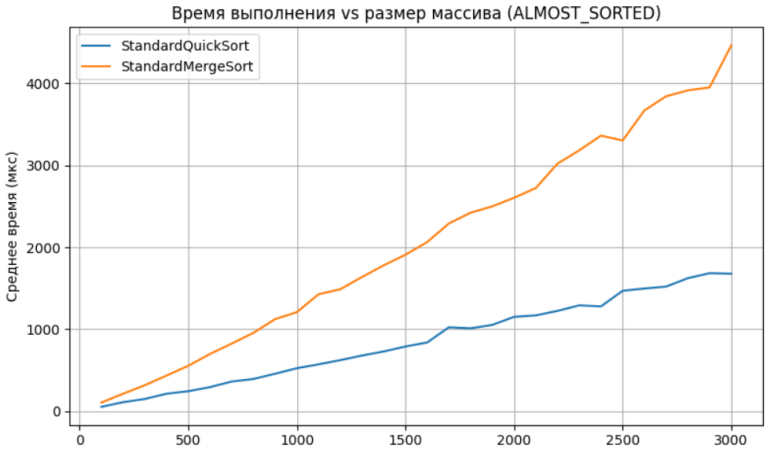
Производит простое посимвольное сравнение строк.

Использует 2 массива: **strings** и **temp**.

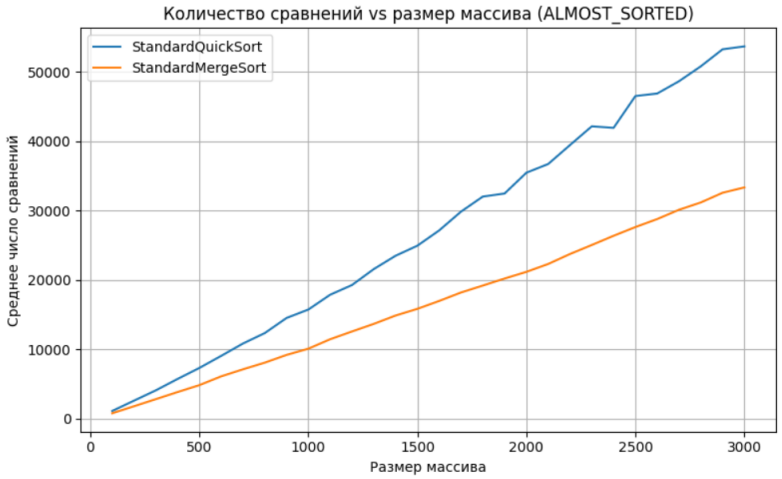
5. Анализ стандартных алгоритмов

Сравним StandardQuickSort и StandardMergeSort. (Графики доступны в **graphs.ipynb**)

QuickSort показал **выигрыш по времени** во всех тестах. Самый значительный отрыв на почти отсортированных массивах:



Хотя при этом количество сравнений у MergeSort на этом же тесте меньше, чем у QuickSort:



Более того, **MergeSort** совершает **меньшее количество сравнений** и на всех остальных типах массивов.

Это связано с накладными расходами MergeSort: использование двух массивов замедляет работу кода, а реализация in-place была бы довольно трудоемкой. Хотя MergeSort требует меньше посимвольных сравнений строк, на практике он медленней.

Меньшее количество сравнений в MergeSort можно связать с тем, что на этапе слияния массивов L и R в тот момент, когда все строки одного из двух массивов будут обработаны, строки второго массива автоматически “вольются” без дополнительных проверок.

6. StringQuickSort

Посылка: **321300227**.

Опорный элемент (pivot) выбирается **случайным образом** на отрезке $[l; r]$.

Работает за счет сравнения строк по “текущему” индексу. In-place создает отрезки $[l; lessThanIndex), [greaterThanIndex + 1, r]$.

7. StringMergeSort

Посылка: **321301245**.

При сливе L и R смотрит на LCP, чтобы ускорять выбор наименьшей строки. При сортировке использую ноды. Для оптимизации в них указатели:

```
struct Node { const std::string *str; int lcp; };
```

Использует 2 массива: **nodes** и **temp**.

8. BasicMsdRadixSort

Посылка: **321301816**.

Для оптимизации вместо std::string использую std::string_view. Заранее создаю 128 бакетов (коды наших символов помещаются в отрезок от 0 до 128):

```
std::array<std::vector<std::string_view>, 128> buckets;
```

Далее сортировка происходит внутри каждого бакета.

9. BoostedMsdRadixSort

Посылка: **321303383**.

Идентичен прошлому коду, за исключением фрагмента:

```
if (strings.size() < 74) {
  sqs_view::StringQuickSort(strings, generalPrefix, comparisonCount);
  return;
}
```

В этом же файле создан namespace sqs_view, в котором прописан StringQuickSort, идентичный ранее написанному StringQuickSort, но уже с std::string_view10.11. вместо std::string. Удобно вызывать StringQuickSort, зная длину нынешнего общего префикса.

10. Анализ адаптированных алгоритмов

Построю графики, показывающие статистику сразу по адаптированным и стандартным алгоритмам. (Также доступно в **graphs.ipynb**)

Перед анализом графиков вспомним **теоретические оценки**:

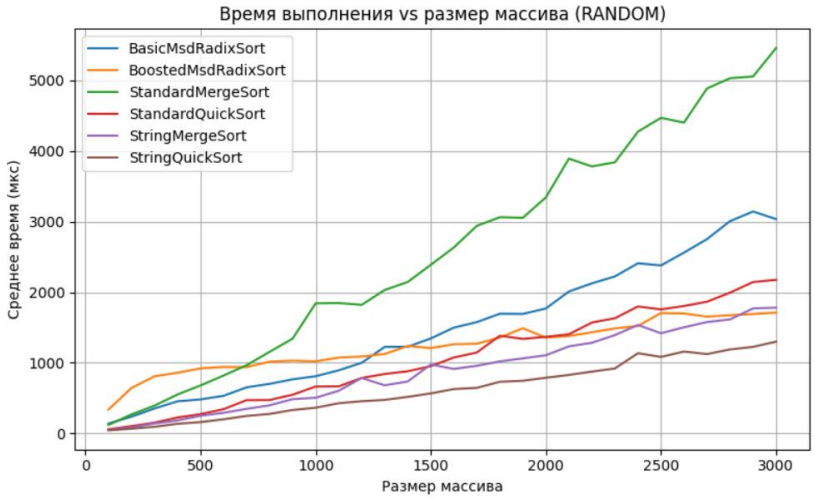
Алгоритм	Среднее	Худшее
StandardQuickSort	O(n log n · L)	O(n^2 · L)
StandardMergeSort	O(n log n · L)	O(n log n · L)

StringQuickSort	O(D + n log n)	$O(n^2 \cdot L)$
StringMergeSort	O(n log n + D)	O(n log n · L)
BasicMsdRadixSort	O(n · L + R)	O(n · L + R)
BoostedMsdRadixSort	O(n · L + R)*	$O(n \cdot L + n^2 + R)$

Где D – общая длина различающих префиксов, L – максимальная длина строки,
R – мощность алфавита.

* + O(n log n) на мелких фрагментах.

A) Случайный массив



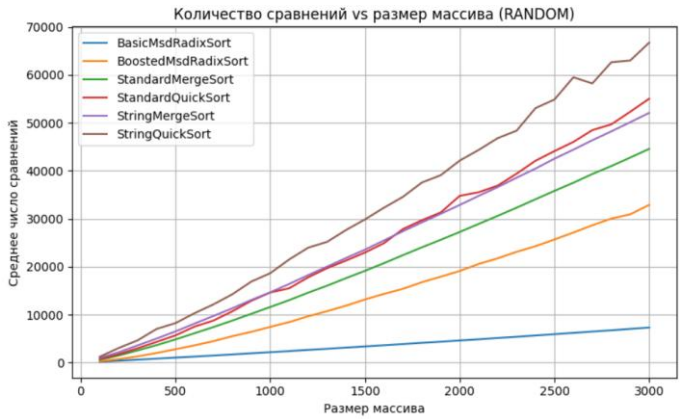
StringQuickSort сильно **обгоняет другие алгоритмы** как на маленьких, так и на больших выборках благодаря низким накладным расходам и in-place вычислениями, несмотря на “не лучшую” асимптотику.

Также заметим, что **на больших выборках BoostedMsdRadixSort работает быстро**, почти вдвое быстрее BasicMsdRadixSort. Действительно, несмотря на линейную асимптотику, при слишком глубокой рекурсии BasicMsdRadixSort показывает плохой результат, в какой-то момент проще сравнивать строки без группировки по бакетам.

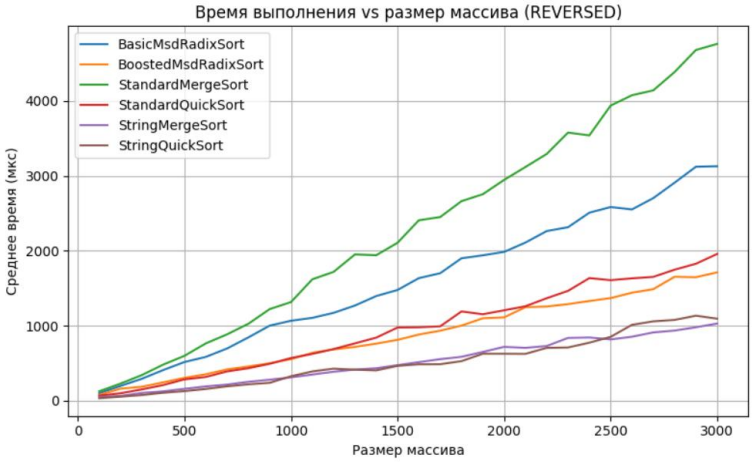
При этом **на маленьких выборках** (от 100 до 600) **BoostedMsdRadixSort работает гораздо медленнее** остальных: алгоритм тратит много времени на накладные расходы, а спустя 1-2 итерации просто переключается на StringQuickSort.

Вполне ожидаемо, что **StandardMergeSort – самый медленный**.

У **StringQuickSort больше всего сравнений**. Это объясняет тернарность (проверки на <, >, =):



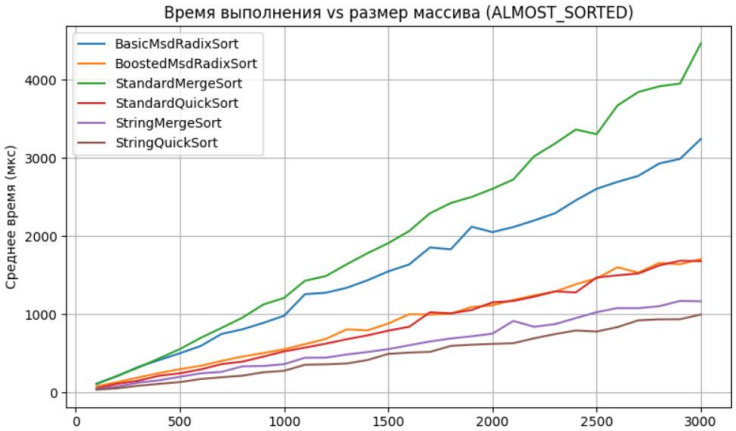
Б) Обратно отсортированный массив



Быстрее всех StringMergeSort, немного опережая StringQuickSort. Достаточно логично, так как при разбиении подмассивов на L и R, элементы из R всегда будут больше, чем из L. Таким образом, после сравнения всех r_i с l_0, **получается "влить" L без дополнительных проверок**. (Работа с LCP еще сильнее ускоряет работу при переборе r_i).

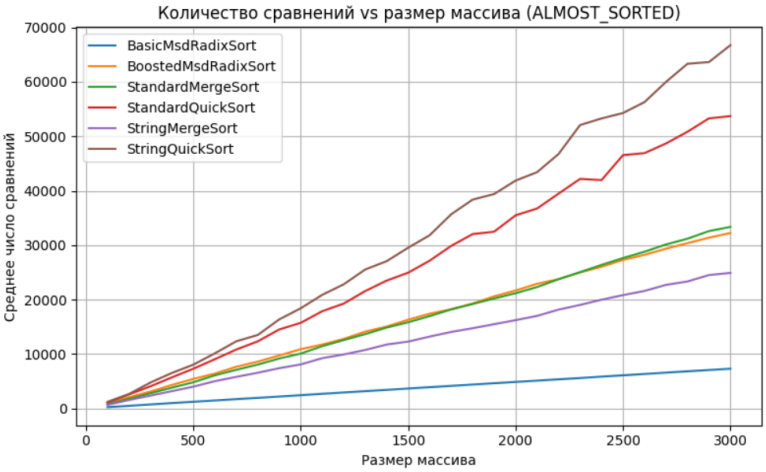
Видно, что **StringMergeSort** проводит настолько **мало сравнений**, что он выигрывает по времени даже несмотря на не лучшую оптимизацию.

В) Почти отсортированный массив



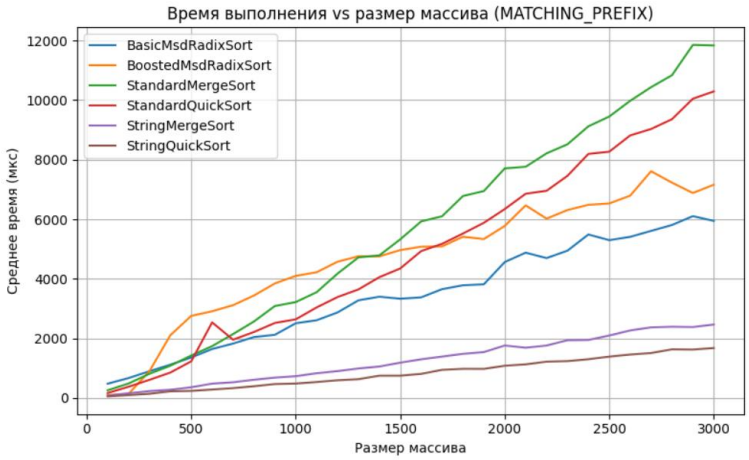
Результат ожидаемый: **в лидерах StringQuickSort и StringMergeSort**. Эти алгоритмы нацелены на кеширование совпадающих префиксов, что ускоряет работу на почти отсортированных выборках.

BoostedMsdRadixSort становится не выгодным – StringQuickSort проще и быстрее. В итоге потери в производительности настолько сильны, что **вровень идет StandardQuickSort**.



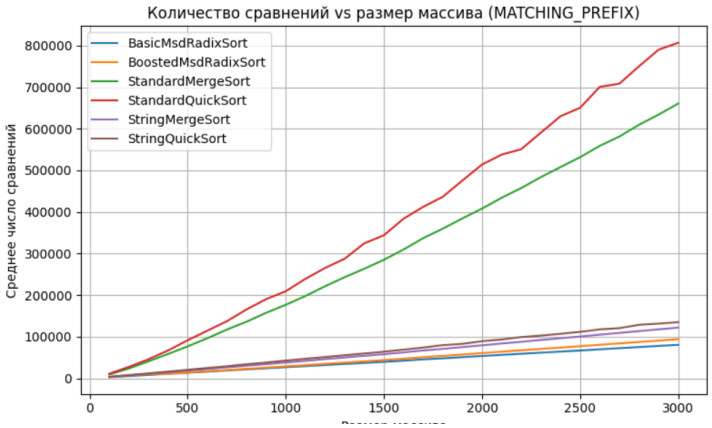
StringMergeSort производит не так много сравнений, но все-таки до уровня обратно отсортированного массива не дотягивает.

Г) Массив строк с общими префиксами



Аналогично прошлому тесту, **StringQuickSort и StringMergeSort** хорошо работают, когда префиксы повторяются.

Но вот что интересно: **BasicMsdRadixSort обошел BoostedMsdRadixSort и стандартные алгоритмы**. В массиве строк с общими префиксами у строк может быть один из 4 заранее заданных префиксов (длиной 10-40 символов). В итоге бакет BasicMsdRadixSort заполняется не на 74 элемента, а просто на 4, что ощутимо упрощает дерево рекурсии и ускоряет вычисления.



Обработка префиксов у **StringQuickSort и StringMergeSort** настолько хороша, что количество сравнений почти доходит до “идеальных” показателей **BasicMsdRadixSort**.