Отчет. А1. Григорьев Владимир.
ID в CF: 321298336, 321298505, 321299103, 321299343
Репозиторий: https://github.com/vovaggri/A1_alorithms

Задачи:

1. Реализовать генератор тестов (StringGenerator)
2. Измерить std::sort и std::stable_sort
3. Измерить тернарный QuickSort, MergeSort+LCP, MSD-Radix, MSD-Radix+QuickSwitch
4. Построить графики времени и числа посимв. сравнений
5. Сравнить результаты с теорией

Этап 1.

StringGenerator

- Описание: какие символы (74 шт.), длины строк (10–200), размеры наборов (100…3000 шаг 100).
- Типы массивов:
- случайный
- обратный (reverse)
- почти отсортированный (almost)

Структура замеров

- Для каждого размера n и каждого типа данных генерируется один «эталонный» вектор и копируется 6 раз.
- Перед каждым запуском счётчик comparisonCount обнуляется.
- Время измеряется через chrono::high_resolution_clock.
- Сохраняем CSV со столбцами algorithm,dataType,n,timeMs,comparisons.

Этап 2. Стандартные алгоритмы
Все графики можно посмотреть в репозитории:
https://github.com/vovaggri/A1_alorithms

1. std_sort vs stable_sort (random data)

График времени

- Оба алгоритма демонстрируют почти линейный рост времени от 0 до ~5.5 мс при n=3000.
- stable_sort чуть медленнее std::sort для малых n (до ~1000 строк), но при n≈1500–1700 дела выравниваются.
- На больших n (>2000) заметен небольшой перевес std::sort (~5.6 мс) над stable_sort (~6.7 мс)… но разница совсем невелика.

График сравнений

- std_sort и stable_sort делают примерно одинаковое число посимв. сравнений: для случайных около 58 000 при n=3000, для «reverse» — около 12 500, для «almost» — около 22 000.
- Линейный характер — почти прямая линия: сравнения ≃ C·n·log n, где константа C зависит от входа (C_random>C_almost>C_reverse).

Выводы

- std::sort на случайных данных выигрывает у stable_sort лишь на 5–10 % по времени, но оба укладываются в Θ(n log n) сравнений.
- stable_sort чуть более предсказуем (почти ровная кривая), тогда как у std::sort есть локальные всплески (внутренние перераспределения).

2. std_sort vs quick3way (random data)

Время

- При n=3000 std_sort ≃5.4 мс, а quick3way — лишь ≃1.4 мс.
- quick3way стабильно быстрее уже с n≈500, разрыв растёт линейно: алгоритм выигрывает за счёт меньшего числа посимв. сравнений и эффективной трёх-путевой партиции.

Сравнения

- quick3way делает около 36 000 сравнений при n=3000 (для random), в то время как std_sort — ≃58 000.
- Особенно выгоден quick3way при случайных строках, где ветвления «<», «=» и «>» уравновешены.

Выводы

- TERNS… STRING QUICKSORT превосходит стандартный QuickSort по числу сравнений почти вдвое и по времени — в 3–4 раза.

3. std_sort vs merge_lcp (random data)

Время

- Для малых n (<600) merge_lcp чуть медленнее, но примерно сопоставимо.
- При n≈600, 1500, 2300 и далее — видны большие «пики» (10-6 мс, 6.5 мс, 6.2 мс, …). Эти пики — это моменты, когда LCP-метод встречает очень длинные общие префиксы и затрачивает много времени на их подсчет.
- В остальное время merge_lcp примерно на 20–30 % медленнее std_sort.

Сравнения

- merge_lcp делает значительно больше посимвольных сравнений: ≃75 000 при n=3000 на random, против ≃58 000 у std_sort.
- Расходуется лишний O(lcp) внутри слияния, когда алгоритм каждый раз «сканирует» общий префикс.

Выводы

- В случайных данных, где общие префиксы малы, LCP-Merge всё равно чаще тратит лишние операции на их вычисление — из-за накладных расходов она проигрывает std_sort.

4. std_sort vs msd_radix (random data)

Время

- msd_radix стартует с ~0.3 мс при n=100 и плавно растёт до ~10.7 мс при n=3000.
- std_sort в тех же точках — от 0.05 до ~5.4 мс. msd_radix медленнее почти вдвое, особенно при больших n из-за большого числа распределений по «корзинам».

Сравнения

- msd_radix не использует посимвольных сравнений (они считаются равными 0), что отражено «нулевой» линией.
- Это ожидаемо: radix-сорт вовсе не сравнивает символы, а оперирует разбивкой по ключам.

Выводы

- По времени MSD-Radix пока уступает std_sort на случайных строках из-за стоимости распределения, но выигрывает по числу сравнений
- Для очень длинных строк или огромных n Radix может стать выгоднее, но при $n \leq 3000$ выигрыша нет.

5. std_sort vs msd_radix_qs (random data)

Время

- msd_radix_qs (оранж.) — от ~0.1 мс при n=100 до ~2.3 мс при n=3000.
- То есть в 2–3 раза быстрее, чем чистый MSD-Radix, и даже быстрее std_sort, начиная с $n \approx 500$.

Сравнения

- График показывает небольшое число сравнений (введённых в quickswitch-ветвях), порядка 50 000 при n=3000, что почти наравне с std_sort.
- Переключаясь на QuickSort там, где «датчики» показывают малый фрагмент, мы минимизируем накладные расходы.

Выводы

- Гибрид MSD-Radix + QuickSort даёт лучшее время: совмещает нулевые сравнения на больших фрагментах со скоростью QuickSort на мелких.

6. stable_sort vs quick3way (random data)

- quick3way выигрывает у stable_sort ещё сильнее, чем у std_sort: ~1.4 мс vs ~3.3 мс при n=3000.
- Характер кривых похож на пункт 2, но stable_sort здесь из-за внутреннего merge-алгоритма ещё медленнее.

7. stable_sort vs merge_lcp (random data)

- Аналогично: merge_lcp часто медленнее stable_sort (пики до 10 мс), сравнения растут выше 75 000.
- Слияние с LCP влечёт избыточные подсчёты, которые не компенсируются даже стабильностью.

8. stable_sort vs msd_radix (random data)

- msd_radix ≃10.7 мс, stable_sort ≃3.3 мс при n=3000.
- Radix без switch плохо работает на небольших n, а у stable_sort нет «комплексного» распределения.

9. stable_sort vs msd_radix_qs (random data)

- msd_radix_qs ≃2.3 мс vs stable_sort ≃3.3 мс, то есть гибрид снова выигрывает.

10. quick3way vs merge_lcp (random data)

- quick3way≃1.4 мс vs merge_lcp≃4.7 мс.
- По сравнениям: quick3way делает ~36 000, а merge_lcp — ~75 000.

11. quick3way vs msd_radix (random data)

- ≃1.4 мс vs ≃10.7 мс.
- Quick3way лучше, когда средняя длина префикса невелика.

12. quick3way vs msd_radix_qs (random data)

- ≃1.4 мс vs ≃2.3 мс.
- Quick3way чуть быстрее гибрида на случайных данных, потому что мелкие QuickSwitch-ветви составляют лишь малую часть работы.

13. merge_lcp vs msd_radix (random data)

- ≃4.7 мс vs ≃10.7 мс.
- LCP-Merge выигрывает у чистого Radix за счёт меньших констант, но всё равно медленнее QuickSort-методов.

14. merge_lcp vs msd_radix_qs (random data)

- ≃4.7 мс vs ≃2.3 мс.
- Гибрид MSD+QS обходится быстрее.

15. msd_radix vs msd_radix_qs (random data)

- ≃10.7 мс vs ≃2.3 мс.
- Очевидно, что дополнительный QuickSwitch даёт огромный выигрыш.

Общие выводы

1. По времени:
    - Самые быстрые на случайных данных — тернарный QuickSort (quick3way) и гибрид MSD-Radix+QuickSwitch (msd_radix_qs), они же минимизируют число сравнений.
    - std::sort/stable_sort занимают среднее положение, показывая классическое Θ(n log n).
    - Чистый MSD-Radix (msd_radix) и Merge+LCP (merge_lcp) хуже из-за высоких накладных расходов на распределение и LCP.
2. По числу сравнений:
    - msd_* алгоритмы (без switch) не делают сравнений.
    - quick3way ≪ std_sort, stable_sort < merge_lcp по сравнениюм.
    - Гибрид msd_radix_qs сравнивается с std_sort, но обходит по времени благодаря отсутствию сравнений в основном.
3. Типы данных:
    - На «reverse» строках quick3way и std_sort ведут себя хуже (много сравнений), тогда как radix-методы остаются стабильными.
    - «Almost» sorted даёт лучшие показатели всем алгоритмам, особенно тем, кто умеет раннее завершение сравнения (LCP-Merge выигрывает больше).
4. Соответствие теории:
    - Θ(n log n) для Quick/Merge подтверждается почти линейно-логарифмическими кривыми сравнений.
    - Radix-методы имеют O(n·W) поведение (равномерный рост времени), но константы при W=200 и R=256 слишком велики для малых n.

Этап 3. Эмпирический анализ адаптированных алгоритмов.
Уже не успел написать(