A1. Анализ строковых сортировок.

Работу выполнил Девятов Денис Сергеевич БПИ-238.

Этап 1. Подготовка тестовых данных.

Для экспериментов используется фиксированный алфавит из 74 символов:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#%:;^&*()-.

Он содержит 26 заглавных и 26 строчных латинских букв, 10 арабских цифр и 12 специальных знаков.

Для воспроизводимого получения тестовых массивов реализован класс-обёртка над std::mt19937. При создании каждой строки случайная длина генерируется из равномерного распределения U(10, 200), то есть каждая строка содержит от 10 до 200 символов включительно. По умолчанию в конструктор передаётся seed - текущее значение высокоточного таймера (steady_clock::now().time_since_epoch().count()). Для повторения эксперимента seed можно задать вручную, передав его в явном виде.

Типы формируемых массивов:

1. random - полностью случайная выборка из 3000 строк. Подмассивы меньшего размера формируются простым срезом первых n элементов.
2. reverse - тот же базовый набор после сортировки по возрастанию и последующего обращения порядка; обеспечивает наихудший лексикографический порядок.
3. almost-sorted - копия отсортированного множества, в которой случайно переставлены 2 % попарных элементов и имитирует почти упорядоченный ввод.
4. prefixed - вспомогательный режим, формирующий строки с длинным общим префиксом, что позволяет протестировать чувствительность алгоритмов к большим LCP.

Все четыре набора создаются заранее для максимального размера 3000, после чего при необходимости извлекаются префиксы требуемой длины 100, 200, …, 3000.

Таблица демонстрирует первые элементы трёх массивов, сгенерированных при seed = 42. Видно, что каждая строка:

1. состоит исключительно из символов разрешённого алфавита;
2. имеет длину в заявленном диапазоне;
3. корректно упорядочена в соответствии с выбранным типом массива.

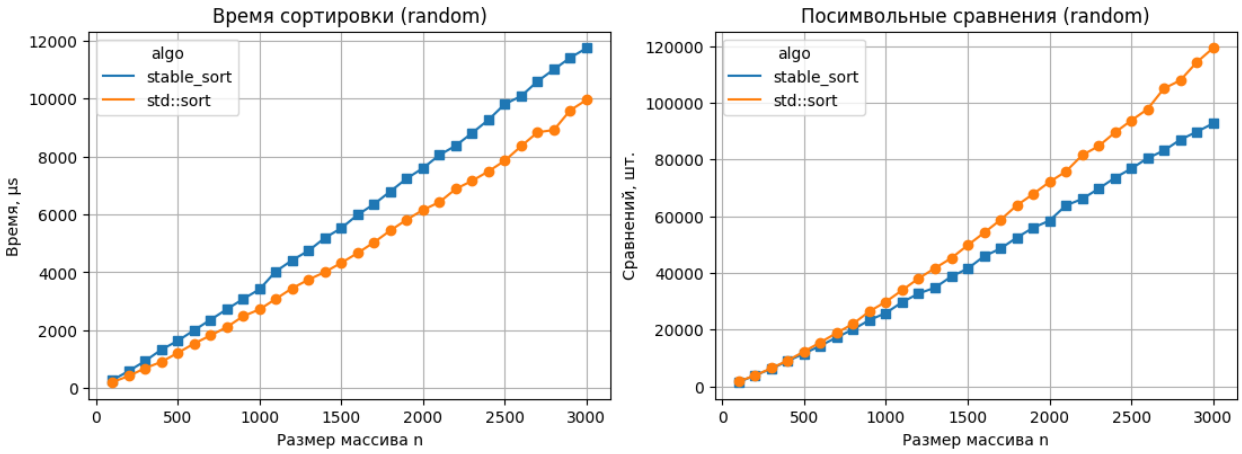
Тип массива	n	Первый элемент	Длина
random	100	b&KDqK8u8!TqT-y4eeBVcMyL6O-%j@5:1)@dB%^gmexjxDixhVFFA5GUpckyWq#ZX5d7SNzPeT:b4SS2cW-&d2CI18;7yULq1C6Jc5X2dNLuLm5zAqQdfVPFguR@UKDj4vG%C)HJh2pwS)ON-Zn!!F)vjGvVRKe7npl	100
reverse	300	zxrwxYBD)tMhiQmUkMB3E)2joGS.2CVt%xT:)B-NFAD..vdmMFcD)Cxu#RsiCO2F1O9BZ;Wprj8:-QXz^JCbb(KpQksdX^&p%pi(BZs0vDK@7@z.&!b:GKfT3bj@gRCejFW2;)zLb2@ap@GP-KV6^;	98
almost-sorted	2000	!#*&-RCSg&%d7lk@T9urB4MflYYc%(m#-IPp-3bdKV8N-6@-*dm5&M8^31qNQTBV8WsFjfmU!EWNY&L)lw)k*kB-lm9!je(1Dr%G.op^X!	105

Эти выборки подтверждают, что генератор удовлетворяет требованиям задачи, и поэтому им можно использоваться для дальнейшего замера производительности алгоритмов.

Этап 2. Эмпирический анализ стандартных алгоритмов сортировки.

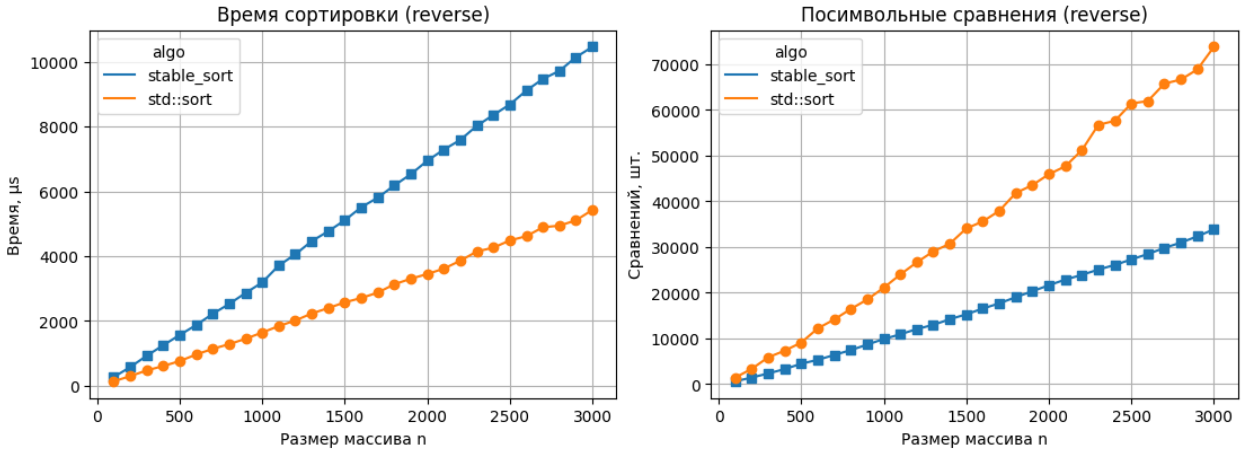
Средние значения получены по 10 независимым прогонам для каждого размера массива n.

1. random
На случайных входных данных алгоритм std::sort стабильно опережает по скорости std::stable_sort. Однако это преимущество сопровождается увеличением числа посимвольных сравнений примерно на 20–30%, особенно заметным при росте размера массива.

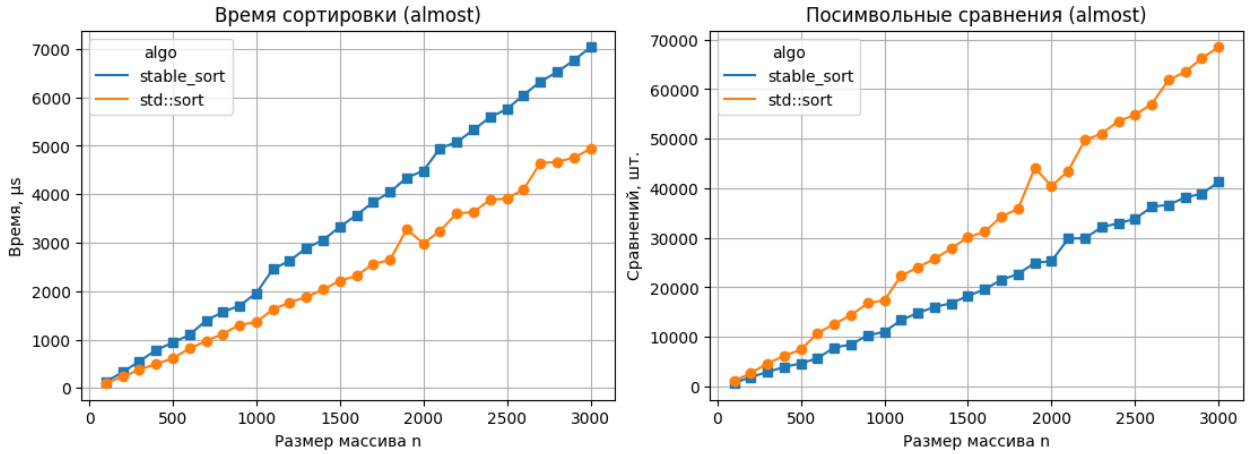


2. reverse
На данных, отсортированных в обратном порядке, std::sort продолжает демонстрировать существенное превосходство в скорости, хотя цена этого выигрыша возрастает - число посимвольных сравнений значительно больше,

чем у std::stable_sort, который стабильно выполняет меньше сравнений благодаря особенностям merge-сортировки.



3. almost
При небольшом количестве случайных перестановок std::sort также быстрее, однако на таких данных появляются заметные всплески времени и числа сравнений. Merge-сортировка (stable_sort) ведёт себя более стабильно, сохраняя меньшую величину посимвольных сравнений, но уступая по общей скорости выполнения.

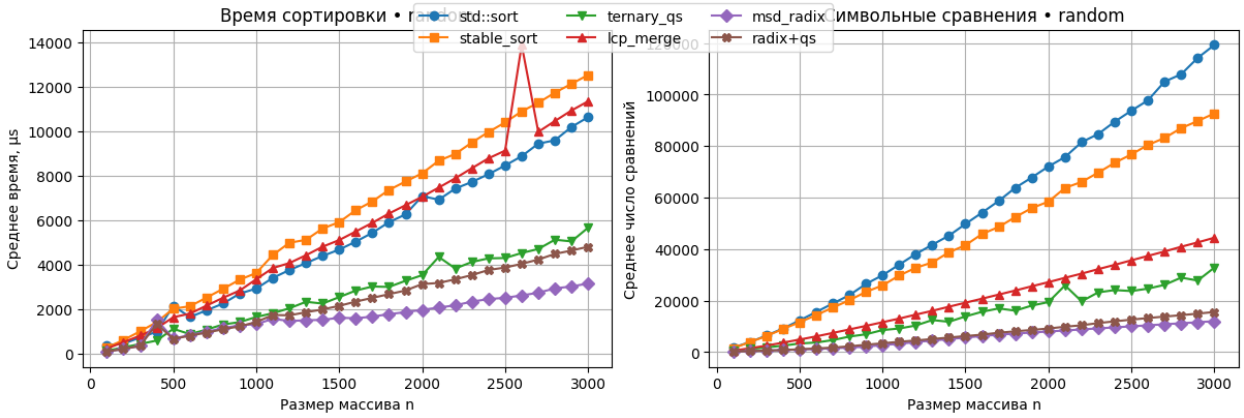


Этап 3. Эмпирический анализ адаптированных алгоритмов сортировки.

1. random

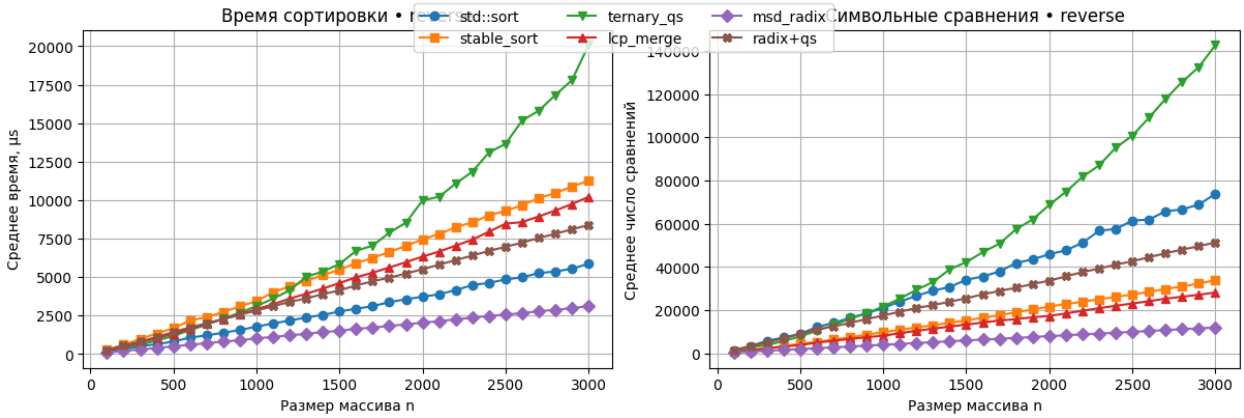
На случайных массивах лучший результат показали алгоритмы MSD-Radix и Radix+QS. Их производительность подтверждает теоретическую оценку O(n + Σ|s|), поскольку число посимвольных сравнений растёт практически линейно, и время выполнения оказывается минимальным среди всех алгоритмов.
Тернарный QuickSort работает быстрее, чем стандартный std::sort, примерно в 2–3 раза, и значительно экономит количество сравнений за счёт быстрого отсечения общих префиксов.
Алгоритм LCP-MergeSort по времени работы находится между стандартными сортировками, показывая значительное снижение числа сравнений (почти в 2

раза меньше, чем у stable_sort). Однако общая сложность остаётся порядка O(n log n).



2. reverse

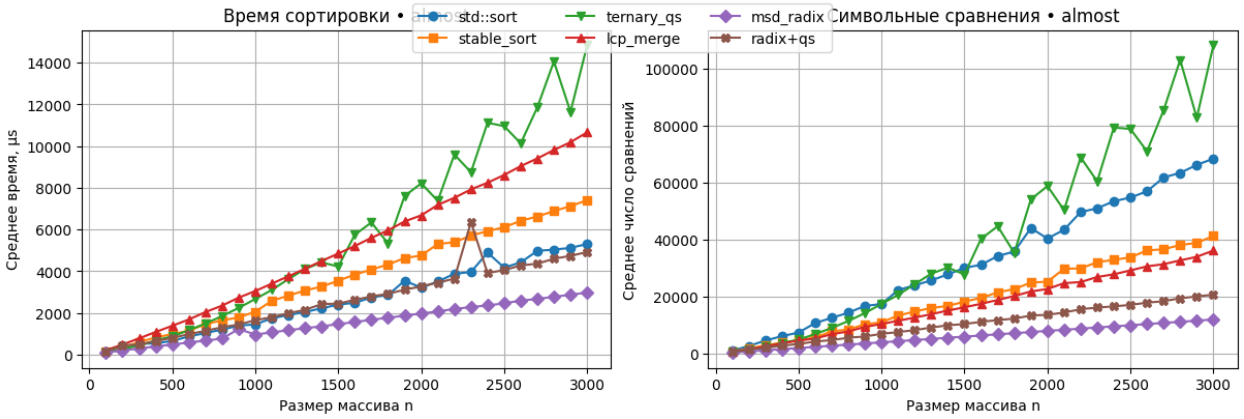
Данный тип входных данных является худшим для обычного QuickSort. Это проявилось в результатах: тернарный QuickSort сильно проигрывает всем другим алгоритмам при росте размера массива более 1000 элементов. Это объясняется тем, что выбор опорного элемента по первому элементу приводит к крайне неэффективному разбиению массива при обратно отсортированных данных.
В то же время алгоритмы семейства MSD-Radix практически не зависят от предварительного порядка элементов, и время их работы увеличивается незначительно. Radix+QS показывает аналогичные результаты, что говорит о правильном выборе порога для переключения на QuickSort.
Алгоритм LCP-MergeSort стабилен, показывает умеренный рост времени и числа сравнений, что соответствует ожиданиям.



3. almost

На данных с малым количеством случайных перестановок алгоритм LCP-MergeSort демонстрирует высокую эффективность по числу посимвольных сравнений, особенно при росте размера массива, за счёт эффективного использования длин совпадающих префиксов. По времени он постепенно приближается к MSD-Radix, хотя последняя сохраняет лидерство.
Тернарный QuickSort на почти отсортированных данных демонстрирует нестабильное поведение - заметные всплески времени выполнения связаны с

неудачным выбором опорного элемента, что периодически ухудшает разбиение массива. Тем не менее, даже при таких пиках, он остаётся конкурентоспособным по сравнению со стандартными алгоритмами.
MSD-Radix и Radix+QS вновь показывают стабильную производительность, подтверждая близость своей асимптотики к линейной (O(n + Σ|s|)).



Полученные данные хорошо согласуются с теоретическими ожиданиями: алгоритмы MSD-Radix и Radix+QS демонстрируют линейный рост производительности, что полностью соответствует оценке сложности O(n + Σ|s|). В то же время, стандартные алгоритмы сортировки явно демонстрируют логарифмический множитель в сложности, что выражается в значительно большем количестве сравнений и времени работы.
Особенно наглядно это заметно на больших размерах данных, где выгода от специализированных подходов становится всё более выраженной.

Замечание.

В ходе реализации тернарного QuickSort опорным элементом всегда выбирался первый элемент подмассива, что соответствует исходному принципу выбора pivot в реализации стандартного алгоритма сортировки std::sort в STL на начальной стадии.

ID посылок:

A1m - 320913551

A1q - 320947750

A1r - 321023779

A1rq - 321024597

Ссылка на репозиторий - https://github.com/BelyLandy/AISD_SET_9

StringGenerator.hpp

```
#include <vector>
#include <string>
#include <random>
#include <algorithm>
#include <chrono>
#include <cstdint>

class StringGenerator {
public:
  explicit StringGenerator(std::uint32_t seed =
    static_cast<std::uint32_t>(
      std::chrono::steady_clock::now().time_since_epoch().count()))
    : _rng(seed),
    _lenDist(10, 200),
    _charDist(0, AlphabetSize - 1)
  {
    buildBaseVectors();
  }

  std::vector<std::string> randomArray(int size) const {
    return slice(_randomBase, size);
  }

  std::vector<std::string> reverseSortedArray(int size) const {
    return slice(_reverseBase, size);
  }

  std::vector<std::string> almostSortedArray(int size, std::size_t swapPairs
= 0) const
  {
    std::vector<std::string> result = slice(_sortedBase, size);
    if (swapPairs == 0) {
      swapPairs = static_cast<std::size_t>(result.size() * 0.02);
    }

    std::uniform_int_distribution<int> idxDist(0,
static_cast<int>(result.size() - 1));

    for (std::size_t k = 0; k < swapPairs; ++k) {
      int i = idxDist(_rng), j = idxDist(_rng);

      if (i != j) {
        std::swap(result[i], result[j]);
      }
    }

    return result;
  }

  std::vector<std::string> randomArrayWithPrefix(int size, int prefixLen)
const
  {
    std::vector<std::string> out;

    out.reserve(size);
    std::string common = randomString(prefixLen);

    for (int i = 0; i < size; ++i) {
      std::size_t len = std::max<std::size_t>(prefixLen + 1, _lenDist(_rng));
      std::string tail = randomString(len - prefixLen);
```

```
      out.push_back(common + tail);
    }

    return out;
  }

private:
  static constexpr char kAlphabet[] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz"
    "0123456789"
    "!@#%:;^&*()-.";

  static constexpr int AlphabetSize = static_cast<int>(sizeof(kAlphabet) -
1);

  mutable std::mt19937 _rng;
  mutable std::uniform_int_distribution<std::size_t> _lenDist;
  mutable std::uniform_int_distribution<int>  _charDist;

  std::vector<std::string> _randomBase;
  std::vector<std::string> _sortedBase;
  std::vector<std::string> _reverseBase;

  std::string randomString(std::size_t len) const {
    std::string s;

    s.reserve(len);

    for (std::size_t i = 0; i < len; ++i) {
      s.push_back(kAlphabet[_charDist(_rng)]);
    }

    return s;
  }

  static std::vector<std::string> slice(const std::vector<std::string>& src,
int size) {
    return { src.begin(), src.begin() + size };
  }

  void buildBaseVectors() {
    const int N = 3000;

    _randomBase.reserve(N);

    for (int i = 0; i < N; ++i) {
      _randomBase.push_back(randomString(_lenDist(_rng)));
    }

    _sortedBase = _randomBase;
    std::sort(_sortedBase.begin(), _sortedBase.end());

    _reverseBase = _sortedBase;
    std::reverse(_reverseBase.begin(), _reverseBase.end());
  }
};
```

StringSortTester.hpp

```
#include "StringSpecialSorts.hpp"

#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
#include <cstdint>
#include <numeric>

class StringSortTester {
public:
  enum class Algo {
    StdQuick,    // std::sort.
    StdMerge,    // std::stable_sort.
    TernaryQuick,  // трёхсторонний QS.
    LcpMerge,    // merge-sort с LCP.
    MsdRadix,    // MSD-radix.
    MsdRadixQuick  // MSD + QS.
  };

  struct Metrics {
    double avgMicroSeconds;
    std::size_t avgCharCompares;
  };

  Metrics measure(const std::vector<std::string>& input, Algo algo, int
repeats = 10) const
  {
    std::vector<double> times;
    std::vector<std::size_t> comps;

    times.reserve(repeats);
    comps.reserve(repeats);

    for (int r = 0; r < repeats; ++r) {
      std::vector<std::string> data = input;
      std::size_t counter = 0;

      auto t0 = Clock::now();
      switch (algo)
      {
      case Algo::StdQuick:
        std::sort(data.begin(), data.end(),
          [&counter](const std::string& a,
            const std::string& b) {
              return lexCmp(a, b, counter);
          });

        break;

      case Algo::StdMerge:
        std::stable_sort(data.begin(), data.end(),
          [&counter](const std::string& a,
            const std::string& b) {
              return lexCmp(a, b, counter);
          });

        break;

      case Algo::TernaryQuick:
        sps::quickSort3(data, 0, static_cast<int>(data.size()) - 1, 0,
&counter);
```

```
        break;

      case Algo::LcpMerge: {
        std::vector<std::string> buf(data.size());
        sps::mergeSort(data, buf, 0, static_cast<int>(data.size()),
&counter);

        break;
      }

      case Algo::MsdRadix: {
        std::vector<std::string> aux(data.size());
        sps::msdRadix(data, aux, 0, static_cast<int>(data.size()) - 1, 0,
&counter);

        break;
      }

      case Algo::MsdRadixQuick: {
        std::vector<std::string> aux(data.size());
        sps::msdRadixQS(data, aux, 0, static_cast<int>(data.size()) - 1, 0,
&counter);

        break;
      }
      }

      auto t1 = Clock::now();

      double us = std::chrono::duration<double, std::micro>(t1 - t0).count();

      times.push_back(us);
      comps.push_back(counter);
    }

    return { mean(times), mean(comps) };
  }

private:
  using Clock = std::chrono::high_resolution_clock;

  static bool lexCmp(const std::string& a, const std::string& b, std::size_t&
counter)
  {
    std::size_t len = std::min(a.size(), b.size());

    for (std::size_t i = 0; i < len; ++i) {
      ++counter;
      if (a[i] < b[i]) {
        return true;
      }

      if (a[i] > b[i]) {
        return false;
      }
    }

    return a.size() < b.size();
  }

  static double mean(const std::vector<double>& v) {
    return v.empty() ? 0.0 : std::accumulate(v.begin(), v.end(), 0.0) /
v.size();
```

```
  }
  static std::size_t mean(const std::vector<std::size_t>& v) {
    if (v.empty()) {
      return 0;
    }

    std::size_t sum = std::accumulate(v.begin(), v.end(), std::size_t{ 0 });

    return (sum + v.size() / 2) / v.size();
  }
};
```

Вспомогательный StringSpecialSorts.hpp

```
#include <vector>
#include <string>
#include <algorithm>
#include <cstddef>

namespace sps {

  inline int charAt(const std::string& s, int d, std::size_t* cmp = nullptr)
  {
    if (cmp) ++(*cmp);
    return d < static_cast<int>(s.size())
      ? static_cast<unsigned char>(s[d])
      : -1;
  }

  inline void insertionSuffix(std::vector<std::string>& a,
    int lo, int hi, int d, std::size_t* cmp)
  {
    for (int i = lo + 1; i <= hi; ++i) {
      std::string key = std::move(a[i]);
      int j = i - 1;
      while (j >= lo && a[j].compare(d, std::string::npos,
        key, d, std::string::npos) > 0)
      {
        a[j + 1] = std::move(a[j]);
        --j;
      }
      a[j + 1] = std::move(key);
    }
  }

  // ======================================================
  // 1. String QuickSort
  // ======================================================
  inline void quickSort3(std::vector<std::string>& a,
    int lo, int hi, int d,
    std::size_t* cmp = nullptr)
  {
    if (hi <= lo) return;
    if (hi - lo < 10) { insertionSuffix(a, lo, hi, d, cmp); return; }

    int lt = lo, gt = hi;
    int v = charAt(a[lo], d, cmp);
    int i = lo + 1;
    while (i <= gt) {
      int t = charAt(a[i], d, cmp);
      if (t < v)     std::swap(a[lt++], a[i++]);
```

```
      else if (t > v)   std::swap(a[i], a[gt--]);
      else        ++i;
    }
    quickSort3(a, lo, lt - 1, d, cmp);
    if (v >= 0) quickSort3(a, lt, gt, d + 1, cmp);
    quickSort3(a, gt + 1, hi, d, cmp);
  }

  // ========================================================
  // 2. String MergeSort
  // ========================================================
  inline int lcpCompare(const std::string& a, const std::string& b,
    std::size_t* cmp)
  {
    std::size_t len = std::min(a.size(), b.size());
    std::size_t i = 0;
    while (i < len && a[i] == b[i]) { if (cmp) ++(*cmp); ++i; }
    if (i == len) return (a.size() < b.size()) ? -1 : (a.size() > b.size());
    if (cmp) ++(*cmp);
    return (a[i] < b[i]) ? -1 : 1;
  }
  inline bool lcpLess(const std::string& a, const std::string& b,
    std::size_t* cmp) {
    return lcpCompare(a, b, cmp) < 0;
  }

  inline void mergeSort(std::vector<std::string>& v,
std::vector<std::string>& buf,
    int L, int R, std::size_t* cmp = nullptr)
  {
    if (R - L <= 1) return;
    int M = (L + R) >> 1;
    mergeSort(v, buf, L, M, cmp);
    mergeSort(v, buf, M, R, cmp);
    int i = L, j = M, k = L;
    while (i < M && j < R)
      buf[k++] = lcpLess(v[i], v[j], cmp) ? std::move(v[i++]) :
std::move(v[j++]);
    while (i < M) buf[k++] = std::move(v[i++]);
    while (j < R) buf[k++] = std::move(v[j++]);
    for (int t = L; t < R; ++t) v[t] = std::move(buf[t]);
  }

  // ========================================================
  // 3. MSD Radix Sort
  // ========================================================
  inline void msdRadix(std::vector<std::string>& a,
    std::vector<std::string>& aux,
    int lo, int hi, int d,
    std::size_t* cmp = nullptr)
  {
    constexpr int R = 256;
    if (hi <= lo) return;
    if (hi - lo < 15) { insertionSuffix(a, lo, hi, d, cmp); return; }

    int freq[R + 2] = { 0 };
    for (int i = lo; i <= hi; ++i) ++freq[charAt(a[i], d, cmp) + 2];
    for (int r = 0; r < R + 1; ++r) freq[r + 1] += freq[r];
    for (int i = lo; i <= hi; ++i)
      aux[lo + freq[charAt(a[i], d, cmp) + 1]++] = std::move(a[i]);
    for (int i = lo; i <= hi; ++i) a[i] = std::move(aux[i]);

    int start = 0;
    for (int r = 0; r < R; ++r) {
```

```
      int cntR = freq[r + 1] - freq[r];
      if (cntR > 1) msdRadix(a, aux, lo + start, lo + start + cntR - 1, d +
1, cmp);
      start += cntR;
    }
  }

  // ====================================================
  // 4. MSD Radix + QuickSort
  // ====================================================
  inline void msdRadixQS(std::vector<std::string>& a,
    std::vector<std::string>& aux,
    int lo, int hi, int d,
    std::size_t* cmp = nullptr)
  {
    constexpr int R = 256;
    constexpr int SWITCH_THRESHOLD = 74;

    if (hi <= lo) return;

    if (hi - lo + 1 < SWITCH_THRESHOLD) {
      quickSort3(a, lo, hi, d, cmp);
      return;
    }

    int freq[R + 2] = { 0 };
    for (int i = lo; i <= hi; ++i) ++freq[charAt(a[i], d, cmp) + 2];
    int pos[R + 2]; pos[0] = 0;
    for (int r = 1; r < R + 2; ++r) pos[r] = pos[r - 1] + freq[r - 1];
    for (int i = lo; i <= hi; ++i) {
      int idx = charAt(a[i], d, cmp) + 2;
      aux[lo + pos[idx]] = std::move(a[i]);
      ++pos[idx];
    }
    for (int i = lo; i <= hi; ++i) a[i] = std::move(aux[i]);

    int start = 0;
    for (int r = 1; r < R + 2; ++r) {
      int cnt = freq[r];
      if (cnt > 1 && r > 1)
        msdRadixQS(a, aux, lo + start, lo + start + cnt - 1, d + 1, cmp);
      start += cnt;
    }
  }

}
```