Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Дискретный анализ»

Студент: К. В. Кознева
Преподаватель: С. А. Михайлова
Группа: М8О-201БВ-24

Дата:
Оценка:
Подпись:

Москва, 2026

Лабораторная работа №1

Задача: Требуется разработать программу, осуществляющую ввод пар «ключ-значение», их упорядочивание по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод отсортированной последовательности.

Вариант сортировки: Поразрядная сортировка.

Вариант ключа: автомобильные номера в формате A 999 BC (используются буквы латинского алфавита).

Вариант значения: Строки фиксированной длины 64 символа, во входных данных могут встретиться строки меньшей длины, при этом строка дополняется до 64-х нулевыми символами, которые не выводятся на экран.

1 Описание

Требуется реализовать поразрядную сортировку для последовательности пар «ключ-значение». Ключ представляет собой автомобильный номер в формате `A 999 BC` (буквы латинского алфавита, цифры и пробелы на фиксированных позициях).

Ключ имеет фиксированную длину 8 символов, при этом позиции 1 и 5 (нумерация с 0) всегда содержат пробелы и не влияют на сортировку. Остальные 6 позиций (буквы и цифры) участвуют в упорядочивании.

Поразрядная сортировка (radix sort) - это алгоритм, который использовался в машинах, предназначенных для сортировки перфокарт. В алгоритме поразрядной сортировки сначала производится сортировка по младшей цифре, после чего перфокарты снова объединяются в одну колоду, в которой сначала идут перфокарты из нулевого приемника, затем - из первого приемника, затем - из второго и т.д. После этого вся колода снова сортируется по предпоследней цифре, и перфокарты вновь собираются в одну стопку тем же образом. Процесс прдолжается до тех пор, пока перфокарты не окажутся отсортированными по всем d цифрам. После этого перфокарты оказываются полностью отсортированы в порядке возрастания d-значных чисел. [1].

В данной работе алгоритм реализован в три этапа для каждого разряда:

1. Подсчёт частот символов на текущей позиции ключа в массиве счётчиков размера 256.
2. Преобразование массива частот в массив префиксных сумм для определения итоговых позиций.
3. Размещение элементов во временном массиве индексов при проходе справа налево, что обеспечивает стабильность сортировки.

Проход справа налево нужен для сохранения исходного порядка элементов с одинаковыми ключами (стабильность сортировки).

Асимптотика реализации:

- время: $O(m \cdot (n + k))$, где n - число пар, $k = 256$, $m = 6$ (количество значимых разрядов);
- дополнительная память: $O(n)$ для хранения индексов (без учёта хранения исходных данных).

2 Исходный код

Программа состоит из функции поразрядной сортировки `RadixSort` и функции `main`. Для хранения элементов используются псевдонимы типов:

- `TKey` - тип ключа (`std::string`), автомобильный номер в формате `A 999 BC` (8 символов);
- `TValue` - тип значения (`std::string`), строка переменной длины (до 64 символов);
- `TObject` - пара `std::pair<TKey, TValue>`.

Также задаются константы:

- `KEY_LENGTH = 8` - фиксированная длина ключа;
- `FIRST_SPACE_POS = 1` и `SECOND_SPACE_POS = 5` - позиции пробелов в ключе (нумерация с 0);
- `ASCII_SIZE = 256` - мощность алфавита для поразрядной сортировки;
- `VALUE_LENGTH = 64` - максимальная длина значения (в данной реализации не используется для добивки).

Логика функции `RadixSort(std::vector<TObject>& objects)`:

1. Если входной массив пуст, функция сразу завершает работу.
2. Для оптимизации памяти создаются два массива индексов: `indices` и `tempIndices` размером `objects.size()`. Массив `indices` инициализируется значениями 0, 1, 2, ..., что соответствует исходному порядку элементов.
3. Выполняется поразрядная сортировка по 8 разрядам (символам) ключа, начиная с младшего (справа):
   (a) Если текущая позиция соответствует пробелу (позиции 1 или 5), она пропускается, так как пробелы не влияют на порядок сортировки.
   (b) Для остальных позиций создаётся массив счётчиков `std::array<int, ASCII_SIZE> count{}`.
   (c) Выполняется подсчёт частот символов на текущей позиции: `++count[static_cast<unsig char>(objects[indices[i]].first[digitPos])]`.

(d) Строятся префиксные суммы: каждая ячейка `count[i]` хранит количество символов с кодом не больше `i`.

(e) Выполняется проход по `indices` с конца к началу для сохранения стабильности сортировки. Для каждого элемента вычисляется итоговая позиция `position = count[digit] - 1`, после чего индекс элемента помещается в `tempIndices[position]`.

(f) Массивы `indices` и `tempIndices` меняются местами через `swap`, чтобы использовать обновлённый порядок на следующем разряде.

4. После завершения всех проходов создаётся результирующий массив `result` с зарезервированной памятью.

5. Выполняется проход по отсортированному массиву индексов `indices` и перемещение объектов в правильном порядке: `result.emplace_back(std::move(objects[indices[i`. Использование `std::move` позволяет избежать копирования строк.

6. Исходный массив `objects` и результирующий массив `result` меняются местами через `swap`.

Логика функции `main`:

1. Создаётся вектор `objects` для хранения пар ключ-значение.

2. В цикле считываются пары `key` и `value` из стандартного потока: `std::getline(std::cin, key, '\t')` читает ключ до символа табуляции, `std::getline(std::cin, value)` читает значение до конца строки.

3. Если значение содержит символ возврата каретки `\r` (Windows-формат), он удаляется с помощью `value.pop_back()`.

4. Считанная пара добавляется в вектор через `objects.emplace_back(std::move(key), std::move(value))` для минимизации копирований.

5. После завершения ввода вызывается функция сортировки `RadixSort(objects)`.

6. Отсортированные пары выводятся в формате `key\tvalue` (ключ, символ табуляции, значение, перевод строки).

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <string>
#include <vector>

const int KEY_LENGTH = 8;
const int FIRST_SPACE_POS = 1;
const int SECOND_SPACE_POS = 5;
const int ASCII_SIZE = 256;
const int VALUE_LENGTH = 64;

using TKey = std::string;
using TValue = std::string;
using TObject = std::pair<TKey, TValue>;

void RadixSort(std::vector<TObject>& objects) {
    if (objects.empty()) {
        return;
    }

    const size_t size = objects.size();
    std::vector<size_t> indices(size);
    std::vector<size_t> tempIndices(size);
    std::array<int, ASCII_SIZE> count{};

    for (size_t i = 0; i < size; ++i) {
        indices[i] = i;
    }

    for (int digitPos = KEY_LENGTH - 1; digitPos >= 0; --digitPos) {
        if (digitPos == FIRST_SPACE_POS || digitPos == SECOND_SPACE_POS) {
            continue;
        }

        count.fill(0);

        for (size_t i = 0; i < size; ++i) {
            ++count[static_cast<unsigned char>(objects[indices[i]].first[digitPos])];
        }

        for (int i = 1; i < ASCII_SIZE; ++i) {
            count[i] += count[i - 1];
        }

        for (int i = static_cast<int>(size) - 1; i >= 0; --i) {
            unsigned char digit = static_cast<unsigned char>(
                objects[indices[i]].first[digitPos]);
            tempIndices[--count[digit]] = indices[i];
```

```
        }

        indices.swap(tempIndices);
    }

    std::vector<TObject> result;
    result.reserve(size);
    for (size_t i = 0; i < size; ++i) {
        result.emplace_back(std::move(objects[indices[i]]));
    }
    objects.swap(result);
}

int main() {
    std::vector<TObject> objects;

    TKey key;
    TValue value;

    while (std::getline(std::cin, key, '\t') && std::getline(std::cin, value)) {
        if (!value.empty() && value.back() == '\r') {
            value.pop_back();
        }

        objects.emplace_back(std::move(key), std::move(value));
    }

    RadixSort(objects);

    for (const TObject& obj : objects) {
        std::cout << obj.first << '\t' << obj.second << '\n';
    }

    return 0;
}
```

3 Консоль

```
PS C:\ Users\ Administrator\ Desktop\ дискретный_анализ\ lab1>g++ main.cpp
-o main.exe
PS C:\ Users\ Administrator\ Desktop\ дискретный_анализ\ lab1>cat input.txt
A 000 AA        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt
Z 999 ZZ        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat
A 000 AA        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa
Z 999 ZZ        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na
PS C:\ Users\ Administrator\ Desktop\ дискретный_анализ\ lab1>Get-Content input.txt
| .\ main.exe
A 000 AA        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt
A 000 AA        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa
Z 999 ZZ        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat
Z 999 ZZ        n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na
```

4 Тест производительности

Для оценки эффективности реализованной поразрядной сортировки было проведено сравнение со стандартной сортировкой `std::sort` из библиотеки C++.

Методика тестирования

- **Генерация данных:** случайные ключи в формате автомобильных номеров `A 999 BC` (буквы латинского алфавита, цифры) и случайные значения длиной до 64 символов.
- **Размеры выборок:** от 100 000 до 1 000 000 элементов с шагом 100 000.
- **Измерения:** для каждого размера проводился однократный запуск. Измерялось только время сортировки (без учёта ввода-вывода и генерации данных).

Результаты тестирования

Был написан специальный бенчмарк, который генерирует тестовые данные и замеряет время работы обеих сортировок.

```
     PS C:\Users\Administrator\Desktop\report_pattern>g++ benchmark.cpp
-o benchmark.exe
     PS C:\Users\Administrator\Desktop\report_pattern>.\benchmark.exe
     Running benchmark...

     n = 100000   RadixSort: 197.713 ms   std::sort: 238.381 ms
speedup: 1.21x
     n = 200000   RadixSort: 445.540 ms   std::sort: 494.019 ms
speedup: 1.11x
     n = 300000   RadixSort: 485.016 ms   std::sort: 761.226 ms
speedup: 1.57x
     n = 400000   RadixSort: 567.417 ms   std::sort: 769.369 ms
speedup: 1.36x
     n = 500000   RadixSort: 735.969 ms   std::sort: 990.981 ms
speedup: 1.35x
     n = 600000   RadixSort: 944.123 ms   std::sort: 1188.150 ms
speedup: 1.26x
     n = 700000   RadixSort: 1051.190 ms  std::sort: 1391.540 ms
speedup: 1.32x
     n = 800000   RadixSort: 1210.400 ms  std::sort: 1623.620 ms
speedup: 1.34x
     n = 900000   RadixSort: 1352.810 ms  std::sort: 1821.060 ms
speedup: 1.35x
```

```
      n = 1000000  RadixSort: 1527.730 ms  std::sort: 2044.920 ms
speedup: 1.34x
```

Графическое представление

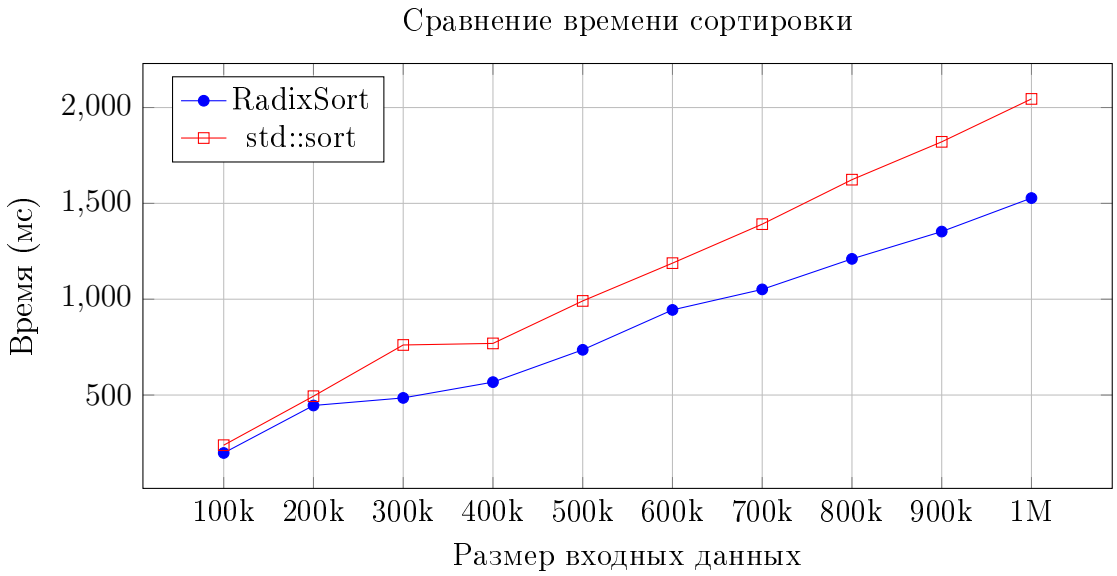

Рис. 1: Зависимость времени сортировки от размера входных данных

Анализ результатов

- **Линейная сложность:** Время работы поразрядной сортировки демонстрирует линейный рост с увеличением количества элементов. При увеличении размера данных в 10 раз (от 100 000 до 1 000 000) время выполнения возрастает с 197.7 мс до 1527.7 мс, что соответствует росту примерно в 7.7 раза — это близко к теоретической линейной зависимости с учётом накладных расходов.
- **Сравнение с std::sort:** На всех размерах выборки поразрядная сортировка показывает лучшее время работы, чем стандартная сортировка. Ускорение варьируется от 1.11x до 1.57x, причём наибольший выигрыш (1.57x) достигается на размере 300 000 элементов. Среднее ускорение по всем размерам составляет примерно 1.3x.
- **Эффективность поразрядной сортировки:** Алгоритм эффективно использует фиксированную длину ключа (8 символов) и малый алфавит (256 символов), что позволяет обходиться без дорогостоящих операций сравнения строк. Вместо сравнений используются простые операции индексации в массиве счётчиков.

5 Выводы

В этой лабораторной работе я реализовала стабильную поразрядную сортировку для пар «ключ-значение» с ключами в формате автомобильных номеров и на практике разобрала полный цикл алгоритма: обработка ключа поразрядно, подсчёт частот символов на каждом разряде, построение префиксных сумм и распределение элементов по итоговым позициям.

Главный технический вывод: оценка $O(m \cdot (n+k))$ достигается благодаря фиксированной длине ключа и малому алфавиту. В моём варианте ключ состоит из 8 символов, два из которых — пробелы на известных позициях, поэтому фактически обрабатывается только $m = 6$ разрядов, а мощность алфавита равна $k = 256$. Это позволяет алгоритму работать предсказуемо по времени даже на больших входных данных. При этом дополнительная память составляет $O(n)$ за счёт сортировки индексов, а не самих объектов.

Также стало понятно, как обеспечивается стабильность сортировки: при распределении элементов по итоговым позициям нужно обходить массив справа налево, иначе относительный порядок элементов с одинаковыми ключами нарушается. Кроме того, я научилась корректно обрабатывать символы через static_cast<unsigned char>, чтобы избежать проблем со знаковым типом char при индексации массива счётчиков.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))