Лабораторная работа № 1 по курсу дискретного анализа: сортировка за линейное время

Выполнил студент группы М8О-203Б-23 МАИ *Арусланов Кирилл*.

Условие

Общая постановка задачи

Требуется разработать программу, которая осуществляет ввод пар *ключ-значение*, упорядочивает их по возрастанию ключа с помощью алгоритма сортировки за линейное время и выводит отсортированную последовательность.

Вариант задания

- **Тип ключа**: почтовые индексы (целые числа).
- **Тип значения**: строки переменной длины (до 2048 символов).
- **Алгоритм сортировки**: сортировка подсчётом.

Пример ввода:

```
000000  xGfxrxGGxrxMMMMfrrrG
999999  xGfxrxGGxrxMMMMfrrr
000000  xGfxrxGGxrxMMMMfrr
999999  xGfxrxGGxrxMMMMfr
```

Пример вывода:

```
000000  xGfxrxGGxrxMMMMfrrrG
000000  xGfxrxGGxrxMMMMfrr
999999  xGfxrxGGxrxMMMMfrrr
999999  xGfxrxGGxrxMMMMfr
```

Метод решения

Для решения задачи применяется алгоритм сортировки подсчётом, который эффективен для целочисленных ключей в ограниченном диапазоне (в данном случае 000000–999999). Алгоритм включает следующие шаги:

1. Инициализация вектора счётчиков размером `MAX_KEY = 1000000`.
2. Подсчёт количества вхождений каждого ключа.
3. Накопление счётчиков для определения позиций элементов.
4. Формирование отсортированного массива.

Сложность алгоритма составляет $O(n+k)$, где n — число элементов, а $k=1000000$ — диапазон ключей. Поскольку k фиксировано, общая сложность является линейной относительно n.

Описание программы

Программа реализована в одном файле и включает следующие компоненты:

- **Структура `Item`:** Определяет пару *ключ-значение*:
  - `key` — тип `size_t` (почтовый индекс).
  - `value` — тип `std::string` (строка до 2048 символов).
- **Функция `countingSort`:** Реализует сортировку подсчётом. Использует вектор `cntVect` размером `MAX_KEY` для хранения счётчиков. Принимает вектор элементов `Item` и возвращает отсортированный вектор.
- **Функция `main`:** Считывает пары *ключ-значение* из стандартного ввода, вызывает `countingSort` и выводит результат с форматированием ключей (6 цифр с ведущими нулями).

Используемые библиотеки: `<iostream>`, `<vector>`, `<string>`, `<iomanip>`.

Дневник отладки

Во время разработки программа не проходила 13-й тест из-за превышения лимита времени. Изначально планировалось использовать функцию `maxKey` для динамического определения максимального ключа, что требовало дополнительного прохода по данным. Для оптимизации было решено задать фиксированное значение `MAX_KEY = 1000000`, соответствующее диапазону почтовых индексов. После этого изменения программа успешно прошла все тесты.

Тест производительности

Для сравнения производительности алгоритмов `countingSort` и `std::sort` были проведены тесты на различных объемах входных данных: $n=1000$, 10000, 50000, 100000, 500000, 1000000. Время выполнения и потребление памяти измерялись с использованием случайных данных: ключи в диапазоне 000000–999999, строки `value` случайной длины до 2048 символов. Результаты представлены в таблице:

n	Время `countingSort` (мс)	Время `std::sort` (мс)	Память `countingSort` (Мб)	Память `std::sort` (Мб)
1000	39	0	7.67	0.04

n	Время `countingSort` (мс)	Время `std::sort` (мс)	Память `countingSort` (Мб)	Память `std::sort` (Мб)
10000	14	3	8.01	0.38
50000	14	14	9.54	1.91
100000	26	33	11.44	3.81
500000	100	194	26.70	19.07
1000000	200	470	45.78	38.15

Анализ показывает, что на малых объемах данных ($n \leq 10000$) `std::sort` работает быстрее: например, для $n = 1000$ его время составляет 0 мс против 39 мс у `countingSort`. Это связано с большими накладными расходами `countingSort` на инициализацию вектора счётчиков размером `MAX_KEY = 1000000`. Однако на средних и больших объемах данных ($n \geq 100000$) `countingSort` становится быстрее благодаря линейной сложности $O(n+k)$: для $n = 1000000$ его время составляет 200 мс против 470 мс у `std::sort`, что в 2.35 раза быстрее. Потребление памяти у `countingSort` всегда выше из-за дополнительного вектора счётчиков и результирующего вектора: для $n = 1000000$ он использует 45.78 Мб против 38.15 Мб у `std::sort`. Таким образом, `countingSort` предпочтителен для больших данных с ограниченным диапазоном ключей, тогда как `std::sort` лучше подходит для малых данных или случаев с ограниченной памятью.

Недочёты

После оптимизации с использованием константы `MAX_KEY` недочётов не выявлено. Все тесты пройдены успешно.

Выводы

Сортировка подсчётом применима для задач с ограниченным диапазоном ключей, таких как сортировка почтовых индексов или идентификаторов. Алгоритм показал линейную сложность и высокую эффективность. Основная трудность заключалась в оптимизации времени работы, решённая отказом от динамического поиска максимального ключа. Программирование оказалось умеренно сложным из-за необходимости точной реализации алгоритма.