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: О. С. Концебалов
Преподаватель: А. А. Никитин
Группа: М8О-209Б-22
Дата: 25.05.2024
Оценка:
Подпись:

Москва, 2024

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.

Тип ключа: `числа от 0 до`
$$2^64 - 1.$$

Тип значения: `числа от 0 до`
$$2^64 - 1.$$

1 Описание

Поразрядная сортировка основана на разбиении элементов на группы по разрядам и последующей упорядоченности в соответствии с этими разрядами, без использования сравнений. По сути, для каждого разряда мы применяем сортировку подсчётом.Этот метод подходит для сортировки целых чисел или строк, когда известна длина разрядов. [?].

2 Исходный код

На каждой непустой строке входного файла располагается пара «ключ-значение», поэтому создадим новую структуру `key_value`, в которой будем хранить ключ и значение. Как я уже сказал выше, поразрядная сортировка представляет собой сортировку подсчётом для каждого разряда ключа, поэтому опишем функцию `countingSort` Создаем два массива: «counterArr», который используется для подсчета количества элементов с определенным значением байта, и «helperArr», который будет содержать отсортированные элементы. Проходимся по всем элементам вектора «data», и для каждого элемента вычисляем значение байта, на основе которого увеличиваем соответствующий счетчик в массиве «counterArr». Для каждого элемента массива «countersArr» вычисляем префиксную сумму, показывающую общее количество элементов, которые должны быть помещены в соответствующие позиции в отсортированном массиве. Используя массив «countersArr», элементы из вектора «data» перемещаем в правильные позиции в массиве «helperArr». Элементы переносятся с учетом их позиции в «countersArr», а затем счетчик уменьшается на единицу. Исходный вектор «data» заменяется вектором «countersArr». Функция `radixSort` реализует алгоритм поразрядной сортировки. На каждой итерации цикла вызывается функция `countingSort` по текущему байту.

```
#include <cstdint>
#include <iostream>
#include <vector>

using ull = unsigned long long;
using array_t = std::vector<std::pair<ull, ull>>;

const int MAX_VALUE_BYTE = 256;
const int BIT_MASK = 0xff;

void countingSort(array_t& data, const int byte) {
    size_t data_size = data.size();

    std::vector<int> counterArr(MAX_VALUE_BYTE, 0);
    array_t helperArr(data_size);

    for (const std::pair<ull, ull>& element: data) {
        ++counterArr[(element.first >> byte) & BIT_MASK];
    }

    for (size_t i = 1; i != MAX_VALUE_BYTE; ++i) {
        counterArr[i] += counterArr[i - 1];
    }

    for (int i = data_size - 1; i >= 0; --i) {
        helperArr[counterArr[(data[i].first >> byte) & BIT_MASK] - 1] = data[i];
        --counterArr[(data[i].first >> byte) & BIT_MASK];
```

```
28          }
29
30          data = std::move(helperArr);
31      }
32
33      void radixSort(array_t& data) {
34          for (int byte = 0; byte != 64; byte += 8) {
35              countingSort(data, byte);
36          }
37      }
38
39      std::istream& operator >> (std::istream& is, std::pair<ull, ull>& pair) {
40          is >> pair.first >> pair.second;
41
42          return is;
43      }
44
45      std::ostream& operator << (std::ostream& os, const array_t& data) {
46          for (const std::pair<ull, ull>& element: data) {
47              os << element.first << "\t" << element.second << "\n";
48          }
49
50          return os;
51      }
52
53      int main() {
54          std::ios_base::sync_with_stdio(false);
55          std::cin.tie(nullptr);
56          std::cout.tie(nullptr);
57
58          array_t data;
59          std::pair<ull, ull> keyValue;
60          while (std::cin >> keyValue) {
61              data.emplace_back(keyValue);
62          }
63
64          if (data.size() == 0) {
65              return 0;
66          }
67
68          radixSort(data);
69
70          std::cout << data << std::endl;
71
72          return 0;
73      }
```

3 Консоль

```
baronpipistron@BaronPIpistron:~/diskran$ g++ -Wall -o out run.cpp
baronpipistron@BaronPIpistron:~/diskran$ ./out
0 15304086347272597642
18446744073709551615 8332868096657407680
0 13545671047587467688
18446744073709551615 9944245427658561165

0 15304086347272597642
0 13545671047587467688
18446744073709551615 8332868096657407680
18446744073709551615 9944245427658561165
```

4 Тест производительности

Тест производительности представляет из себя следующее: время выполнения поразрядной сортировки сравнивается с «std::sort». Формат входных данных состоит из 1 000 000 строк формата "ключ-значение"в соответствии с заданием, описанном выше. Числа от 0 до

$$2^64 - 1$$

.

```
baronpipistron@BaronPIpistron:~/diskran$ g++ -Wall -o out run.cpp
baronpipistron@BaronPIpistron:~/diskran$ ./out <tests.txt
Benchmark
Comparing my RadixSortd and std::sort
/-----------------------------------------------------------------------/
Time of work my RadixSort : 747938 milliseconds
/-----------------------------------------------------------------------/
Time of work my std::sort : 643892 milliseconds
```

Как видно, стандартная сортировка «std::sort» выиграла у поразрядной сортировки. Это произошло по той причине, что у поразрядной сортировки сложность $O(n)$, в то время как «std::sort» представляет из себя множество различных алгоритмов сортировки, которые вызываются в зависимости от входных данных и прочих факторо. Все эти алгоритмы очень хорошо оптимизированы, и поэтому выигрывают в скорости у самописной сортировки.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать поразрядную сортировку и попрактиковался в оптимизации алгоритма в целях ускорения выполнения программы по времени и затрачиваемой памяти. Одним из преимуществ поразрядной сортировки является её эффективность для сортировки целых чисел или строк. Однако поразрядная сортировка требует дополнительной памяти для хранения корзин, что может сделать её менее эффективной для больших данных, особенно если количество уникальных значений велико. Подводя итог, хотел бы сказать, что алгоритм сортировки нужно выбирать исходя из поставленных задач, типа входных данных и их размера. Не всегда самый быстрый алгоритм будет самым удачным выбором. Иногда гораздо оптимальнее и проще будет выбрать алгоритм с меньшей эффективностью.

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.))

[2] *Поразрядная сортировка — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Поразрядная_сортировка`