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: И. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 03.10.2020
Оценка: отлично
Подпись:

Москва, 2020

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: `Сортировка подсчётом.`

Вариант ключа: `Почтовые индексы.`

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

Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x определить количество элементов, которые меньше x [1].

Для этого требуется создать массив $counts$, размер которого есть супремум множества ключей m (определяется при считывании данных). При первом проходе по данным необходимо посчитать, сколько элементов с каждым ключом у нас имеется. На основании этих данных можно посчитать количество элементов, ключи которых не больше ключа i-ого элемента, $i = [1, m]$, и записать его в $counts[i]$. Таким образом, в массиве $counts$ хранятся самые правые позиции, на которые можно поставить элементы с соответствующими ключами в отсортированном массиве $result$.

Выполним проход по исходному массиву справа налево (для устойчивости сортировки). Каждый элемент будем ставить на самую правую возможную позицию (определяем по массиву $counts$). Когда мы вставляем элемент в $result$, в $counts$ необходимо пометить, что мы заняли ячейку и теперь самая правая возможная позиция для элемента с таким же ключом сдвигается на единицу влево.

В итоге мы получаем массив $result$, элементы которого отсортированы по возрастанию ключа.

Пусть дано n элементов. Выполнение программы состоит из следующих этапов: считывание данных и поиск максимального ключа + подсчет количества элементов с каждым ключом + подсчет количества элементов, ключи которых не больше ключа данного элемента + вставка элементов в итоговый массив. Таким образом, сложность алгоритма: $O(n + n + (m - 1) + n) = O(3n + m)$. Будем учитывать, что $m \sim 1e6$, $n \sim 1e6$, значит $m \sim n$, а следовательно итоговая сложность алгоритма будет равна $O(3n + n) = O(4n) = O(n)$.

2 Исходный код

На каждой непустой строке входных данных располагается пара «ключ-значение», поэтому создадим новую структуру $TItem$, в которой будем хранить ключ (индекс) и значение (строку). Максимальное возможное значение ключа - 999999. Поэтому будем хранить ключ как int. Для хранения строки фиксированного размера воспользуемся $char[65]$ (длина строки 64 + символ окончания строки $'\backslash 0'$).

Количество строк входных данных неизвестно, поэтому необходимо реализовать динамическую структуру для хранения любого количества данных. Для этого создадим класс $TVector$, который содержит указатель на динамический массив, текущий размер и вместимость массива. В конструкторе по умолчанию выделяется память на 10 элементов. Если потребуется хранить больше 10 элементов, то вместимость увеличится в два раза. В конструктор можно передать другой начальный размер для динамического массива. Вся выделенная память освобождается в деструкторе. $TVector$ поддерживает добавление элемента в конец динамического массива. Можно создать объект $TVector$ для хранения элементов любого типа данных.

Входные данные считываются в $TVector->data$. Во время считывания также определяется максимальный полученный ключ max_key.

Для сортировки исполльзуется функция $CountingSort$, принимающая исходные данные и максимальное значение ключа. Вектор для отсортированных данных также передается по ссылке в качестве аргумента.

Внутри функции создается вектор целых чисел $counts$ размера max_key. Изначально он заполнен нулями. Мы проходим по всем входным данным и инкрементируем значение массива $counts[i]$, где i - ключ. Таким образом, мы можем узнать, сколько элементов с каждым ключом у нас есть. Теперь необходимо узнать самую правую позицию, на которую можно поставить элемент с каждым ключом в отсортированном векторе. Для этого посчитаем количество всех элементов с ключом, меньшим или равным x, для всех x от 1 до max_key.

Чтобы расставить элементы в отсортированном порядке, нужно пройтись по ним справа налево и каждый элемент располагать в $result$ в ячейку, соответствующую значению $counts[i]-1$, где i - ключ текущего рассматриваемого элемента. После этого нужно сдвинуть самую правую позицию для этого ключа, декрементировав $counts[i]$. Таким образом, получили вектор $result$ с отсортированными входными данными.

```cpp
#include <iostream>
#include <cstdlib>

template <typename T>
class TVector {
public:
    TVector() {
        Data = (T*)calloc(10, sizeof(T));
        Capacity = 10;
        CurSize = 0;
    }

    TVector(int size) {
        Data = (T*)calloc(size, sizeof(T));
        Capacity = size;
        CurSize = size;
    }

    ~TVector() {
        free(Data);
    }

    void PushBack(T elem) {
        if (CurSize == Capacity) {
            Extend();
        }
        Data[CurSize++] = elem;
    }

    int Size() {
        return CurSize;
    }

    T& operator[](int id) {
        return Data[id];
    }

private:
    T* Data;
    int Capacity;
    int CurSize;

    // doubles capacity
    void Extend() {
        Data = (T*)realloc(Data, 2 * Capacity * sizeof(T));
        Capacity *= 2;
    }
};

```

```cpp
struct TItem {
    int key;
    char value[65];
};

void CountingSort(TVector <TItem>& data, TVector <TItem>& result, int max_key) {
    TVector <int> counts(max_key + 1);
    for (int i = 0; i < data.Size(); ++i) { // counting how many times each key appears
         in data
        counts[data[i].key]++;
    }
    for (int i = 1; i <= max_key; ++i) { // counting how many items have keys <= than
        counts[i]
        counts[i] += counts[i - 1];
    }
    for (int i = data.Size() - 1; i >= 0; --i) { // creating sorted vector
        result[--counts[data[i].key]] = data[i];
    }
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    TVector <TItem> data;
    TItem elem;
    int max_key = -1;
    while (std::cin >> elem.key >> elem.value) {
        if (elem.key > max_key) { // searching for upper bound of keys
            max_key = elem.key;
        }
        data.PushBack(elem);
    }
    TVector <TItem> result(data.Size());
    CountingSort(data, result, max_key);
    for (int i = 0; i < result.Size(); ++i) {
        std::cout.fill('0');
        std::cout.width(6);
        std::cout << result[i].key << " " << result[i].value << "\n";
    }
    return 0;
}
```

3 Консоль

```
mosik@LAPTOP-69S778GL:~/da_lab1$ g++ da_lab1.cpp -Wall -o da_lab1
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_generator.py
mosik@LAPTOP-69S778GL:~/da_lab1$ cat test.txt
922480 vzvnt
608394 raofhrclssrojinfjzwyjvmmfpzjelcwhqjfcfxgxcnevyxidtaeqalp
160454 syvcamektjfybgpdewuccwx
307973 bqvpixdgmjjmslholhsexnbzyyzdtapdfyrtouljkuvnysztfozfnsuldneodeo
845272 ndiyahzlcqeqtxveykaqmvlqthhvboxfpnwnirniuagdjppyylzgrxnafdajqgdw
183238 zzyxptgmcjcykndvssmvhotoh
807526 adjlxvjmwvcxujaekz
871501 jpumbbnp
524624 zzcgqhmaxusjynypsdy
038571 q
640307 yddvqw
556955 lsltjtaxzcuchxtazhsjhkvoknhdaejkvvhgvmsvv
455763 gfakjbxeyinjqdlgqdlnbkqptxzjqvxeqtckqdqtlqsdatyhwmtrk
236613 tpwergtdmxide
328559 eyzcqocvckeeiwxxhtldzyhocgrxmx
mosik@LAPTOP-69S778GL:~/da_lab1$ ./da_lab1 <test.txt >result.txt
mosik@LAPTOP-69S778GL:~/da_lab1$ cat result.txt
038571 q
160454 syvcamektjfybgpdewuccwx
183238 zzyxptgmcjcykndvssmvhotoh
236613 tpwergtdmxide
307973 bqvpixdgmjjmslholhsexnbzyyzdtapdfyrtouljkuvnysztfozfnsuldneodeo
328559 eyzcqocvckeeiwxxhtldzyhocgrxmx
455763 gfakjbxeyinjqdlgqdlnbkqptxzjqvxeqtckqdqtlqsdatyhwmtrk
524624 zzcgqhmaxusjynypsdy
556955 lsltjtaxzcuchxtazhsjhkvoknhdaejkvvhgvmsvv
608394 raofhrclssrojinfjzwyjvmmfpzjelcwhqjfcfxgxcnevyxidtaeqalp
640307 yddvqw
807526 adjlxvjmwvcxujaekz
845272 ndiyahzlcqeqtxveykaqmvlqthhvboxfpnwnirniuagdjppyylzgrxnafdajqgdw
871501 jpumbbnp
922480 vzvnt
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_generator.py
mosik@LAPTOP-69S778GL:~/da_lab1$ cat test.txt
```

```
360908 swxelqjpuqxtzocomwijjcfqbkrwcctysfsbj
918658 efytoeifuqewkajsizwdedsg
008403 gudzhlmalrntkxunrvsplwiqamwhknasvfvifpxstnhykpypjdwqkzaxbgzul
315474 qth
664491 hoonqshpsbfnkntdkmkxwl
322775 wslnqnhtqeqzmvduxbpjyvjwxvxcfvnvmlkediwumfspjnsqyesqalymis
599296 huxzealddegxdayyegfmrwbqdiknzawoobekj
mosik@LAPTOP-69S778GL:~/da_lab1$ ./da_lab1 <test.txt >result.txt
mosik@LAPTOP-69S778GL:~/da_lab1$ cat result.txt
008403 gudzhlmalrntkxunrvsplwiqamwhknasvfvifpxstnhykpypjdwqkzaxbgzul
315474 qth
322775 wslnqnhtqeqzmvduxbpjyvjwxvxcfvnvmlkediwumfspjnsqyesqalymis
360908 swxelqjpuqxtzocomwijjcfqbkrwcctysfsbj
599296 huxzealddegxdayyegfmrwbqdiknzawoobekj
664491 hoonqshpsbfnkntdkmkxwl
918658 efytoeifuqewkajsizwdedsg
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
```

4 Тест производительности

Для сравнения производительности мною была написана программа на C++, использующая алгоритм устойчивой сортировки из стандартной библиотеки. Было подготовлено два тестовых набора, один из которых состоял из 1000 элементов, другой из 1000000. Произведены замеры времени работы сортировки подсчетом и сортировки из стандартной библиотеки C++.

```
mosik@LAPTOP-69S778GL:~/da_lab1$ g++ da_lab1.cpp -o count_sort
mosik@LAPTOP-69S778GL:~/da_lab1$ g++ STL_stable_sort.cpp -o std_sort
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_generator.py
mosik@LAPTOP-69S778GL:~/da_lab1$ echo "1000 items"
1000 items
mosik@LAPTOP-69S778GL:~/da_lab1$ ./count_sort <test.txt >result.txt
Time: 0.015625 seconds
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
mosik@LAPTOP-69S778GL:~/da_lab1$ ./std_sort <test.txt >result.txt
Time: 0.015625 seconds
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_generator.py
mosik@LAPTOP-69S778GL:~/da_lab1$ echo "1000000 items"
1000000 items
mosik@LAPTOP-69S778GL:~/da_lab1$ ./count_sort <test.txt >result.txt
Time: 0.671875 seconds
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
mosik@LAPTOP-69S778GL:~/da_lab1$ ./std_sort <test.txt >result.txt
Time: 7.78125 seconds
mosik@LAPTOP-69S778GL:~/da_lab1$ python test_checker.py
Correct!
```

По результатам двух тестов видно, что на небольших данных алгоритм сортировки подсчетом не дает выигрыша по времени. Но когда нужно отсортировать большой набор данных, стандартная сортировка значительно проигрывает по времени.

Это происходит из-за того, что в стандартной библиотеке C++ используется алгоритм сортировки, работающий за $O(n * logn)$.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я понял, что сортировки сравнениями выполняются за $\Omega(n * logn)$. Из этого следует, что для сортировки за линейное время придется отказаться от сравнений элементов и придумать более нестандартный алгоритм.

Мною была изучена и реализована сортировка подсчетом. Она работает за линейное время, однако это не делает ее лучшим алгоритмом сортировки. Дело в том, что реализация такого алгоритма требует $O(max\_key)$ дополнительной памяти. Порой это может быть очень затратно. Например, если у нас есть два элемента: 1 и 100000, то создастся вспомогательный массив на миллион элементов, из которых только два элемента будут ненулевыми.

Еще один минус в том, что сортировать подсчетом можно только целочисленные значения (или в крайнем случае значения тех типов, область значений которых конечна). Вещественные значения, строковые значения, абстрактные типы данных отсортировать не получится.

Поэтому область применимости сортировки подсчетом сильно ограничена. Неуместное применение этой сортировки может привести к нерациональному использованию памяти, что в некоторых случаях может оказаться критичным. Из-за этого в стандартной библиотеке C++ используется более долгий, но в то же время более универсальный алгоритм сортировки.

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.))