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

Выполнил студент группы 08-208 МАИ *Зинин Владислав*.

Условие

Кратко описывается задача:

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

2. Вариант задания: Карманная сортировка.

   Тип ключа: Числа от 0 до $2^{64} - 1$.

   Тип 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.

Метод решения

Для хранения ключа и значения я буду использовать структуру Map, котрая хранит в себе соответственно ключ key типа uint64_t и значение value типа string. Реализация карманной сортировки осуществляется следующим образом: сначала я высчитываю максимальный и минимальный ключи в массиве (в качестве массива я использую vector), после чего я высчитываю интервал между промежутками для определения в нужную корзину того или иного ключа. Формула для вычисдения интервала следующая: из максимального элементы вычитаю минимальный и делю все это на размер массива $interval = \frac{maxElemen - minElemen}{inputed.size()}$. Также создаю массив массивов buckets размером inputed.size(), после чего циклом прохожусь по всем элементам, определяю номер их корзины с помощью формулы $elem = \frac{inputed[i] -> key - minElemen}{interval}$ и распределяю элементы по соответствующим корзинам. Далее я сортирую каждую корзину сортировкой вставками, после чего все числа корзин отсортированны, а также отсортированны корзина одна относительно другой. После чего я скрепляю все корзины попорядку и возвращаю готовый массив и вывожу его.

Описание программы

Программа написана в 1 файлах: main.cpp.

Map - структура для хранения ключа и значения.

InsertionSort() - функция сортировки вставками.

BucketSort() - карманная сортировка.

main() - содержит считывание тестовых данных, применение карманной сортировки и вывод результата.

Исходный код

main.cpp

:

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

using namespace std;

struct Map {
    uint64_t key;
    string value;
};

void InsertionSort(vector<Map*>& buckets){
    for(int16_t i = 1; i < buckets.size(); i++){
        for(int16_t j = i; j > 0; --j){
            if (buckets[j - 1]->key > buckets[j]->key) {
                swap(buckets[j - 1], buckets[j]);
            }
        }
    }
}

void BucketSort(vector<Map*>& inputed){
    uint64_t maxElemen = inputed[0]->key;
    uint64_t minElemen = inputed[0]->key;
    for(const auto &item: inputed){
        if (item->key > maxElemen) maxElemen = item->key;
        if (item->key < minElemen) minElemen = item->key;
    }
    const long double interval = (maxElemen - minElemen) / inputed.size();
    vector<vector<Map*>> buckets(inputed.size());
    for(int i = 0; i < inputed.size(); ++i){
        uint64_t elem = (inputed[i]->key - minElemen) / interval;
        if(elem == inputed.size())
            elem--;
        buckets[elem].push_back(inputed[i]);
    }
```

```
    for(int i = 0; i < buckets.size(); ++i){
        InsertionSort(buckets[i]);
    }
    int pos = 0;
    for(int i = 0; i < buckets.size(); ++i){
        for(int j = 0; j < buckets[i].size(); ++j){
            inputed[pos++] = buckets[i][j];
        }
    }
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    string str;
    uint64_t key;
    vector<Map*> inputed;
    while(cin >> key >> str){
        Map* data = new Map();
        data->key = key;
        data->value = str;
        inputed.emplace_back(data);
    }
    if(inputed.size() != 0){
        BucketSort(inputed);
    }
    for(const auto &item: inputed){
        cout << item->key << '\t' << item->value << "\n";
    }
    for (int i = 0; i < inputed.size(); ++i) {
        delete inputed[i];
    }
    return 0;
}
```

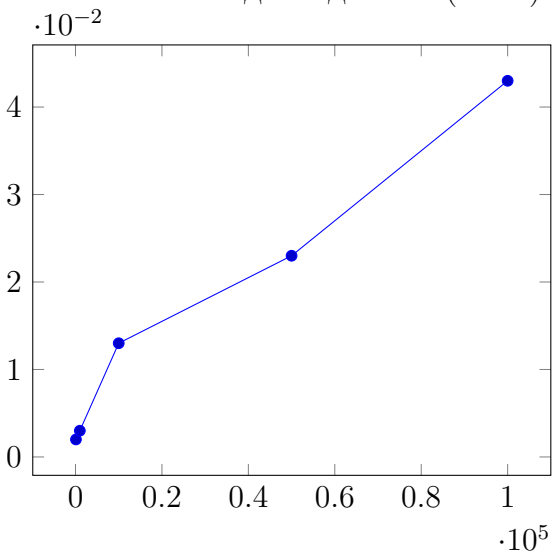
Дневник отладки

Изначально я не неправильно учел размер данных, которые могут поступить в качестве теста, что послужило причиной первых неудач, после чего я поменял размер данных с обычного int до uint64_t. Исправив данную проблему я с толкнулся с еще одной, более сложной для меня - превышенный лимит памяти. Перепробовав много различных

вариантов, я решил отказаться от шаблон класса pair, заменив её на простую структуру Map, в результате чего у меня получилось сократить используемую память почти в два раза.

Тест производительности

Для теста производительности я создал пять файлов, в которых сгенерировал 100, 1000, 10000, 50000, 100000 входных данных.
Получились следующие результаты:
100 входных данных - 0.002 s,
1000 входных данных - 0.003 s,
10000 входных данных - 0.013 s,
50000 входных данных - 0.023 s,
100000 входных данных - 0.043 s,
По результатам теста я построил график зависимости времени работы программы (ось y) от количества входных данных (ось x):



В результате, график получился практически линейный, а следовательно рост времени работы при увеличении объема входных данных согласуется с заявленной сложностью карманной сортировки - O(n+k).

Недочёты

Недочеты мною не были обнаружены, поскольку программа работает успешно при вводе корректных данных. При вводе некорректных данных программа может рабоатть некорректно.

Выводы

В данной лаболаторной работе я на практике познакомился с карманной сортировкой, успешно её реализовал. Если входные элементы подчиняются равномерному закону распределения, то математическое ожидание времени работы алгоритма карманной сортировки является линейным. Это возможно благодаря определенным предположениям о входных данных. При карманной сортировке предполагается, что входные данные равномерно распределены на отрезке $[0, 1)$. В моем же случае входные данные данные - целые числа на отрезке от 0 до $2^{64} - 1$. По моему мнению, данная сортировка сильно деградирует в том случае, когда неправильно подсчитан интервал и все числа оказались в одной корзине. В таком случае сложность алгоритма может стать квадратичной. Также для корзин необходима память, поэтому в случае больших входных данных это может сильно сказаться на потреблении памяти.