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: А. А. Боглаев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22
Дата:
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.

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

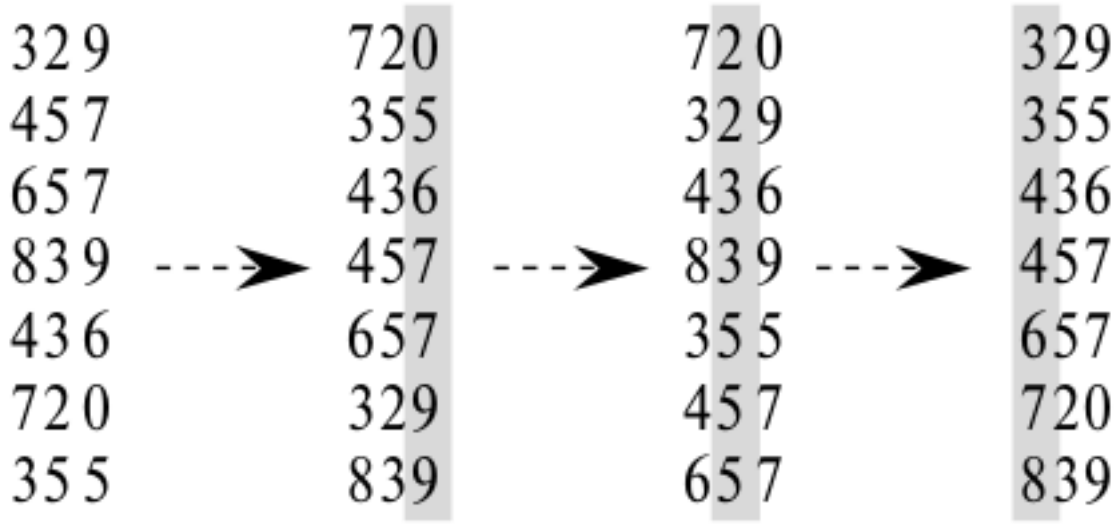
Вариант значения: `строки переменной длины (до 2048 символов)`.

1 Описание

Требуется написать реализацию алгоритма поразрядной сортировки для упорядочивания пар «ключ-значение» по возрастанию.

Как сказано в [1]: «Поразрядная сортировка - это алгоритм, который использовался в машинах, предназначеных для сортировки перфокарт, состоящих из d-значных чисел. 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 продолжается до тех пор, пока перфокарты не окажутся отсортированными по всем d цифрам. После этого перфокарты оказываются полностью отсортированы в порядке возрастания d-значных чисел. Таким обра зом, для сортировки перфокарт требуется лишь d проходов колоды. Ниже представлен пример использования данного алгоритма для сортировки 3-х значныйх чисел.

Важно, чтобы сортировка по цифрам того или иного разряда в этом алгоритме обладала устойчивостью!».



2 Исходный код

На каждой непустой строке входного файла располагается пара «ключ-значение», поэтому создадим новый объект класса *TObject*, в которой будем хранить ключ и значение. У этого класса реализованы коснструкторы и деструкторы, а также переопределен опрератор присваивания и оператор вывода.

- TObject() - дефолтный конструктор, по сути собирает простую пару, где ключ равен 0, а занчение это нулевой указатель
- TObject(key, value) - конструктор, на который принимает ключ и значение
- TObject(TObject & other) - копирующий конструктор
- TObject() - деструктор
- operator= - оператор присваивания
- operator« - опрератор вывода

Важно отметить, что в поле значения класса *TObject* лежит указатель на строку. Как показали тесты, такой способ хранения строк помогает расходовать меньше памяти во время выполенения алгоритма сортировки. При обработке вектора в процессе сортировки происходит перемещение элементов из одного вектора в другой, а также элементы перемещаются из одного буфера в другой при расширении или сужения буфера. Если хранить в поле класса сами строки, то программа будет использовать больше памяти и времени.

Нам нужно где-то хранить наши объекты, чтобы их можно было отсортировать и вывести. Так как мы не знаем, сколько пар поступит на вход программе, нужно использовать структуру, которая может граммотно изменять размеры своего буфера. В ходе лабораторной работы зарпещено использовать «stl» контейнеры. Поэтому напишем свой «простой» вектор. Этот контейнер отдаленно напоминает стандартный вектор. В нем есть такие методы, как:

- PushBack - вставка элемента в конец буфера
- PopBack - удаление элемента из конца буфера
- Size - возвращает значение длины вектора
- Reserve - выделение памяти

- operator[] - получение элемента по индексу
- operator << - печать объектов вектора

Данный вектор написан в виде шаблонного класса $TSimpleVector$. В этом классе есть также конструкторы и деструктор.

- TSimpleVector() - дефолтный конструктор
- TSimpleVector(const int & n) - конструктор, который выделяет буфер на n значений
- TSimpleVector(const int & n, const T & value) - конструктор, который выделяет буфер на n значений и заполняет его значениями value
- TSimpleVector(TSimpleVector<T>&& other) - перемещающий конструктор (move constructor)
- TSimpleVector() - деструктор, специальный метод класса, который автоматически вызывается при уничтожении объекта. Он используется для освобождения ресурсов, выделенных объектом во время его жизни.

Конструктор перемещения - это специальный конструктор класса, который принимает временный объект (rvalue) в качестве параметра и перемещает его ресурсы в новый объект, вместо их копирования.

Важным методом нашего вектора является метод $Reserve$. У вектора есть три параммметра: размер, объем и буфер. В данный метод позволяет изменять объем буфера. Когда нам важно изменять объем буфера?

В основном, когда размер вектора равен объему. В стандартном векторе используются сложные конструкции, чтобы максимально экономно расходовать память. В нашем же случае сложные конструкции пока не известны нам, поэтому мы просто стараемся увеличивать объем в 2 раза.

При выполнении лабораторной работы не использовал метод $PopBack$, но в нем тоже реализован механизм уменьшения памяти.

С хранением элементов разобрались, теперь рассмотрим алгоритм сортировки.

Почему поразрядная сортировка? Потому что сортируются разряды чисел.

Поразрядная сортировка в нашей реализации состоит из двух частей:

- функция для получения разряда числа $GetDigit$

- сортировка разрядов

Для хранения ключа используется тип данных uint64_t - 8 байт. Для получения разряда не нужно использовать деление на 10 и деление с остатком. Этот способ работает, но он для больших чисел он будет неэффективным. Поэтому нужно разбить число на байты. Можно сравнить два подхода. В функции поразрядной сортировки запускаем сотировку разрядов в цикле. Цикл можем запустить от 0 до 14 (макс колво цифр в числе) или от 0 до 8. Очевидно, что второй вариант более привлекателен с точки зрения экономии ресурсов.

Внутри функции *GetDigit* выполняется следующее:

1. Сдвиг числа elem на 8 * i битов вправо с помощью оператора >>. Это позволяет получить нужную позицию цифры в числе.
2. Применение побитовой маски 0xFF с помощью оператора &. Маска 0xFF представляет **байт** со значением 255 в двоичной системе. Применение этой маски позволяет извлечь только младший байт (8 бит) из результата сдвига, что соответствует значению цифры.
3. Полученное значение цифры сохраняется в переменную *digit*.
4. Функция возвращает значение *digit*.

Как уже описывалось выше, разряды чисел должны сортироваться устойчивой сортртировкой. Мы будем использовать сортировку подсчетом.

Приведем краткий разбор алгоритма по шагам.

1. создание массива подсчета
2. подсчет элементов массива
3. вычисление префикс-суммы для каждого элемента в массиве подсчета
4. создание результирующего вектора
5. вычисление позиции текущего элемента в результирующем векторе

Алгоритм сортировки посчетом достаточно прост. Нужно создать массив подсчета, куда будем записывать количество элемнетов которые нам повстречались при обходе. В нашем случае такой массив может быть максимум размером 256 элементов, что соотвествуент 1 байту. Поэтому нет смысла использовать наш вектор.

После нужно вычислить префикс-сумму для каждого значения в массиве подсчета, начиная со 2 элемента. Данная префикс-сумма обозначает количество элементов, которые меньше или равны текущему элементу и стоят где-то перед ним. Поэтому и начинаем со второго элемента, перед первым ничего не стоит.

Далее нам нужен результирующий вектор, куда мы будем помещать элементы.

В сортировки подсчетом нет сравнения элементов. Элементы сортируются за счет вычисления их позиции в результирующем векторе. Позиция вычисляется как значение массива подсчета (от текущего разряда) минус 1.

На вычесленную позицию ставим элемент исходного вектора и уменьшаем значение в массиве подсчета.

Важно отметить, что для расстановки элементов в результирующем векторе цикл запускают от последнего элемента к первому. Именно так можно расставить элементы в правильном порядке, как раз для этого мы вычисляли префикс-сумму.

После выполенения цикла, перемещаем новую последовательность из результирующего вектора в исходный.

После того как выполним 8 раз порязрядную сортировку, наша исходная последовательность будет упорядочена по возрастанию.

Чтобы программа не превысила времени выполнения, придется переписать ввод и вывод пар. Вместо использования cin и cout, бедем использовать scanf и printf.

```
#include <iostream>
#include <memory>

const int START_CAP = 100;
const int BUFFER_EXPAND_VALUE = 2;

template<class T>
class TSimpleVector {
   private:
       T* buffer;
       int size;
       int cap;
   public:
       TSimpleVector();
       TSimpleVector(const int & n);
       TSimpleVector(const int & n, const T & value);
       TSimpleVector(TSimpleVector<T>&& other); // move constructor
       ~TSimpleVector();

```

```
20          void Reserve(const int new_cap)
21          void PushBack(const T & value);
22          void PopBack();
23          int Size() const; // getter
24
25          T& operator[](const int & i);
26          TSimpleVector& operator=(TSimpleVector<T>&& other);
27
28          friend std::ostream& operator<<(std::ostream& os, const TSimpleVector<T>& other
                );
29  };
30
31  class TObject {
32      public:
33          uint64_t key;
34          std::shared_ptr<std::string> value;
35
36          TObject();
37          TObject(const uint64_t& c_key, const std::string& c_value);
38          TObject(const TObject& other);
39          ~TObject() noexcept;
40
41          TObject& operator=(const TObject& other);
42          friend std::ostream& operator<<(std::ostream& os, const TObject & other);
43  };
44
45  const int COUNT_MS_SZ = 256;
46  const int MAX_STR_SIZE = 2049;
47  const int BYTE_VAL = 8;
48
49  int GetDigit(uint64_t & elem, int& i) {
50    int digit = (elem >> (BYTE_VAL * i)) & 0xFF;
51    return digit;
52  }
53
54  void CountingSort(TSimpleVector<TObject>& mas, int& i) {
55    int sz = mas.Size();
56    int cnts[COUNT_MS_SZ] = {0};
57
58    for (int j = 0; j < sz; j++) {
59      cnts[GetDigit(mas[j].key, i)]++;
60    }
61
62    for (int j = 1; j < COUNT_MS_SZ; j++) {
63      cnts[j] += cnts[j - 1];
64    }
65
66    TSimpleVector<TObject> interm_result(sz);
67
```

```
68    for (int j = sz - 1; j >= 0; j--) {
69      int pos = cnts[GetDigit(mas[j].key, i)] - 1;
70      interm_result[pos] = mas[j];
71      cnts[GetDigit(mas[j].key, i)] = pos;
72    }
73
74    mas = std::move(interm_result);
75  }
76
77  void Radix(TSimpleVector<TObject>& mas) {
78    for (int i = 0; i < (int)sizeof(uint64_t); i++) {
79      CountingSort(mas, i);
80    }
81  }
82
83  int main() {
84    TSimpleVector<TObject> mas;
85
86    uint64_t key;
87    char str[MAX_STR_SIZE];
88    while (scanf("%lu\t%[^\n]", &key, str) != EOF) {
89      mas.PushBack(TObject(key, std::string(str)));
90    }
91
92    Radix(mas);
93
94    for (int i = 0; i < mas.Size(); i++) {
95      printf("%lu\t%s\n", mas[i].key, mas[i].value->c_str());
96    }
97
98  }
```

3 Консоль

```
alex@wega:~/da_labs/Lab_01$ make
g++ -std=c++17 -pedantic -Wall main.cpp -o lab1
alex@wega:~/da_labs/Lab_01$ cat tests/01.t
17832977662492897515    lMlOCeWaHK
2996444704890835419     eoRbLbhgnV
14252360749301456558    XViwJgagyL
8509929984979068612     lEPeVZQJxj
16872145630083976482    OzbvidEVHy
12066295488853181134    BfJiYdMhdv
4473799090154691171     RHxbAUdqfb
1681540776916609004     EGLTpVNggw
5616851010130146808     xlYdMhNVCa
6958989457477228380     mXzlaEkRSD
alex@wega:~/da_labs/Lab_01$ ./lab1 <tests/01.t
1681540776916609004     EGLTpVNggw
2996444704890835419     eoRbLbhgnV
4473799090154691171     RHxbAUdqfb
5616851010130146808     xlYdMhNVCa
6958989457477228380     mXzlaEkRSD
8509929984979068612     lEPeVZQJxj
12066295488853181134    BfJiYdMhdv
14252360749301456558    XViwJgagyL
16872145630083976482    OzbvidEVHy
17832977662492897515    lMlOCeWaHK

alex@wega:~/da_labs/Lab_01$ make
g++ -std=c++17 -pedantic -Wall main.cpp -o lab1
alex@wega:~/da_labs/Lab_01$ cat tests/01.t
31      G
10      h
86      T
23      x
33      U
59      s
37      K
28      e
16      V
100     P
alex@wega:~/da_labs/Lab_01$ ./lab1 <tests/01.t
```

```
10      h
16      V
23      x
28      e
31      G
33      U
37      K
59      s
86      T
100     P

alex@wega:~/da_labs/Lab_01$ make
g++ -std=c++17 -pedantic -Wall main.cpp -o lab1
alex@wega:~/da_labs/Lab_01$ cat tests/01.t
44      vADburXqfnTEriMoBSYX
53      fsyBYfJCGcmDRfUyEyKe
0       XEvKskWSoFDjdhrFTBtn
81      BxmBxxSFIAdkdFtZBtaz
87      bqjHGUGbjkMFmYWtJKSP
58      ZGxZTCJICBOHqnmGMXTl
74      NtEemPdRbdbybJMyfhMP
86      sXCpqkqXiBKLhmbyMXKX
41      dUwSlJRZtsodWiZTLNVn
24      tKNWPsWSuvPxydjtcAeM
alex@wega:~/da_labs/Lab_01$ ./lab1 <tests/01.t
0       XEvKskWSoFDjdhrFTBtn
24      tKNWPsWSuvPxydjtcAeM
41      dUwSlJRZtsodWiZTLNVn
44      vADburXqfnTEriMoBSYX
53      fsyBYfJCGcmDRfUyEyKe
58      ZGxZTCJICBOHqnmGMXTl
74      NtEemPdRbdbybJMyfhMP
81      BxmBxxSFIAdkdFtZBtaz
86      sXCpqkqXiBKLhmbyMXKX
87      bqjHGUGbjkMFmYWtJKSP
alex@wega:~/da_labs/Lab_01$
```

4 Тест производительности

Тест производительности представляет из себя следующее: будем сравнивать время выполнения нашей поразрядной сортировки и встроенной сортировки.

```
alex@wega:~/da_labs/Lab_01$ make
g++ -std=c++17 -pedantic -Wall main.cpp -o lab1
g++ -std=c++17 -pedantic -Wall benchmark.cpp -o bench1
alex@wega:~/da_labs/Lab_01$ ./bench1 <tests/01.t
Count lines is: 100
Radix_sort time: 0.243 ms
Stable_sort time: 0.24 ms
Difference: 0.987654
alex@wega:~/da_labs/Lab_01$ ./bench1 <tests/01.t
Count lines is: 1000
Radix_sort time: 0.677 ms
Stable_sort time: 1.118 ms
Difference: 1.6514
alex@wega:~/da_labs/Lab_01$ ./bench1 <tests/01.t
Count lines is: 10000
Radix_sort time: 7.36 ms
Stable_sort time: 16.426 ms
Difference: 2.23179
alex@wega:~/da_labs/Lab_01$ ./bench1 <tests/01.t
Count lines is: 50000
Radix_sort time: 52.646 ms
Stable_sort time: 100.743 ms
Difference: 1.91359
alex@wega:~/da_labs/Lab_01$
```

Можем заметить, что поразрядная сортировка примерно в 2 раза быстрее, это можно объяснить тем, что сложность поразрядной сортировки $O(n*k)$ (n - кол-во элементов в последовательности, а k - количество значащих разрядов в максимальном элементе), а у встроенной сортировки сложность $O(n * logn)$.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я применил полученные знания из курса ООП (Объектно Ориентирванное Программирование) для написания вектора, узнал и реализовал новые алгоритмы сортировки за линейное время. В ходе работы познакомился с новыми инструментами профилирования программ, такими как Valgrind и Massif. С их помощью можно проанализировать использование памяти и времени программой. В ходе тестирования узнал, что функции ввода и вывода в языке программирования C++ работают медленнее, чем в языке программирования Си, поэтому пришлось немного адаптировать программу.

Лабораторная работа достаточно интересная и учит рациональному использованию памяти и времени. Считаю, что полученные мною навыки помогут мне при решении дальнейших задач.

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.))