

**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  $2^{64} - 1$ .

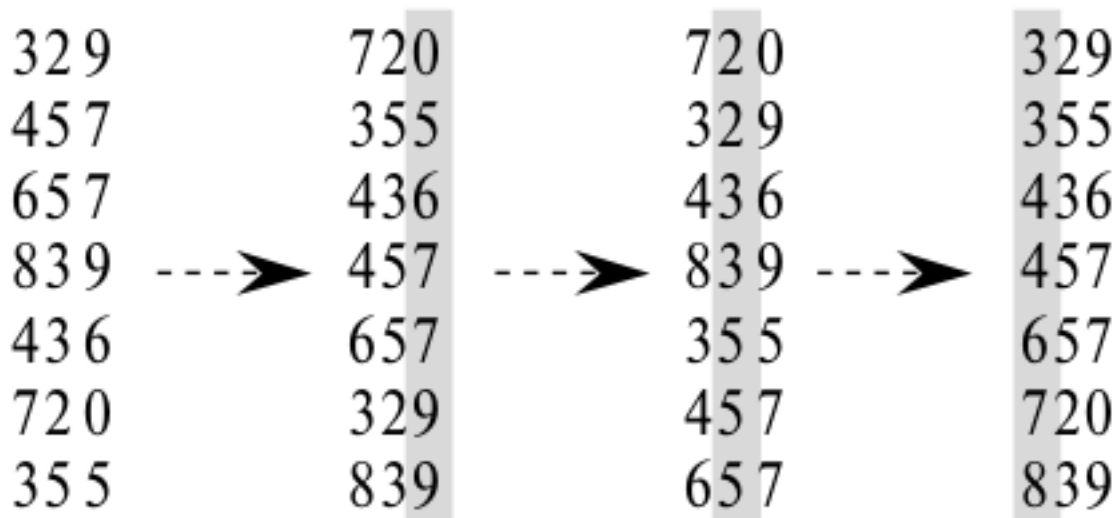
**Вариант значения:** строки переменной длины (до 2048 символов).

# 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, предназначенных 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 образом, для сортировки перфокарт требуется лишь  $d$  проходов колоды. Ниже представлен пример использования данного алгоритма для сортировки 3-х значных 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 конструкторы и деструкторы, а также переопределен оператор присваивания и оператор вывода.

- *TObject()* - дефолтный конструктор, по сути собирает простую пару, где ключ равен 0, а значение это нулевой указатель
- *TObject(key, value)* - конструктор, на который принимает ключ и значение
- *TObject(TObject & other)* - копирующий конструктор
- *TObject()* - деструктор
- *operator=* - оператор присваивания
- *operator«* - оператор 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 выполнения 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 грамотно изменять размеры своего буфера. В ходе лабораторной работы запрещено 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 параметра: 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 сортировку разрядов в цикле. Цикл можем запустить от 0 до 14 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 сортировкой. Мы будем использовать сортировку подсчетом.

Приведем краткий разбор алгоритма по шагам.

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

Алгоритм сортировки подсчетом достаточно прост. Нужно создать массив подсчета, куда будем записывать количество элементов которые нам повстречались при обходе. В нашем случае такой массив может быть максимум размером 256 элементов, что соответствует 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 вычисленную 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 выполнения цикла, перемещаем новую последовательность из результирующего вектора в исходный.

После того как выполним 8 раз порярядную 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`, будем использовать `scanf` и `printf`.

```
1 | #include <iostream>
2 | #include <memory>
3 |
4 | template<class T>
5 | class TSimpleVector {
6 |     private:
7 |         T* buffer;
8 |         int size;
9 |         int cap;
10 |     public:
11 |         TSimpleVector();
12 |         TSimpleVector(const int & n);
13 |         TSimpleVector(const int & n, const T & value);
14 |         TSimpleVector(TSimpleVector<T>&& other); // move constructor
15 |         ~TSimpleVector();
16 |
17 |         void Reserve(const int new_cap)
18 |         void PushBack(const T & value);
19 |         void PopBack();
20 |         int Size() const; // getter
21 |
22 |         T& operator[](const int & i);
```

```

23         TSimpleVector& operator=(TSimpleVector<T>&& other);
24
25         friend std::ostream& operator<<(std::ostream& os, const TSimpleVector<T>& other
26             );
27     };
28     class TObject {
29     public:
30         uint64_t key;
31         std::shared_ptr<std::string> value;
32
33         TObject();
34         TObject(const uint64_t& c_key, const std::string& c_value);
35         TObject(const TObject& other);
36         ~TObject() noexcept;
37
38         TObject& operator=(const TObject& other);
39         friend std::ostream& operator<<(std::ostream& os, const TObject & other);
40     };
41
42     const int COUNT_MS_SZ = 256;
43
44     int GetDigit(uint64_t & elem, int& i) {
45         int digit = (elem >> (8 * i)) & 0xFF;
46         return digit;
47     }
48
49     void CountingSort(TSimpleVector<TObject>& mas, int& i) {
50         int sz = mas.Size();
51         int cnts[COUNT_MS_SZ] = {0};
52
53         for (int j = 0; j < sz; j++) {
54             cnts[GetDigit(mas[j].key, i)]++;
55         }
56
57         for (int j = 1; j < COUNT_MS_SZ; j++) {
58             cnts[j] += cnts[j - 1];
59         }
60
61         TSimpleVector<TObject> interm_result(sz);
62
63         for (int j = sz - 1; j >= 0; j--) {
64             int pos = cnts[GetDigit(mas[j].key, i)] - 1;
65             interm_result[pos] = mas[j];
66             cnts[GetDigit(mas[j].key, i)] = pos;
67         }
68
69         mas = std::move(interm_result);
70     }

```



```

71
72 void Radix(TSimpleVector<TObject>& mas) {
73     for (int i = 0; i < 8; i++) {
74         CountingSort(mas, i);
75     }
76 }
77
78 int main() {
79     TSimpleVector<TObject> mas;
80
81     uint64_t key;
82     char str[2049];
83     while (scanf("%lu\t%[^\\n]", &key, str) != EOF) {
84         mas.PushBack(TObject(key, std::string(str)));
85     }
86
87     Radix(mas);
88
89     for (int i = 0; i < mas.Size(); i++) {
90         printf("%lu\t%s\\n", mas[i].key, mas[i].value->c_str());
91     }
92
93 }

```

### 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      1Ml0CeWaHK
2996444704890835419      eoRbLbhgnV
14252360749301456558      XViwJgagyL
8509929984979068612      lEPeVZQJxj
16872145630083976482      0zbvidEVHy
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      0zbvidEVHy
17832977662492897515      1Ml0CeWaHK

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  $O(n + k)$ , а у встроенной сортировки сложность  $O(n * \log n)$ .

## 5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я применил полученные знания из курса ООП (Объектно Ориентированное 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.

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.))