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

Выполнил студент группы М8О-208Б-21 МАИ *Белоносов Кирилл.*

Условие

1. Общая постановка задачи:

   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(1.2):

   Алгоритм сортировки: Сортировка подсчётом.
   Тип ключа: числа от 0 до 65535.
   Тип значения: строки переменной длины (до 2048 символов).

Метод решения

В данном задании предложено использовать сортировку подсчетом. Так как входные данные представляют собой пару ключ-значение, то будет использована модифицированная версия сортировки подсчетом, предназначенная для более сложных структур, чем числа.

Пусть на вход нашей программе поступает n пар ключ-значение, хранящиеся в массиве A, причем ключ может лежать в диапазоне от 0 до k - 1. Тогда заведем отдельный массив размером k и посчитаем количество вхождений каждого ключа в наши пары, обозначим данный массив как P. Далее мы преобразуем наш массив P в массив префиксных сумм, причем начнем мы сделаем сдвиг на один элемент, а P[0] присвоим значение 0. Пройдемся по исходному массиву и будем добавлять его элементы в результирующий массив B по следующему правилу: A[i] элемент ставим в массив B на место P[A[i].key], причем после увеличиваем значение P[A[i].key] на 1. Таким образом массив P хранит индексы на которых должны стоять элементы с определенным ключом, добавляя элементы в массив B, по указанному выше правилу, мы получим отсортированный массив наших сложных структур.

Также не менее важной задачей является считывание данных и их хранение. Для хранения ключа и строки нам подойдет структура хранящая ключ и указатель на строку (аналог pair из STL). Указатель на строку был выбран ввиду более экономного способа хранения строки в памяти. Далее нам неизвестно количество поступающих пар на вход нашей программы. В этом нам может помочь динамически расширяемая структура данных - Вектор. Создадим свой Вектор и реализуем его основной функционал - в частности метод push_back(). Это позволит нам добавлять элементы в конец константное время, если не закончилась зарезервированная нами память.

Считывать строку мы будем в буффером размером 2049 байт (1 байт мы зарезервировали для терминального символа \0). Также так как мы будем использовать два вектора - входной и результирующий, то выгоднее всего хранить не сами пары, а указатели на них, что позволит сократить использование памяти

Сложность алгоритма можно оценить в O(n + k), так как нам необходимо пройтись по всему массиву n и также по массиву хранящему количество элементов - это k.

Описание программы

Для работы со входными данными я сделал свою реализацию вектора. Хотя она и не обладает всеми методами, но решает конкретно поставленную задачу.

```
template<class T>
  class vector {
  public:

    vector();
    vector(size_t size);
    vector(size_t size, T value);
    vector(const vector& arg);
    vector<T>& operator=(const vector<T>& arg);
    T& operator [](size_t  index);
    const T& operator [](size_t index) const;
    inline size_t size() const;
    void reserve(size_t new_mem);
    void resize(size_t new_size);
    void push_back(const T& arg);
    ~vector();

  private:

    T* _arr;
    size_t _size;
    size_t _mem;

  };
```

Самым интересным является реализация метода push_back(), который позволяет добавлять элементы в конец массива.

```
template<class T>
  void vector<T>::reserve(size_t new_mem) {
    if(new_mem < _mem) {
      return;
```

```
    }

    T* new_arr = new T[new_mem];
    for(int i = 0; i < this->size(); ++i) {
      new_arr[i] = _arr[i];
    }
    delete[] _arr;
    _arr = new_arr;
    _mem = new_mem;
  }

  template<class T>
  void vector<T>::push_back(const T& arg) {
    if(_mem == 0) {
      this->reserve(10);
    }
    else if(_size == _mem) {
      this->reserve(_size * 2);
    }

    this->_arr[this->_size] = arg;
    ++_size;
  }
```

Далее я реализовал структуру для хранения нашего ключа и строки.

```
struct my_pair {
    my_pair() {
        first = -1;
    }
    my_pair(int f, char * s) : first(f), second(s) {
    }

    ~my_pair() {

    }

    my_pair& operator=(const my_pair& rhs) {
    if(this == &rhs) {
      return *this;
    }

    this->first = rhs.first;
        memcpy(second, rhs.second, 2048);
    return *this;
  }
```

```
    int first_get() const {
        return first;
    }

    const char * second_get() const {
        return second;
    }
    int first;
    char * second;
};
```

Теперь разберемся с самой программой, на вход ей поступает ключи и строки, до тех пор пока не будет достигнут EOF. С помощью цикла while производим считывание наших данных, аллоцируем для них место в памяти и будем передавать указать в массив vec. Потом, уже по шагам алгоритма, создадим массив p, в котором будем считать количество вхождений ключа. Преобразуем p в массив префиксных сумм. Далее создадим второй массив - res, который будем заполнять указателями на пары, в соответствии с нашим правилом. Выведем получившийся массив и не забудем очистить память, выделенную на хранения строк и пар.

```
int main () {
  // fast input
  std::cin.tie(0);
  std::ios::sync_with_stdio(false);
    int x;
  // describe buffer
    char buf[2049];
    vector<my_pair *> vec;
  // read pairs and write in vec
    while(std::cin >> x >> buf) {
    char * p = strdup(buf);
        my_pair * pair = new my_pair(x, p);
        vec.push_back(pair);
    }
    vector<my_pair *> res(vec.size());
    vector<size_t> p(65536, 0);
  // counting keys in vec to p
    for(int i = 0; i < vec.size(); ++i) {
        int ind = vec[i]->first_get();
        ++p[ind];
    }
    size_t sum = 0;
  // transform p in prefix sum array
    for(int i = 0; i < p.size(); ++i) {
        size_t t = p[i];
```

```
        p[i] = sum;
        sum += t;
    }
  // sort element and put in res
    for(int i = 0; i < vec.size(); ++i) {
        res[p[vec[i]->first_get()]] = vec[i];
        ++p[vec[i]->first_get()];
    }
    for(int i = 0; i < res.size(); ++i) {
        std::cout << res[i]->first_get() << "\t" <<
           res[i]->second_get() << "\n";
    }
  // free memory
  for(int i = 0; i < res.size(); ++i) {
        free(res[i]->second);
        delete res[i];
    }
}
```

Дневник отладки

Попытка 1-5 (ID 84059192)

Вплоть до этой попытки большинство ошибок было связано с багами в реализации векторов и работе с памятью. Отладка, включавшая использование gdb и valgrind помогли найти утечки

Попытка 6-16 (ID 84092210)

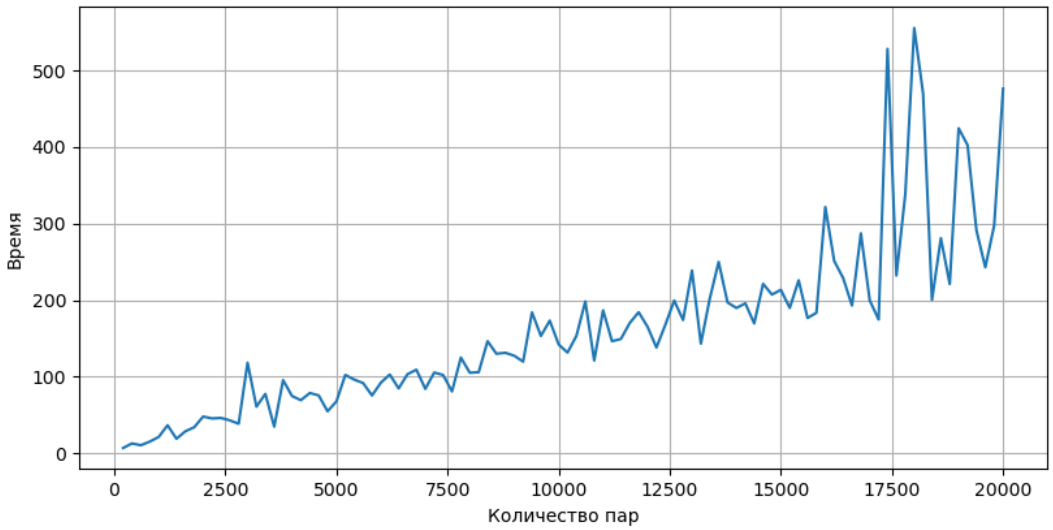
Все попытки, включая эту, были направлены на изменения способа хранения данных в памяти, так как они занимали слишком много места и не укладывались в memory limit. Как уже указал выше, было принято решение хранить не сами экземпляры пар, а указатели на них. Сначала я сделал так для результирующего массива, а в дальнейшем и для входного. Также были замечены странные особенности при смене версии компилятора. Тестирующая система стала показывать WA - неправильный ответ.

Попытка 17-20 (ID 84092810)

В данных попытках я не укладывался в лимит времени. Тестируя и анализируя алгоритм, я не мог понять с чем это связано. В конечном итоге, я пришел к выводу, что это связано с медленным вводом и выводом данных. Поэтому я использовал std::ios::sync_with_stdio(false) - убирающий синхронизацию потоков C и C++, а также std::cin.tie(0), который разделяет потоки ввода и вывода. В итоге программа успешно смогла пройти все тесты.

Тест производительности

Тестирование программы производилось на 100 сгенерированных тестах с шагом в 200 элементов. Значения ключа в диапазоне $[0; 65535]$, а строка имела длинну $[1; 2048]$.



Выводы

При выполнении данной лабораторной работы я познакомился с алгоритмом сортировки подсчётом. Нельзя не отметить, что данный алгоритм очень хорошо применим в тех ситуациях, когда нам заранее известен диапазон наших чисел. Например отсортировать людей по возрасту. Он имеет линейную сложность, что является его главным преимуществом.

Но в то же время данный алгоритм имеет ряд недостатков. В частности если мы имеем диапазон значений, превосходящий количество элементов, то время работы возрастает. Можно сделать вывод, что такой алгоритм идеально подойдет для сортировки элементов, которые мы можем каким-то образом пересчитать - присвоить номер.

При использовании этого алгоритма возникает 3 проблемы: Сортировка сложных объектов, рост времени работы при большом диапазоне, рост используемой памяти при большом диапазоне. Первую проблему конечно можно решить, если, как в задании, присвоить хранящимся данным некоторые ключи. Но если у нас есть много уникальных объектов, с уникальными ключами, то время работы только увелиться и лучше применить сортировки, основанные на сравнениях.