**Министерство образования и науки Российской Федерации**

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ**

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Программная инженерия

Дисциплина «Алгоритмы и структуры данных»

# ОТЧЁТ

по лабораторной работе Жадные алгоритмы (Stepic)

Молодецкий Арсений

группа Р3217

Санкт-Петербург

2019 г.

# Содержание

Задача 1: покрыть отрезки точками ..... 3
Исходный код к задаче 1 ..... 3
Задача 2: непрерывный рюкзак ..... 4
Исходный код к задаче 2 ..... 4
Задача 3: различные слагаемые ..... 5
Исходный код к задаче 3 ..... 5
Задача 4: кодирование Хаффмана ..... 6
Исходный код к задаче 4 ..... 6
Задача 5: декодирование Хаффмана ..... 8
Исходный код к задаче 5 ..... 9
Задача 6: очередь с приоритетами ..... 9
Исходный код к задаче 6 ..... 10

## Задача 1: покрыть отрезки точками

По данным $n$ отрезкам необходимо найти множество точек минимального размера, для которого каждый из отрезков содержит хотя бы одну из точек.

В первой строке дано число $1 \le n \le 100$

отрезков. Каждая из последующих $n$ строк содержит по два числа $0 \le l \le r \le 10^9$, задающих начало и конец отрезка. Выведите оптимальное число $m$ точек и сами $m$

точек. Если таких множеств точек несколько, выведите любое из них.

**Sample Input 1:**

```
3
1 3
2 5
3 6
```

**Sample Output 1:**

```
1
3
```

**Sample Input 2:**

```
4
4 7
1 3
2 5
5 6
```

**Sample Output 2:**

```
2
3 6
```

## Исходный код к задаче 1

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

int main()
{
	unsigned int line_count = 0;
	// Считываем число отрезков
	std::cin >> line_count;
	// Считываем отрезки в формате пар чисел [a,b]
	std::list<std::pair<int, int>> lines;
	for (size_t i = 0; i < line_count; ++i) {
		int a = 0, b = 0;
		std::cin >> a >> b;
		lines.push_back(std::make_pair(a, b));
	}

	// Сортируем отрезки по правому краю
	lines.sort([](const std::pair<int, int> &a, const std::pair<int, int> &b)
				{ return a.second < b.second;});
	// Множество точек
	std::vector<int> points;
	// Пока множество отрезков не пусто
```

```
	while (0 != lines.size()) {
		// Берем первый отрезок (с самым меньшим правым концом)
		// и ставим в этом месте точку
		int p = (*lines.begin()).second;
		points.push_back(p);
		// Удаляем из множества все отрезки перекрытые этой точкой
		while (true) {
			if (lines.size() != 0 && (*lines.begin()).first <= p)
lines.pop_front();
			else break; // Если отрезки закончились или вышли началом за нашу
точку - выходим из цикла
		}
	}

	// Если все отрезки пройдены, выводим результат
	size_t points_count = points.size();
	std::cout << points_count << std::endl;
	for (auto pt : points)
		std::cout << pt << " ";
	std::cout << std::endl;

	return 0;
}
```

## Задача 2: непрерывный рюкзак

Первая строка содержит количество предметов $1 \le n \le 10^3$ и вместимость рюкзака $0 \le W \le 2 \cdot 10^6$. Каждая из следующих $n$ строк задаёт стоимость $0 \le c_i \le 2 \cdot 10^6$ и объём $0 < w_i \le 2 \cdot 10^6$ предмета ($n$, $W$, $c_i$, $w_i$ — целые числа). Выведите максимальную стоимость частей предметов (от каждого предмета можно отделить любую часть, стоимость и объём при этом пропорционально уменьшатся), помещающихся в данный рюкзак, с точностью не менее трёх знаков после запятой.

**Sample Input:**

```
3 50
60 20
100 50
120 30
```

**Sample Output:**

```
180.000
```

## Исходный код к задаче 2

```
#include <vector>
#include <list>
#include <iostream>
#include <algorithm>


int main()
{
	unsigned int items_count = 0;
	unsigned int bag_volume = 0;
	// Считываем число предметов и емкость корзины
	std::cin >> items_count >> bag_volume;
	// Считываем предметы в формате пар чисел [цена предмета, объем предмета]
	std::list<std::pair<int, int>> items;
	for (size_t i = 0; i < items_count; ++i) {
		int cost = 0, volume = 0;
		std::cin >> cost >> volume;
		items.push_back(std::make_pair(cost, volume));
	}
```

```
	// Сортируем предметы по удельной стоимости: цена/объем
	// (в начале предмет с максимальной удельной стоимостью)
	items.sort([](const std::pair<int, int> &a, const std::pair<int, int> &b)
	{ return (((double)a.first) / ((double)a.second)) >(((double)b.first) /
((double)b.second));});
	// Пока есть доступное место в корзине, добавляем в нее первый предмет
	// (с наибольшей удельной стоимостью) либо его часть если весь не влазит
	// в корзину и удаляем его из доступных предметов
	double bag_cost = 0.0;
	while (bag_volume > 0)
	{
		if (items.empty()) break; // Если нет доступных предметов то выходим из
цикла
		unsigned int item_cost = (*items.begin()).first;
		unsigned int item_volume = (*items.begin()).second;
		// Если объем предмета больше доступного места - добавляем его часть
		if (bag_volume < item_volume) {
			bag_cost += ((double)bag_volume) * ((double)item_cost) /
((double)item_volume);
			bag_volume = 0;
		}
		else {
			// Инначе добавляем в корзину предмет полностью и удаляем его из
доступных
			bag_cost += (double)item_cost;
			bag_volume -= item_volume;
			items.pop_front();
		}
	}
	std::cout.setf(std::ios::fixed, std::ios::floatfield);
	std::cout.precision(3);
	std::cout << bag_cost << std::endl;
	return 0;
}
```

## Задача 3: различные слагаемые

По данному числу $1 \le n \le 10^9$ найдите максимальное число $k$, для которого $n$ можно представить как сумму $k$ различных натуральных слагаемых. Выведите в первой строке число $k$, во второй — $k$ слагаемых.

**Sample Input 1:**

```
4
```

**Sample Output 1:**

```
2
1 3
```

**Sample Input 2:**

```
6
```

**Sample Output 2:**

```
3
1 2 3
```

## Исходный код к задаче 3

```
#include <iostream>
#include <cmath>


int main()
```

```
{
	unsigned int number = 0;
	// Считываем число
	std::cin >> number;
	// Находим число k, такое что сумма чисел от 1 до k меньше либо равна нашему
числу number
	double kd = (-1.0 + std::sqrt(1.0 + 8.0 * ((double)number))) / 2.0;
	unsigned int k = (unsigned int)kd; // Отбрасываем дробную часть
	                                                    // Выводим количество
слагаемых
	std::cout << k << std::endl;
	// Выводим натуральные числа от 1 до k-1 и результат выражения от number -
sum(k-1)
	for (unsigned int i = 1; i <= k - 1; ++i) std::cout << i << " ";
	std::cout << (number - (k - 1) * k / 2) << std::endl;

	return 0;
}
```

## Задача 4: кодирование Хаффмана

По данной непустой строке $s$ длины не более $10^4$, состоящей из строчных букв латинского алфавита, постройте оптимальный беспрефиксный код. В первой строке выведите количество различных букв $k$, встречающихся в строке, и размер получившейся закодированной строки. В следующих $k$ строках запишите коды букв в формате "letter: code". В последней строке выведите закодированную строку.

**Sample Input 1:**

```
a
```

**Sample Output 1:**

```
1 1
a: 0
0
```

**Sample Input 2:**

```
abacabad
```

**Sample Output 2:**

```
4 14
a: 0
b: 10
c: 110
d: 111
01001100100111
```

## Исходный код к задаче 4

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <queue>
#include <list>
#include <memory>


struct Node {
	Node(char ch, unsigned int freq, std::shared_ptr<Node> left_node = nullptr,
std::shared_ptr<Node> right_node = nullptr)
```

```
            : frequency(freq), letter(ch), left(left_node), right(right_node) {}

      std::shared_ptr<Node> left;
      std::shared_ptr<Node> right;
      const unsigned int frequency;
      const char letter;

      struct CompareNode {
            bool operator()(const std::shared_ptr<Node> &e1, const
std::shared_ptr<Node> &e2) const
            {
                  return e1->frequency > e2->frequency;
            }
      };
};

void make_code_table(const std::shared_ptr<Node> &node, std::unordered_map<char,
std::string> &map, std::string path = std::string()) {
      // Если мы спустились в лист - добавляем найденный путь к символу
      if (node->letter != 0) {
            map.insert(std::pair<char, std::string>(node->letter, path));
            return; // Выходим из данной ветви рекурсии
      }
      // Иначе спускаемся ниже
      make_code_table(node->left, map, path + "0");
      make_code_table(node->right, map, path + "1");
}

int main()
{
      // Считываем строку символов
      std::string line;
      std::getline(std::cin, line);
      //line = line.substr(0, line.size() - 1);
      // Подсчитываем количество вхождений каждого символа в строке
      std::unordered_map<char, unsigned int> character_map;
      for (auto ch : line) {
            if (character_map.find(ch) != character_map.end())
++character_map.at(ch);
            else character_map.insert(std::pair<char, unsigned int>(ch, 1));
      }
      // Складываем все узлы содержащие пары буква-частота в приоритетную очередь по
частоте вхождения
      std::priority_queue<std::shared_ptr<Node>, std::vector<std::shared_ptr<Node>>,
Node::CompareNode> letters;
      for (auto elem : character_map) {
            auto node = std::make_shared<Node>(Node(elem.first, elem.second));
            letters.push(node);
      }
      // Создаем граф - дерево Хаффмана
      // Пока длина очереди более 1, продолжаем сосдавать дерево
      // объединяя попарно узлы с минимальной частотой
      while (1 < letters.size()) {
            // Изымаем из очереди два узла с минимальной частотой
            auto node1 = std::make_shared<Node>(*letters.top()); letters.pop();
            auto node2 = std::make_shared<Node>(*letters.top()); letters.pop();
            // Создаем новый узел на их основе и помещаем его обратно в очередь
            if (node1->frequency < node2->frequency) {
                  auto new_node = Node(0, node1->frequency + node2->frequency,
node1, node2);
                  auto node = std::make_shared<Node>(new_node);
                  letters.push(node);
            }
            else {
                  auto new_node = Node(0, node1->frequency + node2->frequency,
node2, node1);
                  auto node = std::make_shared<Node>(new_node);
                  letters.push(node);
            }
      }
      // Строим таблицу кодов Хаффмана для каждого символа
      // совершая рекурсивный обход по графу и сохраняя путь к каждому символу:
```

```
	// при спуске на лево - 0, при спуске на право - 1
	std::unordered_map<char, std::string> haffman_map;
	if (character_map.size() == 1) {
		haffman_map.insert(std::pair<char,
std::string>((*character_map.begin()).first, "1"));
	}
	else make_code_table(letters.top(), haffman_map);
	// Создаем закодированную строку
	std::string code_string;
	for (auto ch : line) code_string += haffman_map.at(ch);
	// Выводим количество букв и длинну закодированной строки
	std::cout << haffman_map.size() << " " << code_string.length() << std::endl;
	// Выводим значение кодов для каждого символа
	for (auto elem : haffman_map) std::cout << elem.first << ": " << elem.second <<
std::endl;
	// Выводим закодированную строку
	std::cout << code_string << std::endl;
	return 0;
}
```

## Задача 5: декодирование Хаффмана

Восстановите строку по её коду и беспрефиксному коду символов.

В первой строке входного файла заданы два целых числа $k$ и $l$ через пробел — количество различных букв, встречающихся в строке, и размер получившейся закодированной строки, соответственно. В следующих $k$

строках записаны коды букв в формате "letter: code". Ни один код не является префиксом другого. Буквы могут быть перечислены в любом порядке. В качестве букв могут встречаться лишь строчные буквы латинского алфавита; каждая из этих букв встречается в строке хотя бы один раз. Наконец, в последней строке записана закодированная строка. Исходная строка и коды всех букв непусты. Заданный код таков, что закодированная строка имеет минимальный возможный размер.

В первой строке выходного файла выведите строку $s$. Она должна состоять из строчных букв латинского алфавита. Гарантируется, что длина правильного ответа не превосходит $10^4$ символов.

**Sample Input 1:**

```
1 1
a: 0
0
```

**Sample Output 1:**

```
a
```

**Sample Input 2:**

```
4 14
a: 0
b: 10
c: 110
d: 111
01001100100111
```

**Sample Output 2:**

```
abacabad
```

## Исходный код к задаче 5

```
#include <iostream>
#include <string>
#include <unordered_map>


int main()
{
	// Считываем количество букв и размер закодированной строки
	unsigned int letters_num = 0, code_line_size = 0;
	std::cin >> letters_num >> code_line_size;
	// Считываем буквы с кодами
	std::unordered_map<std::string, char> haffman_map;
	for (unsigned int i = 0; i < letters_num; ++i) {
		std::string code;
		std::getline(std::cin, code);
		if (code.size() < 4) { --i; continue; }
		char ch = code[0];
		std::string cd = code.substr(3, code.size());
		haffman_map.insert(std::pair<std::string, char>(cd, ch));
	}
	// Считываем закодированную строку
	std::string code_line;
	std::getline(std::cin, code_line);
	// Проходим посимвольно по закодированной строке
	// и выводим раскодированные символы
	std::string code;
	for (auto ch : code_line) {
		code += ch;
		if (haffman_map.find(code) != haffman_map.end()) {
			std::cout << haffman_map.at(code);
			code.erase();
		}
	}

	return 0;
}
```

## Задача 6: очередь с приоритетами

Первая строка входа содержит число операций $1 \le n \le 10^5$. Каждая из последующих $n$

строк задают операцию одного из следующих двух типов:

- Insert x, где $0 \le x \le 10^9$ — целое число;
- ExtractMax.

Первая операция добавляет число $x$ в очередь с приоритетами, вторая — извлекает максимальное число и выводит его.

**Sample Input:**

```
6
Insert 200
Insert 10
ExtractMax
Insert 5
Insert 500
ExtractMax
```

**Sample Output:**

```
200
500
```

## Исходный код к задаче 6

```
#include <iostream>
#include <queue>
#include <string>
#include <vector>

template <typename T>
struct max_prior_queue
{
	max_prior_queue()
	{}
	void push(T val) {
		// Добавляем элемент в конец и поднимаем его до его позиции
		data_.push_back(val);
		move_up(data_.size() - 1);
	}

	T top() {
		// Возвращаем элемент с начала, заменяем его на последний элемент
		// и опускаем его до его позиции
		if (data_.size() == 0) return T();
		T max_val = data_.at(0);
		data_.at(0) = data_.at(data_.size() - 1);
		data_.pop_back();
		if (data_.size() == 0) return max_val; // Если нет больше элементов
		move_down(0);
		return max_val;
	}


private:
	std::vector<T> data_;
	void move_up(unsigned int pos) {
		if (pos == 0) return; // Если поднялись на самый верх - завершаем подъем
		unsigned int parent_pos = pos / 2;
		if (data_.at(parent_pos) > data_.at(pos)) return; // Поднялись до
нужного уровня

	  // Иначе меняем местами родительский и текущий узел
		T temp = data_.at(pos);
		data_.at(pos) = data_.at(parent_pos);
		data_.at(parent_pos) = temp;
		// И продолжаем подъем
		move_up(parent_pos);
	}
	void move_down(unsigned int pos) {
		unsigned int child1 = 2 * pos, child2 = 2 * pos + 1;
		if (child1 > (data_.size() - 1)) return; // Если (нет детей) опустились
на самый низ - завершаем спуск
		if (child2 > (data_.size() - 1)) { // Если есть только один узел
			if (data_.at(pos) > data_.at(child1)) return; // Опустились до
нужного уровня

	  // Иначе меняем местами дочерний и текущий узел
			T temp = data_.at(pos);
			data_.at(pos) = data_.at(child1);
			data_.at(child1) = temp;
			// И выходим, т.к. если один дочерний узел - то это последний
элемент
			return;
		}
		else {
			// Если есть оба дочерних узла
			// и если оба узла меньше - то опустились до нужного уровня
			if (data_.at(pos) >= data_.at(child1) && data_.at(pos) >=
data_.at(child2)) return;
			else {
```

```
                        // Выбираем для обмена узел с наибольшим значением
                        unsigned int pos_to_swap = data_.at(child1) >
data_.at(child2) ? child1 : child2;
                        // и обмениваем
                        T temp = data_.at(pos);
                        data_.at(pos) = data_.at(pos_to_swap);
                        data_.at(pos_to_swap) = temp;
                        // И продолжаем спуск
                        move_down(pos_to_swap);
                    }
                }
        }
};

int main()
{
        const std::string insert = "Insert";
        const std::string extract = "ExtractMax";
        // Считываем число операций
        unsigned int instruction_number = 0;
        std::cin >> instruction_number;
        max_prior_queue<int> numbers;
        for (unsigned int i = 0; i < instruction_number; ++i) {
                std::string operation;
                std::cin >> operation;
                if (operation == insert) {
                        unsigned int number = 0;
                        std::cin >> number;
                        numbers.push(number);
                }
                else if (operation == extract) {
                        std::cout << numbers.top() << std::endl;
                }
        }
        return 0;
}
```