**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 «Aлгоритмы и структуры данных»

**ОТЧЁТ**

по лабораторной работе Жадные алгоритмы (Stepic)

Молодецкий Арсений

группа P3217

Санкт-Петербург

2019 г.

Содержание

[Задача 1: покрыть отрезки точками 3](#_Toc8427447)

[Исходный код к задаче 1 3](#_Toc8427448)

[Задача 2: непрерывный рюкзак 4](#_Toc8427449)

[Исходный код к задаче 2 4](#_Toc8427450)

[Задача 3: различные слагаемые 5](#_Toc8427451)

[Исходный код к задаче 3 5](#_Toc8427452)

[Задача 4: кодирование Хаффмана 6](#_Toc8427453)

[Исходный код к задаче 4 6](#_Toc8427454)

[Задача 5: декодирование Хаффмана 8](#_Toc8427455)

[Исходный код к задаче 5 9](#_Toc8427456)

[Задача 6: очередь с приоритетами 9](#_Toc8427457)

[Исходный код к задаче 6 10](#_Toc8427458)

# 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≤*n*≤100

отрезков. Каждая из последующих *n* строк содержит по два числа 0≤*l*≤*r*≤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≤*n*≤103 и вместимость рюкзака 0≤*W*≤2⋅106. Каждая из следующих *n* строк задаёт стоимость 0≤*ci*≤2⋅106 и объём 0<*wi*≤2⋅106 предмета (*n*, *W*, *ci*, *wi* — 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≤*n*≤109 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 104, 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 спустильсь 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 104

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≤*n*≤105. Каждая из последующих *n*

строк задают операцию одного из следующих двух типов:

* Insert x, где 0≤*x*≤109  — 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;

}