#include <iostream>

#include <vector>

#include <list>

#include <algorithm>

using namespace std;

bool pred(pair<int, int> a, pair<int, int> b)

{

return a.second < b.second;

}

int main()

{

int line\_count = 0;

cin >> line\_count;

list<pair<int, int>> lines;

for (size\_t i = 0; i < line\_count; ++i) {

int a = 0, b = 0;

cin >> a >> b;

lines.push\_back(make\_pair(a, b));

}

// Сортируем отрезки по правому краю

lines.sort([](const pair<int, int>& a, const pair<int, int>& b) { return a.second < b.second; });

// Множество точек

vector<int> points;

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 точкойй

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();

cout << points\_count << endl;

for (auto pt : points) cout << pt << " ";

cout << endl;

return 0;

}

#include <iostream>

#include <iomanip>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;

void fill\_section(vector<pair<double, double>>& v, const int n) {

for (int i = 0; i < n; ++i) {

double l, r;

cin >> l >> r;

v.push\_back(make\_pair(l, r));

}

}

bool weight\_cmp(const pair<double, double>& a, const pair<double, double>& b) {

return (a.first / a.second) > (b.first / b.second);

}

double calc\_cost(const pair<double, double>& c\_w) {

return c\_w.first / c\_w.second;

}

double fill\_knapsack(vector<pair<double, double>>& kn, int w) {

double sum\_cost = 0;

sort(kn.begin(), kn.end(), weight\_cmp);

for (int i = 0; i < kn.size() && w > 0; ++i) {

auto item\_weight = kn[i].second;

auto item\_cost = kn[i].first;

if (w >= item\_weight) {

sum\_cost += item\_cost;

w -= item\_weight;

}

else {

sum\_cost += calc\_cost(kn[i]) \* w;

w = 0;

}

}

return sum\_cost;

}

int main() {

int n, w;

vector<pair<double, double>> v;

cin >> n >> w;

fill\_section(v, n);

auto sum = fill\_knapsack(v, w);

cout << fixed << setprecision(3) << sum;

return 0;

}

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector <int> vec;

int n; cin >> n;

int temp = n, number = 1;

while (temp > 0)

{

if (n == 1) { //isklychenie

vec.push\_back(1);

break;

}

vec.push\_back(number); //zapisvaem number poka ne doidem do poslednego chisla

temp -= ++number;

if (temp - (number + 1) < 0) { // schitaem poslednee chislo

temp = n;

number--;

int number\_mem = number;

while (number > 0)

{

temp -= number;

number--;

}

if (temp == number\_mem) { //esli poslednee chislo ravno predposlednemy

//cout << temp + number\_mem;

vec[vec.size() - 1] = temp + number\_mem;

break;

}

//cout << temp << endl;

vec.push\_back(temp);

break;

}

}

cout << vec.size() << endl;

for (int i = 0; i < vec.size(); i++) cout << vec[i] << " ";

vec.clear();

cout << endl;

}

#include <algorithm>

#include <cassert>

#include <cstddef>

#include <iostream>

#include <string>

#include <tuple>

#include <unordered\_map>

#include <vector>

#include <queue>

class Huffman {

struct CharSetFrequency {

std::string char\_set;

int frequency;

bool operator < (const CharSetFrequency& other) const {

return std::tie(frequency, char\_set) > std::tie(other.frequency, other.char\_set);

}

};

public:

static std::unordered\_map<char, std::string> encode(const std::string& text);

static std::string decode(const std::string& text, const std::unordered\_map<char, std::string>& huffman\_encoding);

};

std::unordered\_map<char, std::string> Huffman::encode(const std::string& text) {

std::unordered\_map<char, int> char\_frequencies;

for (auto c : text) {

char\_frequencies[c]++;

}

std::vector<CharSetFrequency> frequencies;

for (auto char\_frequency : char\_frequencies) {

frequencies.push\_back({ std::string(1,char\_frequency.first), char\_frequency.second });

}

if (frequencies.size() == 1) {

std::unordered\_map<char, std::string> result;

result[frequencies[0].char\_set[0]] = "0";

return result;

}

std::unordered\_map<char, std::string> result;

std::priority\_queue<CharSetFrequency> q(frequencies.begin(), frequencies.end());

while (q.size() >= 2) {

auto first = q.top();

q.pop();

auto second = q.top();

q.pop();

for (auto c : first.char\_set) {

result[c] = "0" + result[c];

}

for (auto c : second.char\_set) {

result[c] = "1" + result[c];

}

q.push({ first.char\_set + second.char\_set, first.frequency + second.frequency });

}

return result;

}

std::string Huffman::decode(const std::string& text, const std::unordered\_map<char, std::string>& huffman\_encoding) {

size\_t len = text.size();

size\_t pos = 0;

std::string result;

while (pos < len) {

for (auto& encoded : huffman\_encoding) {

if (text.substr(pos, encoded.second.size()) == encoded.second) {

result += encoded.first;

pos += encoded.second.size();

break;

}

}

}

return result;

}

int main() {

std::string text;

std::cin >> text;

auto huffman\_encoding = Huffman::encode(text);

std::string encoded\_text;

for (auto c : text) {

encoded\_text += huffman\_encoding[c];

}

std::cout << huffman\_encoding.size() << " " << encoded\_text.size() << std::endl;

for (auto& encoded : huffman\_encoding) {

std::cout << encoded.first << ": " << encoded.second << std::endl;

}

std::cout << encoded\_text << std::endl;

}