

#include <iostream>

using namespace std;

class HashTable {

public:

HashTable();

~HashTable();

HashTable( const HashTable& other );

HashTable& operator=( const HashTable& other) //перегрузка оператора присваивания

{

HashTable temp(other);

swap (this->tableSize, temp.tableSize);

swap (this->data, temp.data);

return \*this;

}

void add(const string element);

void Hash(int64\_t &hash\_1, int64\_t &hash\_2, const string &str);

int tableSize;

int count;

void write();

bool remove(const string element);

int64\_t search(const string str);

private:

void expand();

struct Element {

Element(const string element = "");

string element;

bool deleted;

};

Element \*\*data;

};

HashTable::Element::Element(const string my\_string) :

element(my\_string), deleted(false) {}

HashTable::HashTable() {

tableSize = 8;

count = 1;

data = new Element \*[tableSize];

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

data[i] = NULL;

}

}

HashTable::HashTable( const HashTable& other )

{ tableSize = other.tableSize;

data = new Element \*[tableSize];

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

if (other.data[i]!=NULL){

if (other.data[i]->element != ""){

data[i] = new Element (\*other.data[i]);

}

}

}

}

//HashTable

HashTable::~HashTable() {

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

if (data[i] != NULL)

delete data[i];

}

delete[] data;

}

void HashTable::expand() {

tableSize = tableSize \* 2;

Element \*\*newnodes = data;

data = new Element \*[tableSize];

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

data[i] = NULL;

}

count = 0;

for (int i = 0; i < tableSize / 2; i++) {

if (newnodes[i] != NULL) {

if (newnodes[i]->deleted == false) {

add(newnodes[i]->element);

}

delete newnodes[i];

}

}

delete[] newnodes;

}

void HashTable::Hash(int64\_t &hash\_1, int64\_t &hash\_2, const string &str) {

for (int i = 0; i < str.length(); ++i) {

hash\_1 = (hash\_1 \* 131 + str[i]) % tableSize;

hash\_2 = (hash\_2 \* 149 + str[i]) % tableSize;

}

hash\_2 = (hash\_2 \* 2 + 1) % tableSize;

}

void HashTable::add(const string element) {

if (count >= 0.75 \* tableSize) {

expand();

}

int64\_t hash\_1 = 0, hash\_2 = 0;

Hash(hash\_1, hash\_2,element);

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

if (data[hash\_1] == NULL) {

count++;

data[hash\_1] = new Element(element);

return;

} else if (data[hash\_1]->deleted) {

count++;

data[hash\_1]->deleted = false;

data[hash\_1]->element = element;

return;

}

//или он удален, то есть него можно записывать

//cout<<"element "<<element;

// cout<<"index "<<hash\_1<<endl;

hash\_1 = (hash\_1 + hash\_2) % tableSize;

}

}

int64\_t HashTable::search(const string element) {

int64\_t hash\_1 = 0, hash\_2 = 0;

int i = 0;

Hash(hash\_1, hash\_2, element);

while (i < tableSize && data[hash\_1] != NULL) {

if (!data[hash\_1]->deleted && data[hash\_1]->element == element) {

return hash\_1;

}

i++;

hash\_1 = (hash\_1 + hash\_2) % tableSize;

}

return -1;

}

bool HashTable::remove(const string element) {

int index = search(element);

if (index == -1) {

return false;

//cout << "FAIL";

} else {

data[index]->deleted = true;

count--;

return true;

//cout << "OK";

}

}

int main() {

std::ios\_base::sync\_with\_stdio(false);

std::cin.tie(NULL);

HashTable myhash;

string s;

char operation;

while (cin >> operation >> s) {

if (operation == '+') {

if (myhash.search(s) == -1) {

cout<<"OK"<<endl;

myhash.add(s);

} else {

cout<<"FAIL"<<endl;

}

} else if (operation == '-') {

if (myhash.remove(s)){

cout<<"OK"<<endl;

}

else{

cout<<"FAIL"<<endl;

}

} else if (operation == '?') {

if (myhash.search(s) == -1) cout << "FAIL"<<endl;

else

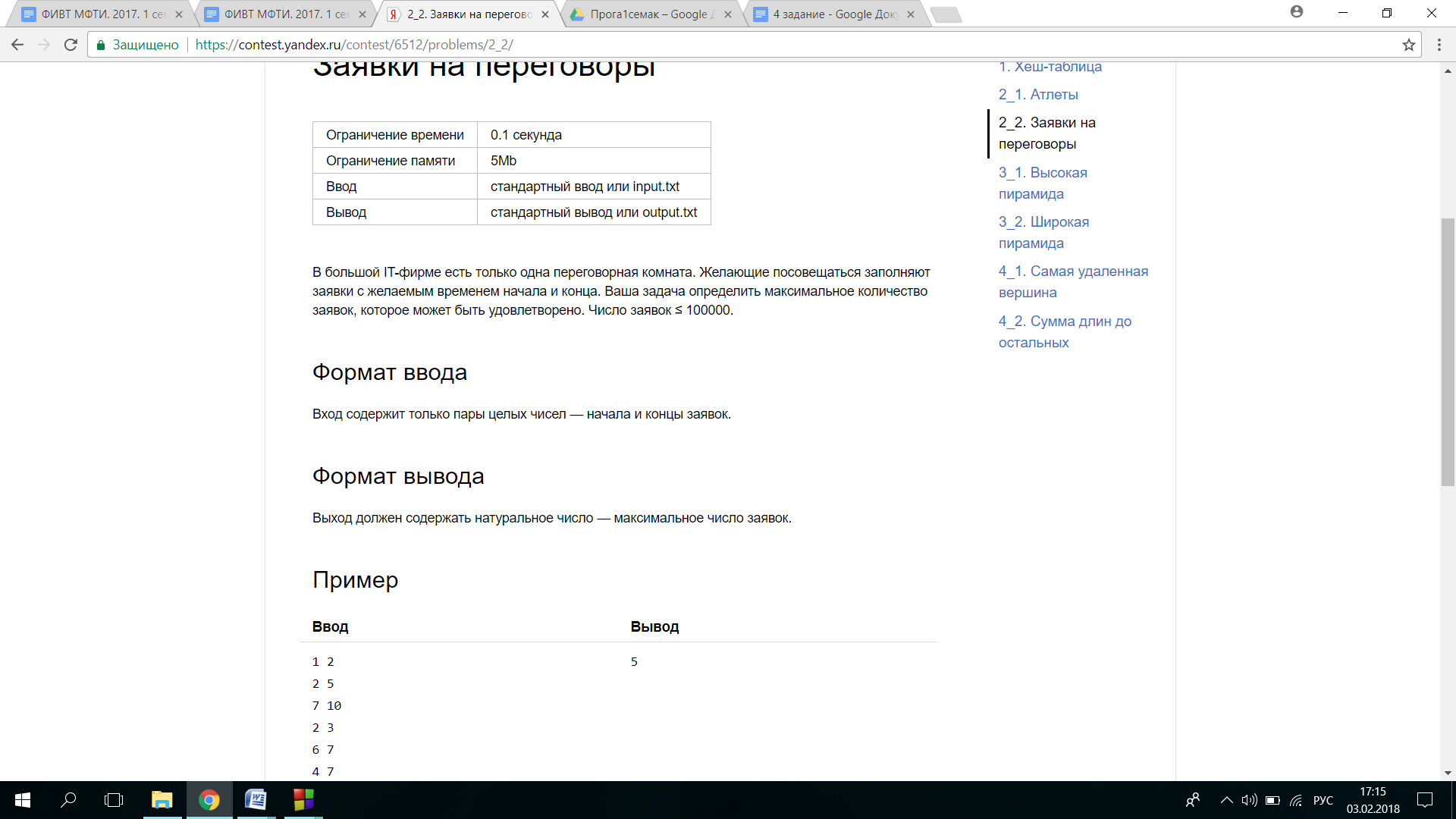
cout << "OK"<<endl;

}

}

return 0;

}

#include <stack>

#include <iostream>

#include <vector>

#include <cstring>

#include <cstdio>

using namespace std;

struct Segment {

Segment(int NEWRIGHT = 0, int left = 0) : right(NEWRIGHT), left(left) {}

bool operator>(const Segment &rhs) const {

if (left != rhs.left) {

return left > rhs.left;

} else return right > rhs.right;

}

int right;

int left;

};

void Write(const vector<Segment> Time) {

int i = 0;

for (int i = 0; i < Time.size(); i++) {

cout << i << " " << Time[i].right << " " << Time[i].left << endl;

}

}

template<class Iterator>

Iterator Partition(Iterator begin, Iterator end) {

if (end <= begin + 1) {

return begin;

}

const Segment &pivot = \*(end - 1);

Iterator i = begin;

Iterator j = i;

while (j < end - 1) {

if (\*(j) > pivot) {

j++;

} else {

swap(\*(i), \*(j));

i++;

j++;

}

}

swap(\*(i), \*(end - 1));

return i;

}

template<class Iterator>

void QuickSort(Iterator begin, Iterator l, Iterator r) {

stack<Iterator> s;

s.push(r);

s.push(l);

while (!s.empty()) {

l = s.top();

s.pop();

r = s.top();

s.pop();

if (r <= l) continue;

Iterator i = Partition(l, r);

if (i - begin - 1 > r - i) {

s.push(i);

s.push(l);

s.push(r);

s.push(i + 1);

} else {

s.push(r);

s.push(i + 1);

s.push(i);

s.push(l);

}

}

}

void push(vector<Segment> &Time) {

int i = 0;

int left, right;

while (cin>>left>>right) {

Time.push\_back(Segment(left, right));

}

}

int solution(const vector<Segment> &Time) {

int i = 0;

int last = Time[i].left;

int count = 1;

for (int i = 1; i < Time.size(); i++) {

if (Time[i].right >= last) {

count++;

last = Time[i].left;

}

}

return count;

}

int main() {

vector<Segment> Time;

push(Time);

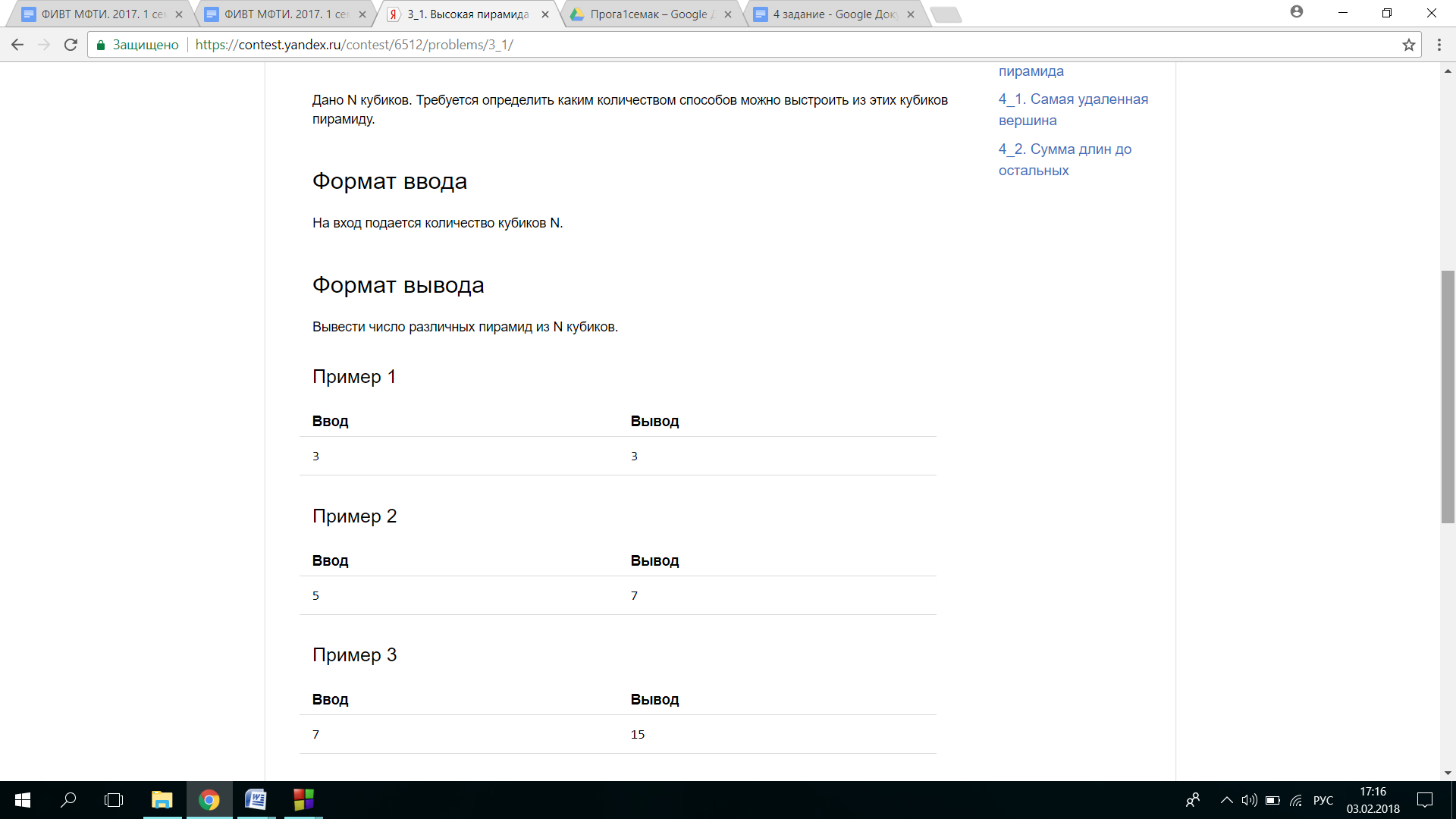
QuickSort(Time.begin(), Time.begin(), Time.end());//сортируем по возрастанию y, а потом по возрастанию x.

//Это дает нам возмодность гарантировать,что внутри выбранного отрещка нет других, которые бы закончились раньше его.

cout << solution(Time);

return 0;

}

#include <iostream>

#include <vector>

using namespace std;

long long int Count(int n){

vector <vector <long long int> > Cubecount( n+1, vector <long long int> (n+1));

Cubecount[1][1] = 1;

for (long long int i = 2; i <= n; i++) { //идем по количеству кубиков, будем считать сколько можно построить из 2 кубиков, ..

//...из трех кубиков и т д до нужного нам

for (long long int j = 1; j <= i; j++) {//

long long int nextindex = 1;//сколько кубиков можно положить в следующий слой!!именно в слудующий слой а не вообще оставшихся кубиков

Cubecount[i][j] = 0;

if (i == j){

Cubecount[i][j] = 1;//это тот случай, когда в следующий слой 0 кубиков, то есть два слоя равны

}

while (nextindex <= i - j && nextindex <= j) {//следующий слой должен быть не шире основания и состоять из числа кубиков, сколько у нас есть

Cubecount[i][j] = Cubecount[i][j] + Cubecount[i - j][nextindex];//+ количество способов сделать пирамидку из i-j кубика, в основании p

nextindex++;

}

}

}

long long int count = 0;

for (long long int i = 1; i <= n; i++) {

count = count + Cubecount[n][i];

}

return count;

}

int main() {

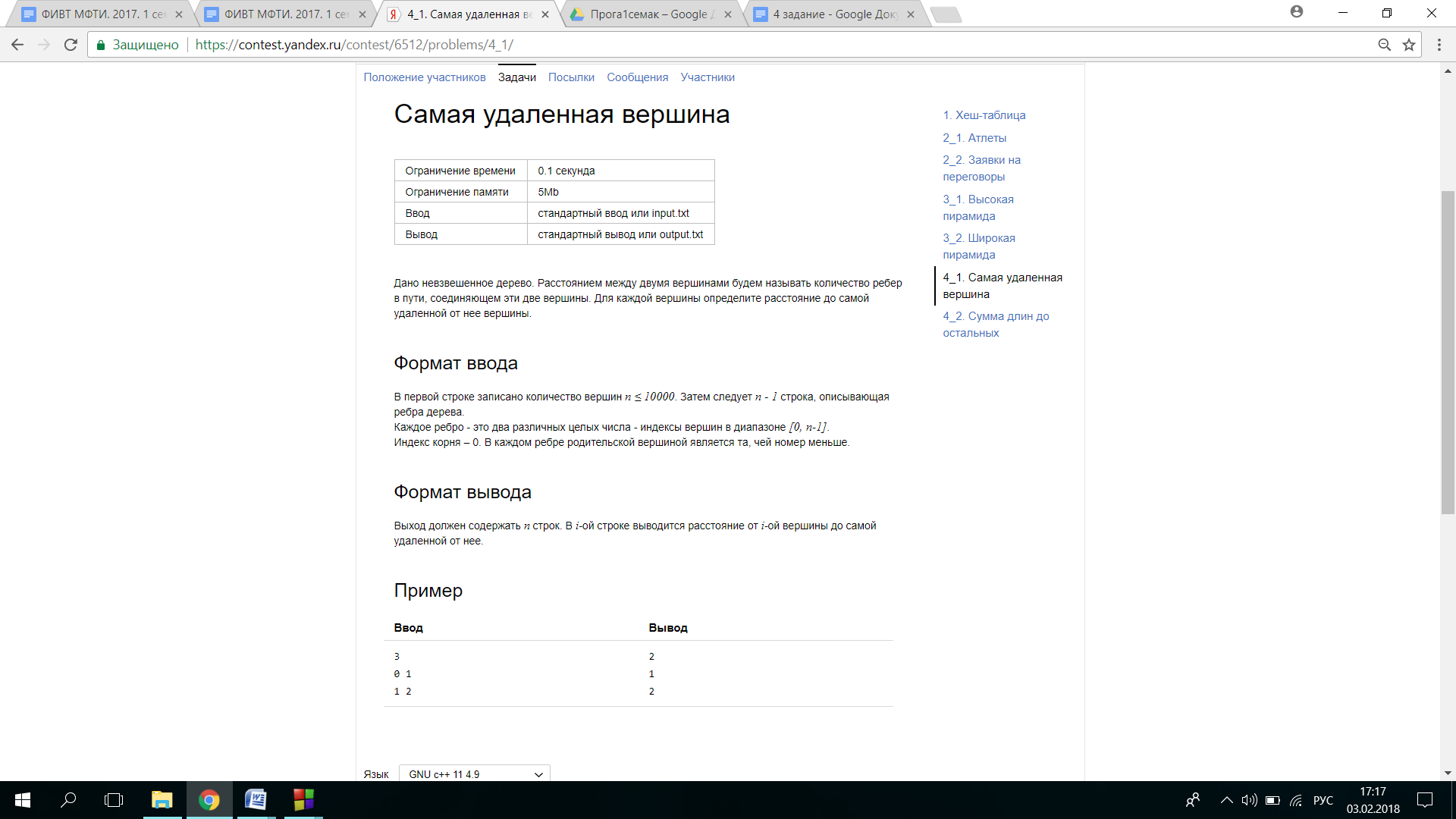
long long int n;

cin >> n;

cout<<Count(n);

return 0;

}

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

public:

Graph(int n);

struct Node {

vector<Node \*> children;

int i;

};

void add(int a, int b);

void bfs(Node \*node, vector<int> &count, Node \*&Di, vector<int> &countfinal, int index);

void solution(vector<int> &count);

vector<Node> nodes;

};

Graph::Graph(int n) :

nodes(n) {

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

nodes[i].i = i;

}

}

void Graph::add(int vertex1, int vertex2) {

nodes[vertex2].children.push\_back(&nodes[vertex1]);

nodes[vertex1].children.push\_back(&nodes[vertex2]);

}

void Graph::bfs(Node \*node, vector<int> &count, Node \*&Diameter, vector<int> &countfinal, int index) {

queue<Node \*> myqueue;

count[node->i] = 0;

vector<bool> was(count.size());

was[node->i] = true;

myqueue.push(node);

int datediam = 0;

while (!myqueue.empty()) {

Node \*vertex = myqueue.front();

myqueue.pop();

for (int i = 0; i < vertex->children.size(); i++) {

if (vertex->children[i] != NULL && was[vertex->children[i]->i] == false) {

myqueue.push(vertex->children[i]);

was[vertex->children[i]->i] = true;

count[vertex->children[i]->i] = count[vertex->i] + 1;

if (index == -1) {

if (count[vertex->children[i]->i] > datediam) {

datediam = count[vertex->children[i]->i];

Diameter = vertex->children[i];

}

}

}

}

}

if (index != -1) {

for (int i = 0; i < count.size(); i++) {

count[i] = max(count[i], countfinal[i]);

}

}

}

void Graph::solution(vector<int> &count) {

Node \*Diameter;

Node \*Diameter2;

vector<int> countfinal(count.size());

int index = -1;

bfs(&nodes[0], count, Diameter, countfinal, index);

for (int i = 0; i < count.size(); i++) {

count[i] = 0;

}

bfs(Diameter, count, Diameter2, countfinal, index);

index = 1;

bfs(Diameter2, countfinal, Diameter2, count, index);

for (int i = 0; i < countfinal.size(); i++) {

cout << countfinal[i] << endl;

}

}

int main() {

int n;

cin >> n;

vector<int> count(n);

int vertex1, vertex2;

Graph tree(n);

for (int i = 0; i < n - 1; i++) {

cin >> vertex1 >> vertex2;

tree.add(vertex1, vertex2);

}

tree.solution(count);

return 0;

}