Algolab
HS 2012

Pascal Spoerri
pascal@spoerri.io

January 23, 2013

Contents

Part I

Week 1 - Playground

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2011
Exercise Sheet 1

Algorithms Lab

Exercise 0 – *Count the faces*

A *planar graph* is a graph with an embedding in to the plane such that no two edges cross in the embedding. Figure 1 shows a plane embedding of K_4, with the vertices numbered 1–4.

A *face* of the planar graph is a region of the plane which is bounded by edges. In the figure, the faces are labelled with letters.

Your task is to compute the number of faces, given the graph. The number of faces only depends on the graph, not on the exact embedding.

Input The input is a series of test cases, terminated by EOF.

Each test case starts with the number of vertices $n \leq 10000$ and edges $m \leq 10000$ on a line separated by space. Then there are m lines, one for each edge, consisting of two integers u, v which denote the endpoint of the edge. The vertices are numbered from 1 to n, i.e., $1 \leq u, v \leq n$.

Output For each test case, print a line containing the integer f, which is the number of faces in a planar embedding of the graph.

Sample Input

```
2 1
1 2
4 6
1 2
2 3
1 3
1 4
2 4
3 4
```

Sample Output

```
1
4
```

(100 Points)

```cpp
#include <iostream>
#include <vector>
#include <cassert>
#include <algorithm>

using namespace std;
void dfs(vector<int>& vertices, vector<vector<int> >& edges, int v, int component) {
    if (vertices[v] > 0)
        return;
    vertices[v] = component;
    for (vector<int >::iterator it=edges[v].begin(); it != edges[v].end(); it++) {
        dfs(vertices, edges, (*it), component);
    }

}
void add_edge(vector<vector<int> >& edges, int i, int j) {
    if (find(edges[i].begin(), edges[i].end(), j) != edges[i].end())
        return;
    edges[i].push_back(j);
    edges[j].push_back(i);
}

int compute_faces(int v, int e) {
    vector<int> vertices(v, 0);
    vector<vector<int> > edges(v, vector<int>(0));
    int i, j;
    for (int k=0; k<e; k++) {
        cin >> i;
        cin >> j;
        i--;
        j--;
//        edges[i][j] = 1;
//        edges[j][i] = 1;
        add_edge(edges, i,j);
    }
    int components = 0;
    for (unsigned int k=0; k< vertices.size(); k++) {
        if (vertices[k] == 0) {
            components++;
            dfs(vertices, edges, k, components);
        }
    }
    // V - E + F - C = 1
    int f = 1 - v + e + components;
    return f;
}

int main() {
    int n;
    while (cin >> n) {
        int vertices = n;
        int edges;
        cin >> edges;
        cout << compute_faces(vertices, edges) << "\n";
    }
    return 0;
}
```

week_1_playground/00/faces.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2011
Exercise Sheet ACM Week I

Algorithms Lab

Exercise 1 – *Build The Sum*

Build the sum of the numbers you'll get.

Input The first line of the input file will contain an integer giving the number of test cases that follow.

Each test case starts with a line containing m ($0 \leq m \leq 10$), the number of numbers you have to sum up. The following m lines contain m floating point numbers you'll have to read and sum up.

Output For each test case write the sum you've calculated in a single line. Your output will be accepted if it differs from the correct result by at most 10^{-6} in terms of either absolute or relative precision.

Sample Input

```
2
1
5.5
3
1.2
2.3
3.4
```

Sample Output

```
5.5
6.9
```

```cpp
#include <iostream>
using namespace std;
int main() {
    int numTestcases;

    cin >> numTestcases;

    for (int i=0; i < numTestcases; i++) {
        int numValues;
        cin >> numValues;
        double sum = 0, summand;
        for (int j=0; j < numValues; j++) {
            cin >> summand;
            sum += summand;
        }
        cout << sum << "\n";
    }
    return 0;
}
```

week_1_playground/01/buildthesum.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2011
Exercise Sheet ACM Week I

Algorithms Lab

Exercise 2 – *Shelves*

Description The computer science library is in the process of being moved to a new building. As an intern in the library it's your task to help the head librarian with the planning. From the floor plan it's pretty clear that some walls should be filled with as much shelf-space as possible. However, it's not entirely clear just how much of the wall you can optimally cover. The library can order two kinds of shelves which have different lengths. The larger of the two is slightly cheaper per unit of length, so you prefer them to the smaller ones. However cost is secondary and as a very first priority you wish to minimize the empty space on the wall. You think to yourself that this optimization problem should easily be solved with a clever computer program.

Input The first line of the input contains the number $N \leq 1000$ of test cases. Then N lines follow. Each line is a test-case and contains 3 strictly positive integers ℓ, m and n. The number $\ell < 2^{31}$ denotes the length of the wall, while m and n denote the length of the two shelf types, $m \leq n < 2^{31}$.

Output For each test case you should output the optimal number of shelves of length m and n such that the sum of their lengths is at most ℓ and as close to ℓ as possible. If there are multiple solutions you should prefer the one which uses the largest amount of shelves of length n.

For each test case output one line containing 3 integers separated by a single space. The first two integers are the number of shelves of length m and n, the third is the amount of wall that remains uncovered in an optimal solution.

Sample input

```
3
24 3 5
29 3 9
42 4 5
```

Sample output

```
3 3 0
0 3 2
3 6 0
```

```cpp
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;

pair<int, int>* db;
vector<int> visited;

int gcd(int a, int b) {
    if (b==0) {
        return a;
    }
    return gcd(b, a % b);
}

pair<int, int> shelves(unsigned int wall, unsigned int n, unsigned int m){
    assert(wall > 0);
    assert(n <= m);
    pair<unsigned int,unsigned int> result(0,0);
    if (wall < n)
        return result;
//    if (wall == n) {
//        if (n==m)
//            result.second = 1;
//        else
//            result.first = 1;
//        return result;
//    }
//    if (wall < m) {
//        result.first = 1;
//        return result;
//    }
    unsigned int i = 0;
    unsigned int j = (wall/m);
    unsigned int bestValue = j*m;
    pair<unsigned int, unsigned int> best(i,j);
    for (i=0; i*n <=wall; i++) {
        unsigned int remainder = wall - i*n;
        j = remainder/m;

        unsigned int val = n*i+m*j;
        //cout << i << " " << remainder << " " << val << " " << wall << "\n";

        if (val == wall) {
            bestValue = val;
            best = make_pair(i,j);
            break;
        }
        if (val > bestValue) {
            bestValue = val;
            best = make_pair(i,j);
        }
        if (i > m)
            break;
    }

    result.first = best.first;
    result.second = best.second;
    return result;
}

int main() {

    int testcases;
    unsigned int length, n, m;
    cin >> testcases;
    for (int i=0;i<testcases;i++) {
        cin >> length;
        cin >> n;
        cin >> m;
        pair<unsigned int, unsigned int> result = shelves(length, n, m);
        cout << result.first << " " << result.second << " " << (length-result.first*n-result.second*
            m) << "\n";
    }
```

```
    return 0;
}
```

./week_1_playground/02/shelves.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2011
Exercise Sheet ACM Week I

Algorithms Lab

Exercise 3 – *Checking Change*

Description You are working for company producing vending machines and you are investigating some problem reports from your customers. In certain countries the machines seem to give change using too many coins instead of the minimum possible number. In other countries on the other hand it seems that the machine is unable to hand out the correct change in some cases, and always gives back slightly more than what is necessary. This seems strange to you, so you try to find out for any given set of coins that your machine accepts and some common return values what the optimal number of coins should be.

Input The first line of the input contains n, the number of currencies you wish to test. Each test case starts with one line containing two numbers c_i and m_i, $1 \leq i \leq n$. The integer c_i, $1 \leq c_i \leq 100$ is the amount of different coin-values that the machine accepts in that currency, and m_i is the number of sample return values you would like to test. The next line contains c_i different integers, sorted in ascending order, which represent the coin values for the currency. Then m_i lines follow, each containing an integer t in the range $1 \leq t < 10000$ representing the change your machine needs to hand out. These numbers are in the same unit as the given coin values.

Output For each test case i you should output m_i lines, each with the minimum number of coins necessary to hand out the given change. If there is no possible way of constructing the change value you should output "not possible".

Sample input

```
3
3 1
1 2 5
17
1 1
3
2
2 3
2 3
3
5
1
```

Sample output

```
4
not possible
1
2
not possible
```

```cpp
#include <iostream>
#include <vector>
#include <cassert>
#include <climits>
using namespace std;
#define MAXCOINS 101
#define MAXCURRENCY 10001

vector<int> visited(MAXCURRENCY+1, 0);
vector<int> possible(MAXCURRENCY+1, 0);
vector<vector<int> > db(MAXCURRENCY+1, vector<int>(MAXCOINS, 0));

int countCoins(vector<int>& coins, int value) {
    int r = 0;
    assert(value >= 0);
    assert(db[value].size() >= coins.size());
    for (unsigned int i=0; i<coins.size();i++) {
        r+= db[value][i];
    }
    return r;
}

int checkValue(vector<int>& coins, int value) {
    int r = 0;
    assert(value >= 0);
    assert(db[value].size() >= coins.size());
    for (unsigned int i=0; i<coins.size();i++) {
        r+= db[value][i]*coins[i];
    }
    return r;
}
// Broken

void recursion(vector<int>& coins, int value) {
    if (value <= 0)
        return;
    if (visited[value])
        return;
    visited[value] = 1;
    if (value < coins[0]) {
        possible[value] = 0;
        return;
    }
    if (value <= coins.back()) {
        for (unsigned int i=0; i<coins.size(); i++) {
            if (value == coins[i]) {
                possible[value] = 1;
                fill(db[value].begin(), db[value].end(), 0);
                db[value][i] = 1;
                return;
            }
        }
//        possible[value] = 0;
//        return;
    }

    int min = -1;
    int minCoins = INT_MAX;
    // Search for minimum
    for (unsigned int i=0; i < coins.size(); i++) {
        int checkVal = value-coins[i];
        if (checkVal < 0)
            continue;
        recursion(coins, checkVal);
        if (possible[checkVal]) {
            int numCoins = countCoins(coins, checkVal);
            if (numCoins < minCoins) {
                min = i;
                minCoins = numCoins;
            }
        }
    }

    if (min == -1) {
```

```
75          possible[value] = 0;
            return;
77      }
        assert(min >= 0);
79      // Update
//      for (unsigned int i=0; i < coins.size(); i++) {
81  //          copy(db[value].begin(), db[value].end(), db[value-coins[min]].begin());
//          db[value][i] += db[value][i];
83  //      }
        possible[value] = 1;
85      copy( db[value-coins[min]].begin(), db[value-coins[min]].end(), db[value].begin());
        db[value][min] += 1;
87  }

89  void test(vector<int>& coins, int value) {

91      recursion(coins, value);
        int numCoins = 0;
93      for (unsigned int i=0; i<coins.size();i++) {
            numCoins += db[value][i];
95      }
        if (!possible[value])
97          cout << "not possible\n";
        else
99          cout << numCoins << "\n";
    }
101
    void testcase() {
103     int nCoins;
        int nTestValues;
105     cin >> nCoins;
        cin >> nTestValues;
107
        vector<int> coins(nCoins);
109     // Read coins
        for (int i=0; i<nCoins;i++)
111         cin >> coins[i];
        // Read and process test values
113     int testval;
        fill(visited.begin(), visited.end(), 0);
115     possible[0] = 1;
        for (int i=0; i<nTestValues;i++) {
117         //cin >> testvalues[i];
            cin >> testval;
119         test(coins, testval);
        }
121 }

123 int main() {
        int nTestcases;
125     cin >> nTestcases;

127     for (int i=0; i < nTestcases; i++) {
            testcase();
129     }
    }
```

./week_1_playground/03/checkingchange.cpp

Part II

Week 2 - Playground - Part 1

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week II

Algorithms Lab

Exercise 1 – *False Coin*

The "Gold Bar" bank received information from reliable sources that in their last group of N coins exactly one coin is false and differs in weight from other coins (while all other coins are equal in weight). After the economic crisis they have only a simple balance available. Using this balance, one can only determine if the weight of objects in the left pan is less than, greater than, or equal to the weight of objects in the right pan.

In order to detect the false coin the bank employees numbered all coins by the integers from 1 to N, thus assigning each coin a unique integer identifier. After that they carried out various weighings, where in each weighing they placed *the same number of coins* in both pans. (Needless to say, a single coin can not appear in both pans at the same time.) The identifiers of coins and the results of the weighings were carefully recorded. You are to write a program that will help the bank employees to determine the identifier of the false coin using the results of these weighings.

Input The first line of the input is an integer M denoting the number of datasets. There is a blank line between datasets, as well as between the first line of the input and the first data set.

The first line of each dataset contains two integers N and K, separated by spaces, where N is the number of coins ($1 \leq N \leq 100$) and K is the number of weighings carried out ($1 \leq K \leq 100$). The following $2K$ lines describe all weighings. Each weighing is described by two consecutive lines. The first of these starts with a number P_i (where $1 \leq P_i \leq N/2$), representing the number of coins placed in each pan, followed by the P_i identifiers of the coins placed in the left pan, and the P_i identifiers of the coins placed in the right pan. All numbers are separated by spaces. The second line contains one character describing the outcome of the weighing: '<', '>', or '=' depending on whether the total weight of the coins in the left pan is less than, greater than, or equal to the total weight of the coins in the right pan.

Output For each dataset, output the identifier of the false coin in a line of its own. If the given weighings do not determine the false coin uniquely, output a line containing '0'.

Sample Input

```
1

5 3
2 1 2 3 4
<
1 1 4
=
1 2 5
=
```

Sample Output

```
3
```

(Points)*100

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <cassert>
#include <algorithm>

using namespace std;

set<int> unknown;
set<int> unequal;
set<int> eq;
bool valfind(int v, set<int>& vec) {
    if (find(vec.begin(), vec.end(), v) == vec.end())
        return false;
    return true;
}

void check(int c, vector<int>& values) {
    for (unsigned int i=0; i<values.size(); i++) {
        if ((i+1) == values.size()/2)
            c *= -1;
        int value = values[i];
        if (c == 0) {
            if (valfind(value, unequal)) {
                // Remove from unequal and add it to equal
                unequal.erase(value);
            }
            eq.insert(eq.end(), value);
        } else {
            if (!valfind(value, eq))
                unequal.insert(unequal.end(), value);
        }
        unknown.erase(value);
    }
}

void testcase() {
    int coins;
    int weightings;
    cin >> coins;
    cin >> weightings;
    unequal.clear();
    unknown.clear();
    eq.clear();
    for (int i=1;i<=coins;i++) {
        unknown.insert(unknown.end(),i);
    }
    for (int i=0; i<weightings; i++) {
        int values;
        cin >> values;
        vector<int> input(values*2);
        for (int i=0; i<values*2; i++) {
            cin >> input[i];
        }
        char c;
        cin >> c;
        int wcase = 0; // 0  => "="
                       // <0 => "<"
                       // >0 => ">"
        if (c == '<')
            wcase = -1;
        else if (c == '>')
            wcase = 1;
        else
            wcase = 0;

        check(wcase, input);
    }
}

int main() {
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases;i++) {
```

```
        testcase();
        if (unequal.size() == 1) {
            cout << *unequal.begin();
        }
        else if (unequal.size() == 0 && unknown.size() == 1)
            cout << *unknown.begin();
        else
            cout << 0;

        cout << "\n";

    }
    return 0;
}
```

./week_2_fundamental/00/coins.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week II

Algorithms Lab

Exercise 2 – *Race Tracks*

Many boring math classes have been spent playing Race Tracks, where two players have to maneuver their cars on a race track drawn on a piece of paper, while their cars can only accelerate by a limited (positive or negative) amount per move.

A variant of Race Tracks involves Hoppers. Hoppers are people on a jump stick who can jump from one square to another, without touching the squares in between (a bit like a knight in chess). Just like the aforementioned cars, they can pick up speed and make bigger hops, but their acceleration per move is limited, and they also have a maximum speed.

Let's be a bit more formal: our variant of Race Tracks is played on a rectangular grid, where each square on the grid is either empty or occupied. While hoppers can fly over any square, they can only land on empty squares. At any point in time, a hopper has a velocity (x, y), where x and y are the speed (in squares) parallel to the grid in x- and y-direction respectively . Thus, a speed of $(2, 1)$ corresponds to a knight jump (as does $(-2, 1)$ and 6 other speeds).

Whenever a hopper stands on a square he can change its speed in each direction separately by -1, 0, or 1. Thus, while having speed $(2, 1)$, the hopper can change to speeds $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, $(2, 1)$, $(2, 2)$, $(3, 0)$, $(3, 1)$ and $(3, 2)$. It is impossible for the hopper to obtain a velocity of 4 in either direction, so the x and y component will **always** stay between -3 and 3 inclusive.

The goal of Hopping Race Tracks is to get from start to finish as quickly as possible (i.e. in the least number of hops), without landing on occupied squares. You are to write a program which, given a rectangular grid, a start point S, and a finish point F, determines the least number of hops in which you can get from S to F. The hopper starts with initial speed $(0, 0)$ and he does not care about his speed when he arrives at F.

Input The first line contains the number of test cases N your program has to process. Each test case consists of a first line containing the width X (where $1 \leq X \leq 30$) and height Y (where $1 \leq Y \leq 30$) of the grid. Next is a line containing four integers separated by blanks, of which the first two indicate the start point (x_1, y_1) and the last two indicate the end point (x_2, y_2) (where $0 \leq x_1, x_2 < X$ and $0 \leq y_1, y_2 < Y$). The third line of each test case contains an integer $P \leq 10$ indicating the number of obstacles in the grid. Finally, the test case consists of P lines, each specifying an obstacle.

Each obstacle consists of four integers x_1, y_1, x_2, y_2 (where $0 \leq x_1 \leq x_2 < X$ and $0 \leq y_1 \leq y_2 < Y$) meaning that all squares (x, y) with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$ are occupied.

The start point will never be occupied.

Output The string '`No solution.`' if there is no way the hopper can reach the finish point from the start point without hopping on an occupied square. Otherwise, the text '`Optimal solution takes` N `hops.`', where N is the number of hops needed to get from start to finish point.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <cassert>
#include <algorithm>
#include <climits>
using namespace std;

typedef struct {
    int x;
    int y;
} point;

point speed[] = {
    {-1,-1},
    {-1,0},
    {-1,1},
    {0,-1},
    {0,0},
    {0,1},
    {1,-1},
    {1,0},
    {1,1}
};

typedef struct {
    int x; // Current position
    int y;
    int sx;// Current speed
    int sy;
    int hops;
} hproperties;

hproperties make_hproperties(int x, int y, int sx, int sy, int hops) {
    hproperties h = {x, y, sx, sy, hops};
//     h.x = x;
//     h.y = y;
//     h.sx = sx;
//     h.sy = sy;
//     h.hops = hops;
    return h;
}

int map_speed(int x, int y) {
    return (x+3)*7+y+3;
}

int map_speed(point p) {
    return map_speed(p.x, p.y);
}

void testcase() {

    int X;
    int Y;
    cin >> X;
    cin >> Y;
    vector<vector<int> > db(X, vector<int>(Y, 0));
    vector<vector<vector<int> > > level(X, vector<vector<int> >(Y, vector<int>(49, -1)));
//     cout << level[0][2][4] << "\n";
    queue<hproperties> hqueue;

    point startpoint;
    point endpoint;
    // Start and End
    cin >> startpoint.x;
    cin >> startpoint.y;
    cin >> endpoint.x;
    cin >> endpoint.y;

    // Insert obstacles
    int numObstacles;
    cin >> numObstacles;
    for (int k=0; k<numObstacles; k++) {
```

```
        int x1, y1, x2, y2;
        cin >> x1;
        cin >> y1;
        cin >> x2;
        cin >> y2;

        for (int i=x1; i<=x2; i++) {
            for (int j=y1; j<=y2; j++) {
                db[i][j] = 1;
            }
        }
    }

    // Insert start point
    //db[startpoint.x][startpoint.y] = 0;
    if (db[endpoint.x][endpoint.y] > 0) {
        cout << "No solution.\n";
        return;
    }

    int posx = startpoint.x;
    int posy = startpoint.y;
    int numhops = 0;
    hqueue.push(make_hproperties(posx, posy, 0, 0, numhops));
    level[startpoint.x][startpoint.y][map_speed(0, 0)] = 0;
    while ( !(posx == endpoint.x && posy == endpoint.y) && !hqueue.empty()) {
        hproperties item = hqueue.front();
        hqueue.pop();
        numhops = item.hops + 1;
        assert(numhops >= 0);

        for (unsigned int k=0; k < 9; k++) {
            //cout << k << "\n";
            int speedx = item.sx + speed[k].x;
            int speedy = item.sy + speed[k].y;

            assert(speed[k].x >= -1 && speed[k].x <= 1 );
            assert(speed[k].y >= -1 && speed[k].y <= 1 );

            if (speedx > 3 || speedy > 3 || speedx < -3 || speedy < -3)
                continue;

            posx = item.x + speedx;
            posy = item.y + speedy;
            if (posx >= X || posy >= Y || posx < 0 || posy < 0)
                continue;
            if (db[posx][posy] > 0) // Obstacles
                continue;

            int speedhash = map_speed(speedx, speedy);

            if (level[posx][posy][speedhash] > -1)
                continue;

            level[posx][posy][speedhash] = numhops;

            if (posx == endpoint.x && posy == endpoint.y) {
                cout << "Optimal solution takes " << numhops << " hops.\n";
                return;
            }

            //db[cx][cy] = thops;
            hqueue.push(make_hproperties(posx, posy, speedx, speedy, numhops));
        }
    }
    if (posx == endpoint.x && posy == endpoint.y)
        cout << "Optimal solution takes " << numhops << " hops.\n";
    else
        cout << "No solution.\n";
}

int main() {
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases;i++) {
```

```
150             testcase();
        }
152 }
```

./week_2_fundamental/01/tracks.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week II

Algorithms Lab

Exercise 3 – *Forced Landing*

Imagine the airplane you are currently flying with encounters a problem with it's engines and has to make a forced landing. Luckily there is a big, rectangular area that is flat enough for landing. The only remaining problem is that there are many trees growing in this area and the pilot has problems finding enough space to land.

Since you are the only passenger who knows a lot about efficient algorithms they asked you for help. Their precise description is:

There is this rectangular area of width w and height h (in meters) we would like to land in. ("In" meaning entirely inside!) Unfortunately there are n trees on integer positions (x_i, y_i) for $1 \leq i \leq n$. Our airplane needs a strip that is s meters in width and at least r meters long. Since there is strong wind coming from east and our engines are broken, we'll have to land directly towards east or we will die. If you find more then one possible place to land, please point us to the one as far as possible in the west, and as far possible north if there are multiple equally-west solutions. This is the direction we are coming from and therefore maximizes our chances of survival. Please find the place for our forced landing if it exists!

Input The first line of the input file will contain an integer giving the number of test cases that follow.

Each test case starts with a line containing w and h, followed by a line containing the needed width s, the needed runway length r, and n, the number of trees. (w, h, s, r and n are integers). After that you'll find n lines, the i-th line being "x_i y_i", the integer coordinates of the i-th tree ($0 \leq x_i \leq w$, $0 \leq y_i \leq h$). The origin of the coordinate system lies in the north-west corner.

Output For each test case either the coordinates of the upper left corner of the area the airplane can land in, or the word "`Impossible`" if there is not enough space for landing. In the latter case let's hope all this was just one of your algorithmic dreams and not your end...

Remember that you should not just pick any place big enough, but the one as far as possible in the north-west, west being more important.

Constraints

- Small and bordercase datasets - $0 \leq h, w \leq 500$, $0 \leq n \leq 5000$
- Medium dataset - $0 \leq h, w \leq 100000$, $0 \leq n \leq 5000$
- Large dataset - $0 \leq h, w \leq 100000$, $0 \leq n \leq 100000$

Sample Input

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

Warning: Solution broken

```cpp
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm>

using namespace std;

void testcase() {
    int w,h; // width, height
    cin >> w;
    cin >> h;

    int s,r; // strip height, strip length
    cin >> s;
    cin >> r;

    int n; // trees
    cin >> n;

    // Reversed indexing
    vector<vector<int> > area(h+1, vector<int>(0));
    vector<vector<int>::iterator > lt(h+1);
    // Fill in trees
    for (int k=0; k<n; k++) {
        int x,y;
        cin >> x;
        cin >> y;
        area[y].push_back(x);
    }

    for (vector<vector<int> >::iterator it=area.begin(); it!=area.end(); it++) {
        sort(it->begin(), it->end());
    }
    for (int i=0; i<(h+1); i++) {
        lt[i] = area[i].begin();
    }

    for (int i=0; i<=(w-r); i++) {
//        cout << "i: " << i << "\n";
        int j=0;
        while (j <= (h-s)) {
            int notempty = 0;
            for (int l=1; l<s; l++) {
                vector<int>& loc = area[j+l];
                int foundtree = 0;
                int maxRange = i+r;
                vector<int>::iterator it = lt[j+l];
                for (; it!=loc.end(); it++) {
                    int v = *it;
                    if (v > i) {
                        if (v < maxRange ) {
                            lt[j+l] = it;
                            foundtree = 1;
                        }
                        break;
                    }
                }

                if (foundtree) {
                    notempty = 1;
                    j += (l);
                    if (l==0)
                        j++;
                    break;
                }
            }

            if (!notempty) {
                cout << i << " " << j << "\n";
                return;
            }
        }
    }
```

```
74      cout << "Impossible\n";

76  }

78  int main() {
        int testcases;
80      cin >> testcases;
        for (int i=0; i<testcases; i++) {
82          testcase();
        }
84  }
```

./week_2_fundamental/02/landing4.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week II

Algorithms Lab

Exercise 4 – *Demolition*

Description The first rule of Fight Club is: You do not talk about Fight Club. The second rule of Fight Club is: You do not talk about Fight Club...

As your next duty in Fight Club (led by Tyler Durden and Edward Norton), you have been assigned to the project Mayhem. What is the project Mayhem? Well, one simply does not speak of it, and one does not have a name once one becomes part of it. Anyhow, to make a long story short, your next mission in the project Mayhem is to demolish a building.

The building you want to demolish has n floors, and for each floor you know how much weight it can carry before it collapses, its current weight and how much it would cost to blow it up. There are two ways for a floor to collapse. Firstly, it can be simply blown up at a given cost. It can also collapse under excessive weight, i.e., if the total weight of the current floor and all the debris that fell on it from the previous collapses strictly exceeds its capacity. Whenever a floor collapses, all of its weight and all the debris lying on it fall down onto the floor below. The most important part of the building is the ground floor, and the whole building breaks down once it collapses. Thus, your goal is to destroy the ground floor. However, as the budget of the project Mayhem is a bit short these days, you have to do it as efficiently as possible!

Input The first line of the input contains $t \leq 5$, the number of testcases. Each test case starts with one line containing the number of floors $1 \leq n \leq 100000$, followed by n lines, specifying characteristics of the floors from bottom to top. Each of these n lines contains three positive integers which describe the current weight, total capacity and the cost of blowing up the i-th floor (in that order, weight $\leq$ capacity). All integers are in the interval $[0, 2^{31})$.

Output For each test case you should output a line containing the minimum cost of destroying the building.

Sample input

```
2
3
5 10 100
5 10 1
5 10 1
5
2 20 20
10 15 15
20 30 5
8 16 1
4 20 50
```

Sample output

```
2
5
```

Warning: Solution broken

```cpp
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;
typedef struct {
    int weight;
    int capacity;
    int cost;
} floor;

floor make_floor(int weight, int capacity, int cost) {
    floor f = {weight, capacity, cost};
    return f;
}

void testcase() {
    int n;
    cin >> n;
    vector<floor> building;
    for (int i=0; i<n; i++) {
        int weight, capacity, cost;
        cin >> weight;
        cin >> capacity;
        cin >> cost;
        building.push_back(make_floor(weight, capacity, cost));
    }
}

int main() {
    int testcases;
    cin >> testcases;
    for (int k=0; k<testcases; k++) {
        testcase();
    }
    return 0;
}
```

./week_2_fundamental/03/demolition.cpp

Part III

Week 3 - Playground - Part 2

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week III

Algorithms Lab

Exercise 2 – *Longest Path*

If you don't know about the longest path problem, listen to this song `http://www.youtube.com/watch?v=a3ww0gwEszo`.

Finding the longest path in a general graph is notoriously difficult task. Does it become easier if we consider only trees instead?

Input The first line of the input contains $t \leq 10$, the number of testcases. Each test case starts with one line containing the number of vertices $1 \leq n \leq 100000$, followed by $n - 1$ lines, each containing two numbers – labels of vertices which are connected by an edge. Each vertex has a unique label from the interval $[0, n - 1]$ and it is guaranteed that a given graph is a tree.

Output For each test case you should output a line containing the length of the longest path, that is, the number of vertices in the longest path.

Sample input

```
2
8
1 4
3 4
5 4
4 2
2 7
6 0
0 7
8
0 6
6 5
5 2
2 4
4 3
3 1
1 7
```

Sample output

```
6
8
```

Challenge If you find this exercise too easy, write a nonrecursive DFS to make it slightly trickier.

```cpp
#include <iostream>
#include <cassert>
#include <vector>
#include <queue>
//#include <pair.h>
#include <algorithm>
using namespace std;

int dfs(vector<vector<int> >& graph, int origin, int source) {
    int maxdepth = 1;
    for (vector<int>::iterator it=graph[origin].begin(); it != graph[origin].end(); it++) {
        int c = *it;

        if (c != source) {
            int depth = dfs(graph, c, origin)+1;
            if (depth > maxdepth) {
                maxdepth = depth;
            }
        }
    }
//    std::cout << "in: " << origin << " source: " << source << " distance: " << maxdepth <<  endl;
    return maxdepth;
}

pair<int, int> dfs_maxdist(vector<vector<int> >& graph, int origin, int source) {
    pair<int, int> res = make_pair(origin, 0);
    for (vector<int>::iterator it=graph[origin].begin(); it != graph[origin].end(); it++) {
        int c = *it;

        if (c != source) {
            pair<int, int> k = dfs_maxdist(graph, c, origin);
            k.second += 1;
            if (k.second > res.second) {
               res = k;
            }
        }
    }
    return res;
}

void testcase() {
    int v;
    cin >> v;
    vector<vector<int> > graph(v, vector<int>());
    for (int i=0; i < (v-1); i++) {
        int p1, p2;
        cin >> p1;
        cin >> p2;
        graph[p1].push_back(p2);
        graph[p2].push_back(p1);
    }


    // Make a dfs

    int maxdepth = 1;
    int origin = dfs_maxdist(graph, 0,0).first;
    for (vector<int>::iterator it=graph[origin].begin(); it != graph[origin].end(); it++) {
        int c = *it;
        maxdepth += dfs(graph, c, origin);
    }
    cout << maxdepth << endl;

}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases; i++) {
        testcase();
    }
```

```
75      return 0;
}
```

./week_3_fundamental_2/00/longest_path/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week III

Algorithms Lab

Exercise 2 – *Place the Beasts*

The professors of the ETH have a secret place where they keep new exams. No one knows what the size of that place really is. Some say that it is 4×4, others say that it is $\mathbf{100 \times 100}$. It can't be bigger because there is simply not enough space in the catacombs beneath the ETH main building. The rooms in this secret place are all square shaped and of unit length and width. So a 4×4 place has 16 rooms as shown in figure 1.

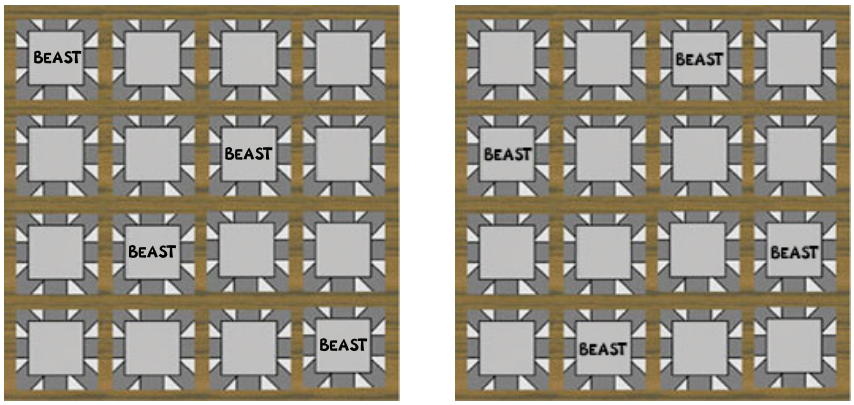
A $n \times n$ place has therefore n^2 rooms, and n evil beasts are used to look after the $n \times n$ sized secret place, such that no student who finds the secret entrance to the catacombs will ever exit them again.

In figure 1 below, each second largest square denotes a room. The gray square inside each room indicates free space where the evil beasts live or exams are kept. The outgoing tunnels (dark gray in color) are the only connection between the rooms. The tunnels are, as you can see, designed in such a way that only the beasts on the same row, same column, or same diagonal can see each other.

In figure 1(a) beast $(1, 1)$ and beast $(4, 4)$, as well as beast $(3, 2)$ and beast $(2, 3)$, can see each other thru the available tunnels. (Coordinate (3,2) denotes the 2nd column in the 3rd row; rows and columns are counted starting from 1.)

The professors have to place the evil beasts in such a way that no beast can see any other beast (like shown in figure 1(b)), otherwise they would immediately start to kill each other. (This is also the reason why they are called *evil* beasts!)

Please help the professors to place the beasts in such a valid way.



(a) **(b)**

Figure 1: (a) an invalid beast placement, (b) a valid one

```cpp
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <algorithm>

using namespace std;

int prev_run;
int counter_fallof;

#define MAX_GRID 100
bool tried[MAX_GRID+1][MAX_GRID+1];
int grid[MAX_GRID+1];
int n;
// Recursive Method: Slow
bool recursion(int depth) {
    if (depth > n) {
        return true;
    }

    int irun;
    tried[depth][0] = true;
    fill(&tried[depth][1], &tried[depth][n+1],false);
    int z = n+1;
    for (int i=0; i<n; i++) {
        while(true) {
            irun = (rand())%(z);
            if (!tried[depth][irun]) {
                break;
            }
//              cout << depth << " " << irun << endl;
        }
        tried[depth][irun] = true;

        bool cont = false;
        for (int j=1; j < depth; j++) {
            int l = grid[depth-j];
            if (grid[j] == irun || l == (irun-j) ||l == (irun+j)) {
                cont = true;
                break;
            }
        }
        if (cont) {
            continue;
        }

        grid[depth] = irun;

        if (recursion(depth+1)) {
            return true;
        }
    }
    return false;
}

void testcase() {
//     int n;
    cin >> n;
//     vector<int> grid(n+1,0);
//     for (int i=0; i < n+1; i++) {
//         grid[i] = i;
//     }
    if (recursion(1)) {
        for (int i=1; i<=n; i++) {
            cout << grid[i] << " ";
        }
        cout << endl;
        return;
    }
    cout << "Impossible" << endl;
    prev_run = n;
}

int main()
```

```
75 {
       srand(time(NULL));
77     cin.sync_with_stdio(false);
       cout.sync_with_stdio(false);
79     int testcases;
       cin >> testcases;
81     prev_run = -1;

83     for (int i=0; i<testcases; i++) {
//         testcase(grid);
85         testcase();
       }
87     return 0;
   }
```

./week_3_fundamental_2/01/beasts/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week III

Algorithms Lab

Exercise 3 – *Burning Coins from Two Sides*

A friend of yours who recently sold his internet startup for several million dollars wants to share parts of his money with you. The catch is that he will not just give it to you but that he wants to play a game that will determine how much you actually get. The game works as described below.

There are n coins lined up in a row ($0 \leq n \leq 1000$). Each coin has a specific, known value $0 < v_i \leq 1000$. The game is played in turns and you are the one to start. Every time it is your turn you can either take the first or last coin in the row. Once you took the coin of your choice your friend will do the same (also either take the now first or last coin in the row), and this procedure will be iterated till someone has taken the last coin. The money you get at the end is the sum of the values of the coins you have collected while playing this simple game.

For a given row of coins $c_1, c_2, \ldots, c_n$ with corresponding values $v_1, v_2, \ldots, v_n$ we want to know how much money you are guaranteed to win if you manage to play optimally. (Independent of the strategy of your friend.)

Input The first line contains c, the number of testcases following. Each testcase consists of a single line containing $n + 1$ integers. The first integer is n, the number of coins used for the game, followed by n integers representing the individual coin values $v_1, v_2, \ldots, v_n$.

Output After analyzing the game simply print w_{min}, the amount you are guaranteed to win if both players play an optimal strategy, on a line of its own. Remember that you start taking the first coin.

Sample Input

```
2
3 3 1 2
4 1 4 9 4
```

Sample Output

```
4
10
```

(100 Points)

```cpp
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;


int recursion(vector<vector<int > >& map, vector<int>& values, int left, int right) {
    if (left==right) {
        return values[left];
    }
    if (map[left][right] > 0) {
        return map[left][right];
    }
    int d = right-left;
    if (d <= 2) {
        // Case [left|right]
        int l = values[left];
        int m = values[left+1];
        int r = values[right];
        if (d == 1) {
            if (l>r) {
                return l;
            }
            return r;
        }
        // Case [left|middle|right]
        int v;
        if (l>r) {
            v = l;
            if (m > r) {
                v += r;
            } else {
                v += m;
            }
        } else {
            v = r;
            if (m > l) {
                v += l;
            } else {
                v += m;
            }
        }
        return v;
    }

    // We choose the left value:
    int ll = recursion(map, values, left+2, right);    // He chose the left value as well
    int lr = recursion(map, values, left+1, right-1); // Chose the right value

    // We choose the right value
    int rl = lr;
    int rr = recursion(map, values, left, right-2);

    int k = max(values[left]+min(ll, lr),
                values[right]+min(rl, rr));
    map[left][right] = k;
    //    return make_pair(r.first+values[right], r.second);
    return k;
}


void testcase() {
    vector<int> values;

    int n;
    cin >> n;
    if (n==0) {
        cout << 0 << endl;
        return;
    }
    vector<vector<int > > map(n+1,  vector<int>(n+1, -1));

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

```
        int t;
        cin >> t;
        values.push_back(t);
    }

//    int max = recursion(map, values, 0, n-1);
    cout << recursion(map, values, 0, n-1) << endl;
//    pair<int, int> ret = me(map, values, 0, n-1);
//    cout << ret.first << endl;
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_3_fundamental_2/02/burningcoins/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet ACM Week III

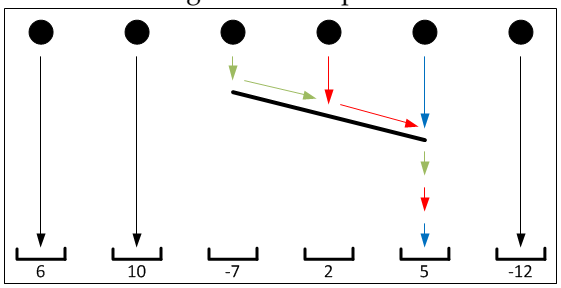
Algorithms Lab

Exercise 4 – *Pinball*

During your last archaeological excavation in Scotland, you found a star map that might lead you to an answer to the question about the origins and the purpose of the mankind. However, since the star map is pointing to a planet which is far far away, and the rest of your crew are quite boring people, you have to find a way to amuse yourself for the next couple of years (how much it takes to get to that planet).

Fortunately, beside the star map, you also discovered an ancient version of a pinball game, probably of an alien origin. This version has n balls, each placed above a hole (as in the picture), and you have to place *exactly one* obstacle that is tilted strictly to the right. Your obstacle must cause at least one ball to fall into a hole that is not directly below it. Each hole has a number of points associated with it, and for each ball that falls into it you get that many points. Your goal is to achieve the largest possible score!

Figure 1: Example.



Input The first line of the input contains $t \leq 6$, the number of testcases. Each test case starts with one line containing the number of holes (which is also the number of balls) $2 \leq n \leq 300000$, followed by a line which contains n numbers, specifying number of points associated with each hole, from left to right. Points are integers in the interval $[-2^{31}, 2^{31})$.

Output For each test case you should output a line containing the maximum score you can achieve.

Sample input

```
3
6
6 10 -7 2 5 -12
5
1 6 1 2 2
5
5 4 3 2 1
```

Sample output

```
19
17
14
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

/**
SPOILER <<<

    First find the optimal placement for the median of S.
    This gives an upper bound for every element in S which
    is smaller than the median, and a lower bound for every
    element which is larger then the median. Then try to recursively
    continue in this manner. Hope this helps!

>>>
*/

inline int compute(vector<int>& values, vector<int>& sums, int Sn, int i, int j) {
    return Sn-(sums[j]-sums[i-1])+(j-i+1)*values[j];
}

void testcase() {
    vector<int> values;
    values.push_back(0);
    vector<int> sums;
    sums.push_back(0);


    int n;
    cin >> n;
    if (n==0) {
        cout << 0 << endl;
        return;
    }

    int sum = 0;
    sums.reserve(n+1);
    values.reserve(n+1);
    for (int i=0; i<n; i++) {
        int t;
        cin >> t;
        values.push_back(t);
        sum += t;
        sums.push_back(sum);
    }

    // Set active values
    vector<int> activearr;
    int a = 0;
    for (int i=1; i<n; i++) {
        if (values[i] > values[i+1]) {
            if (a == 0) {
                activearr.push_back(i);
            }
            a++;
        } else {
            if (a > 0) {
                if (activearr.back() != (i-1)) {
                    activearr.push_back(i-1);
                }
            }
            a=0;
        }
    }
    if (a > 0) {
        if (activearr.back() != (n-1)) {
            activearr.push_back(n-1);
        }
    }
    if (!activearr.back() != (n)) {
        activearr.push_back(n);
    }

//    for (vector<int>::iterator it=active.begin(); it!=active.end(); it++) {
//        cout << *it << " ";
```

```
//     }
//     cout << endl;

//     cout << "Active Array" << endl;
//     for (vector<int>::iterator it=activearr.begin(); it!=activearr.end(); it++) {
//         cout << *it << " ";
//     }
//     cout << endl;

    int Sn = sums.back();
    int r = compute(values, sums, Sn, 1,2);

    for (int j=n; j>0; j--) {

        int Sj = sums[j];
        int SnSj = Sn-Sj;
        int vj = values[j];
        if (active[j]) {
            for (vector<int>::iterator it=activearr.begin(); it!=activearr.end(); it++) {
                int i = *it;
                int i2=i+1;
                if (i2 >= j) {
                    break;
                }
                int k2 = SnSj+sums[i2-1]+(j-i2+1)*vj;
                if (k2>r) {
                    r = k2;
//                     cout << i2 << " " << j << endl;
                }

                int k = SnSj+sums[i-1]+(j-i+1)*vj;
                if (k>r) {
                    r = k;
//                     cout << i << " " << j << endl;
                }
            }
        }
    }
    cout << r << endl;
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_3_fundamental_2/03/pinball/main.cpp

Part IV

Week 4 - BGL Introduction

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week I

Algorithms Lab

Exercise 0 – *First steps with BGL*

Read a weighted undirected graph, compute the total weight of its minimum spanning tree and the distance from node 0 to the node furthest from it.

Input The first line of the input file contains $t \leq 100$, the number of the test cases.
Each test case starts with a line containing $n \leq 100, m \leq \frac{n \cdot (n-1)}{2}$, the number of vertices and edges of the graph. m lines follow, each defining the starting point, ending point, and weight of an graph edge. All weights are non-negative integers not bigger than 1000.
The input graph will always be connected.

Output For each test case output a single line containing w, the sum of the weights of all edges of the minimum spanning tree, and d, the distance from node 0 to the node furthest from it.

Sample input

```
1
5 6
0 1 1
0 2 2
1 2 5
1 3 1
3 2 2
2 4 3
```

Sample output

```
7 5
```

```cpp
#include <iostream>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       no_property, property <edge_weight_t, int > > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

typedef pair<int, int> E;

void testcase() {
    int n; // vertices
    int m; // edges
    cin >> n;
    cin >> m;
    Graph g(n);

    property_map<Graph, edge_weight_t>::type weightmap = get(edge_weight, g);

    for (int i=0; i<m; i++) {
        int u, v, w;
        cin >> u;
        cin >> v;
        cin >> w;
        bool success;
        Edge e;
        tie(e, success) = add_edge(u, v, g);
        weightmap[e] = w;
    }

    property_map<Graph, edge_weight_t>::type weight = get(edge_weight, g);
    vector<Edge> spanning_tree;
    kruskal_minimum_spanning_tree(g, back_inserter(spanning_tree));

    int total_weight = 0;
    for (vector < Edge >::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei) {
        total_weight += weight[*ei];
    }

    // Let's take out the hammer and compute dijkstra as well...
    std::vector<Vertex> predecessors(num_vertices(g));
    std::vector<int> distances(num_vertices(g));
    dijkstra_shortest_paths(g, 0,
      predecessor_map(&predecessors[0]).distance_map(&distances[0]));

    int d = 0;
    for (vector<int>::iterator it = distances.begin(); it != distances.end(); it++) {
        int i = *it;
        if (i > d)
            d = i;
    }
    cout << total_weight << " " << d << "\n";
}

int main()
{
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_4_bgl_introduction/0_graphs/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week I

Algorithms Lab

Exercise 1 – *Ant Challenge*

Ants are amongst the most intensively studied forest insects. Especially their genius in the field of carrying stuff does not cease to amaze researchers. No wonder that all the other forest bugs are full of envy. After years of suffering, the forest bugs under the leadership of the termite faction have decided to show the world that they do possess similar logistics genius. For that purpose they have decided to organize a demonstration in which they intend to prove that by working together they can carry a breadcrumb even faster from one end of the forest to another than the ants.

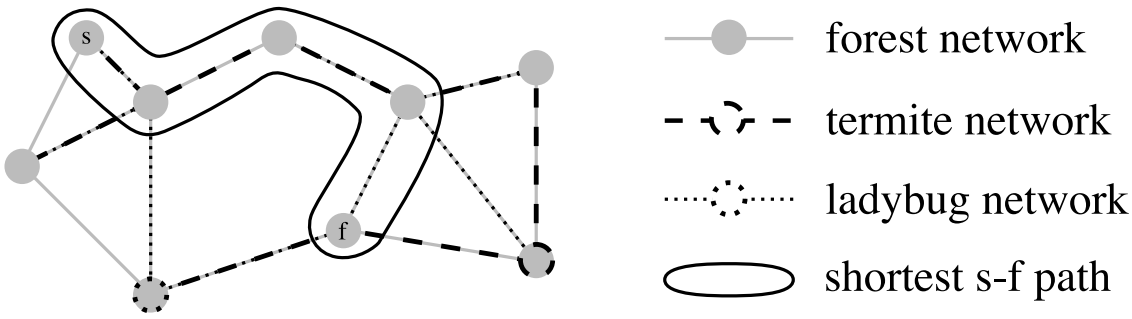
Now that the date of the event is drawing near, some of the insects start to be afraid that they can not meet the termites' expectations. They have asked you for help!

Problem We assume that insects move from tree to tree such that we can model the forest as a graph in which vertices correspond to the trees of the forest. There is an edge between two trees, if there is a direct route between them in the forest. The duration of traveling along an edge of this graph differs from species to species. In addition, an insect can only move along edges that are part of its species' private network.

The private network of each species was established years ago during the exploration of the forest: Starting from its hive located at a particular tree, each species explored new trees one at a time. The next tree to be explored was always the one fastest to be reached from the already explored territory (you may assume that every edge has a unique length for every species). The private network of every species covers all trees of the forest and contains all edges that were used during this species' exploration of the forest.

For all species of forest insects, you are provided their hive's location together with the time it takes an insect of this species to travel along each tree-to-tree edge. Given a start tree and a finish tree, you are to find the duration of the fastest route through the combined private networks of all species from start to finish. The different species can alternate carrying the breadcrumb: Every edge in the shortest path must be covered by at least one private network and if an edge is covered by multiple networks, you are free to choose the species that can travel the edge fastest for carrying along that edge.

See the figure below for an example with two species: Both species need to take over parts of the shortest route.



```
#include <iostream>
#include <climits>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
using namespace std;

using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       no_property, property <edge_index_t, size_t > > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

// Obviously we need Dijkstra...
void testcase() {
    int nvertices;
    int nedges;
    cin >> nvertices;
    cin >> nedges;
    int nspecies;
    cin >> nspecies; // Species

    int start;
    int end;
    cin >> start;
    cin >> end;

    Graph g(nvertices);
    property_map<Graph, edge_index_t>::type indices = get(edge_index, g);

    vector<vector<int> > species_weights(nspecies, vector<int>(nedges, 0));
    // Load all edges
    for (int i=0; i<nedges; i++) {
        int u, v;
        cin >> u;
        cin >> v;
        bool success;
        Edge e;
        // Don't forget to set edge_index through i (we need it below)
        tie(e, success) = add_edge(u, v, g);
        // Insert weight for all species individually
        indices[e] = i;
        for (int k=0; k<nspecies; k++) {
            int t;
            cin >> t;
            species_weights[k][i] = t;
        }
    }

    int b;
    for (int k=0; k<nspecies; k++) {
        cin >> b;
    }

    vector<int> edge_weights(nedges, INT_MAX);
    for (int i=0; i<nspecies; i++) {
        vector<Edge> spanning_tree;
        kruskal_minimum_spanning_tree(g,
                                      back_inserter(spanning_tree),
                                      weight_map(
                                          make_iterator_property_map(
                                              species_weights[i].begin(),
                                              indices
                                              )
                                          )
                                      );

        for (vector < Edge >::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei)
            {
```

```
            Edge e = *ei;
            if (edge_weights[indices[e]] > species_weights[i][indices[e]]) {
                edge_weights[indices[e]] = species_weights[i][indices[e]];
            }
        }
    }

    // Compute dijkstra
    std::vector<Vertex> predecessors(num_vertices(g));
    std::vector<int> distances(num_vertices(g), INT_MAX);

    dijkstra_shortest_paths(g, start,
                            predecessor_map(&predecessors[0])
                            .distance_map(&distances[0])
                            .weight_map(
                                make_iterator_property_map(
                                    &edge_weights[0],
                                    indices
                                    )
                                )
                            );

    cout << distances[end] << "\n";

}

int main()
{
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_4_bgl_introduction/1_ant_challenge/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week I

Algorithms Lab

Exercise 2 – *Deleted Entries*

Thanks to the hard work of the assistants, the infamous lab at the Algorithmic University is getting harder each year. Finally, in 2021 it appears impossible for a single student to cover the whole curriculum. Since the course consists of three independent sections, the teachers decide to split the students into three groups so that each of them learns only a single part. Of course at least someone should participate in each section, so each group should be nonempty.

To "help" the students work independently, members of each group are requested to mutually delete each other's details from their contact books (if they had their details in the first place). On the other hand, it is very advisable to communicate with the students from other groups, exchanging the acquired abilities and knowledge. What is more, it should be possible for a student to reach any other student in case of emergency.

We say that Alice can reach Bob if:

- She has Bob's number in her contact book or
- She has the contact details of someone who can reach Bob.

You should determine if it is possible to divide the students into nonempty groups such that each student can reach any other student even after deleting the contact entries inside the groups. Additionally, if such division is possible you should find it.

Input The first line of the input contains $1 \leq t \leq 100$, the number of testcases. Each testcase starts with a line containing $1 \leq n \leq 10^6, 0 \leq m \leq 2 \cdot 10^6$, where n is the number of students and m the number of pairs of students that have each other's contact details. We assume that the details are exchanged mutually, i.e., if Alice has Bob's details, then Bob also has her details. We number the students with integers from 0 to $n-1$. m lines follow, each defining a pair of students that exchanged contact details: $0 \leq a_i < b_i < n$. In each line consecutive numbers are separated with single spaces. No pair of students appears more than once in a testcase.

Output You should output a separate answer for each testcase as follows. If it is not possible to divide the students into the groups as required, you should print `no` on a single line. Otherwise, you should print `yes` on a single line, followed by the description of a correct division in the three following lines. Each of them should start with the number of students in the respective group, followed by a list of the students assigned to the group. If there is more than one correct solution, output any of them.

NOTE The inputs to this problem are huge, make sure your I/O is efficient.

```cpp
#include <iostream>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/prim_minimum_spanning_tree.hpp>
#include <boost/graph/breadth_first_search.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       no_property, property <edge_weight_t, int > > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;


int color_one_counter;
class coloring_visitor: public default_bfs_visitor {
public:
    vector<int> &coloring;
    coloring_visitor(vector<int> &colors):
        coloring(colors) {

    }

    void tree_edge(Edge e, const Graph& g) {
        Vertex sn = source(e,g);
        Vertex en = target(e,g);
        if (coloring[sn] == 0) {
            coloring[en] = 1;
            color_one_counter++;
        } else {
            coloring[en] = 0;
        }
    }


};

void testcase() {
    int n; // students
    cin >> n;
    int m; // pairs
    cin >> m;
    Graph g(n);
    for (int i=0; i<m; i++) {
        int u, v;
        cin >> u;
        cin >> v;
        Edge e;
        bool success;
        tie(e, success) = add_edge(u,v,g);
    }
    if (n < 3) {
        cout << "no" << endl;
        return;
    }

    // Compute connected components
    vector<int> components(n);
    if (connected_components(g, &components[0]) > 1) {
        cout << "no" << endl;
        return;
    }

    // Create MST
    vector<Vertex> predecessors(n);
    prim_minimum_spanning_tree(g, &predecessors[0]);

    Graph mst(n);
    for (int i=0; i<n; i++) {
        Edge e;
```

```
75          bool success;
            if (predecessors[i] != i) {
77              // Upon completion of the algorithm, the edges (p[u],u)
                // for all u in V are in the minimum spanning tree.
79              // If p[u] = u then u is either the root of the tree or is
                // a vertex that is not reachable from the root.
81              tie(e, success) = add_edge(predecessors[i], i, mst);
            }
83      }

85      // Make graph coloring
        vector<int> coloring_groups(n,0);
87      color_one_counter = 0;
        // initialize visitor
89      coloring_visitor vis(coloring_groups);
        breadth_first_search(mst, vertex(0, mst), visitor(vis));
91      // Make first color 2:
        coloring_groups[0] = 2;
93      // the first color is at the center of the star, so we make the second vertex a 0,
        if (color_one_counter == (n-1)) {
95          coloring_groups[1] = 0;
        }
97      cout << "yes" << endl;

99      // 0 has it's own color
        cout << "1 0" << endl;
101
        int c0 = 0;
103     for (int i=0; i<n; i++) {
            if (coloring_groups[i] == 0) {
105             c0++;
            }
107     }
        cout << c0;
109     for (int i=0; i<n; i++) {
            if (coloring_groups[i] == 0) {
111             cout << " " << i;
            }
113     }
        cout << endl;
115
        int c1 = 0;
117     for (int i=0; i<n; i++) {
            if (coloring_groups[i] == 1) {
119             c1++;
            }
121     }
        cout << c1;
123     for (int i=0; i<n; i++) {
            if (coloring_groups[i] == 1) {
125             cout << " " << i;
            }
127     }
        cout << endl;
129 }


131 int main()
133 {
        cin.sync_with_stdio(false);
135     cout.sync_with_stdio(false);
        int testcases;
137     cin >> testcases;

139     for (int i=0; i<testcases; i++) {
            testcase();
141     }
        return 0;
143 }
```

./week_4_bgl_introduction/2_deleted_entries/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week I

Algorithms Lab

Exercise 3 – *Shy Programmers*

Finally, you are in charge of your own software project! This is not a piece of cake, though. You hired some programmers and now need to place their workstations in a rectangular room. The space is not a problem – your programmers need hardly more than a chair and a small desk, so you can assume they occupy a single point on the plane – just don't put two of them in the same point!

The real issue is a social one. Some of your programmers are friends with each other. Of course each pair of friends likes to talk to each other from time to time, so you want to make a placement such that the friends can move in a straight line between their workstations – no other workstation on their way (you don't want the shape of the path to be more complicated than a straight segment in case a programmer is busy thinking about her code on the way).

What is more, your employees do not like surprises, especially when it comes to bumping into other people, sometimes people who they don't even know! Hence, to avoid the risk of unexpected encounters, you want to place them so that no two paths connecting friends cross or touch each other except in their endpoints.

Last but not least, even your hard-working coders need to have lunch or use a toilet from time to time. Again, you want to make it easy and predictable for them. Thus, you decide to put a separate door for each programmer in the room's wall so that a programmer can go from her place to the door in a straight line. As previously, those paths should not cross with each other or any of the previous paths except in their endpoints.

As a first step to the success of your project you want to establish if such a placement is even possible given the friendship network of your employees.

Input The first line of the input contains $1 \leq t \leq 100$, the number of testcases. Each testcase starts with a line containing $1 \leq n \leq 10^5, 0 \leq m \leq 2 \cdot 10^5$, where n is the number of employees and m is the number of pairs of friends. We assume that employees are numbered from 0 to $n-1$. m lines follow, each defining a pair of friends $0 \leq a_i < b_i < n$. In each line consecutive numbers are separated with single spaces. No pair of friends appear more than once in a testcase.

Output For each testcase output a single line containing `yes` if a placement is possible and `no` otherwise.

NOTE Watch out for large inputs and read the data efficiently.

```cpp
#include <iostream>

#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       property<vertex_index_t, int>,
                       property<edge_index_t, int>
                      > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

void testcase() {
    int n; // students
    cin >> n;
    int m; // pairs
    cin >> m;
    Graph g(n+1);
    for (int i=0; i<m; i++) {
        int u, v;
        cin >> u;
        cin >> v;
        Edge e;
        bool success;
        tie(e, success) = add_edge(u,v,g);
    }

    for (int i=0; i<n; i++) {
        Edge e;
        bool success;
        tie(e, success) = add_edge(i,n,g);
    }

//    vector< vector<Edge> > embedding(n+1);
    bool is_planar = boyer_myrvold_planarity_test(boyer_myrvold_params::graph = g
//                                                  boyer_myrvold_params::embedding = &embedding[0]
                                                 );

    if (!is_planar) {
        cout << "no" << endl;
        return;
    }
    cout << "yes" << endl;
//    cout << "Testcase, n: " << n << endl;


//    for (vector<vector<Edge> >::iterator it=embedding.begin();
//            it != embedding.end(); it++) {
//        cout << (*it).size() << endl;
//    }

//    // Check for K4
//    for (vector<vector<Edge> >::iterator it=embedding.begin();
//            it != embedding.end(); it++) {
////        cout << (*it).size() << endl;

//    }
//    for (vector<Edge>::iterator it=kuratowski_edges.begin();
//         it != kuratowski_edges.end(); it++) {
//        cout << *it << endl;
//    }
}
int main()
{
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases; i++) {
        testcase();
```

```
75      }
        return 0;
77  }
```

./week_4_bgl_introduction/3_shy_programmers/shy_programmers/main.cpp

Part V

Week 5 - Flows

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week II

Algorithms Lab

Exercise 1 – *Risky Sports Betting*

Betina and Winson have recently discovered their new favorite pastime: betting on sport leagues. They especially enjoy bets against all odds that a certain underdog team will win its league. To get a better idea of the quality of the individual teams, Betina and Winson start betting only after the season has progressed sufficiently. They want to avoid betting on teams that stand no chance at all, however it is often difficult to see which of the teams are theoretically still capable of becoming a champion. This is where you come in.

Problem Given the number t of teams, the current standings of the league and all remaining matchups, determine which teams can still become a champion of the league. A team is said to be a champion when it has the most or is tied for the most points in a season. The team winning a matchup is awarded one point and there are no draws.

Input The first line of the input contains the number of test cases $n \approx 100$, n test cases follow. Each test case starts with a line containing the number of teams $1 \leq t \leq 100$ and the number of remaining matchups $0 \leq m \leq 100$ separated with a single space. The next t lines contain an integer $0 \leq p \leq 10^8$ and a string $name$ consisting of at most 50 alphanumeric ('A'-'Z', 'a'-'z', '0'-'9') characters each (again separated by a single space), meaning that the specified team has scored p points in the season so far. Finally another m lines of the form "$name1$ vs. $name2$" follow, each specifying one matchup that still has to be played.

Output For every test case output a single line containing the names of the teams that can still win the league ordered lexicographically (based on ASCII code, i.e. 'Z' < 'a') and separated with single spaces.

```
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <algorithm>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
                       property<edge_capacity_t, long,
                       property<edge_residual_capacity_t, long,
                       property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor Edge;

inline int get_index(const vector<string> &team_name, const string &str) {
    int i=0;
    for (vector<string>::const_iterator it=team_name.begin();
         it != team_name.end(); it++) {
        if (str.compare((*it)) == 0) {
            return i;
        }
        i++;
    }
}

void testcase() {
    int n;
    int m;
    cin >> n;
    cin >> m;


    Graph g(n+m+2);
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);

    vector<string> team_name(n);
    vector<int> team_points(n);

    int source_edge = n+m;
    int sink_edge = n+m+1;

    long sum_points = m; // the number of matches and the sum of all player points
    vector<Edge> sink_edges;
    for (int i=0; i<n; i++) {
        string t;
        int p;
        cin >> p;
        cin >> t;
        team_name[i] = t;
        team_points[i] = p;
        sum_points += p;
        Edge e, rev_e;
        bool success;
        // Create edge from source to player
        tie(e, success) = add_edge(source_edge, i, g);
        tie(rev_e, success) = add_edge(i, source_edge, g);
        capacity[e] = p;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;

        // Create edge from player to sink
        tie(e, success) = add_edge(i, sink_edge, g);
        tie(rev_e, success) = add_edge(sink_edge, i, g);
        capacity[e] = p;
```

```
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
        // Store sink edges
        sink_edges.push_back(e);
    }

    // Process matchings
    for (int i=0; i<m; i++) {
        string t1, t2, vs;
        cin >> t1 >> vs >> t2;
        int i1 = get_index(team_name, t1);
        int i2 = get_index(team_name, t2);

        // Count the matches
        team_points[i1]++;
        team_points[i2]++;

        Edge e, rev_e;
        bool success;
        // Set edge from source to match
        tie(e, success) = add_edge(source_edge, n+i, g);
        tie(rev_e, success) = add_edge(n+i, source_edge, g);
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;

        // Set edge from match to player1
        tie(e, success) = add_edge(n+i, i1, g);
        tie(rev_e, success) = add_edge(i1, n+i, g);
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;

        // Set edge from match to player2
        tie(e, success) = add_edge(n+i, i2, g);
        tie(rev_e, success) = add_edge(i2, n+i, g);
        capacity[e] = 1;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
    }

    vector<string> winning_teams;
    for (int i=0; i<n; i++) {
        // Set all sink edges to the maximum capacity of the current team
        long max_points = team_points[i];
        for (vector<Edge>::iterator it=sink_edges.begin(); it != sink_edges.end(); it++) {
            capacity[*it] = max_points;
        }


        int max = push_relabel_max_flow(g, source_edge, sink_edge);
        // use sum_points to check if we got the maximum and all edges are used
        if (max == sum_points) {
            winning_teams.push_back(team_name[i]);
        }
    }
    sort(winning_teams.begin(), winning_teams.end());
    for (vector<string>::iterator it=winning_teams.begin(); it!= winning_teams.end(); it++) {
        cout << *it;
        if ((it+1) != winning_teams.end()) {
            cout << " ";
        }
    }
    cout << endl;

}

int main()
{
    int testcases;
    cin >> testcases;
```

```
150     for (int i=0; i<testcases; i++) {
            testcase();
152     }
        return 0;
154 }
```

./week_5_flows/0_tournament/tournament/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week II

Algorithms Lab

Exercise 2 – *Algocoön Group*

Your idea for a startup established with your partner turned out to be a great success. On impulse, you jointly decided to spend a big chunk of your profits on one of those ancient Greek sculptures depicting a group of mythological figures entangled in a complicated configuration.

Unfortunately, you didn't give this idea enough thought. The problem is that both of you want to put the masterpiece in your living rooms, but you can afford only one sculpture. You already committed yourself to the transaction, but the auction house gave you some choice – you can choose one of the several sculptures they have in stock.

The only feasible solution at this point is to choose one of the artworks and hire a stonecutter to cut the piece in two parts so that each of you can take one part home. Fortunately none of you is a greedy person and you will be fine with any cut as long as each of you is going to get *something* to put in their room.

To minimize the cost of your whim, write a program that will process the descriptions of the sculptures and output the cost-optimal way of cutting each of them.

Input The first line of the input contains $1 \leq t \leq 100$, the number of sculptures. For each sculpture, its description starts with the line containing $2 \leq n \leq 100, 0 \leq m \leq 10^4$, where n is the number of the figures in the sculpture and m is the number of places where two of the figures are entangled and require work from the stonecutter.

m lines follow, each of them containing $a, b, c, 0 \leq a, b < n, 1 \leq c \leq 1000, a \neq b$, indicating that figures a and b are intertwined such that the stonecutter will charge c franks for separating the link. Note that a pair of figures can appear several times, being entangled in more than one place. Also note that the sculpture can even consist of several disconnected pieces. As usual, in each line consecutive numbers are separated with single spaces.

Output For each sculpture, output the way of cutting the links that will cost as little as possible and allow both you and your partner to take home a nonempty part of the sculpture.

In the first line, print the total cost of the optimal cut. In the second line, output the number of figures you are going to take home, followed by a single-space separated list of those figures. It is assumed that all the other figures will be taken by your associate. If there is more than one solution, output any of them.

```cpp
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/stoer_wagner_min_cut.hpp>
#include <boost/graph/one_bit_color_map.hpp>
#include <boost/property_map/property_map.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       no_property, property <edge_weight_t, int > > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

// Solving the problem using mincut

void testcase() {
    int n;
    int m;
    cin >> n >> m;
    Graph g(n);

    property_map<Graph, edge_weight_t>::type weightmap = get(edge_weight, g);

    for (int i=0; i<m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        bool s;
        Edge e;
        tie(e, s) = add_edge(u,v,g);
        weightmap[e] = w;
    }
//    vector<default_color_type> parities(n);
    BOOST_AUTO(parities, boost::make_one_bit_color_map(num_vertices(g), get(boost::vertex_index, g))
        );
    int w = boost::stoer_wagner_min_cut(g, weightmap,parity_map(parities));
//    cout << "The min-cut weight of G is " << w << ".\n" << endl;

//    cout << "One set of vertices consists of:" << endl;
//    size_t i;
//    for (i = 0; i < num_vertices(g); ++i) {
//      if (get(parities, i))
//        cout << i << endl;
//    }
//    cout << endl;

//    cout << "The other set of vertices consists of:" << endl;
//    for (i = 0; i < num_vertices(g); ++i) {
//      if (!get(parities, i))
//        cout << i << endl;
//    }
//    cout << endl;

    vector<int> my_stuff;
    int i;
    for (i = 0; i < num_vertices(g); ++i) {
      if (get(parities, i))
//        cout << i << endl;
        my_stuff.push_back(i);
    }
    cout << w << endl;
    cout << my_stuff.size() << " ";
    for (vector<int>::iterator it=my_stuff.begin(); it !=my_stuff.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
```

```
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
//         testcase(grid);
        testcase();
    }
    return 0;
}
```

./week_5_flows/1_algocon_group/algocon/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week II

Algorithms Lab

Exercise 3 – *Kingdom Defence*

In a land far far away lies the mysterious kingdom of swamps. For centuries it has been a peaceful country little heeded by the lands around it. But times change and the ever-growing greed of its neighbors has brought the kingdom of swamps to the brink of war. While the peaceful inhabitants of the kingdom deeply despise war, some cities have been wise enough to start training soldiers early and now have well equipped garrisons. Other cities only command negligible defences.

The king of the swamps is a wise and gentle ruler. He wants to avoid open war at all costs. With its abundance of impenetrable swamps, his kingdom is well protected and the king hopes that if the most important strategic locations are strengthened and enough military presence is openly displayed, no neighbor will dare to attack. Moving battalions of soldiers around through the swamps is difficult however. There are only very narrow paths which can only be used in one direction as it is not always possible to make room for oncoming traffic. The king has asked you for help.

Problem You are given a map of the (directed) paths between the most important strategic locations in the kingdom of swamps. Every location has a number of soldiers available and every location needs a certain number of soldiers to defend itself. In order to display military presence, each path has to be travelled by some minimum number of soldiers. Also, because soldiers on the move constantly need to supply themselves with food, every path has a maximum number of soldiers that can walk along it in total.

You have to decide whether it is possible to satisfy all these requirements by moving battalions of soldiers around in the kingdom.

Input The first line of the input contains the number n of test cases to follow. Every test case starts with a single line containing two space-separated integers $1 \leq l \leq 500, 1 \leq p \leq l^2$, specifying the number of locations and paths, respectively. The next l lines each give the details of one location. The i-th such line contains two space-separated integers $1 \leq g_i, d_i \leq 10^6$, the number of soldiers stationed at location i and the number of soldiers that location i needs to defend itself, respectively. Note that $g_i > 0$ for all locations. After that, there are p lines each specifying one path in the kingdom with four integers $0 \leq f_j, t_j < l$ and $0 \leq c_j \leq C_j \leq 10^6$. The j-th path is one-way and goes from location f_j to location t_j (zero-based indexing, f_j and t_j can be equal!). The path needs to be traversed by at least c_j and by at most C_j soldiers. Note that a single soldier may use the same path multiple times, but each time counts as one traversal.

Output For every test case output a single line containing the word "`yes`", if the soldiers can be moved such that during the move enough military presence is displayed along every path and after moving every location is well defended, and the word "`no`" otherwise.

```cpp
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <algorithm>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
                       property<edge_capacity_t, long,
                       property<edge_residual_capacity_t, long,
                       property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor Edge;

class Location {
public:
    int have;
    int need;
    Location(int have, int need):
        have(have),
        need(need)
    { }
};

void testcase() {
    int l;
    int p;
    cin >> l >> p;

    Graph g(l+2);
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);
    int source_edge = l;
    int sink_edge = l+1;

    int overall_need = 0;

    vector<Location> locations;
    for (int i=0; i<l; i++) {
        int g;
        int d;
        cin >> g >> d;
        locations.push_back(Location(g,d));
    }

    for (int i=0; i<p; i++) {
        int a;
        int b;
        int cmin;
        int cmax;

        cin >> a >> b >> cmin >> cmax;
        // adapt locations
        locations[a].need += cmin;
        locations[b].need -= cmin;

        // add delta as flow
        Edge e, rev_e;
        bool success;
        // Set edge from source to match
        tie(e, success) = add_edge(a, b, g);
        tie(rev_e, success) = add_edge(b, a, g);
        capacity[e] = cmax-cmin;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
```

```
    }

    // Add have to graph
    for (int i=0; i<l; i++) {
        Edge e, rev_e;
        bool success;
        // Set edge from source to match
        tie(e, success) = add_edge(source_edge, i, g);
        tie(rev_e, success) = add_edge(i, source_edge, g);
        capacity[e] = locations[i].have;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
    }
    // Add need to graph
    for (int i=0; i<l; i++) {
        if (locations[i].need < 0) {
            locations[i].need = 0;
            continue;
        }
        Edge e, rev_e;
        bool success;
        // Set edge from source to match
        tie(e, success) = add_edge(i, sink_edge, g);
        tie(rev_e, success) = add_edge(sink_edge, i, g);
        capacity[e] = locations[i].need;
        capacity[rev_e] = 0;
        rev_edge[e] = rev_e;
        rev_edge[rev_e] = e;
        overall_need += locations[i].need;
    }

    int max = push_relabel_max_flow(g, source_edge, sink_edge);
    // use sum_points to check if we got the maximum and all edges are used
    if (max >= overall_need) {
        cout << "yes" << endl;
        return;
    }
    cout << "no" << endl;
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
//        testcase(grid);
        testcase();
    }
    return 0;
}
```

./week_5_flows/2_defence/defence/main.cpp

Part VI

Week 6 - Matchings

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week III

Algorithms Lab

Exercise 1 – *Buddy Selection*

At the beginning Mathilda was very enthusiastic about her school's new buddy program. Students were to be grouped into pairs of buddies that were supposed to help each other out in their studies. As one of the first, Mathilda composed a short profile of herself describing her hobbies etc. The school director Dr. Fuzzman intended to use these profiles to find a suitable partner for every student. Alas, when Mathilda got the profile of her assigned partner she was very disappointed: Her partner was to be male, like soccer and be crazy about military gear. In short, her partner was the direct opposite of herself.

Mathilda went straight to Dr. Fuzzman to complain about the assignment. However he could not understand her discontent as Mathilda's partner liked math and chemistry just like Mathilda and enjoyed watching tv, just like her. Nodding to a huge stack student profiles he said with a mean smile: "I can give you the profiles of everybody and you will see that no better pairing is possible – it is a chance that everybody can be paired at all!". You have to help Mathilda.

Problem Given a list of students and several characteristics per student, check whether there is a buddy assignment in which partners share more common characteristics than in the solution of Dr. Fuzzman.

Input The first line of input contains the number of testcases. Every testcase starts with a single line containing three integers n, c, f separated by single spaces: $2 \leq n \leq 400, n \equiv 0 \bmod 2$ is the number of students, $c \leq 10$ is the number of characteristics per student and $f \geq 1$ is the minimum number of common characteristics over all pairs of buddies in Dr. Fuzzbrain's solution. The description of the n students follows. Each student is described by a single line containing c space separated keywords. Every keyword describes a characteristic. Keywords consist of less than 20 lowercase letters and no two keywords for one student are the same.

Output For every testcase, if there is a buddy assignment in which all pairs of buddies share more than f characteristics, output "`not optimal`" otherwise output "`optimal`". Two characteristics are considered equal only if they have identical keywords.

Sample Input

```
2
6 2 1
neat tv
tv soccer
neat soccer
soccer tv
neat soccer
```

```cpp
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

#include <algorithm>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       property<vertex_index_t, int>,
                       property<edge_index_t, int>
                      > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor VertexDescriptor;


int characterisitics_map[400][10];
int characterisitics_student[400][400];

inline int get_characteristics_index(vector<string>& characteristics, string& c) {
    int i;
    for (i=0; i<characteristics.size();i++) {
        if (c.compare(characteristics[i]) == 0) {
            return i;
        }
    }
    characteristics.push_back(c);
    return i;
}

void testcase() {
    int n, c, f;
    cin >> n >> c >> f;

//    cout << "Reading" << endl;
    vector<string> characteristics;
    for (int i=0; i<n; i++) {
        for (int j=0; j<c; j++) {
            string s;
            cin >> s;
            int idx = get_characteristics_index(characteristics, s);
            characterisitics_map[i][j] = idx;
        }
    }

//    cout << "arMap" << endl;
    fill(&characterisitics_student[0][0], &characterisitics_student[399][399], 0);

    for (int x=0; x<n; x++) {
        for (int i=0; i<c; i++) {
            int ci = characterisitics_map[x][i];
            for (int y=0; y<n; y++) {
                if (x == y) {
                    continue;
                }
                for (int j=0; j<c; j++) {
                    int cj = characterisitics_map[y][j];
                    if (ci == cj) {
                        characterisitics_student[x][y]++;
//                        characterisitics_student[y][x]++;
                        break;
                    }
                }
            }
        }
    }

    Graph g(n);
    for (int x=0; x<n; x++) {
        for (int y=x+1; y<n; y++) {
```

```
75              if (characterisitics_student[x][y] > f) {
                    Edge e;
77                  bool success;
                    tie(e, success) = add_edge(x,y,g);
79              }
            }
81      }
        vector<VertexDescriptor> mate(n);
83      edmonds_maximum_cardinality_matching(g, &mate[0]);
        const VertexDescriptor NULL_VERTEX = graph_traits<Graph>::null_vertex();
85      int m=0;
        for(int i = 0; i < n; ++i) {
87          if(mate[i] == NULL_VERTEX) {
//              cout << i << " -- " << mate[i] << endl;
89  //            m++;
                cout << "optimal" << endl;
91              return;
            }
93      }
        cout << "not optimal" << endl;
95  }

97  int main()
    {
99      cin.sync_with_stdio(false);
        cout.sync_with_stdio(false);
101     int testcases;
        cin >> testcases;
103
        for (int i=0; i<testcases; i++) {
105         testcase();
        }
107     return 0;
    }
```

./week_6_matchings/0_buddies/buddies/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week III

Algorithms Lab

Exercise 2 – *Fluid Borders*

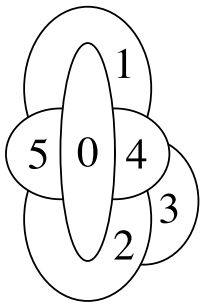
The distant planet Li-Kuid III hosts life – but not as we know it! Li-Kuid III is inhabited by gelatinous spheres, so-called "blobs" (beings that link to organized basins). Multiple blobs form communities by fusing together into a big gelatinous meta-blob. The surface of Li-Kuid III is covered by many such meta-blobs of various sizes. Meta-blobs are like earth countries and just like on earth it is very important for a meta-blob who its neighbors (other meta-blobs sharing a nonzero length portion of the meta-blob's boundary) are. However, meta-blobs are much less rigid than earth countries – in fact, once in a Li-Kuid III year, every meta-blob has a big election in which every individual blob votes for a preferred neighbor. Depending on the outcome of the different elections, the planets surface is completely redistributed amongst the meta-blobs in order to satisfy the wishes of the individual blobs as much as possible. After the reformation, the surface of the planet usually looks completely different as blobs are quickly fed up with their current neighbors – however, no blob ever changes the meta-blob it lives in.

Due to increasing population, in recent years the process of deciding how to reform the borders has become more and more involved. Your help is required!

Problem Given the election outcomes of every meta-blob, determine how good the votes can be implemented. The goal is to find the maximum threshold T such that the T most requested neighbors of every meta-blob can be established. Evey blob can move around arbitrarily, however once the new borders are established, all inhabitants of a meta-blob must be connected, i.e. no meta-blob may be split. In addition, different meta-blobs may not overlap.

The following table is one example for election outcomes, the figure next to it gives a realizing layout for $T = 2$.

meta-blob	election oucome 1st	2nd	3rd	4th	5th
0	5	4	3	1	2
1	0	4	5	3	2
2	4	0	3	1	5
3	2	4	1	0	5
4	3	2	0	1	5
5	0	1	2	4	3



Input The first line of input contains the number of test cases n. Each test case begins with a line containing the number $0 < m \leq 200$ of meta-blobs; m lines describing the election results of the different meta-blobs follow. The i-th line contains $m - 1$ zero based, space separated indices $0 \leq p_{ij} < m$ ordered descending by the number of votes each other meta-blob got in the election of meta-blob i (i.e. p_{i0} is the index of the meta-blob that got the most votes).

```cpp
#include <iostream>

#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       property<vertex_index_t, int>,
                       property<edge_index_t, int>
                      > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor Vertex;

int election_results[201][201];
void testcase() {
    int m;
    cin >> m;
    int n=m-1;
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            cin >> election_results[i][j];
        }
    }
    if (m==1) {
        cout << 0 << endl;
        return;
    }
    Graph g(m);
    int k;
    for (k=0; k<n; k++) {
        for (int i=0; i<m; i++) {
            Edge e;
            bool success;
            tie(e, success) = add_edge(i, election_results[i][k], g);
        }
        bool is_planar = boyer_myrvold_planarity_test(boyer_myrvold_params::graph = g);
        if (!is_planar) {
            cout << (k) << endl;
            return;
        }
    }
    cout << (n) << endl;
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_6_matchings/1_borders/borders/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week III

Algorithms Lab

Exercise 3 – *Satellites*

Euclidean Satellite Academy (ESA) has satellites in orbit and radio stations on the ground. Each satellite is connected to a few ground stations by wireless links. ESA plans to monitor the performance of the wireless links. It does this with software that can be installed on a satellite or a ground station and monitors all links connected to the entity. Where should they install the monitoring software if they want to install it in as few places as possible?

Given a set of satellites, ground stations, and the links between them, decide which nodes in the system (by nodes we mean satellites and ground stations) must install the monitoring software if all links need to be monitored by at least one of their end-points but the number of nodes with the software should be minimized. If there are multiple solutions, output any single one of them.

Input First line of input contains $1 \leq t \leq 100$, the number of test cases. Each test case starts with three numbers: $0 \leq g \leq 100$, $0 \leq s \leq 100$, and $0 \leq l \leq 10000$, denoting the number of ground stations, satellites, and links respectively. Ground stations are given labels from 0 to $g-1$ and satellites from 0 to $s-1$. The following l pairs of numbers denote the (ground station, satellite) pairs defining each link.

Output For each testcase output two lines. The first one should contain two integers, g' and s' denoting respectively the number of ground stations and satellites that will have the monitoring system installed, followed by g' numbers with ground station IDs and s' numbers with satellite IDs on the second line.

Sample Input

```
2
3 3 5
0 1
1 0
1 2
2 1
2 2
5 4 8
0 0
1 0
2 0
3 0
4 0
3 1
3 2
3 3
```

Sample Output

```
3 0
0 1 2
1 1
3 0
```

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

#include <algorithm>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       property<vertex_index_t, int>,
                       property<edge_index_t, int>
                      > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor VertexDescriptor;

const VertexDescriptor NULL_VERTEX = graph_traits<Graph>::null_vertex();

void coverPath(Graph& g, int current, vector<VertexDescriptor>& mate, vector<bool>& covered, vector<
    vector<int> >& connections) {
    for (vector<int>::iterator it=connections[current].begin(); it != connections[current].end(); it
        ++) {
        int nb = *it;
        covered[nb] = true;
        if (mate[nb] != NULL_VERTEX) {
            int ground_node = mate[nb];
            if (!covered[ground_node]) {
                covered[ground_node] = true;
                coverPath(g, ground_node, mate, covered, connections);
            }
        }
    }
}

void testcase() {
    int num_g, num_s, l;
    cin >> num_g >> num_s >> l;

    int total_vertices = num_g+num_s;
    Graph graph(total_vertices);

    vector<vector<int> > connections(total_vertices, vector<int>());
    for (int i=0; i<l; i++) {
        int gi, si;
        cin >> gi >> si;
        Edge e;
        bool success;
        tie(e, success) = add_edge(gi,num_g+si, graph);
        connections[gi].push_back(num_g+si);
        connections[num_g+si].push_back(num_g);
    }

    vector<VertexDescriptor> mate(total_vertices);
    vector<bool> covered(total_vertices,false);
    vector<bool> unmatched(total_vertices, false);

    edmonds_maximum_cardinality_matching(graph, &mate[0]);
    int m=0;
    for(int i = 0; i < num_g; ++i) {
        if(mate[i] == NULL_VERTEX) {
            ////              cout << i << " -- " << mate[i] << endl;
            ////              m++;
            //              cout << "optimal" << endl;
            //              return;
            covered[i]   = true;
            unmatched[i] = true;
        }
    }

    // make use of structure of the graph.
```

```
    for (int i=0; i < num_g; i++) {
        if (unmatched[i]) {
            coverPath(graph, i, mate, covered, connections);
        }
    }

    vector<int> out_grounds;
    vector<int> out_sattellite;
    for (int i=0; i<num_g; i++) {
        if (!covered[i]) {
            out_grounds.push_back(i);
        }
    }
    for (int i=num_g; i<total_vertices; i++) {
        if (covered[i]) {
            out_sattellite.push_back(i-num_g);
        }
    }
    cout << out_grounds.size() << " " << out_sattellite.size() << endl;
    for (vector<int>::iterator it=out_grounds.begin(); it!=out_grounds.end(); it++) {
        cout << *it << " ";
    }
    for (vector<int>::iterator it=out_sattellite.begin(); it!=out_sattellite.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;


}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_6_matchings/2_satellites/satellites/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week III

Algorithms Lab

Exercise 4 – *Tiles*

Domino Magic has been a producer of stylish domino game pieces for several years. Their exclusive designs and quality materials have always guaranteed them a safe position on the market. Recently however, due to the global financial crisis, the demand for luxurious domino games has drastically dropped. In order to stay afloat, the company has decided to expand into other areas. Their newest plan is to produce garden floor tiles using their exclusive domino designs. The problem is that these designs require rectangular tiles of a length to width ratio of 2:1. Many of Domino Magic's rich customers have complex garden layouts which they want to tile and it is often difficult to decide whether that is possible with 2:1 tiles.

Problem Write a program to help Domino Magic decide whether a given layout can be tiled with 2:1 tiles. Every tile covers exactly two horizontally or vertically adjacent spaces of the garden layout. Every space that is to be tiled must be covered by exactly one tile while the other spaces must remain untiled. The supply of tiles is unlimited.

Input The first line of input contains the number n of test cases to follow ($0 \leq n \leq 100$). Every test case starts with a single line containing two space-separated integers $1 \leq w, h \leq 50$, specifying the width and the height of the garden layout at hand, respectively. The next h lines each describe one row of the layout. Each such line consists of w characters: `'.'` if that space is to be tiled and `'x'` if it is to be left free.

Output For every test case output a single line containing the word "`yes`" if the given layout can be tiled and "`no`" otherwise.

Sample Input

```
2
4 4
x..x
....
....
x..x
4 3
x..x
..x.
....
```

Sample Output

```
yes
no
```

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

#include <algorithm>
#include <cassert>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
                       property<vertex_index_t, int>,
                       property<edge_index_t, int>
                      > Graph;

typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIterator;
typedef graph_traits<Graph>::vertex_descriptor VertexDescriptor;


bool tiles[52][52];
int get_index(int w, int i, int j) {
    assert(i>0);
    assert(j>0 && j <= w);
    return ((w)*(i-1))+j-1;
}

pair<int,int> get_coord(int w, int idx) {
    assert(idx >= 0);
    if (idx < w) {
//        cout << idx << ": " << idx+1 << " " << 1 << " (w: " << w << ")" <<endl;
        return make_pair(1,idx+1);
    }
    int j=idx%w;
    int i=((idx-j)/w);
//    cout << idx << ": " << i+1 << " " << j+1 << " (w: " << w << ")" <<endl;
    return make_pair(i+1,j+1);
}

void testcase() {
    int h, w;
    cin >> w >> h;
    for (int i=1; i<=h; i++) {
        for (int j=1; j<=w; j++) {
            char c;
            cin >> c;
            bool t = true;
            if (c == 'x') {
                t = false;
            }
            tiles[i][j] = t;
        }
    }
    for (int i=0; i<=w+1; i++) {
        tiles[0][i] = false;
        tiles[h+1][i] = false;
    }
    for (int i=1; i<=h; i++) {
        tiles[i][0] = false;
        tiles[i][w+1] = false;
    }
    int total_vertices = h*w;
    int num_active_tiles = 0;
    Graph g(total_vertices);
    for (int i=1; i<=h; i++) {
        for (int j=1; j<=w; j++) {
            if (!tiles[i][j]) {
                continue;
            }
            num_active_tiles++;
            int s = get_index(w, i, j);
            if (tiles[i][j+1]) {
```

```
                add_edge(s, get_index(w, i, j+1), g);
            }
            if (tiles[i][j-1]) {
                add_edge(s, get_index(w, i, j-1), g);
            }
            if (tiles[i+1][j]) {
                add_edge(s, get_index(w, i+1, j), g);
            }
            if (tiles[i-1][j]) {
                add_edge(s, get_index(w, i-1, j), g);
            }
        }
    }

    const VertexDescriptor NULL_VERTEX = graph_traits<Graph>::null_vertex();
    vector<VertexDescriptor> mate(total_vertices);
    edmonds_maximum_cardinality_matching(g, &mate[0]);
//    if (num_active_tiles%2 == 1) {
//        cout << endl;
//        for (int i=0; i<=h+1; i++) {
//            for (int j=0; j<=w+1; j++) {
//                if (tiles[i][j])
//                    cout << ".";
//                else
//                    cout << "x";
//            }
//            cout << endl;
//        }
//    }
    for (int i=0; i<total_vertices; i++) {
//        cout << "mate " << i << ": " << mate[i] << endl;
        if(mate[i] == NULL_VERTEX || mate[i] == i) {
            pair<int, int> cd = get_coord(w, i);
//            cout << i << ": " << cd.first << " " << cd.second << " (w: "<<w<<")" << endl;
            if (tiles[cd.first][cd.second]) {
                cout << "no" << endl;
                return;
            }
        }
//        if (num_active_tiles%2 == 1) {
//            pair<int, int> c = get_coord(w, i);
//            int l = mate[i];
//            pair<int, int> cl = get_coord(w, i);
//            cout << i << ":" << l << " " << c.first << " " << c.second << " has mate " << cl.first
    << " " << cl.second << endl;
//        }
    }
    cout << "yes" << endl;

}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    int testcases;
    cin >> testcases;

    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_6_matchings/3_tiles/tiles/main.cpp

Part VII

Week 7 - Mixed

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week IV

Algorithms Lab

Exercise 1 – *Knights*

After the Black Beast of Aaauugh has met its untimely demise in the middle of the cave of Caerbannog, the knights of the round table needed to leave the cave in order to continue their journey to the Castle of Uuggggggh.

As in any proper movie however, the cave was quickly collapsing; every hallway segment *and intersection* collapsed after a single knight ran through it. On the way to the Gorge of Eternal Peril, Sir Bedevere was wondering why only so few knights made it out of the cave alive and if they couldn't have ran through the hallways in such a way that more of them would have survived.

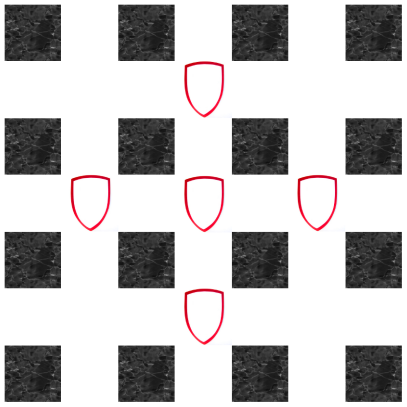


Figure 1: A situation with three vertical and three horizontal hallways and 5 knights depicted by shields. (Second sample input)

Problem Surprisingly, the cave of Caerbannog has a grid graph layout, where all the north-south (vertical when drawn on paper) hallways are numbered 0 to $m-1$ (from west to east), while the west-east (horizontal) hallways are numbered 0 to $n-1$ (from north to south). All hallways are connected to the outside on their ends. Knights start at the intersections given in the input file.

Every segment and intersection of a hallway that a knight passes through will collapse immediately behind him, in a movie-like fashion. Because two knights never arrive at an intersection at exactly the same time, every hallway intersection and hallway segment can only be used by a single knight. Note that the hallway intersection the knight starts in also collapses immediately. Find out how many knights can get to safety.

Input The first line of the input contains t, the number of test cases. The first line for each test case will contain three numbers: m, n – the dimensions of the cave, and k, the number of knights in the cave. The next line will contain $2k$ numbers, with the $2i$-th and $(2i+1)$-th numbers denoting the x (column) and y (row) coordinates of the starting position of the knight i in the cave. You can expect each value t, m, and n to be less than 50.

Output For every test case output a single line containing the maximum number of knights that can escape from the cave.

```cpp
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>

#include <algorithm>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Traits;

typedef adjacency_list<vecS, vecS, directedS, no_property,
                       property<edge_capacity_t, long,
                       property<edge_residual_capacity_t, long,
                       property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor Edge;

int m, n, k;
inline int get_index(int i, int j) {
    return m*j+i;
}

void madd_edge(Graph& g,
                    EdgeCapacityMap& capacity,
                    ReverseEdgeMap& rev_edge,
                    ResidualCapacityMap& res_capacity,
                    int w, int src, int end) {
    Edge e, rev_e;
    bool success;
    // Create edge from source to player
    tie(e, success) = add_edge(src, end, g);
    tie(rev_e, success) = add_edge(end, src, g);
    capacity[e] = w;
    capacity[rev_e] = 0;
    rev_edge[e] = rev_e;
    rev_edge[rev_e] = e;
}

void testcase() {
    cin >> m >> n >> k;
    int source_edge = 2*m*n;
    int sink_edge = 2*m*n+1;
    int offset = m*n;
    Graph g(2*m*n+2);
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap rev_edge = get(edge_reverse, g);
    ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);

    // Connect left and right outgoing edges to sink
    for (int i=0; i<m; i++) {
        int src;
        // left
        src = get_index(i, 0) + offset;
        madd_edge(g, capacity, rev_edge, res_capacity, 1, src, sink_edge);

        // right
        src = get_index(i,n-1) + offset;
        madd_edge(g, capacity, rev_edge, res_capacity, 1, src, sink_edge);
    }

    // Connect upper and lower edges to sink
    for (int j=0; j<n; j++) {
        int src;

        // upper
        src = get_index(0, j)+offset;
        madd_edge(g, capacity, rev_edge, res_capacity, 1, src, sink_edge);

```

```
        // lower
        src = get_index(m-1, j)+offset;
        madd_edge(g, capacity, rev_edge, res_capacity, 1, src, sink_edge);
    }

    // Insert knights
    for (int i=0; i<k; i++) {
        int x, y;
        cin >> x >> y;

        int end = get_index(x, y);
        // Create edge from edge to sink
        madd_edge(g, capacity, rev_edge, res_capacity, 1, source_edge, end);

    }

    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            int a, b;
            int c, d;
            a = get_index(i, j);
            c = a+offset;
            madd_edge(g, capacity, rev_edge, res_capacity, 1, a, c);

            if (i+1 < m) {
                b = get_index(i+1, j);
                d = b + offset;

                madd_edge(g, capacity, rev_edge, res_capacity, 1, c, b);
                madd_edge(g, capacity, rev_edge, res_capacity, 1, d, a);
            }
            if (j+1 < n) {
                b = get_index(i, j+1);
                d = b + offset;

                madd_edge(g, capacity, rev_edge, res_capacity, 1, c, b);
                madd_edge(g, capacity, rev_edge, res_capacity, 1, d, a);
            }
        }
    }


    int max = push_relabel_max_flow(g, source_edge, sink_edge);
    cout << max << endl;
//    long flow = edmonds_karp_max_flow(g, source_edge, sink_edge);
    //capacity, res_capacity, rev_edge, &end_color[0], &pred[0]);
//    cout << flow << endl;
}

int main()
{
    int testcases;
    cin >> testcases;
    for (int i=0; i<testcases; i++) {
        testcase();
    }
    return 0;
}
```

./week_7_mixed/0_knights/knights/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week IV

Algorithms Lab

Exercise 2 – *Shopping Trip*

Rick needs to do the shopping for the whole week this afternoon. He wants to visit a number of stores, but he wants to go home after visiting each one to drop off what he bought. Aditionally, he does not want to use the same street twice while visiting the different stores, lest it will be too boring. He is now looking at the map and wondering if it is possible at all. He clearly needs your help.

Problem You are given a map of the city specified as a graph with street intersections as vertices and streets as edges. Aditionally you know the locations of the stores Rick wants to visit. Stores are always located on street intersections. Find out if it is possible for Rick to visit all stores and return home between each successive visits. He can use the same street while going to one store and returning from the same store, but not while on the way to or from different stores. Through intersections he can go as many times as he wants. Also note that two streets can cross without intersecting as there are over- and under-passes.

Input The first line of the input contains T, the number of the test cases. Each of T subsequent lines will contain one testcase. A testcase starts with n, m and s, the numbers of vertices (intersections), edges (streets) and stores that need to be visited respectively. Following this come s numbers specifying the store locations given by intersection numbers. The following $2m$ numbers specify which intersections are connected by streets, $2i$-th and $2i + 1$-st numbers specifing end points of i-th street. Rick's house is always at intersection 0 (the first intersection). You can expect n to be less than 500.

Output The output should contain a single line for every test case. The line should contain single word "yes" or "no", depending on whether it is, or is not, possible to solve the problem for that test case.

Sample Input

```
2
6 7 3 3 4 5 0 1 0 2 0 3 1 3 2 3 3 4 3 5
4 4 3 1 2 3 0 1 0 2 2 3 1 3
```

Sample Output

```
yes
no
```

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week IV

Algorithms Lab

Exercise 3 – *Sweepers*

After a long day of doing magic, the corridors of the Unseen University (UU) of Ankh-Morpork need to be cleaned to get rid of the banana peels, billiard balls, dead pigeons, disoriented rabbits, half-eaten cupcakes, burnt flower bouquets, and a malfunctioning stone for a catapult.

To do this, UU hired a few sweepers, and some wizard teleported them to random locations in the University. Since they are pretty terrified of the place, they would like to sweep the corridors clean as quickly as possible and get out. They have been initially placed in rooms and they need to clean all the corridors of the building they are in, but they don't want to sweep any corridor twice. They need your help to tell them if it is even possible.

Problem UU has rooms labeled 0 to $n-1$. The rooms are connected by corridors, a corridor always connects two rooms. Some rooms have a door leading outside. Sweepers start in rooms given in the input. They want to sweep each corridor of the building(s) but they don't want to sweep or otherwise traverse any corridor twice. They can cross a room any number of times. Corridors are swept by a simple traversal by the sweeper. When done sweeping, each sweeper must leave the building through one of the doors that lead outside. Each door can be only used by one sweeper and there is an equal number of sweepers and doors. However note that more sweepers can start in the same room and each room can have several doors.

Input The first line of the input contains t, the number of test cases. The first line for each test case will contain three numbers: n, the number of rooms of the UU, m, the number of corridors in UU, s, the number of sweepers and at the same time the number of rooms with doors leading outside. The next line will contain $2s$ numbers: the first s numbers denoting the starting locations of the sweepers, the second s numbers denoting the room numbers of the s doors leading to the outside. The line after that will contain $2m$ numbers, with the $2i$-th and $(2i+1)$-th numbers denoting the end points of the i-th corridor. You can expect the value of s, t, and n to be at least 0 and at most 100, and that $0 \leq m < n^2$.

Output For every test case output a single line containing the word "yes" if the corridors can be swept and "no" if they cannot.

Sample Input

```
2
6 7 1
1 4
0 1 1 2 1 4 0 3 3 4 4 5 2 5
6 7 2
1 3 4 4
0 1 1 2 1 4 0 3 3 4 4 5 2 5
```

Sample Output

```
yes
no
```

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet BGL Week IV

Algorithms Lab

Exercise 4 – *Cantonal Courier*

To earn money for the journey of your dreams (Barcelona, Gothenburg and London) you come up with an idea for a shipping company. You are going to have an employee (a courier) in each canton of Algoland. The caveat that is going to let you offer competitive rates while promoting sustainability is that your couriers are going to use public transport. The distinctive feature of Algoland's public transport system is that each canton consists of several zones – depending on the route you need to buy tickets for some subset of those zones.

For each canton you are given a list of possible assignments and the reward for each of them that a customer will pay if you agree to take it. For each assignment you are also given the list of zones for which you need a ticket in case you take it. Finally, you are given the price of the ticket for each zone. All the tickets are day passes and your couriers are truly excellent, so you can be sure that once bought, a single ticket can be reused for several jobs.

Find the optimal profit (payment from jobs minus costs of tickets) you can achieve for each canton.

Input The first line of the input contains $1 \leq T \leq 100$, the number of testcases.

Each testcase describes a single canton and starts with a line holding two integers: $1 \leq Z, J \leq 100$, where Z is a number of zones and J number of jobs. In the next line there are Z integers $c_1, \ldots, c_Z, 1 \leq c_i \leq 5000$, where c_i is the cost of the ticket for zone i. The third line contains J integers $p_1, \ldots, p_J, 1 \leq p_i \leq 5000$, where p_i is the reward for job i.

J lines follow: i-th of those lines describes the tickets needed for job i. Each of those lines starts with $0 \leq N_i \leq Z$, followed by a strictly increasing sequence of N_i zones (1-based) for which the tickets are needed.

All consecutive numbers in a line are single-space separated.

Output For each testcase output a single line with an integer: your profit given optimum choice of jobs.

Sample input

```
2
4 3
1 5 6 7
3 4 10
2 1 2
1 2
2 3 4
3 3
3 3 3
4 4 4
3 1 2 3
3 1 2 3
3 1 2 3
```

Sample output

```
1
3
```

Part VIII

Week 8 - CGAL

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week I

Algorithms Lab

Exercise 1 – *Hit?*

A laser source shot Phileas Photon into some direction. Will he stop at an obstacle or travel to infinity?

The scenery is considered as seen from above, such that obstacle walls appear as line segments and the trajectory of Phileas is described by a ray.

Input The input file consists of several test cases. Each of them starts with a line containing one integer n ($1 \leq n \leq 20'000$). The following line describes the ray along which Phileas travels. It contains integers `x y a b` where (x, y) are the coordinates of the starting point of the ray and $(a, b) \neq (x, y)$ is another point on the ray. The following n lines describe an obstacle line segment each. The i-th of these lines contains four integers `r s t u`, where (r, s) and (t, u) are the endpoints of the i-th segment. All the above coordinates are integers that are in absolute value smaller than 2^{51}. In particular, you cannot represent them using a 32-bit integer data type in general. All numbers on a single line are separated by a single space. The input is terminated by a single line containing `0` (i.e., an empty testcase).

Output The output for each testcase appears on a separate line. The line consists of the word `yes`, if the ray hits an obstacle[1] and `no`, otherwise.

Sample Input

```
1
0 0 1 1
2 0 1 2
1
1 1 0 0
0 -2 -1 0
2
0 0 1 1
-1 -2 -1 0
2 0 2 1
2
0 1 1125899906842623 1125899906842623
1 2 1 3
1125899906842621 1125899906842620 1125899906842621 1125899906842621
3
1125899906842623 1125899906842623 0 1
1125899906842621 1125899906842620 1125899906842621 1125899906842621
1 2 1 3
-1 0 -1 1
0
```

[1] The obstacles segments are relatively closed, that is, both endpoints are included into consideration.

```cpp
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

void testcases() {
    while(true) {
        int n;
        cin >> n;
        if (n == 0) {
            break;
        }
        double lx, ly, rx, ry;
        cin >> lx >> ly >> rx >> ry;
        K::Ray_2 r(K::Point_2(lx, ly), K::Point_2(rx, ry));
        bool intersected = false;
        for (int i=0; i < n; i++) {
            cin >> lx >> ly >> rx >> ry;
            if (!intersected) {
                K::Segment_2 s(K::Point_2(lx, ly), K::Point_2(rx, ry));
                if (CGAL::do_intersect(r,s)) {
                    intersected = true;
                }
            }
        }
        if (intersected) {
            cout << "yes" << endl;
        } else {
            cout << "no" << endl;
        }
    }
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_8_cgal/0_hit/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week I

Algorithms Lab

Exercise 2 – *Antenna*

After the invention of radio, Theirland wants to demonstrate its technological superiority and builds a first radio transmitter. The transmitter must cover the whole population. It is characterized by a location and a transmission radius (within which a reception of the signal is guaranteed). Not surprisingly, transmitters with a higher radius require more advanced technology and more time to build and—last but not least—they cost much more. Thus, the government decided to find a location where the transmission radius is as small as possible, but every single citizen can receive the signal at home. This is not an easy goal to achieve, though...

Input The input contains several test cases. Each of them begins with a line containing one integer n ($1 \leq n \leq 200'000$), denoting the number of citizens. The next n lines contain coordinates x_i y_i of homes of citizens (x_i, y_i integral with $|x_i|, |y_i| < 2^{48}$). All numbers on a single line are separated by a single space. The input is terminated by a single line containing `0` (i.e., an empty testcase).

Output For each input, write on a single line the smallest integral transmission radius needed to cover all citizens.

Sample Input

```
2
1 7
31 -6
5
0 0
1 0
2 0
3 0
4 0
0
```

Sample Output

```
17
2
```

```cpp
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>
#include <vector>
#include <cmath>

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K>  Min_Circle_Traits;
typedef  CGAL::Min_circle_2<Min_Circle_Traits> Min_Circle;
using namespace std;
double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}
double ceil_to_double(const K::FT& x)
{
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

void testcases() {
    vector<K::Point_2> points;
    cout << fixed << setprecision(0);
    while(true) {
        int n;
        cin >> n;

        if (n <= 0)
            break;

        points.reserve(n);
        for (int i=0; i <n; i++) {
//            K::Point_2 p;
//            cin >> p;
            double x, y;
            cin >>x >> y;
//            p(x,y);
            K::Point_2 p(x, y);
            points.push_back(p);
        }

        Min_Circle mc(points.begin(), points.begin()+n, true);
        Min_Circle_Traits::Circle c = mc.circle();
        K::FT r = sqrt(c.squared_radius()); // you need
            Exact_predicates_exact_constructions_kernel_with_sqrt
        cout << ceil_to_double(r) << endl;
        points.clear();
    }
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_8_cgal/1_antenna/antenna/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week I

Algorithms Lab

Exercise 3 – *First hit*

A laser source shot Phileas Photon into some direction. How far will he have to travel? Where will he end up?

The scenery is considered as seen from above, such that obstacle walls appear as line segments and the trajectory of Phileas is described by a ray.

Input The input file consists of several test cases. Each of them starts with a line containing one integer n ($1 \leq n \leq 30'000$). The following line describes the ray along which Phileas travels. It contains integers `x y a b` where (x, y) are the coordinates of the starting point of the ray and $(a, b) \neq (x, y)$ is another point on the ray. The following n lines describe an obstacle line segment each. The i-th of these lines contains four integers `r s t u`, where (r, s) and (t, u) are the endpoints of the i-th segment. All the above coordinates are integers that are in absolute value smaller than 2^{51}. All numbers on a line are separated by a single space. The input is terminated by a single line containing `0` (i.e., an empty testcase).

Output The output for each testcase appears on a separate line. This line contains the coordinates of the first intersection of the ray with any obstacle segment[1], where both coordinates are rounded[2] down to the next integer. If there is no such intersection, the line consists of the word `no`.

Sample Input

```
1
0 0 1 1
2 0 1 2
1
1 1 0 0
0 -2 -1 0
2
0 0 1 1
-1 -2 -1 0
2 0 2 1
2
0 1 1125899906842623 1125899906842623
1 2 1 3
1125899906842621 1125899906842620 1125899906842621 1125899906842621
3
1125899906842623 1125899906842623 0 1
1125899906842621 1125899906842620 1125899906842621 1125899906842621
```

[1] The obstacles segments are relatively closed, that is, both endpoints are included into consideration.
[2] That is, for a coordinate a output the unique integer i, for which $i \leq a < i + 1$.

```cpp
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>


#include <iostream>
#include <vector>
#include <algorithm>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}
double ceil_to_double(const K::FT& x)
{
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

void shorten_segment(K::Segment_2 &s, CGAL::Object o) {
    if (const K::Point_2 *p = CGAL::object_cast<K::Point_2>(&o)) {
        s = K::Segment_2(s.source(), *p);
    } else if (const K::Segment_2 *t = CGAL::object_cast<K::Segment_2>(&o)) {
        // select endpoint of *t closer to s.source()
        if (CGAL::collinear_are_ordered_along_line(s.source(), t->source(), t->target())) {
            s = K::Segment_2(s.source(), t->source());
        } else {
          s = K::Segment_2(s.source(), t->target());
        }
    } else {
        throw runtime_error("Strange Error");
    }
}

void testcases() {
//    cout << fixed << setprecision(0);
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);

    while(true) {
        size_t n;
        cin >> n;

        if (n == 0)
            break;

        double ix, iy, jx, jy;
//        cin >> ix >> iy >> jx >> jy;
//        K::Ray_2 r(K::Point_2(ix, iy), K::Point_2(jx, jy));
        K::Ray_2 r;
        cin >> r;
        K::Segment_2 rc(r.source(), r.source());

        vector<K::Segment_2> segments;
        segments.reserve(n);

        size_t i = 0;
        for (; i<n; ++i) {
            // We are losing accuracy
            cin >> ix >> iy >> jx >> jy;
            segments.push_back(K::Segment_2(K::Point_2(ix, iy), K::Point_2(jx, jy)));
//            K::Segment_2 ss;
//            cin >> ss;
//            segments.push_back(ss);
        }
//        random_shuffle(segments.begin(), segments.end());

        for (i=0; i<n; ++i) {
            if (CGAL::do_intersect(segments[i], r)) {
```

```
                shorten_segment(rc, CGAL::intersection(r, segments[i]));
                break;
            }
        }
        if (i >= n) {
            cout << "no" << endl;
            continue;
        }
        while (++i < n) {
            if (CGAL::do_intersect(segments[i], rc)) {
                shorten_segment(rc, CGAL::intersection(r, segments[i]));
            }
        }
        cout << floor_to_double(rc.target().x()) << " "
             << floor_to_double(rc.target().y()) << endl;
        segments.clear();
    }
}

int main()
{
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_8_cgal/2_first_hit/first_hit/main.cpp

Part IX

Week 9 - Proximity Sources

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week II

Algorithms Lab

Exercise 1 – *Graypes*

The forests of Grayland are inhabited by a unique species developed from apes, called graypes. Apart from many biological differences, which have been puzzling zoologists during recent years, graypes also exhibit a very specific social behavior. If a graype is in a difficult situation or if it is afraid of something, it seeks company of another graype. In other words, it simply runs toward the nearest graype so they can solve the problem or fight the danger together. For a long time scientists have been monitoring the migration of graypes caused by this phenomenon.

One day, an earthquake struck Grayland. This caused a huge chaos and massive migration of graypes from one place to another. Moni Torer, a scientist at GPL, the Graype Positioning Lab, observed that all graypes ran at the same speed of $1\mathrm{m/sec}$.. Soon after, the lab members started betting at which time the first graype will reach its company...

Input The input contains several test cases. Each of them begins with a line containing one integer n ($2 \leq n \leq 100'000$), denoting the number of graypes. The next n lines describe the current position of all graypes, measured in meter. Each position is defined by two integer coordinates x, and y, separated by space, with $|x|, |y| < 2^{25}$. You may assume that the input points in each test case are unique, that is, no two graypes are at the same position initially. If there is a graype that has several closest graypes, then it runs towards the one among those that is located at a lexicographically smallest position (that is, a position with smallest x-coordinate and among those that have the same x-coordinate, the one with a smallest y-coordinate). The input is terminated by a line containing a single value 0.

Output For each input, the output is a single integer on a separate line. The output is the time in hundredth of a second needed for the first graype to reach another graype (i.e., until two graypes occupy the same position), rounded up to the next integer.

Sample Input

```
3
0 0
-100 -100
3536 3536
2
0 0
3535 3535
0
```

Sample Output

```
7072
249963
```

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <iostream>
#include <vector>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_edges_iterator  Edge_iterator;

using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

void testcases() {
    size_t n;
    std::vector<Triangulation::Point> points;
    while (cin >> n && n > 0) {
        points.reserve(n);
        for (size_t i = 0; i < n; i++) {
            Triangulation::Point p;
            cin >> p;
            points.push_back(p);
        }

        Triangulation t;
        t.insert(points.begin(), points.end());

        double distance = -1;
        for (Edge_iterator it = t.finite_edges_begin(); it != t.finite_edges_end(); it++) {
//            K::Ft d = CGAL::squared_distance()
//            K::FT d = CGAL::squ
            Triangulation::Segment seg = t.segment( it );

            Triangulation::Point p0 = seg.point(0);
            Triangulation::Point p1 = seg.point(1);
            double d = CGAL::to_double(CGAL::squared_distance(p0, p1));
//            cout << d << endl;
            if (d < distance || distance < 0.0) {
                distance = d;
            }
        }

        cout << ceil(sqrt(distance)/2.0*100.0) << endl;
        points.clear();
    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_9_proximity_sources/0_graypes/graypes/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week II

Algorithms Lab

Exercise 2 – *Bistro*

Sunbunny is a chain of bistros that provide among the finest snacks available in the country of Sealand. These places have become so popular that at lunch and dinner prime time one can regularly find long queues of people waiting to be served. Due to the continuing success the head office started to evaluate possible locations where to open another restaurant. One important selection criterion is that the new location must not be too close to an existing Sunbunny restaurant...

Input The input contains several test cases. Each of them begins with a line containing one integer n ($1 \leq n \leq 110'000$), denoting the number of existing restaurants. The next n lines describe the location x_i y_i of these restaurants (x_i, y_i integers, $|x_i|, |y_i| < 2^{24}$). It follows the number m ($1 \leq m \leq 110'000$) of possible locations for the new restaurant, and on the next m lines the coordinates of these locations, in the same format as for the existing restaurants. The input is terminated by a line containing a single number 0.

Output For each possible location for the new restaurant, write on a single line the squared (Euclidean) distance to the closest existing restaurant.

Sample Input

```
4
0 0
0 100
100 0
200 200
2
100 100
300 300
0
```

Sample Output

```
10000
20000
```

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <iostream>
#include <vector>
#include <cmath>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_edges_iterator  Edge_iterator;

using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

void testcases() {
    size_t n;
    std::vector<Triangulation::Point> points;
    while (cin >> n && n > 0) {
        points.reserve(n);
        for (size_t i = 0; i < n; i++) {
            Triangulation::Point p;
            cin >> p;
            points.push_back(p);
        }

        Triangulation t;
        t.insert(points.begin(), points.end());

        size_t nres;
        cin >> nres;
        for (size_t i=0; i<nres; i++) {
            Triangulation::Point p;
            cin >> p;
            Triangulation::Point v = t.nearest_vertex(p)->point();
            double d = CGAL::to_double(CGAL::squared_distance(p, v));
            cout << d << endl;
        }
        points.clear();
    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_9_proximity_sources/1_bistro/bistro/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week II

Algorithms Lab

Exercise 3 – *H1N1*

In times of an epidemy, it is important to stick to strict hygienic rules and to keep distance to infected individuals.

An influential pharmaceutical company has decided to develop a groundbreaking device to prevent its users from coming into contact with infected people. You are asked to develop part of the software for its central server. Your program has coordinates of all known infected individuals at its disposal. It is supposed to answer queries whether a given user of the device is able to escape the contaminated area, assuming that all infected individuals stay at their current location. In order to escape, the user must be able to get arbitrarily far from all infected people, without ever coming closer than a given distance d to any of them.

This should not be too hard to do, should it?

Input We consider the situation as seen from above, such that people appear as points in the plane. The input contains several test cases. Each of them is described as follows.

- It starts with a line that contains a single integer n ($2 \leq n \leq 10^5$), denoting the number of infected people in the area.
- The next n lines contain their integer coordinates `x y` separated by space and so that $|x|, |y| < 2^{24}$.
- The following line contains a single integer m ($1 \leq m \leq 10^5$) denoting the number of users asking for an escape route. These are to be checked one by one separately, that is, they should be considered separate queries. None of them is infected.
- Finally, each of the subsequent m lines describes a user who wants to escape. Each user is specified by three integer coordinates x, y, and d, separated by space and such that $|x|, |y| < 2^{24}$ and $0 \leq d < 2^{49}$. The person is located at position (x, y) and wants to keep distance at least $\sqrt{d}$ from the infected (i.e., $\sqrt{d}$ is still allowed).

The input is terminated by a line containing a single value 0.

Output For every test case the corresponding output appears on a single line consisting of m characters "y" or "n" (one letter per query). Output the letter "y", if there is a way to escape, and "n", otherwise.

```cpp
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <iostream>
#include <vector>
#include <cmath>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_edges_iterator  Edge_iterator;
typedef Triangulation::Face_handle  Face_handle;
typedef Triangulation::Point  Point;
using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

bool dfs(Triangulation &t, double distance, Face_handle origin, map<Face_handle, int>& visitor_map,
    int cur_it) {
  for (int i=0; i<3; i++) {
        Triangulation::Segment s = t.segment(origin,i);
        double d = CGAL::to_double(s.squared_length());
        if (d >= distance) {
          Face_handle nb = origin->neighbor(i);
          if (visitor_map[nb] == cur_it) {
            continue;
          }

          if (t.is_infinite(nb)) {
            return true;
          }
          visitor_map[nb] = cur_it;
          if (dfs(t, distance, nb, visitor_map, cur_it)) {
            return true;
          }
        }
    }
    return false;
}

void testcases() {
    size_t n;
    std::vector<Triangulation::Point> points;
    while (cin >> n && n > 0) {
        points.reserve(n);
        for (size_t i = 0; i < n; i++) {
          double x, y;
          cin >> x >> y;
            Triangulation::Point p(x,y);
//            cin >> p;
            points.push_back(p);
        }

        Triangulation t;
        t.insert(points.begin(), points.end());
        map<Face_handle, int> visitor_map;
        for (Triangulation::Face_iterator it=t.finite_faces_begin(); it !=t.finite_faces_end(); it
            ++) {
            visitor_map[it] = -1;
        }

        size_t m;
        cin >> m;
        for (size_t i=0; i<m; i++) {
          double x, y;
          cin >> x >> y;
            Triangulation::Point p(x,y);
//            Triangulation::Point p;
//            cin >> p;
```

```
            double d;
            cin >> d;
            Triangulation::Point nv = t.nearest_vertex(p)->point();
            double distance2 = CGAL::to_double(CGAL::squared_distance(nv, p));
            if (distance2 < d) {
                cout << "n";
                continue;
            }
            Triangulation::Face_handle f = t.locate(p);

            visitor_map[f] = i;
            if (t.is_infinite(f) || dfs(t, d*4.0, f, visitor_map, i)) {
                cout << "y";
            } else {
                cout << "n";
            }

        }
        cout << endl;

        points.clear();
    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_9_proximity_sources/2_h1n1/h1n1/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week II

Algorithms Lab

Exercise 4 – *Germs*

Microbactus solitarius is a very interesting type of organism: it grows as long as it lives, but it dies immediately as soon as something impedes its growing. In Labland they plan to run experiments to find out how long these bacteria live under certain circumstances. However, the plans stalled as soon as the national ethics commission heard of it, because any experiments on living organisms come under very close scrutiny. As of yet it is unclear whether these experiments will be allowed eventually, and there is not even a timeframe known when such a decision will be made. Therefore, it is currently being considered whether computer simulations form a viable alternative. This is where you come into play...

The setup is as follows. An initial configuration of bacteria is placed in a rectangular dish. Bacteria are modeled as disks, whose initial radius is $0.5\mu m$. Each bacterium grows such that its center remains stationary and its radius grows quadratically, described by the function $\varrho(t) = t^2 + \frac{1}{2}$, where t is the time (measured in hours) that passed since the start of the simulation. For instance, after one hour the radius is $1.5\mu m$ and after two hours the radius is $4.5\mu m$. In the very moment a bacterium touches another bacterium or the outer boundary of the dish, it dies.

A dead bacterium stays where it is. In particular, it continues to be an obstacle to the growth (and life) of other bacteria. In principle, a dead bacterium would remain at the same size it had the moment it died. But for a first prototype of the simulation software, we consider a simplified model: We pretend that bacteria continue to grow, regardless of whether they are alive or not. In other words, in this model bacteria can grow "through" each other and even extend beyond the boundary of the dish. Of course, this is a somewhat unrealistic setup. But it will serve its purpose for a first prototype...

Input The input contains several test cases. Each of them is described as follows.

- It starts with a line that contains a single integer n ($1 \leq n \leq 10^5$), denoting the number of bacteria.
- The next line describes the dish by four integers `l b r t`, with $|l|, |b|, |r|, |t| < 2^{25}$. The dish consists of all points (x, y), for which $l \leq x \leq r$ and $b \leq y \leq t$.
- The following n lines describe the position of the bacteria, given by the integer coordinates of their centers `x y`, separated by space and so that $|x|, |y| < 2^{24}$. You may assume that all bacteria are located within the dish and they are at pairwise distinct positions.

The input is terminated by a line containing a single value 0. All coordinates are measured in μm.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
#include <algorithm>

typedef CGAL::Exact_predicates_inexact_constructions_kernel  K;
typedef CGAL::Triangulation_vertex_base_with_info_2<K::FT,K> Vb;
typedef CGAL::Triangulation_face_base_2<K>                   Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>          Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>                Delaunay;
typedef Delaunay::Finite_vertices_iterator                   VI;
typedef Delaunay::Finite_edges_iterator                      EI;
using namespace std;

double floor_to_double(const K::FT& x)
{
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

int hours(K::FT& in) {
  double d = CGAL::to_double(in);
  double z = sqrt(d)-0.5;
  int k;
  if (z <= 0.0) {
    k = 0;
  } else {
    k = ceil(sqrt(z));
  }
  return k;
}

void testcases() {
    size_t n;
    std::vector<Delaunay::Point> points;
    while (cin >> n && n > 0) {
        points.reserve(n);

        double l, b, r, t;
        // left, bottom, right, top
        cin >> l >> b >> r >> t;

        for (size_t i = 0; i < n; i++) {
          double x, y;
          cin >> x >> y;
            Delaunay::Point p(x,y);
            points.push_back(p);
        }


        Delaunay dt;
        dt.insert(points.begin(), points.end());
        // info (-> squared distance to nearest neighbor, initially, the boundary)
        for (VI v = dt.finite_vertices_begin(); v != dt.finite_vertices_end(); ++v) {
          v->info() = min(min(v->point().x() - l, r - v->point().x()),
                std::min(v->point().y() - b, t - v->point().y()));
          v->info() *= v->info();
         }

        // compute all nearest neighbors
        for (EI e = dt.finite_edges_begin(); e != dt.finite_edges_end(); ++e) {
            Delaunay::Vertex_handle v1 = e->first->vertex(dt.cw(e->second));
            Delaunay::Vertex_handle v2 = e->first->vertex(dt.ccw(e->second));
            K::FT d = CGAL::squared_distance(v1->point(), v2->point()) / 4;
            v1->info() = std::min(v1->info(), d);
            v2->info() = std::min(v2->info(), d);
        }
```

```
75          vector<K::FT> ltv;
            ltv.reserve(n);
77          for (VI v = dt.finite_vertices_begin(); v != dt.finite_vertices_end(); ++v) {
              ltv.push_back(v->info());
79          }
            sort(ltv.begin(), ltv.end());
81  //        for (vector<int>::iterator it=minf.begin(); it!=minf.end(); it++) {
    //          cout << *it << " ";
83  //        }
    //        cout << endl;
85          cout << hours(ltv[0]) << " " << hours(ltv[n/2]) << " " << hours(ltv[n-1]) << endl;
            points.clear();
87      }
    }
89
    int main()
91  {
        std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
93      cin.sync_with_stdio(false);
        cout.sync_with_stdio(false);
95      testcases();
        return 0;
97  }
```

./week_9_proximity_sources/3_germs/germs/main.cpp

Part X

Week 10 - Linear and Quadratic Programming

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week III

Algorithms Lab

Exercise 1 – *Maximize it!*

Your company often needs to solve simple optimization problems of two types.

$$
\begin{aligned}
(1) \quad & \max b \cdot y - a \cdot x^2 \\
& s.t.\ x, y \geq 0 \\
& x + y \leq 4 \\
& 4x + 2y \leq ab \\
& -x + y \leq 1
\end{aligned}
$$

$$
\begin{aligned}
(2) \quad & \min a \cdot x^2 + b \cdot y + z^4 \\
& s.t.\ x, y \leq 0 \\
& x + y \geq -4 \\
& 4x + 2y + z^2 \geq -ab \\
& -x + y \geq -1
\end{aligned}
$$

For any values of parameters a and b, find the optimal value.

Input The input consist of several test sets. Each test set is on a separate line, constisting of three values $p\ a\ b$, where $p \in \{1, 2\}$ is the type of the problem to solve (1. or 2.) and a, b are the values of the paremeters. Both a and b are integers such that $0 \leq a \leq 100$ and $-100 \leq b \leq 100$. The input is terminated by a line containing only `0`.

Output For each input, write on a single line the optimal value of the problem p with parameters a, b, rounded down to the next integer for the maximization problem, and rounded up to the next integer for the minimization problem. If there is no solution, write `no` and if there are solutions of arbitrarily high value (arbitrarily low for minimization), then output `unbounded`.

Sample Input

```
1 1 1
1 3 -3
2 1 1
2 2 1
0
```

Sample Output

```
0
no
0
-1
```

```cpp
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <vector>
#include <cmath>

using namespace std;

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}


void testcases() {
    size_t p;
  const int X = 0;
  const int Y = 1;
  const int ZZ = 2;
  int a, b;

    while (cin >> p && p > 0) {
      int a, b;
      cin >> a >> b;

        Program qp1 (CGAL::SMALLER, true, 0, false, 0);
        // -b*y + a*x^2
          qp1.set_c(Y, -b);
          qp1.set_d(X, X, 2*a);

        // x + y <= 4
        qp1.set_a(X, 0, 1);
        qp1.set_a(Y, 0, 1);
        qp1.set_b(0, 4);
      // 4x + 2y <= ab
        qp1.set_a(X, 1, 4);
        qp1.set_a(Y, 1, 2);
        qp1.set_b(1, a*b);

      // -x + y <= 1
        qp1.set_a(X, 2, -1);
        qp1.set_a(Y, 2, 1);
        qp1.set_b(2, 1);

      // X >= 0
//        qp1.set_l(X, true, 0);
      // Y >= 0
//      qp1.set_l(Y, true, 0);

      Program qp2 (CGAL::LARGER, false, 0, false, 0);
      // a*x^2 + b*y + zz^2
        qp2.set_d(X, X, 2*a);
        qp2.set_c(Y, b);
        qp2.set_d(ZZ, ZZ, 2);

      // x + y >= -4
        qp2.set_a(X, 0, 1);
        qp2.set_a(Y, 0, 1);
```

```
        qp2.set_b(0, -4);
        // 4x + 2y  + zz >= -ab
        qp2.set_a(X, 1, 4);
        qp2.set_a(Y, 1, 2);
        qp2.set_a(ZZ, 1, 1);
        qp2.set_b(1, -a*b);

      // -x + y >= -1
        qp2.set_a(X, 2, -1);
        qp2.set_a(Y, 2, 1);
        qp2.set_b(2, -1);

      // X <= 0
        qp2.set_u(X, true, 0);
      // Y <= 0
        qp2.set_u(Y, true, 0);
      // ZZ >= 0
        qp2.set_l(ZZ, true, 0);


    Solution s = (p==1 ?
            CGAL::solve_nonnegative_quadratic_program(qp1, ET()) :
            CGAL::solve_quadratic_program(qp2, ET()) );
    if (s.is_infeasible()) std::cout << "no" << std::endl;
    else if (s.is_unbounded()) std::cout << "unbounded" << std::endl;
    else std::cout << (p==1?-1:1)*ceil_to_double(s.objective_value())
       << std::endl;


    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_10_linear_and_quadratic_programming/0_maximize/maximize/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week III

Algorithms Lab

Exercise 2 – *Diets*

The economical crisis came and people in a poor country of Theirlands do have hardly enough money to feed themselves. The government decided to help them by designing a cheap, yet sufficient, diet for all its citizens. There are some constraints on the diet that need to be fulfilled for a long-term normal functioning of the human body, as for instance a daily intake of between around 1500 and 2500 kilocalories, some minimal amounts of vitamins, antioxidants and some other nutrients which are considered "generally healthy" – and much more. You can describe all these constraints in the form "the daily amount of x should be between a and b". These constraints slightly differ from person to person and are exactly determined by voluntary-unpaid doctors in the country.

What the government wants from you is a program that, given the constraints for an individual and the food which is available in the area where the individual lives, calculates the cheapest diet (that means amounts of each product such that the total amounts of each nutrient fulfills the prescribed conditions; these amounts do not need to be integral - it is possible to consume any fractional amount of a product in a day). They understand though, that it is an uneasy task on such a short notice and will for now be happy with an alpha version which only calculates the price of a cheapest diet.

Input The input file consists of several test cases. Each of them starts with a line containing two integers n and m ($1 \leq n \leq 40$ and $1 \leq m \leq 100$). The following n lines describe the nutrients. The ith of these lines contains two integers $\min_i$ and $\max_i$, the minimal and maximal daily amount of the ith nutrient. The next m lines describe the foods. Each of them contains an integer price of a unit of the jth product, p_j, and another n integers $C_{j,1} \dots C_{j,n}$ describing the amounts of nutrients $1 \dots n$ in a unit of product j. All the above integer values are in absolute value smaller than 2^{20}. The input is terminated by a line `0 0`.

All the numbers on a single line are separated by a single space and there are no trailing whitespaces at the end of a line.

Output The output for each testcase is a line containing a single number c, the cost of the cheapest diet rounded down to an integer. If there is no diet fulfilling the criteria, output the line `No such diet`. No trailing spaces at the end of the line are allowed.

Sample Input

```
1 1
1500 2500
2 512
2 3
1000 2000
```

```cpp
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <vector>
#include <cmath>

using namespace std;

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}


void testcases() {
    size_t p;
//  const int X = 0;
//  const int Y = 1;
//  const int ZZ = 2;

  int num_nutrients, num_foods;
  int nutrients[41][2];
  int food_nutrients[101][41];
  int prices[101];
    while (true) {
      cin >> num_nutrients >> num_foods;
      if (num_nutrients == 0 && num_foods == 0) {
        break;
      }
      for (int i=0; i<num_nutrients; i++) {
        cin >> nutrients[i][0] >> nutrients[i][1];
      }
      for (int i=0; i<num_foods; i++) {
        cin >> prices[i];
        for (int j=0; j<num_nutrients; j++) {
          cin >> food_nutrients[i][j];
        }
      }
      int FOODS = 0;
      Program qp (CGAL::SMALLER, true, 0, false, 0);
      // function to minimize
      for (int i=0; i < num_foods; i++) {
        qp.set_c(FOODS+i, prices[i]);
      }

      int eq_counter = 0;

      for (int i=0; i<num_nutrients; i++) {
        for (int j=0;j<num_foods;j++) {
            qp.set_a(FOODS+j,eq_counter,food_nutrients[j][i]);
        }
        qp.set_b(eq_counter, nutrients[i][1]);
        eq_counter++;
      }

      for (int i=0; i<num_nutrients; i++) {
        for (int j=0;j<num_foods;j++) {
            qp.set_a(FOODS+j,eq_counter,food_nutrients[j][i]);
```

```
        }
        qp.set_b(eq_counter, nutrients[i][0]);
        qp.set_r(eq_counter, CGAL::LARGER);
        eq_counter++;
      }

      Solution s = CGAL::solve_nonnegative_quadratic_program(qp, ET());
      if (s.is_optimal()) {
      cout << floor(CGAL::to_double(s.objective_value())) << endl;
    } else {
      cout << "No such diet.\n";
    }


    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_10_linear_and_quadratic_programming/1_diet/diet/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week III

Algorithms Lab

Exercise 3 – *Portfolios*

In a little country of Bankland, people are so rich, that they do not know what to do with their money. The definitely do not want to spend it for some cheap fun like going to parties, but rather want to invest it wisely to become even richer.

There are several assets into which one may invest. Starting with Bankland's own government bonds or Neighborland's government which involve a bit more risk and ending with a true gambling such as betting on a zero in roulette.

Let us be more precise now. We have n assets called (for the lack of imagination) $1, \ldots, n$. Each asset i has a cost c_i and an expected return r_i per unit. To model risk, we will use variance and covariances (in the same meaning as for random variables) – each pair of assets i and j has a covariance v_{ij} (variance of an asset i is the covariance v_{ii}; the covariances are symmetric, i.e. $v_{ij} = v_{ji}$ and positive semidefinite, i.e. the matrix $(v_{ij})_{i,j}$ is positive semidefinite). If we buy a portfolio of assets $1 \ldots n$ in the amounts $\alpha_1, \ldots, \alpha_n$ then the total variance of the portfolio is calculated as

$$V = \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j v_{ij}$$

and the total expected return $R = \sum_i \alpha_i r_i$.

Every one person in Bankland has different requirements on their portfolio. They have different starting capital, require different minimum expected return and as well accept some maximum risk (total variance of the portfolio). Your goal is to decide whether there exists a portfolio fulfilling their requirements. You may assume, that each asset can be bought in fractional amounts (i.e., smaller than one unit) but you can only buy nonnegative amounts of each asset.

Input The input consists of several test cases. Each of them starts with a line consisting of two integers n and m ($1 \leq n \leq 100$, $1 \leq m \leq 10$). The value n is the number of total assets available and m is the number of people asking you for the portfolio. The following n lines consist of 2 integers each. The i-th line consists two numbers c_i r_i, the cost and expected return of the i-th asset ($1 \leq r_i, r_j \leq 10^6$). The following n lines describe the covariances. The i-th line consists of n integers $v_{i1} \ldots v_{in}$ ($-10^6 \leq v_{ij} \leq 10^6$).

Each of the following m lines describes the individual investors and consists of three numbers C R V, where C is the maximum cost of the portfolio, R is the minimum expected return and V is the maximum variance of the portfolio. The matrix $(v_{ij})_{i,j}$ is symmetric and positive semidefinite (as such covariance matrices tend to be). The input is terminated by a line `0 0`.

Output For each of the test cases, the output should consist of m lines, each containing either the word "`Yes.`" or "`No.`" depending whether the portfolio for the investor exists (="`Yes.`") or not (="`No.`"). Do not include any other whitespaces or symbols.

```
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <vector>
#include <cmath>

using namespace std;

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

typedef struct {
  int cost;
  int ret;
  int var;
} cust;
void testcases() {
    size_t p;
//  const int X = 0;
//  const int Y = 1;
//  const int ZZ = 2;

    int costs[101];
    int ereturn[101];
    int covariances[101][101];
    cust customers[11];

    while (true) {
      int n_assets, m_customers;
      cin >> n_assets >> m_customers;
      if (n_assets == 0 && m_customers == 0) {
        break;
      }
      for (int i=0; i<n_assets; i++) {
        cin >> costs[i] >> ereturn[i];
      }
      for (int i=0; i<n_assets; i++) {
        for (int j=0; j<n_assets; j++) {
          cin >> covariances[i][j];
        }
      }

      for (int i=0; i<m_customers; i++) {
        int cost, ret, var;
        cin >> cost >> ret >> var;
        Program qp (CGAL::SMALLER, true, 0, false, 0);
        for (int i=0; i<n_assets; i++) {
          for (int j=0; j<n_assets; j++) {
            qp.set_d(i, j, 2*covariances[i][j]);
          }
        }

        for (int i=0; i<n_assets; i++) {
          qp.set_a(i, 0, ereturn[i]);

          qp.set_a(i, 1, costs[i]);
```

```
76          }
        qp.set_r(0, CGAL::LARGER);
78      qp.set_b(0, ret);

80      qp.set_b(1, cost);

82      Solution s = CGAL::solve_nonnegative_quadratic_program(qp, ET());
          if (s.is_optimal() && CGAL::to_double(s.objective_value()) <= var) {
84          //cout << CGAL::to_double(s.objective_value()) << " var: " << var << " ";
            cout << "Yes." << endl;
86        } else {
            cout << "No." << endl;
88        }
        }
90

    }
92  }

94  int main()
    {
96      std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
        cin.sync_with_stdio(false);
98      cout.sync_with_stdio(false);
        testcases();
100     return 0;
    }
```

./week_10_linear_and_quadratic_programming/2_portfolios/portfolios/main.cpp

ETH Zurich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Prof. Dr. Emo Welzl
Prof. Dr. Peter Widmayer

HS 2012
Exercise Sheet CGAL Week III

Algorithms Lab

Exercise 4 – *Inball*

The many-dimensional country of Ballland experienced a natural disaster and all its inhibitants have to seek shelter in its sophisticated system of underground caves. Unfortunatelly, some of the inhabitants of the Ballland have grown quite large (they are ball-shaped, indeed), it is necessary to figure out, how large inhabitants cave can still be accomodated in each cave. Each cave C is of a polyhedral shape, i.e., is described by n linear inequalities

$$C = \{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{a}_i^T \mathbf{x} \leq b_i \, , i = 1, \ldots, n\},$$

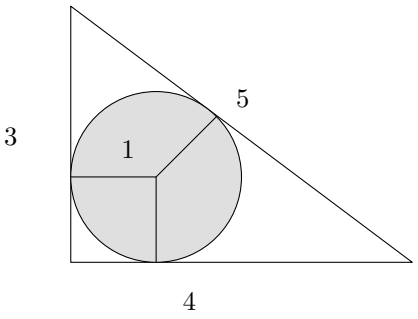
where $\mathbf{a}_i \in \mathbb{R}^d, b_i \in \mathbb{R}$. Here $\mathbf{a}_i^T \mathbf{x}$ denotes the standard scalar product of $\mathbf{a}_i$ and $\mathbf{x}$, i.e., $\mathbf{a}_i^T \mathbf{x} := \sum_{i=1}^{d} (\mathbf{a}_i)_j \mathbf{x}_j$.

Input The input contains several test cases. Each test case describes one cave. It begins with a line containing two integers n and d ($1 \leq d \leq 10$, $1 \leq n \leq 10^3$) where n is the number of inequalities describing the cave and d is its dimension.

Each subsequent line describes one inequality $\mathbf{a}_i^T \mathbf{x} \leq b_i$ and consist of $d + 1$ space separated integers $(\mathbf{a}_i)_1 (\mathbf{a}_i)_2 \ldots (\mathbf{a}_i)_d b_i$ ($-2^{10} \leq (\mathbf{a}_i)_1, \ldots, (\mathbf{a}_i)_d, b_i \leq 2^{10}$). It is guaranteed, that the norm $\|\mathbf{a}_i\|_2 = \sqrt{\sum_{j=1}^{d} ((\mathbf{a}_i)_j)^2}$ of each $\mathbf{a}_i$ is an integer.

The input is terminated by a line containing a single value 0.

Output For each input, the output appears on a single and separate line. This line consists of a single integer r denoting the maximum integral radius of a d-dimensional ball, which fits into the cave. If the cave is an empty set, the word `none` is to be printed. If an arbitrarily large ball can be fit into the cave, the word `inf` should be printed.



```cpp
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <vector>
#include <cmath>

using namespace std;

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

// program and solution types
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

double floor_to_double(const CGAL::Quotient<ET>& x)
{
    double a = floor(CGAL::to_double(x));
    while (a > x) a -= 1;
    while (a+1 <= x) a += 1;
    return a;
}

double ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

void testcases() {
  while (true) {
    int n, d;
      cin >> n;
      if (n == 0) {
        break;
      }
      cin >> d;

      int R = d;
      Program qp (CGAL::SMALLER, false, 0, false, 0);
      qp.set_c(R, -1);
      for (int i=0; i<n; i++) {
        int la = 0;
        for (int j=0; j<d; j++) {
          int a;
          cin >> a;
            qp.set_a(j, i, a);
            qp.set_a(j, i+n, a);

            la += a*a;
        }
        qp.set_a(R, i, int(sqrt(la)));

        int b;
        cin >> b;
        qp.set_b(i, b);
        qp.set_b(i+n, b);
      }

      Solution s = CGAL::solve_linear_program(qp, ET());
      if (s.is_infeasible()){
      cout << "none" << endl;
    } else if (s.is_unbounded()) {
      cout << "inf" << endl;
    } else {
//            CGAL::Quadratic_program_solution<ET>::Variable_value_iterator
```

```
//              opt = s.variable_values_begin();
//              CGAL::Quotient<ET> radius = *(opt);
//              cout << floor_to_double(radius) << endl;
      cout << abs( ceil_to_double(s.objective_value()) ) << endl;
      }
    }
}

int main()
{
    std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    testcases();
    return 0;
}
```

./week_10_linear_and_quadratic_programming/3_inball/inball/main.cpp