AlgoLab ETH 2013

name	problem	solution	P
Dominoes	Given a list of tiles of different heights, determine how many tiles will fall after toppling the left-most tile.	-	15
Aliens	Given a set of intervals, count those that have an element that is not contained in any other interval.	-	20
Shelves	Given two different types of shelves with length m and n, $m \leq n$ and an empty space with length l, determine the optimal number of shelves x and y such that $x * m + y * n = l - \epsilon$, $epsilon > 0$, epsilon is minimized. Second objective is minimizing y.	clever loop with branching that reduces runtime to $O(\sqrt{l})$	16
False Coin	Given a series weighing outcomes of several coins, determine which coin has a different weight than the other coins.	-	24
Light Pattern	Given a sequence of n bulbs that are in the states "on" or "off", determine the minimal number of "changes" that are necessary such that each interval $i * k$ to $(i+1) * k$ of the bulbs follows a given pattern. A "change" is either changing the state of a single bulb or change the state of every bulb from 0 up to $t * k - 1$.	-	35
Divisor Distance	Given a number n. Let G_n be a graph with 1 to n as vertices. Further, in G_n there is an (undirected) edge between vertices i and j with $i > j$ iff j is the largest proper divisor of i. Determine the minimal distance of two given vertices in this graph.	-	101
Formula One	Given a sequence of integers, determine the minimum number of "swaps" of numbers at position i and $i+1$ are necessary to sort the list.	merge sort and count swaps	26
Boats	Given a set of boat lengths l_i and positions of rings p_i, determine the maximum number of boats that can be tied.	Greedy (Boat with earliest end then longest)	22
Checking Change	Given a number of different coin-values c_i, output the minimum number of coins that are necessary to represent m_i.	DP	12
Even Pairs	Given a list $x_1, \ldots, x_n$, count the number of pairs $1 \leq i \leq j \leq n$ for which the sum is even.	DP	18
Burning Coins	Given a sequence of coins $c_1, \ldots, c_n$ with values $v_1, \ldots, v_n$. Each player $\in \{A, B\}$ is alternately allowed to remove the leftmost or rightmost coin of the sequence until the sequence is empty. Output the maximum value of coins that A can remove.	DP	33
Jump	Given n cells, a number representing how far the agent can jump and the cost a_i to land on a cell. Determine the minimum cost to get from cell 1 to cell n.	DP	34

Poker Chips	Given some stacks S_i with chips c_{ij} each having a color. An agent A is allowed to remove the topmost chip from each stack provided they have the same color. If A removed $k > 1$ chips, he is awarded with 2^{k-2} points. Determine the maximum number of points for the given S_i.	DP	89
Connecting capitals	Given a tree. Determine the largest set of vertex disjoint edges.	DP	lec. 12
Race Tracks	Given a rectangular grid with obstacles, a start S and goal position. An agent A starts at S, has a x-y-velocity (a, b) $-3 \leq a, b \leq 3$. After each move A can adjust $a' = a + n_a, b' = b + n_b, n_a, n_b \in \{-1, 0, 1\}$. Determine the minimum number of moves necessary for A to reach the goal.	represent $(x, y, (a, b))$ as a node in a graph, BFS	29
Next Path	Given a directed Graph with vertices s and t, determine the length of the second shortest path.	BFS	115
Longest Path	Determine the length of the longest path in a tree.	DFS while recording tree height	37
Odd Route	Given a directed weighted graph G and two vertices s, t of G. Find the shortest path from s to t such that both its number of edges and total weight are odd.	Construct G', then Dijkstra	117
Build The Graph		MST and Dijkstra	42
Ants	Given an undirected graph G, and nodes S_i of subgraphs i. Build the subgraphs i by starting at S_i and do a uniform cost search until all nodes are explored. Find the cheapest path by combining all S_i (not G) from node a to b.	MST, then Dijkstra	38
Deleted Entries	Given a undirected graph G. Find k-coloring such that the graph stays connected after deleting all edges inside color classes.	MST and BFS to k-color	43
Monkey Island	Let G be a undirected graph (V, E) with costs on vertices. Let S be a subset of vertices, such that $\forall v \in V, \exists s \in S$ and there is a path from s to v. Find S that minimises $\sum_{s \in S} cost(s)$	Strongly connected components	81
Bridges	Given a connected undirected graph G, find the edges whose lacking would result in two connected components	Biconnected components	40
Deleted Entries Strike Back	Given a connected undirected graph G, find 3-coloring such that the graph stays biconnected after deleting all edges inside color classes.	Biconnected components, DFS-coloring using lowpoint map	103
Shy Programmers	Given a undirected graph G, decide if it is outerplanar, i.e., you can draw it in a plane so that all vertices lie on a circle and all edges are straight and don't intersect.	construct G' then planarity test	46
Buddy Selection	Given n students with c hobbies each. Determine if they can be matched in pairs such that each pair shares at least f hobbies.	Maximum matching size	50

Tiles	Given a two-dimensional array of where each enty is either marked as "obstacle" or "no obstacle". Determine the maximum number of "tiles" consisting of two neighboring cells fit in the array. No cell of a tile can be placed on an "obstacle" cell.	Maximum matching	102
Consecutive Constructions	Given a DAG. Determine the maximum number of edges that can be packed in vertex-disjoint paths.	Maximum matching	lec. 12
Satellites	Minimum vertex cover of unweighted bipartite undirect graph.	Compute maximum matching and use Konigs Theorem	51
Placing Knights	Determine the maximum number of chess-knights on a field with given size and obstacles.	Maximum independent set (n-maximum matching)	82
Algocoon	Given a directed graph G find the global minimum (directed) cut.	Global minimum cut: find source/sink that minimize flow. Start BFS from source to get the cut.	47
Coin Tossing	Given a sequence of m games, some with known result, some without and $s_1, \ldots, s_n$ the number of won games for each player. Determine if it is possible to assign outcomes to unknown games and get standings.	Maximum flow	55
Shopping Trip	Given a Graph and a set of nodes S and a node s. Determine if its possible to find $\|S\|$ edge disjoint paths from s to $S_i \forall S_i \in S$.	Maximum flow	85
Tetris	Given an interval of width w and a list B of intervals with coordinates $[a_i, b_i], 0 \leq a_i, b_i \leq w$ each. A "full line" L is a sequence of intervals of B such that $[0, w]$ is contained in the concatenation of l. Determine the maximum number of "full lines" F that can be constructed using the intervals of B while each interval can only be used once. Moreover each interval border in F is only allowed to occur once in F.	Maximum flow	95
Kingdom Defense	Given directed graph with maximum and minimum edge capacities and starting and minimum ending vertex budget. Determine if its possible to send have a flow in the network such that min and max edge capacities and the minimum ending vertex budget holds.	Model minimal edge capacities, then maximum flow problem	53
Sweepers	Given an undirected graph G, a list of start vertices S and a list of end vertices $E, \|S\| = \|E\|$. Determine if it is possible to find $\|S\|$ paths each starting at a vertex of S and end at a vertex of E, (where every start and end vertex can only be used in one path) such that all edges of the graph occur in exactly one path.	Maximum flow and eulerian path check	109
Knights	Given an undirected graph G, a list of start vertices S and a list of end vertices E. Determine the maximum number of edge and vertex disjoint paths from S to E.	Maximum flow	113

The Bracelet	Given a list B of tuples $t_i = (c_{i,1}, c_{i,2})$ and a set of colors $c_{i,j} \in C \subset \mathbb{R}$. Determine if it is possible to construct a strongly connected directed graph such that all tuples in B occur once as vertices, each tuple has exactly one ingoing and exacly one outgoing edge and the following edges are allowed: There can be an edge from t_i to t_k or from t_k to t_i if $c_{i,l_1} = c_{k,l_2}, l_1, l_2 \in \{1, 2\}$.	Eulerian path construction	111
Hit	Given a ray and some line segments. Determine if ray intersects any segment.	loop & test intersection	60
First Hit	Given a ray and some line segments, where does the ray first intersect a segment?	randomize segment order, loop while minimizing intersection	61
Antenna	Given a set of points, find the minimum enclosing circle.	Minimum enclosing circle	57
Almost Antenna	Given a set of points, find the minimum circle that encloses all but one point.	Minimum enclosing circle	58
Theev	Given a set of points P and the center of a circle C_1 in the plane. Determine the radius of two circles with centers C_1 and C_2 such that all points in P are covered.	Sorting and Min circle	87
Search Snippets	Given a set of numbers and their 1D-positions (many for a single number) in a sequence, find the minimum length of a sequence such that all numbers are occuring.	Sweepline	63
Search Snippets	Given a set of numbers and their 1D-positions (many for a single number) in a sequence, find the minimum length of a sequence such that all numbers are occuring.	Sweepline	63
Beach Bar	Given a set of points P in the line. Determine the integral position of a point x such that $\lvert P_x \rvert$ is maximum, where P_x are the points in P that are closer than 100 units away from x. Further, if multiple x are possible minimize the distance d_x to the farthest point in P_x. Output all optimal positions, $\lvert P_x \rvert$ and d_x	Sweepline	97
Hiking Maps	Given a polygonal path $p_0, \ldots, p_{m-1}$ and triangles $t_0, \ldots, t_{n-1}$, what is the minimum length of an interval $[b, e)$contained in$[0, n)$ such that each leg $p_i p_{i+1}$ of the path is contained in at least one of $t_b, \ldots, t_{e-1}$	Fast triangle intersection test and Sweepline	72
Bistro	Given a set of points in the plane S and a set of points T. Determine for each point in T the distance to the closest point in S.	Delaunay triangulation on S	65
Graypes	Given a set of points G find the shortest distance between any pair of the points.	Delaunay triangulation	68
Collisions	Given a set P of points in the plane and a number d, how many points from P have at least one other point from P in distance $< d$?	Delaunay triangulation	77
Germs	Given a rectangle R and a set of points G inside the rectangle. For every point determine the minimal distance between either any other point or the rectangle boundary.	Delaunay triangulation, then find shortest incident triangulation edge	66

H1N1	How to move a disk without colliding with a given point set?	Precompute for each Delaunay-face the escape radius using DFS.	69
Cover	Given a rectangle and the center of n disks. Determine the minimal radius s.t. the rectangle is fully covered.	Delaunay triangulation (Voronoi)	99
Light the Stage	Given a set D of m disks (players) in the plane, a number h and a sequence $I_1, \ldots, I_n$ of n disks of radius h (lamps), denote by $A_i, i \in \{0, \ldots, i\}$ the set of all disks $d \in D$ such that d does not overlap with I_j, for any $j \in \{1, \ldots, i\}$. Determine the largest k such that A_k is not empty. Output A_k	Delaunay and binary search	105
Radiation 2	Given n healthy points and m cancer points in the plane, find a disk containing no healthy point but a maximum number of cancer points.	Delaunay and Sweepline along Voronoi Edge	??
Diet	Given the price of a food product and some nutritional constraints, calculate the cheapest diet.	Linear Programming	78
Inball	Given a cave $C = \{x \in \mathbb{R}^d \mid a_i^T x \leq b_i, i = 1, \ldots, n\}$. Find the ball with the largest radius that is contained in C.	Linear Programming	80
Radiation	Given a set of points H and a set of points T in the plane. Determine the smallest number d, such that a polynomial of degree d can completely distinguish all points in H from all points T.	Linear programming	107
Missing Roads	Given a weighted undirected graph and a number k, determine the matching M consisting of k edges where the sum of the weights in M is minimized.	Linear programming.	122
Software update	Given a list of centers of disks in the plane. Determine the maximal sum of the radii such that no two disks overlap	Linear programming.	lec.13
Maximize It	Solve two quadratic programs.	Quadratic Programming	75
Portfolios	Given price, expected return and the covariance matrix of some assets and the minimum expected portfolio return, maximum portfolio cost and maximum portfolio variance of some investor. Determine if a portfolio for the investor exists.	Quadratic Programming	79
Stamp Exhibition	Given a set of stamps S, a set of lamps L and a set of segments W in the plane. Each $s_i \in S$ has a maximal allowed light intensity M_i. Determine if its possible to set the power p_i of the i-th lamp such that each stamp is illuminated with intensity > 1 but smaller M_i.	Quadratic Programming	93
Portfolios revisited	Given price, expected return and the covariance matrix of some assets and the maximum portfolio cost and maximum portfolio variance of some investor. Determine the maximum possible expected return.	Quadratic Programming and Binary search	91

Misc

Methods

- -
- Sorting
- Binary search
- Greedy
- Dynamic programming (Trigger: current computation does not depend on previous computation)
- BFS/DFS
- Shortest path/Dijkstra
- Minimum spanning tree
- (strongly) Connected components
- Biconnected components
- Planarity test
- Maximum matching (Biconnected: Max Vertex Cover/Maximum Independent Set)
- (global) Minimum cut
- Network flows
- Eulerian path
- Minimum enclosing circle
- Sweepline
- Delaunay triangulation
- Linear Programming / Quadratic Programming
- Backtracking

Potential problems: too slow

- `std::ios_base::sync_with_stdio(false);`
- check if there is possibility for a cache over testcases
- check if every non-primitive variable is called by reference
- find upper/lower bound
- try binary search or combination of binary search and linear search
- If CGAL: try randomization of input
- If QP/LP:solve LP with **linear** program solver
- If QP/LP:use nonnegative program where possible

Potential problems: error

- Assertion Error: off-by-one/memory
- CGAL: Think of infinite vertex
- Input size requirements

Useful Code Snippets

```cpp
#include <iostream>
#include <vector>
#include <sstream>
#include <string.h>
#include <stdio.h>

using namespace std;

/*
 * debug macro
 */
#define DEBUG
#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

/*
 * fast I/O
 */
ios_base::sync_with_stdio(false);

/*
 * output floating point numbers in fixed point notation
 * setprecision: number of digits after the decimal point
 */
std::cout<<std::setiosflags(std::ios::fixed) << std::setprecision(0)<< ceil_to_double(sqrt(CGAL::to_double(maxDist)))
            <<std::endl;

/*
 * number to string
 */
template <typename T>
    string NumberToString ( T Number )
    {
     ostringstream ss;
     ss << Number;
     return ss.str();
    }

/*
 * highest and lowest of orderable types
 */
#include <limits>
int max = numeric_limits<int>::max();
int min = numeric_limits<int>::min();

/*
 * enums
 */
enum enumeration_name {
  value1,
  value2,
  value3,
};

/*
 * power
 */
#include <math.h>
pow(7,3) //7^3

/*
 * sqrt
 */
#include <math.h>
double result = sqrt(1024.0);

#include <cstdlib>
double result = std::abs(-1024.0);

/*
 * sum vector contents
```

```
 */
#include<numeric>
std::accumulate(newTopChipPosition.begin(),
                                    newTopChipPosition.end(),0);

/*
 * sort array
 */
#include<algorithm>
std::sort(parasols.begin(), parasols.end());

/*
 * find in vector
 */
std::find(aVector.begin(), aVector.end(), item)!=aVector.end()

/*
 * get index of max/min element in vector
 */
#include <algorithm>
std::vector<int> v(10,0);
int a[] = {1,2,3,2};
int maxIndex = std::max_element(v.begin(), v.end());
int maxIndex2 = std::max_element(a, a+4);

/*
 * set intersection
 */
std::vector<string> inters;
std::set_intersection(chars[i].begin(), chars[i].end(),
    chars[j].begin(), chars[j].end(),
    std::back_inserter(inters));

/*
 * queue reverse order
 */
priority_queue<int, vector<int>, greater<int>> Q;

/*
 * comparator
 */
struct faceComparator {
    bool operator() (const Face_handle &x, const Face_handle &y) {
        return (x->info() < y->info());
    }
};
std::set<Face_handle, faceComparator> faces;

/*
 * non-static comparator
 */
struct CityComparator{
    P mainCity;
    bool operator()(P x, P y) {
        K::FT d1 = S(mainCity, x).squared_length();
        K::FT d2 = S(mainCity, y).squared_length();
        return(d1 <d2);
    }

};

/*
 * Binary Search
 */
int midpoint(int lowerBound, int upperBound) {
    return lowerBound + (upperBound-lowerBound)/2;
}

int binary_search(int A[], int key, int imin, int imax)
{
  // continue searching while [imin,imax] is not empty
  while (imax >= imin)
    {
      // calculate the midpoint for roughly equal partition
      int imid = midpoint(imin, imax);
      if(A[imid] == key)
```

```
        // key found at index imid
        return imid;
      // determine which subarray to search
      else if (A[imid] < key)
        // change min index to search upper subarray
        imin = imid + 1;
      else
        // change max index to search lower subarray
        imax = imid - 1;
    }
  // key was not found
  return KEY_NOT_FOUND;
}



/*
 * BOOST
 */
/*
 * print graph
 */
void printGraph(Graph g, WeightMap weight) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        std::cout << source(*eiter, g) << " <--> " << target(*eiter, g)
          << " with weight of " << weight[*eiter]
          << std::endl;
    }
}

/*
 * shortest path BFS
 */
int shortestPath(const Graph &g, int x, int y) {
        //BFS
        int n=num_vertices(g);
        vector<bool> visited(n, false);
        vector<int> distance(n, -1);
        std::queue<int> bfs_queue;

        vector<int> bfs_num(n);
        bfs_queue.push(x);
        visited[x] = true;
        distance[x] = 0;
        while (!bfs_queue.empty()) {
        int v = bfs_queue.front();
        bfs_queue.pop();
        //cerr << "visit  : " << v << "\n";
        GraphTraits::out_edge_iterator e, e_end;
        for (tie(e, e_end) = out_edges(v, g); e != e_end; ++e) {
            Vertex t = target(*e, g);
            if (!visited[t]) {
            visited[t] = true;
            distance[t] = distance[v] + 1;
            //cerr << "enqueue " << t << "\n";
            if(t==y) return distance[t];
            bfs_queue.push(t);
            }
        }
    }
    return 0;
}

/*
 * non-recursive DFS
 */
int main ()
{
    int n;
    vector<vector<int> > edges;
    read_graph(n, edges);
    /* 8< */
    vector<bool> visited(n, false);
    vector<int> dfs_stack;
    vector<int> dfs_neighbor_pos;
```

```
227     int next_num = 0;
228     vector<int> dfs_num(n);
229     for (int u = 0; u < n; u++) {
230     if (visited[u])
231         continue;
232     dfs_stack.push_back(u);
233     dfs_neighbor_pos.push_back(0);
234     visited[u] = true;
235     dfs_num[u] = next_num++;
236     cerr << "visit␣" << dfs_num[u] << ":␣" << u << "\n";
237     while (!dfs_stack.empty()) {
238         int v = dfs_stack.back();
239         int i = dfs_neighbor_pos.back();
240         dfs_stack.pop_back();
241         dfs_neighbor_pos.pop_back();
242         vector<int>& neighbors = edges[v];
243         for (; i < (int)neighbors.size(); i++) {
244         int w = neighbors[i];
245         if (!visited[w]) {
246             /* defer looking at v */
247             dfs_stack.push_back(v);
248             dfs_neighbor_pos.push_back(i+1);
249             /* look at w next */
250             visited[w] = true;
251             dfs_num[w] = next_num++;
252             dfs_stack.push_back(w);
253             dfs_neighbor_pos.push_back(0);
254             cerr << "visit␣" << dfs_num[w] << ":␣" << w << "\n";
255             break;
256         }
257         /* if we fall off this loop (as opposed to break), we
258          * are done looking at v */
259         }
260     }
261     }
262     /* >8 */
263 }
264
265 /*
266  * Boost named parameters and visitor
267  */
268
269 vector<int> discoverTimeToVertex;
270
271 // Custom visitor used to record DFS order.
272 struct CustomVisitor : public default_dfs_visitor {
273     void discover_vertex(int u, const Graph& G) {
274         discoverTimeToVertex.push_back(u);
275     }
276 };
277
278 vector<Vertex> low(numVertices);
279 vector<Vertex> vertexToDiscoverTime(numVertices);
280 vector<Vertex> predecessor(numVertices);
281 discoverTimeToVertex.clear();
282 size_t num_comps
283     = biconnected_components(g,component,
284                              lowpoint_map(&low[0])
285                              .discover_time_map(&vertexToDiscoverTime[0])
286                              .predecessor_map(&predecessor[0])
287                              .visitor(CustomVisitor()));
288
289 /*
290  * Boost: defining a custom property
291  */
292 namespace boost {
293     enum edge_info_t { edge_info = 219 }; // A unique ID.
294     BOOST_INSTALL_PROPERTY(edge, info);
295 }
296
297 struct EdgeInfo {
298 ...
299 }
300
301 typedef adjacency_list<vecS, vecS, directedS,
302     no_property, // vertex property, none this time
```

```
    property<edge_info_t, EdgeInfo> //edge property, could also use a predefined property like edge_name_t and define it with integers
    > Graph;
typedef property_map<Graph, edge_info_t>::type InfoMap;
...
InfoMap info_map = get(edge_info, G);
info_map[e] = ...

/*
 * CGAL: add property to a point
 */
template <typename I>
struct MyP : public K::Point_2 {
    MyP(const K::FT& x, const K::FT& y, const I& i_) : K::Point_2(x,y), i(i_) {}
    I i;
};
typedef MyP<std::pair<std::size_t,K::FT> > Participant;
Participant(x,y,make_pair(i,h));

/*
 * CGAL: Voronoi dual
 */
// process all Voronoi vertices
for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f) {
    K::Point_2 p = t.dual(f);
    ...
}
// process all Voronoi edges
for(Edge_iterator e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e) {
    CGAL::Object o = t.dual(e);
    // o can be a segment, a ray or a line ...
    ...
}

/*
 * CGAL: Enhancing faces/vertices
 */
//Can use maps for {Face,Vertices,Edges(with some additional work)}-handles
//or:
#include <CGAL/Triangulation_face_base_with_info_2.h>
enum Color { Black = 0, White = 1, Red = 2};
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
//New face class, vertex class stays the same. With info parameter. Here each face gets a color
typedef CGAL::Triangulation_face_base_with_info_2<Color,K> Fb;
//change the underlying triangulation structure
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Triangulation;

Triangulation t;

// color all infinite faces black
Triangulation::Face_circulator f = t.incident_faces(t.infinite_vertex());
do {
    f->info() = Black;
} while (++f != t.incident_faces(t.infinite_vertex()));


/*
 * QP Debugging
 */
cout<<qp.get_n()<<std::endl;
cout<<qp.get_m()<<std::endl;
CGAL::print_quadratic_program(std::cerr, qp, "qp");

CGAL::Quadratic_program_options options;
options.set_pricing_strategy(CGAL::QP_BLAND);
Solution s = CGAL::solve_linear_program(qp, ET(), options);
```

Checking Change

```
#include <iostream>
#include <vector>
#include <utility>
#include <limits>
#include <stdio.h>
#include <string.h>

using namespace std;

/*
 * c[] coin denominations
 * ci |c[]|
 * n change
 */
int *change(int c[], int ci, int n) {
    int *C = new int[n+1];
    C[0] = 0;
    for(int p=1; p <= n; p++) {
        int min = numeric_limits<int>::max() - 1;
        for(int i=0; i < ci; i++) {
            if(c[i] <= p) {
                if (1 + C[p-c[i]] < min) {
                    min = 1 + C[p-c[i]];
                }
            }
        }
        C[p] = min;
    }
    return C;
}

int changeRec(int c[], int ci, int n) {
    if(n==0) {
        return 0;
    } else {
        int min = numeric_limits<int>::max() - 1;
        for(int i=ci-1; i >= 0; i--) {
            if(c[i] <= n) {
                int tryChange = changeRec(c, ci, n-c[i]);
                if (1 + tryChange < min) {
                    min = 1 + tryChange;
                }
            }
        }
        return min;
    }
}


//C[n] initialised with -1
int changeFast(vector<int> &C, int c[], int ci, int n) {
    if(n==0) {
        C[n] = 0;
    } else if(C[n] == -1) {
        int min = numeric_limits<int>::max() - 1;
        for(int i=ci-1; i >= 0; i--) {
            if(c[i] <= n) {
                int tryChange = changeFast(C, c, ci, n-c[i]);
                if (1 + tryChange < min) {
                    min = 1 + tryChange;
                }
            }
        }
        C[n] = min;
    }
    return C[n];
}

void reverse(int *word, int len)
{
    int temp;
    for (int i=0;i<len/2;i++)
    {
```

```
            temp=word[i];
            word[i]=word[len-i-1];
            word[len-i-1]=temp;
    }
}

int *changeFaster(int c[],int n, const int A, int **C) {
    reverse(c, n);
    for(int j=0; j<A; j++) {
        C[n-1][j] =j;
    }
    for(int i=n-2; i>=0; i--) {
        for(int j=0; j<A; j++) {
            if(c[i] > j || C[i+1][j] < 1 + C[i][j-c[i]]) {
                C[i][j] = C[i+1][j];
            } else {
                C[i][j] = 1 + C[i][j-c[i]];
            }
        }
    }
}


int count( int S[], int m, int n )
{
    // table[i] will be storing the number of solutions for
    // value i. We need n+1 rows as the table is consturcted
    // in bottom up manner using the base case (n = 0)
    int table[n+1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 0;

    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin
    for(int i=0; i<m; i++)
        for(int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];

    return table[n];
}

// fill vector up such that all indices n - former C.size()  are -1
void fillVector(vector<int> &C, int n) {
    for(int i=C.size(); i < n+1; i++) {
        C.push_back(-1);
    }
}

void printVector(vector<int> &C) {
    for(int i=0; i < C.size(); i++) {
        cout << C[i] << "␣";
    }
    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++) {
        int ci, mi;
        cin >> ci >> mi;
        int c[ci];
        for(int j=0; j < ci; j++) {
            int coinValue;
            cin >> coinValue;
            c[j] = coinValue;
        }
        vector<int> C;
```

```
150         for(int k=0; k < mi; k++) {
151             int n;
152             cin >> n;
153             fillVector(C, n);
154 
155             changeFast(C, c, ci, n);
156             int numCoins = C[n];
157             if(numCoins == numeric_limits<int>::max() - 1) {
158                 cout << "not possible" << endl;
159             } else {
160                 cout << numCoins << endl;
161             }
162             // delete C;
163         }
164     }
165 }
```

Dominoes

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <limits>
#include <algorithm>

using namespace std;

int main() {
    std::ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;

    for(int k=0; k < testCases; k++) {
        int n;
        cin >> n;
        int numFall = 0;

        int h0; //height of the first dominoe which is surely falling
        cin >> h0;
        int accH = h0; //accH is the currently highest falling dominoe

        for(numFall=1; numFall < n; numFall++) {
            int h;
            cin >> h;
            accH= accH-1;
            if(accH > 0) {
                accH = max(accH, h);
            } else {
                break; //this dominoe does not fall
            }
        }
        //read rest of the file
        for(int r=numFall+1; r<n;r++) {
            int h;
            cin >> h;
        }

        cout << numFall << endl;
    }
}
```

Shelves

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <limits>
#include <math.h>

using namespace std;

struct triple {
    int first;
    int second;
    int third;
};

void computeSlow(triple &result, int lTotal, int lM, int lN) {
        for(int uncovered = 0; uncovered < lTotal; uncovered++) {
            int rest = lTotal - uncovered;
            // cout << uncovered << " rest:" << rest << " ";
            for(int numN = rest/lN; numN >= 0; numN--) {
                rest = lTotal - uncovered - numN*lN;
                // cout << " numN: " << numN << " rest:"<< rest << " ";
                for(int numM = rest/lM; numM >= 0; numM--) {
                    rest = lTotal - uncovered -numN*lN - numM*lM;
                    // cout << " numM: " << numM << " rest:"<< rest << " "<< endl;
                    if(rest==0) {
                        result.first=numM;
                        result.second=numN;
                        result.third=uncovered;

                        return;
                    }
                }
            }
        }
        result.first = 0;
        result.second = 0;
        result.third = lTotal;
}

void computeFast(triple &result, int lTotal, int lM, int lN) {
    int rest = lTotal;
    int a, b = 0;
    for(int bp=0; bp<=lTotal/lN; bp++) {
        int ap = (lTotal-bp*lN)/lM;
        int restp = lTotal - ap*lM - bp*lN;
        if(restp <= rest) {
            rest = restp;
            b = bp;
            a = ap;
        }
    }
    result.first = a;
    result.second = b;
    result.third = rest;
    return;
}

// runtime sqrt(lTotal)
void computeVeryFast(triple &result, int lTotal, int lM, int lN) {
    int rest = lTotal;
    int a, b = 0;
    if(lN < sqrt(lTotal)) { // lTotal is big
        for(int ap=lN-1; ap >=0; ap--) {
            int bp = (lTotal-ap*lM)/lN;
            int restp = lTotal - ap*lM - bp*lN;
            if(restp <= rest) {
                rest = restp;
                b = bp;
                a = ap;
            }
        }

    } else { //lN > sqrt(lTotal) => loop iterates < lTotal/sqrt(lTotal) < sqrt(lTotal)
```

```
        for(int bp=0; bp<=lTotal/lN; bp++) {
            int ap = (lTotal-bp*lN)/lM;
            int restp = lTotal - ap*lM - bp*lN;
            if(restp <= rest) {
                rest = restp;
                b = bp;
                a = ap;
            }
        }
    }
    result.first = a;
    result.second = b;
    result.third = rest;
    return;
}


int main() {
    int testCases;

    cin >> testCases;

    for(int i=0; i < testCases; i++) {
        int lTotal, lM, lN, nM, nN, uncovered;
        cin >> lTotal >> lM >> lN;
        triple result;
        computeVeryFast(result, lTotal, lM, lN);
        cout << result.first << "␣" << result.second << "␣" << result.third << endl;
    }

    return 0;
}
```

Even Pairs

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <limits>
#include <math.h>

using namespace std;

struct triple {
    int first;
    int second;
    int third;
};

void computeSlow(triple &result, int lTotal, int lM, int lN) {
        for(int uncovered = 0; uncovered < lTotal; uncovered++) {
            int rest = lTotal - uncovered;
            // cout << uncovered << " rest:" << rest << " ";
            for(int numN = rest/lN; numN >= 0; numN--) {
                rest = lTotal - uncovered - numN*lN;
                // cout << " numN: " << numN << " rest:"<< rest << " ";
                for(int numM = rest/lM; numM >= 0; numM--) {
                    rest = lTotal - uncovered -numN*lN - numM*lM;
                    // cout << " numM: " << numM << " rest:"<< rest << " "<< endl;
                    if(rest==0) {
                        result.first=numM;
                        result.second=numN;
                        result.third=uncovered;

                        return;
                    }
                }
            }
        }
        result.first = 0;
        result.second = 0;
        result.third = lTotal;
}

void computeFast(triple &result, int lTotal, int lM, int lN) {
    int rest = lTotal;
    int a, b = 0;
    for(int bp=0; bp<=lTotal/lN; bp++) {
        int ap = (lTotal-bp*lN)/lM;
        int restp = lTotal - ap*lM - bp*lN;
        if(restp <= rest) {
            rest = restp;
            b = bp;
            a = ap;
        }
    }
    result.first = a;
    result.second = b;
    result.third = rest;
    return;
}

// runtime sqrt(lTotal)
void computeVeryFast(triple &result, int lTotal, int lM, int lN) {
    int rest = lTotal;
    int a, b = 0;
    if(lN < sqrt(lTotal)) { // lTotal is big
        for(int ap=lN-1; ap >=0; ap--) {
            int bp = (lTotal-ap*lM)/lN;
            int restp = lTotal - ap*lM - bp*lN;
            if(restp <= rest) {
                rest = restp;
                b = bp;
                a = ap;
            }
        }

    } else { //lN > sqrt(lTotal) => loop iterates < lTotal/sqrt(lTotal) < sqrt(lTotal)
```

```
        for(int bp=0; bp<=lTotal/lN; bp++) {
            int ap = (lTotal-bp*lN)/lM;
            int restp = lTotal - ap*lM - bp*lN;
            if(restp <= rest) {
                rest = restp;
                b = bp;
                a = ap;
            }
        }
    }
    result.first = a;
    result.second = b;
    result.third = rest;
    return;
}


int main() {
    int testCases;

    cin >> testCases;

    for(int i=0; i < testCases; i++) {
        int lTotal, lM, lN, nM, nN, uncovered;
        cin >> lTotal >> lM >> lN;
        triple result;
        computeVeryFast(result, lTotal, lM, lN);
        cout << result.first << "␣" << result.second << "␣" << result.third << endl;
    }

    return 0;
}
```

Aliens

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

using namespace std;

int numSuperior(vector<pair<int, int> > &interval, int numHumans) {
    sort(interval.begin(), interval.end());
    int rightmost = 0;
    for(int i=0; i<interval.size(); i++) {
        if(interval[i].first > rightmost + 1)
            return 0;
        else
            rightmost = max(rightmost, interval[i].second);
    }
    if(rightmost < numHumans)
        return 0;

    rightmost = 0;
    int rightMostI = -1;
    int beforeLeft = 0;
    int numSuperiorAliens;
    int beforeI;
    vector<bool> isSuperior(interval.size(), true);

    for(int i=0; i<interval.size(); i++) {
        if(interval[i].first != beforeLeft) {
            if(interval[i].second <= rightmost) {
                isSuperior[i] = false;
                // cout << "right intervall smaller equal rightmost" << endl;
            }
        } else {
            if(interval[i].second < rightmost) {
                isSuperior[i] = false;
                // cout << "right intervall smaller equal rightmost" << endl;
            } else if(interval[i].second == rightmost) {
                isSuperior[i] = false;
                isSuperior[beforeI] = false;
                //cout << "right intervall equal rightmost";
            } else {
                isSuperior[beforeI] = false;
            }
        }
        if(rightmost <= interval[i].second) {
            rightmost = interval[i].second;
        }
        beforeI = i;
        beforeLeft = interval[i].first;
    }
    numSuperiorAliens=0;
    for(int i=0; i<isSuperior.size(); i++) {
        // cout << "alien: " << interval[i].first << " " << interval[i].second << " ";
        if(isSuperior[i]) {
            numSuperiorAliens++;
            // cout << "is superior!!";
        }
        // cout << endl;
    }

    return numSuperiorAliens;
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;

    for(int t=0; t < testCases; t++) {
        int numHumans, numAliens;
        cin >> numAliens >> numHumans;

```

```
74          vector<pair<int, int> > interval;
75          for(int i=0; i < numAliens; i++) {
76              int intervalLeft;
77              int intervalRight;
78              cin >> intervalLeft >> intervalRight;
79              if(intervalLeft != 0 && intervalRight!=0) {
80                  pair<int, int> anInterval = pair<int, int>(intervalLeft, intervalRight);
81                  interval.push_back(anInterval);
82              }
83          }
84          cout << numSuperior(interval, numHumans) << endl;
85      }
86  }
```

Boats

```
#include <iostream>
#include <limits>
#include <set>
#include <vector>
#include <algorithm>
using namespace std;

class boat {
    public:
        int length;
        int position;
        boat(int len, int pos) {
            length = len;
            position = pos;
        }
        bool operator< (const boat& other) const {
            return position < other.position;
        }
        bool operator== (const boat& other) const {
            return length == other.length
                && position == other.position;
        }
};

int numBoats(int numWizards, set<boat> &boats) {
    int curPos = numeric_limits<int>::min();
    int numBoats = 0;
    int i = 0;
    while(boats.size() > 0 ) {
        set<boat>::iterator it;
        for(it=boats.begin(); (*it).position < curPos; it++) {
            boats.erase(it);
        }
        // if(it==boats.end()) {
        //     return numBoats;
        // }
        int bestEnd = numeric_limits<int>::max();
        // boat bestBoat(bestEnd, bestEnd);
        const boat *bestBoat = NULL;
        set<boat>::iterator bestBoatIterator;
        for(it=boats.begin();it!=boats.end(); it++) {
            int begin = ((*it).position - (*it).length > curPos) ? ((*it).position-(*it).length) : curPos;
            int end = begin + (*it).length;
            if(end < bestEnd) {
                bestEnd = end;
                bestBoat = &(*it);
                bestBoatIterator = it;
            }
            if((*it).position > bestEnd) {
                break;
            }
        }
        curPos=bestEnd;
        bestBoat = bestBoat;
        numBoats+=1;
        i++;
        // cout << "boat " << i << " with p:" << bestBoat->position << " l:" << bestBoat->length << " , end is at " ↙
            ↳ << curPos << endl;
        boats.erase(bestBoatIterator);
    }
    return numBoats;

}

int numBoatsFast(int numWizards, vector<boat> &boats) {
    int i = 0;
    sort(boats.begin(), boats.end());
    // int curPos = (*boats.begin()).position;
    int curPos = numeric_limits<int>::min();
    // i++;
    // int numBoats = 1;
    int numBoats = 0;
    while(i < numWizards) {
```

```
73          for(;i < numWizards && boats[i].position < curPos; i++);
74          if(i==numWizards) break;
75          int newMaxPos = numeric_limits<int>::max();
76          int j = i;
77          while(j < numWizards && boats[j].position < newMaxPos){
78              int proposed = max(boats[j].position, curPos + boats[j].length);
79              if(proposed < newMaxPos) {
80                  newMaxPos = proposed;
81              }
82              j++;
83          }
84          curPos = newMaxPos;
85          // cout << "boat " << j-1 << " with p:" << boats[j-1].position << " l:" << boats[j-1].length << " , end is at ↙
                ↳ " << curPos << endl;
86          numBoats++;
87          i+=(j-i);
88          // i++;
89      }
90      return numBoats;
91  }
92
93  int main() {
94      ios_base::sync_with_stdio(false);
95      int testCases;
96      cin >> testCases;
97
98      for(int i=0; i < testCases; i++) {
99          int numWizards;
100         cin >> numWizards;
101         vector<boat> boats (numWizards, boat(0,0));
102         for(int j=0; j < numWizards; j++) {
103             int length;
104             int position;
105             cin >> length >> position;
106             boats[j] = boat(length, position);
107         }
108         cout << numBoatsFast(numWizards, boats) << endl;
109     }
110 }
```

False Coin

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <limits>
#include <stdio.h>
#include <string.h>
#include <sstream>

using namespace std;

bool *initializeArray(bool init, int n) {
    bool *array = new bool[n];
    for(int i=0; i < n; i++) {
        array[i] = init;
    }
    return array;
}

string falseCoin(bool suspicious[], int numCoins) {
    int fCoin = -1;
    for(int i=0; i<numCoins; i++) {
        if(suspicious[i]) {
            if(fCoin!=-1) {
                return "0";
            } else {
                fCoin = i+1;
            }
        }
    }
    return static_cast<ostringstream*>( &(ostringstream() << fCoin) )->str();
}

int main() {
    cin.sync_with_stdio(false);
    int testCases;
    cin >> testCases;

    for(int i=0; i < testCases; i++) {
        // char newline;
        // cin >> newline;

        int numCoins, numWeighings;
        cin >> numCoins >> numWeighings;
        bool *suspicious = initializeArray(true, numCoins);

        for(int j=0; j < numWeighings; j++) {
            int amountInPan;
            cin >> amountInPan;
            bool *weighingCoins = initializeArray(false, numCoins);
            // read currently weighted coins
            for(int k=0; k < 2*amountInPan; k++) {
                int currentCoin;
                cin >> currentCoin;
                weighingCoins[currentCoin-1] = true;
            }

            char outcome;
            cin >> outcome;
            if(outcome == '=') {
                //every weighing coin true, is false in coins
                for(int k=0; k<numCoins; k++) {
                    if(weighingCoins[k]) {
                        suspicious[k] = false;
                    }
                }
            } else {
                //set all others to false
                for(int k=0; k<numCoins; k++) {
                    if(!weighingCoins[k]) {
                        suspicious[k] = false;
                    }
                }
            }
```

```
74              delete weighingCoins;
75          }
76
77          cout << falseCoin(suspicious, numCoins) << endl;
78          delete suspicious;
79      }
80  }
```

Formula One

```
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

void swap(int* array, int i, int j) {
    int tmp;
    tmp = array[j];
    array[j] = array[i];
    array[i] = tmp;
}

int overpasses;

//! \brief Merges two sorted vectors into one sorted vector
//! \param left A sorted vector of integers
//! \param right A sorted vector of integers
//! \return A sorted vector that is the result of merging two sorted
//! vectors.
vector<int> merge(const vector<int>& left, const vector<int>& right)
{
    // Fill the resultant vector with sorted results from both vectors
    vector<int> result;
    unsigned left_it = 0, right_it = 0;

    while(left_it < left.size() && right_it < right.size())
    {
        // If the left value is smaller than the right it goes next
        // into the resultant vector
        if(left[left_it] < right[right_it])
        {
            result.push_back(left[left_it]);
            left_it++;
            overpasses= (overpasses + right_it) % 10000;
        }
        else
        {
            result.push_back(right[right_it]);
            right_it++;
        }
    }

    // Push the remaining data from both vectors onto the resultant
    while(left_it < left.size())
    {
        result.push_back(left[left_it]);
        overpasses= (overpasses + right_it) % 10000;
        left_it++;
    }

    while(right_it < right.size())
    {
        result.push_back(right[right_it]);
        right_it++;
    }

    return result;
}
//! \brief Performs a recursive merge sort on the given vector
//! \param vec The vector to be sorted using the merge sort
//! \return The sorted resultant vector after merge sort is
//! complete.
vector<int> merge_sort(vector<int>& vec)
{
    // Termination condition: List is completely sorted if it
    // only contains a single element.
    if(vec.size() == 1)
    {
        return vec;
    }

    // Determine the location of the middle element in the vector
```

```
74      std::vector<int>::iterator middle = vec.begin() + (vec.size() / 2);
75
76      vector<int> left(vec.begin(), middle);
77      vector<int> right(middle, vec.end());
78
79      // Perform a merge sort on the two smaller vectors
80      left = merge_sort(left);
81      right = merge_sort(right);
82
83      return merge(left, right);
84  }
85
86  /*
87   * return number of bubble sort steps necessary
88   */
89  int bubbleSort(int* order, int numRacers) {
90      bool swapped = false;
91      int numSwaps = 0;
92      int n = numRacers;
93
94      do {
95          int newN = 0;
96          swapped = false;
97          for(int i=1; i<n; i++) {
98              if(order[i-1] > order[i]) {
99                  swap(order, i-1, i);
100                 swapped = true;
101                 newN = i;
102                 numSwaps++;
103             }
104         }
105         n = newN;
106     } while(!n==0);
107
108     return numSwaps;
109 }
110
111 // uses bubbleSort to control for correctness of the merge sort overpasses counting
112 void controlIfCorrect(vector<int> order) {
113         int orderArray[order.size()];
114         copy(order.begin(), order.end(), orderArray);
115
116         int overpBubble = bubbleSort(orderArray, order.size());
117
118         if(overpBubble == overpasses) {
119             // cout << "ok: " << overpasses << endl;
120         } else {
121             cout << "different␣answer:␣bubble:␣" << overpBubble << "merge:␣" << overpasses << endl;
122             cout << "testset:␣" << endl;
123             copy(order.begin(), order.end(), ostream_iterator<int>(cout, "␣"));
124         }
125 }
126
127 int main() {
128     cin.sync_with_stdio(false);
129     int testCases;
130     cin >> testCases;
131
132     for(int i=0; i < testCases; i++) {
133         int numRacers;
134         cin >> numRacers;
135
136         // int order[numRacers];
137         vector<int> order;
138         for(int j=0; j < numRacers; j++) {
139             int racer;
140             cin >> racer;
141             order.push_back(racer);
142         }
143
144         overpasses = 0;
145         merge_sort(order);
146
147         // controlIfCorrect(order);
148
149         cout << overpasses % 10000 << endl;
```

```
150
151         }
152     }
```

Race Tracks

```
#include <iostream>
#include <limits>
#include <vector>
#include <queue>
#include <set>
#include <sstream>
#include <cstdlib>

using namespace std;

template <typename T>
    string NumberToString ( T Number )
    {
     ostringstream ss;
     ss << Number;
     return ss.str();
    }

class point {
    private:
        int x;
        int y;

    public:
        point(int X, int Y) {
            x = X;
            y = Y;
        }
        point(const point &p) {
            x = p.x;
            y = p.y;
        }
        bool operator== (const point &other) const {
            return (x == other.getX() &&
                    y == other.getY());
        }
        bool operator!= (const point &other) const {
            return !(*this == other);
        }
        bool operator< (const point &other) const {
            if(x < other.x) {
                return true;
            } else if((x == other.x) && (y < other.y)) {
                return true;
            }
            return false;
        }
        int getX() const { return x; }
        int getY() const { return y; }
        string str() const { return "(" + NumberToString(x) + "␣" + NumberToString(y) + ")"; }
};

class vertex {
    private:
        point position;
        point velocity;
        int distance;
    public:
        vertex(point &i_position, point &i_velocity) :
                position(0,0), velocity(0,0) {
            position = i_position;
            velocity = i_velocity;
            distance = numeric_limits<int>::max();
        }
        bool operator== (const vertex &other) const {
            return (position == other.position
                    && velocity == other.velocity
                );
        }
        bool operator!= (const vertex &other) const {
            return !(*this == other);
        }
        bool operator < (const vertex& other) const {
```

```
74              if(position < other.getPosition()) {
75                      return true;
76                  } else if(position == other.position) {
77                      return (velocity < other.velocity);
78                  }
79              return false;
80          }
81          point getPosition() const { return position; }
82          point getVelocity() const { return velocity; }
83          int getDistance() const { return distance; }
84          void setDistance(int distance_arg) { distance = distance_arg; return;}
85          string str() const { return "(" + NumberToString(distance) + "," + position.str() + "," + velocity.str() + ↙
                ↳ ")"; }
86  };
87
88  struct OrderByDistance {
89      bool operator() (const vertex& v1, const vertex& v2) const {
90          return (v1.getDistance() > v2.getDistance());
91      }
92  };
93
94  struct Obstacle {
95      int x1;
96      int x2;
97      int y1;
98      int y2;
99  };
100
101 bool validPosition(point p, int width, int height, Obstacle obstacles[], int numObstacles) {
102     if(!((0<=p.getX()) && (p.getX()<width) && (0<=p.getY()) && (p.getY()<height))) {
103         return false;
104     }
105     for(int i=0; i < numObstacles; i++) {
106         Obstacle o = obstacles[i];
107         if((o.x1 <= p.getX()) && (p.getX() <= o.x2) &&
108                 (o.y1 <= p.getY()) && (p.getY() <= o.y2)) {
109             // cout << o.x1 << " " << p.getX() << " " << o.x2 << endl;
110             // cout << o.y1 << " " << p.getY() << " " << o.y2 << endl;
111             return false;
112
113         }
114     }
115     return true;
116 }
117
118 void printQueue(priority_queue<vertex,vector<vertex>,OrderByDistance> queue) {
119     cout << "Queue:␣";
120     while(queue.size() > 0) {
121         cout << queue.top().str() << ","; queue.pop();
122     }
123     cout << endl;
124 }
125
126 void printSet(set<vertex> visit) {
127     cout << "Visited:␣";
128     while(visit.size() > 0) {
129         set<vertex>::iterator firstEle = visit.begin();
130         cout << (*firstEle).str() << ","; visit.erase(firstEle);
131     }
132     cout << endl;
133 }
134
135 int vertexToHash(vertex v, int width, int height) {
136     // x y xa ya
137     return (v.getPosition().getX() * (height * 7 * 7)
138             + v.getPosition().getY() * 7 * 7
139             + (v.getVelocity().getX()+3) * 7
140             + (v.getVelocity().getY()+3));
141 }
142
143 vertex hashToVertex(int hash, int width, int height) {
144     int ya = (hash%7)-3;
145     hash = hash/7;
146     int xa = (hash%7)-3;
147     hash = hash/7;
148     int y = hash%height;
```

```
    hash = hash/height;
    int x = hash;
    point pos =point(x,y);
    point vel = point(xa,ya);
    return vertex(pos,vel);
}

void printVisited(bool *visit, int width, int height) {
    int n = width*height*7*7;
    cout << "Visited:␣";
    for(int i=0; i < n; i++) {
        if(visit[i]) {
            cout << hashToVertex(i, width, height).str() << ",";
        }
    }
}


int shortestPath(point start, point end, int width, int height, Obstacle obstacles[], int numObstacles) {
    // set<vertex> visit;
    int numVisit = width*height*7*7;
    bool visit[numVisit];
    for(int v=0; v < numVisit; v++) {
        visit[v] = false;
    }
    // flags to set nodes state to     visited
    priority_queue<vertex,vector<vertex>,OrderByDistance> queue ;

    point zero(0,0);
    vertex startVertex(start, zero);
    startVertex.setDistance(-1);
    queue.push(startVertex);

    while (!queue.empty()) {
        vertex cur = queue.top(); queue.pop();
        if(visit[vertexToHash(cur,width,height)])
           continue;

        // printQueue(queue);
        // cout<<"current vertex: " << cur.str() << endl;
        // printVisited(visit, width, height);
        visit[vertexToHash(cur,width,height)] = true;
        if(cur.getPosition() == end) {
            return cur.getDistance();
        }

        point newPosition(cur.getPosition().getX() + cur.getVelocity().getX(),
                          cur.getPosition().getY() + cur.getVelocity().getY());

        if(validPosition(newPosition, width, height, obstacles, numObstacles)) {
            for(int xa=-1; xa <=1; xa++) {
                for(int ya=-1; ya <= 1; ya++) {
                    // if(xa == 0 && ya == 0 && ) {
                    //     continue;
                    // }
                    if(abs(cur.getVelocity().getX() + xa) > 3
                            || abs(cur.getVelocity().getY() + ya) > 3) {
                        continue;
                    }

                    point newVelocity(cur.getVelocity().getX()+xa,
                                                cur.getVelocity().getY()+ya);
                    vertex to = vertex(newPosition, newVelocity);
                    int altDistance = cur.getDistance() + 1;
                    if((!visit[vertexToHash(to,width,height)]) //if not in visited set
                        && altDistance < to.getDistance() ) {
                            to.setDistance(altDistance);
                            queue.push(to); //push again to the queue? not bad I think
                    }

                }

            }

        }
    }
```

```
    return -2;
}


int main() {
    cin.sync_with_stdio(false);
    int testCases;
    cin >> testCases;

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

        int width, height;
        cin >> width >> height;
        int sx, sy, ex, ey;
        cin >> sx >> sy >> ex >> ey;
        point start(sx,sy), end(ex,ey);

        int numObstacles;
        cin >> numObstacles;
        Obstacle obstacles[numObstacles];
        for(int j=0; j<numObstacles; j++) {
            cin >> obstacles[j].x1 >> obstacles[j].y1 >> obstacles[j].x2 >> obstacles[j].y2;
        }
        int sp = shortestPath(start, end, width, height, obstacles, numObstacles);
        if(sp<-1) {
            cout << "No solution." << endl;
        } else if(sp==-1) {
            cout << "Optimal solution takes 0 hops." << endl;
        } else {
            cout << "Optimal solution takes " << sp << " hops." << endl;
        }
    }
}
```

Burning Coins

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

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;

    for(int i=0; i < testCases; i++) {
        int numCoins;
        cin >> numCoins;
        int minOpt[numCoins][numCoins];
        for(int j=0; j < numCoins; j++) {
            cin >> minOpt[j][j];
        }
        for(int j=0; j < numCoins-1; j++) {
            minOpt[j][j+1] = max(minOpt[j][j], minOpt[j+1][j+1]);
            // cout << j << " " << j+1 << " " << minOpt[j][j+1] << endl;
        }
        for(int m=2; m < numCoins; m++) {
            for(int j=0; j < numCoins-m; j++) {
                int k= j+m;

                minOpt[j][k]
                    = max(minOpt[j][j] + ((minOpt[j+2][k] > minOpt[j+1][k-1]) ? minOpt[j+1][k-1] : minOpt[j+2][k]),
                            minOpt[k][k] + ((minOpt[j+1][k-1] > minOpt[j][k-2]) ? minOpt[j][k-2] : minOpt[j+1][k-1]));
                // cout << j << " " << k << " " << minOpt[j][k] << endl;

            }
        }
        cout << minOpt[0][numCoins-1] << endl;
    }
}
```

Jump

```cpp
#include <iostream>
#include <limits>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numCells, maxJump;
        cin >> numCells >> maxJump;

        int vCell[numCells];
        long long minCost[numCells];

        minCost[0] = 0;
        cin >> vCell[0];
        int globalBestPos = 0;

        for(int curCell=1; curCell < numCells; curCell++) {
            if(globalBestPos < curCell-maxJump) {
                // find minCost
                long long min = numeric_limits<long long>::max();
                long long pos = curCell-maxJump;
                for(int k=curCell-maxJump; k < curCell; k++) {
                        long long curMin = minCost[k];
                        if(curMin < min) {
                            min = curMin;
                            pos = k;
                        }
                    }
                globalBestPos = pos;

            }
            cin >> vCell[curCell];
            long long curMinPos = minCost[globalBestPos] + vCell[curCell];
            minCost[curCell] = curMinPos;
            if(minCost[globalBestPos] >= curMinPos) {
                globalBestPos = curCell;
            }

            // cout << "Min on pos " << i << " is " << minCost[curCell] << endl;
        }

        cout << minCost[numCells-1] << endl;
    }
    return 0;
}
```

Light Pattern

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

using namespace std;

void decToBin(int dec, vector<int> &bin, int binLen) {
    int i=binLen-1;
    while(dec > 1) {
        if(dec%2==0) {
            bin[i]=0;
        } else {
            bin[i]=1;
        }
        dec=dec/2;
        i--;
    }
    bin[i] = (dec==0) ? 0 : 1;
    i--;
    while(i >=0) {
        bin[i] = 0;
        i--;
    }
}

void printVector(vector<int> &v) {
    for(vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << "␣";
    }
    cout << endl;
}

int computeSteps(vector<int> &bulb, vector<int> &resultPattern, int numBulbs, int patternLength) {
    int numPatterns = numBulbs/patternLength;
    int curReplacesToPattern = 0;
    int curReplacesToOpposite = 0;
    //go over every pattern
    for(int i=0; i<numPatterns; i++) {
        int replacesToPattern=0;
        int replacesToOpposite=0;
        for(int j=0; j<patternLength; j++) {
            int iBulb = i*patternLength+j;
            if(bulb[iBulb] != resultPattern[j]) {
                replacesToPattern++;
                // cout << "bulb " << iBulb << " is like opposite" << endl;
            } else {
                replacesToOpposite++;
                // cout << "bulb " << iBulb << " is like pattern" << endl;
            }
        }
        // cout << "pattern " << i << " needs " << replacesToPattern << " and " << replacesToOpposite << endl;
        int bothReplacesToPattern = curReplacesToPattern+replacesToPattern;
        int bothReplacesToOpposite = curReplacesToOpposite+replacesToOpposite;
        curReplacesToPattern = (bothReplacesToPattern < bothReplacesToOpposite + 1)
                               ? bothReplacesToPattern : bothReplacesToOpposite+1;
        curReplacesToOpposite = (bothReplacesToPattern+1 < bothReplacesToOpposite)
                               ? bothReplacesToPattern+1 : bothReplacesToOpposite;
    }

    return curReplacesToPattern;
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;

    for(int i=0; i < testCases; i++) {
        int numBulbs, patternLength, x;
        cin >> numBulbs >> patternLength >> x;

        vector<int> bulb(numBulbs);
        for(int j=0; j<numBulbs; j++) {
            int state;
```

```
            cin >> state;
            bulb[j] = state;
        }
        // cout << "bulbs: " ;
        // printVector(bulb);

        vector<int> binX(patternLength);
        decToBin(x, binX, patternLength);
        // cout << "result pattern: " ;
        // printVector(binX);
        cout << computeSteps(bulb, binX, numBulbs, patternLength) <<endl;

    }
}
```

Longest Path

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

using namespace std;

pair<int, int> longestPath(int node, int ancestor, const vector<vector<int> > &adjList) {
    const vector<int> *neighbors = &adjList[node];
    if(neighbors->size() == 1 && ancestor != -1) {
        return pair<int, int>(1,1);
    } else if (neighbors->size() == 2 && ancestor != -1) {
        for(vector<int>::const_iterator it = neighbors->begin(); it != neighbors->end(); it++) {
            if(*it != ancestor) {
                pair<int, int> child = longestPath(*it, node, adjList);
                if(child.first + 1 > child.second) {
                    return pair<int, int>(child.first+1, child.first+1);
                } else {
                    return pair<int, int>(child.first+1, child.second);
                }
            }
        }
    } else {
        int maxHeight = 0;
        int secondMaxHeight = 0;
        int maxPath = 0;
        for(vector<int>::const_iterator it = neighbors->begin(); it != neighbors->end(); it++) {
            if(*it != ancestor) {
                pair<int, int> child = longestPath(*it, node, adjList);
                if(child.first > maxHeight) {
                    secondMaxHeight = maxHeight;
                    maxHeight = child.first;
                } else if(child.first > secondMaxHeight) {
                    secondMaxHeight = child.first;
                }
                if(child.second > maxPath) {
                    maxPath = child.second;
                }
            }
        }
        int altMaxPath = maxHeight+secondMaxHeight+1;
        if(altMaxPath > maxPath) {
            return pair<int, int>(maxHeight+1, altMaxPath);
        } else {
            return pair<int, int>(maxHeight+1, maxPath);
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numVertices;
        cin >> numVertices;
        vector<vector<int> > adjList(numVertices);
        int accArray[numVertices];

        for(int i=0; i<numVertices-1; i++) {
            int v1, v2;
            cin >> v1 >> v2;
            adjList[v1].push_back(v2);
            adjList[v2].push_back(v1);
            accArray[i] = -1;
        }
        accArray[numVertices-1] = -1;
        cout << longestPath(0, -1, adjList).second << endl;
    }
    return 0;
}
```

Ants

```
// would be faster, when not every species has its own graph but only the property maps change.
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/graphviz.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> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

void printGraph(Graph g, WeightMap weight) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        std::cout << source(*eiter, g) << " <--> " << target(*eiter, g)
          << " with weight of " << weight[*eiter]
          << std::endl;
    }
}


int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numTreeNodes, numTreeEdges;
        int numSpecies, startTree, finishTree;
        cin >> numTreeNodes >> numTreeEdges >> numSpecies
                >> startTree >> finishTree;

        //construct graph for each species
        Graph speciesGraph[numSpecies];
        WeightMap weightMap[numSpecies];
        for(int i=0; i < numSpecies; i++) {
            speciesGraph[i] = Graph(numTreeNodes);
            weightMap[i] = get(edge_weight, speciesGraph[i]);
        }
        for(int e=0; e < numTreeEdges; e++) {
            int v1, v2;
            cin >> v1 >> v2;
            for(int i=0; i < numSpecies; i++) {
                Edge e;
                int w;
                cin >> w;
                tie(e,tuples::ignore) = add_edge(v1,v2,speciesGraph[i]);
                weightMap[i][e] = w;
            }

        }
        int hive[numSpecies];
        for(int i=0; i < numSpecies; i++) {
            int aHive;
            cin >> aHive;
            hive[i] = aHive;
            // cout << "species " << i << endl;
            // printGraph(speciesGraph[i], weightMap[i]);
        }

        // compute minimum spanning tree for each species
        vector<Edge> spanning_tree[numSpecies];
        Graph speciesMST[numSpecies];
        WeightMap weightMapMST[numSpecies];
        for(int i=0; i < numSpecies; i++) {
```

```
74              kruskal_minimum_spanning_tree(speciesGraph[i], back_inserter(spanning_tree[i]));
75              //construct new tree
76              for(vector<Edge>::iterator it=spanning_tree[i].begin(); it!=spanning_tree[i].end(); it++) {
77                  Edge e;
78                  tie(e,tuples::ignore) = add_edge(source(*it,speciesGraph[i]),
79                                                   target(*it,speciesGraph[i]),
80                                                   speciesMST[i]);
81                  weightMapMST[i][e] = weightMap[i][*it];
82              }
83              // cout << "species MST" << i << endl;
84              // printGraph(speciesMST[i], weightMapMST[i]);
85          }
86
87          // build combined graph
88          Graph combinedGraph(numTreeNodes);
89          WeightMap combinedWeightMap;
90          combinedWeightMap = get(edge_weight, combinedGraph);
91
92          graph_traits<Graph>::edge_iterator eiter, eiter_end;
93          //iterate over all possible edges
94          for (tie(eiter, eiter_end) = edges(speciesGraph[0]); eiter != eiter_end; ++eiter) {
95              Vertex u = get(vertex_index, speciesGraph[0], source(*eiter, speciesGraph[0]));
96              Vertex v = get(vertex_index, speciesGraph[0], target(*eiter, speciesGraph[0]));
97
98              int minWeight = numeric_limits<int>::max();
99              for(int s=0; s < numSpecies;s++) {
100                 Edge speciesEdge;
101                 bool hasEdge = false;
102                 tie(speciesEdge, hasEdge) = edge(u,v,speciesMST[s]);
103                 if(hasEdge) {
104                     int weight = get(weightMapMST[s],speciesEdge);
105                     if(weight < minWeight) {
106                         minWeight = weight;
107                     }
108                 }
109             }
110             if(minWeight < numeric_limits<int>::max()) {
111                 Edge e;
112                 tie(e,tuples::ignore) = add_edge(u,v,combinedGraph);
113                 combinedWeightMap[e] = minWeight;
114             }
115
116         }
117
118
119         // cout << "combined graph" << endl;
120         // printGraph(combinedGraph, combinedWeightMap);
121
122         // dijkstra
123         vector<int> distances(num_vertices(combinedGraph));
124         dijkstra_shortest_paths(combinedGraph, startTree,
125           distance_map(&distances[0]));
126         // for(vector<int>::iterator it=distances.begin(); it!=distances.end(); it++) {
127         //     cout << "in dlist: " << *it << endl;
128         // }
129         cout << distances[finishTree] << endl;
130     }
131 }
```

Bridges

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/biconnected_components.hpp>
#include <boost/graph/connected_components.hpp>

using namespace std;
using namespace boost;

namespace boost
{
  struct edge_component_t
  {
    enum
    { num = 555 };
    typedef edge_property_tag kind;
  }
  edge_component;
}


typedef adjacency_list<setS, vecS, undirectedS, no_property,
        property<edge_component_t, std::size_t> > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_component_t>::type ComponentMap;

void bridgesWithBiconnected(Graph &g, std::set<pair<int, int> > &criticalBridges) {
    ComponentMap componentMap = get(edge_component, g);
    size_t num_comps = biconnected_components(g, componentMap);
    vector<vector<Edge> > numEleOfComp(num_comps, vector<Edge>());

    IndexMap index = get(vertex_index, g);
    Traits::edge_iterator ei, ei_end;
    // cout << "new:"<<endl;
    for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
        // std::cout << "(" << index[source(*ei, g)]
        //       << "," << index[target(*ei, g)] << ") ";
        // cout << " comp: " << componentMap[*ei] << endl;
        numEleOfComp[componentMap[*ei]].push_back(*ei);
    }


    //every edge that is alone in a biconnected component is an important bridge
    for(int i=0; i<num_comps; i++) {
        if(numEleOfComp[i].size() ==1) {
            Edge e = numEleOfComp[i][0];
            int u = index[source(e, g)];
            int v = index[target(e, g)];
            pair<int,int> uv = (u<v)?pair<int,int>(u, v):pair<int,int>(v, u);
            criticalBridges.insert(uv);
        }
    }
}
void bridgesWithBruteForce(Graph &g, std::set<pair<int, int> > &criticalBridges) {
    vector<Edge> edgeVector;
    std::vector<int> componentMap(num_vertices(g));
    int components = connected_components(g, &componentMap[0]);

    Traits::edge_iterator ei, ei_end;
    for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
        edgeVector.push_back(*ei);
    }
    for(int i=0; i<edgeVector.size(); i++) {
        Edge anEdge = edgeVector[i];
        int sourceNode = source(anEdge, g);
        int targetNode = target(anEdge,g);
```

```
        remove_edge(sourceNode, targetNode, g);
        if(connected_components(g, &componentMap[0]) > components)
            criticalBridges.insert(pair<int,int>(source(anEdge, g), target(anEdge,g)));
        Edge e;
        tie(e, tuples::ignore) = add_edge(sourceNode,targetNode,g);
    }

}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numCities, numBridges;
        cin >> numCities >> numBridges;

        Graph g(numCities);
        for(int i=0; i < numBridges; i++) {
            Edge e;
            int u,v;
            cin >> u >> v;
            tie(e, tuples::ignore) = add_edge(u,v,g);
        }

        std::set<pair<int, int> > criticalBridges;

        // if(numCities > 50) {
            bridgesWithBiconnected(g,criticalBridges);
        // } else {
            // bridgesWithBruteForce(g,criticalBridges);
        // }

        int nCriticalBridges = criticalBridges.size();
        cout << nCriticalBridges << endl;
        for(set<pair<int,int> >::iterator eit=criticalBridges.begin();
                eit!=criticalBridges.end(); eit++){
             cout << (*eit).first
              << "␣" << (*eit).second << endl;
        }
    }
}
```

Build The Graph

```cpp
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/graphviz.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> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

pair<int, int> computeTask(Graph &g, WeightMap &weightMap) {
    vector<Edge> spanning_tree;
    kruskal_minimum_spanning_tree(g, back_inserter(spanning_tree));

    int sumMSTWeights = 0;
    // cout << "Print the edges in the MST:" << endl;
  for (vector < Edge >::iterator ei = spanning_tree.begin();
       ei != spanning_tree.end(); ++ei) {
    // cout << source(*ei, g) << " <--> " << target(*ei, g) << " with weight of " << weightMap[*ei] << endl;
    sumMSTWeights += weightMap[*ei];
  }

  //get vertex 0
  graph_traits<Graph>::vertex_iterator vert_0;
  tie(vert_0, tuples::ignore) = vertices(g);
  vector<int> distances(num_vertices(g));
  vector<Vertex> predecessor(num_vertices(g));

  dijkstra_shortest_paths(g, 0,
          predecessor_map(&predecessor[0]).distance_map(&distances[0]));
  return pair<int, int>(sumMSTWeights, *max_element(distances.begin(), distances.end()));

}
int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numVertices, numEdges;
        cin >> numVertices >> numEdges;
        Graph g(numVertices);
        WeightMap weightMap = get(edge_weight, g);
        for(int edgeIndex=0; edgeIndex < numEdges; edgeIndex++) {
            int vo,vi,w;
            cin >> vo >> vi >> w;
            Edge e;
            tie(e,tuples::ignore) = add_edge(vi,vo,g);
            weightMap[e] = w;
        }
        // cout << computeTask(g) << endl;
        pair<int, int> result = computeTask(g, weightMap);
        cout << result.first << "␣" << result.second << endl;


    }

    return 0;
}
```

Deleted Entries

```cpp
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/sequential_vertex_coloring.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

void printGraph(Graph g) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        std::cout << source(*eiter, g) << " <--> " << target(*eiter, g)
          << std::endl;
    }
}

// typedef vector<Vertex> PredecessorMap;
// typedef vector<int> ColorMap;

// class colorize : public default_bfs_visitor {
//     private:
//         PredecessorMap m_predecessor;
//         ColorMap m_colorMap;
//         ColorMap m_prevColorMap;
//         int maxColors;
//     public:
//         //give numColors as parameter
//         colorize(PredecessorMap p, ColorMap c, ColorMap prevC, int numColors)
//             : m_predecessor(p), m_colorMap(c),
//                 m_prevColorMap(prevC) {
//                     maxColors = numColors;
//
//             }
//         int newColor(int c) {
//             return (c+1)%maxColors;
//         }
//         void tree_edge(Edge e, Graph g) {
//             Vertex s = source(e, g);
//             Vertex t = target(e, g);
//             put(m_predecessor, t, s);
//             int color = newColor(m_prevColorMap[s]);
//             if(color == m_colorMap[s]) {
//                 color = newColor(color);
//             }
//             put(m_colorMap, t, color);
//             put(m_prevColorMap, s, color);
//         }
//         //void on_start_vertex default color
// };

void colorize(Graph &g, int numColors, vector<int> &color) {
    IndexMap index = get(vertex_index, g);
    vector<bool> visited(num_vertices(g), false);
    std::queue<Vertex> bfs_queue;

    //get start vertex
    Traits::vertex_iterator vi;
    tie(vi, tuples::ignore) = vertices(g);
    Vertex startV = *vi;

    color[index[startV]] = 0;
    visited[index[startV]] = true;
```

```
74      bfs_queue.push(startV);
75
76      int curColor = 0;
77      while (!bfs_queue.empty()) {
78          Vertex v = bfs_queue.front();
79          bfs_queue.pop();
80          int parentColor = color[index[v]];
81          // cout << "current " << index[v] << " col:" << color[index[v]] << endl;
82
83          Traits::adjacency_iterator adjV, adEnd;
84          tie(adjV, adEnd) = adjacent_vertices(v, g);
85          for (; adjV != adEnd; adjV++) {
86              int vIndex = index[*adjV];
87              if(!visited[vIndex]) {
88                  visited[vIndex] = true;
89                  curColor=(curColor+1)%numColors;
90                  if(curColor==parentColor)
91                      curColor=(curColor+1)%numColors;
92                  color[vIndex] = curColor;
93                  bfs_queue.push(*adjV);
94              }
95
96          }
97      }
98  }
99
100 int main() {
101     ios_base::sync_with_stdio(false);
102     int testCases = 0;
103     cin >> testCases;
104     for(int testCase=0; testCase < testCases; testCase++) {
105         int numStudents, numEdges, numGroups;
106         cin >> numStudents >> numEdges >> numGroups;
107         Graph g(numStudents);
108         for(int i = 0; i < numEdges; i++) {
109             Edge e;
110             int u,v;
111             cin >> u >> v;
112             tie(e, tuples::ignore) =add_edge(u,v,g);
113         }
114         // cout << "whole graph: " << endl;
115         // printGraph(g);
116
117         // vector<Edge> spanning_tree;
118         // //connected components
119         // kruskal_minimum_spanning_tree(g, back_inserter(spanning_tree));
120
121         // vector<vector<int> > groupList(numGroups);
122         Graph MST = g;
123         // Graph MST(spanning_tree.begin(), spanning_tree.end(), numStudents);
124         // for(vector<Edge>::iterator it=spanning_tree.begin(); it !=spanning_tree.end(); it++) {
125         //     Edge e;
126         //     tie(e, tuples::ignore) =add_edge(source(*it,g),target(*it,g),MST);
127         // }
128
129         // cout << "MS tree: " << endl;
130         // printGraph(MST);
131
132         //not enough vertices? -> no
133         if(num_vertices(MST) < numGroups) {
134             cout << "no" << endl;
135             continue;
136         }
137         vector<int> color(num_vertices(g), -1);
138         colorize(MST, numGroups, color);
139
140         // not connected? -> no
141         if(find(color.begin(), color.end(), -1) != color.end()) {
142             cout << "no" << endl;
143             continue;
144         }
145
146
147         vector< vector<int> > groups(numGroups, vector<int>());
148         for(int i=0; i<numStudents; i++) {
149             int studGroup = color[i];
```

```
150             groups[studGroup].push_back(i);
151         }
152 
153         cout << "yes" << endl;
154         for(vector<vector<int> >::iterator it=groups.begin(); it != groups.end(); it++) {
155             cout << (*it).size();
156             for(int i=0; i < (*it).size(); i++) {
157                 cout << "␣" <<(*it)[i];
158             }
159             cout << endl;
160         }
161 
162     }
163     return 0;
164 }
```

Shy Programmers

```cpp
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numCoders, numFriends;
        cin >> numCoders >> numFriends;
        Graph g(numCoders+1);

        for(int i=0; i<numFriends; i++) {
            Edge e;
            int u,v;
            cin >> u >> v;
            tie(e, tuples::ignore) =add_edge(u,v,g);
        }


        //add vertex, connected to all vertices
        for(int i=0; i<numCoders; i++) {
            Edge e;
            tie(e, tuples::ignore) = add_edge(numCoders,i,g);
        }
        if(boyer_myrvold_planarity_test(g)){
            cout << "yes" << endl;
        } else {
            cout << "no" << endl;
        }
    }
    return 0;
}
```

Algocoon Group

```
#include <boost/config.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <iostream>
#include <limits>
#include <queue>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph> GraphTraits;

void printGraph(Graph g) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        std::cout << source(*eiter, g) << "␣<-->␣" << target(*eiter, g)
          << std::endl;
    }
}

void findCut(Graph &g, EdgeCapacityMap &capacity,
                ResidualCapacityMap &res_capacity,
                std::set<Vertex> &ret,
                Vertex sourceV, Vertex sinkV) {
    IndexMap index = get(vertex_index, g);
    vector<bool> visited(num_vertices(g), false);
    std::queue<Vertex> bfs_queue;

    //get start vertex
    Vertex startV = sourceV;

    visited[index[startV]] = true;
    bfs_queue.push(startV);
    ret.insert(startV);

    while (!bfs_queue.empty()) {
        Vertex v = bfs_queue.front();
        bfs_queue.pop();

        graph_traits<Graph>::adjacency_iterator adjV, adEnd;
        GraphTraits::out_edge_iterator out_i, out_end;
        for (tie(out_i, out_end) = out_edges(v, g);
           out_i != out_end; ++out_i) {
            Edge e = *out_i;
            Vertex src = source(e, g), targ = target(e, g);
            int flow = capacity[e] - res_capacity[e];
            // cout << "edge " << src << " " << targ << " c:" << capacity[e] << " f:" << flow << endl;
            int vIndex = index[targ];
            if(flow < capacity[e] && !visited[vIndex]) {
                bfs_queue.push(targ);
                visited[vIndex] = true;
                ret.insert(targ);
            }
        }
    }
}

int cutCapacity(Graph &g, set<Vertex> &sSet, EdgeCapacityMap &capacity) {
    int cutCapacity=0;
    for(set<Vertex>::iterator it=sSet.begin();
```

```
                it != sSet.end(); it++) {
        Vertex v = *it;
        GraphTraits::out_edge_iterator out_i, out_end;
        for (tie(out_i, out_end) = out_edges(v, g);
           out_i != out_end; ++out_i) {
            Edge e = *out_i;
            Vertex targ = target(e, g);
            if(find(sSet.begin(), sSet.end(), targ) == sSet.end()) {
                //in T set
                cutCapacity+=capacity[e];
            }
        }
    }
    return cutCapacity;
}

pair<int, int> findBestSourceSinkSlow(Graph &g, int numFigures) {
    pair<int, int> bestSourceSink;
    int minMaxFlow = numeric_limits<int>::max();
    for(int i=0; i<numFigures; i++) {
        for(int j=i+1; j<numFigures; j++) {
            long flow = push_relabel_max_flow(g, i, j);
            if(flow < minMaxFlow) {
                bestSourceSink = pair<int, int>(i, j);
                minMaxFlow = flow;
            }
            flow = push_relabel_max_flow(g, j, i);
            if(flow < minMaxFlow) {
                bestSourceSink = pair<int, int>(j, i);
                minMaxFlow = flow;
            }
        }
    }
    return bestSourceSink;
}


pair<int, int> findBestSourceSinkFast(Graph &g, int numFigures) {
    pair<int, int> bestSourceSink;
    int minMaxFlow = numeric_limits<int>::max();
    int u = 0;
    for(int i=1; i<numFigures; i++) {
        long flow = push_relabel_max_flow(g, u, i);
        if(flow < minMaxFlow) {
            bestSourceSink = pair<int, int>(u,i);
            minMaxFlow = flow;
        }
    }
    for(int i=1; i<numFigures; i++) {
        long flow = push_relabel_max_flow(g, i, u);
        if(flow < minMaxFlow) {
            bestSourceSink = pair<int, int>(i,u);
            minMaxFlow = flow;
        }
    }
    return bestSourceSink;
}
int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numFigures, numLimbs;
        cin >> numFigures >> numLimbs;
        Graph g(numFigures);
        EdgeCapacityMap capacity = get(edge_capacity, g);
        ReverseEdgeMap rev_edge = get(edge_reverse, g);
        ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);

        for(int i=0; i<numLimbs;i++) {
            int u, v, c;
            cin >> u >> v >> c;
            Edge e, reverseE;
            tie(e, tuples::ignore) = add_edge(u,v,g);
            tie(reverseE, tuples::ignore) = add_edge(v, u, g);
            capacity[e] = c;
```

```
            capacity[reverseE] = 0;
            rev_edge[e] = reverseE;
            rev_edge[reverseE] = e;
        }

        pair<int, int> bestSourceSink = findBestSourceSinkFast(g, numFigures);
        int sourceV = bestSourceSink.first;
        int sinkV = bestSourceSink.second;
        long flow = push_relabel_max_flow(g, sourceV, sinkV);
        cout << flow << endl;
        std::set<Vertex> oneSet;
        findCut(g, capacity, res_capacity, oneSet, sourceV, sinkV);
        IndexMap index = get(vertex_index, g);
        cout << oneSet.size() << "␣";
        for(set<Vertex>::iterator it=oneSet.begin();
                it != oneSet.end(); it++) {
            cout << index[*it] << "␣";
        }
        cout << endl;
        // cout << "cut capacity " << cutCapacity(g, oneSet, capacity) << endl;

    }
}
```

Buddy Selection

```cpp
#include <boost/config.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <iostream>
#include <set>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numStudents, numChars, numCommChar;
        cin >> numStudents >> numChars >> numCommChar;

        Graph g(numStudents);
        vector<set<string> > chars(numStudents, set<string>());

        // cout << "testset:" << testCase << endl;
        for(int i=0; i < numStudents; i++) {
            for(int j=0; j < numChars; j++) {
                string characteristic;
                cin >> characteristic;
                chars[i].insert(characteristic);
            }
            for(int j=0; j < i; j++) {
                std::vector<string> inters;
                std::set_intersection(chars[i].begin(), chars[i].end(),
                                      chars[j].begin(), chars[j].end(),
                                      std::back_inserter(inters));
                if(inters.size() > numCommChar) {
                    // cout << "v:" << i << " u:" << j << endl;
                    //add edge
                    Edge e;
                    tie(e, tuples::ignore)=add_edge(i,j,g);
                }
            }
        }
        vector<Vertex> mate(numStudents);
        bool foundMatching = checked_edmonds_maximum_cardinality_matching(g, &mate[0]);
        if(matching_size(g, &mate[0])*2 >= numStudents) {
            cout << "not␣optimal" << endl;
        } else {
            cout << "optimal" << endl;
        }
    }
}
```

Satellites

```
/* minimum vertex cover */
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

void minVCoverBipDfs(Graph &g, vector<Vertex> &mate, vector<bool> &visited,
        int node, vector<bool> &isLeft) {
    visited[node] = true;
    Traits::out_edge_iterator out_i, out_end;
    for (tie(out_i, out_end) = out_edges(node, g);
           out_i != out_end; ++out_i) {
        Vertex targ = target(*out_i, g);
        int targetNode = get(vertex_index, g)[targ];
        if(visited[targetNode])
            continue;
        if(isLeft[node]) {
            if(mate[node] != targ) {
                minVCoverBipDfs(g, mate, visited, targetNode, isLeft);
            }
        } else {
            if(mate[node] == targ) {
                minVCoverBipDfs(g, mate, visited, targetNode, isLeft);
            }
        }
    }
}


/* takes bipartite graph and a vector that specifies whether
 * a node is on the left side.
 * Returns a vector containing the min vertex cover
 */
vector<int> findMinVertexCoverBipartite(Graph &g, vector<bool> &isLeft) {
    int numNodes = isLeft.size();
    vector<Vertex> mate(numNodes);
    edmonds_maximum_cardinality_matching(g, &mate[0]);

    vector<bool> visited(numNodes, false);
    const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
    for(int i=0;i<numNodes;i++) {
        if(isLeft[i] && mate[i] == NULL_VERTEX) {
            visited[i]=true;
        }
    }

    for(int i = 0; i < numNodes; i++) {
        if(isLeft[i] && visited[i]) {
            vector<bool> dfsVisited(numNodes, false);
            minVCoverBipDfs(g, mate, visited, i, isLeft);
        }
    }

    vector<int> minVertexCover;
    for(int i = 0; i < numNodes; i++) {
        if(isLeft[i] && !visited[i]) {
            minVertexCover.push_back(i);
        } else if(!isLeft[i] && visited[i]){
            minVertexCover.push_back(i);
        }
    }
    return minVertexCover;
```

```
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numStations, numSatellites, numLinks;
        cin >> numStations >> numSatellites >> numLinks;
        Graph g(numStations + numSatellites);
        vector<bool> isLeft(numStations+numSatellites,false);
        for(int i=0; i<numLinks; i++) {
            int station, satellite;
            cin >> station >> satellite;
            Edge e;
            tie(e, tuples::ignore) =add_edge(station,numStations+satellite,g);
            isLeft[station] = true;
        }

        vector<int> minVertexCover = findMinVertexCoverBipartite(g, isLeft);

        vector<int> minStations;
        vector<int> minSatellites;

        for(int i=0; i<minVertexCover.size(); i++) {
            int node = minVertexCover[i];
            if(node<numStations)
                minStations.push_back(node);
            else
                minSatellites.push_back(node-numStations);
        }

        cout << minStations.size() << "␣" << minSatellites.size() << endl;
        for(vector<int>::iterator it=minStations.begin(); it!=minStations.end(); it++) {
            cout << *it << "␣";
        }
        for(vector<int>::iterator it=minSatellites.begin(); it!=minSatellites.end(); it++) {
            cout << *it << "␣";
        }

        cout << endl;
    }
}
```

Kingdom Defense

```
#include <boost/config.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <iostream>
#include <limits>
#include <queue>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph> GraphTraits;

void printGraph(Graph g, EdgeCapacityMap &capacity, vector<pair<int, int> > &locProps) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        if(capacity[*eiter] > 0) {
            int aSource = source(*eiter, g);
            int aTarget = target(*eiter, g);
            std::cout << aSource <<":␣("
                << locProps[aSource].first<< "|" << locProps[aSource].second << ")" <<
                "␣-" << capacity[*eiter] << "->␣" << aTarget
                << ":␣(" << locProps[aTarget].first<< "|" << locProps[aTarget].second << ")"
              << std::endl;
        }
    }
}

void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u,v,g);
    tie(reverseE, tuples::ignore) = add_edge(v, u, g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numVertices, numEdges;
        cin >> numVertices >> numEdges;
        vector<pair<int, int> > locProp(numVertices, pair<int, int>(0,0));

        for(int i=0; i<numVertices; i++) {
            int numStationed,numNeeded;
            cin >> numStationed >> numNeeded;
            locProp[i] = pair<int, int>(numStationed, numNeeded);
        }

        Graph g(numVertices);
        EdgeCapacityMap capacity = get(edge_capacity, g);
        ReverseEdgeMap rev_edge = get(edge_reverse, g);
        ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);
        for(int i=0; i<numEdges; i++) {
            int from,to, minPassing, maxPassing;
            cin >> from >> to >> minPassing >> maxPassing;

```

```
            //force soldiers to walk along
            locProp[from].first = locProp[from].first - minPassing;
            locProp[to].first = locProp[to].first + minPassing;
            maxPassing-=minPassing;

            //build actual graph
            addFlowEdge(g, capacity, rev_edge, from, to, maxPassing);
        }

        // printGraph(g, capacity, locProp);

        const int SOURCE = numVertices;
        const int SINK = numVertices+1;
        int numNonForcedSoldiers=0;
        //add source and sink
        for(int i=0; i<numVertices; i++) {
            int soldiersStationed = locProp[i].first;
            int soldiersNeeded = locProp[i].second;
            if(soldiersStationed > 0) {
                addFlowEdge(g, capacity, rev_edge, SOURCE, i, soldiersStationed);
                addFlowEdge(g, capacity, rev_edge, i, SINK, soldiersNeeded);
            } else {
                soldiersNeeded-=soldiersStationed;
                addFlowEdge(g, capacity, rev_edge, i, SINK, soldiersNeeded);
            }
            numNonForcedSoldiers+=soldiersNeeded;
        }
        // cout << "after adding source and sink:" << endl;
        // printGraph(g, capacity, locProp);

        //compute maxFlow
        long flow = push_relabel_max_flow(g, SOURCE, SINK);
        // long flow = 0;

        // cout << numNonForcedSoldiers << endl;
        if(flow==numNonForcedSoldiers) {
            cout << "yes";
        } else {
           cout << "no";
        }
        cout << endl;
    }
}
```

Coin Tossing

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

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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 graph_traits<Graph> GraphTraits;
typedef GraphTraits::vertex_descriptor Vertex;
typedef GraphTraits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;



void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u,v,g);
    tie(reverseE, tuples::ignore) = add_edge(v, u, g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

int sum(vector<int> &aVector) {
    int sum_of_elems = 0;
    for(std::vector<int>::iterator j=aVector.begin();j!=aVector.end();++j)
        sum_of_elems += *j;
    return sum_of_elems;
}


int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numPlayers, numRounds;
        int outcome = 0; //-1 false, 0 don't know, 1 true
        cin >> numPlayers >> numRounds;

        vector<int> scoreBoard(numPlayers, 0);
        const int SOURCE = numPlayers;
        const int SINK = numPlayers+1;
        vector<vector<int> > helperNode(numPlayers, vector<int>(numPlayers, -1));
        Graph g;
        EdgeCapacityMap capacity = get(edge_capacity, g);
        ReverseEdgeMap rev_edge = get(edge_reverse, g);
        ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);
        int outOfSink = 0;

        for(int i=0; i<numRounds;i++) {
            int playerA, playerB, outcome;
            cin >> playerA >> playerB >> outcome;
            if(outcome == 1) {
                scoreBoard[playerA] -=1;
            } else if(outcome == 2) {
                scoreBoard[playerB] -=1;
            } else {
                //construct graph
                if(helperNode[playerA][playerB] == -1) {
```

```
                    //no helper node before
                    int aHelperNode = SINK+1+i;
                    helperNode[playerA][playerB] = aHelperNode;
                    helperNode[playerB][playerA] = aHelperNode;

                    addFlowEdge(g, capacity, rev_edge, SOURCE, aHelperNode, 1);
                    addFlowEdge(g, capacity, rev_edge, aHelperNode, playerA, 1);
                    addFlowEdge(g, capacity, rev_edge, aHelperNode, playerB, 1);
                } else {
                    int aHelperNode = helperNode[playerA][playerB];
                    Edge sourceToHelper, helperToA, helperToB;
                    sourceToHelper = edge(SOURCE, aHelperNode, g).first;
                    helperToA = edge(aHelperNode, playerA, g).first;
                    helperToB = edge(aHelperNode, playerB, g).first;
                    capacity[sourceToHelper] += 1;
                    capacity[helperToA] += 1;
                    capacity[helperToB] += 1;
                }
                outOfSink++;
            }
        }
        for(int i=0; i<numPlayers; i++) {
            int desiredScore;
            cin >> desiredScore;
            scoreBoard[i] += desiredScore;
            if(scoreBoard[i] < 0) {
                //a player won more often than in the desired scoreboard
                outcome = -1;
            }
        }
        if(outcome >= 0 && outOfSink==sum(scoreBoard)) {

            // if(scoreBoard.sum() < outCapacitySource) {
                // outcome = -1;
            // }

            // add edges to sink
            for(int i=0; i<numPlayers; i++) {
                if(scoreBoard[i] > 0) {
                    addFlowEdge(g, capacity, rev_edge, i, SINK, scoreBoard[i]);
                }
            }


            long flow = push_relabel_max_flow(g, SOURCE, SINK);
            if(flow != sum(scoreBoard)) {
                // cout << "flow != sum(scoreBoard)" << endl;
                cout << "no" << endl;
            } else {
                cout << "yes" << endl;
                // cout << flow << " " << sum(scoreBoard) << endl;
            }
        } else {
            // cout << "a player won more often than in the desired scoreboard" << endl;
            cout << "no" << endl;
        }
    }
}
```

Antenna

```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>

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

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;
}


int main() {
    ios_base::sync_with_stdio(false);

    while(true) {
        int numCitizens;
        cin >> numCitizens;
        if(numCitizens == 0)
            break;

        K::Point_2 P[numCitizens];
        for(int i=0; i<numCitizens; i++) {
            double x,y;
            cin >> x >> y;
            P[i] = K::Point_2(x, y);
        }
        Min_circle radio(P, P+numCitizens, true);
        Traits::Circle c = radio.circle();
        K::FT radius = sqrt(c.squared_radius());
        cout << std::setiosflags(std::ios::fixed) << setprecision(0) << ceil_to_double(radius) << endl;
        // cout << std::round(CGAL::to_double(radius)) << endl;
    }
}
```

Almost Antenna

```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>

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

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;
}

double computeSlow(vector<K::Point_2> P) {
    K::FT radius;
    for(vector<K::Point_2>::iterator it = P.begin();
            it != P.end(); it++) {
        vector<K::Point_2> tmpP(P.begin(), it);
        tmpP.insert(tmpP.end(), it+1, P.end());

        Min_circle radio(tmpP.begin(), tmpP.end(), true);
        Traits::Circle c = radio.circle();
        K::FT aRadius = c.squared_radius();
        if(it!=P.begin()) {
            if(aRadius < radius)
                radius = aRadius;
        } else {
            radius = aRadius;
        }
    }
    return ceil_to_double(sqrt(radius));
}

//remove only the defining points of the whole minCircle
double computeFast(vector<K::Point_2> P) {
    Min_circle radio(P.begin(), P.end(), true);

    K::FT radius;
    for(int i=0; i<radio.number_of_support_points(); i++) {
        K::Point_2 sp = radio.support_point(i);
        vector<K::Point_2> tmpP(P.begin(), P.end());
        vector<K::Point_2>::iterator toSp = find(tmpP.begin(),tmpP.end(),sp);
        tmpP.erase(toSp);

        Min_circle partRadio(tmpP.begin(), tmpP.end(), true);
        Traits::Circle c = partRadio.circle();
        K::FT aRadius = c.squared_radius();
        if(i!=0) {
            if(aRadius < radius)
                radius = aRadius;
        } else {
            radius = aRadius;
        }

    }

    return ceil_to_double(sqrt(radius));
}

int main() {
    ios_base::sync_with_stdio(false);
```

```
74
75      while(true) {
76          int numCitizens;
77          cin >> numCitizens;
78          if(numCitizens == 0)
79              break;
80
81          vector<K::Point_2> P(numCitizens);
82          for(int i=0; i<numCitizens; i++) {
83              double x,y;
84              cin >> x >> y;
85              P[i] = K::Point_2(x, y);
86          }
87              cout << std::setiosflags(std::ios::fixed) << setprecision(0)
88                  << computeFast(P) << endl;
89      }
90  }
```

Hit

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

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel K;


bool hitsObstacle(int numObstacles) {

    bool ret = false;
    double x, y, a, b;
    cin >> x >> y >> a >> b;
    K::Ray_2 phil(K::Point_2(x,y), K::Point_2(a,b));
    // cout << "Phil " << phil << endl;

    for(int i=0; i<numObstacles; i++) {
        double r,s,t,u;
        cin >> r >> s >> t >> u;
        K::Segment_2 obstacle(K::Point_2(r,s), K::Point_2(t,u));
        if(!ret && CGAL::do_intersect(phil, obstacle)) {
            ret=true;
        }
    }

    return ret;
}

int main() {
    ios_base::sync_with_stdio(false);

    while(true) {
        int numObstacles;
        cin >> numObstacles;
        // cout << "numObstacles " << numObstacles << endl;
        if(numObstacles == 0) {
            return 0;
        }
        if(hitsObstacle(numObstacles)) {
            cout << "yes" << endl;
        } else {
            cout << "no" << endl;
        }
    }
}
```

First Hit

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

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 P;
typedef K::Segment_2 S;

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 hitsObstacle(int numObstacles, K::Point_2 &isPos) {

    bool isHit = false;
    double x, y, a, b;
    cin >> x >> y >> a >> b;
    P philStart(x,y);
    K::Ray_2 phil(philStart, K::Point_2(a,b));
    P minIntersectionPoint;
    // cerr << "Phil " << phil << endl;

    for(int i=0; i<numObstacles; i++) {
        // cerr << "Loop " << i << endl;
        double r,s,t,u;
        cin >> r >> s >> t >> u;
        K::Segment_2 obstacle(K::Point_2(r,s), K::Point_2(t,u));
        if(CGAL::do_intersect(phil, obstacle)) {
            // cerr << "Intersection " << i << endl;
            P intersectionPoint;
            CGAL::Object o = CGAL::intersection(phil, obstacle);
            if(const P* op = CGAL::object_cast<P>(&o)) {
                intersectionPoint = *op;
            } else if(const S* os = CGAL::object_cast<S>(&o)) {
                //take the segment end that is nearer
                intersectionPoint = CGAL::squared_distance(philStart, os->source())
                                    < CGAL::squared_distance(philStart, os->target())
                                    ? os->source() : os->target();
            }
            else {
                throw std::runtime_error("strange segment intersection");
            }

            // cerr << "Know Intersection " << i << endl;
            if(isHit) {
                if(CGAL::squared_distance(philStart, intersectionPoint)
                        < CGAL::squared_distance(philStart, minIntersectionPoint)) {
                    // cerr << "new intersection" << i << endl;
                    minIntersectionPoint = intersectionPoint;
                }
            } else {
                // cerr << "min intersection NULL " << i << endl;
                minIntersectionPoint = intersectionPoint;
            }

            // cerr << "end of loop" << i << endl;
            isHit=true;
        }
    }

    // cerr << "return" << endl;
    if(isHit) {
        isPos = minIntersectionPoint;
    }
    // cerr << "return" << endl;
    return isHit;
}

int main() {
```

```
74      ios_base::sync_with_stdio(false);
75
76      while(true) {
77          // cerr << "beginLoop" << endl;
78          int numObstacles;
79          cin >> numObstacles;
80          // cerr << "numObstacles " << numObstacles << endl;
81          if(numObstacles == 0) {
82              return 0;
83          }
84          K::Point_2 isPos;
85          if(hitsObstacle(numObstacles, isPos)) {
86              // cerr << "afterreturn" << endl;
87              cout << std::setiosflags(std::ios::fixed) << setprecision(0)
88                   << floor_to_double(isPos.x())
89                   << "␣"
90                   << floor_to_double(isPos.y()) << endl;
91              // cerr << "afterout" << endl;
92          } else {
93              cout << "no" << endl;
94          }
95          // cerr << "endloop" << endl;
96      }
97  }
```

Search Snippets

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

using namespace std;

const int MAX = numeric_limits<int>::max();

pair<int,int> minOfVector(vector<int> &v) {
        int min = MAX;
        int minPos;
        for(vector<int>::iterator it=v.begin(); it!= v.end(); it++) {
                if(min > *it) {
                        min = *it;
                        minPos = it-v.begin();
                }
        }
        return pair<int,int>(min,minPos);
}

int computeInterval(priority_queue<pair<int,int> > &Q, int numWords) {
        vector<int> currentPosition(numWords);
        vector<bool> hasObserved(numWords, false);
        int numNotObserved = numWords;

        int minInterval = MAX;
        int interval1 = 0;
        int interval2 = 0;
        while(Q.size() > 0){
                pair<int, int> occurence = Q.top();
                Q.pop();
                int word = occurence.second;
                int pos = -occurence.first;
//              cout << "current wordpos of " << word << " is " << pos << endl;

                currentPosition[word] = pos;
                if(hasObserved[word] == false) {
                        hasObserved[word] = true;
                        numNotObserved--;
                }
                if(numNotObserved > 0)
                        continue;

                pair<int,int> aMinOfVector = minOfVector(currentPosition);
                int curInterval = pos - aMinOfVector.first +1;

                if(curInterval<minInterval) {
                        minInterval=curInterval;
                        interval1 =aMinOfVector.second;
                        interval2 =pos;
//                      cout << minInterval << " between ["<<  interval1<<  ","<<  interval2<<  "]" << endl;
                }
        }

        return minInterval;
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;
    for(int i=0; i<testCases; i++) {
        int numWords;
        cin >> numWords;
        int numAppears[numWords];
        for(int j=0;j<numWords; j++) {
                int num;
                cin >> num;
                numAppears[j] = num;
        }

        priority_queue<pair<int,int> > Q;
```

```
74
75          for(int j=0;j<numWords; j++) {
76                  for(int k=0; k<numAppears[j]; k++) {
77                          pair<int, int> occurence;
78                          int position;
79                          cin>>position;
80                          Q.push(pair<int,int>(-position, j));
81                  }
82          }
83          cout << computeInterval(Q, numWords) << endl;
84
85          }
86  }
```

Bistro

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef CGAL::Segment_2<K> Segment;
int main() {
    std::ios_base::sync_with_stdio(false);

    int numBistros;
    while(std::cin >> numBistros) {
        if(numBistros==0) break;

        // std::cout << numBistros << std::endl;
        std::vector<K::Point_2> bistro(numBistros);
        Triangulation t;
        for(int i=0; i<numBistros;i++) {
            double x, y;
            std::cin >> x >> y;
            bistro[i] = K::Point_2(x,y);
            // std::cout << bistro[i] << std::endl;
        }
        t.insert(bistro.begin(), bistro.end());

        int numNewBistros;
        std::cin >> numNewBistros;
        // std::cout << numNewBistros << std::endl;
        for(int i=0; i<numNewBistros;i++) {
            double x, y;
            std::cin >> x >> y;
            K::Point_2 newBistro(x,y);
            Triangulation::Vertex_handle nearestBistro = t.nearest_vertex(newBistro);
            Segment shortestPath(nearestBistro->point(), newBistro);
            std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0)
                << shortestPath.squared_length() << std::endl;
        }


    }
}
```

Germs

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef CGAL::Segment_2<K> Segment;
typedef K::Point_2 Point;

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;
}

int main() {
    std::ios_base::sync_with_stdio(false);

    int numBacteria;
    while(std::cin >> numBacteria) {
        if(numBacteria==0) break;
        int l,b,r,t;
        std::cin >> l >> b >> r >> t;
        std::vector<Segment> dish(4);
        dish[0] = Segment(Point(l,b),Point(r,b));
        dish[1] = Segment(Point(r,b),Point(r,t));
        dish[2] = Segment(Point(r,t),Point(l,t));
        dish[3] = Segment(Point(l,t),Point(l,b));


        std::vector<K::Point_2> bacteria(numBacteria);
        for(int i=0;i<numBacteria;i++) {
            int x,y;
            std::cin >> x >> y;
            bacteria[i] = K::Point_2(x,y);
        }

        std::priority_queue<K::FT> q;
        if(numBacteria > 1) {
            Triangulation tria;
            tria.insert(bacteria.begin(), bacteria.end());

            Triangulation::Vertex_iterator v = tria.finite_vertices_begin();
            for(Triangulation::Vertex_iterator v = tria.finite_vertices_begin();
                v != tria.finite_vertices_end(); ++v) {
                bool firstEdge = true;
                K::FT minDist;


                Triangulation::Edge_circulator c = tria.incident_edges(v);
                do {
                    if(!tria.is_infinite(c)) {
                        Triangulation::Vertex_handle v1 = c->first->vertex((c->second + 1) % 3);
                        Triangulation::Vertex_handle v2 = c->first->vertex((c->second + 2) % 3);

                        K::FT candidateDist = Segment(v1->point(),v2->point()).squared_length();
                        if(firstEdge == true || minDist > candidateDist) {
                            minDist = candidateDist;
                            // std::cout << "canididate minimal distance for "<< v->point() << " between " << ↙
                                ↳ v1->point() << " and " << v2->point() << ": " << minDist << std::endl;
                            firstEdge = false;
                        }
                    }
                } while(++c != tria.incident_edges(v));
                //other bacterium extends as well
                minDist = sqrt(minDist)/2;
                // std::cout << "tentative minimal distance for "<< v->point() <<": " << minDist << std::endl;

                //compare with dish
```

```
                for(std::vector<Segment>::iterator s = dish.begin(); s!=dish.end(); s++) {
                    K::FT candidateDist = sqrt(squared_distance(v->point(),*s));
                    if(firstEdge == true || minDist > candidateDist) {
                        minDist = candidateDist;
                        firstEdge = false;
                    }

                }
                //compute time
                K::FT time = sqrt(minDist-0.5);

                q.push(-time);
                // std::cout << "minimal distance for "<< v->point() <<": " << minDist << std::endl;

            }

        } else {
            bool firstEdge = true;
            K::FT minDist;

            //compare with dish
            for(std::vector<Segment>::iterator s = dish.begin(); s!=dish.end(); s++) {
                K::FT candidateDist = squared_distance(bacteria[0],*s);
                if(firstEdge == true || minDist > candidateDist) {
                    minDist = candidateDist;
                    bool firstEdge = false;
                }

            }
            // std::cout << "minimal distance: " << minDist << std::endl;
            //compute time
            K::FT time = sqrt(sqrt(minDist)-0.5);

            q.push(-time);

        }

        std::cout << ceil_to_double(-q.top()) << "␣";
        int mid = numBacteria/2;
        for(int i=0; i < mid; i++) {
            q.pop();
            // std::cout << "1popped" << std::endl;
        }
        std::cout << ceil_to_double(-q.top()) << "␣";
        // std::cout << "mid: " << mid << "numBacteria: " << numBacteria  << std::endl;
        for(int i=mid; i < numBacteria-1; i++) {
            q.pop();
            // std::cout << "2popped" << std::endl;
        }
        std::cout << ceil_to_double(-q.top());
        std::cout << std::endl;
    }
}
```

Graypes

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef CGAL::Segment_2<K> Segment;

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;
}

int main() {
    std::ios_base::sync_with_stdio(false);

    int numGraypes;
    while(std::cin >> numGraypes) {
        if(numGraypes==0) break;
        std::vector<K::Point_2> graype(numGraypes);
        Triangulation t;
        for(int i=0; i<numGraypes;i++) {
            double x, y;
            std::cin >> x >> y;
            graype[i] = K::Point_2(x,y);
        }
        t.insert(graype.begin(), graype.end());

        Segment minEdge;
        bool firstEdge = true;
        // output all edges
        for(Edge_iterator e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e) {
            if(firstEdge == true || minEdge.squared_length() > t.segment(e).squared_length()) {
              firstEdge = false;
              minEdge = t.segment(e);
            }
        }
        std::cout << ceil_to_double(50*sqrt(minEdge.squared_length())) << std::endl;
    }
}
```

H1N1

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
#include <iostream>
#include <vector>
#include <algorithm>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<K::FT,K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>  Triangulation;
typedef Triangulation::Face_iterator Face_iterator;
typedef Triangulation::All_vertices_iterator Vertex_iterator;
typedef Triangulation::Face_handle Face_handle;
typedef Triangulation::Edge Edge;
typedef CGAL::Point_2<K> Point;


struct faceComparator {
    bool operator() (const Face_handle &x, const Face_handle &y) const {
        return (x->info() > y->info());
    }
};

bool isInfinite(Triangulation &t, Face_handle &f) {
    return t.is_infinite(Edge(f,0))
            || t.is_infinite(Edge(f,1))
            || t.is_infinite(Edge(f,2));

}

bool has_infinite_vertex(Triangulation &t, Face_handle &f) {
    return t.is_infinite(f->vertex(0))
        || t.is_infinite(f->vertex(1))
        || t.is_infinite(f->vertex(2));
}

void initFaces(Triangulation &t, int startValue) {
    //initialize escape radius
    for (Triangulation::All_faces_iterator i = t.all_faces_begin();
            i != t.all_faces_end(); i++) {
        i->info() = K::FT(startValue);
    }
}


void maxEscapeRadiusPerFace(Triangulation &t) {
    //initialize escape radius
    for (Face_iterator i = t.finite_faces_begin();
            i != t.finite_faces_end(); i++) {
        i->info() = K::FT(0);
    }


    //initialize boundary
    Triangulation::Face_circulator fib = t.incident_faces(t.infinite_vertex());
    Triangulation::Face_circulator fie = fib;

    do {
        for(int i=0; i<3; i++) {
            if(!t.is_infinite(fib->neighbor(i))) {
                fib->neighbor(i)->info()
                    = t.segment(Triangulation::Edge(fib,i)).squared_length()/4;
                Face_handle mf = fib->neighbor(i);
                break;
            }
        }
    } while(++fib != fie);

    //build search data structure -> Dijkstra
    std::set<Face_handle, faceComparator> faces;
    for(Face_iterator i=t.finite_faces_begin();
```

```
            i!=t.finite_faces_end(); i++) {
        faces.insert(Face_handle(i));
    }

    while(!faces.empty()) {
        //remove face with the best escape radius
        Face_handle mf = *(faces.begin());
        faces.erase(faces.begin());
        // std::cout << mf->info() << ": " << mf->vertex(1)->point() << "," << mf->vertex(2)->point() << "," << ↙
            ↳ mf->vertex(3)->point() << std::endl;
        //and relax all edges
        for(int i=0; i<3; i++) {
            if(!t.is_infinite(mf->neighbor(i))) {
                K::FT tentativeRadius = t.segment(Edge(mf,i)).squared_length()/4;
                K::FT w;
                if(mf->info() < tentativeRadius) {
                    w = mf->info();
                } else {
                    w = tentativeRadius;
                }

                if(w > mf->neighbor(i)->info()) {
                        faces.erase(mf->neighbor(i));
                        mf->neighbor(i)->info() = w;
                        faces.insert(mf->neighbor(i));
                }
            }
        }
    }
}


bool canEscape(Triangulation &t, Point &p, K::FT &squaredRadius) {
    if(squaredRadius <= 0) {
        std::cout << "one" << std::endl;
        return true;
    }

    Face_handle f = t.locate(p);

    // std::cout << CGAL::squared_distance(p, t.nearest_vertex(p,f)->point()) << " " << squaredRadius << std::endl;
    if(CGAL::squared_distance(p, t.nearest_vertex(p,f)->point()) < squaredRadius) {
        // std::cout << "three" << std::endl;
        return false;
    }

    if(t.is_infinite(f)) {
        // std::cout << "two" << std::endl;
        return true;
    }

    std::cout << f->info() << ":␣" << f->vertex(0)->point() << "," << f->vertex(1)->point() << "," << ↙
        ↳ f->vertex(2)->point() << (has_infinite_vertex(t,f) ? "␣has␣infinite␣vertex" : "") << std::endl;
    return(f->info() >= squaredRadius);
}

bool canEscapeSlow(Triangulation &t, Point &s, K::FT &r, int queryNumber) {
    int i = queryNumber;
    if(r <= 0)
        return true;
    Face_handle f = t.locate(s);
    if(CGAL::squared_distance(s, t.nearest_vertex(s, f)->point()) < r)
        return false;

    // DFS
    std::vector<Face_handle> stack;
    stack.push_back(f);
    f->info() = i;
    while(!stack.empty()) {
        f = stack.back();
        stack.pop_back();
        // std::cout << f->info() << ": " << f->vertex(0)->point() << "," << f->vertex(1)->point() << "," << ↙
            ↳ f->vertex(2)->point() << (has_infinite_vertex(t,f) ? " has infinite vertex" : "") << std::endl;
        if(t.is_infinite(f))
            return true;
        for(int j = 0; j < 3; ++j) {
```

```
147             if(f->neighbor(j)->info() < i
148                     && t.segment(Triangulation::Edge(f,j)).squared_length() >= 4* r) {
149                 stack.push_back(f->neighbor(j));
150                 // std::cout << "push back" << std::endl;
151                 f->neighbor(j)->info() = i;
152             }
153         }
154     }
155     // std::cout << "four" << std::endl;
156     return false;
157 }
158 
159 int main() {
160     std::ios_base::sync_with_stdio(false);
161 
162     int numVertices;
163     while(std::cin >> numVertices) {
164         if(numVertices <= 0) break;
165 
166         std::vector<Point> verticesList(numVertices);
167         for(int i=0; i<numVertices; i++) {
168             int x,y;
169             std::cin >> x >> y;
170             Point p(x,y);
171             verticesList[i] = p;
172         }
173 
174         Triangulation t;
175         t.insert(verticesList.begin(), verticesList.end());
176         // maxEscapeRadiusPerFace(t);
177 
178         // std::cout << "faces: " << std::endl;
179         // for(Face_iterator i = t.faces_begin(); i !=t.faces_end(); i++) {
180         //     std::cout << i->info() << ": " << i->vertex(1)->point() << "," << i->vertex(2)->point() << "," << ↙
                ↳ i->vertex(3)->point() << std::endl;
181         // }
182         // std::cout << "faces end " << std::endl;
183         initFaces(t,-1);
184 
185         int numCircles;
186         std::cin >> numCircles;
187         for(int i=0; i<numCircles; i++) {
188             int x,y;
189             K::FT squaredRadius;
190             std::cin >> x >> y >> squaredRadius;
191             Point p(x,y);
192             std::cout << (canEscapeSlow(t, p, squaredRadius,i) ? "y" : "n");
193         }
194 
195         std::cout << std::endl;
196     }
197 }
```

Hiking Maps

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
#include <set>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef pair<Point, Point> Segment;
typedef vector<Point> Triangle;

pair<int,int> minOfVector(vector<int> &v) {
        int min = numeric_limits<int>::max();
        int minPos;
        for(vector<int>::iterator it=v.begin(); it!= v.end(); it++) {
                if(min > *it) {
                        min = *it;
                        minPos = it-v.begin();
                }
        }
        return pair<int,int>(min,minPos);
}

bool contained(const Segment &s, const Triangle &t) {
    //test whether endpoints of the segments have the same orientation to
    //the points on the side of the triangle as those points have to each
    //other (same orientation or collinear)

    bool right1 = CGAL::right_turn(t[0], t[1], t[2]);
    bool right2 = CGAL::right_turn(t[2], t[3], t[4]);
    bool right3 = CGAL::right_turn(t[4], t[5], t[0]);

    CGAL::Orientation orient1 = !right1 ? CGAL::RIGHT_TURN : CGAL::LEFT_TURN;
    CGAL::Orientation orient2 = !right2 ? CGAL::RIGHT_TURN : CGAL::LEFT_TURN;
    CGAL::Orientation orient3 = !right3 ? CGAL::RIGHT_TURN : CGAL::LEFT_TURN;


    bool ret = CGAL::orientation(t[0], t[1], s.first) != orient1
                && CGAL::orientation(t[2], t[3], s.first) != orient2
                && CGAL::orientation(t[4], t[5], s.first) != orient3
                && CGAL::orientation(t[0], t[1], s.second) != orient1
                && CGAL::orientation(t[2], t[3], s.second) != orient2
                && CGAL::orientation(t[4], t[5], s.second) != orient3;
    return ret;
}

void myApproach(const vector<Segment>& legs, int numLegs, int numMaps) {
    vector<Triangle > maps(numMaps, Triangle(6));
    int minInterval = numeric_limits<int>::max();
    vector<int> newestCover(numLegs); //map from leg to triangle
    vector<bool> isCovered(numLegs, false); //for the beginning to test whether all are overed
    int numNotCovered = numLegs;

    int curInterval = 0;
    bool mustRecompute = true;
    int curMinIndex = -1;

    for(int i=0;i<numMaps;i++) {
        for(int j=0; j<6; j++) {
            int x,y;
            cin >> x >> y;
            maps[i][j] = Point(x,y);
        }

        //test intersection
        for(int j=0; j<legs.size(); j++) {
            if(contained(legs[j], maps[i])) {
                newestCover[j] = i;
                if(j==curMinIndex) mustRecompute = true;
                if(isCovered[j] == false) {
```

```
                    isCovered[j] = true;
                    numNotCovered--;
                }
            }

        }
        if(numNotCovered > 0)
            continue;

        if(mustRecompute) {
            pair<int,int> resultOfMin = minOfVector(newestCover);
            curInterval = i-resultOfMin.first+1;
            curMinIndex = resultOfMin.second;
            mustRecompute = false;
            if(curInterval < minInterval)
                minInterval=curInterval;

        }
    }

    cout << minInterval << endl;

}

void benApproach(const vector<Segment> &legs, int numLegs, int numMaps) {
    vector<Triangle > maps(numMaps, Triangle(6));
    int minInterval = numeric_limits<int>::max();
    vector<queue<int> > coveredBy(numLegs); //map from leg to triangle

    for(int i=0;i<numMaps;i++) {
        for(int j=0; j<6; j++) {
            int x,y;
            cin >> x >> y;
            maps[i][j] = Point(x,y);
        }

        //test intersection
        for(int j=0; j<legs.size(); j++) {
            if(contained(legs[j], maps[i])) {
                // cout<<"leg "<<j<<" covered by map "<< i<<std::endl;
                coveredBy[j].push(i);
            }
        }
    }

    //find the shortest interval s.t. all legs are covered
    priority_queue<int> maxMapSTAllCovered;
    priority_queue<pair<int,int>,vector<pair<int,int> >, greater<pair<int,int> > >
        minMapSTAllCovered;

    for(int i=0; i<numLegs; i++) {
        int val = coveredBy[i].front(); coveredBy[i].pop();
        maxMapSTAllCovered.push(val);
        minMapSTAllCovered.push(make_pair(val,i));
    }

    while(true) {
        int minMap, minMapLeg;
        boost::tie(minMap,minMapLeg)
            = minMapSTAllCovered.top(); minMapSTAllCovered.pop();
        int maxMap = maxMapSTAllCovered.top(); //maxMapSTAllCovered.pop();

        int curInterval = maxMap - minMap + 1;
        minInterval = min(curInterval, minInterval);

        if(coveredBy[minMapLeg].empty()) break;

        int newMapInInterval = coveredBy[minMapLeg].front();
        coveredBy[minMapLeg].pop();
        maxMapSTAllCovered.push(newMapInInterval);
        minMapSTAllCovered.push(make_pair(newMapInInterval, minMapLeg));
    }

    cout << minInterval << endl;

}
```

```
150
151 int main() {
152     ios_base::sync_with_stdio(false);
153
154     int numTestCases;
155     cin >> numTestCases;
156
157     for(int t=0; t<numTestCases; t++) {
158         int numLegs, numMaps;
159         cin >> numLegs >> numMaps;
160
161         numLegs-=1;
162         vector<Segment> legs(numLegs);
163         int x,y;
164         cin >> x>>y;
165         Point p1(x,y);
166         for(int i=0;i<numLegs;i++) {
167             cin >> x >> y;
168             Point p2(x,y);
169             legs[i]=Segment(p1,p2);
170             p1 = p2;
171         }
172
173         //100points
174         // myApproach(legs, numLegs, numMaps);
175         //gives only 80 points for me, but interesting sweepline algorithm
176         benApproach(legs, numLegs, numMaps);
177     }
178 }
```

Maximize It!

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

// choose exact integral type
#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 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;
}

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

int main() {

  int problemType;
  while(std::cin >> problemType) {
        if(problemType==0) break;
        int a,b;
        std::cin >> a >> b;

        const int X = 0;
        const int Y = 1;
        Program lp (CGAL::SMALLER, false, 0, false, 0);

        if(problemType==1) {
            //minimize the negative problem
            lp.set_c(Y, -b);
            lp.set_d(X,X,2*a);

            lp.set_l(X,true,0);
            lp.set_l(Y,true,0);

            lp.set_a(X,0,1); lp.set_a(Y,0,1); lp.set_b(0, 4);
            lp.set_a(X,1,4); lp.set_a(Y,1,2); lp.set_b(1, a*b);
            lp.set_a(X,2,-1); lp.set_a(Y,2,1); lp.set_b(2, 1);
        } else {
            const int Z = 2;
            lp.set_d(X,X,2*a);
            lp.set_d(Z,Z,2);
            lp.set_c(Y, b);

            lp.set_u(X,true,0);
            lp.set_l(X,false);
            lp.set_u(Y,true,0);
            lp.set_l(Y,false);
            lp.set_u(Z,false);
            lp.set_l(Z,false);

            lp.set_a(X,0,1); lp.set_a(Y,0,1); lp.set_b(0, -4);
            lp.set_r(0, CGAL::LARGER);
            lp.set_a(X,1,4); lp.set_a(Y,1,2); lp.set_a(Z,1,1);
            lp.set_b(1, -a*b);
```

```
74              lp.set_r(1, CGAL::LARGER);
75              lp.set_a(X,2,-1); lp.set_a(Y,2,1); lp.set_b(2, -1);
76              lp.set_r(2, CGAL::LARGER);
77
78          }
79          Solution s= CGAL::solve_quadratic_program(lp, ET());
80          assert(s.solves_quadratic_program(lp));
81
82          // std::cout << s;
83
84          if(s.is_infeasible())
85              std::cout << "no" << std::endl;
86          else if(s.is_unbounded())
87              std::cout << "unbounded" << std::endl;
88          else {
89              CGAL::Quotient<ET> exactValue = s.objective_value();
90              int value = problemType==1 ?
91                              floor_to_double(-exactValue) :
92                              ceil_to_double(exactValue);
93
94              std::cout << value << std::endl;
95          }
96      }
97  }
```

Collisions

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef Triangulation::Vertex_iterator Vertex_iterator;
typedef K::Point_2 Point;
typedef CGAL::Segment_2<K> Segment;

int main() {
    std::ios_base::sync_with_stdio(false);
    int testCases;
    std::cin >> testCases;
    for(int i=0; i<testCases; i++) {
        int numPlanes;
        K::FT minDistance;
        std::cin >> numPlanes>> minDistance;

    std::vector<Point> planes(numPlanes);

    Triangulation t;

        for(int j=0; j<numPlanes;j++) {
            int x, y;
            std::cin >> x >> y;
            //t.insert(Triangulation::Point(x,y));
            planes[j] = Point(x,y);
        }

    t.insert(planes.begin(), planes.end());

    int numEndangered = 0;
    K::FT minSquaredDistance = minDistance * minDistance;

    for(Vertex_iterator v=t.finite_vertices_begin();
        v!=t.finite_vertices_end(); v++) {
        // std::cout<<v->point() << std::endl;
        Triangulation::Edge_circulator c = t.incident_edges(v);
        K::FT minDist;
        bool firstEdge = true;

        do {
          if(!t.is_infinite(c)) {
            Triangulation::Vertex_handle v1 = c->first->vertex((c->second+1)%3);
            Triangulation::Vertex_handle v2 = c->first->vertex((c->second+2)%3);

            K::FT candidateDist = Segment(v1->point(), v2->point()).squared_length();
            // std::cout << "candidate min dist for " << v->point() << " between " << v1->point() << " " << ↙
                ↳ v2->point() << " is " << candidateDist << std::endl;
            if(firstEdge == true || minDist > candidateDist) {
              minDist = candidateDist;
              firstEdge = false;
            }
        }
         } while(++c != t.incident_edges(v));

         if(!firstEdge && minDist < minSquaredDistance)
        numEndangered++;
    }

    std::cout << numEndangered << std::endl;

    }
}
```

Diet

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

// choose exact integral type
#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 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;
}

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

int main() {

  int numNutrients, numFoods;
  while(std::cin >> numNutrients >> numFoods) {
      if(numNutrients == 0 && numFoods == 0)
          break;
      Program lp (CGAL::LARGER, true, 0, false, 0);
      for(int nutrient=0;nutrient<numNutrients;nutrient++) {
          int l,u;
          std::cin >> l >> u;
          lp.set_b(2*nutrient, l);
          lp.set_b(2*nutrient+1, u);
          lp.set_r(2*nutrient+1, CGAL::SMALLER);
      }

      for(int food=0;food<numFoods;food++) {
          int price;
          std::cin >> price;
          for(int nutrient=0; nutrient<numNutrients; nutrient++) {
              int price,C;
              std::cin >> C;
              lp.set_a(food, 2*nutrient, C);
              lp.set_a(food, 2*nutrient+1, C);
          }
          lp.set_c(food, price);
      }

      Solution s= CGAL::solve_quadratic_program(lp, ET());
      assert(s.solves_quadratic_program(lp));

      if(!s.is_optimal())
          std::cout << "No such diet." << std::endl;
      else
          std::cout << floor_to_double(s.objective_value()) << std::endl;
  }
}
```

Porfolios

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

// choose exact integral type
#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 main() {

    int numAssets, numPeople;
    while(std::cin >> numAssets >> numPeople) {
        if(numAssets == 0 && numPeople == 0)
            break;

        Program qp (CGAL::SMALLER, true, 0, false, 0);
        for(int i=0; i<numAssets; i++) {
            int cost, expectedReturn;
            std::cin >> cost >> expectedReturn;
            qp.set_a(i, 0, expectedReturn);
            qp.set_r(0, CGAL::LARGER);
            qp.set_a(i, 1, cost);
        }
        for(int i=0; i<numAssets; i++) {
            for(int j=0; j<numAssets; j++) {
                int covariance;
                std::cin >> covariance;
                if(j<=i)
                    qp.set_d(i,j,2*covariance);
            }

        }
        for(int i=0; i<numPeople; i++) {
            int maxCost, minReturn, maxVariance;
            std::cin >> maxCost >> minReturn >> maxVariance;
            qp.set_b(0, minReturn);
            qp.set_b(1, maxCost);
            Solution s= CGAL::solve_quadratic_program(qp, ET());
            assert(s.solves_quadratic_program(qp));

            if(!s.is_optimal() || s.objective_value() > maxVariance)
                std::cout << "No." << std::endl;
            else
                std::cout << "Yes." << std::endl;
        }
    }
}
```

Inball

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

// choose exact integral type
#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;

using namespace std;

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

int main() {
    ios_base::sync_with_stdio(false);

    int numInequalities, numDimensions;
    while(true) {
        cin >> numInequalities;
        if(numInequalities==0) break;
        cin >> numDimensions;

        Program lp (CGAL::SMALLER, false, 0, false, 0);
        for(int i=0; i<numDimensions; i++) {
            lp.set_c(i,0);
        }
        lp.set_c(numDimensions,-1);

        for(int i=0; i<numInequalities; i++) {
            long normA=0;
            for(int j=0; j<numDimensions; j++) {
                int a;
                cin >> a;
                normA = normA + a*a;
                lp.set_a(j,i,a);
            }
            lp.set_a(numDimensions, i, sqrt(normA));
            int b;
            cin >> b;
            lp.set_b(i,b);
        }
        lp.set_l(numDimensions, true, 0);
        Solution s= CGAL::solve_quadratic_program(lp, ET());
        assert(s.solves_quadratic_program(lp));

        if(s.is_infeasible())
            std::cout << "none" << std::endl;
        else if(s.is_unbounded())
            std::cout << "inf" << std::endl;
        else {
            CGAL::Quotient<ET> exactValue = s.objective_value();
            cout << floor_to_double(-exactValue) << endl;
        }
    }
}
```

Monkey Island

```cpp
#include <iostream>
#include <vector>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/strong_components.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::edge_descriptor Edge;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_iterator Eit;

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numLocations, numRoads;
        cin >> numLocations >> numRoads;

        Graph g(numLocations);
        vector<int> cost(numLocations);
        for(int i=0;i<numRoads;i++) {
            Edge e;
            int u,v;
            cin >> u >> v;
            tie(e, tuples::ignore) = add_edge(u-1,v-1,g);
        }
        for(int i=0;i<numLocations;i++) {
            int c;
            cin >> c;
            cost[i] = c;
        }

        vector<int> componentMap(numLocations);
        int numComponents = strong_components(g, &componentMap[0]);
        vector<int> componentCosts(numComponents, 100);

        // cout << "numcomp " << numComponents << endl;

        for(int i=0; i<numLocations; i++) {
            if(cost[i] < componentCosts[componentMap[i]]) {
                componentCosts[componentMap[i]] = cost[i];
            }
        }

        Eit ei, ei_end;
        for(tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
            Vertex sV = source(*ei,g);
            Vertex tV = target(*ei,g);

            if(componentMap[sV] != componentMap[tV]) {
                componentCosts[componentMap[tV]] = 0;
            }
        }

        int minPStation = 0;
        for(int i=0;i<numComponents;i++) {
            minPStation += componentCosts[i];
        }
        cout << minPStation << endl;
    }
}
```

Placing Knights

```
#include <iostream>
#include <vector>
#include <set>
#include <cstdlib>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/config.hpp>

using namespace std;
using namespace boost;

typedef pair<int,int> Pos;
typedef adjacency_list<setS, vecS, undirectedS > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;


//returns the knights that the given knight threatens
vector<Pos> threatens(vector<vector<int> > &board, Pos &knight) {
    vector<Pos> ret;
    for(int i=-2;i<=2;i++) {
        for(int j=-2;j<=2;j++) {
            if(!((abs(i)== 1 && abs(j) ==2)
                    || (abs(i)== 2 && abs(j) ==1))) continue; //move not allowed
            unsigned int newX = knight.first+i;
            unsigned int newY = knight.second+j;
            if(0 <= newX && newX < board.size()
                    && 0 <= newY && newY < board.size()
                    && board[newX][newY] != 0) {
                ret.push_back(Pos(newX, newY));
            }
        }
    }
    return ret;
}

void setToZero(vector<vector<int> > &board, vector<Pos> &threatened) {
    for(unsigned int i=0; i<threatened.size(); i++) {
        Pos pos = threatened[i];
        board[pos.first][pos.second] = 0;
    }
}

void printBoard(vector<vector<int> > &board) {
    for(unsigned int i=0; i<board.size(); i++) {
        for(unsigned int j=0; j<board.size(); j++) {
            cout << board[i][j];
        }
        cout << endl;
    }
}


void reduceNumThreatened(set<pair<int, pair<int, int> > > &Q, vector<vector<int> > &board, vector<vector<int> > ↙
    ↳ &boardThreatening, vector<Pos> &threatened) {
   for(unsigned int i=0; i<threatened.size(); i++) {
       Pos pos = threatened[i];
       int numThreatens = boardThreatening[pos.first][pos.second];
       Q.erase(pair<int, Pos>(numThreatens, pos));

       vector<Pos> toReduce = threatens(board, pos);
       for(unsigned int j=0; j<toReduce.size(); j++) {
           Pos toReducePos = toReduce[j];
           numThreatens = boardThreatening[toReducePos.first][toReducePos.second];
           Q.erase(pair<int, Pos>(numThreatens, toReducePos));
           numThreatens--;
           Q.insert(pair<int, Pos>(numThreatens, toReducePos));
           boardThreatening[toReducePos.first][toReducePos.second] = numThreatens;
       }
   }
```

```
}

int numKnights(vector<vector<int> > board) {
    int numKnights = 0;
    int numPosLoc = 0;
    set<pair<int, pair<int, int> > > Q;
    int n = board.size();
    vector<vector<int> > boardThreatening(n,vector<int>(n));

    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            if(board[i][j] != 0) {
                Pos pos =Pos(i,j);
                int numThreatening = threatens(board, pos).size();
                Q.insert(pair<int, pair<int, int> >(numThreatening,pos));
                boardThreatening[pos.first][pos.second] = numThreatening;
                numPosLoc++;
            }
        }
    }
    while(Q.size() > 0 && numPosLoc > 0) {
        // cout<< "board" << std::endl;
        // printBoard(board);
        // cout<<"num threatening" <<std::endl;
        // printBoard(boardThreatening);
        pair<int, pair<int, int> > p = *(Q.begin());
        Q.erase(Q.begin());
        pair<int, int> pos = p.second;
        if(board[pos.first][pos.second] != 0) {
            vector<Pos> threatened = threatens(board, pos);
            setToZero(board, threatened);
            board[pos.first][pos.second] = 0;
            numPosLoc -= (threatened.size() + 1);
            numKnights++;
            threatened.push_back(pos);
            reduceNumThreatened(Q, board, boardThreatening, threatened);
        }
    }
    return numKnights;
}

int posToInt(pair<int,int> pos, int n) {
    return pos.first*n+pos.second;
}

int numKnightsCorrect(vector<vector<int> > board) {
    Graph g;
    int n = board.size();
    int numVertices = 0;

    //build bipartite graph
    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            if(board[i][j] == 0) continue;
            Pos curPos(i,j);
            vector<Pos> threatened = threatens(board, curPos);
            numVertices++;
            for(int k=0; k<threatened.size(); k++) {
                Edge e;
                tie(e, tuples::ignore) = add_edge(
                        posToInt(curPos,n),
                        posToInt(threatened[k],n),g);
            }
        }
    }
    vector<Vertex> mate(num_vertices(g));
    edmonds_maximum_cardinality_matching(g, &mate[0]);
    const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
    int numMatched = 0;
    for(int i=0; i<mate.size(); i++) {
        if(mate[i] != NULL_VERTEX) numMatched++;
    }
    return numVertices - (numMatched/2);
}

int main() {
```

```
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;
    for(int t=0; t < testCases; t++) {
        int n;
        cin >> n;
        vector<vector<int> > board(n,vector<int>(n));

        for(int i=0;i<n;i++) {
            for(int j=0;j<n;j++) {
                int status;
                cin >> status;
                board[i][j] = status;
            }
        }

        cout << numKnightsCorrect(board) << std::endl;

    }
    return 0;
}
```

Shopping Trip

```
#include <boost/config.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <iostream>
#include <limits>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, vertex_index_t>::type IndexMap;

typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph> GraphTraits;

void printGraph(Graph g, EdgeCapacityMap &capacity) {
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        if(capacity[*eiter] > 0) {
            int aSource = source(*eiter, g);
            int aTarget = target(*eiter, g);
            std::cout << aSource <<"␣"
                "␣-" << capacity[*eiter] << "->␣" << aTarget
              << std::endl;
        }
    }
}

void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u,v,g);
    tie(reverseE, tuples::ignore) = add_edge(v, u, g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}


int main() {
    ios_base::sync_with_stdio(false);

    int testCases = 0;
    cin >> testCases;
    for(int testCase=0; testCase < testCases; testCase++) {
        int numVertices, numEdges, numStores;
        cin >> numVertices >> numEdges >> numStores;

        Graph g(numVertices);
        EdgeCapacityMap capacity = get(edge_capacity, g);
        ReverseEdgeMap rev_edge = get(edge_reverse, g);
        ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);

        //add sink
        const int SOURCE = 0;
        //addFlowEdge(g, capacity, rev_edge, isStore, SINK, 1);
        const int SINK = numVertices;

        for(int i=0; i<numStores;i++) {
            int isStore;
            cin >> isStore;
            addFlowEdge(g, capacity, rev_edge, isStore, SINK, 1);
        }

```

```
74          for(int i=0; i<numEdges;i++) {
75              int u,v;
76              cin >> u >> v;
77              addFlowEdge(g, capacity, rev_edge, u, v,1);
78              addFlowEdge(g, capacity, rev_edge, v, u,1);
79          }
80
81          //printGraph(g, capacity) ;
82
83          long flow = push_relabel_max_flow(g, SOURCE, SINK);
84
85          cout << ((flow >= numStores) ? "yes" : "no") << endl;
86      }
87  }
```

TheeV

```cpp
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>

using namespace std;

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;
typedef K::Point_2 P;
typedef K::Segment_2 S;

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;
}

struct CityComparator{
    P mainCity;
    bool operator()(P x, P y) {
        K::FT d1 = S(mainCity, x).squared_length();
        K::FT d2 = S(mainCity, y).squared_length();
        return(d1 > d2);
    }

};

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin>>testCases;

    for(int t=0; t<testCases;t++) {
        int numCities;
        cin>>numCities;
        vector<P> cities;


        for(int i=0; i<numCities;i++) {
            int x,y;
            cin>>x>>y;
            P p(x,y);
            cities.push_back(p);
        }

        P mainCity = cities[0];

        CityComparator comp;
        comp.mainCity = mainCity;

        sort(cities.begin(), cities.end(), comp);
        //sort from highest distance to lowest

        K::FT otherRadius;
        K::FT mainCityRadius;
        K::FT beforeMainCityRadius;
        Min_circle otherRadio;
        for(vector<P>::iterator i = cities.begin();
                i!=cities.end(); i++) {

            // cout << "begin " << *(cities.begin()) << endl;
            // cout << "i " << *i << endl;
            // cout << "end " << *(cities.end()-1) << endl;
            mainCityRadius = S(*(i+1), mainCity).squared_length();

            otherRadio.insert(*i);
            Traits::Circle otherCircle = otherRadio.circle();
            otherRadius = otherCircle.squared_radius();

            // cout << fixed<<setprecision(2)<<"radii " << mainCityRadius << " " << otherRadius << ":" << ↙
```

```
                ↪ otherRadio.number_of_points()<< endl;
74
75              if(otherRadius >= mainCityRadius)
76                  break;
77              beforeMainCityRadius=mainCityRadius;
78          }
79          cout<<fixed<<setprecision(0)<<ceil_to_double(min(beforeMainCityRadius,otherRadius))<<endl;
80
81      }
82
83  }
```

Poker Chips

```cpp
#include <iostream>
#include <numeric>
#include <vector>
#include <map>
#include <bitset>

using namespace std;

int optimalPoints(vector<vector<int> > &chips,
        map<vector<int>,int> &table, vector<int> topChipPosition){
    int numStacks = chips.size();

    map<vector<int>,int>::iterator found = table.find(topChipPosition);
    if(found != table.end()) {
        return found->second;
    }

    //return min over all subsets of possible taking
    int maxPoints = 0;
    int maxSubset = 0;
    //iterate over all subsets of taking coins from the top
    for(int s=1; s < (1<<numStacks); s++) {
        int numChipsTaken = 0;
        //s represents a subset of {0,...,n-1}
        // cout<<"take subset " << std::bitset<5>(s).to_string() << std::endl;
        vector<int> newTopChipPosition(numStacks);
        int color=-1;
        for(int k=0; k<numStacks; k++) {
            //iterate over all elements of s
            newTopChipPosition[k] = topChipPosition[k];
            if((s & (1<<k)) != 0){
                // cout<<"stack " << k << " is in subset" << std::endl;
                if(topChipPosition[k]<0) {
                    // cout<<"stack " << k << " is empty" << std::endl;
                    continue;
                }

                //k is in S
                //ensure that colors are the same
                int stackColor = chips[k][topChipPosition[k]];
                if(color == -1) {
                    color = stackColor;
                    // cout<<"take color " << color << std::endl;
                } else if (color!=stackColor) {
                    // cout<<"stack " << k << "has different color"<< std::endl;
                    continue;
                }
                // cout<<"remove chip from stack " << k << std::endl;
                ++numChipsTaken;
                newTopChipPosition[k] = newTopChipPosition[k]-1;
            }
        }
        int sumChips=0;
        for(int i=0; i< numStacks; i++) {
            sumChips +=newTopChipPosition[i]+1;
        }

        int newPoints = 0;
        if(numChipsTaken > 1)
            newPoints = (1<<(numChipsTaken-2));

        if(sumChips > 0 && numChipsTaken>0)
            newPoints = newPoints + optimalPoints(chips, table,
                        newTopChipPosition);

        // cout<<"points: "<<newPoints<<endl;
        if(newPoints > maxPoints) {
            maxPoints = newPoints;
            maxSubset = s;
        }
    }
    // cout<<"max points for this stack: "<< maxPoints<< " using "
        // << std::bitset<5>(maxSubset).to_string()<< endl;
```

```
74      table[topChipPosition] = maxPoints;
75      return maxPoints;
76  }
77
78  int main() {
79      std::ios_base::sync_with_stdio(false);
80      int testCases;
81      cin >> testCases;
82
83      for(int k=0; k < testCases; k++) {
84          int numStacks;
85          cin >> numStacks;
86
87          vector< vector<int> > chips(numStacks);
88          vector<int> topChipPosition(numStacks);
89          for(int i=0; i<numStacks; i++) {
90              int stackHeight;
91              cin >> stackHeight;
92              chips[i] = vector<int>(stackHeight);
93              topChipPosition[i] = stackHeight-1;
94          }
95
96          for(int i=0; i<numStacks; i++) {
97              for(int j=0; j<chips[i].size(); j++) {
98                  //the chip at the top of the stack is added last
99                  int c;
100                 cin >> c;
101                 chips[i][j] = c;
102             }
103         }
104         map<vector<int>, int> table;
105         cout << optimalPoints(chips,table,topChipPosition) << endl;
106     }
107 }
```

Portfolios Revisited

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

// choose exact integral type
#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 midPoint(int lowerBound, int upperBound) {
    return lowerBound + (upperBound-lowerBound)/2;
}

int main() {
    int numAssets, numPeople;
    while(std::cin >> numAssets >> numPeople) {
        if(numAssets == 0 && numPeople == 0)
            break;

        Program qp (CGAL::SMALLER, true, 0, false, 0);
        std::vector<int> costs(numAssets);
        std::vector<int> returns(numAssets);
        std::vector<double> returnPerCost(numAssets);
        for(int i=0; i<numAssets; i++) {
            int cost, expectedReturn;
            std::cin >> cost >> expectedReturn;
            qp.set_a(i, 0, expectedReturn);
            qp.set_r(0, CGAL::LARGER);
            qp.set_a(i, 1, cost);
            costs[i] = cost;
            returns[i] = expectedReturn;
            returnPerCost[i] = (double)expectedReturn/(double)cost;
        }
        for(int i=0; i<numAssets; i++) {
            for(int j=0; j<numAssets; j++) {
                int covariance;
                std::cin >> covariance;
                if(j<=i)
                    qp.set_d(i,j,2*covariance);
            }

        }
        for(int i=0; i<numPeople; i++) {
            int maxCost, minReturn, maxVariance;
            std::cin >> maxCost >> maxVariance;
            qp.set_b(1, maxCost);

            int indexMinCost = std::max_element(returnPerCost.begin(), returnPerCost.end()) - returnPerCost.begin();
            long returnUpperBound = ceil(returns[indexMinCost]*maxCost/costs[indexMinCost])+1;

            // the following lower bound is not correct, since it might not be possible to buy all high cost assets ↙
                ↳ because this might exceed the risk
            // long returnLowerBound = returns[indexMaxCost]*(double)maxCost/costs[indexMaxCost];

            long returnLowerBound = 0;

            // binary search
            while(returnLowerBound <= returnUpperBound) {
                long maxReturn = midPoint(returnLowerBound,returnUpperBound);

                qp.set_b(0, maxReturn);
                Solution s= CGAL::solve_quadratic_program(qp, ET());
                assert(s.solves_quadratic_program(qp));
```

```
                if(s.is_optimal() && s.objective_value() <= maxVariance) {
                    returnLowerBound = maxReturn+1;
                } else {
                    returnUpperBound = maxReturn-1;
                }
            }
            std::cout << returnUpperBound << std::endl;
        }
    }
}
```

Stamp Exhibition

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

// choose exact integral type
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpzf.h>
typedef CGAL::Gmpzf ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

using namespace std;

// program and solution types
typedef pair<int,int> Pos;
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
typedef K::Point_2 P;
typedef K::Segment_2 S;


int main() {
    ios_base::sync_with_stdio(false);

    int testCases;
    cin>>testCases;
    for(int tcase=0; tcase<testCases; tcase++)  {
        Program lp (CGAL::SMALLER, true, 1, true, 4096);

    int numLights, numStamps, numWalls;
    cin >> numLights >> numStamps >> numWalls;
    vector<P> lights(numLights);
    vector<P> stamps(numStamps);
    vector<S> walls(numWalls);
    vector<long> stampIntensity(numStamps);

    for(int l=0; l<numLights; l++) {
        int x,y;
        cin>>x>>y;
        lights[l]=P(x,y);
        lp.set_c(l,1);
    }

    for(int s=0; s<numStamps; s++) {
        int x,y,intensity;
        cin>>x>>y>>intensity;
        stamps[s] = P(x,y);
        stampIntensity[s] = intensity;

    }

    for(int w=0; w<numWalls; w++) {
        int x1,y1,x2,y2;
        cin>>x1>>y1>>x2>>y2;
        walls[w] = S(P(x1,y1),P(x2,y2));
    }


    for(int s=0; s<numStamps; s++) {
        for(int l=0; l<numLights; l++) {
            S segment =S(stamps[s],lights[l]);
            K::FT r2 = segment.squared_length();
            double quotient2 = 1.0/r2;
            lp.set_a(l,s,quotient2);
            lp.set_a(l,numStamps+s,quotient2);
```

```
74              for(int w=0; w<numWalls; w++) {
75                  if(CGAL::do_intersect(walls[w], segment)) {
76                  // cout<<"stamp"<<s<<" lamp"<<l<<" " <<quotient2;
77                      lp.set_a(l,s,0);
78                      lp.set_a(l,numStamps+s,0);
79                      break;
80                  }
81
82              }
83          }
84          lp.set_b(s, stampIntensity[s]);
85          lp.set_r(numStamps+s, CGAL::LARGER);
86          lp.set_b(numStamps+s, 1);
87      }
88
89      Solution s=CGAL::solve_linear_program(lp,ET());
90      assert(s.solves_quadratic_program(lp));
91      // CGAL::print_linear_program(std::cerr, lp, "lp");
92
93      if(s.is_optimal())
94          cout<< "yes" << endl;
95      else
96          cout << "no" << endl;
97      }
98  }
```

Tetris

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

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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 graph_traits<Graph> GraphTraits;
typedef GraphTraits::vertex_descriptor Vertex;
typedef GraphTraits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u,v,g);
    tie(reverseE, tuples::ignore) = add_edge(v, u, g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

void testCase() {
    int width, numBricks;
    cin >> width >> numBricks;

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


    // does not work for bordercases
    // for(int i=0; i<numBricks; i++) {
    //     int u,v;
    //     cin>>u>>v;
    //     int from = min(u,v);
    //     int to = max(u,v);
    //     addFlowEdge(g, capacity, rev_edge, 2*from+1, 2*to, 1);
    // }
    // for(int i=1; i<width; i++) {
    //     addFlowEdge(g, capacity, rev_edge, 2*i, 2*i+1, 1);
    // }

    for(int i=0; i<numBricks; i++) {
        int u,v;
        cin>>u>>v;
        int from = min(u,v);
        int to = max(u,v);
        if(from == 0) {
            addFlowEdge(g, capacity, rev_edge, from, to, 1);
        } else {
            addFlowEdge(g, capacity, rev_edge, from+width, to, 1);
        }
    }
    for(int i=1; i<width; i++) {
        addFlowEdge(g, capacity, rev_edge, i, i+width, 1);
    }

    long flow = push_relabel_max_flow(g, 0, width);

```

```
74      cout<<flow<<std::endl;
75  }
76
77  int main() {
78      ios_base::sync_with_stdio(false);
79      int testCases = 0;
80      cin >> testCases;
81      while(testCases--) testCase();
82  }
```

Beach Bar

```cpp
#include <iostream>
#include <queue>
#include <set>
#include <algorithm>
#include <limits>

using namespace std;

const int MAX = numeric_limits<int>::max();

pair<int,int> optimalPosition(vector<int> &parasols, set<int> &bestPos) {
    int numParasols = parasols.size();
    sort(parasols.begin(), parasols.end());

    int bestNumParasols = 0;
    int bestMaxDistToWalk = MAX;
    priority_queue<int, vector<int>, greater<int> > Q;
    for(int i=0; i<numParasols; i++) {
        int rightMostParasol = parasols[i];
        Q.push(rightMostParasol);
        while(Q.top() < rightMostParasol-200) {
            Q.pop();
        }
        int leftMostParasol = Q.top();
        int numCovered = Q.size();
        int span = rightMostParasol-leftMostParasol;
        int barPos = (span/2)+ leftMostParasol;
        int barPos2 = barPos;
        if(span%2!=0) barPos2 = barPos+1;
        int maxDistToWalk = max(rightMostParasol-barPos,
                                -(leftMostParasol-barPos));

        if(numCovered > bestNumParasols) {
           bestNumParasols = numCovered;
           bestMaxDistToWalk = maxDistToWalk;
           bestPos.clear();
           bestPos.insert(barPos);
           bestPos.insert(barPos2);
        } else if(numCovered == bestNumParasols) {
            if(maxDistToWalk < bestMaxDistToWalk) {
                bestPos.clear();
                bestPos.insert(barPos);
                bestPos.insert(barPos2);
                bestMaxDistToWalk = maxDistToWalk;
            } else if(maxDistToWalk == bestMaxDistToWalk) {
                bestPos.insert(barPos);
                bestPos.insert(barPos2);
            }
        }
    }

    return pair<int,int>(bestNumParasols, bestMaxDistToWalk);
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;
    for(int t=0;t<testCases;t++) {
        int numParasol;
        cin >> numParasol;
        vector<int> parasols(numParasol);
        for(int i=0;i<numParasol;i++) {
            int pos;
            cin >> pos;
            parasols[i] = pos;
        }
        set<int> optimalPos;
        pair<int, int> optimalValues = optimalPosition(parasols, optimalPos);
        cout<<optimalValues.first<<"␣"<<optimalValues.second<<endl;
        for(set<int>::iterator it = optimalPos.begin();
                it!=optimalPos.end(); it++) {
            if(it!=optimalPos.begin()) {
```

```
74                  cout<<" ";
75              }
76              cout<<*it;
77          }
78          cout<<endl;
79      }
80  }
```

Cover

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

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef CGAL::Segment_2<K> Segment;
typedef CGAL::Point_2<K> Point;
typedef CGAL::Ray_2<K> Ray;
typedef Triangulation::Finite_faces_iterator FFiter;
typedef CGAL::Iso_rectangle_2<K> Rect;
typedef CGAL::Line_2<K> Line;
// typedef CGAL::Triangulation_data_structure_2 Tds;
typedef Triangulation::Face_handle Face_handle;
typedef Triangulation::Vertex_handle Vertex_handle;

const int MAX = std::numeric_limits<int>::max();


double ceil_to_double(double x) {
    double a = std::ceil(x);
    while(a<x) a+=1;
    while(a-1>=x) a-=1;
    return a;
}

K::FT sdistance(Point x, Point y) {
    return Segment(x,y).squared_length();
}

std::vector<Segment> rectToSegment(Rect r) {
    std::vector<Segment> v(4);
    v[0] = Segment(r.vertex(0), r.vertex(1));
    v[1] = Segment(r.vertex(1), r.vertex(2));
    v[2] = Segment(r.vertex(2), r.vertex(3));
    v[3] = Segment(r.vertex(3), r.vertex(4));
    return v;
}

template<typename T>
K::FT check_intersection_segments(T &voronoiEdge, Rect &area, Triangulation &t) {
        std::vector<Segment> areaSegments = rectToSegment(area);
        Segment intersectedAreaEdge;
        bool isIntersected =false;
        for(int i=0;i<4;i++) {
            if(CGAL::do_intersect(voronoiEdge,areaSegments[i])) {
                intersectedAreaEdge = areaSegments[i];
                isIntersected =true;
            }
        }
        if(!isIntersected) return -1;

        CGAL::Object o2 = CGAL::intersection(voronoiEdge, intersectedAreaEdge);
        if(const Point* op = CGAL::object_cast<Point>(&o2)) {
            Vertex_handle vh = t.nearest_vertex(*op);
            return sdistance(*op, vh->point());

        } else {
            std::cerr<<"runtime error";
            return -1;
            throw std::runtime_error("strange segment intersection");
        }
}


void testcase(int numAntenna) {
    double x1,x2,y1,y2;
    std::cin >> x1 >> y1 >> x2 >> y2;
    Rect area = Rect(Point(x1,y1),Point(x2,y2));
```

```
74      std::vector<Point > antennas(numAntenna);
75      Triangulation t;
76      for(int i=0; i<numAntenna; i++) {
77          double x,y;
78          std::cin>>x>>y;
79          antennas[i] = Point(x,y);
80      }
81
82      t.insert(antennas.begin(), antennas.end());
83      K::FT maxDist = -1;
84
85      //case 1: go over finite faces get circumcenter
86      for(FFiter i=t.finite_faces_begin();
87              i!=t.finite_faces_end();
88              i++) {
89          // Point c = t.circumcenter(i);
90          Point c = t.dual(i);
91          if(c.x() >= x1 && c.x() <= x2 && c.y() >= y1 && c.y() <= y2) {
92              K::FT tentativeMax = sdistance(c, i->vertex(1)->point());
93              if(tentativeMax > maxDist)
94                  maxDist = tentativeMax;
95          }
96      }
97      // std::cout<<"maxDist " <<ceil_to_double_sqrt(maxDist) << std::endl;
98
99      //case 2: go over corners of area, get nearest vertex
100     //for every corner
101     for(int i=0; i<4; i++) {
102         Point corner = area.vertex(i);
103         //for every nearestVertex
104         Vertex_handle vh = t.nearest_vertex(corner);
105         K::FT tentMax = sdistance(vh->point(), corner);
106
107         if(tentMax > maxDist) {
108             maxDist = tentMax;
109         }
110     }
111     // std::cout<<"maxDist " <<ceil_to_double_sqrt(maxDist) << std::endl;
112
113     //case 3: go over infinite faces
114     // process all Voronoi edges
115     for(Edge_iterator e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e) {
116         CGAL::Object o = t.dual(e);
117         // o can be a segment, a ray or a line ...
118         Point* areaIntersection;
119         if(const Ray* oray = CGAL::object_cast<Ray>(&o)) {
120             maxDist = max(maxDist,check_intersection_segments(*oray, area, t));
121         } else if(const Line* oray = CGAL::object_cast<Line>(&o)) {
122             maxDist = max(maxDist,check_intersection_segments(*oray, area, t));
123         } else if(const Segment* oray = CGAL::object_cast<Segment>(&o)) {
124             maxDist = max(maxDist,check_intersection_segments(*oray, area, t));
125         }
126     }
127     std::cout<<std::setiosflags(std::ios::fixed) << std::setprecision(0)<< ↙
          ↳ ceil_to_double(sqrt(CGAL::to_double(maxDist)))
128             <<std::endl;
129     // std::cout<< numAntenna<<std::endl;
130 }
131 int main() {
132     std::ios_base::sync_with_stdio(false);
133     while(true) {
134         int numAntenna;
135         std::cin>>numAntenna;
136         if(numAntenna == 0) return 0;
137         testcase(numAntenna);
138     }
139 }
```

Divisor Distance

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

using namespace std;

int greatestDivisor(int number) {
    int i;
    for (i = 2; i <=sqrt(number); i++) {
        if (number % i == 0) {
            return number/i;
        }
    }
    return 1;
}

int cacheDivisor(vector<int> &graph, int i) {
    if(graph[i] == -1) {
        int gcd = greatestDivisor(i);
        graph[i] = gcd;
        return gcd;
    } else {
        return graph[i];
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;

    const int maxN = 10000000;
    vector<int> g(maxN,-1);
    for(int testCase=0; testCase < testCases; testCase++) {
        int n, numPairs;

        cin >> n;

        cin >> numPairs;
        for(int i=0; i<numPairs; i++) {
            int x,y;
            cin >> x >> y;
            int pathLen=0;
            while(true) {
                if(x==y) break;
                if(x>y) {
                    x=cacheDivisor(g, x);
                } else {
                    y=cacheDivisor(g, y);
                }
                ++pathLen;
            }
            cout<<pathLen<<endl;
        }
    }
}
```

Tiles

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


using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;


void testCase() {
    int width,height;
    cin >> width >> height;

    vector<vector<int> > field =
        vector<vector<int> >(width, vector<int>(height, -1));
    int curNumTilable = 0;
    Graph g;

    for(int i=0; i<height; i++) {
        for(int j=0; j<width; j++) {
            char place;
            cin>>place;
            if(place == '.') {
                field[j][i] = curNumTilable;
                if(i>0 && field[j][i-1] != -1) {
                    Edge e;
                    tie(e, tuples::ignore)=add_edge(field[j][i-1],curNumTilable,g);
                }
                if(j>0 && field[j-1][i] != -1) {
                    Edge e;
                    tie(e, tuples::ignore)=add_edge(field[j-1][i],curNumTilable,g);
                }
                curNumTilable++;
            }
        }
    }

    vector<Vertex> mate(curNumTilable);
    edmonds_maximum_cardinality_matching(g, &mate[0]);
    // for(i     get(mate,v)
    // graph_traits::null_vertex();
    if(matching_size(g,&mate[0])*2 == curNumTilable) {
        cout<<"yes"<<std::endl;
    } else {
        cout<<"no"<<std::endl;
    }

}

int main() {
    int testCases;
    cin>>testCases;
    while(testCases--) testCase();
    return 0;
}
```

Deleted Entries Stike Back

```
#include <iostream>
#include <vector>
#include <queue>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/biconnected_components.hpp>
#include <boost/graph/connected_components.hpp>

using namespace std;
using namespace boost;

// #define DEBUG

#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

namespace boost {
    struct edge_component_t {
        enum {
            num=555
        };
        typedef edge_property_tag kind;
    }
    edge_component;
}

typedef adjacency_list<vecS,vecS,undirectedS,no_property,
            property<edge_component_t,std::size_t> > Graph;
typedef graph_traits<Graph> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_component_t>::type ComponentMap;

vector<int> discoverTimeToVertex;

// Custom visitor used to record DFS order.
struct CustomVisitor : public default_dfs_visitor {
    void discover_vertex(int u, const Graph& G) {
        discoverTimeToVertex.push_back(u);
    }
};

int newColor(int col1, int col2) {
    int newCol = (col1+1)%3;
    if(newCol == col2) {
        newCol = (newCol+1)%3;
    }
    return newCol;
}

void testCase() {
    int numVertices, numEdges;
    cin>>numVertices>>numEdges;

    Graph g(numVertices);
    ComponentMap component = get(edge_component,g);

    for(int i=0; i<numEdges; i++) {
        Edge e;
        int u,v;
        cin>>u>>v;
        tie(e,tuples::ignore) = add_edge(u,v,g);
    }

    //is connected?
    vector<int> con_component(numVertices);
    size_t num_comps = connected_components(g, &con_component[0]);
    if(num_comps!=1) {
```

```
74          cout<<"no"<<endl;
75          return;
76      }
77
78      //is biconnected?
79      vector<Vertex> low(numVertices);
80      vector<Vertex> vertexToDiscoverTime(numVertices);
81      vector<Vertex> predecessor(numVertices);
82      discoverTimeToVertex.clear();
83
84      size_t num_bicomps
85          = biconnected_components(g,component,
86                                   lowpoint_map(&low[0])
87                                   .discover_time_map(&vertexToDiscoverTime[0])
88                                   .predecessor_map(&predecessor[0])
89                                   .visitor(CustomVisitor()));
90      if(num_bicomps!=1) {
91          cout<<"no"<<endl;
92          return;
93      }
94
95      cout<<"yes"<<endl;
96      vector<int> color(numVertices, 0);
97      vector<bool> visited(numVertices, false);
98
99      D(for(int i=0; i<numVertices; i++) {
100         cout<<i << "␣" << vertexToDiscoverTime[i]<< "␣" << predecessor[i] << "␣" << low[i] << endl;
101     })
102
103     for(int i=1; i< numVertices; i++) {
104         int v = discoverTimeToVertex[i];
105         color[v] = newColor(color[predecessor[v]], color[discoverTimeToVertex[low[v]-1]]);
106     }
107
108     vector<vector<int> > colors(3);
109     for(int i=0; i<numVertices; i++) {
110         colors[color[i]].push_back(i);
111     }
112
113     for(int i=0; i<3; i++) {
114         cout<<colors[i].size();
115         for(int j=0; j<colors[i].size();j++) {
116             cout<< "␣";
117             cout<<colors[i][j];
118         }
119         cout<<endl;
120     }
121
122 }
123
124 int main() {
125     ios_base::sync_with_stdio(false);
126     int testCases;
127     cin>>testCases;
128     while(testCases--) testCase();
129 }
```

Light The Stage

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

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Face_handle Face;
typedef Triangulation::Vertex_handle Vertex;
typedef K::Point_2 P;
typedef K::Segment_2 S;
typedef K::Circle_2 C;

using namespace std;

int midpoint(int lowerBound, int upperBound) {
    return lowerBound + (upperBound-lowerBound)/2;
}

void winnersGivenNumLamps(const vector<P> &lamps, const int numLamps, const int height,
                          const vector<int> &peopleR, const vector<P> &people,
                          vector<int> &winners) {
    // construct triangulation
    Triangulation t;
    t.insert(lamps.begin(), lamps.begin()+numLamps+1);

    for(int i=0; i<people.size(); i++) {
        Vertex nearestLamp = t.nearest_vertex(people[i]);

        K::FT maxDist = height;
        maxDist+=peopleR[i];
        maxDist = maxDist * maxDist;
        K::FT dist = CGAL::squared_distance(nearestLamp->point(),people[i]);

        if(dist >= maxDist) winners.push_back(i);
    }

}

void testcase() {
    int numPeople, numLamps;
    cin >> numPeople >> numLamps;

    vector<P> people(numPeople);
    vector<int> peopleR(numPeople);
    for(int i=0; i<numPeople; i++) {
        int x,y,r;
        cin>>x>>y>>r;
        people[i] = P(x,y);
        peopleR[i] = r;
    }

    int height;
    cin >> height;
    vector<P> lamps(numLamps);
    for(int i=0; i<numLamps; i++) {
        int x,y;
        cin>>x>>y;
        lamps[i] = P(x,y);
    }

    vector<int> winners;

    // try all lamps
    winnersGivenNumLamps(lamps, numLamps, height,
                          peopleR, people, winners);
    if(winners.size() != 0) {
        for(int i=0; i< winners.size();i++) {
            cout<<winners[i]<<"␣";
        }
        cout<<endl;
        return;
    }
```

```
74
75
76      //try binary search
77      int maxLampsMin = 0;
78      int maxLampsMax = lamps.size()-1;
79
80      while (maxLampsMax >= maxLampsMin) {
81          winners.clear();
82
83          int maxLamps = midpoint(maxLampsMin, maxLampsMax);
84
85          winnersGivenNumLamps(lamps, maxLamps, height,
86                               peopleR, people, winners);
87
88          if(winners.size() == 0) {
89              maxLampsMax = maxLamps-1;
90          } else {
91              maxLampsMin = maxLamps+1;
92          }
93      }
94
95      winners.clear();
96      winnersGivenNumLamps(lamps, maxLampsMax, height,
97                               peopleR, people, winners);
98      for(int i=0; i< winners.size();i++) {
99          cout<<winners[i]<<"␣";
100     }
101     cout<<endl;
102
103 }
104
105 int main() {
106     ios_base::sync_with_stdio(false);
107     int testcases;
108     cin >> testcases;
109     while(testcases--) testcase();
110
111 }
```

Radiation

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

// choose exact integral type
#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<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

using namespace std;


vector<vector<double> > powArray(2048,vector<double>(31,-1));

struct Point3 {
    int x,y,z;
};

Point3 P(int x, int y, int z) {
    Point3 p;
    p.x = x;
    p.y = y;
    p.z = z;
    return p;
}

double getPowerArray(int x, int y) {
    double tmp;
    if(powArray[x+1024][y] == -1) {
        tmp = pow(x,y);
        powArray[x+1024][y] = tmp;
        return tmp;
    } else {
        return powArray[x+1024][y];
    }
}

int midpoint(int lowerBound, int upperBound) {
    return lowerBound + (upperBound-lowerBound)/2;
}

bool testDegree(vector<Point3> &cells, int numHealthy, int degree) {
    Program lp (CGAL::SMALLER, false, 0, false, 0);
    for(int c=0; c<cells.size(); c++) {
        int row = c;
        int rowIndex = 0;
        for(int i=0; i <= degree; i++) {
            for(int j=0; j <= degree-i; j++) {
                for(int k=0; k <= degree-i-j; k++) {
                    lp.set_a(rowIndex, row,
                            getPowerArray(cells[c].x, i)
                            * getPowerArray(cells[c].y, j)
                            * getPowerArray(cells[c].z, k)
                            );
                    if(row<numHealthy) {
                        //healthy cell
                        lp.set_b(row,-1);
                        lp.set_r(row, CGAL::SMALLER);
                    } else {
                        //tumor cell
                        lp.set_b(row,1);
                        lp.set_r(row, CGAL::LARGER);
                    }
```

```
                    ++rowIndex;
                }
            }
        }
    }
    CGAL::Quadratic_program_options options;
    options.set_pricing_strategy(CGAL::QP_BLAND);
    Solution s = CGAL::solve_linear_program(lp, ET(), options);
    assert(s.solves_linear_program(lp));

    return !s.is_infeasible();
}

int findBestDegree(vector<Point3> &cells, int numHealthy) {
    const int maxLinear = -1;
    for(int degree = 0; degree <= maxLinear; degree++) {
        if(testDegree(cells, numHealthy, degree)) {
            return degree;
        }
    }

    int degreeMin = maxLinear+1;
    int degreeMax = 30;
    while(degreeMax >= degreeMin) {
        int degree = midpoint(degreeMin, degreeMax);

        if(testDegree(cells, numHealthy, degree)) {
            degreeMax = degree-1;
        } else {
            degreeMin = degree+1;
        }
    }

    return degreeMin;
}

void testCase() {
    int numHealthy, numTumor;
    cin>>numHealthy>>numTumor;

    vector<Point3> cells(numHealthy+numTumor);

    for(int i=0; i<cells.size(); i++) {
        int x,y,z;
        cin>>x>>y>>z;
        cells[i] = P(x,y,z);
    }

    int degree = findBestDegree(cells, numHealthy);
    if(degree <= 30) {
        cout<< degree << std::endl;
    } else {
        cout << "Impossible!" << std::endl;
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;
    while(testCases--) {
        testCase();
    }
}
```

Sweepers

```
#include <iostream>
#include <vector>
#include <queue>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<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 graph_traits<Graph> GraphTraits;
typedef GraphTraits::vertex_descriptor Vertex;
typedef GraphTraits::edge_descriptor Edge;
typedef GraphTraits::out_edge_iterator edge_iterator;
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;

void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u,v,g);
    tie(reverseE, tuples::ignore) = add_edge(v, u, g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

void testCase() {
    int numVertices, numEdges, numSweepers;
    cin>>numVertices>>numEdges>>numSweepers;

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

    const int SOURCE = numVertices;
    const int SINK = numVertices+1;
    vector<int> startLocations, exitLocations;

    //read start and exit locations
    for(int i=0; i<numSweepers; i++) {
        int startLocation;
        cin>>startLocation;
        startLocations.push_back(startLocation);
    }
    for(int i=0; i<numSweepers; i++) {
        int exitLocation;
        cin>>exitLocation;
        exitLocations.push_back(exitLocation);
    }


    // the following is a incorrect interpretation of the task:
    // correct: if there are sweepers they should clean every corridor just once
    // there seems to be no requirement that corridors have to be cleaned
    // if(numEdges > 0 && numSweepers == 0) {
    //     cout<< "no"<< std::endl;
    //     return;
    // }

    //build flow graph
    for(int i=0; i<numEdges; i++) {
        int u,v;
```

```
        cin>>u>>v;
        addFlowEdge(g, capacity, rev_edge, u, v, 1);
        addFlowEdge(g, capacity, rev_edge, v, u, 1);
    }

    // check if all vertices of non-zero degree are reachable from some source
    vector<bool> visited(numVertices,false);
    std::queue<Vertex> Q;
    for(int i=0; i<startLocations.size(); i++) {
       int v = startLocations[i];
       visited[v] = true;
       Q.push(v);
       while(Q.size() > 0) {
           v = Q.front(); Q.pop();
           edge_iterator out_i, out_end;
           for(tie(out_i, out_end) = out_edges(v,g);
                   out_i!=out_end; ++out_i) {
               Edge e = *out_i;
               int targ = target(e,g);
               if(targ<visited.size() && !visited[targ]) {
                   Q.push(targ);
                   visited[targ] = true;
               }
           }
       }
    }

    for(int i=0; i<visited.size(); i++) {
        if(!visited[i] && out_degree(i,g)>0) {
            cout<< "no"<< std::endl;
            return;
        }
    }

    //check eulerian tour
    for(int i=0; i<numVertices; i++) {
        int v = vertex(i, g);
        int numStartLocation = count(startLocations.begin(), startLocations.end(), v);
        int numExitLocation = count(exitLocations.begin(), exitLocations.end(), v);

        if((out_degree(v,g)/2+numStartLocation+numExitLocation)%2 == 1) {
            cout<<"no";
            cout<<std::endl;
            return;
        }
    }

    // add source and sink
    for(int i=0; i<startLocations.size(); i++) {
        addFlowEdge(g, capacity, rev_edge, SOURCE, startLocations[i], 1);
    }
    for(int i=0; i<exitLocations.size(); i++) {
        addFlowEdge(g, capacity, rev_edge, exitLocations[i], SINK, 1);
    }

    //compute flow
    long flow = push_relabel_max_flow(g, SOURCE, SINK);

    if(flow == numSweepers)
        cout<<"yes"<<std::endl;
    else
        cout<<"no"<<std::endl;

    return;

}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases = 0;
    cin >> testCases;
    while(testCases--) testCase();
}
```

The Bracelet

```
#include <iostream>
#include <vector>
#include <stack>
#include <map>
#include <set>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/connected_components.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> Traits;
typedef Traits::vertex_descriptor Vertex;
typedef Traits::edge_descriptor Edge;
typedef Traits::vertex_iterator vertex_iter;
typedef Traits::out_edge_iterator edge_iterator;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

void eulerTourDfs(Graph &g, WeightMap &marked, Vertex v, vector<int> &tour) {

    edge_iterator out_i, out_end;
    for (tie(out_i, out_end) = out_edges(v, g);
            out_i!=out_end; ++out_i) {
        if(marked[*out_i]==0) {
            marked[*out_i]=1;
            eulerTourDfs(g,marked,target(*out_i, g),tour);
        }
    }
    tour.push_back(v);
}
void eulerTour(Graph &g, WeightMap &marked, Vertex v, vector<int> &tour) {
    stack<int> S;
    S.push(v);
    while(!S.empty()) {
        Vertex v = S.top();
        edge_iterator out_i, out_end;
        bool hasUnmarkedEdge=false;
        for (tie(out_i, out_end) = out_edges(v, g);
                out_i!=out_end; ++out_i) {
            if(marked[*out_i]==0) {
                hasUnmarkedEdge=true;
                marked[*out_i]=1;
                S.push(target(*out_i,g));
                break;
            }
        }

        if(!hasUnmarkedEdge) {
            S.pop();
            tour.push_back(v);
        }
    }
}


void testCase(int num) {
    cout<<"Case #"<<num<<std::endl;
    int numBeads;
    cin>>numBeads;

    Graph g;
    WeightMap marked = get(edge_weight, g);
    vector<int> colorMap(51,-1);
    int colorIndex = 0;

    for(int i=0;i<numBeads;i++) {
        int color1, color2;
        cin>>color1>>color2;

```

```
        // map colors to gapless indexes starting with 0
        if(colorMap[color1] == -1) {
            colorMap[color1] = colorIndex;
            ++colorIndex;
        }
        if(colorMap[color2] == -1) {
            colorMap[color2] = colorIndex;
            ++colorIndex;
        }

        Edge e;
        tie(e, tuples::ignore) =add_edge(colorMap[color1],colorMap[color2],g);
        marked[e]=0;
    }

    vector<int> componentMap(num_vertices(g));
    int components = connected_components(g, &componentMap[0]);
    if(components > 1) {
        cout<<"some beads may be lost"<<std::endl;
        // cout<<"reason 1: "<<components<<std::endl;
        return;
    }

    std::pair<vertex_iter,vertex_iter> vi;
    for(vi=vertices(g); vi.first != vi.second; ++vi.first) {
        int degree = out_degree(*(vi.first),g);
        if(degree%2==1) {
            cout<<"some beads may be lost"<<std::endl;
            // cout<<"reason 2"<<std::endl;
            return;
        }
    }

    //print eulerian tour
    vector<int> tour;
    // eulerTourDfs(g,marked,0,tour);
    eulerTour(g,marked,0,tour);

    // reverse colorMap
    vector<int> inverseColorMap(51,-1);
    for(int i=0; i<colorMap.size(); i++) {
        if(colorMap[i] != -1) {
            inverseColorMap[colorMap[i]] = i;
        }
    }

    for(int i=0; i<tour.size()-1; i++) {
        cout<<inverseColorMap[tour[i]]<<" "<<inverseColorMap[tour[i+1]]<<std::endl;
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    int testCases;
    cin >> testCases;
    int i = 0;
    while(testCases--) {
        testCase(++i);
        if(testCases>0) cout<<std::endl;
    }
}
```

Knights

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

using namespace std;
using namespace boost;

// #define DEBUG

#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

typedef adjacency_list_traits<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 graph_traits<Graph> GraphTraits;
typedef GraphTraits::vertex_descriptor Vertex;
typedef GraphTraits::edge_descriptor Edge;
typedef property_map<Graph,vertex_index_t>::type IndexMap;
typedef property_map<Graph,edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph,edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph,edge_reverse_t>::type ReverseEdgeMap;

void addFlowEdge(Graph &g, EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, int u, int v, int c) {
    Edge e, reverseE;
    tie(e,tuples::ignore) = add_edge(u,v,g);
    tie(reverseE,tuples::ignore) = add_edge(v,u,g);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
}

void testCase() {
    // graph is given as a grid
    int gridX, gridY, numSources;
    cin>>gridX>>gridY>>numSources;

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

    //need two "layers" one for in-vertices, the other for out-vertices because
    //the "vertex"-capacity is 1
    //the in-vertex is on the first layer, meaning its index <= numVerticesOnOneLayer
    int numVerticesOnLayer = gridX*gridY;

    const int SOURCE = numVerticesOnLayer*2;
    const int SINK = numVerticesOnLayer*2+1;

    for(int i=0; i<gridX; i++) {
        for(int j=0; j<gridY; j++) {
            int curVertex = numVerticesOnLayer + i*gridY+j;
            // upper row
            if(j>0) {
                addFlowEdge(g, capacity, rev_edge, curVertex, i*gridY+j-1,1);
            } else {
                addFlowEdge(g, capacity, rev_edge, curVertex, SINK,1);
            }
            // right column
            if(i<gridX-1) {
                addFlowEdge(g, capacity, rev_edge, curVertex, (i+1)*gridY+j,1);
            } else {
```

```
                addFlowEdge(g, capacity, rev_edge, curVertex, SINK,1);
            }
            // lower row
            if(j<gridY-1) {
                addFlowEdge(g, capacity, rev_edge, curVertex, i*gridY+j+1,1);
            } else {
                addFlowEdge(g, capacity, rev_edge, curVertex, SINK,1);
            }
            // left column
            if(i>0) {
                addFlowEdge(g, capacity, rev_edge, curVertex, (i-1)*gridY+j,1);
            } else {
                addFlowEdge(g, capacity, rev_edge, curVertex, SINK,1);
            }

            //connect out-vertex to its in-counterpart
            addFlowEdge(g, capacity, rev_edge, curVertex-numVerticesOnLayer, curVertex,1);
        }
    }

    for(int i=0; i<numSources; i++) {
        int x,y;
        cin>>x>>y;
        addFlowEdge(g, capacity, rev_edge, SOURCE, x*gridY+y,1);
    }

    D(cout<<"num␣vertices␣"<<num_vertices(g)<<endl;)

    long flow = push_relabel_max_flow(g, SOURCE, SINK);

    cout<<flow<<endl;

}

int main() {
    int numCases;
    cin >> numCases;
    while(numCases--) testCase();
    return 0;
}
```

Next Path

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// #define DEBUG
#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

typedef vector<vector<int> > AdjacencyList;

const int MAX = numeric_limits<int>::max();

int shortestPath(const AdjacencyList &adjacencyList, int start, int end, vector<int> &predecessorMap) {
    queue<int> Q;
    vector<bool> visited(adjacencyList.size(),false);
    vector<int> distance(adjacencyList.size());
    Q.push(start);
    visited[start] = true;
    distance[start] = 0;
    if(start == end) {
        return 0;
    }

    while(!Q.empty()) {
        int v = Q.front();
        Q.pop();
        for(int i =0; i< adjacencyList[v].size(); i++) {
            int childV = adjacencyList[v][i];
            if(!visited[childV]) {
                visited[childV] = true;
                predecessorMap[childV] = v;
                distance[childV] = distance[v] + 1;
                if(childV == end) {
                    return distance[childV];
                } else {
                    Q.push(childV);
                }
            }
        }
    }
    return -1;
}

void testCase() {
    int numEdges, numVertices;
    cin>>numVertices>>numEdges;
    int start, end;
    cin>>start>>end;
    start-=1; end-=1;

    AdjacencyList adjacencyList(numVertices, vector<int>(0));
    for(int i=0; i<numEdges; i++) {
        int u,v;
        cin>>u>>v;
        adjacencyList[u-1].push_back(v-1);
    }

    vector<int> predecessorMap(numVertices, -1);
    int secondShortestPathLength = MAX;
    int shortestPathLength = shortestPath(adjacencyList, start, end, predecessorMap);
    D(cout<<"shortest:␣"<<shortestPathLength<<endl;)

    if(shortestPathLength == -1) {
        cout<<"no"<<endl;
        D(cout<<"reason1"<<endl;)
        return;
    }
```

```
74
75      vector<int> path;
76      int curVertex = end;
77      D(cout<<"backwards␣path:␣";)
78      while(curVertex != -1) {
79          D(cout<<curVertex <<"␣";)
80          path.push_back(curVertex);
81          curVertex = predecessorMap[curVertex];
82      }
83      D(cout<<endl;)
84
85      for(int i = 0; i<path.size(); i++) {
86          int v = path[i];
87          int pathLenSoFar = path.size() - i - 1;
88          for(int j =0; j< adjacencyList[v].size(); j++) {
89              int childV = adjacencyList[v][j];
90              if(i==0 || childV != path[i-1]) {
91                  int shortestPathFromChild = shortestPath(adjacencyList, childV , end, predecessorMap);
92                  D(cout<<"vertex␣"<<v<< "␣path␣len␣so␣far␣on␣shortest:␣" << pathLenSoFar <<";␣shortest␣path␣from␣" << ↙
                        ↪ childV << "␣is␣" << shortestPathFromChild << endl;)
93                  if(shortestPathFromChild >= 0) {
94                      secondShortestPathLength
95                          = min(secondShortestPathLength, pathLenSoFar + 1 + shortestPathFromChild);
96                  }
97              }
98          }
99      }
100
101     if(secondShortestPathLength==MAX) {
102         // secondShortestPathLength = shortestPathLength;
103         cout<<"no"<<endl;
104         D(cout<<"reason2"<<endl;)
105         return;
106     }
107
108     cout<<secondShortestPathLength<<endl;
109
110 }
111
112 int main() {
113     int testCases;
114     cin>>testCases;
115     while(testCases--) testCase();
116     return 0;
117 }
```

Odd Route

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace std;
using namespace boost;

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

void testCase() {
    int numVertices, numEdges;
    cin>>numVertices>>numEdges;
    Graph g(numVertices*4);

    int source,target;
    cin>>source>>target;

    target = numVertices*3 + target;

    property_map<Graph, edge_weight_t>::type weightMap = get(edge_weight, g);
    for(int i = 0; i < numEdges; ++i)
    {
        int u,v,w;
        cin>>u>>v>>w;
        bool success;

        Edge e;
        if(w%2==1) {
            tie(e, success) = add_edge(u, v+3*numVertices, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+numVertices, v+2*numVertices, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+2*numVertices, v+numVertices, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+3*numVertices, v, g);
            weightMap[e] = w;

        } else {
            tie(e, success) = add_edge(u, v+2*numVertices, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+numVertices, v+3*numVertices, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+2*numVertices, v, g);
            weightMap[e] = w;
            tie(e, success) = add_edge(u+3*numVertices, v+numVertices, g);
            weightMap[e] = w;


        }
    }

    std::vector<Vertex> predecessors(num_vertices(g));
    std::vector<int> distancesFromSource(num_vertices(g));

    dijkstra_shortest_paths(g, source,
        predecessor_map(&predecessors[0]).distance_map(&distancesFromSource[0]));

    if(distancesFromSource[target] < numeric_limits<int>::max()) {
        cout<<distancesFromSource[target]<<endl;
    } else {
        cout<<"no"<<endl;
    }

}


```

```
74 int main()
75 {
76     int testCases;
77     cin>>testCases;
78 
79     while(testCases--) testCase();
80 }
```

Radiation 2

not correct

```
#include <iostream>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>


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

// #define DEBUG

#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

using namespace std;

// program and solution types
typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
typedef pair<double,double> Point2;

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Edge_iterator  Edge_iterator;
typedef Triangulation::Face_handle  Face_handle;
typedef CGAL::Segment_2<K> Segment;
typedef CGAL::Point_2<K> P;
typedef Triangulation::Finite_faces_iterator Fiter;


//does not work
void lpApproach(vector<Point2> &cells, int numHealthy) {
    Program program (CGAL::SMALLER, false, 0, false, 0);
    const int numW = 5;
    for(int row=0; row<cells.size(); row++) {
        program.set_a(0, row, 1);
        program.set_a(1, row, cells[row].first);
        program.set_a(2, row, cells[row].second);
        program.set_a(3, row, cells[row].first * cells[row].first);
        program.set_a(4, row, cells[row].second * cells[row].second);

        //slack variables
        for(int i=0; i<cells.size(); i++) {
            if(i==row && row>=numHealthy) {
                program.set_a(numW+i, row, 1);
            } else {
                program.set_a(numW+i, row, 0);
            }
        }
        if(row<numHealthy) {
            //healthy cell
            program.set_b(row,-1);
            program.set_r(row, CGAL::SMALLER);
        } else {
            //tumor cell
            program.set_b(row,1);
            program.set_r(row, CGAL::LARGER);
        }

    }
```

```
    //slack variables
    for(int i=0; i<numW; i++) {
        int row = cells.size()+i;
        for(int j=0; j<cells.size()+numW; j++) {
            program.set_a(j, row, 0);
        }
        program.set_r(row, CGAL::EQUAL);
        program.set_b(row,0);
        program.set_c(i, 1);
    }
    for(int i=0; i<cells.size(); i++) {
        program.set_c(numW+i, 1);
        program.set_l(numW+i, true, 0);
    }

    CGAL::Quadratic_program_options options;
    options.set_pricing_strategy(CGAL::QP_BLAND);
    Solution s = CGAL::solve_linear_program(program, ET(), options);
    assert(s.solves_linear_program(program));

    if(s.is_infeasible()) {
        // cout<<program.get_m()<<std::endl;
        // cout<<program.get_n()<<std::endl;

        cout<<"not feasible"<<std::endl;
        return;
    } else if (s.is_unbounded()) {
        cout<<(cells.size() - numHealthy)<<std::endl;
        return;
    }

    int countNotZero=0;
    for(Solution::Variable_value_iterator sol_it = s.variable_values_begin(); sol_it != s.variable_values_end(); ↙
        ↳ sol_it++) {
        int index = sol_it - s.variable_values_begin();
        // cout<<*sol_it<<std::endl;
        if(index > numW-1 && *sol_it != 0) {
            ++countNotZero;
        }
    }
    int countDestroyed = (cells.size() - numHealthy) - countNotZero;

    // CGAL::print_quadratic_program(std::cerr, program, "lp");

    cout<<countDestroyed<<std::endl;
}

void delauneyApproach(vector<Point2> &cells, int numHealthy) {
    vector<P> goodCells;
    vector<P> badCells;

    for(int i=0; i<cells.size(); i++) {
        if(i<numHealthy)
            goodCells.push_back(P(cells[i].first, cells[i].second));
        else
            badCells.push_back(P(cells[i].first, cells[i].second));

    }

    Triangulation goodTria;
    goodTria.insert(goodCells.begin(), goodCells.end());

    map<Face_handle, set<int> > conflictMap;

    for(int i=0; i<badCells.size(); i++) {
        vector<Face_handle> conflictFaces;
        goodTria.get_conflicts(badCells[i], back_inserter(conflictFaces));
        for(int j=0; j<conflictFaces.size(); j++) {
            map<Face_handle,set<int> >::iterator it = conflictMap.find(conflictFaces[j]);
            if(it == conflictMap.end()) {
                conflictMap[conflictFaces[j]] = set<int>();
            }
            conflictMap[conflictFaces[j]].insert(i);
            D(cout<<"face " << j << " in conflict with bad cell " << i<<endl;)
        }
    }
```

```
148
149     int maxDestroyed=0;
150     for(Edge_iterator e = goodTria.finite_edges_begin(); e != goodTria.finite_edges_end(); ++e) {
151         Segment eSegment = goodTria.segment(e);
152         Face_handle f1, f2;
153         f1 = e->first;
154         f2 = f1->neighbor(e->second);
155         set<int> conflicts1 = conflictMap[f1];
156         set<int> conflicts2 = conflictMap[f2];
157         vector<int> conflict_intersection;
158         set_intersection(conflicts1.begin(), conflicts1.end(),
159                          conflicts2.begin(), conflicts2.end(),
160                          back_inserter(conflict_intersection));
161
162         vector<int> conflict_difference;
163         set_difference(conflicts1.begin(), conflicts1.end(),
164                          conflicts2.begin(), conflicts2.end(),
165                          back_inserter(conflict_difference));
166
167         int maxPointDestroys=0;
168         for(vector<int>::iterator it = conflict_difference.begin(); it!=conflict_difference.end();
169                 it++) {
170             int thisPointDestroys = 0;
171
172             for(vector<int>::iterator it2 = conflict_difference.begin(); it2!=conflict_difference.end();
173                     it2++) {
174                 if(side_of_bounded_circle(eSegment.source(), eSegment.target(), badCells[*it], badCells[*it2])
175                         == CGAL::ON_BOUNDED_SIDE) {
176                     if(*it!=*it2) {
177                         thisPointDestroys++;
178                     }
179                 }
180             }
181             maxPointDestroys= max(maxPointDestroys, thisPointDestroys+1);
182         }
183
184         int cisize = conflict_intersection.size();
185         maxDestroyed = max(maxDestroyed, maxPointDestroys+cisize);
186       }
187
188     cout<<maxDestroyed<<std::endl;
189 }
190
191
192
193 void testCase() {
194     int numHealthy, numTumor;
195     cin>>numHealthy>>numTumor;
196
197     vector<Point2> cells(numHealthy+numTumor);
198
199     for(int i=0; i<cells.size(); i++) {
200         double x,y;
201         cin>>x>>y;
202         cells[i] = Point2(x,y);
203     }
204
205     // lpApproach(cells, numHealthy);
206     delauneyApproach(cells, numHealthy);
207 }
208
209 int main() {
210     ios_base::sync_with_stdio(false);
211     int testCases;
212     cin >> testCases;
213     while(testCases--) {
214         testCase();
215     }
216 }
```

Missing Roads

Standard maximum matching in Linear Programming formulation:

For each edge between u and v introduce variable x_{uv}. If the variable is set to 1 the the edge is part of the matching, and if it set to 0, the edge is not in the matching. $\max \sum x_{uv}$ Subject to the constraint that no for every vertex u $x_{uv_1} + x_{uv_2} + \cdots \leq 1$