Solver Github link :
shorturl.at/iAV38

Problem A

Crystal Crosswind

Time limit: 5 seconds

You are part of a scientific team developing a new technique to image crystal structures at the molecular level. The technique involves blowing a very fine wind over the surface of the crystal at various angles to detect boundaries (indicated by molecules that are exposed to the wind). This is repeated with different wind directions and the boundaries observed for each direction are recorded. Your team has already collected the data, but – as is often the case with applied science – now the real work, analysis, must begin.

For a given crystal, you will receive the directions in which wind blew over the surface, and the locations of all boundaries encountered by each of these winds. For a wind blowing in direction (w_x, w_y), a boundary is defined as a location (x, y) such that a molecule exists at (x, y) and no molecule exists at $(x - w_x, y - w_y)$. Note that for technical reasons w_x and w_y are not necessarily relatively prime.

The data might not uniquely determine the structure of the crystal. You must find the two unique structures with the minimal and maximal number of molecules consistent with the observations.

For example, in the first sample input, nine different molecules are directly encountered by the given winds. There must be a molecule at location (3, 3) because otherwise (4, 2) would be a boundary for the third wind. For similar reasons, there must be molecules at (4, 4) and (5, 5). There cannot be any further molecules as they would result in additional observations for some of the winds.

Input

The first line of input contains three integers d_x, d_y, and k, where d_x and d_y ($1 \le d_x, d_y \le 10^3$) are the maximum dimensions of the crystal structure, and k ($1 \le k \le 10$) is the number of times wind was blown over the crystal.

Each of the remaining k lines specifies the data for one wind. These lines each start with two integers w_x and w_y ($-d_x \le w_x \le d_x$ and $-d_y \le w_y \le d_y$, but not both zero) denoting the direction of the wind. Then comes an integer b ($0 \le b \le 10^5$) giving the number of boundaries encountered by this wind. The line finishes with b distinct pairs of integers x, y ($1 \le x \le d_x$ and $1 \le y \le d_y$) listing each observed boundary.

You may assume the input is consistent with at least one crystal and that no molecules exist outside the specified dimensions.

Output

Output two textual representations of the crystal structure separated by an empty line. Each structure has d_y rows of d_x characters, with the top-left corner corresponding to location (1, 1). The first is the structure with the minimal number of molecules consistent with the observations, the second is the maximal one. Use '#' for a location where a molecule exists and '.' for a location where no molecule exists.

Sample Input 1

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

Sample Output 1

```
..#...
.#.#..
#.#.#.
.#.#.#
..#.#.
...#..

..#...
.#.#..
#.#.#.
.#.#.#
..#.#.
...#..
```

Sample Input 2

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

Sample Output 2

```
#..#.
.#..#
.#...
..###

##.##
.##.#
.###.
..###
```

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

int X, Y, N;
vector<vector<char>> g;
vector<int> wm, wn;

void doit(int x, int y) {
  char ch = (x >= 1 && x <= X && y >= 1 && y <= Y ? g[y][x] : '.');
  if (!ch) return;
  for (int i = 0; i < N; i++) {
    int x2 = x + (ch == '.' ? wm[i] : -wm[i]);
    int y2 = y + (ch == '.' ? wn[i] : -wn[i]);
    if (x2 >= 1 && x2 <= X && y2 >= 1 && y2 <= Y && !g[y2][x2]) {
      g[y2][x2] = ch;
      doit(x2, y2);
    }
  }
}

int main() {
  while (cin >> X >> Y >> N) {
    g = vector<vector<char>>(Y+1, vector<char>(X+1));
    wm = wn = vector<int>(N);
    for (int i = 0; i < N; i++) {
      int B, x, y;
      cin >> wm[i] >> wn[i] >> B;
      for (int j = 0; j < B; j++) {
        cin >> x >> y;
        g[y][x] = '#';
        int x2 = x-wm[i], y2 = y-wn[i];
        if (x2 >= 1 && x2 <= X && y2 >= 1 && y2 <= Y) g[y2][x2] = '.';
      }
    }
    for (int y = -Y; y <= 2*Y; y++) for (int x = -X; x <= 2*X; x++) doit(x, y);

    for (int y = 1; y <= Y; y++) {
      for (int x = 1; x <= X; x++) cout << (g[y][x] ? g[y][x] : '.');
      cout << endl;
    }
    cout << endl;
    for (int y = 1; y <= Y; y++) {
      for (int x = 1; x <= X; x++) cout << (g[y][x] ? g[y][x] : '#');
      cout << endl;
    }
  }
}
```

Problem B

Dungeon Crawler

Time limit: 5 seconds

Alice and Bob are in charge of testing a new escape room! In this escape room, customers are trapped in a dungeon and have to explore the entire area. The dungeon consists of n rooms connected by exactly $n-1$ corridors. It is possible to travel between any pair of rooms using these corridors.

The helix key
Image generated by DALL-E

Two of the dungeon rooms are special. One of these rooms contains a protective idol known as the "helix key." A different room contains a nasty "dome trap," which prevents the player from moving once activated. Entering the room with the trap before acquiring the key will result in the player being trapped in the dungeon forever. The player cannot start in the same room as the key or the trap.

There are q different scenarios that Alice and Bob wish to examine. In the i^{th} scenario, the player starts in room s_i, the key is in room k_i, and the trap is in room t_i. For each scenario, compute the minimum amount of time needed to explore the entire dungeon without getting trapped.

Input

The first line of input contains two integers n and q, where n ($3 \le n \le 2\,000$) is the number of rooms and q ($1 \le q \le 200\,000$) is the number of scenarios to consider. Rooms are numbered from 1 to n. The next $n-1$ lines each contain three integers u, v, and w indicating that there is a corridor between rooms u and v ($1 \le u, v \le n, u \neq v$) that takes time w ($1 \le w \le 10^9$) to traverse.

Then follow q lines: the i^{th} of these lines contains three distinct integers s_i, k_i, and t_i ($1 \le s_i, k_i, t_i \le n$) indicating the room where the player starts, the room with the key, and the room with the trap, respectively.

Output

For each scenario, output the minimum amount of time needed to visit every room at least once. If it is impossible to visit every room at least once, output `impossible`.

Sample Input 1	Sample Output 1
5 4 1 2 3 1 3 1 3 4 4 3 5 2 1 2 4 1 4 2 5 2 1 4 3 1	15 17 impossible 12

Sample Input 2	Sample Output 2
7 4 1 2 1 1 3 1 1 4 1 1 5 1 1 6 1 1 7 1 1 2 3 5 4 1 3 1 4 2 4 5	11 impossible 10 10

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

int main() {
  int N, Q, U, V, W, S, K, T;
  while (cin >> N >> Q) {
    vector<vector<pair<int,int>>> c(N);
    int64_t tot = 0;
    for (int i = 0; i < N-1; i++) {
      cin >> U >> V >> W;
      U--; V--;
      c[U].push_back({V, W});
      c[V].push_back({U, W});
      tot += W;
    }

    vector<int> depth(N);
    vector<vector<pair<int64_t,int>>> longest(N);
    function<int64_t(int,int,int)> doLongest = [&](int x, int prev, int dp) {
      depth[x] = dp;
      int64_t ret = 0;
      for (auto [y, d] : c[x]) {
        longest[x].push_back({y == prev ? -1 : d+doLongest(y, x, dp+1), y});
        ret = max(ret, longest[x].back().first);
      }
      return ret;
    };
    doLongest(0, -1, 0);
    function<int64_t(int,int,int)> getLongest = [&](int x, int ex1, int ex2) ->
int64_t {
      for (auto [l, y] : longest[x]) {
        if (y != ex1 && y != ex2) return l;
      }
      return 0;
    };
    function<void(int,int,int64_t)> doParLongest = [&](int x, int prev, int64_t
parLongest) {
      for (auto& [l, _] : longest[x]) if (l == -1) l = parLongest;
      sort(longest[x].begin(), longest[x].end(), greater<pair<int64_t,int>>());
      for (auto [y, d] : c[x]) if (y != prev) doParLongest(y, x, d +
getLongest(x, y, -1));
    };
    doParLongest(0, -1, 0);
```

```
    vector<vector<int>> skipNd(N);  // skip-paths going up the tree of length
2^n
    vector<vector<int>> skipPrev(N);  // first node on the skip-path going down
    vector<vector<int64_t>> skipSUp(N);  // max path-to-endpoint, starting at
bottom, maybe entering interior subtree
    vector<vector<int64_t>> skipSDn(N);  // same, but starting at top instead
    vector<vector<int64_t>> skipKUp(N);  // skipSUp, but costs along the skip
path are subtracted, not added
    vector<vector<int64_t>> skipKDn(N);  // same, but starting at top instead
    vector<vector<int64_t>> skipDist(N);  // total length of skip
    function<void(int,int,int64_t)> doSkip = [&](int x, int prev, int64_t d) {
      skipNd[x].push_back(prev);
      skipPrev[x].push_back(x);
      skipDist[x].push_back(d);
      skipSUp[x].push_back(d);
      skipSDn[x].push_back(d);
      skipKUp[x].push_back(0);
      skipKDn[x].push_back(0);
      for (int b = 1; (depth[x]&((1<<b)-1)) == 0; b++) {
        int y = skipNd[x][b-1];
        skipNd[x].push_back(skipNd[y][b-1]);
        skipPrev[x].push_back(skipPrev[y][b-1]);
        skipDist[x].push_back(skipDist[x][b-1] + skipDist[y][b-1]);
        int64_t ymx = getLongest(y, skipPrev[x][b-1], skipNd[y][0]);
        skipSUp[x].push_back(max(skipSUp[x][b-1],  skipDist[x][b-1] + max(ymx,
skipSUp[y][b-1])));
        skipSDn[x].push_back(max(skipSDn[y][b-1],  skipDist[y][b-1] + max(ymx,
skipSDn[x][b-1])));
        skipKUp[x].push_back(max(skipKUp[x][b-1], -skipDist[x][b-1] + max(ymx,
skipKUp[y][b-1])));
        skipKDn[x].push_back(max(skipKDn[y][b-1], -skipDist[y][b-1] + max(ymx,
skipKDn[x][b-1])));
      }
      for (int i = 0; i < c[x].size(); i++) if (c[x][i].first != prev)
doSkip(c[x][i].first, x, c[x][i].second);
    };
    for (int i = 0; i < c[0].size(); i++) doSkip(c[0][i].first, 0,
c[0][i].second);

    auto anc = [&](int x, int y) {
      vector<pair<int,int>> ret;
      while (depth[y] > depth[x]) {
        for (int b = skipNd[y].size()-1; b >= 0; b--) if (depth[y]-(1<<b) >=
depth[x]) {
          y = skipNd[y][b];
          break;
        }
```

```
      }
      while (depth[x] > depth[y]) {
        for (int b = skipNd[x].size()-1; b >= 0; b--) if (depth[x]-(1<<b) >=
depth[y]) {
          ret.push_back({x, b});
          x = skipNd[x][b];
          break;
        }
      }
      while (x != y) {
        for (int b = skipNd[x].size()-1; b >= 0; b--) if (b == 0 || skipNd[x][b]
!= skipNd[y][b]) {
          ret.push_back({x, b});
          x = skipNd[x][b]; y = skipNd[y][b];
          break;
        }
      }
      ret.push_back({x, -1});
      return ret;
    };

    for (int q = 0; q < Q; q++) {
      cin >> S >> K >> T;
      S--; K--; T--;

      auto sk = anc(S, K), st = anc(S, T), ks = anc(K, S), kt = anc(K, T);
      auto path1 = (depth[sk.back().first] > depth[st.back().first] ? sk : st);
      auto path2 = anc(path1.back().first, K);
      auto path4 = (depth[ks.back().first] > depth[kt.back().first] ? ks : kt);
      auto path3 = anc(path4.back().first, S);
      if (path1.back().first == T || path4.back().first == T) { cout <<
"impossible" << endl; continue; }

      int x = S, prev = -1;
      int64_t base = 0, ret = 0;
      for (int i = 0; i+1 < path1.size(); i++) {  // path1 rises from S, not
overlapping path T->K
        auto [y, b] = path1[i];
        ret = max(ret, base + getLongest(x, prev, skipNd[y][0]));
        ret = max(ret, base + skipSUp[y][b]);
        base += skipDist[y][b];
        prev = skipPrev[y][b];
        x = skipNd[y][b];
      }
      for (int i = 0; i+1 < path2.size(); i++) {  // path2 rises from path1,
overlapping path T->K
        auto [y, b] = path2[i];
```

```
        ret = max(ret, base + getLongest(x, prev, skipNd[y][0]));
        ret = max(ret, base + skipKUp[y][b]);
        base -= skipDist[y][b];
        prev = skipPrev[y][b];
        x = skipNd[y][b];
      }
      for (int i = path3.size()-2; i >= 0; i--) {  // path3 descends from path2,
not overlapping path T->K
        auto [y, b] = path3[i];
        ret = max(ret, base + getLongest(x, prev, skipPrev[y][b]));
        ret = max(ret, base + skipSDn[y][b]);
        base += skipDist[y][b];
        prev = skipNd[y][0];
        x = y;
      }
      for (int i = path4.size()-2; i >= 0; i--) {  // path4 descends from path3,
overlapping path T->K
        auto [y, b] = path4[i];
        ret = max(ret, base + getLongest(x, prev, skipPrev[y][b]));
        ret = max(ret, base + skipKDn[y][b]);
        base -= skipDist[y][b];
        prev = skipNd[y][0];
        x = y;
      }
      ret = max(ret, base + getLongest(x, prev, -1));
      cout << 2*tot - ret << endl;
    }
  }
}
```

Problem C

Fair Division

Time limit: 3 seconds

After sailing the Seven Seas and raiding many ships, Cap'n Red and his crew of fellow pirates are finally ready to divide their loot. According to ancient traditions, the crew stands in a circle ordered by a strict pirate hierarchy. Cap'n Red starts by taking a fraction f of the loot and passing the remainder on to the next pirate. That pirate takes the same fraction f of the loot left over by the previous pirate and passes the remainder on to the following pirate. Each pirate behaves in the same way, taking a fraction f of what is left and passing on the rest. The last pirate in the hierarchy passes the remainder on to Cap'n Red, who starts another round of this "fair" division, and so on, indefinitely.

Fortunately, pirates in the 21st century can use a computer to avoid this lengthy process and constant nitpicking when the fraction f does not ex-

Image by ZedH at Pixabay

actly divide the loot at some step. You have been captured by the pirates and asked to come up with a suitable fraction f. As an incentive, Cap'n Red has promised to leave you alive if you succeed.

The fraction f needs to be a rational number strictly between 0 and 1. It is not necessary that f exactly divides the loot remaining at any step of the round-robin process described above. However, the total loot that would be assigned to each pirate by carrying out this process infinitely needs to be an integer.

Input

The input contains one line with two integers n and m, where n ($6 \le n \le 10^6$) is the number of pirates including Cap'n Red and m ($1 \le m \le 10^{18}$) is the total value of their loot.

Output

Output one line with two positive integers p and q, where $f = \frac{p}{q}$ as specified above. If there are multiple suitable fractions, choose one with the smallest q. Among multiple suitable fractions with the same smallest q choose the one with the smallest p. If there is no suitable fraction, output `impossible` instead and hope for mercy.

Sample Input 1	Sample Output 1
8 51000	1 2

Sample Input 2	Sample Output 2
6 91000	2 3

Sample Input 3	Sample Output 3
10 1000000000000000000	impossible

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

int main() {
  int64_t N, M;
  while (cin >> N >> M) {
    if (N > 200) N = 200;

    int p, q;
    vector<double> pw(2);
    for (q = 2; ; q++) {
      pw.push_back(pow(q, N));
      for (p = 1; p < q; p++) {
        double d = pw[q]-pw[q-p];
        if (d > 1.1*M*q) {
          if (p == 1) goto fail;
          continue;
        }
        __int128_t qp = 1, pp = 1;
        for (int i = 0; i < N; i++) qp *= q;
        for (int i = 0; i < N; i++) pp *= q-p;
        if (__int128_t(M)*p % (qp-pp) == 0) goto done;
      }
    }

done:
    cout << p << ' ' << q << endl;
    continue;
fail:
    cout << "impossible" << endl;
  }
}
```

Problem D
Guardians of the Gallery
Time limit: 5 seconds

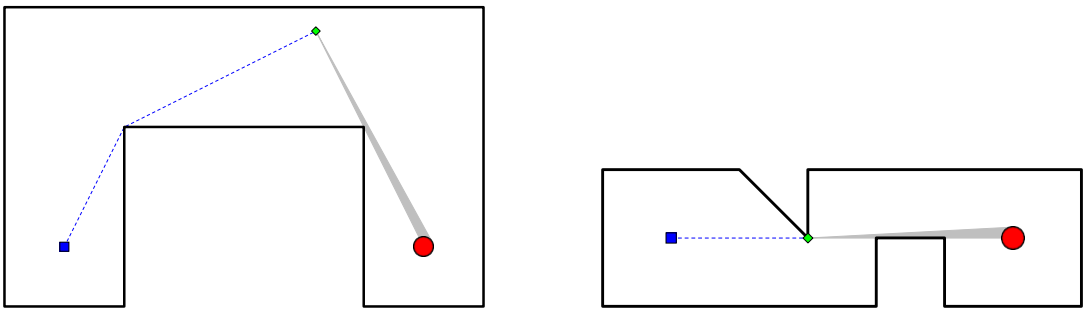
Your local art gallery is about to host an exciting new exhibition of sculptures by world-renowned artists, and the gallery expects to attract thousands of visitors. Unfortunately, the exhibition might also attract the wrong kind of visitors, namely burglars who intend to steal the works of art. In the past, the gallery directors did not worry much about this problem, since their permanent collection is, to be honest, not really worth stealing.

The gallery consists of rooms, and each sculpture in the new exhibition will be placed in a different room. Each room has a security guard and an alarm to monitor the artwork. When an alarm sounds, the guard will run (without leaving the room) from their post to a position where they can see the sculpture directly. This is to check whether the sculpture has in fact been stolen, or whether this is yet another false alarm.

To figure out where to best station the security guard, the gallery directors would like to know how long it takes for the guard to see a given sculpture. They hope that you can help!

Every room is on a single floor, and the layout of the walls can be approximated by a simple polygon. The locations of the guard and the sculpture are distinct points strictly inside the polygon. The sculpture is circular, with a negligibly small (but positive) radius. To verify that the sculpture is still present, the guard needs to be able to see at least half of it.

Figure D.1 illustrates two examples. In each case, the guard starts at the blue square on the left, and the sculpture is located at the red circle on the right. The dotted blue line shows the optimal path for the guard to move. Once the guard reaches the location marked by the green diamond, half of the sculpture can be seen.



(a) Sample Input 1 (b) Sample Input 2

Figure D.1: Illustration of sample inputs.

Input

The first line of input contains an integer n ($3 \leq n \leq 100$), the number of vertices that describe the polygon. This is followed by n lines each containing two integers x and y ($0 \leq x, y \leq 1\,000$), giving the coordinates of the polygon vertices in counterclockwise order. The next line contains two integers x_g and y_g, which specify the location of the guard. Finally, the last line contains two integers x_s and y_s, which specify the location of the center of the sculpture. The polygon is simple, that is, its vertices are distinct and no two edges of the polygon intersect or touch, other than consecutive edges which touch at their common vertex. In addition, no two consecutive edges are collinear.

Output

Output the minimum distance that the guard has to move to be able to see at least half the sculpture. Your answer must have an absolute or relative error of at most 10^{-6}.

Sample Input 1	Sample Output 1
8 0 0 20 0 20 30 60 30 60 0 80 0 80 50 0 50 10 10 70 10	58.137767414994535

Sample Input 2	Sample Output 2
11 0 0 4 0 4 1 5 1 5 0 7 0 7 2 3 2 3 1 2 2 0 2 1 1 6 1	2.0

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

#define EPS 1e-9

struct Point {
  long double x, y;
  Point(long double x = 0.0, long double y = 0.0) : x(x), y(y) {}
  Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
  Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
  Point operator*(long double c) const { return {x*c, y*c}; }
  Point operator/(long double c) const { return {x/c, y/c}; }
  long double Len() const { return hypot(x, y); }
};

inline long double CrossProd(const Point& a, const Point& b) {
  return a.x*b.y - a.y*b.x;
}

inline long double DotProd(const Point& a, const Point& b) {
  return a.x*b.x + a.y*b.y;
}

long double RayIntersect(const Point& a, const Point& b, const Point& c, const
Point& d, int* sides = NULL) {
  long double cp1 = CrossProd(c-a, b-a), cp2 = CrossProd(d-a, b-a);
  long double dp1 = DotProd(c-a, b-a), dp2 = DotProd(d-a, b-a);
  if (sides) *sides = (cp1 < -EPS || cp2 < -EPS) + 2 * (cp1 > EPS || cp2 > EPS);
  if (cp1 < -EPS && cp2 < -EPS || cp1 > EPS && cp2 > EPS) return -1.0;
  return (abs(cp1) < EPS && abs(cp2) < EPS) ? min(dp1, dp2) : (dp1*cp2-
dp2*cp1)/(cp2-cp1);
}

bool PointOnLine(const Point& a, const Point& b, const Point& p) {
  long double ln = (b-a).Len(), cp = CrossProd(b-a, p-a), dp = DotProd(b-a, p-
a);
  return abs(cp/ln) < EPS && dp/ln > -EPS && dp/ln < ln+EPS;
}

int N;
vector<Point> p;
```

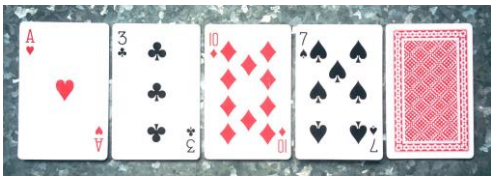
```
int main() {
  while (cin >> N) {
    p.resize(N+2);
    for (int i = 0; i < N+2; i++) cin >> p[i].x >> p[i].y;

    for (int i = 0; i < N+1; i++) {
      Point a = p[N+1], b = p[i];
      if ((b-a).Len() < EPS) { p.push_back(b); continue; }
      b = (b-a)/(b-a).Len() + a;
      vector<pair<long double, int>> inter;
      for (int j = 0; j < N; j++) {
        int sides = 0;
        long double rd = RayIntersect(a, b, p[j], p[(j+1)%N], &sides);
        if (rd < 0) continue;
        inter.push_back({rd, sides});
      }
      sort(inter.begin(), inter.end());
      long double maxd = 0.0;
      for (int j = 0, sides = 0; sides != 3; j++) {
        maxd = inter[j].first;
        sides |= inter[j].second;
      }
      p.push_back((b-a)*maxd + a);
      for (int j = 0; j <= N; j++) {
        long double rd = RayIntersect(a, b, p[j], p[j]+Point(b.y-a.y, a.x-
b.x)*1.1e3);
        if (rd < 0) rd = RayIntersect(a, b, p[j], p[j]+Point(a.y-b.y, b.x-
a.x)*1.1e3);
        if (rd > EPS && rd < maxd-EPS) p.push_back((b-a)*rd + a);
      }
    }

    vector<long double> dist(p.size(), 1e10);
    priority_queue<pair<long double, int>> q;
    q.push({0.0, N});
    for (;;) {
      int i = q.top().second;
      if (i > N) break;
      long double d = -q.top().first;
      q.pop();
      if (d >= dist[i]) continue;
      dist[i] = d;
      for (int j = 0; j < p.size(); j++) {
        Point a = p[i], b = p[j];
        long double ln = (b-a).Len();
        int ni = 0;
        if (ln < EPS) goto pass;
        if (i < N && PointOnLine(p[i], p[(i+1  )%N], p[j])) goto pass;
        if (i < N && PointOnLine(p[i], p[(i+N-1)%N], p[j])) goto pass;
```

```
        b = (b-a)/ln + a;
        for (int k = 0; k < N; k++) {
          long double rd = RayIntersect(a, b, p[k], p[(k+1)%N]);
          if (rd > EPS && rd < ln-EPS) goto fail;
        }
        a = (p[i]+p[j])/2;
        b = a+Point(cos(42), sin(42));
        for (int k = 0; k < N; k++) {
          long double rd = RayIntersect(a, b, p[k], p[(k+1)%N]);
          ni += (rd > EPS);
        }
        if (ni%2 == 0) goto fail;
pass:   q.push({-d-ln, j});
fail:;
      }
    }
    printf("%.12Lf\n", -q.top().first);
  }
}
```

Problem E

Hand of the Free Marked

Time limit: 2 seconds



Example placement of cards for $k = 5$

There is a fairly well-known mentalism trick known as the Fitch Cheney trick. From a deck of n playing cards, k are selected uniformly at random and given to an assistant while the magician is out of the room. The assistant places $k - 1$ of the selected cards on a table, face up, and the single remaining card face down. The cards are placed in a single row with the face-down card at the end (see the picture for an example). The magician enters the room, looks at the cards on the table, and announces what the k^{th} card is, although its face is hidden. The trick is typically done with $n = 52$ and $k = 5$.

The assistant uses two ways of passing information to the magician. First, they can pick which one of the k cards to keep hidden. Second, they can rearrange the other $k - 1$ cards in a specific way. For the case $n = 52$ and $k = 5$ both techniques are needed, since there are only 24 ways of rearranging four cards, which is not enough to reliably signal the fifth card. It is an interesting exercise to come up with a simple, easy-to-remember strategy for executing this trick, but right now you have another concern.

You were planning to perform this trick today, but just now you have learned that the deck has more cards than you expected. The trick may be impossible! In desperation, you have decided to cheat a little. You have m distinguishable ways of marking the backs of the playing cards. You have marked the backs of all n cards, allowing you to narrow down the possibilities for the k^{th} card. For example, if there are 6 cards marked with a particular method, and you see that the back of the k^{th} card is marked with that method, you know it must be one of those 6 cards.

Determine the probability that you will successfully guess the k^{th} card, assuming you and the assistant execute an optimal (but likely very complicated!) strategy.

Input

The input contains one line with several integers. The first integer gives k ($2 \le k \le 10$), the number of cards that will be selected. The second integer gives m ($1 \le m \le 10$), the number of ways of marking the cards. The line is completed by m positive integers, giving the number of cards marked with each distinct method. The sum of these m integers is n ($k \le n \le 10^9$), which is the size of the deck.

Output

Output the highest possible probability of guessing the k^{th} card correctly, accurate up to an absolute error of 10^{-9}.

Sample Input 1	Sample Output 1
4 1 28	0.96

Sample Input 2	Sample Output 2
3 3 5 12 3	0.854385964912

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

int main() {
  int K, M;
  while (cin >> K >> M) {
    vector<int> col(M);
    for (int i = 0; i < M; i++) cin >> col[i];

    double hands = 0.0, matched = 0.0;
    vector<int> v(K);
    vector<double> fact(K);
    function<void(int,int)> rec = [&] (int i, int x) {
      if (i < K) {
        for (v[i] = x; v[i] < M; v[i]++) rec(i+1, v[i]);
        return;
      }
      double h = 1.0, m = 0.0;
      for (int i = 0, last = -1, n = 1; i < K; i++) {
        if (v[i] == last) n++; else n = 1;
        last = v[i];
        fact[i] = double(col[v[i]]-(n-1)) / n;
        h *= fact[i];
      }
      if (h == 0.0) return;
      for (int i = 0; i < K; i++) if (i == K-1 || v[i] != v[i+1]) {
        m += h/fact[i];
      }
      for (int i = 1; i <= K-1; i++) m *= i;
      hands += h;
      matched += min(h, m);
    };
    rec(0, 0);
    printf("%.12lf\n", matched/hands);
  }
}
```

Problem F

Islands from the Sky

Time limit: 2 seconds

You might never have heard of the island group of Iceepeecee, but that suits their inhabitants just fine.

Located in a remote part of the South Pacific, they are truly off the beaten track, without any regular air or sea traffic, and they have remained a tropical paradise with unspoiled local fauna and flora.

Being off the map is great when you don't want to be overrun by hordes of tourists, but not so ideal when you actually do need a map for some reason. One such reason came up recently: Iceepeecee's central government needs an exact map of the islands to apportion government funds. Even tropical paradises need money, so Iceepeecee needs a map!

The easiest way to create a map would be an aerial survey. After dismissing chartering planes as too expensive, building an air balloon as too dangerous, and fitting carrier pigeons with cameras as too cruel to animals, they had a brilliant idea. Even with its remote location, there are still plenty of commercial airplanes crossing the skies above Iceepeecee. What if one mounted cameras on flights that are already scheduled to fly anyway? That would be a cheap solution to the problem!

Iceepeecee's plan is to install line-scan cameras on the planes. These cameras point straight downwards and collect images one line segment at a time, orthogonal to the flight path. The photographed line segment will be determined by the altitude that the plane is flying at, and the camera's aperture angle θ (see Figure F.1). Greater angles θ mean that the camera can see more, but also that the camera is more expensive.

Moreover, Iceepeecee wants to make sure that each island is observed in its entirety by at least one flight. That means it is not sufficient that an island is only partially photographed by multiple flights, even if the combination of the photographs covers the whole island.

Flight paths follow straight line segments in three-dimensional space, that is, $(x_1, y_1, z_1) - (x_2, y_2, z_2)$ (see Figure F.2), where the z-coordinates give the altitude of the plane. Photographs are taken only along these line segments.

Given the location of their islands and flights, Iceepeecee wants to find the smallest aperture angle θ that allows for a successful survey. Can you help?

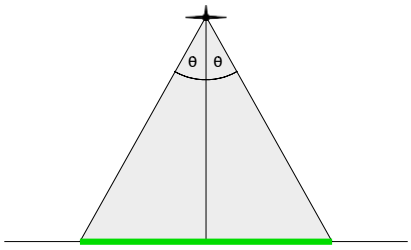
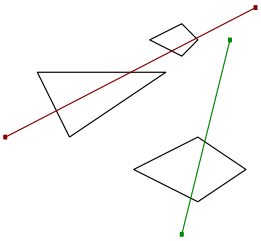
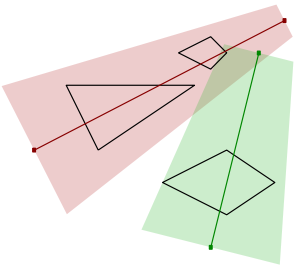


Figure F.1: A view of the plane, shown head-on. Its camera points downward and can see the part of the ground underneath the plane that is shown in green. How much is visible depends on the aperture angle θ.



(a) Three islands (shown in black) and two flight paths (red and green). Altitudes are not shown.



(b) The shaded areas represent the ground visible on the two flight paths for an optimally chosen θ.

Figure F.2: Surveying three islands via two flight paths. This corresponds to the first sample input.

Input

The input describes a set of islands and flight paths. It starts with a line containing two integers n and m, the number n of islands, and the number m of flight paths, respectively ($1 \le n, m \le 100$). This is followed by descriptions of the n islands. Each island description starts with a line containing a single integer n_i, the number of vertices of the polygon describing the i^{th} island ($3 \le n_i \le 100$). It is followed by n_i lines, each containing two integers x_{ij}, y_{ij} ($|x_{ij}|, |y_{ij}| \le 10^6$), specifying the vertices for the i^{th} island in counterclockwise order. Each island's polygon is simple, that is, its vertices are distinct and no two edges of the polygon intersect or touch, other than consecutive edges which touch at their common vertex. Different islands do not intersect or touch.

The input concludes with another m lines, each describing a flight path. Each such line contains six integers x_1, y_1, z_1, x_2, y_2, z_2 ($|x_i|, |y_i|, |z_i| \le 10^6$, $z_i > 0$ and $(x_1, y_1) \neq (x_2, y_2)$). They specify that aflight takes place from (x_1, y_1, z_1) to (x_2, y_2, z_2).

Output

Output the smallest angle θ (in degrees) that allows for a complete survey of the islands with the given flights. The answer should be exact to an absolute or relative error of 10^{-6}. If there is no such angle, then output `impossible`. The input is chosen such that if the coordinates of the island vertices are changed by at most $\pm 10^{-8}$, then the answer will not change more than the allowed rounding error.

Sample Input 1

```
3 2
3
20 30
50 50
10 50
4
40 20
60 10
75 20
60 30
4
45 60
55 55
60 60
55 65
0 30 20 78 70 5
55 0 20 70 60 10
```

Sample Output 1

```
48.031693036
```

Sample Input 2

```
1 1
4
0 0
10 0
10 10
0 10
5 5 10 15 5 10
```

Sample Output 2

```
impossible
```

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

const long double PI = 2*acosl(0);

struct Point {
  long double x, y;
  Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
  Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
  Point operator*(long double c) const { return {x*c, y*c}; }
  Point operator/(long double c) const { return {x/c, y/c}; }
  long double len() const { return hypot(x, y); }
};

inline long double DotProd(const Point& a, const Point& b) {
  return a.x*b.x + a.y*b.y;
}

inline long double CrossProd(const Point& a, const Point& b) {
  return a.x*b.y - a.y*b.x;
}

bool LineSegIntersection(const Point& a1, const Point& a2, const Point& b1,
const Point& b2) {
  long double cp1 = CrossProd(b2-b1, a1-b1);
  long double cp2 = CrossProd(b2-b1, a2-b1);
  if (cp1 > 0 && cp2 > 0) return false;
  if (cp1 < 0 && cp2 < 0) return false;
  cp1 = CrossProd(a2-a1, b1-a1);
  cp2 = CrossProd(a2-a1, b2-a1);
  if (cp1 > 0 && cp2 > 0) return false;
  if (cp1 < 0 && cp2 < 0) return false;
  return true;
}

int main() {
  int N, M, NI;
  while (cin >> N >> M) {
    vector<vector<Point>> I(N);
    for (auto& island : I) {
      cin >> NI;
      island.resize(NI);
      for (int i = 0; i < NI; i++) cin >> island[i].x >> island[i].y;
    }
    vector<Point> F1(M), F2(M);
    vector<long double> FZ1(M), FZ2(M);
```

```
    for (int i = 0; i < M; i++) cin >> F1[i].x >> F1[i].y >> FZ1[i] >> F2[i].x
>> F2[i].y >> FZ2[i];

    long double lo = 0.0, hi = PI/2;
    for (int rep = 0; rep < 64; rep++) {
      long double th = (hi+lo)/2;
      vector<bool> seen(N);
      for (int f = 0; f < M; f++) {
        vector<Point> poly;
        Point ortho{F1[f].y-F2[f].y, F2[f].x-F1[f].x};
        ortho = ortho / ortho.len();
        poly.push_back(F1[f] - ortho * (FZ1[f]*tan(th)));
        poly.push_back(F2[f] - ortho * (FZ2[f]*tan(th)));
        poly.push_back(F2[f] + ortho * (FZ2[f]*tan(th)));
        poly.push_back(F1[f] + ortho * (FZ1[f]*tan(th)));
        long double mxx = 1e7;
        for (auto [x, _] : poly) mxx = max(mxx, x);
        for (int i = 0; i < I.size(); i++) if (!seen[i]) {
          bool fail = false;
          for (auto const& p : I[i]) {
            int cnt = 0;
            for (int i = 0; i < poly.size(); i++) {
              Point& a = poly[i];
              Point& b = poly[(i+1)%poly.size()];
              cnt += LineSegIntersection(a, b, p, Point{mxx+1337, p.y+7331});
            }
            if (cnt%2 == 0) { fail = true; break; }
          }
          if (!fail) seen[i] = true;
        }
      }
      if (seen == vector<bool>(N, true)) hi = th; else lo = th;
    }

    if (hi == PI/2) {
      cout << "impossible" << endl;
    } else {
      cout << fixed << setprecision(9) << (hi+lo)/2 * 180/PI << endl;
    }
  }
}
```

Problem G

Mosaic Browsing

Time limit: 6 seconds

The International Center for the Preservation of Ceramics (ICPC) is searching for motifs in some ancient mosaics. According to the ICPC's definition, a *mosaic* is a rectangular grid where each grid square contains a colored tile. A *motif* is similar to a mosaic but some of the grid squares can be empty. Figure G.1 shows an example motif and mosaic.

The rows of an $r_q \times c_q$ mosaic are numbered 1 to r_q from top to bottom, and the columns are numbered 1 to c_q from left to right.

A contiguous rectangular subgrid of the mosaic matches the motif if every tile of the motif matches the color of the corresponding tile of the subgrid. Formally, an $r_p \times c_p$ motif appears in an $r_q \times c_q$ mosaic at position (r, c) if for all $1 \le i \le r_p$, $1 \le j \le c_p$, the tile $(r + i - 1, c + j - 1)$ exists in the mosaic and either the square (i, j) in the motif is empty or the tile at (i, j) in the motif has the same color as the tile at $(r + i - 1, c + j - 1)$ in the mosaic.

Given the full motif and mosaic, find all occurrences of the motif in the mosaic.

Figure G.1: Motif (left) and mosaic (right) of Sample Input 1.

Input

The first line of input contains two integers r_p and c_p, where r_p and c_p ($1 \le r_p, c_p \le 1\,000$) are the number of rows and columns in the motif. Then r_p lines follow, each with c_p integers in the range $[0, 100]$, denoting the color of the motif at that position. A value of 0 denotes an empty square.

The next line of input contains two integers r_q and c_q where r_q and c_q ($1 \le r_q, c_q \le 1\,000$) are the number of rows and columns in the mosaic. Then r_q lines follow, each with c_q integers in the range $[1, 100]$, denoting the color of the mosaic at that position.

Output

On the first line, output k, the total number of matches. Then output k lines, each of the form r c where r is the row and c is the column of the top left tile of the match. Sort matches by increasing r, breaking ties by increasing c.

Sample Input 1	Sample Output 1
2 2 1 0 0 1 3 4 1 2 1 2 2 1 1 1 2 2 1 3	3 1 1 1 3 2 2

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

typedef bitset<1000> bs;

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int PY, PX, QY, QX, PC, QC;
  while (cin >> PY >> PX) {
    vector<bs> g(PX);
    vector<vector<bs>> cg(101, vector<bs>(PX));
    for (int y = 0; y < PY; y++)
    for (int x = 0; x < PX; x++) {
      cin >> PC;
      if (!PC) continue;
      g[PX-1-x].set(PY-1-y);
      cg[PC][PX-1-x].set(PY-1-y);
    }
    for (auto& v : cg) for (auto& b : v) b = ~b;

    cin >> QY >> QX;
    vector<bs> fail(QX);
    vector<vector<bs>> gg(101, vector<bs>(PX));
    vector<vector<int>> cookie(101, vector<int>(PX, -1));
    for (int y = 0; y < QY; y++) {
      for (int x = 0; x < QX; x++) {
        cin >> QC;
        for (int px = max((PX-1)-x, 0); px < PX; px++) {
          int fx = x-(PX-1)+px;
          if (fx > QX-PX) break;
          if (cookie[QC][px] != y) {
            cookie[QC][px] = y;
            if (y < PY) {
              gg[QC][px] = (g[px] & cg[QC][px]) >> (PY-1)-y;
            } else {
              gg[QC][px] = (g[px] & cg[QC][px]) << y-(PY-1);
            }
          }
          fail[x-(PX-1)+px] |= gg[QC][px];
        }
      }
    }

    vector<pair<int, int>> v;
    for (int y = 0; y <= QY-PY; y++)
    for (int x = 0; x <= QX-PX; x++) if (!fail[x][y]) {
```

```
      v.push_back({x, y});
    }
    cout << v.size() << endl;
    for (auto [x, y] : v) cout << y+1 << ' ' << x+1 << endl;
  }
}
```

Problem H

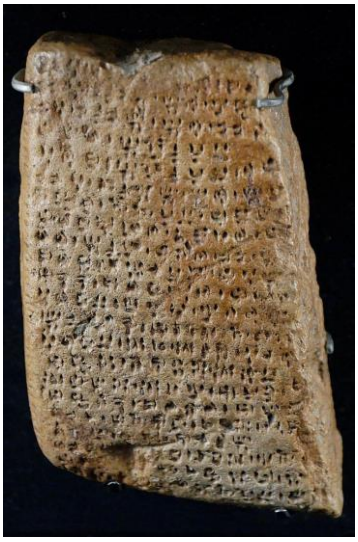
Prehistoric Programs

Time limit: 6 seconds

Archaeologists have discovered exciting clay tablets in deep layers of Alutila Cave. Nobody was able to decipher the script on the tablets, except for two symbols that seem to describe nested structures not unlike opening and closing parentheses in LISP. Could it be that humans wrote programs thousands of years ago?

Taken together, the tablets appear to describe a great piece of work – perhaps a program, or an epic, or even tax records! Unsurprisingly, after such a long time, the tablets are in a state of disorder. Your job is to arrange them into a sequence so that the resulting work has a properly nested parenthesis structure. Considering only opening and closing parentheses, a properly nested structure is either

- (), or
- (A), where A is a properly nested structure, or
- AB, where A and B are properly nested structures.



Clay tablet with undeciphered script
Source: Wikimedia Commons

Input

The first line of input contains one integer n ($1 \leq n \leq 10^6$), the number of tablets. Each of the remaining n lines describes a tablet, and contains a non-empty string of opening and closing parentheses; symbols unrelated to the nesting structure are omitted. The strings are numbered from 1 to n in the order that they appear in the input. The input contains at most 10^7 parentheses.

Output

Output a permutation of the numbers from 1 to n such that concatenating the strings in this order results in a properly nested structure. If this happens for multiple permutations, any one of them will be accepted. If there is no such permutation, output `impossible` instead.

Sample Input 1	Sample Output 1
2 ())()() (()	2 1

Sample Input 2	Sample Output 2
5 ( )) (( )) (	1 5 3 2 4

Sample Input 3	Sample Output 3
2 (( )	impossible

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

int main() {
  int N;
  while (cin >> N) {
    vector<vector<int>> fv, bv;
    int tot = 0;
    for (int i = 0; i < N; i++) {
      string S;
      cin >> S;
      int b = 0, mn = 0;
      for (int j = 0; j < S.size(); j++) {
        b += (S[j] == '(') - (S[j] == ')');
        mn = min(mn, b);
      }
      if (b >= 0) fv.push_back({-mn, b, i}); else bv.push_back({b-mn, -b, i});
      tot += b;
    }
    if (tot) goto fail;

    for (int i = 0; i < 2; i++) {
      auto& v = i ? fv : bv;
      sort(v.begin(), v.end());
      for (int j = 0, cur = 0; j < v.size(); j++) {
        if (cur < v[j][0]) goto fail;
        cur += v[j][1];
      }
    }

    reverse(bv.begin(), bv.end());
    for (auto const& v : fv) cout << v[2]+1 << endl;
    for (auto const& v : bv) cout << v[2]+1 << endl;
    continue;
fail:
    cout << "impossible" << endl;
  }
}
```

Problem I

Spider Walk

Time limit: 6 seconds

Charlotte the spider sits at the center of her spiderweb, which consists of a series of silken straight strands that go from the center to the outer boundary of the web. Charlotte's web also has bridges, each of which connects two adjacent strands. The two endpoints of a bridge always have the same distance to the center of the spiderweb.

Image by FBR

When Charlotte has finished a late-night feasting in the center and wants to retreat to some corner, she walks to the edge on autopilot. To do this, she picks a starting strand, and walks along it until she meets the first bridge on that strand. She will cross the bridge and go to the other strand, and then keeps walking outwards until she meets another bridge. Then she will cross that bridge, and repeat this process, until there are no more bridges on the current strand, and then she will walk to the end of the current strand. Note that Charlotte must cross all the bridges that she meets. Figure I.1 illustrates one path Charlotte could take.

Charlotte's favorite corner to sleep in during the daytime is at the end of strand s. For each possible starting strand, she wants to know the minimum number of bridges to add to the original web in order to end at s. Charlotte can add a bridge at any point along the strand, as long as the added bridge does not touch any other bridge. The two endpoints of any added bridge must have the same distance to the center of the spiderweb, and the bridge must connect two adjacent strands.

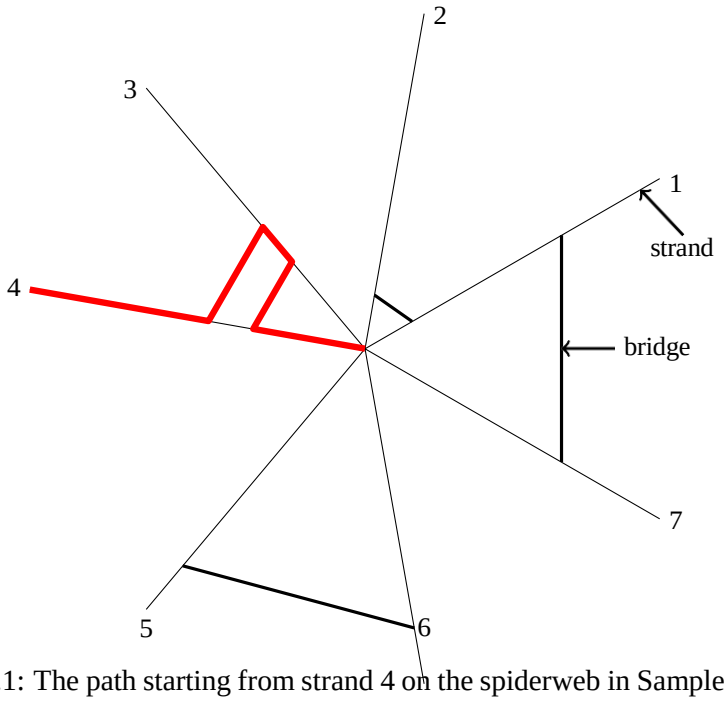


Figure I.1: The path starting from strand 4 on the spiderweb in Sample Input 1.

Input

The first line of input has three integers n, m, and s, where n ($3 \le n \le 200\,000$) is the number of strands, m ($0 \le m \le 500\,000$) is the number of bridges, and s ($1 \le s \le n$) is Charlotte's favorite strand. Strands are labeled from 1 to n in counterclockwise order. Each of the remaining m lines contains two integers d and t describing a bridge, where d ($1 \le d \le 10^9$) is the bridge's distance from the center of the spiderweb and t ($1 \le t \le n$) is the first strand of the bridge in counterclockwise order. Specifically, if $1 \le t < n$, then the bridge connects strands t and $t+1$. If $t = n$, then the bridge connects strands 1 and n. All bridge distances d are distinct.

Output

Output n lines, where the i^{th} line is the minimum number of bridges Charlotte needs to add in order to end at strand s after walking on autopilot from strand i.

Sample Input 1

```
7 5 6
2 1
4 3
6 3
8 7
10 5
```

Sample Output 1

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

Sample Input 2

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

Sample Output 2

```
1
1
0
1
```

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

int main() {
  int N, M, S, D, T;
  while (cin >> N >> M >> S) {
    S--;
    vector<pair<int, int>> b;
    for (int i = 0; i < M; i++) {
      cin >> D >> T;
      b.push_back({-D, T-1});
    }
    sort(b.begin(), b.end());

    map<int, int> m;
    m[S] = 1;
    m[(S+N/2)%N] = 0;
    m[(S+(N+1)/2)%N] = -1;
    auto pred = [&](map<int,int>::iterator it) { return --(it == m.begin() ?
m.end() : it); };
    auto succ = [&](map<int,int>::iterator it) { ++it; return (it == m.end() ?
m.begin() : it); };
    auto getAll = [&](int x) {
      auto it = pred(m.upper_bound(x)), pit = it, nit = succ(it);
      if (pit->first == x) pit = pred(pit);
      if (nit->first != (x+1)%N) nit = it;
      return make_tuple(pit, it, nit);
    };
    auto set = [&](int x, int d) {
      auto [pit, it, nit] = getAll(x);
      int xd = it->second, pd = pit->second, nd = nit->second;
      if (xd == d) return;
      if (d == pd) m.erase(x); else m[x] = d;
      if (nd == d) m.erase((x+1)%N); else m[(x+1)%N] = nd;
    };
    auto swp = [&](int x) {
      auto [pit, it, nit] = getAll(x);
      int xd = it->second, pd = pit->second, nd = nit->second;
      if (xd == 0) return;
      xd = -xd; pd -= xd; nd -= xd;
      if (pd ==  2) { pd--; xd++; }
      if (nd == -2) { nd++; xd--; }
      if (pd == -2) {
        pd++;
        set((pit->first + (N-1))%N, pred(pit)->second - 1);
```

```
      }
      if (nd == 2) {
        nd--;
        set(succ(nit)->first, succ(nit)->second + 1);
      }
      set((x+N-1)%N, pd);
      set(x, xd);
      set((x+1)%N, nd);
    };

    int s = S;
    for (auto [_, t] : b) {
      swp(t);
      if (s == t) s = (s+1)%N; else if (s == (t+1)%N) s = t;
    }

    vector<int> ret(N);
    for (int i = 0, cur = 0, d = 0; i < N; i++) {
      ret[s] = cur;
      if (m.count(s)) d = m[s];
      cur += d;
      s = (s+1)%N;
    }
    for (auto x : ret) cout << x << endl;
  }
}
```

Problem J
Splitstream
Time limit: 3 seconds

A splitstream system is an acyclic network of nodes that processes finite sequences of numbers. There are two types of nodes (illustrated in Figure J.1):

- A *split* node takes a sequence of numbers as input and distributes them alternatingly to its two outputs. The first number goes to output 1, the second to output 2, the third to output 1, the fourth to output 2, and so on, in this order.
- A *merge* node takes two sequences of numbers as input and merges them alternatingly to form its single output. The output contains the first number from input 1, then the first from input 2, then the second from input 1, then the second from input 2, and so on. If one of the input sequences is shorter than the other, then the remaining numbers from the longer sequence are simply transmitted without being merged after the shorter sequence has been exhausted.

Ganga-Brahmaputra delta

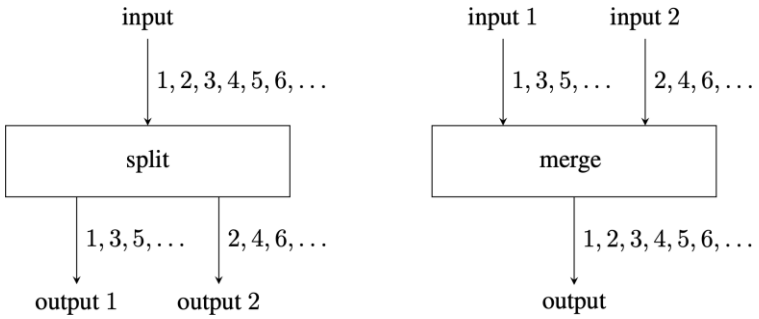


Figure J.1: Illustration of how split and merge nodes work.

The overall network has one input, which is the sequence of positive integers $1, 2, 3, \ldots, m$. Any output of any node can be queried. A query will seek to identify the k^{th} number in the sequence of numbers for a given output and a given k. Your task is to implement such queries efficiently.

Input

The first line of input contains three integers m, n, and q, where m ($1 \leq m \leq 10^9$) is the length of the input sequence, n ($1 \leq n \leq 10^4$) is the number of nodes, and q ($1 \leq q \leq 10^3$) is the number of queries.

The next n lines describe the network, one node per line. A split node has the format `S` x y z, where x, y and z identify its input, first output and second output, respectively. A merge node has the format `M` x y z, where x, y and z identify its first input, second input and output, respectively. Identifiers x, y and z are distinct positive integers. The overall input is identified by 1, and the remaining input/output identifiers form a consecutive sequence beginning at 2. Every input identifier except 1 appears as exactly one output. Every output identifier appears as the input of at most one node.

Each of the next q lines describes a query. Each query consists of two integers x and k, where x ($2 \leq x \leq 10^5$) is a valid output identifier and k ($1 \leq k \leq 10^9$) is the index of the desired number in that sequence. Indexing in a sequence starts with 1.

Output

For each query x and k output one line with the k^{th} number in the output sequence identified by x, or `none` if there is no element with that index number.

Sample Input 1

```
200 2 2
S 1 2 3
M 3 2 4
4 99
4 100
```

Sample Output 1

```
100
99
```

Sample Input 2

```
100 3 6
S 1 4 2
S 2 3 5
M 3 4 6
6 48
6 49
6 50
6 51
6 52
5 25
```

Sample Output 2

```
47
98
49
51
53
100
```

Sample Input 3

```
2 3 3
S 1 2 3
S 3 4 5
M 5 2 6
3 1
5 1
6 2
```

Sample Output 3

```
2
none
none
```

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

int main() {
  int M, N, Q;
  while (cin >> M >> N >> Q) {
    vector<vector<int>> nd(1);
    int mx = 0;
    for (int i = 1; i <= N; i++) {
      char ch;
      int x, y, z;
      cin >> ch >> x >> y >> z;
      mx = max<int>({mx, x, y, z});
      if (ch == 'S') nd.push_back({x, 0, y, z}); else nd.push_back({x, y, z,
0});
    }
    vector<int> oin(mx+1), oout(mx+1);
    for (int i = 1; i <= N; i++) oin[nd[i][0]] = oin[nd[i][1]] = oout[nd[i][2]]
= oout[nd[i][3]] = i;

    vector<int> osz(mx+1, -1);
    osz[0] = 0;
    function<void(int,int)> rec = [&](int x, int sz) {
      osz[x] = sz;
      if (oin[x] == 0) return;
      auto const& v = nd[oin[x]];
      if (osz[v[0]] == -1 || osz[v[1]] == -1) return;
      if (v[1]) {
        rec(v[2], osz[v[0]]+osz[v[1]]);
      } else {
        rec(v[2], (osz[v[0]]+1)/2);
        rec(v[3], (osz[v[0]]  )/2);
      }
    };
    rec(1, M);
    for (int i = 2; i <= mx; i++) if (!oout[i]) rec(i, 0);

    for (int q = 0; q < Q; q++) {
      int x, k;
      cin >> x >> k;
      if (k > osz[x]) { cout << "none" << endl; continue; }
      while (x != 1) {
        auto const& v = nd[oout[x]];
        if (v[1]) {
          int sz = min(osz[v[0]], osz[v[1]]);
          if (k <= 2*sz) {
```

```
            x = v[!(k%2)];
            k = (k+1)/2;
          } else {
            x = v[osz[v[1]] > osz[v[0]]];
            k -= sz;
          }
        } else {
          k = 2*k - (v[2] == x);
          x = v[0];
        }
      }
      cout << k << endl;
    }
  }
}
```

Problem K

Take On Meme

Time limit: 4 seconds

The Internet can be so fickle. You work for a small ad agency, Mimi's Mammoth Memes. Your ad campaigns are very cheap, and rely on the hope of producing the next hit viral meme. Unfortunately, the last four hundred or so memes have failed to take off, despite having been precisely engineered to appeal to every single person on Earth. You're not sure what exactly went wrong, but you've decided to try a new approach: crowd sourcing!

According to your scientific meme theory, all memes can be rated from $-\infty$ to ∞ on two scales: xanthochromism, and yellowishness, also known as (x, y) values. Obviously (you think), the best memes are memorable for being particularly xanthochromic, yellowish, unxanthochromic, or unyellowish. You feel that the "quality" of any meme is directly measurable as its squared Euclidean distance $(x^2 + y^2)$ from the Base Meme $(0, 0)$, otherwise known as All Your Base.

To produce the ultimate viral meme, you'll be taking your company's last few failed memes and throwing them into a tournament, decided by online voting. The tournament can be represented as a rooted tree. Input memes come in at the leaves, and at each internal node, a vote will be held among its k child memes $(x_1, y_1), \ldots, (x_k, y_k)$. After the vote, all the memes will be horrifically mangled and merged into a brand new meme, specifically calculated to emphasize the winner and de-emphasize all the losers: the resultant x value will be

$$\sum_{i=1}^{k} w_i \cdot x_i,$$

where w_i is 1 if the i^{th} child won, and -1 otherwise. The y value is computed similarly. This new meme will move on to the next vote in the tournament – or, if there is no parent, it will be declared the champion and the ultimate meme!

You already have the structure of the tournament planned out, including all the input memes and the internal voting nodes. What is the largest possible quality for any meme that the tournament could produce?

Input

The first line of input contains an integer n ($1 \le n \le 10^4$), giving the total number of nodes in the tournament tree. The next n lines each describe a single tree node indexed from 1 to n. The line for node i starts with an integer k_i ($0 \le k_i \le 100$), the number of children of that node. If k_i is 0, then node i is an input meme and there will be two more integers x_i and y_i ($-10^3 \le x_i, y_i \le 10^3$) describing it. If $k_i > 0$, then k_i different integers j ($i < j \le n$) will follow, giving the indices of the k_i nodes entering this voting step.

All input memes will eventually be merged into the final output meme at node 1. The complete tree will have a height of no more than 10.

Output

Output the largest possible quality for the champion meme at node 1.

Sample Input 1

```
4
3 2 3 4
0 10 1
0 3 6
0 2 7
```

Sample Output 1

```
169
```

Sample Input 2

```
8
3 4 2 5
2 3 8
0 -3 9
0 -5 -7
2 6 7
0 1 4
0 -3 -1
0 1 4
```

Sample Output 2

```
314
```

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <tuple>
#include <vector>
using namespace std;

void Init() {
  srand(time(0));
  ios::sync_with_stdio(false);
  cin.tie(NULL);
}

int64_t cmpx = 1, cmpy = 0;
struct Point {
  int64_t x, y;
  Point operator-() const { return {-x, -y}; }
  Point& operator+=(const Point& p) { x += p.x; y += p.y; return *this; }
  Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
  Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
  bool operator<(const Point& p) const { return x*cmpx + y*cmpy < p.x*cmpx +
p.y*cmpy; }
  bool operator==(const Point& p) const { return x == p.x && y == p.y; }
  Point ortho() const { return {-y, x}; }
  int64_t lensqr() const { return x*x+y*y; }
};

int main() {
  Init();

  int N, M;
  while (cin >> N) {
    vector<vector<int>> ch(N+1);
    vector<Point> p(N+1);
    for (int i = 1; i <= N; i++) {
      cin >> M;
      if (M == 0) {
        cin >> p[i].x >> p[i].y;
      } else {
        ch[i].resize(M);
        for (auto& x : ch[i]) cin >> x;
      }
    }
```

```
    int64_t ret = 0;
    auto tryAngle = [&](Point dir) -> pair<Point, Point> {
      cmpx = dir.x; cmpy = dir.y;
      function<pair<Point,Point>(int)> doit = [&](int x) -> pair<Point,Point> {
        if (ch[x].size() == 0) return {p[x], p[x]};
        auto [mntot, mxtot] = doit(ch[x][0]);
        Point mndiff = mxtot+mntot, mxdiff = mndiff;
        for (int i = 1; i < ch[x].size(); i++) {
          auto [mn, mx] = doit(ch[x][i]);
          mntot += mn;
          mxtot += mx;
          mndiff = min(mndiff, mx+mn);
          mxdiff = max(mxdiff, mx+mn);
        }
        return {-mxtot+mndiff, -mntot+mxdiff};
      };
      auto [mn, mx] = doit(1);
      ret = max(ret, mx.lensqr());
      ret = max(ret, mn.lensqr());
      return {mn, mx};
    };
    function<void(Point,Point)> traceHull = [&](Point a, Point b) {
      if (a == b) return;
      auto [_, c] = tryAngle((b-a).ortho());
      if (a < c) { traceHull(a, c); traceHull(c, b); }
    };
    auto [left, right] = tryAngle({1, 0});
    traceHull(left, right);
    traceHull(right, left);

    printf("%lld\n", ret);
  }
}
```

Problem L

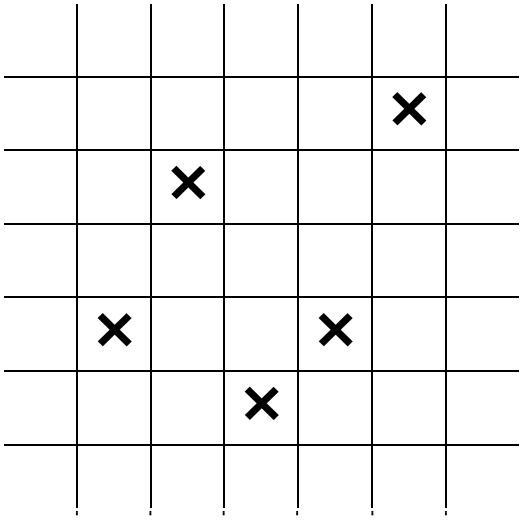
Where Am I?

Time limit: 2 seconds

Who am I? What am I? Why am I? These are all difficult questions that have kept philosophers reliably busy over the past millennia. But when it comes to “Where am I?”, then, well, modern smartphones and GPS satellites have pretty much taken the excitement out of that question.

To add insult to the injury of newly unemployed spatial philosophers formerly pondering the “where” question, the Instant Cartographic Positioning Company (ICPC) has decided to run a demonstration of just how much more powerful a GPS is compared to old-fashioned maps. Their argument is that maps are useful only if you already know where you are, but much less so if you start at an unknown location.

For this demonstration, the ICPC has created a test area that is arranged as an unbounded Cartesian grid. Most grid cells are empty, but a finite number of cells have a marker at their center (see Figure L.1(a) for an example with five marked cells). All empty grid cells look the same, and all cells with markers look the same. Suppose you are given a map of the test area (that is, the positions of all the markers), and you are placed at an (unknown to you) grid cell. How long will it take you to find out where you actually are? ICPC's answer is clear: potentially a very, very long time, while a GPS would give you the answer instantly. But how long exactly?



(a) Grid with markers (✕) corresponding to the first sample input.

...	9	10	11	12	
23	8	1	2	13	
22	7	0	3	14	
21	6	5	4	15	
20	19	18	17	16	

(b) The order in which a test subject visits grid cells, relative to starting cell 0.

Figure L.1: Sample grid and the order in which test subjects explore the grid.

In the trial, test subjects will explore their environment by following an expanding clockwise spiral whose first few steps are shown in Figure L.1(b). The starting cell is labeled "0", and the numbers show the order in which other cells are visited. The test subjects can see a marker only if they are at its grid cell, and they will stop their exploration as soon as they know where they are based on the grid cells that they have seen so far. That means that the observed sequence of empty and marked grid cells could have begun only at a single possible starting position. The grid is unbounded, but the exploration will be finite since once a test subject has seen all markers on the grid, they'll definitely know where they are.

Having hundreds of test subjects literally running in circles can be expensive, so the ICPC figures that writing a simulation will be cheaper and faster. Maybe you can help?

Input

The input describes a single grid. It starts with a line containing two integers d_x, d_y ($1 \le d_x, d_y \le 100$). The following d_y lines contain d_x characters each, and describe part of the test grid. The i^{th} character of the j^{th} line of this grid description specifies the contents of the grid cell at coordinate $(i, d_y - j + 1)$. The character is either '`.`' or '`X`', meaning that the cell is either empty, or contains a marker, respectively.

The total number of markers in the grid will be between 1 and 100, inclusive. All grid cells outside the range described by the input are empty.

Output

In ICPC's experiment, a test subject knows they will start at some position (x, y) with $1 \le x \le d_x$, $1 \le y \le d_y$.

Output three lines. The first line should have the expected number of steps needed to identify the starting position, assuming that the starting position is chosen uniformly at random. This number needs to be exact to within an absolute error of 10^{-3}.

The second line should have the maximum number of steps necessary until one can identify the starting position.

The third line should list all starting coordinates (x, y) that require that maximum number of steps. The coordinates should be sorted by increasing y-coordinates, and then (if the y-coordinates are the same) by increasing x-coordinates.

Sample Input 1

```
5 5
....X
.X...
.....
X..X.
..X..
```

Sample Output 1

```
9.960
18
(1,4) (4,5)
```

Sample Input 2

```
5 1
..XX.
```

Sample Output 2

```
4.600
7
(1,1) (5,1)
```

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

int main() {
  int X, Y;
  while (cin >> X >> Y) {
    vector<string> g(Y);
    for (int y = Y-1; y >= 0; y--) cin >> g[y];
```

```
    vector<vector<int>> dist(201, vector<int>(201));
    for (int x = 100, y = 100, dx = 0, dy = 1, step = 0, stepn = 1, cur = 0; y <
201; cur++) {
      dist[y][x] = cur;
      x += dx; y += dy;
      if (++step == stepn) {
        swap(dx, dy);
        dy = -dy;
        step = 0;
        if (dy) stepn++;
      }
    }

    vector<vector<int>> obs(40001);
    for (int y = 0; y < Y; y++)
    for (int x = 0; x < X; x++) if (g[y][x] == 'X')
    for (int sy = 0, i = 0; sy < Y; sy++)
    for (int sx = 0; sx < X; sx++, i++) {
      obs[dist[y-sy+100][x-sx+100]].push_back(i);
    }

    vector<int> comp(X*Y), compt(X*Y), compsz{X*Y};
    for (int t = 0; compsz.size() < X*Y; t++) if (obs[t].size()) {
      vector<int>& v = obs[t];
      sort(v.begin(), v.end(), [&](int x, int y) { return comp[x] < comp[y]; });
      for (int i = 0, j = 0; i < v.size(); i = j) {
        for (j++; j < v.size() && comp[v[j]] == comp[v[i]]; j++)
          ;
        int& sz = compsz[comp[v[i]]];
        if (j-i == sz) continue;
        if (j-i == 1) compt[compsz.size()] = t;
        sz -= j-i;
        if (sz == 1) compt[comp[v[i]]] = t;
        for (int k = i; k < j; k++) comp[v[k]] = compsz.size();
        compsz.push_back(j-i);
      }
    }

    int mx = 0, tot = 0;
    for (int i = 0; i < X*Y; i++) {
      mx = max(mx, compt[i]);
      tot += compt[i];
    }

    cout << fixed << setprecision(9) << double(tot)/X/Y << endl;
    cout << mx << endl;
    bool first = true;
    for (int i = 0; i < X*Y; i++) if (compt[comp[i]] == mx) {
      if (!first) cout << ' ';
```

```
      first = false;
      cout << '(' << i%X+1 << ',' << i/X+1 << ')';
    }
    cout << endl;
  }
}
```