University of Virginia ICPC Notebook

Contents

1 World Final 2004

1.1 A. Carl the Ant

```
#include <stdio.h>
#include <string.h>

// information about an ant
struct info {
    // coordinates: (x, y)
    // length traveled so far: len
    // time waited at an intersection: wait
    // direction it is facing: dir
    int x, y, len, wait, dir;
};

// max 50 line segments, 100 ants, 250 line segment length
const int MaxN = 50, MaxM = 100, MaxL = 250;

// unit vectors for adding coordinates easily
const int Add[4][2] = {-1, 0, 0, 1, 1, 0, 0, -1};

// the i th unit vector has direction route[i]
int route[MaxN * MaxL];

// the latest direction of coordinate (x,y) is map[x][y]
int map[MaxL+1][MaxL+1];

// ants[x][y][k] means there's an ant at (x,y) facing direction k
int ants[MaxL+1][MaxL+1][4];

// global variables
int N, cases, n, m, D, Rs, fin, sta, T, ex, ey;

// the ant at index list[i] is the first ith ant that completes
int list[MaxM];

// the array of ants
info ant[MaxM];


// gets the input for current test case and calculates the route that
// ant Carl has passed through
void init() {
    int i, j, k, t, x1, y1, x2, y2;
    scanf("%d %d %d", &n, &m, &D);

    x1 = y1 = 0; Rs = 0;
    // calculate the route Carl has passed through
    for (i = 0; i < n; i++) {
        scanf("%d %d", &x2, &y2);

        // determine the direction of the line segment
        if (x1 > x2) t = 0;
        else if (y1 < y2) t = 1;
        else if (x1 < x2) t = 2;
        else t = 3;

        // set the values for "route" in the current line segment
        while (x1 != x2 || y1 != y2) {
            route[Rs] = t; Rs++;
            x1 += Add[t][0]; y1 += Add[t][1];
        }
    }

    // no ants have finished, and no ants have started
    fin = 0; sta = 0;
    // initialize the array of ants
    for (i = 0; i <= MaxL; i++) for (j = 0; j <= MaxL; j++) {
        for (k = 0; k < 4; k++) ants[i][j][k] = -1;
    }
}


// simulate one time step of ant movement
void work() {
    int ok, i, j, x, y, d, p, tmp, m1, m2, m3;
    // go[i] means whether ant i can possibly move (0) or not (1)
    int go[MaxM]; memset(go, 0, sizeof(go));

    // how many ants have finished so far
    ok = 0;
    for (i = 0; i < sta; i++) if (ant[i].x < 0) ok++;

    // whether an ant can move based on its priority
    for (i = 0; i < sta; i++) if (ant[i].x >= 0) {
        x = ant[i].x; y = ant[i].y;
        for (j = 0; j < 4; j++) {
            if (ants[x][y][j] >= 0) {
                tmp = ants[x][y][j];
                // wait longer, or same wait but travels longer
                m1 = (ant[tmp].wait > ant[i].wait);
                m2 = (ant[tmp].wait == ant[i].wait);
                m3 = (ant[tmp].len > ant[i].len);

                // if ant "tmp" has higher priority than ant "i",
                // then ant "i" cannot move
                if (m1 || (m2 && m3)) {
                    p = 1; go[i] = 1; ok++;
                    break;
                }
            }
        }
    }

    // if there's an ant that blocks the way, some more ants might
    // cannot move, handle them in this part
    do {
        ok = 0;
        for (i = 0; i < sta; i++) {
            if (!go[i] && ant[i].x >= 0) {
                // the movement direction of ant "i"
                d = map[ant[i].x][ant[i].y];

                // the new coordinate of ant "i"
                x = ant[i].x + Add[d][0];
                y = ant[i].y + Add[d][1];

                // if there's some ant that blocks the way, then
                // ant "i" cannot move, should be updated
                if (ants[x][y][d] >= 0 && go[ants[x][y][d]] == 1) {
                    go[i] = 1; ok = 1;
                    continue;
                }
            }
        }
    } while (ok);

    for (i = 0; i < sta; i++) {
        if (!go[i] && ant[i].x >= 0) {
            // update the ant that can actually be moved
            ant[i].len++; ant[i].wait = 0;
            ants[ant[i].x][ant[i].y][ant[i].dir] = -1;
        } else if (ant[i].x >= 0) {
            // or otherwise the wait time should be incremented
            ant[i].wait++;
        }

        for (i = 0; i < sta; i++) {
            if (!go[i] && ant[i].x >= 0) {
                x = ant[i].x; y = ant[i].y;
                // if the start point is clear, add a new ant
                if (x == 100 && y == 100 && i != sta-1) {
                    ants[100][100][map[100][100]] = i+1;
                }
                // update the current ant's info
                d = map[x][y];
                ant[i].x += Add[d][0];
                ant[i].y += Add[d][1];
                ant[i].dir = d;
                // if some ant has reached the dest, remove it
                if (ant[i].x == ex && ant[i].y == ey) {
                    list[fin] = i; fin++;
                    ant[i].x = -1;
                } else {
                    ants[ant[i].x][ant[i].y][d] = i;
                }
            }
        }
    }
}


// output the solution of the current test case
void write() {
    int i;
    printf("Case %d:\n", cases);
    printf("Carl finished the path at time %d\n", ant[0].len+1);
    printf("The ants finish in the following order:\n");
    printf("%d", list[0]);
    for (i = 1; i < m; i++) printf(" %d", list[i]);
    printf("\n");
    printf("The last ant finished the path at time %d\n", T+1);
    if (cases < N) printf("\n");
}


int main() {
    int X, Y;
    scanf("%d", &N);

    for (cases = 1; cases <= N; cases++) {
        init();
        // array indices are made positive by adding 100
        X = 100; Y = 100; ex = -1; ey = -1;

        for (T = 0; fin < m; T++) {
            // update map indices based on Carl's route
            if (T < Rs) {
                map[X][Y] = route[T];
                X += Add[route[T]][0]; Y += Add[route[T]][1];
                if (T == Rs - 1) {
                    ex = X; ey = Y;
                }
            }
            // if there's some ant that has just started
            if (T % D == 0 && sta < m) {
                if (ants[100][100][map[100][100]] < 0)
                    ants[100][100][map[100][100]] = sta;
                ant[sta].x = 100; ant[sta].y = 100;
                ant[sta].len = 0; ant[sta].wait = 0;
                ant[sta].dir = map[100][100];
                sta++;
            }

            work();
        }

        write();
    }

    return 0;
}
```

1.2 B. Heliport

```
#include <stdio.h>
#include <math.h>

// required precision of radius is 1 * 10^(-6)
const double eps = 1e-6;

// the radius has a range ra <= r <= rb
double ra, rb, r;
int x[25], y[25], n, i, len, px, py, cases;

// the direction of an edge is representable by 1-byte char
char dr;


// check whether the heliport with center (ox, oy) and radius r
// satisfies all the requirements
bool check(double ox, double oy) {
    // a ray from (ox, oy) intersects the polygon s times
    int i, s; s = 0;

    // if the ray intersects the polygon even number of times,
    // then (ox, oy) is outside the polygon; not applicable
    for (i = 0; i < n; i++)
        if (x[i] > ox && ((y[i] > oy) ^ (y[i+1] > oy))) s++;
    if (s % 2 == 0) return false;

    // check each edge/corner if it's inside the heliport
    for (i = 0; i < n; i++) {
        // a corner is inside the circle
        if ((x[i]-ox) * (x[i]-ox) + (y[i]-oy) * (y[i]-oy)
            < (r-eps) * (r-eps)) return false;

        // a horizontal edge is inside the circle
        if (x[i] == x[i+1] && ((y[i] > oy) ^ (y[i+1] > oy))
            && fabs(x[i]-ox) < r-eps) return false;

        // a vertical edge is inside the circle
        if (y[i] == y[i+1] && ((x[i] > ox) ^ (x[i+1] > ox))
            && fabs(y[i]-oy) < r-eps) return false;
    }
    // otherwise the radius satisfies our requirements
    return true;
}


// check whether one can build heliport somewhere w/ radius r
// three possible cases, as indicated by three for loops
bool ok() {
    int i, j; double di, dd, mx, my, dx, dy;

    // heliport is beside two perpendicular edges
    for (i = 0; i < n; i++) {
        if (x[i] == x[i+1]) {
            for (j = 0; j < n; j++) {
                if (y[j] == y[j+1]) {
                    // two edges found, check the points with
                    // distances r from both edges
                    if (check(x[i]+r, y[j]+r)) return true;
                    if (check(x[i]+r, y[j]-r)) return true;
                    if (check(x[i]-r, y[j]+r)) return true;
                    if (check(x[i]-r, y[j]-r)) return true;
                }
            }
        }
    }

    // heliport is beside one edge and intersects one corner
    for (i = 0; i < n; i++) for (j = 0; j < n; j++) {
        if (x[i] == x[i+1]) {
            // center is on the right of a vertical edge
            di = fabs(x[j] - (x[i]+r));
            if (di < r) {
                dd = sqrt(r * r - di * di);
                if (check(x[i]+r, y[j]+dd)) return true;
                if (check(x[i]+r, y[j]-dd)) return true;
            }

            // center is on the left of a vertical edge
            di = fabs(x[j] - (x[i]-r));
            if (di < r) {
                dd = sqrt(r * r - di * di);
                if (check(x[i]-r, y[j]+dd)) return true;
                if (check(x[i]-r, y[j]-dd)) return true;
            }
        } else {
            // center is above a horizontal edge
            di = fabs(y[j] - (y[i]+r));
            if (di < r) {
                dd = sqrt(r * r - di * di);
                if (check(x[j]+dd, y[i]+r)) return true;
                if (check(x[j]-dd, y[i]+r)) return true;
            }

            // center is below a horizontal edge
            di = fabs(y[j] - (y[i]-r));
            if (di < r) {
                dd = sqrt(r * r - di * di);
                if (check(x[j]+dd, y[i]-r)) return true;
                if (check(x[j]-dd, y[i]-r)) return true;
            }
        }
    }

    // heliport is intersecting two corners
    for (i = 0; i < n-1; i++) for (j = i+1; j < n; j++) {
        mx = (x[i] + x[j]) / 2.0;
        my = (y[i] + y[j]) / 2.0;
        di = sqrt((x[i]-mx)*(x[i]-mx) + (y[i]-my)*(y[i]-my));
        if (di > 0 && di < r) {
            // some point with distance r from both corners
            dd = sqrt(r * r - di * di);
            dx = (my-y[i]) / di * dd;
            dy = (x[i]-mx) / di * dd;
            if (check(mx+dx, my+dy)) return true;
            if (check(mx-dx, my-dy)) return true;
        }
    }

    // no possible case; this radius r is not applicable
    return false;
}


int main() {
    while (scanf("%d", &n) && n) {
        if (cases) printf("\n");
        // current coordinates (px, py) while reading input
        px = 0; py = 0;
        for (i = 1; i <= n; i++) {
            scanf("%d %c", &len, &dr);

            if (dr == 'R') px += len;
            else if (dr == 'L') px -= len;
            else if (dr == 'U') py += len;
            else py -= len;

            x[i] = px; y[i] = py;
        }

        // radius is between 0 and 999 by constraints
        ra = 0; rb = 999;
        // binary search for the maximum applicable radius
        while (rb - ra > eps) {
            r = (ra + rb) / 2;
            if (ok()) ra = r;
            else rb = r;
        }
        printf("Case Number %d radius is: %.2lf\n",
            ++cases, r);
    }
    return 0;
}
```

1.3 C. Image is Everything

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;

/*    to fold
5|          front, left, back, right, top, bottom
2|1|4|3|      1     2     3     4      5     6
     |6|
*/

const int MaxN=11; // at most size of 10, but use 11 to avoid seeing the additional side
char f[6][MaxN][MaxN]; // color view read from data: [6]each view [][] square matrix
int p[6][MaxN][MaxN];  // number of cubic(s) removed: [6]each view [][] square matrix
int g[MaxN][MaxN][MaxN]; // 3D matrix: 0 as not exist, -1 no info yet, char A-Z (1-27) as color
int n; //cubic size
```

```
bool update(int m) {
   bool upd=false;
   for(int i=0; i<n; ++i) for(int j=0; j<n; ++j){ //analyze each pixel in current view
      int &t=p[m][i][j]; //the number of cubic removed at this pixel
      char ch=f[m][i][j]; //the color at this pixel in the original view
      if(t==n) continue; //already removed all cubics at this pixel
      int *val; //value of actual cubic at this pixel to analyze
      switch (m) {
          case 0: // front
            val = &(g[n-t-1][j][i]);
            break;
          case 1: // left
            val = &(g[j][t][i]);
            break;
          case 2: // back
            val = &(g[t][n-j-1][i]);
            break;
          case 3: // right
            val = &(g[n-j-1][n-t-1][i]);
            break;
          case 4: // top
            val = &(g[i][j][t]);
            break;
          case 5: // bottom
            val = &(g[n-i-1][j][n-t-1]);
            break;
          default:
            break;
      }
      if (*val==0){ ++t; upd=true;}
        //the cubic here does not exist
      else if(ch=='.'){ *val=0; ++t; upd=true; }
        //update the cubic here as not exist
      else if(*val==-1){ *val=ch-'A'+1; upd=true;}
        //if the cubic color is not analzed, use the color on the other side
      else if(*val!=ch-'A'+1){ *val=0; ++t; upd=true; }
        //if different from the cubic color from the other side, update the cubic here as not exist,
   }
   return upd;
}

int main() {
  while(true) {
    scanf("%d", &n); // read size of cubic
    if (n==0) break;
    for (int i=0; i<n; ++i) scanf(" %s %s %s %s %s %s", f[0][i], f[1][i], f[2][i], f[3][i], f[4][i], f
          [5][i]); // read view
    for (int i=0; i<6; ++i) fill(p[i][0], p[i][n], 0); //initize all view as non removed
    for (int i=0; i<n; ++i) fill(g[i][0], g[i][n], -1); //initize all cubics not exist
    // fill(g[0][0], g[n][0], -1);
    while (true) {
       bool quit=true;
       for (int i=0; i<6; i++) if(update(i)) quit=false;
       if(quit) break;
    }
    int ans=n*n*n; // init weight for perfect cubic
    for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) for(int k=0; k<n; ++k)
      if(g[i][j][k]==0) --ans; // if the cubic at position[i][j][k] does not exit, remove this weight
    printf("Maximum weight: %d gram(s)\n", ans);
  }
  return 0;
}
```

1.4 D. Insecure in Prague

```
#include <stdio.h>
#include <memory.h>
#include <string.h>

// a is the input character, count is the frequency of characters,
// current is the letters in the first entry, mark is the position
// marker, remain is the letters not covered in the first entry,
// ans is the best answer
char a[41], count[26], current[21], mark[41], remain[81], ans[41];

// pos[len][j][k] is the resulting character of starting from the
// first character, go over a length "len" string "k" times with a
// step of "j"
char pos[41][41][41];

int i, j, k, n;
int go, len, step, skips, maxl, remains, start, unique, cases, ok;


// pre-process "pos[][][]" using a dynamic programming approach so
// that some repeat calculations can be avoided later
void preprocess() {
    // iterate over a string of length "len", with step "step"
    for (len = 1; len < 41; len++) {
        for (step = 1; step < 41; step++) {
            // no position has been reached at the beginning
            memset(mark, 0, sizeof(mark));
            // initialize the first position in the string
            go = 0; mark[0] = 1; pos[len][step][0] = 0;

            // iterate over the string "len-1" times
            for (i = 1; i < len; i++) {
                // how many positions have been skipped
                skips = (step-1) % (len-i) + 1;

                // if the current position is not marked, then we
                // record this visit and decrement "skips"
                while (skips) {
                    go++;
                    if (go >= len) go = 0;
                    if (!mark[go]) skips--;
                }
                mark[go] = 1;

                // record the position after stepping "i" times
                pos[len][step][i] = go;
            }
        }
    }
}


int main() {
    preprocess();

    while (true) {
        scanf("%s", &a);
        n = strlen(a); if (n < 2) break;

        // the longest solution's length is initialized to 0
        maxl = 0;
        // whether solution of length "maxl" is unique
        unique = 0;
        memset(ans, 0, sizeof(ans));
        memset(count, 0, sizeof(count));

        // calculate the frequency of each character and store them
        for (i = 0; i < n; i++) count[a[i]-'A']++;

        // i is the starting position of the first pass
        // this character has to appear more than 1 time
        for (i = 0; i < n; i++) if (count[a[i]-'A'] > 1) {
            // j is the step length of the first pass
            for (j = 1; j < n; j++) {
                memset(current, 0, sizeof(current));
                memset(mark, 0, sizeof(mark));
                // the current position is i
                go = i;

                // len is the length of a possible solution
                for (len = 0; len < n/2; len++) {
                    // record character we've just passed through
                    current[len] = a[go]; mark[go] = 1;

                    // if we obtain a length greater than maxl, and
                    // the solution is a different one
                    if (len >= maxl && strcmp(current, ans)) {
                        remains = 0; ok = 0;
                        // record and count the remaining characters
                        for (k = 0; k < n; k++) if (!mark[k])
                            remain[remains++] = a[k];

                        // to make the problem simpler, copy all the
                        // remaining characters to end of string
                        for (k = 0; k < remains; k++)
                            remain[remains+k] = remain[k];

                        // start is the position of the second pass
                        for (start = 0; start < remains; start++)
                        if (remain[start] == current[0]) {
                            // step is step length of second pass
                            for (step = j+1; step <= n; step++) {
                                // check next characters one by one
                                for (k = 1; k <= len; k++) {
                                if (remain[pos[remains][step][k]
                                    + start] != current[k]) break;
                                }
                                // if all characters match, then we
                                // compare with recorded solution
                                if (k > len) {
                                    ok = 1;
                                    if (len > maxl) {
                                        // longer solution found,
                                        // update length/uniqueness
                                        maxl = len;
                                        unique = 1;
                                        memcpy(ans, current,
                                            sizeof(current));
                                    } else if (unique) {
                                        // same length, not unique
                                        maxl++;
                                        unique = 0;
                                    } else {
                                        // current length does not
                                        // have a solution
                                        unique = 1;
                                        memcpy(ans, current,
                                            sizeof(current));
                                    }
                                    break;
                                }
                            }
                            // if obtain some solution, then break
                            if (ok) break;
                        }
                    }
                    // step over the next j characters
                    skips = j;
                    while (skips) {
                        go++;
                        if (go >= n) go = 0;
                        if (!mark[go]) skips--;
                    }
                }
            }
        }
        if (unique) printf("Code %d: %s\n", ++cases, ans);
        else printf("Code %d: Codeword not unique\n", ++cases);
    }

    return 0;
}
```

1.5 E. Intersecting Dates

```
#include <stdio.h>
#include <vector>
#include <stdlib.h>
#include <unordered_map>
using namespace std;

// there are at most 147000 different days between 1700 and 2100
#define MAX_DATE 147000

// keep track of the case number (needed for output)
static int caseNum = 1;

// records which dates are valid dates for our output
bool dates[MAX_DATE];

// "itod" (Index to Date): given a specific index for "dates",
// returns the associated calendar date for this index
int itod[MAX_DATE];

// "dtoi" (Date to Index): given a date in calendar,
// returns the index it is represented in our "dates" array
// **This is exactly the reverse of "itod";
// it has to be stored as unordered_map,
// since 21001231 exceeds the max memory size for an array
unordered_map<int, int> dtoi;

// given an integer between 17000101 and 21001231,
// returns whether this integer is a valid date in calendar
bool isCorrectDate(int date) {
    int y = date / 10000, m = (date%10000) / 100, d = date % 100;
    if (m == 0 || m > 12) return false;
    if (d == 0 || d > 31) return false;
    if (d==31 && (m==4 || m==6 || m==9 || m==11)) return false;
    if (m == 2) {
        if (y%4 == 0 && (y%100 != 0 || y%400 == 0)) return d <= 29;
        else return d <= 28;
    }
    return true;
}

// pre-generate "itod" and "dtoi" for all possible dates
void genMaps() {
    int ind = 0;
    for (int i = 17000101; i <= 21001231; i++) {
        if (!isCorrectDate(i)) continue;
        itod[ind++] = i;
        dtoi[i] = ind-1;
    }
}

// given two dates "start" and "end" (possibly the same),
// prints the output in the correct format to stdout
void printOutput(int start, int end) {
    int y1 = start / 10000, m1 = (start%10000) / 100, d1 = start%100;
    int y2 = end / 10000, m2 = (end%10000) / 100, d2 = end%100;
    if (start == end) // required to print only one date
        printf("    %d/%d/%d\n", m1, d1, y1);
    else
        printf("    %d/%d/%d to %d/%d/%d\n", m1, d1, y1, m2, d2, y2);
}

// main part of the solution
void solve(int b, int a) {
    printf("Case %d:\n", caseNum);

    // reads all dates whose data have been previously retrieved
    // and store them in a vector (we need these data later)
    vector<int> dates1, dates2;
    for (int i = 0; i < b; i++) {
        int d1, d2; scanf("%d %d\n", &d1, &d2);
        dates1.push_back(d1);
        dates2.push_back(d2);
    }

    // reads all remaining dates whose data are to be retrieved
    for (int i = 0; i < a; i++) {
        int d1, d2; scanf("%d %d\n", &d1, &d2);
        // these dates are required for output; set them to true
        fill(dates+dtoi[d1], dates+dtoi[d2]+1, true);
    }

    for (int i = 0; i < b; i++) {
        // these dates have been previously retrieved (no longer needed)
        fill(dates+dtoi[dates1[i]], dates+dtoi[dates2[i]]+1, false);
    }

    // iterate over all possible dates to get desired time intervals
    int i = 0;
    bool need = false;
    while (i <= dtoi[21001231]) {
        if (dates[i]) {
            int start = itod[i];
            while (dates[++i]); // scan for the end of this interval
            int end = itod[i-1];
            printOutput(start, end);

            // the "no quotes" message should no longer be displayed
            need = true;
        } else
            i++;
    }

    // no dates are required, print the corresponding message
    if (!need) printf("    No additional quotes are required.\n");
}

int main() {
    genMaps();

    bool need = false;
    for (;;) {
        // the array "dates" need to be initialized as false everywhere
        fill(dates, dates+MAX_DATE, false);

        // read input
        int b, a; scanf("%d %d\n", &b, &a);

        // check if we reach the end of test file
        if (b == 0 && a == 0) break;

        // keep track of line feeds between cases
        if (!need) need = true;
        else printf("\n");

        solve(b, a);
        caseNum++;
    }

    return 0;
}
```

1.6 F. Merging Maps

```
#include <stdio.h>
#include <string.h>
#include <algorithm>

using std::max; using std::min;

// information about a map: its numbers of rows and columns, and its features
struct MAPS {
    int n, m;
    char feature[100][100];
};

// info about merging two maps, including its scores and the offset in the
// horizontal and vertical directions
struct Merge {
    int score, r, c;
};

// the current row of a particular map
char s[2000];

// there are n maps in total, and the highest merging score is denoted by max
int cases, n, i, j, k, sum, maxx, x, y;

// an array of all maps (original)
MAPS map[20];

// merge[x][y] is the score of merging maps x and y
Merge merge[20][20];

// the remaining signs of those maps
bool sign[20];


// calculate the merging score of two characters, 0 for no feature, 1 for any
// matching feature, and -10000 if they do not match
int calc(char a, char b) {
    if (a == ’-’ || b == ’-’) return 0;
    if (a == b) return 1;
    return -10000;
}


// try to merge maps x and y, and calculate the info denoted by struct Merge
Merge Try(int x, int y) {
    int r, c, i, j, k, sc, x1, y1, x2, y2;
    Merge tmp;

    // initialize the score to be 0
    tmp.score = 0;
    for (r = -map[y].n+1; r < map[x].n; r++) {
        for (c = -map[y].m+1; c < map[x].m; c++) {
            // calculate the vertices of merging maps x and y
            x1 = max(0, r); y1 = max(0, c);
            x2 = min(map[x].n, map[y].n+r); y2 = min(map[x].m, map[y].m+c);

            // calculate the score of merging
            sc = 0;
            for (i = x1; i < x2; i++) for (j = y1; j < y2; j++)
                sc += calc(map[x].feature[i][j], map[y].feature[i-r][j-c]);

            // update the optimal score of merging
            if (sc > tmp.score) {
                tmp.score = sc; tmp.r = r; tmp.c = c;
            }
        }
    }
    return tmp;
}


// merge maps indicated by indices x and y into a new map with index z
void add_map(int x, int y, int z) {
    int i, j, k, Dx, Dy;
    // calculate the offset of map x in the process of merging
    Dx = max(-merge[x][y].r, 0); Dy = max(-merge[x][y].c, 0);
    // calculate the dimensions of map z
    map[z].n = max(map[x].n, map[y].n+merge[x][y].r) + Dx;
    map[z].m = max(map[x].m, map[y].m+merge[x][y].c) + Dy;

    // fill the feature map of z to have no features
    for (i = 0; i < map[z].n; i++) for (j = 0; j < map[z].m; j++)
        map[z].feature[i][j] = ’-’;

    // fill the region of the feature map with features from map x
    for (i = 0; i < map[x].n; i++) for (j = 0; j < map[x].m; j++)
        map[z].feature[i+Dx][j+Dy] = map[x].feature[i][j];

    // update the indices to reflect the offset
    Dx += merge[x][y].r; Dy += merge[x][y].c;

    // fill the region of the feature map with features from map y
    for (i = 0; i < map[y].n; i++) for (j = 0; j < map[y].m; j++)
```

```
        if (map[y].feature[i][j] != '-')
            map[z].feature[i+Dx][j+Dy] = map[y].feature[i][j];
}


int main() {
    for (cases = 1; ; cases++) {
        scanf("%d", &n); if (n == 0) break;
        // read input about all original maps
        for (i = 1; i <= n; i++) {
            scanf("%d %d ", &map[i].n, &map[i].m);
            for (j = 0; j < map[i].n; j++) {
                scanf("%s", &s);
                for (k = 0; k < map[i].m; k++) map[i].feature[j][k] = s[k];
            }
        }

        // calculate merging info about any two given maps
        for (i = 1; i <= n; i++) for (j = i+1; j <= n; j++)
            merge[i][j] = Try(i, j);
        // initialize all maps to be usable in the sign[] array
        memset(sign, 1, sizeof(sign));

        while (true) {
            maxx = 0;
            // find two maps with the highest merging score among all maps
            for (i = 1; i <= n; i++) if (sign[i])
            for (j = 1; j <= n; j++) if (sign[j]) {
                if (merge[i][j].score > maxx) {
                    maxx = merge[i][j].score;
                    x = i; y = j;
                }
            }
            // if the max merging score is 0, then process completed, exit
            if (maxx == 0) break;

            // maps that have already been merged are not usable any more
            sign[x] = 0; sign[y] = 0;
            add_map(x, y, ++n);
            // calculate the new map's merging score against all others
            for (i = 1; i < n; i++) if (sign[i])
                merge[i][n] = Try(i, n);
        }

        // output resulting map according to format
        if (cases > 1) printf("\n");
        printf("Case %d:\n", cases);
        x = 0;
        for (i = 1; i <= n; i++) if (sign[i]) {
            if (x) printf("\n"); x++;
            printf("    MAP %d:\n", i);

            printf("    +");
            for (j = 0; j < map[i].m; j++) printf("-");
            printf("+\n");

            for (j = 0; j < map[i].n; j++) {
                printf("    |");
                for (k = 0; k < map[i].m; k++)
                    printf("%c", map[i].feature[j][k]);
                printf("|\n");
            }

            printf("    +");
            for (j = 0; j < map[i].m; j++) printf("-");
            printf("+\n");
        }
    }

    return 0;
}
```

1.7 G. Navigation

```
#include <stdio.h>
#include <math.h>

// (ox[], oy[]) are coordinates of the signal's source, r[] is the transmit
// distance of the signal, (dx, dy) are differences of the circle center's
// x and y coordinates, (px, py) are the cartesian coordinates of the signal
double t, x, y, ox[12], oy[12], r[12], px, py, dx, dy;

// other global variables necessary for this program
double dr, degree, ti, pi, dis, lx, ly, xa, ya, xb, yb;
int n, i, cases, c1, c2;
```

```
// check if point (x, y) is close enough to all the circles
int check(double x, double y) {
    int i; double dx, dy;
    // check the distance of the point to every circle
    for(i = 0; i < n; i++) {
        // compute the distance "dr" between (x, y) and circle "i"
        dx = x - ox[i]; dy = y - oy[i];
        dr = sqrt(dx*dx + dy*dy) - r[i];
        if (fabs(dr) > 0.1) return 0;
    }
    return 1;
}


int main() {
    // first of all, the accurate value of pi is given by arccos(-1)
    pi = acos(-1.0);

    while (scanf("%d%lf%lf%lf", &n, &t, &x, &y) && n) {
        // read input about the n signals
        for (i = 0; i < n; i++) {
            scanf("%lf%lf%lf%lf", &px, &py, &degree, &ti);
            // these are relevant info about signal i
            degree = (90-degree) / 180 * pi;
            dis = 100 * ti;
            ox[i] = px + dis * cos(degree);
            oy[i] = py + dis * sin(degree);
            r[i] = 350 * (t-ti);
        }

        printf("Trial %d: ", ++cases);
        // based on first circle, check if the later circles overlap with it
        for (i = 1; i < n; i++) {
            // calculate the differences of the center's coordinates/radius
            dx = ox[i] - ox[0];
            dy = oy[i] - oy[0];
            dr = r[i] - r[0];
            // if the sum of these three squares are positive, then, the two
            // circles do not overlap, so we can select the current one
            if (dx*dx + dy*dy + dr*dr > 0.01) break;
        }

        // if all the circles overlap, then we have multiple solutions
        if (i >= n)      {
            printf("Inconclusive\n"); continue;
        }
        // calculate the distance between two circles' center
        dis = sqrt(dx*dx + dy*dy);
        // two circles have too much difference in radius, no solution
        if (dis < 0.1) {
            printf("Inconsistent\n"); continue;
        }
        // if "lx" too large, then two circles do not intersect, no solution
        lx = (dis*dis + r[0]*r[0] - r[i]*r[i]) / dis / 2;
        if (fabs(lx) > r[0] + 0.1) {
            printf("Inconsistent\n"); continue;
        }

        // if lx is greater than the radius, then change it to the radius
        // lx could also be negative, we should change it as well
        if (lx > r[0]) lx = r[0];
        if (lx < -r[0]) lx = -r[0];
        // calculate the ly value according to the given figure
        ly = sqrt(r[0]*r[0] - lx*lx);
        // normalize the distance differences dx and dy
        dx /= dis; dy /= dis;

        // calculate the (x, y) coordinates of the two intersecting points
        xa = ox[0] + dx*lx - dy*ly;
        ya = oy[0] + dy*lx + dx*ly;
        xb = ox[0] + dx*lx + dy*ly;
        yb = oy[0] + dy*lx - dx*ly;
        // if the two points are too close to each other, delete one of them
        if (sqrt((xa-xb) * (xa-xb) + (ya-yb) * (ya-yb)) < 0.1) {
            xb = 1e9; yb = 1e9;
        }

        // check if the two points satisfy the problem's requirements
        c1 = check(xa,ya); c2 = check(xb,yb);
        if (c1 + c2 == 1) {
            // if exactly one of the two points satisfies requirements, cast
            // c2 to c1 so that we can deal with them consistently
            if (c2) {
                xa = xb; ya = yb;
            }
            // calculate the distance between this point and the destination
            dx = x - xa; dy = y - ya; dis = sqrt(dx*dx + dy*dy);

            if (dis < 0.1) {
                printf("Arrived\n");
            } else {
                // calculate direction required to transmit to destination
                if (dy > 0) degree = acos(dx / dis);
                else degree = pi*2 - acos(dx / dis);
                // transform the degree to the correct units and format
                degree = 90 - degree / pi * 180;
                if (degree < 0) degree += 360;
                if (degree > 360) degree -= 360;
                printf("%.0lf degrees\n", degree);
            }
        } else if (c1) {
            // if both of the two points satisfy the requirements
            printf("Inconclusive\n");
        } else {
            // if none of the points satisfy the requirements, no solution
            printf("Inconsistent\n");
        }
    }

    return 0;
}
```

1.8 H. Tree-lined Street

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <algorithm>
using namespace std;
struct line {                 //line structure with two points (x1,y1) (x2,y2)
  int x1,y1,x2,y2;
};

int cases,n,ans,s,i,j,k;    //global variables
// cases : case number    n : number of lines   ans : output
// s : number of intersections on one certain line    i, j, k: temporary variables

line a[110];                 //record lines
double b[200];               //record each intersection on one line by its distance to the origin
double len;
bool check(int x1,int y1,int x2,int y2,int x3,int y3) {  //check if p(x1,y1)(x2,y2) is at the clock-
      wise side of p(x1,y1)(x3,y3)                                       //used for
      determining if two lines intersect
   long long t1,t2;
   t1=(long long)x1*(long long)y2-(long long)x2*(long long)y1;
   t2=(long long)x1*(long long)y3-(long long)x3*(long long)y1;
   return((t1>=0&&t2<=0)||(t1<=0&&t2>=0));
}

bool intersect(line A, line B) {                          //check if two points are on the different
      side of one line and vice versa, if both so they intersect
  return (check(A.x2-A.x1,A.y2-A.y1,B.x1-A.x1,B.y1-A.y1, B.x2-A.x1,B.y2-A.y1)
  && check(B.x2-B.x1,B.y2-B.y1,A.x1-B.x1,A.y1-B.y1, A.x2-B.x1,A.y2-B.y1));
}

double dis(double x,double y) { //distance between one node to the origin
   return sqrt(x*x+y*y);
}

void cross(line A,line B) { //for every two lines determine if they intersect, if so update b and s
   double A1,B1,C1,A2,B2,C2,x,y;
   A1=A.y2-A.y1;B1=A.x1-A.x2;C1 = -(A.x1*A1 + A.y1*B1);
   A2=B.y2-B.y1; B2=B.x1-B.x2; C2 =-(B.x1*A2+B.y1*B2);
   x=-(C1*B2-C2*B1) / (A1*B2-A2*B1);
   y=-(A1*C2-A2*C1) / (A1*B2-A2*B1);
   double l = dis(x-A.x1,y-A.y1);
   if (isfinite(l) && l < dis(A.x2-A.x1,A.y2-A.y1) ) {b[s] = l;}
}

bool init() { //initiate and input data to the global variable, if get 0 end
   int i;
   scanf("%d",&n);
   if(n==0)
   return false;
   for(i=1;i<=n;i++)
    scanf("%d %d %d %d", &a[i].x1,&a[i].y1,&a[i].x2,&a[i].y2);
   return true;
}

int main() {
   cases=0;
   while(init()) {
    ans=0;
    for(i=1;i<=n;i++){
      s = 2;
      b[1]=-25; b[2]=dis(a[i].x2-a[i].x1,a[i].y2-a[i].y1)+25;   //mark two end point 25m further so
            they can be viewed as intersections
      for(j=1;j<=n;j++)
        if(i!=j) //if two lines are the same skip
        if(intersect(a[i], a[j])) { //if two lines intersect
          s++;cross(a[i],a[j]); //increase the number of intersections (s), and call cross to update b
        }
      sort(b+1, b+s+1); //sort b so that all nodes are arranged in order
      for(j = 1; j < s ; j++) { // for each segments the max number of trees to be planted was [l/50]
            (lower bound)
        len = b[j+1] - b[j];
        k = (int)(len/50);
        ans+=k; //add to the output
      }
    }
    printf("Map %d\nTrees = %d\n",++cases,ans);
  }
}
```

1.9 I. Suspense!

```
#include <stdio.h>
#include <string.h>
#include <math.h>

// the information about which animal lives in a particular window
char ch;

// Jan and Tereza lives on the Tj, Tt 'th floor, respectively
int C, Tj, Tt, i, j, k;

// d is the distance between the two buildings, ans is the length of the rope
// that connects the two windows
double d, ans;

// a[] is the info about all animals in Jan's building, and b[] is the info
// about all animals in Tereza's building
int a[30], b[30];


// helper method to calculate the length of a parabola given a, c, and x
double calc_2(double a, double c, double x) {
    return x / 2 * sqrt(a*x*x + c) + c / (2*sqrt(a)) *
        log(x * sqrt(a) + sqrt(a*x*x + c));
}

// calculate the length of a parabola with formula Y = A (x - B)^2 + C, after
// some mathematic manipulations
double calc_1(double C) {
    // use a, b, c to represent the coefficients of the parabola in order to
    // simplify the calculations of solving an equation
    double A, B, a, b, c;
    if (Tt == Tj) B = d / 2;
    else {
        a = (Tt-Tj) * 3; b = 2 * d * (Tj*3-2-C); c = -d * d * (Tj*3-2-C);
        B = (-b + sqrt(b*b - 4*a*c)) / (2*a);
        if (B < 0 || B > d) B = (-b - sqrt(b*b - 4*a*c)) / (2*a);
    }
    A = (Tj*3 - 2 - C) / (B*B);
    return calc_2(4*A*A, 1, B) + calc_2(4*A*A, 1, d-B);
}


// determine the type of the animal: 1 means cat, 2 means bird, 0 means other
int in() {
    scanf("%c", &ch);
    // discard any whitespace characters by continuing scanf
    while (ch != 'C' && ch != 'B' && ch != 'N') scanf("%c", &ch);
    if (ch == 'C') return 1;
    else if (ch == 'B') return 2;
    else return 0;
}


int main() {
    C = 0;

    while (true) {
        scanf("%d %d %lf", &Tj, &Tt, &d);
        if (Tj == 0 && Tt == 0 && d < 1e-6) break;

        memset(a, 0, sizeof(a)); memset(b, 0, sizeof(b));
        // fill in the info about Jan's and Tereza's buildings
        for (i = 1; i <= Tj; i++) a[i] = in();
        for (i = 1; i <= Tt; i++) b[i] = in();

        // if not the first, then print an empty line to match the format
        if (C) printf("\n");

        // search over all the floors below where Jan and Tereza live
        ans = -1;
        for (i = 1; i < Tt && i < Tj; i++) {
            // at least 1 meter from the ground, so 1st floor is excluded
            // if there's not cat to jump onto the rope, and no birds to
            // attack the rope, then satisfies our requirements
            if (i > 1 && ((a[i] != 1 && b[i] != 1) ||
                (a[i-1] != 2 && b[i-1] != 2)))
            {
                ans = calc_1(i*3 - 2 - 0.5 + 1);
                printf("Case %d: %.3lf\n", ++C, ans);
                break;
            }

            // same as before, second case
            if ((a[i] != 1 && b[i] != 1) || (a[i] != 2 && b[i] != 2)) {
                ans = calc_1(i*3 - 2 + 1);
                printf("Case %d: %.3lf\n", ++C, ans);
                break;
            }

            // same as before, third case
            if ((a[i+1] != 1 && b[i+1] != 1) || (a[i] != 2 && b[i] != 2)) {
                ans = calc_1(i*3 - 2 + 0.5 + 1);
                printf("Case %d: %.3lf\n", ++C, ans);
                break;
            }
        }

        // if no such bridge can be built, then output impossible
        if (ans < 0) printf("Case %d: impossible\n", ++C);
    }

    return 0;
}
```

1.10 J. Air Traffic Control

```
#include <stdio.h>
#include <algorithm>
#include <string.h>
#include <math.h>

using namespace std;

// required precision of radius is 1 * 10^(-5)
const double eps = 1e-5;

// struct of the coordinates of a point
struct coor {
    double x, y;
};
// operator overload: "-" means shift point a to the position of point b
coor operator -(coor a, coor b) {
    a.x -= b.x;
    a.y -= b.y; return a;
}

// NP is the plane number, NC is the number of control centers, and ok is to
// check whether some conditions are met
double r; int C, NP, NC, m, i, j, k, ok;

// covers[i] records how many control centers plane i is controlled, ans[i]
// records how many planes are under control of center i, cov[i] records the
// best control radius of control center i, and Tc[i] records the coverage of
// planes under the control of center i
int covers[100], ans[10], cov[100], Tc[100];

// coordinates of all planes, and some extra global structs
coor A, B; coor P[100];


// compare the priorities of point a and point b, based on their coordinates;
// pay attention to the error possibly caused by precision
bool cmp(coor a, coor b) {
    return a.y > b.y+eps || (a.y > b.y-eps && a.x > b.x);
}


// returns the distance from point a to the origin
double len(coor a) {
    return sqrt(a.x*a.x + a.y*a.y);
}


// given 3 points A, B, and C, determine the center of the unique circle that
// intersects all 3 points
bool cross(coor A, coor B, coor C, coor &O) {
    double A1, B1, C1, A2, B2, C2, tmp;

    A1 = B.x - A.x; B1 = B.y - A.y;
    // the line perpendicular to AB and which passes through midpoint of AB
    C1 = (A.y*A.y - B.y*B.y + A.x*A.x - B.x*B.x) / 2.0;
    A2 = C.x - A.x; B2 = C.y - A.y;
    // the line perpendicular to ABCand which passes through midpoint of AC
    C2 = (A.y*A.y - C.y*C.y + A.x*A.x - C.x*C.x) / 2.0;

    tmp = B1*A2 - B2*A1;
    // if the two lines are parallel to each other, then no circle made
    if (fabs(tmp) < eps) return false;
    // otherwise, fill the coordinates of the center in O and return true
    O.y = (C2*A1 - C1*A2) / tmp; O.x = (C1*B2 - C2*B1) / tmp;
    return true;
}


// check whether plan "Tc" is better then the best-so-far solution "cov"
bool check() {
    int i;
    for (i = 0; i < NP; i++) {
        if (cov[i] && !Tc[i]) return false;
        if (!cov[i] && Tc[i]) return true;
    }
    return true;
}


// calculate the best control radius based on border coordinates of A B, and
// coordinates of plane C
void work(coor A, coor B, coor C) {
    coor O; double Tr; int i, in, on;

    // if the radius cannot be determined, then returns, otherwise calculate
    // the circle center denoted by "O"
    if (!cross(A, B, C, O)) return;

    Tr = len(A-O); in = 0; on = 0;
    // calculate number of planes inside control, denoted by "in", and number
    // of planes on the border, denoted by "on"
    for (i = 0; i < NP; i++) {
        if (len(O-P[i]) < Tr-eps) in++;
        else if (len(O-P[i]) < Tr+eps) on++;
    }
    // if obtained illegal values for "in" or "on", then returns
    if (in > m || in+on < m) return;

    // calculate the remaining number of planes on the border
    on = m-in;
    // iterate over all planes to determine if each plane is under control
    for (i = 0; i < NP; i++) {
        if (len(O-P[i]) < Tr-eps) Tc[i] = 1;
        else if (len(O-P[i]) < Tr+eps && on) { Tc[i] = 1; on--; }
        else Tc[i] = 0;
    }
    // compare this solution with the best obtained solution so far
    if (r < 0|| Tr < r-eps || (Tr < r+eps && check())) {
        r = Tr;
        for (i = 0; i < NP; i++) cov[i] = Tc[i];
    }
}


int main() {
    while (scanf("%d %d", &NP, &NC) && NP) {
        for (i = 0; i < NP; i++) scanf("%lf %lf", &P[i].x, &P[i].y);
        // sort the planes according to their coordinates
        sort(P, P+NP, cmp);
        memset(covers, 0, sizeof(covers)); ok=1;

        for (i = 0; i < NC; i++) {
            scanf("%d %lf %lf %lf %lf", &m, &A.x, &A.y, &B.x, &B.y);
            r = -1;
            // for each plane, work over all regions to get the best one
            for (j = 0; j < NP; j++) work(A, B, P[j]);
            // accumulate the count of how many controls are on this plane
            for (j = 0; j < NP; j++) covers[j] += cov[j];
            // for some control center, no region is legal (r<0), so not ok
            if (r < 0) ok = 0;
        }

        memset(ans, 0, sizeof(ans));
        // calculate the number of planes center i is now controlling
        for (i = 0; i < NP; i++) ans[covers[i]]++;

        // output the results of this test case
        printf("Trial %d: ", ++C);
        if (ok) for (i = 0; i <= NC; i++) printf("%d  ", ans[i]);
        else printf("Impossible");
        printf("\n\n");
    }

    return 0;
}
```

2 World Final 2005

2.1 A. Eyeball Benders

```
#include <stdio.h>
#include <math.h>

// the struct to represent a line segment by two points
struct line {
    double x1, y1, x2, y2;
};
// the struct to represent a point
struct point {
    double x, y;
};

// based on the required precision, we can calculate with an eps of 1*10^(-6)
const double eps = 1e-6;

// scale is the ratio between the large and the small picture
bool ok; int T, n, m, i, j, k; double scale;

// pL, pS are endpoints on the large and the small picture, respectively
point pL, pS;

// record the line segments on both the large and the small picture
line tL; line L[60], S[60], SN[60];


// read input and return a line struct
line in() {
    double tmp; line t;
    scanf("%lf %lf %lf %lf", &t.x1, &t.y1, &t.x2, &t.y2);
    if (t.x1 > t.x2 + eps) {
        // horizontal line, x coordinates are different
        tmp = t.x1; t.x1 = t.x2; t.x2 = tmp;
    }
    if (t.y1 > t.y2 + eps) {
        // vertical line, y coordinates are different
        tmp = t.y1; t.y1 = t.y2; t.y2 = tmp;
    }
    return t;
}


// compare if doubles x and y are equal or not, floating precision handled
bool same(double x, double y) {
    return (y - eps < x && x < y + eps);
}


// calculate the distance between point (x, y) and the origin
double dis(double x, double y) {
    return sqrt(x * x + y * y);
}


// determines if a line p in the small picture corresponds to some lines in
// the large picture
bool findS(line p) {
    // iterate over all line segments in the large picture
    int i;
    for (i = 0; i < n; i++) {
        // if line p in the small picture corresponds to line i in the large
        // picture, then found, return true
        if (same(p.x1, p.x2) && same(L[i].x1, L[i].x2) &&
            same(p.x1, L[i].x1)) if (L[i].y1 <= p.y1 && p.y2 <= L[i].y2)
                return true;
        if (same(p.y1, p.y2) && same(L[i].y1, L[i].y2) &&
            same(p.y1, L[i].y1)) if (L[i].x1 <= p.x1 && p.x2 <= L[i].x2)
                return true;
    }
    // no matches found, line p does not correspond to any line segments
    return false;
}


// determines if a line p in the large picture corresponds to some lines in
// the small picture, similar to findS
bool findL(line p) {
    int i;
    // iterate over all line segments in the small picture
    for (i = 0; i < m; i++) {
        // if line p in the large picture corresponds to line i in the small
        // picture, then found, return true
        if (same(p.x1, p.x2) && same(SN[i].x1, SN[i].x2) &&
            same(p.x1, SN[i].x1)) if (SN[i].y1 <= p.y1 && p.y2 <= SN[i].y2)
                return true;
        if (same(p.y1, p.y2) && same(SN[i].y1, SN[i].y2) &&
            same(p.y1, SN[i].y1)) if (SN[i].x1 <= p.x1 && p.x2 <= SN[i].x2)
                return true;
    }
    // no matches found, line p does not correspond to any line segments
    return false;
}


// fix some scale and corresponding features, determine if the small and the
// large picture match with each other
bool check() {
    double MaxX, MaxY, MinX, MinY; int i, j, k;
    // determine the 4 bounding coordinates of the sliding window
    MaxX = MinX = pL.x; MaxY = MinY = pL.y;
    // for each line segment in the small picture, calculate its coordinates
    // after scaling and shifting, handle sliding window accordingly
    for (i = 0; i < m; i++) {
        SN[i].x1 = (S[i].x1 - pS.x) * scale + pL.x;
        SN[i].y1 = (S[i].y1 - pS.y) * scale + pL.y;
        SN[i].x2 = (S[i].x2 - pS.x) * scale + pL.x;
        SN[i].y2 = (S[i].y2 - pS.y) * scale + pL.y;

        // if the line SN[i] has appeared in the large picture before, then
        // terminate this method directly
        if (!findS(SN[i])) return false;

        if (SN[i].x1 < MinX) MinX = SN[i].x1;
        if (SN[i].x2 > MaxX) MaxX = SN[i].x2;
        if (SN[i].y1 < MinY) MinY = SN[i].y1;
        if (SN[i].y2 > MaxY) MaxY = SN[i].y2;
    }

    // for each line segment in the large picture, determine, based on if the
    // line is horizontal or vertical, if there’s any corresponding line in
    // the small picture inside the sliding window
    for (i = 0; i < n; i++) {
        if (same(L[i].x1, L[i].x2)) {
            // if the line i in the large picture is vertical
            if (L[i].x1 > MaxX) {
                // if line i is strictly one the left or right of the sliding
                // window and the distance is less than eps, return false
                if (L[i].y1 > MaxY)
                    if (dis(L[i].x1-MaxX, L[i].y1-MaxY) < 0.005) return false;
                else if (L[i].y2 < MinY)
                    if (dis(L[i].x1-MaxX, L[i].y2-MinY) < 0.005) return false;
                else
                    if (L[i].x1 - MaxX < 0.005) return false;
            } else if (L[i].x2 < MinX) {
                // similar case as before
                if (L[i].y1 > MaxY)
                    if (dis(L[i].x2-MinX, L[i].y1-MaxY) < 0.005) return false;
                else if (L[i].y2 < MinY)
                    if (dis(L[i].x2-MinX, L[i].y2-MinY) < 0.005) return false;
                else
                    if (MinX - L[i].x2 < 0.005) return false;
            } else {
                if (L[i].y2 < MinY-0.005 || L[i].y1 > MaxY+0.005) continue;
                if (L[i].y2 < MinY || L[i].y1 > MaxY) return false;
                // select the portion of line i inside the sliding window, in
                // the large picture, and see if it matches with small picture
                tL = L[i];
                if (tL.y1 < MinY) tL.y1 = MinY;
                if (tL.y2 > MaxY) tL.y2 = MaxY;
                if (!findL(tL)) return false;
            }
        } else {
            // if the line i in the large picture is horizontal
            if (L[i].y1 > MaxY) {
                // similar as in the first portion of the outer if statement
                if (L[i].x1>MaxX)
                    if (dis(L[i].x1-MaxX, L[i].y1-MaxY) < 0.005) return false;
                else if (L[i].x2<MinX)
                    if (dis(L[i].x2-MinX, L[i].y1-MaxY) < 0.005) return false;
                else
                    if (L[i].y1 - MaxY < 0.005) return false;
            } else if (L[i].y2 < MinY) {
                // similar as in the first portion of the outer if statement
                if (L[i].x1 > MaxX)
                    if (dis(L[i].x1-MaxX, L[i].y2-MinY) < 0.005) return false;
                else if (L[i].x2 < MinX)
                    if (dis(L[i].x2-MinX, L[i].y2-MinY) < 0.005) return false;
                else
                    if (MinY - L[i].y2 < 0.005) return false;
            } else {
                // similar as in the first portion of the outer if statement
                if (L[i].x2 < MinX-0.005 || L[i].x1 > MaxX+0.005) continue;
                if (L[i].x2 < MinX || L[i].x1 > MaxX) return false;
                tL = L[i];
                if (tL.x1 < MinX) tL.x1 = MinX;
                if (tL.x2 > MaxX) tL.x2 = MaxX;
                if (!findL(tL)) return false;
            }
        }
    }

    // if reached this point, then all lines in the window match correctly
    return true;
}


// determine if the pair of small/large pictures is a valid puzzle
void work() {
    int k;
    // if there's only one line in the small picture
    if (m == 1) {
        // calculate the scale and determine validness in this scenario
        scale = 0.005 / (S[0].x2 - S[0].x1 + S[0].y2 - S[0].y1);
        ok = scale<=1 && check();
    } else for (k = 0; k < n; k++) {
        // iterate over each line in the large picture that may correspond to
        // line 1 in the small picture
        if ((L[k].x1 - L[k].x2) * (S[1].x1 - S[1].x2) ||
            (L[k].y1 - L[k].y2) * (S[1].y1 - S[1].y2))
        {
            if (S[1].x1 != S[1].x2) {
                // line 1 is horizontal
                if (same(S[1].y1, pS.y)) {
                    // if the fixed endpoint has same y coordinates as line 1
                    // we calculate the distance in x direction
                    if (!same(S[1].x1, pS.x))
                        scale = (L[k].x1 - pL.x) / (S[1].x1 - pS.x);
                    else
                        scale = (pL.x - L[k].x2) / (pS.x - S[1].x2);
                } else {
                    // if the fixed endpoint does not have same y coordinates
                    // then we can directly calculate scale in y direction
                    scale = (L[k].y1 - pL.y) / (S[1].y1 - pS.y);
                }
            } else {
                // line 1 is vertical, similar code as before
                if (same(S[1].x1, pS.x)) {
                    if (!same(S[1].y1, pS.y))
                        scale = (L[k].y1 - pL.y) / (S[1].y1 - pS.y);
                    else
                        scale = (pL.y - L[k].y2) / (pS.y - S[1].y2);
                } else {
                    scale = (L[k].x1 - pL.x) / (S[1].x1 - pS.x);
                }
            }
            // if the scale is less than 1 and the match is successful, then
            // the work is finished, note that we have to keep those scales
            // positive, or otherwise the image might be flipped
            ok = scale <= 1 && scale > eps && check();
            if (ok) return;
        }
    }
}


int main() {
    T = 0;

    while (scanf("%d %d", &m, &n) && (m || n)) {
        for (i = 0; i < m; i++) S[i] = in();
        for (i = 0; i < n; i++) L[i] = in();

        ok = false;
        for (i = 0; i < n; i++) for (j = 0; j < m; j++) if (!ok) {
            // mark line j in small picture as line 0, for convenience
            tL = S[0]; S[0] = S[j]; S[j] = tL;
            // assume the matching endpoint is the left/bottom corner
            pL.x = L[i].x1; pL.y = L[i].y1;
            pS.x = S[0].x1; pS.y = S[0].y1;
            work(); if (ok) break;
            // assume the matching endpoint is the top/right corner
            pL.x = L[i].x2; pL.y = L[i].y2;
            pS.x = S[0].x2; pS.y = S[0].y2;
            work(); if (ok) break;
        }

        if (ok) printf("Case %d: valid puzzle\n\n", ++T);
        else printf("Case %d: impossible\n\n", ++T);
    }

    return 0;
}
```

2.2 B. Simplified GSM Network

```
#include <stdio.h>
#include <math.h>
#include <memory.h>

// the struct to represent a point
struct coor     {
    double x,y;
};

// some limits of unit towers, and cities, given by the problem statement
const int MaxB = 50;
const int MaxC = 50;
const int MaxS = 50 * 50;

// BTS[] is the array of BTS towers, and city[] is the array of cities, S[] is
// the matrix to store the shortest paths
int B, C, R, Q, i, j, k, cases;
coor BTS[MaxB], city[MaxC];     int S[MaxC][MaxC];


// calculate the distance between point (x1, y1) and point (x2, y2)
double dis(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}


// calculate the nearest unit's id "k", from a given point (x, y)
int area(double x, double y) {
    int i, k; double min, tmp;
    // initialize min distance to be the distance from (x, y) to unit 0 (k=0)
    min = dis(x, y, BTS[0].x, BTS[0].y); k = 0;
    // iterate over each unit tower and find out the smallest distance
    for (i = 1; i < B; i++)     {
        tmp = dis(x, y, BTS[i].x, BTS[i].y);
        if (tmp < min) { min = tmp; k = i; }
    }
    return k;
}


// use binary search to calculate how many units have been passed on the path
// from (x1, y1) to (x2, y2)
int get_switch(double x1, double y1, double x2, double y2) {
    // if the closest unit are the same for two points, then distance is 0
    if (area(x1, y1) == area(x2, y2)) return 0;
    // if the two points are actually the same point, then distance is 1
    if (dis(x1, y1, x2, y2) < 1e-6) return 1;

    // handle the left part (closer to (x1, y1)) and the right part (closer to
    // (x2, y2)) separately and recursively
    return get_switch(x1, y1, (x1+x2) / 2, (y1+y2) / 2)
        + get_switch((x1+x2) / 2, (y1+y2) / 2, x2, y2);
}


// read input about a GSM network, and initialize global variables
bool init() {
    scanf("%d %d %d %d", &B, &C, &R, &Q);
    if (B == 0 && C == 0 && R == 0 && Q == 0) return false;
    for (i = 0; i < B; i++) scanf("%lf %lf", &BTS[i].x, &BTS[i].y);
    for (i = 0; i < C; i++) scanf("%lf %lf", &city[i].x, &city[i].y);

    memset(S, 0, sizeof(S));
    for (i = 0; i < R; i++)     {
        // store array indices should be 1 less than the id range from 1 to R
        scanf("%d %d", &j, &k); j--; k--;
        S[j][k] = S[k][j] = 1;
    }
    // calculate how many units between cities i and j, if no paths, then set
    // the distance to be the maximum value
    for (i = 0; i < C; i++) for (j = i+1; j < C; j++) {
        if (S[i][j] == 0) S[i][j] = MaxS;
        else S[i][j] = get_switch(city[i].x, city[i].y, city[j].x, city[j].y);
        S[j][i] = S[i][j];
    }
    // the distance between city i and itself is always equal to 0
    for (i = 0; i < C; i++) S[i][i] = 0;

    return true;
}


int main() {
    cases = 0;

    while (init()) {
        printf("Case %d:\n", ++cases);
        // calculate the shortest path based on the floyd-warshall algorithm
        for (k = 0; k < C; k++)
            for (i = 0; i < C; i++) for (j = 0; j < C; j++)
                if (S[i][j] > S[i][k] + S[k][j]) S[i][j] = S[i][k] + S[k][j];
        // based on the request, determine if it is possible, output results
        for (i = 0; i < Q; i++) {
            scanf("%d %d", &j, &k); j--; k--;
            if (S[j][k] >= MaxS) printf("Impossible\n");
            else printf("%d\n", S[j][k]);
        }
    }

    return 0;
}
```

2.3 C. The Traveling Judges Problem

```
#include <stdio.h>
#include <string.h>
#include <memory.h>

// constraints about maximum cities and judges, defined by problem statement
const int MaxC = 20;
const int MaxJ = 10;

// road[][] stores the graph, judges[] stores the starting city of each judge
// and the back pointers in the spanning tree is stored in fa[]
int cases, NC, NJ, T, DC, map, best_map, ans, i, k;
int road[MaxC][MaxC], judges[MaxJ], fa[MaxC];


// read input about cities, notice that all indices are decremented by 1, to
// fit into those array indices
bool init() {
    int NR, i, C1, C2;
    scanf("%d", &NC); if (NC == -1) return false;
    // T is set to 2ˆDC, so it can denote a city bitmask
    scanf("%d", &DC); DC--; T = 1 << DC;
    scanf("%d", &NR);

    memset(road, 0, sizeof(road));
    // initialize ans to some large value
    ans = 1000;
    for (i = 0; i < NR; i++) {
        scanf("%d %d", &C1, &C2); C1--; C2--;
        scanf("%d", &road[C1][C2]);
        // the graph is undirected, so road[C2][C1] is same as road[C1][C2]
        road[C2][C1] = road[C1][C2];
        ans += road[C1][C2];
    }

    scanf("%d", &NJ);
    // read in info about all judges and set the masks correctly
    for (i = 0; i < NJ; i++) {
        scanf("%d", &judges[i]); judges[i]--;
        // set bit judge[i] in the city bitmask T, so this city is required
        T |= 1 << judges[i];
    }
    return true;
}


// according to a bitmask that denotes the required cities, calculate the min
// spanning tree using prim’s algorithm
void check(int map) {
    int i, j, k, sum, min; int d[MaxC];
    // d[i] = -2 if node i does not connect to the spanning tree, = -1 if node
    // i is inside the spanning tree, or > 0 if it denotes the min distance
    // between point i and the currently constructed spanning tree
    for (i = 0; i < MaxC; i++) d[i] = -2;
    // initialize the destination city (node) to be inside the spanning tree
    d[DC] = -1; sum = 0;

    // iterate over all edges connected to DC, to find a smallest one
    for (i = 0; i < MaxC; i++) if (((1<<i) & map) && road[DC][i]) {
        d[i] = road[DC][i]; fa[i] = DC;
    }
    // body part of prim’s algorithm: constructing the minimum spanning tree
    while (true) {
        min = ans + 10;
        // calculate the node k that is closest to the current tree
        for (i = 0; i < MaxC; i++) if (d[i] > 0 && d[i] < min) {
            min = d[i]; k = i;
        }
        // if there’s no point to expand, then exit the construction
        if (min > ans) break;
        // select node k into the tree, and accumulate the total tree length
        sum += min; d[k] = -1;
        // update the distances and the back pointers of point k, i
        for (i = 0; i < MaxC; i++) if (((1<<i) & map) && road[k][i]) {
            if (d[i] == -2 || d[i] > road[k][i]) {
                d[i] = road[k][i]; fa[i] = k;
            }
        }
    }

    // if there’s no expansion, then this plan is not working; abort
    for (i = 0; i < MaxC; i++) if (((1<<i) & map) && d[i] == -2) return;

    if (sum < ans) {
        // if the tree’s length is smaller than the optimal solution, then we
        // update the optimal solution and the corresponding map
        ans = sum; best_map = map;
    } else if (sum == ans) {
        // if the length is equal, then compare how many edges two solutions
        // involve, and select the smaller one
        j = 0; k = 0;
        for (i = 0; i < MaxC; i++) {
            if ((1<<i) & best_map) j++;
            if ((1<<i) & map) k++;
        }
        if (j > k) {
            best_map = map; return;
        }
        if (j < k) return;
        // if i was not previously selected but is selected now, then we know
        // this solution is better, and we update it to be the optimal one
        for (i=0;i<MaxC;i++) {
            if (((1<<i) & best_map) && !((1<<i) & map)) return;
            if (!((1<<i) & best_map) && ((1<<i) & map)) {
                best_map = map; return;
            }
        }
    }
}


int main() {
    cases = 0;

    while (init()) {
        best_map = 0;
        // iterate over all possible maps (city bitmasks)
        for (map = 0; map < (1<<NC); map++) if ((map&T) == T) check(map);
        printf("Case %d: distance = %d\n", ++cases, ans);
        // get the corresponding route to transfer all judges, and output
        check(best_map);
        for (i = 0; i < NJ; i++) {
            printf("   %d", judges[i]+1); k = judges[i];
            while (k != DC) {
                printf("-%d", fa[k]+1); k = fa[k];
            }
            printf("\n");
        }
        printf("\n");
    }

    return 0;
}
```

2.4 D. cNteSahruPfefrlefe

```
#include <stdio.h>
#include <memory.h>
#include <math.h>

// go[] is the direction that cards go, correct[i][] denotes the correct order
// of cards after shuffle i times, card[] is the current shuffled order, ans[]
// is the final answer, err[i] is the incorrect position in i’th shuffle
int go[52], correct[11][52], card[52], next[52], err[11], ans[11];
int n, i, j, dfs, errors, best, cases, s;


// reverse the process of a shuffle, so we can find out incorrect cards
void anti_shuffle(int n) {
    int tmp[52], i;
    // store the anti-shuffled cards in a temporary array, then update next[]
    for (i = 0; i < 52; i++) tmp[correct[n][i]] = next[i];
    memcpy(next, tmp, sizeof(tmp));
}


// calculate the difference function between sequence a and sequence b
int differents(int *a, int *b) {
    int i, j, s; int step[52], go[52];
    memset(step, 0, sizeof(step));
    // the going direction of element a[i] is stored in go[a[i]]
    for (i = 0; i < 52; i++) go[a[i]] = b[i];

    s = 0;
    // if element i is not visited, then set it as the start of iterations
    for (i = 0; i < 52; i++) if (!step[i]) {
        j = i;
        // if not going back to the start, then continue iterating
        while (!step[j]) {
            step[j] = 1; j = go[j];     s++;
        }
        // the return value is one less than the loop length of the cards
        s--;
    }
    return s;
}


// recursively identify each incorrect position during every shuffle
void trytrytry(int i) {
    int j, save[52];
    // calculate the difference function between the correct card sequence and
    // the current obtained sequence
    dfs = differents(card, correct[i]);
    // if the value is larger than number of shuffles, then stop recursion
    if (dfs > i) return;
    // base case, update best answer, and record current incorrect position
    if (!i)     {
        if (errors < best)      {
            best = errors; memcpy(ans, err, sizeof(err));
        }
        return;
    }

    // copy the current card sequence in order for backup
    memcpy(save, card, sizeof(card));
    // initialize this time's error to be equal to -1, which means no errors
    err[i] = -1;
    for (j = 0; j < 51; j++) {
        // initialize a new sequence of cards
        memcpy(next, card, sizeof(card));
        // exchange the positions of two neighboring cards
        next[j] += next[j+1];
        next[j+1] = next[j] - next[j+1]; next[j] -= next[j+1];

        // calculate difference function, and anti-shuffle the sequence once
        dfs = differents(next, correct[i]);
        anti_shuffle(1);

        // accumulate numbef of errors, and record the error position j
        memcpy(card, next, sizeof(card)); errors++; err[i] = j;
        // when there's a potentially better answer, enter the next recursion
        if (errors + dfs < best) trytrytry(i-1);
        // the backtracking step, return the error number to the last state
        errors--; err[i] = -1; memcpy(card, save, sizeof(card));
    }

    // if this shuffle does not contain any errors, then anti-shuffle 1 time,
    // and directly enter the next recursion
    memcpy(next, card, sizeof(card));
    anti_shuffle(1);
    memcpy(card, next, sizeof(card));
    trytrytry(i-1);
    // the backtracking step, return the card sequence to the last state
    memcpy(card, save, sizeof(card));
}


int main() {
    // initialize the card sequences and the go[] directions
    for (i = 0; i < 52; i++) {
        go[i] = 26 * (1 - i%2) + i/2;
        correct[0][i] = i;
    }
    // correctly shuffle 10 times, and record the correct sequences
    for (i = 1; i < 11; i++) for (j = 0; j < 52; j++)
        correct[i][j] = correct[i-1][go[j]];

    scanf("%d", &s);
    for (cases = 1; cases <= s; cases++) {
        for (i = 0; i < 52; i++) scanf("%d", &card[i]);
        for (i = 0; i < 11; i++) {
            // compare this sequence with the correct sequence
            dfs = differents(card, correct[i]);
            // if the function value is not greater than time shuffled, then
            // the given sequence must have been shuffled i times
            if (dfs <= i) break;
        }
        printf("Case %d\nNumber of shuffles = %d\n", cases, i);
        if (!dfs) {
            printf("No error in any shuffle\n\n"); continue;
        }
        n = i;  best = dfs + 1; errors = 0;
        // check any incorrect position for all i times of shuffle
        trytrytry(i);
        for (i = 1; i <= n; i++) if (ans[i] > -1)
            printf("Error in shuffle %d at location %d\n", i, ans[i]);
        printf("\n");
    }

    return 0;
}
```

2.5 E. Lots of Sunlight

```
#include <stdio.h>
#include <math.h>

// h[] is each building's height, d[] is the distance from the next building
// to the left, and other global variables
int n, h[111], d[111], width, height, sum;
int i, di, num, level, id, cases, dh, dw, h1, m1, h2, m2;

// Half_pi stores the value pi/2, corresponding to 45 degrees angle
double min, max, angle, Half_pi;


int main() {
    // the precise value of pi/2 is given by arccos(0)
    Half_pi = acos(0);

    while (scanf("%d", &n) && n) {
        scanf("%d %d", &width, &height);
        // calculate the distance between current and the leftmost building
        sum = 0;
        for (i = 0; i < n; i++) {
            scanf("%d", &h[i]); d[i] = sum;
            if (i < n-1) {
                scanf("%d", &di); sum += di + width;
            }
        }

        printf("Apartment Complex: %d\n", ++cases);
        while (scanf("%d", &num) && num) {
            printf("Apartment %d: ", num);
            // calculate this apartment's height and its id
            level = num / 100 - 1; id = num % 100 - 1;
            // if the querying building does not exist, then exit directly
            if (id < 0 || id >= n || level < 0 || level >= h[id]) {
                printf("Does not exist\n"); continue;
            }
            // initialize the starting and the ending angles to be 0
            min = 0; max = 0;

            // check the left buildings one by one
            for (i = 0; i < id; i++) {
                // calculate the horizontal and vertical distances
                dw = d[id] - d[i] - width; dh = height * (h[i] - level);
                // if the height is no lower, then possible to have sunlight
                // calculate the angle and compare it with optimum
                if (dh > 0) {
                    angle = asin(dh / sqrt(dh*dh + dw*dw));
                    if (angle > min) min = angle;
                }
            }

            // check the right buildings one by one, similar to left case
            for (i = id+1; i < n; i++) {
                dw = d[i] - d[id] - width; dh = height * (h[i] - level);
                if (dh > 0)     {
                    angle = asin(dh / sqrt(dh*dh + dw*dw));
                    if (angle > max) max = angle;
                }
            }

            // convert the starting and ending angles to time, in seconds
            min = 20220 + 22800 * min / Half_pi;
            max = 65820 - 22800 * max / Half_pi;
            // calculate the times, according to the output format
            h1 = int(min / 3600); min -= h1 * 3600;
            m1 = int(min / 60); min -= m1 * 60;
            h2 = int(max / 3600); max -= h2 * 3600;
            m2 = int(max / 60); max -= m2 * 60;

            printf("%02d:%02d:%02d - %02d:%02d:%02d\n", h1, m1, int(min),
                h2, m2, int(max));
        }
    }

    return 0;
}
```

2.6 F. Crossing Streets

```
#include <stdio.h>
#include <algorithm>

using namespace std;

// MaxS is the maximum number of streets, MaxN is the maximum coordinates
const int MaxS = 500;
const int MaxN = 500 * 2 + 4;

// the unit vector in 4 horizontal and vertical directions, for simplicity
const int add[4][2] = {0,1,  0,-1, -1,0, 1,0};

// home coordinates (xh, yh), university coordinates (xu, yu), answer is ans
int cases, xh, yh, xu, yu, n, N, q1, q2, q3, ans, i;

// grid to push flow over the graph, so we could determine the mininum number
// of streets Peter has to cross
int coor[2][MaxS * 2 + 4];
int w[MaxS * 2 + 4][MaxS * 2 + 4];
int list[MaxN * MaxN][2];

// if there's a street blocking the way, represented by a 4-bit bitmask
int map[MaxN][MaxN];


// binary search over the array coor[type] for an edge with index value T
int find(int type,int T) {
    int l, r, m; l = 0; r = n + n + 1;
    while (l != r) {
        m = (l + r) / 2;
        if (coor[type][m] < T) l = m + 1; else r = m;
    }
    return l;
}


// read input and set up the coordinates and the flow graph
bool init() {
    int i, j, k, tmp; int streets[MaxS][4];
    scanf("%d", &n); if (n == 0) return false;

    // read in each street's original coordinates
    for (i = 0; i < n; i++){
        for (j = 0; j < 4; j++) scanf("%d", &streets[i][j]);
        coor[0][i] = streets[i][0];
        coor[1][i] = streets[i][1];
        coor[0][i + n] = streets[i][2];
        coor[1][i + n] = streets[i][3];
    }
    // read in peter's home and university coordinates
    scanf("%d %d %d %d", &xh, &yh, &xu, &yu);
    coor[0][n + n] = xh; coor[1][n + n] = yh;
    coor[0][n + n + 1] = xu; coor[1][n + n + 1] = yu;

    // sort all coordinates; shift all coordinates right and up by 1, so we
    // could leave space for Peter to start his walk
    sort(coor[0],coor[0] + n + n + 2);
    sort(coor[1],coor[1] + n + n + 2);

    // normalize, ensure that x1 < x2 or y1 < y2, for convenience purposes
    for (i = 0; i < n; i++) {
        if (streets[i][0] < streets[i][2]) {
            tmp = find(0, streets[i][0]) + 1;
            streets[i][2] = find(0, streets[i][2]) + 1;
            streets[i][0] = tmp;
        } else {
            tmp = find(0, streets[i][2]) + 1;
            streets[i][2] = find(0, streets[i][0]) + 1;
            streets[i][0] = tmp;
        }
        if (streets[i][1] < streets[i][3]) {
            tmp = find(1, streets[i][1]) + 1;
            streets[i][3] = find(1, streets[i][3]) + 1;
            streets[i][1] = tmp;
        } else {
            tmp = find(1, streets[i][3]) + 1;
            streets[i][3] = find(1, streets[i][1]) + 1;
            streets[i][1] = tmp;
        }
    }

    // get the new coordinates of university and home after normalizing
    xh = find(0, xh) + 1; yh = find(1, yh) + 1;
    xu = find(0, xu) + 1; yu = find(1, yu) + 1;

    // the range now is actually 0 to (n + n + 3) in each dimension
    N = n + n + 3;
    // draw the streets on the graph, and represent if there's a street using
    // 4-bit binary values (bitmasks)
    memset(map, 0, sizeof(map));

    for (i = 0; i < n; i++) {
        // handle horizontal and vertical streets separately
        if (streets[i][0] == streets[i][2]) {
            for (j = streets[i][1]; j < streets[i][3]; j++) {
                map[streets[i][0]][j] |= 4;
                map[streets[i][0] - 1][j] |= 8;
            }
        } else {
            for (j = streets[i][0]; j < streets[i][2]; j++) {
                map[j][streets[i][1]] |= 2;
                map[j][streets[i][1] - 1] |= 1;
            }
        }
    }
    return true;
}


// simulate the scenario if Peter wants to finish crossing under t streets
void go(int x,int y,int t) {
    int i, h, l;
    // try to push flow in all the 4 directions
    for (i = 0; i < 4; i++) if (!(map[x][y] & (1<<i)) || t) {
        h = x + add[i][0]; l = y + add[i][1];
        if (h < 0 || l < 0 || h > N || l > N || w[h][l]) continue;

        w[h][l] = 1;  list[q3][0] = h; list[q3][1] = l; q3++;
        if (map[x][y] & (1<<i)) go(h, l, t-1);
        else go(h, l, t);
    }
}


int main() {
    cases = 0;

    while (init()) {
        memset(w, 0, sizeof(w)); w[xh][yh] = 1;
        list[0][0] = xh; list[0][1] = yh;
        // push flow from the starting point, not crossing any streets now
        q3 = 1; go(xh, yh, 0);
        // when doing BFS, q1 is the source, q2 is the sink, and q3 is sink
        // for the next level of flow
        q1 = 0; q2 = q3; ans = 0;

        while (true) {
            if (w[xu][yu]) break;
            // from the previous level, try to push flow one level further
            for (i = q1; i < q2; i++) go(list[i][0], list[i][1], 1);
            q1 = q2; q2 = q3; ans++;
        }

        printf("City %d\n", ++cases);
        printf("Peter has to cross %d streets\n", ans);
    }

    return 0;
}
```

2.7 G. Tiling the Plane

```
#include <stdio.h>
#include <memory.h>

// c[] is the the direction sequence, sequence_a[] and sequence_b[] are 2 sub
// sequences to be matched
char c[55], ch[5555], sequence_a[2777], sequence_b[2777];

// num[] is the length array, combo_a[] and combo_b[] denote the number of the
// repeated elements in sequence a and b (see above), total[] is the frequency
// of one of the 4 directions
int num[55], combo_a[2777], combo_b[2777], total[4];

// other necessary global variables
int n, i, j, k, len, count, reverse, ok, cases;
```

```
// the process of tiling the plane given the polygon’s shape
void tr(int i, int point) {
    int j, min, step;
    // if all parts the sequence is matched, then we can tile the plane
    if (point >= len) {
        ok = 1; return;
    }
    // if we have tried at least 3 parts, then we can abort the tiling
    if (i > 2) return;

    // try to match over the next dividing point (end of two sides)
    for (j = 0; j < len-point; j++) {
        step = 0;
        // check whether two portions of the sequence can match
        while (sequence_a[point+step] && sequence_a[point+step]
            == sequence_b[j+step])
        {
            // use the portion repeated less times as the next checkpoint
            if(combo_a[point+step] > combo_b[j+step]) min = combo_b[j+step];
            else min = combo_a[point+step];
            step += min + 1;
        }
        // if two sequences match in this checkpoint, then move on to next one
        if (j+step >= len-point) tr(i+1, point+step);
    }
}


int main() {
    while (scanf("%d", &n) && n) {
        // initialize all sequences to 0
        memset(ch, 0, sizeof(ch));
        memset(sequence_a, 0, sizeof(sequence_a));
        memset(sequence_b, 0, sizeof(sequence_b));
        // read input of each line segment’s length and direction
        for (i = 0; i < n; i++) scanf(" %c %d", &c[i], &num[i]);

        ok = 0;
        // iterate over each dividing point in the two sequences
        for (i = 0; i < n/2; i++) {
            len = 0;
            // try to expand the direction sequence
            for (j = 0; j < n; j++)     {
                count = (i + j) % n;
                // repeat a given number of times for the current direction
                for (k = 0; k < num[count]; k++) ch[len++] = c[count];
            }

            // calculate over part a of the sequence
            count = 0;
            for (j = 0; j < 4; j++) total[j] = 0;
            for (j = len/2 - 1; j >= 0; j--) {
                // the first len/2 characters belong to sequence a
                sequence_a[j] = ch[j];
                // accumulate the frequency of the 4 directions
                if (ch[j] == ’N’) total[0]++;
                if (ch[j] == ’S’) total[1]++;
                if (ch[j] == ’W’) total[2]++;
                if (ch[j] == ’E’) total[3]++;
                if (j < len/2 - 1 && ch[j] == ch[j+1]) count++;
                else count = 0;
                // store the time of repeated characters into array combo[]
                combo_a[j] = count;
            }

            // calculate over part b of the sequence
            count = 0;
            for (j = len/2; j < len; j++) {
                // reverse each direction and accumulate, in this case
                reverse = len - j - 1;
                if (ch[j] == ’N’) {
                    sequence_b[reverse] = ’S’; total[1]--;
                }
                if (ch[j] == ’S’) {
                    sequence_b[reverse] = ’N’; total[0]--;
                }
                if (ch[j] == ’W’) {
                    sequence_b[reverse] = ’E’; total[3]--;
                }
                if (ch[j] == ’E’) {
                    sequence_b[reverse] = ’W’; total[2]--;
                }
                if (j > len/2 && ch[j] == ch[j-1]) count++;
                else count = 0;
                combo_b[reverse] = count;
            }

            // determine if the directions are same; if not, stop counting
            for (j = 0; j < 4; j++)     if (total[j]) break;
            // if all directions have same frequency, set "length" to be the
            // length of sequence a, and now iterate over all breakpoints
            if (j > 3) {
                len /= 2; tr(0, 0);
                if (ok) break;
            }
        }

        printf("Polygon %d: ", ++cases);
        if (ok) printf("Possible\n"); else printf("Impossible\n");
    }

    return 0;
}
```

2.8 H. The Great Wall Game

```
#include <stdio.h>
#include <string.h>

// the max dimension of the board is 15, as stated in the problem statement
const int MaxN = 15;

// a[][] is the adjacency matrix of the bipartite graph, Lx[] and Ly[] are the
// coordinates of possible moves, matchY[] denotes the matching edges
int n, ans, cases;
int a[MaxN+1][MaxN+1], Lx[MaxN+1], Ly[MaxN+1], matchY[MaxN+1];

// usedX[], usedY[] denotes the validity of visiting points in sets X and Y
bool usedX[MaxN+1], usedY[MaxN+1];


// whether there’s an augmenting path from node r in the equivalent subgraph
bool path(int r) {
    int i; usedX[r] = true;
    for (i = 0; i < n; ++i) if (Lx[r] + Ly[i] == a[r][i]) if (!usedY[i]) {
        usedY[i] = true;
        if (matchY[i] < 0 || path(matchY[i])) {
            matchY[i] = r; return true;
        }
    }
    return false;
}


// complete the bipartite matching, using a graph that represent the board
void work() {
    int i, j, k, res = 0;
    // calculate initial values for the array Lx[]
    for (i = 0; i < n; ++i)     {
        Lx[i] = -2147483647;
        for (j = 0; j < n; ++j) if (a[i][j] > Lx[i]) Lx[i] = a[i][j];
    }
    // initialize Ly to be zeros, and no initial matching edges (value = -1)
    memset(Ly, 0, sizeof(Ly));
    memset(matchY, -1, sizeof(matchY));

    for (i = 0; i < n; ++i) while (true) {
        // set the nodes in X and Y to be unvisited at the beginning
        memset(usedX, false, sizeof(usedX));
        memset(usedY, false, sizeof(usedY));
        // if there’s an augmenting path from i, then break out of the loop
        if (path(i)) break;

        // calculate how much can be improved (denoted by "delta")
        int delta = 2147483647, v;
        for (j = 0; j < n; ++j) if (usedX[j])
            for (k = 0; k < n; ++k) if(!usedY[k])
        {
            v = Lx[j] + Ly[k] - a[j][k]; if (v < delta) delta = v;
        }
        // update the corresponding labels in usedX[] and usedY[]
        for (j = 0; j < n; ++j) if (usedX[j]) Lx[j] -= delta;
        for (j = 0; j < n; ++j) if (usedY[j]) Ly[j] += delta;
    }

    // calculate the sum of weights of matching weights, and update optimum
    for (i = 0; i < n; ++i) if (matchY[i] >= 0) res += a[matchY[i]][i];
    if (res > ans) ans = res;
}


// calculate the distance from point (x, y) to the origin
int dis(int x, int y) {
    if (x * y >= 0) x += y;
    else x -= y;
    if (x > 0) return x; else return -x;
}
```

```
int main() {
    int i, j, k; int P[MaxN+1][2];

    while (true) {
        scanf("%d", &n); if (n == 0) break;
        ans = - (n * n + n);
        // we are calculate a maximum for matching, so we negate every value
        // and so we are searching for "minumum negative"
        for (i = 1; i <= n; i++) scanf("%d %d", &P[i][0], &P[i][1]);

        for (i = 1; i <= n; i++) {
            // construct a bipartite map based on the i’th row of the board
            for (j = 1; j <= n; j++) for (k = 1; k <= n; k++)
                a[j-1][k-1] = - dis(P[j][0] - i, P[j][1] - k);
            work();
        }
        for (i = 1; i <= n; i++) {
            // construct a bipartite map based on the i’th column of the board
            for (j = 1; j <= n; j++) for (k = 1; k <= n; k++)
                a[j-1][k-1] = - dis(P[j][0] - k,P[j][1] - i);
            work();
        }

        // construct the bipartite based on the 2 main diagonals of the board
        for (j = 1; j <= n; j++) for (k = 1; k <= n; k++)
            a[j-1][k-1] = - dis(P[j][0] - k,P[j][1] - k);
        work();
        for (j = 1; j <= n; j++) for (k = 1; k <= n; k++)
            a[j-1][k-1] = - dis(P[j][0] - k,P[j][1] - (n - k + 1));
        work();

        printf("Board %d: %d moves required.\n\n", ++cases, -ans);
    }

    return 0;
}
```

2.9 I. Workshops

```
#include <stdio.h>
#include <algorithm>

using namespace std;

// the struct to represent a workshop, and its room assignment
struct info {
    int p,t;
};

// the comparator of struct info, used for sorting an array
bool cmp(info x, info y) {
    return (x.t < y.t);
}

// some constraints defined by the problem statement
const int MaxW = 1000; const int MaxR = 1000;

// sum_w, sum_p denotes the workshops and people yet to be assigned
int w, r, cases, sum_w, sum_p, i, j, l;

// workshop[] is the array of workshops, room[] is the array of rooms
info workshop[1100], room[1100];


// read input and initialize all global variables and arrays
bool init() {
    int i, hh, mm;
    scanf("%d", &w); if (w == 0) return false;

    sum_w = w; sum_p = 0;
    for (i = 0; i < w; i++) {
        scanf("%d %d", &workshop[i].p, &workshop[i].t);
        sum_p += workshop[i].p;
    }

    scanf("%d", &r);
    for (i=0;i<r;i++) {
        scanf("%d %d:%d", &room[i].p, &hh, &mm);
        // calculate the time that room i can be used for workshop
        room[i].t = (hh - 14) * 60 + mm;
    }

    // sort both workshop[] and room[] according to time, in increasing order
    sort(workshop, workshop + w, cmp);
    sort(room, room + r, cmp);

    return true;
}


int main() {
    cases = 0;

    while (init()) {
        // use a greedy strategy, assign rooms according to their time order
        for (i = 0; i < r; i++) {
            l = -1;
            for (j = 0; j < w; j++) {
                // if there’s some workshop that cannot be assigned to room i
                // at any time, then break out, since later workshops are even
                // longer, and still cannot fit into this room
                if (workshop[j].t > room[i].t) break;

                // otherwise, if workshop j satisfies all requirements, then
                // assign it to room i
                else if (workshop[j].p <= room[i].p && workshop[j].p >= 0) {
                    if (l<0 || workshop[l].p<workshop[j].p) l = j;
                }
            }

            // if the room i is assigned some workshop, then calculate number
            // of people in the workshop, and set the flag
            if (l >= 0) {
                sum_w --; sum_p -= workshop[l].p; workshop[l].p = -1;
            }
        }

        printf("Trial %d: %d %d\n\n", ++cases, sum_w, sum_p);
    }

    return 0;
}
```

2.10 J. Zones

```
#include <stdio.h>

// MaxN is the maximum number of towers, MaxM is the maximum number of zones
const int MaxN = 20;
const int MaxM = 10;

// n is how many towers are planned, r is how many towers are actually built,
// m is number of zones, tot is current coverage, T is the best plan
int cases, n, m, r, tot, ans, T, i, j;

// sum[] is how many people are served for each tower, common[i][1] is the
// tower selected in zone i, common[i][2] is the number of users covered
int sum[MaxN], common[MaxM][2], S[1 << MaxN];


// read input and initialize all global variables and arrays
bool init() {
    int i, j, k, tmp;
    scanf("%d %d", &n, &r); if (n == 0 && r == 0) return false;
    for (i = 0; i < n; i++) scanf("%d", &sum[i]);

    scanf("%d", &m);
    for (i = 0; i < m; i++) {
        scanf("%d", &k);
        // for each tower in the zone, construct a corresponding bitmask
        common[i][0] = 0;
        for (j = 0; j < k; j++) {
            scanf("%d", &tmp); common[i][0] |= 1 << (tmp - 1);
        }
        scanf("%d", &common[i][1]);
    }

    return true;
}


int main() {
    cases = 0;
    // calculate the number of 1’s in a particular bitmask
    for (i = 0; i < (1<<20); i++) for (j = 0; j < 20; j++)
        if (i & (1<<j)) S[i]++;

    while (init()) {
        ans = -1;
        // the number of 1’s in the bitmask is the number of towers to build
        for (i = 0; i < (1<<n); i++) if (S[i] == r) {
            tot = 0;
            // calculate the number of users covered by these towers
            for (j = 0; j < n; j++) if (i & (1<<j)) tot += sum[j];
            // calculate the total coverage by number of towers in each zone
            for (j = 0; j < m; j++)     if (common[j][0] & i)
                tot -= common[j][1] * (S[common[j][0] & i] - 1);

            // if there's a greater coverage, update the optimal solution
            if (tot > ans) {
                ans = tot; T = i;
            } else if (tot == ans) {
                // if the total coverage is the same as the optimal solution
                // then if tower j in status i was not in the optimum, then
                // update the optimal solution, otherwise keep
                for (j = 0; j < n; j++) {
                    if (((1<<j) & i) && !((1<<j) & T)) {
                        T = i; break;
                    }
                    if (((1<<j) & T) && !((1<<j) & i)) break;
                }
            }
        }

        printf("Case Number %d\n", ++cases);
        printf("Number of Customers: %d\n", ans);
        printf("Locations recommended:");
        for (i = 0; i < n; i++) if ((1<<i) & T) printf(" %d", i+1);
        printf("\n\n");
    }

    return 0;
}
```

3 World Final 2005

3.1 A. Low Cost Air Travel

```
#include <stdio.h>

// several constraints, defined by the problem statements
const int Max_City = 200;
const int Max_Ticket = 20;
const int Max_TicketLength = 10;
const int Max_TripLength = 10;

// the struct to represent info about a discounted ticket
struct info {
    int cost, n;
    int city[Max_TicketLength];
};

// global variables needed for this program
int cases, NT, NI, i, j, k, l, tmp, trip, change, sum, n;

// city[] is the array of cities, list[i] denotes the i'th city on the current
// min-cost trip route, route[] is all tickets used by current route, ticket[]
// is the array of tickets
int city[Max_City], list[Max_TripLength], route[Max_TripLength * Max_City];
info ticket[Max_Ticket];

// f[n][m][0] is the minimum cost of traveling the first n cities, and finally
// arriving at city m, f[n][m][1] is the ticket number, f[n][m][2] is number
// of cities on this ticket, f[n][m][3] is the starting city of this ticket
int f[Max_TripLength][Max_City][4];


// find the index of city x, if not possible, update the array of cities
int find(int x) {
    int i;
    for (i = 0; i < sum; i++) if (city[i] == x) return i;
    city[sum] = x; sum++;
    return (sum - 1);
}


int main() {
    cases = 0;

    while (true) {
        scanf("%d", &NT); if (NT == 0) break;
        sum = 0;
        // read input about each type of ticket and the cities it covers
        for (i = 0; i < NT; i++) {
            scanf("%d %d", &ticket[i].cost, &ticket[i].n);
            for (j = 0; j < ticket[i].n; j++) {
                scanf("%d", &tmp); ticket[i].city[j] = find(tmp);
            }
        }
        scanf("%d", &NI);
        for (trip = 0; trip < NI; trip++) {
            // read input about the i'th minimum-cost route
            scanf("%d", &n);
            for (i = 0; i < n; i++) {
                scanf("%d", &tmp); list[i] = find(tmp);
            }
            // several initializations on the 3-d array f
            for (i = 0; i < n; i++) for (j = 0; j < sum; j++) f[i][j][0] = -1;
            for (i = 0; i < 4; i++) f[0][list[0]][i] = 0;

            // iterate over the cities the route covers so far
            for (i = 0; i < n; i++) {
                while (true) {
                    change = 0;
                    // for the city the traveler is in, if we already have the
                    // minimum cost to arrive at city j, then we can iterate
                    // over all the tickets starting from city j
                    for (j = 0; j < sum; j++) if (f[i][j][0] >= 0)
                        for (k = 0; k < NT; k++) if (ticket[k].city[0] == j)
                    {
                        // starting from the i+1'th city, iterate over all the
                        // cities reachable by the ticket k
                        tmp = i + 1;
                        for (l = 1; l < ticket[k].n; l++) {
                            // if ticket k can extend the route to tmp cities,
                            // and we have not got a minimum cost for such a
                            // plan, then we update the optimal solution
                            if (tmp < n && list[tmp] == ticket[k].city[l])
                                tmp++;

                            if(f[tmp-1][ticket[k].city[l]][0] < 0 ||
                                f[tmp-1][ticket[k].city[l]][0] >
                                f[i][j][0]+ticket[k].cost)
                            {
                                f[tmp - 1][ticket[k].city[l]][0] =
                                    f[i][j][0] + ticket[k].cost;
                                f[tmp - 1][ticket[k].city[l]][1] = k;
                                f[tmp - 1][ticket[k].city[l]][2] = i;
                                f[tmp - 1][ticket[k].city[l]][3] = j;
                                // indicate that some changes have been made
                                change++;
                            }
                        }
                    }

                    // if no updates have been made so far, breakj while loop
                    if (change == 0) break;
                }
            }

            printf("Case %d, Trip %d: Cost = %d\n", ++cases, trip+1,
                f[n - 1][list[n - 1]][0]);

            // reverse search the array to find the sequence of all tickets
            // and print them out at the end
            l = 0; i = n - 1; j = list[n - 1];
            while (i || j != list[0]) {
                // record all info about the cities involved in this ticket
                route[l] = f[i][j][1]; k = f[i][j][2]; j = f[i][j][3];
                i = k; l++;
            }
            printf(" Tickets used:");
            for (i = l-1; i >= 0; i--) printf(" %d",route[i] + 1);
            printf("\n");
        }
    }

    return 0;
}
```

3.2 B. Remember the A La Mode!

```
#include <stdio.h>
#include <memory.h>

// profit[][] is the profit of each combination, pies[] is the number of each
// pie, ices[] is number of each ice cream
int profit[55][55], pies[55], ices[55], pies_left[55], ices_left[55];

// these are arrays needed to complete the max flow algorithm
int profit_p[55], profit_i[55], combins[55][55], from_p[55], from_i[55];

// other necessary global variables
int n, m, i, j, k, new_profit, sum, cases, inf, updated;
double total_profit, x;
```

```
// compute the max flow min cost algorithm
void go() {
    // initialize the combinations, each has a flow of 0
    memset(combins, 0, sizeof(combins));
    // initialize the total profit
    total_profit = 0;

    // for each pie compute the augmenting path associated with it; each time
    // increment the total flow by 1
    for(i = 0; i < sum; i++) {
        // initialize all costs to infinity
        for(j = 0; j < m; j++) profit_i[j] = inf;
        for(j = 0; j < n; j++) {
            // initialize this pie node's source and costs
            from_p[j] = -1; profit_p[j] = inf;
            // if there are pies left, then this node's cost is set to zero
            if (pies_left[j]) profit_p[j] = 0;
        }

        // while there's an augmenting path, keep the solution updated
        updated = 1;
        while (updated) {
            updated = 0;
            // check each pie's node, if there's an augmenting path that pass
            // this point, then pie j and ice cream k can be combined
            for (j = 0; j < n; j++)     if (profit_p[j] > inf)
                for (k = 0; k < m; k++) if (profit[j][k] > 0)
            {
                // compute the new augmenting path's cost, and compare it w/
                // the original cost
                new_profit = profit_p[j] + profit[j][k];
                if( profit_i[k] == inf || new_profit < profit_i[k]) {
                    profit_i[k] = new_profit; from_i[k] = j;
                    updated = 1;
                }
            }
            // check each ice cream's node, similar to the pie case above
            for (j = 0; j < m; j++)     if (profit_i[j] > inf)
                for (k = 0; k < n; k++) if (combins[k][j])
            {
                new_profit = profit_i[j] - profit[k][j];
                if (profit_p[k] == inf || new_profit < profit_p[k]) {
                    profit_p[k] = new_profit; from_p[k] = j;
                    updated = 1;
                }
            }
        }

        // the index of ice cream corresponding to the minimum cost
        k = 0;
        // check each ice cream node's augmenting path costs; if smaller than
        // previous node's cost, then update the node index
        for (j = 1; j < m; j++)
            if (profit_i[k] == inf || !ices_left[k] ||
                profit_i[j] < profit_i[k] && ices_left[j]) k = j;

        // decrement the remaining ice creams, and update the total flow
        ices_left[k]--; total_profit += profit_i[k];

        // find out the augmenting path in this time step
        while (true) {
            // pie j is combined with ice cream k, increment the flow by one
            j = from_i[k]; combins[j][k]++;
            // if no pie j left, the break out of this loop
            if (from_p[j] < 0) break;
            // backtracking step, revert the flow back one step
            k = from_p[j]; combins[j][k]--;
        }

        // used pie j, the remaining value should be decremented by one
        pies_left[j]--;
    }
}


int main() {
    inf = -(1<<30);

    while (scanf("%d%d", &n, &m) && n) {
        sum = 0;
        // read input about pies
        for (i = 0; i < n; i++) {
            scanf("%d", &pies[i]);
            pies_left[i] = pies[i]; sum += pies_left[i];
        }
        // read input about ice creams
        for (i = 0; i < m; i++) {
            scanf("%d", &ices[i]);
            ices_left[i] = ices[i];
        }
        // read input about the possible profits
        for (i = 0; i < n; i++) for (j = 0; j < m; j++) {
            scanf("%lf", &x);
            profit[i][j] = int(x * 100 + 0.1);
        }

        // the main algorithm, this time for the min profit
        go();

        printf("Problem %d: %.2lf to ", ++cases, total_profit/100);
        // recover the original pies and ice creams info
        memcpy(pies_left, pies, sizeof(pies));
        memcpy(ices_left, ices, sizeof(ices));
        for (i = 0; i < n; i++) for (j = 0; j < m; j++)
            if (profit[i][j] > 0) profit[i][j] = 1001 - profit[i][j];

        // main algorithm again, this time for the max profit
        go();

        printf("%.2lf\n", (sum * 1001 - total_profit) / 100);
    }

    return 0;
}
```

3.3 C. Ars Longa

```
#include <stdio.h>
#include <memory.h>
#include <math.h>

// x[], y[] and z[] are coordinates of the balls, a[][] are tension forces on
// each ball, b[][] are the systems of equations
double x[101], y[101], z[101], a[300][101], b[300][101];

// other global variables necessary for the program
double eps, c; int n, m, r, i, j, k, l, t;


// use Gaussian elimination to determine if a system of equation has solution
bool ok() {
    l = 0;
    // eliminate the m variables, one by one
    for (i = m; i; i--) {
        k = l;
        // for the i'th variable, find the equation w/ greatest coefficients
        for (j = l+1; j < r; j++) if (fabs(b[j][i]) > fabs(b[k][i])) k = j;
        // if all equations have zero coefficients on this variable, skip it
        if (fabs(b[k][i]) < eps) continue;

        // change this equation (as above) to the l'th row in the system
        memcpy(x, b[l], sizeof(x));
        memcpy(b[l], b[k], sizeof(x));
        memcpy(b[k], x, sizeof(x));

        // eliminate all the i'th variable in the remaining equations below
        for (j = l+1; j < r; j++) {
            c = b[j][i] / b[l][i];
            for (k = 0; k < i; k++) b[j][k] -= c * b[l][k];
            b[j][i] = 0;
        }
        l++;
    }
    for (i = l; i < r; i++) if (fabs(b[i][0]) > eps) return false;
    return true;
}


int main() {
    eps = 0.1;

    while (scanf("%d%d", &n, &m) && n) {
        memset(a, 0, sizeof(a));
        for (i = 0; i < n; i++) scanf("%lf%lf%lf", &x[i], &y[i], &z[i]);

        for (i = 1; i <= m; i++) {
            // the indices should fit the array dimensions, so decrement by 1
            scanf("%d%d", &j, &k); j--; k--;
            // if ball j does not touch the ground (may fall down)
            if (z[j]) {
                // the tension forces on ball j in x, y, and z directions
                a[j*3][i] = x[k] - x[j];
                a[j*3+1][i] = y[k] - y[j];
                a[j*3+2][i] = z[k] - z[j];
            }
            // if ball k does not touch the ground (may fall down)
            if (z[k]) {
                // the tension forces on ball k in x, y, and z directions
                a[k*3][i] = x[j] - x[k];
                a[k*3+1][i] = y[j] - y[k];
                a[k*3+2][i] = z[j] - z[k];
            }
        }

        // record how many systems of equations are to be established
        r = 0;
        // for each ball, if it does not touch the ground (may fall), then we
        // establish a system of equations to calculate its equillibrium
        for (i = 0; i < n; i++) if (z[i] > 0) {
            // add an equation to the system in x, y, and z directions
            memcpy(b[r++], a[i*3], sizeof(x));
            memcpy(b[r++], a[i*3+1], sizeof(y));
            memcpy(b[r++], a[i*3+2], sizeof(z));
            // the combined force in z direction should be same as gravity
            b[r-1][0] = 1;
        }

        printf("Sculpture %d: ", ++t);
        // if no solution, then the system is not static
        if (!ok()) {
            printf("NON-STATIC\n"); continue;
        }
        r = 0;
        // check each ball again, to get the result of whether it's stable
        for (i = 0; i < n; i++) if (z[i] > 0) {
            memcpy(b[r++], a[i*3], sizeof(x));
            memcpy(b[r++], a[i*3+1], sizeof(y));
            memcpy(b[r++], a[i*3+2], sizeof(z));
            b[r-1][0] = sin(i*3+1);
            b[r-2][0] = sin(i*3+2);
            b[r-3][0] = sin(i*3+3);
        }
        if (!ok()) {
            printf("UNSTABLE\n"); continue;
        }
        printf("STABLE\n");
    }

    return 0;
}
```

3.4 D. Bipartite Numbers

```
#include <stdio.h>

// ones[] and tens[] are arrays used for computing the divisors of any given
// bipartite numbers, others are global variables
int ones[9999], tens[999], n, i, j, k, s, t, best_i, best_j, best_s, best_t;


// check if the current bipartite number is greater than n
bool ck() {
    int p, q, r;
    if (i > 5) return 1;
    p = s; r = t;
    for (q = 0; q < j; q++) p = p * 10 + s;
    for (q = 0; q < i-j; q++) p = p * 10;
    for (q = 1 ;q < i-j; q++) r = r * 10 + t;
    return p + r > n;
}


int main() {
    while(scanf("%d", &n) && n) {
        printf("%d: ", n);
        // base case: the public key is just 1, output bipartite number 10
        if (n == 1) {
            printf("1 1 1 0\n"); continue;
        }
        // initialize the first term in the arrays a and b
        ones[0] = 1; tens[0] = 1;
        // generate the array a: a[i+1] = (a[i]*10 + 1) % n
        for (i = 1; i < 9999; i++) ones[i] = (ones[i-1] * 10 + 1) % n;
        // generate the array b: b[i+1] = b[i] % n
        for (i = 1; i < 999; i++) tens[i] = tens[i-1] * 10 % n;

        // iterate over the total length of the bipartite number
        for (i = 1, best_s = 0; i < 9999; i++) {
            k = 0;
            // limit the length of the first portion, to improve efficiency
            if ((n % 10 == 0 || n % 25 == 0) && i > 11) k = i - 11;
            // if t and s are not equal and n divides the bipartite number,
            // and the number is greater than n, then this solution is good
            // and so we update the solution
            for (j = k; j < i; j++) for (s = 1; s < 10; s++)
                for (t = 0; t < (n%10 ? 10 : 1); t++)
                    if (t != s && (((long long)ones[j]) * tens[i-j] * s +
                        ones[i-j-1] * t) % n == 0 && ck() && (!best_s ||
                        s < best_s || s == best_s && j>best_j && s<best_t))
            {
                best_i = i; best_j = j; best_s = s; best_t = t;
            }
            // if we have found the solution, then no need to check longer
            // bipartite numbers any more
            if (best_s) break;
        }

        printf("%d %d %d %d\n", best_j+1, best_s, best_i-best_j, best_t);
    }

    return 0;
}
```

3.5 E. Bit Compressor

```
#include <stdio.h>

// a[] is the compressed data, and other necessary global variables
char a[44]; int cases, inf, n, m, len, total, ones, solutions;


// check every possible decompression method
void go(int i) {
    int j, k, tmp_len, all_ones;
    // if the decompressed message is too long, or if the number of 1's is too
    // large, or there has already been more than one solutions
    if (total > n || ones > m || solutions > 1) return;
    // if the decompression is finished, check its validity and # solutions
    if (i >= len) {
        if (total + ones == n + m) solutions++; return;
    }

    // iterate over all bits that start with 1, try to decompress the message
    // at any possible length
    if (a[i] == '1') for (j = i, k = 0, all_ones = 1; j < len; j++) {
        // calculate how many bits of 1's current portion is representing
        if (k < inf) k = k * 2 + a[j] - '0';
        // mark about whether these few bits are all 1's
        all_ones &= a[j] - '0';

        if (a[j+1] != '1') {
            // if the length of these bits are greater than 2, then enter next
            // level of recursion and accumulate number of 1's
            if (k > 2) {
                total += k; ones += k;
                go(j+1);
                // the backtracking step: return to last state after go(j+1)
                total -= k; ones -= k;
            }
            // if these bits have a length <= 2, then must be part of original
            // message, enter next recursion accordingly
            if (all_ones && j-i < 2) {
                tmp_len = j - i + 1; total += tmp_len; ones += tmp_len;
                go(j+1);
                // the backtracking step: return to last state after go(j+1)
                total -= tmp_len; ones -= tmp_len;
            }
        }
    } else {
        // if the current bit is 0, then must be part of original message
        total++; go(i+1); total--;
    }
}


int main() {
    inf = 99999;

    for (cases = 1; scanf("%d%d\n", &n, &m) && n+m; cases++) {
        scanf("%s", &a);

        // increment the value of "len", denoting the length of the bits
        for (len = 0; a[len] > ' '; len++);
        total = 0; ones = 0; solutions = 0;
        // recursive backtracking algorithm to decompress the message
        go(0);

        printf("Case #%d: %s\n", cases, solutions ? solutions-1 ?
            "NOT UNIQUE" : "YES" : "NO");
    }

    return 0;
}
```

3.6 F. Building a Clock

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

#define PB push_back

// the gear with name s has T[s] teeth
map<char, int> T;

// the best solution is stored in the following global variables
string AnswerM , AnswerH; int AnswerShaft , AnswerGear;

// other necessary global variables for this program
int N, R; bool used[7]; char ch[7];


// given a plan of a clock hand denoted by "s", compute its rotation velocity
int Time(string s) {
    int Rs = R, Rt = 1, lasT;
    // record the number of teeth of the last gear in the rotation
    for (int i = 1; i < s.size(); i++) {
        if (s[i] == ’-’);
        else if (s[i-1] == ’-’) {
            Rs *= -lasT; Rt *= T[s[i]]; lasT = T[s[i]];
        } else lasT = T[s[i]];
    }
    // if two gears match with each other, than return their velocity ratio
    if (Rs % Rt == 0) return Rs/Rt;
    else return 0;
}


// update the best solution to build the clock
void Refresh(string sm , string sh , int shaft , int gear) {
    // if the minute hand currently does not have a best solution, use current
    // solution as the best one
    if (AnswerM == "X") {
        AnswerM = sm; AnswerH = sh; AnswerShaft = shaft; AnswerGear = gear;
        return;
    }
    // initialize the best common string of the best plan and the current plan
    string sold , snew; sold.clear(); snew.clear();

    // store the gear that drive the minute hand and the hour hand (best plan)
    // into the public string, denoted by "sold"
    for (int i = 0; i < AnswerM.size(); i++)
        if (AnswerM[i] != ’*’ && AnswerM[i] != ’-’) sold.PB(AnswerM[i]);
    for (int i = 0; i < AnswerH.size(); i++)
        if (AnswerH[i] != ’*’ && AnswerH[i] != ’-’) sold.PB(AnswerH[i]);

    // store the gears that drive the minute hand and the hour hand (current
    // plan) into the public string, denoted by "snew"
    for (int i = 0; i < sm.size(); i++)
        if (sm[i] != ’*’ && sm[i] != ’-’) snew.PB(sm[i]);
    for (int i = 0; i < sh.size(); i++)
        if (sh[i] != ’*’ && sh[i] != ’-’) snew.PB(sh[i]);

    // change the best plan according to number of shafts, of gears, and the
    // lexographical order of the plan
    if (shaft < AnswerShaft) {
        AnswerM = sm; AnswerH = sh; AnswerShaft = shaft; AnswerGear = gear;
    } else if (shaft == AnswerShaft && gear < AnswerGear) {
        AnswerM = sm; AnswerH = sh; AnswerShaft = shaft; AnswerGear = gear;
    } else if (shaft == AnswerShaft && gear == AnswerGear && snew < sold) {
        AnswerM = sm; AnswerH = sh; AnswerShaft = shaft; AnswerGear = gear;
    }
}


// depth first search over the minute and the hour hand, using the number of
// shafts as the first keyword, the number of gears as the second keyword
void Dfs(string sm, string sh, int shaft, int gear) {
    // there should not be a gear in the third position
    if (sm.size() == 3 || sh.size() == 3) return;
    // there should not be three consecutive gears in the minute hand
    if (sm.size() >= 4 && sm[sm.size()-1] != ’-’ &&
        sm[sm.size()-2] != ’-’ && sm[sm.size()-3] != ’-’) return;
    // there should not be three consecutive gears in the hour hand
    if (sh.size() >= 4 && sh[sh.size()-1] != ’-’ &&
        sh[sh.size()-2] != ’-’ && sh[sh.size()-3] != ’-’) return;
    // if the current plan is worse than the best plan, quit directly
    if (shaft > AnswerShaft || (shaft == AnswerShaft && gear > AnswerGear))
        return;
    if (Time(sm) == 24 && Time(sh) == 2) {
        // if the minute hand rotates once every hour, the hour hand rotates
        // twice every day, than update the best plan
        Refresh(sm, sh, shaft, gear);
    } else if (Time(sm) == 24) {
        // if the minute hand is correct, then handle the hour hand
        if (sm.size() >= 3 && sm[sm.size()-2] != ’-’) return;
        // iterate over each unused gear
        for (int i = 1; i <= N; i++) if (!used[i]) {
            used[i] = true;
            // add this gear on the same shaft
            Dfs(sm, sh+ch[i], shaft, gear+1);
            // add a new shaft, and connect it to the last gear
            if (sh.size() > 1) Dfs(sm, sh+’-’+ch[i], shaft+1, gear+1);
            // backtracking step, revert back to the last state
            used[i] = false;
        }
    } else {
        // the minute hand is incorrect, then handle it, similar to last case
        for (int i = 1; i <= N; i++) if (!used[i]) {
            used[i] = true;
            Dfs(sm+ch[i], sh, shaft, gear+1);
            if (sm.size() > 1) Dfs(sm+’-’+ch[i], sh, shaft+1, gear+1);
            used[i] = false;
        }
    }
}


// depth first search over common part, using the number of shafts as the 1st
// keyword, the number of gears as the second keyword
void Dfs(string s, int shaft, int gear) {
    // there should not be two common shafts on the original shaft
    if (s.size() == 3) return;
    // there should not be three consecutive shafts on the common path
    if (s.size() >= 4 && s[s.size()-1] != ’-’ &&
        s[s.size()-2] != ’-’ && s[s.size()-3] != ’-’) return;
    // if there are already too much shafts and gears
    if (shaft > AnswerShaft || (shaft == AnswerShaft && gear > AnswerGear))
        return;

    // depth first search, similar to the case in the last method
    Dfs(s, s, shaft, gear);
    for (int i = 1; i <= N; i++) if (!used[i]) {
        used[i] = true;
        Dfs(s+ch[i], shaft, gear+1);
        if (s.size() > 1) Dfs(s+’-’+ch[i], shaft+1, gear+1);
        used[i] = false;
    }
}


int main() {
    int Case = 0;

    while (cin >> N && N) {
        if (Case > 0) cout << "\n";
        cin >> R;
        int x;
        // initialize all gears to be unused, then read input
        for (int i = 1; i <= N; i++) used[i] = false;
        for (int i = 1; i <= N; i++) {
            cin >> ch[i] >> x; T[ch[i]] = x;
        }
        // initialize the best plan, minute hand and hour hand shoube be empty
        // number of used shafts and gears should be infinity
        AnswerM = "X"; AnswerH = "X"; AnswerShaft = AnswerGear = 100000000;

        // calculate the best plan from the first shaft
        Dfs("*", 1, 0);

        if (AnswerM == "X") cout << "Trial " << ++Case << " IS IMPOSSIBLE\n";
        else cout << "Trial " << ++Case << "\nMinutes: " << AnswerM <<
            "\nHours: " << AnswerH <<"\n";
    }

    return 0;
}
```

3.7 G. Pilgrimage

```
#include <stdio.h>
#include <string.h>

// the maximum amount of people in the group
const int MaxP = 2000 * 50;
```

```
// the keyword of an operation
char s[20];

// record the list of operation, list[i][0] is the type of the i’th operation,
// and list[i][1] is the operation count
int list[50][2];

// the number of times the group satisfies the requirements of PAY
int a[MaxP+1];

// other necessary global variables for the program
int n, m, i, j, k, P, sum, delta, min;


int main() {
    while (scanf("%d", &n) && n) {
        // initialize the number of PAY operations
        sum = 0; P = 0;
        for (i = 0; i < n; i++) {
            scanf("%s %d", s, &m);
            // ignore operations of type COLLECT
            if (s[0] == ’C’) continue;
            // accumulate number of PAY operations, and set value of list[][]
            if (s[0] == ’P’) P += m;
            else {
                if (P && sum) {
                    list[sum][0] = 2; list[sum][1] = P; sum ++;
                }
                // only PAY operations between IN and OUT are valid
                if (s[0] == ’I’) list[sum][0] = 0;
                else list[sum][0] = 1; list[sum][1] = m;
                sum ++; P = 0;
            }
        }

        // a[x] = k means a group of x people can satisfy k PAY operations, in
        // this way, the final answer should be number of PAY for all people
        memset(a, 0, sizeof(a));
        delta = 0; min = 0; P = 0;
        for (i = 0; i < sum; i++) {
            // delta is the change of group size and min records the change of
            // group size when it reaches the lowest
            if (list[i][0] == 0) delta += list[i][1];
            if (list[i][0] == 1) delta -= list[i][1];
            if (min > delta) min = delta;
            if (list[i][0] != 2) continue;
            // set all factors Y in PAY operatuon X to initial group size Z
            P++;
            for (j = 1; j <= list[i][1]; j++)
                if (j > delta && list[i][1] % j == 0) a[j-delta]++;
        }

        // if no valid P opeeration, output the lower bound of group size, or
        // otherwise output the last group size satisfying all requirements
        if (P == 0) printf("SIZE >= %d\n", 1 - min);
        else {
            k = 0;
            for (i = 1-min; i <= MaxP; i++) if (a[i] == P) {
                if (k) printf(" ");
                k = i; printf("%d", k);
            }
            // if no valid group size at any time, then output "IMPOSSIBLE"
            if (!k) printf("IMPOSSIBLE");
            printf("\n");
        }
    }

    return 0;
}
```

3.8 H. Pockets

```
#include <stdio.h>
#include <string.h>

// the struct to represent info about a crease, sign means whether this is an
// external (1) crease or not (0), if it’s external, then sum counts how many
// pieces of paper are visible, if it’s internal, then up, down count how many
// pieces of paper are visible from up and down
struct info     {
    int sign, up, down, uCover, dCover, sum;
};

// the maximum number of crease along a single direction
const int MaxN = 64;

// hor[][] records horizontal creases, and ver[][] records vertical creases,
// Nhor[] and Nver[] records the information after a fold
info hor[MaxN+2][MaxN+1], ver[MaxN+1][MaxN+2];
info Nhor[MaxN+2][MaxN+1], Nver[MaxN+1][MaxN+2];

// Lhor[i] = k means the horizontal crease with index i is the k’th crease in
// the order, same idea for Lver
int Lhor[MaxN+1], Lver[MaxN+1];

// other global variables necessary for the program
int Cases, N, n, m, K, ans; char dir;


// initialize all variables
void init()     {
    int i;
    memset(hor, 0, sizeof(hor));
    memset(ver, 0, sizeof(ver));
    // mark all creases as external creases, and visible pieces equal to 1
    for (i = 0; i < n; i++) {
        hor[0][i].sign = hor[n][i].sign = 1;
        hor[0][i].sum = hor[n][i].sum = 1;
        ver[i][0].sign = ver[i][n].sign = 1;
        ver[i][0].sum = ver[i][n].sum = 1;
    }
    // the crease indices are ordered from up to bottom and from left to right
    for (i = 0; i <= n; i++) Lhor[i] = Lver[i] = i;
}


// fold crease x along line in the direction dir, and calculates the index of
// this particular crease in the new picture
int get(int x, int s, int line, char dir) {
    if (dir == ’D’ || dir == ’R’) {
        return x-line > 0 ? x-line : line-x;
    } else if (line + line >= s) {
        if (x <= line) return x;
        return line - (x - line);
    } else {
        if (x >= line) return s - x;
        return (s - line) - line + x;
    }
}


// simulate a time step of folding the paper
void work() {
    int i, j, k, nNew, mNew, x, y; int line; char dir;
    scanf("%d%c", &line, &dir);
    while (dir!=’U’ && dir!=’D’ && dir!=’R’ && dir!=’L’) scanf("%c", &dir);

    // determine which line the crease follows, calculate the new paper’s size
    // and re-index all the creases after the fold
    if (dir == ’U’ || dir == ’D’) {
        line = Lhor[line]; mNew = m;
        if (line > n - line) nNew = line;
        else nNew = n - line;
        for (i = 0; i <= N; i++) Lhor[i] = get(Lhor[i], n, line, dir);
    } else {
        line = Lver[line]; nNew = n;
        if (line > m - line) mNew = line;
        else mNew = m - line;
        for (i = 0; i <= N; i++) Lver[i] = get(Lver[i], m, line, dir);
    }

    // record which creases are external in the new picture
    memset(Nhor, 0, sizeof(Nhor)); memset(Nver, 0, sizeof(Nver));
    for (i = 0; i < nNew; i++) Nver[i][0].sign = Nver[i][mNew].sign = 1;
    for (i = 0; i < mNew; i++) Nhor[0][i].sign = Nhor[nNew][i].sign = 1;

    // for each fold, relate it with the last picture, and accumulate values
    if (dir == ’U’) for (i = 0; i < mNew; i++) Nhor[nNew][i].sum = 1;
    if (dir == ’D’) for (i = 0; i < mNew; i++) Nhor[0][i].sum = 1;
    if (dir == ’L’) for (i = 0; i < nNew; i++) Nver[i][mNew].sum = 1;
    if (dir == ’R’) for (i = 0; i < nNew; i++) Nver[i][0].sum = 1;

    // handle all the horizontal creases
    for (i = 0; i <= n; i++) for (j = 0; j < m; j++) {
        // determine new indices for crease (i, j)-(i, j+1) in old picture
        x = i; y = j;
        if (dir == ’U’ || dir == ’D’) x = get(x, n, line, dir);
        else y = get(y, m, line, dir);
        // creases (x, y)-(x, y+1) could have coordinates (x’, y’)-(x’, y’-1)
        // after folding the paper
        if ((dir == ’R’ && j < line) || (dir == ’L’ && j >= line)) y--;

        // some creases change to external creases after the fold
        if (Nhor[x][y].sign) {
            // if already external, then accumulate the total number, if it
            // is internal, then turn the bottom piece to the external side
            if (hor[i][j].sign) Nhor[x][y].sum += hor[i][j].sum;
            else Nhor[x][y].sum += hor[i][j].down;
        } else {
            // some external crease changes to internal
            if (hor[i][j].sign) {
                // divide into two cases: whether the piece falls into the
                // upper or the lower part of the new crease
                if ((i < line && dir == 'U') || (i >= line && dir == 'D'))
                    Nhor[x][y].down += hor[i][j].sum;
                else Nhor[x][y].up += hor[i][j].sum;
            } else {
                // preserve the number of visible pieces, and record that the
                // crease's lower part has been covered
                if ((i < line && dir == 'U') || (i >= line && dir == 'D') ||
                    (j < line && dir == 'L') || (j >= line && dir == 'R'))
                {
                    // if this crease isn't covered, then we know the visible
                    // pieces are still visible, and accumulate the 2 values
                    Nhor[x][y].down = hor[i][j].down; Nhor[x][y].dCover = 1;
                    if (!Nhor[x][y].uCover) Nhor[x][y].up += hor[i][j].up;
                } else {
                    // similar to above case, handle the upper part
                    Nhor[x][y].up = hor[i][j].down; Nhor[x][y].uCover = 1;
                    if (!Nhor[x][y].dCover) Nhor[x][y].down += hor[i][j].up;
                }
            }
        }
    }

    // handle all vertical creases, refer to the above loop for explanations
    for (i = 0; i < n; i++) for (j = 0; j <= m; j++) {
        x = i; y = j;
        if (dir == 'U' || dir == 'D') x = get(x, n, line, dir);
        else y = get(y, m, line, dir);
        if ((dir == 'U' && i >= line) || (dir == 'D' && i < line)) x--;

        if (Nver[x][y].sign) {
            if (ver[i][j].sign) Nver[x][y].sum += ver[i][j].sum;
            else Nver[x][y].sum += ver[i][j].down;
        } else {
            if (ver[i][j].sign) {
                if ((j < line && dir == 'L') || (j >= line && dir=='R'))
                    Nver[x][y].down += ver[i][j].sum;
                else Nver[x][y].up += ver[i][j].sum;
            } else {
                if ((i < line && dir == 'U') || (i >= line && dir == 'D') ||
                    (j < line && dir == 'L') || (j >= line && dir == 'R'))
                {
                    Nver[x][y].down = ver[i][j].down; Nver[x][y].dCover = 1;
                    if (!Nver[x][y].uCover) Nver[x][y].up += ver[i][j].up;
                } else {
                    Nver[x][y].up = ver[i][j].down; Nver[x][y].uCover = 1;
                    if (!Nver[x][y].dCover) Nver[x][y].down += ver[i][j].up;
                }
            }
        }
    }

    // replace old data with the data after folding the paper
    memcpy(hor, Nhor, sizeof(hor));
    memcpy(ver, Nver, sizeof(ver));
    n = nNew; m = mNew;
}


int main() {
    while (scanf("%d %d", &n, &K) && (n || K)) {
        n++; N = m = n;

        init();
        // simulate every time step of folding the paper
        for (; K; K--) work();  //                Ł

        // accumulate number of all visible pieces and print the result
        ans = hor[0][0].sum - 1 + hor[1][0].sum - 1 + ver[0][0].sum - 1 +
            ver[0][1].sum - 1;
        printf("Case %d: %d pockets\n", ++ Cases, ans);
    }

    return 0;
}
```

3.9 I. Degrees of Separation

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;

// constraints, and necessary global variables for the program
const int MaxP = 50;
int P, R, cases, i, j, k, sum, ans;

// name[] is the list of names, and sum denotes its length
string name[MaxP], s1, s2;

// adjacency matrix to represent the graph
int a[MaxP][MaxP];


// given a name and return its node index, if the name has not appeared before
// then update the list of names
int find(string s) {
    int i;
    for (i = 0; i < sum; i++) if (name[i] == s) return i;
    name[sum] = s; sum++; return sum - 1;
}


// read input and initialize all variables to initial values
bool init() {
    scanf("%d %d", &P, &R);
    if (P == 0 && R == 0) return false;

    sum = 0; memset(a, 0, sizeof(a));
    // constructing the adjacency matrix
    for (i = 0; i < R; i++) {
        cin >> s1 >> s2;
        j = find(s1); k = find(s2); a[j][k] = a[k][j] = 1;
    }
    return true;
}


int main() {
    cases = 0;

    while (init()) {
        // use floyd-warshall algorithm to compute the shortest path between
        // any pair of nodes in the graph
        for (k = 0; k < P; k++)
            for (i = 0; i < P; i++) for (j = 0; j < P; j++)
                if (a[i][k] && a[k][j])
                    if (a[i][j] == 0 || a[i][j] > a[i][k] + a[k][j])
                        a[j][i] = a[i][j] = a[i][k] + a[k][j];

        ans = 0;
        // if there are node pairs that can never be reached, then the graph
        // is not connected, or the max degree of separation is the greatest
        // value of length of all shortest paths
        for (i = 0; i < P; i++) for (j = i+1; j < P; j++) {
            if (a[i][j] == 0) ans = P + 10;
            else if (a[i][j] > ans) ans = a[i][j];
        }

        if (ans > P) printf("Network %d: DISCONNECTED\n\n", ++cases);
        else printf("Network %d: %d\n\n", ++cases, ans);
    }

    return 0;
}
```

3.10 J. Routing

```
#include <stdio.h>

// several constraints, defined by the problem statements
const int MaxN = 100;
const int Max = 1000;

// dis[x][y] is the shortest path from x to y, tr[] is the segment tree, used
// for getting the smallest answer and book-keeping when answer is updated,
// ans[x][y] means there are at least this many nodes in paths 1->x and 1->y
int dis[MaxN][MaxN], tr[MaxN * MaxN * 3], ans[MaxN][MaxN];

// other necessary global variables for this program
int Cases, n, m, i, j, k, x, y;


// from the root node n and its corresponding interval, set all values to Max
// in the segment tree
void Make(int n, int x, int y) {
    tr[n] = Max;
    if (x == y) return;
    Make(n + n, x, (x + y) / 2);
    Make(n + n + 1, (x + y) / 2 + 1, y);
}
```

```
// from the root node n, update node ch's value to data and book-keeping
void Change(int n, int x, int y, int ch, int data) {
    if (x == y) {
        tr[n] = data;
    } else if ((x+y) / 2 >= ch) {
        Change(n + n, x, (x + y) / 2, ch, data);
        if (tr[n+n] < tr[n+n+1]) tr[n] = tr[n + n];
        else tr[n] = tr[n + n + 1];
    } else {
        Change(n + n + 1, (x + y) / 2 + 1, y, ch, data);
        if (tr[n+n] < tr[n+n+1]) tr[n] = tr[n + n];
        else tr[n] = tr[n + n + 1];
    }
}


// get the position of the min value in the segment tree
int Get_Min(int n, int x, int y) {
    if (x == y) return x;
    if (tr[n+n] < tr[n+n+1]) return Get_Min(n + n, x, (x + y) / 2);
    else return Get_Min(n + n + 1, (x + y) / 2 + 1, y);
}


int main() {
    Cases = 0;

    while (scanf("%d %d", &n, &m) && n) {
        // initialize the shortest path lengths, arrays index is 0..(n-1)
        for (i = 0; i < n; i++) for (j = 0; j < n; j++) dis[i][j] = Max;
        // construct the adjacency matrix
        for (i = 0; i < m; i++) {
            scanf("%d %d", &x, &y); dis[x - 1][y - 1] = 1;
        }
        // any shortest path length from and to the same node is 0
        for (i = 0; i < n; i++) dis[i][i] = 0;

        // use floyd-warshall algorithm to compute the shortest path length
        // for any pair of nodes
        for (k = 0; k < n; k++)
            for (i = 0; i < n; i++) for (j = 0; j < n; j++)
                if (dis[i][j] > dis[i][k] + dis[k][j])
                    dis[i][j] = dis[i][k] + dis[k][j];

        // initialize the segment tree and change the value of first index
        Make(1, 0, n * n - 1);
        Change(1, 0, n * n - 1, 0, 1);
        // ans[0][0] = 1other values of ans[][] on the diagonal should be Max
        for (i = 0; i < n; i++) for (j = 0; j < n; j++) ans[i][j] = Max;
        ans[0][0] = 1;

        while (tr[1] < Max) {
            // get the interval corresponding to the min value in the tree
            x = Get_Min(1, 0, n * n - 1);
            y = x % n; x /= n;

            // case 3: x and y are all repeated points, then exit the loop
            if (x == 1 && y == 1) break;

            if (x != y && ans[y][x] <= Max &&
                ans[x][y] + dis[x][y] - 1 < ans[y][x])
            {
                ans[y][x] = ans[x][y] + dis[x][y] - 1;
                Change(1, 0, n * n - 1, y * n + x, ans[y][x]);
            }

            // iterate over all points in the interval [x, y], not equal
            for (i = 0; i < n; i++) if (i != x && i != y) {
                // case 1: the next point of x is not repeated
                if (dis[x][i] == 1 && ans[i][y] <= Max &&
                    ans[x][y] + 1 < ans[i][y])
                {
                    ans[i][y] = ans[x][y] + 1;
                    Change(1, 0, n * n - 1, i * n + y, ans[i][y]);
                }

                // case 2: the previous point of y is not repeated
                if (dis[i][y] == 1 && ans[x][i] <= Max &&
                    ans[x][y] + 1 < ans[x][i])
                {
                    ans[x][i] = ans[x][y] + 1;
                    Change(1, 0, n * n - 1, x * n + i, ans[x][i]);
                }
            }

            // case 4: x is repeated and is also the previous point of y
            if (dis[x][y] == 1 && ans[y][y] <= Max && ans[x][y] < ans[y][y]) {
                ans[y][y] = ans[x][y];
                Change(1, 0, n * n - 1, y * n + y, ans[x][y]);
            }

            // case 5: y is repeated and is also the next point of x
            if (dis[x][y] == 1 && ans[x][x] <= Max && ans[x][y] < ans[x][x]) {
                ans[x][x] = ans[x][y];
                Change(1, 0, n * n - 1, x * n + x, ans[x][y]);
            }

            // mark the values of ans already used in the recursion
            ans[x][y] = Max + 1;
            Change(1, 0, n * n - 1, x * n + y, ans[x][y]);
        }

        printf("Network %d\n", ++Cases);
        if (ans[1][1] == Max) printf("Impossible\n\n");
        else printf("Minimum number of nodes = %d\n\n", ans[1][1]);
    }

    return 0;
}
```