Cleaning the Room

PREREQUISITES: Implementation

Quick Explanation:

Find the length of the biggest pencil which can fit in the box. Check for each pencil, if the length of the pencil is smaller than or equal to the length of biggest pencil, then it will fit in, otherwise it won't.

Detailed Explanation:

The question is simple and you only need to follow the instructions and implement it. We have been provided with 2 dimensions of the box. Using these dimensions, we need to find length of the largest pencil that can be fit-in the box.
The length of the largest or longest pencil that can fit-in can be calculated using Pythagoras theorem. Then, we need to check for each pencil's length. If it is smaller than or equal to pencil's length, then print "DA", otherwise print "NE".

Time Complexity: O(N), where N is the number of queries or number of pencils on the floor.

Editorialist's code:

```
import java.util.Scanner;

public class Main {
	public static void main(String args[]) {
		Scanner scn = new Scanner(System.in);


		int numberofpencils = scn.nextInt(); // number of pencils
		int widthofbox = scn.nextInt(); // width of box
		int heightofbox = scn.nextInt(); // height of box


		// calculating the length of longest pencil
		int longestpencil = (int) Math.sqrt(widthofbox * widthofbox
+ heightofbox * heightofbox);

		// for each query, I am checking if the length of the
pencil is smaller than or
		// equal to the length of
		// longest pencil that can fit in the box.
		for (int i = 0; i < numberofpencils; i++) {
			int query = scn.nextInt();
			if (query <= longestpencil) {
				System.out.println("DA");
			} else {
				System.out.println("NE");
			}
		}
	}
}
```

Helping the Granfather

PREREQUISITES: Implementation, 2d-Arrays

Quick and Detailed Explanation:

Each element of the input 2D Array has to be magnified and same element has to be kept in 2D array of size ZR, ZC. So, the magnified 2D array is of the size R*ZR, C*ZC.

This can be easily done by making a new 2D array of size R*ZR, C*ZC and the element at i, j of the new 2D array is the element at position i/ZR, j/ZC of the input 2D array.

Time Complexity: O(R*ZR*C*ZC), where the symbols have their usual meaning, as explained in the question.

Editorialist's code:

```
import java.util.Scanner;

public class Main {

	public static void main(String[] args) {
		Scanner scn=new Scanner(System.in);
		int row=scn.nextInt();
		int col=scn.nextInt();
		int zr=scn.nextInt();
		int zc=scn.nextInt();

		char[][] char_array=new char[row][col];
		for(int i=0; i<row; i++) {
			String str=scn.next();
			char_array[i]=str.toCharArray();
		}

		for(int i=0; i<row*zr; i++) {
			for(int j=0; j<col*zc; j++) {
				System.out.print(char_array[i/zr][j/zc]);
			}
			System.out.println();
		}
	}

}
```

Irregular Pairs

PREREQUISITES: Prefix sum, DP

Quick Explanation:

Find the answers of n-1 for 0<= j (number of irregular pairs) <=k

and use those answers for getting answers for n for 0<=j<=k using prefix sum and difference arrays (+1 -1 for range update).

Detailed Explanation:

The solution is similar to how we prove theorems/formulas using PMI (principle mathematical induction)

let a 2D array be a[n][k+1] where a[i][j] stores the number of permutations having 1 to i+1 element (both inclusive) and number of irregular pairs=j

Base case n=1: its obvious that a[0][0]=1 and a[0][j]=0 for j>0

Now use answers of n-1 elements for finding n and hence find all the answers for i=1 to n using previous one.

Each new element (nth element) (assume that its greater than all other elements present in array previously) can add 0 to n-1 irregular pairs based on position at which we insert it

Hence just add value of

a[i-1][j] to a[i][j], a[i][j+1], ..., a[i][j+n-1] for all j

Note: obviously don't add values beyond the range of size of array.

Time Complexity: O(n*k).

Editorialist's code:

```
#include<bits/stdc++.h>
using namespace std;

#define ll long long int
#define FIO ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0)
#define mod 1000000007

int add(int x, int y){
    return (x+y)%mod;
}

int sub(int x,int y){
    return ((x-y)%mod+mod)%mod;
}

int main()
{
    FIO;
    int t,n,k,i,j;
    cin >> n >> k;
```

```
    int a[n][k+1];

    memset(a[0],0,sizeof(a[0]));

    for(i=1;i<n;i++)
        memset(a[i],0,sizeof(a[0]));

    a[0][0]=1;

    for(i=1;i<n;i++){
        for(j=0;j<=k;j++){
            a[i][j]=add(a[i][j],a[i-1][j]);
            if(j+i<k)
                a[i][j+i+1]=sub(a[i][j+i+1],a[i-1][j]);
        }
        partial_sum(a[i],a[i]+k+1,a[i],add);
    }

    cout << a[n-1][k] << endl;

        return 0;
}
```

Cross Country Race

PREREQUISITES: DFS, topological sort, DP

Quick Explanation:

Use DFS to check if there exist an infinite loop(path) between node 1 and 2. Find the topological sort of the nodes which are between path of both nodes.

Start from the last element of sorted list of nodes and use DP to find the answer.

Detailed Explanation:

For checking if there exist an infinite loop between both nodes.

We mark all the nodes which can be visited from node 1 and also check all the nodes which can visit 2. And then find their intersection and then check if the path containing only these nodes have any infinite loop or not.

Checking if there is a cycle or not is very simple and can be done using two different boolean arrays where one shows if the node is visited or not and one shows if the node is parent (can be visited from current node or not) of current node or not.

Then use topological sort.

Then starting with last element add the number of ways we can visit current node from their neighbours (to which current node points).

Check editorialist's code to understand it better.

Also use BIG INTEGER Library and don't forget to print only last 9 digits (don't print extra 0's if value is smaller than 9 digits but print 0's if value is larger).

Time Complexity: O(n + m). (number of nodes+ number of edges)

Editorialist's code:

```
#include<bits/stdc++.h>
using namespace std;

#define ll long long int
#define FIO ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0)
#define mod 1000000000

bool isCyclic(vector<int> v[],int node,bool visited[],bool visited1[],bool
visited2[],bool visited3[]){
    if(visited2[node] && visited3[node]){
        visited[node]=true;
        visited1[node]=true;

        for(int x:v[node])
            if((!visited[x] &&
isCyclic(v,x,visited,visited1,visited2,visited3)) || visited1[x])
                return true;
        visited1[node]=false;
    }
    return false;
```

```
}

void dfs(vector<int> v[],int node,bool visited[]){
    visited[node]=true;

    for(int x:v[node])
        if(!visited[x])
            dfs(v,x,visited);
}

void dfs2(vector<int> v[],int node,bool visited[],bool visited1[]){
    if(visited1[node]){
        visited[node]=true;

        for(int x:v[node])
            if(!visited[x])
                dfs2(v,x,visited,visited1);
    }
}

string add(string str1, string str2) {
    if (str1.length() > str2.length())
        swap(str1, str2);

    string str = "";

    int n1 = str1.length(), n2 = str2.length();

    reverse(str1.begin(), str1.end());
    reverse(str2.begin(), str2.end());

    int carry = 0;
    for (int i=0; i<n1; i++)
    {
        int sum = ((str1[i]-'0')+(str2[i]-'0')+carry);
        str.push_back(sum%10 + '0');

        carry = sum/10;
    }

    for (int i=n1; i<n2; i++)
    {
        int sum = ((str2[i]-'0')+carry);
        str.push_back(sum%10 + '0');
        carry = sum/10;
    }

    if (carry)
        str.push_back(carry+'0');

    reverse(str.begin(), str.end());
    return str;
}

int main()
{
    FIO;
    int n,m,i,j,k;
    cin >> n >> m;

    int cnt[n]={0};
```

```
    string dp[n];
    vector<int> v[n],v2,v3[n];
    bool visited[n]={0},visited1[n]={0},visited2[n]={0},visited3[n]={0};

    for(i=0;i<n;i++)
        dp[i]="0";

    while(m--){
        cin >> j >> k;
        v[j-1].push_back(k-1);
        v3[k-1].push_back(j-1);
        cnt[k-1]++;
    }

    dfs(v,0,visited2);
    dfs2(v3,1,visited3,visited2);



    // inf or not

    if(isCyclic(v,0,visited,visited1,visited2,visited3)){
        cout << "inf";
        return 0;
    }

    for(i=0;i<n;i++)
        if(!visited[i])
            for(int x: v[i])
                cnt[x]--;

    // topological sort
    queue<int> q;

    for (i=0;i<n;i++)
                if(cnt[i]==0 && visited2[i] && visited3[i])
                        q.push(i);

    while(!q.empty()){
        j=q.front();
        q.pop();
        v2.push_back(j);

        for(i=0;i<v[j].size();i++){
            cnt[v[j][i]]--;
            if(cnt[v[j][i]]==0 && visited2[v[j][i]] && visited3[v[j][i]])
                q.push(v[j][i]);
        }
    }

    dp[1]="1";
    bool flag=false;

    for (i=v2.size()-1;i>=0;i--)
                for (j=0;j<v[v2[i]].size();j++)
            if(visited2[v2[i]] && visited2[v[v2[i]][j]] && visited3[v2[i]]
&& visited3[v[v2[i]][j]])
                            dp[v2[i]]= add(dp[v2[i]],dp[v[v2[i]][j]]);

    if(dp[0].length()>9){
        for(i=dp[0].length()-9;i<dp[0].length();i++)
```

```
                cout << dp[0][i];
    }else
        cout << dp[0];

        return 0;
}
```

WarRoom

PREREQUISITES: Geometry, loops

Quick Explanation:

Merge the line segments which intersects on infinite points (or at one point with same slope).

Then find all the intersection points of line segments.

Then check if there exist a line segment containing pair of points (for each pair). Store result in a Boolean matrix.

Check all triplets of points.

Detailed Explanation:

1) Merging:

Use map or priority queue for merging node.

a) Map

Keep line segments having same slope and same y intercept (if extended till infinity on both sides).

Sort them according to x coordinate of point having smaller x coordinate (if same x the smaller y coordinate) (if slope is not infinity)

Else sort them according to y coordinate.

Now just simple use a for loop for checking consecutive line segments (after sorting) and merge them.

b) Priority Queue

Sort them according to slope and if slope is same then sort by y intercept (or y coordinate if slope=infinity)

and keep merging it till we can.

2) Finding Intersection Point

Just check each and every pair of line segment for intersection points.

And use SET for ensuring that points don't repeat.

3) Check all smaller line segments (every pair of points)

Just check each and every pair of points if a line segment contains it.

Store result in a Boolean matrix.

4) Check all triplets of points having all three-line segments true

Also check if they are not collinear.

Note: Precision issues can be there. Please use double and don't compare values directly. consider them equal if the difference is very less.

Time Complexity: O(n*log(n) + n^2 + n^4 + n^6).

Editorialist's code:

```
#include<bits/stdc++.h>
using namespace std;

#define ll long long int
#define FIO ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0)
#define mod 1000000007

struct line_seg{
    double x,y,x1,y1;
};

bool equal(double a,double b){
    return abs(a-b)<0.00000001;
}

bool not_equal(double a,double b){
    return abs(a-b)>=0.00000001;
}

pair<double,double> find_intersection(line_seg &a,line_seg &b){

    double a1 = a.y1-a.y;
    double b1 = a.x-a.x1;
    double c1 = a1*(a.x) + b1*(a.y);

    double a2 = b.y1-b.y;
    double b2 = b.x-b.x1;
    double c2 = a2*(b.x) + b2*(b.y);

    double determinant = a1*b2 - a2*b1;

    if(equal (determinant,0))
        return make_pair(DBL_MAX, DBL_MAX);
    else{
        pair<double,double> p={(b2*c1 - b1*c2)/determinant,(a1*c2 -
a2*c1)/determinant};

        if(not_equal(a.x,a.x1)){
            double t=(p.first-a.x)/(a.x1-a.x);
            if(t<0 || t>1)
                return make_pair(DBL_MAX, DBL_MAX);
        }else if(not_equal(a.y,a.y1)){
            double t=(p.second-a.y)/(a.y1-a.y);
            if(t<0 || t>1)
                return make_pair(DBL_MAX, DBL_MAX);
        }else if(not_equal(p.first,a.x) || not_equal(p.second,a.y)){
            return make_pair(DBL_MAX, DBL_MAX);
        }


        if(not_equal(b.x,b.x1)){
            double t=(p.first-b.x)/(b.x1-b.x);
            if(t<0 || t>1)
                return make_pair(DBL_MAX, DBL_MAX);
        }else if(not_equal(b.y,b.y1)){
            double t=(p.second-b.y)/(b.y1-b.y);
            if(t<0 || t>1)
```

```
                return make_pair(DBL_MAX, DBL_MAX);
        }else if(not_equal(p.first,b.x) || not_equal(p.second,b.y)){
            return make_pair(DBL_MAX, DBL_MAX);
        }

        return p;
    }
}

bool check(line_seg &a,line_seg &b){
    double t,t2;

    if((equal (a.x,b.x) && equal (a.y,b.y) && equal(a.x1,b.x1) &&
equal(a.y1,b.y1) || ((equal(a.x,b.x1) && equal(a.y,b.y1) && equal(a.x1,b.x)
&& equal(a.y1,b.y)))))
        return true;
    else if(equal(a.x,a.x1)){
        if(equal(b.x,b.x1) && equal(a.x,b.x) && not_equal(a.y,a.y1)){
            t2=(b.y-a.y)/(a.y1-a.y);
            if(t2<(double)0 || t>(double)1)
                return false;
            t2=(b.y1-a.y)/(a.y1-a.y);
            if(t2<(double)0 || t>(double)1)
                return false;
            return true;
        }
    }else if(equal(a.y,a.y1)){
        if(equal(b.y,b.y1) && equal(a.y,b.y)){
            t=(b.x-a.x)/(a.x1-a.x);
            if(t<(double)0 || t>(double)1)
                return false;
            t=(b.x1-a.x)/(a.x1-a.x);
            if(t<(double)0 || t>(double)1)
                return false;
            return true;
        }
    }else{
        t=(b.x-a.x)/(a.x1-a.x);
        t2=(b.y-a.y)/(a.y1-a.y);
        if(not_equal(t,t2) || t<(double)0 || t>(double)1)
            return false;
        t=(b.x1-a.x)/(a.x1-a.x);
        t2=(b.y1-a.y)/(a.y1-a.y);
        if(not_equal(t,t2) || t<(double)0 || t>(double)1)
            return false;
        return true;
    }

    return false;
}



bool is_not_collinear(pair<double,double> &a, pair<double,double> &b,
pair<double,double> &c){
    return (!equal((a.first * (b.second - c.second) +
            b.first * (c.second - a.second) +
            c.first * (a.second - b.second)),(double)0));
}
```

```
bool cust(line_seg &a ,line_seg &b){

    if(not_equal(a.x,b.x))
        return a.x < b.x ;
    else
        return a.y<b.y;
}

int main()
{
    FIO;
    int n,i,j,k;
    double slope,c;

    cin >> n;

    vector<line_seg> l;
    line_seg temp2,temp3[n];
    set< pair<double,double> > s;
    map< pair<double,double> , vector<line_seg> > m;
    map< pair<double,double> , vector<line_seg> > :: iterator it;

    vector< pair<double,double> > points;
    pair <double,double> temp;

    s.insert({DBL_MAX,DBL_MAX});

    for(i=0;i<n;i++){
        cin >> temp3[i].x >> temp3[i].y >> temp3[i].x1 >> temp3[i].y1 ;

        if(temp3[i].x>temp3[i].x1 || (temp3[i].x==temp3[i].x1 &&
temp3[i].y>temp3[i].y1 )){
            slope=temp3[i].x;
            temp3[i].x=temp3[i].x1;
            temp3[i].x1=slope;

            slope=temp3[i].y;
            temp3[i].y=temp3[i].y1;
            temp3[i].y1=slope;
        }

        if(temp3[i].x1!=temp3[i].x){
            slope=(temp3[i].y1-temp3[i].y)/(temp3[i].x1-temp3[i].x);
            c=temp3[i].y-slope*temp3[i].x;
        }else{
            slope=DBL_MAX;
            c=temp3[i].x;
        }

        it=m.find({slope,c});
        if(it==m.end()){
            vector<line_seg> v;
            v.push_back(temp3[i]);
            m.insert({{slope,c},v});
        }else
            it->second.push_back(temp3[i]);
    }

    for(auto it2:m){
        slope=DBL_MAX;
        c=DBL_MIN;
```

```
    sort(it2.second.begin(),it2.second.end(),cust);

    if(it2.first.first<DBL_MAX){
        i=0;
        while(i<it2.second.size()){
            slope=it2.second[i].x;
            c=it2.second[i].x1;

            i++;
            while(i<it2.second.size()){
                if(it2.second[i].x>c)
                    break;
                c=max(c,it2.second[i].x1);
                i++;
            }

            temp2.x=slope;
            temp2.y=slope*it2.first.first + it2.first.second;
            temp2.x1=c;
            temp2.y1=c*it2.first.first + it2.first.second;
            l.push_back(temp2);
        }
    }else{

        i=0;
        while(i<it2.second.size()){
            slope=it2.second[i].y;
            c=it2.second[i].y1;

            i++;
            while(i<it2.second.size()){
                if(it2.second[i].y>c)
                    break;
                c=max(c,it2.second[i].y1);
                i++;
            }

            temp2.x=it2.first.second;
            temp2.y=slope;
            temp2.x1=it2.first.second;
            temp2.y1=c;
            l.push_back(temp2);
        }
    }
}

for(i=0;i<l.size();i++){
    j=i;
    while(j-->0){
        temp=find_intersection(l[i],l[j]);
        if(s.find(temp)==s.end()){
            s.insert(temp);
            points.push_back(temp);
        }
    }
}

bool is_seg[points.size()][points.size()];

for(i=0;i<points.size()-1;i++){
```

```
        temp2.x=points[i].first;
        temp2.y=points[i].second;

        is_seg[i][i]=true;
        for(j=i+1;j<points.size();j++){

            temp2.x1=points[j].first;
            temp2.y1=points[j].second;

            for(k=0;k<n;k++)
                if(check(l[k],temp2)){
                    is_seg[i][j]=true;
                    is_seg[j][i]=true;
                    break;
                }

            if(k==n){
                is_seg[i][j]=false;
                is_seg[j][i]=false;
            }
        }
    }

    is_seg[i][i]=true;

    n=0;
    for(i=0;i<points.size()-2;i++)
        for(j=i+1;j<points.size()-1;j++)
            if(is_seg[i][j])
                for(k=j+1;k<points.size();k++)
                    if(is_seg[i][k] && is_seg[j][k] &&
is_not_collinear(points[i],points[j],points[k]))
                        n++;
    cout << n;

    return 0;
}
```