



**bubble  
cup**

student coding competition

[bubblecup.org](http://bubblecup.org)

# ProblemSet booklet



# **BUBBLE CUP 2014**

Student programming contest  
Microsoft Development Center Serbia

## ***Problem set & Analysis from the Finals and Qualification rounds***

Belgrade, 2014



**Scientific committee – High school**

Vanja Petrović Tanković  
Danilo Vunjak  
Andrija Jovanović  
Marko Rakita  
Mladen Radojević  
Saket Bharambe  
Dušan Zdravković  
Luka Milićević  
Aleksandar Ivanović  
Lazar Milenković

**Qualification analyses**

Karolis Kusas  
Stefan Velja  
Marko Baković  
Lazar Milenković  
Mislav Bradač  
Bartłomiej Dudek  
Ivan Stošić  
Petar Veličković  
Marko Stanković  
Saša Vučković  
Nikola Jovanović  
Mislav Balunović  
Encho Mishinev  
Dragan Marković  
Marek Sokołowski  
Uglješa Stojanović  
Aleksandar Ogrizović

**Cover:**

Sava Čajetinac

**Typesetting:**

Aleksandar Ivanović

**Proofreader:**

Vanja Petrović Tanković

**Volume editor:**

Dragan Tomić

**Scientific committee – University**

Đorđe Maksimović  
Aleksandar Samardžija  
Andreja Ilić  
Aleksandar Tomić  
Miloš Todić  
Nikola Puzović  
Filip Panjević  
Stefan Tarana  
Abhijith Padmakumar  
Vuk Jovanović

# Contents

Preface.....	6
About Bubble Cup and MDCS .....	7
Bubble Cup Finals 2014.....	8
Bubble Run.....	9
Problem A: Battleships.....	11
Problem B: “Kontra tablic” (Counter-Table) card game .....	14
Problem C: Robo language.....	17
Problem D: Graph to Grid .....	19
Problem E: Kinect object recognition .....	22
Problem F: Integram.....	26
Problem G: Message decoding .....	28
Problem H: DNA Alignment.....	30
Bubble Cup.....	32
Problem A: Forest Snake .....	33
Problem B: Calculator.....	35
Problem C: ForEST .....	39
Problem D: Search.....	41
Problem E: Cycles .....	43
Problem F: Compression .....	45
Problem G: Sticks .....	47
Problem H: Vectors.....	49
Problem I: Queries on an array .....	51
Qualifications .....	54
Problem R1 01: Segment Tree (ID: 6578).....	55
Problem R1 02: TRIVIADOR (ID: 10328) .....	59
Problem R1 03: Greens Land (ID: 10454).....	61
Problem R1 04: Chemistry (ID: 7692).....	63
Problem R1 05: Eight Directions Crossword (ID: 9857).....	65
Problem R1 06: Another understanding of Super Dice Game (ID: 2877) .....	67
Problem R1 07: Snakes and Ladders Again (ID: 13092) .....	73
Problem R1 08: Pythagorean triples (medium) (ID: 14542) .....	74
Problem R1 09: Foxic Expressions (ID: 14975) .....	77
Problem R1 10: [CH] Japan Crossword (ID: 316).....	80
Problem R2 01: Digital Image Recognition (ID: 3360).....	83
Problem R2 02: FLING1 (ID: 13884) .....	86

Problem R2 03: One Instruction Computer Simulator (ID: 2023).....	89
Problem R2 04: Fight with functions (ID: 3902).....	93
Problem R2 05: Soccer Choreography (ID: 850).....	95
Problem R2 06: Yet Another Assignment Problem (ID: 6819) .....	97
Problem R2 07: Illumination (ID: 2661).....	100
Problem R2 09: [CH] Guess The Number With Lies v2 (ID: 17308) .....	104
Problem R2 10: [CH] Colour Brick Game (ID: 18073).....	108

## Preface

*Dear Finalist of Bubble Cup 7,*

*Thank you for participating in the seventh edition of the Bubble Cup. On behalf of Microsoft Development Center Serbia (MDCS), I hope you had a great time in Belgrade and enjoyed yourself.*

*MDCS has a keen interest in putting together this world class event. Most of our team members participated in similar competitions in the past and have passion for solving difficult technical problems.*

*This edition of the Bubble Cup was very special, since it was the most international competition that we have had so far. For the second time competitors have competed in two categories. University students have battled with problems in a 24 hour contest. High school students competed in our traditional format. Not only did we have the participants from the region (Bulgaria, Croatia, Montenegro and Serbia) but also teams from Lithuania and Poland fought their way to the Finals. This means that the Bubble Cup has increased its scope and popularity every year in its history.*

*Given that we live in the world where technological innovation will shape coming decades, your potential future impact on humankind will be great. Take this opportunity to advance your technical knowledge and to build relationships that could last you a lifetime.*

*I thank you all for participating in Bubble Cup Finals.*

*Thanks,*

*Dragan Tomić*

*MDCS PARTNER Engineer manager/Director*

## About Bubble Cup and MDCS

**Bubble Cup** is a coding contest started by Microsoft Development Center Serbia in 2008 with a purpose of creating a local competition similar to the ACM Collegiate Contest, but soon that idea was outgrown and the vision was expanded to attracting talented programmers from the entire region and promoting the values of communication, companionship and teamwork.

Format of the competition has remained the same this year. All competitors battled for the place in finals during two qualifications rounds. They were split into two categories in the finals, with top 10 university teams competing in Bubble Run, the 24 hours long contest, and top 14 high school teams competing in Bubble Cup, traditional 5 hours long contest.

**Microsoft Development Center Serbia (MDCS)** was created with a mission to take an active part in conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 8 years a number of products came out as a result of great team work and effort.

Our development center is becoming widely recognized across Microsoft as a center of excellence for the following domains: computational algebra engines, pattern recognition, object classification, computational geometry and core database systems. The common theme uniting all of the efforts within the development center is applied mathematics. MDCS teams maintain collaboration with engineers from various Microsoft development centers around the world (Redmond, Israel, India, Ireland and China), and Microsoft researchers from Redmond, Cambridge and Asia.



Microsoft  
Development Center Serbia

## Problem set & Analysis





## Bubble Run

Bubble Run finals were held on September 5<sup>th</sup> and 6<sup>th</sup> 2014 in Mikser House. Competition lasted for 24 hours and included top 10 university teams. Tasks on the finals were mostly open problems with no known solution. Problems were from wide variety of areas, including artificial intelligence, classification, sound and image processing, cryptography, DNA sequences, graph theory etc. The competition was tough and competitors had to keep their concentration up all the time, whether to improve their bots in a continuous running game, or to solve a new problem that appeared in the middle of the night.

As the scoreboard froze couple of hours before the end of competition, there was no way of guessing who the winner would be, as the points of top teams were really close to each other. Battle was tight until the very end. After 24 hours of coding, final difference in points between the top 3 teams was only around 400 points. The winners of the Bubble Cup category for university students were **Magowie Psychodelicznej Klawiatury** (Poland), team **KTU #1** (Lithuania) won the second place for the second time in a row, and team **Grafom plovi jedan mali Dijkstra** (Serbia) were third.

Place	Team	Score
1.	Magowie Psychodelicznej Klawiatury	7177
2.	KTU #1	6982
3.	Grafom plovi jedan mali Dijkstra	6762
4.	Wroclaw Cheetahs	5025
5.	skim ščim zbajagom	4574
6.	' OR '1'='1	4392
7.	Young Padawans	3511
8.	RAFnut	2844
9.	101010	2635
10.	Zrakomlati++	2548

Table 1. Final results of Bubble Run

Team	A	B	C	D	E	F	G	H
' OR '1'='1	880	424	290	320	546	400	923	609
101010	260	0	427	184	0	0	994	770
Grafom plovi jedan mali Dijkstra	1505	1407	651	394	856	200	994	755
KTU #1	2165	622	889	474	970	98	994	770
Magowie Psychodelicznej Klawiatury	1720	1228	934	334	931	292	994	744
RAFnut	865	249	169	7	0	0	852	702
skim ščim zbajagom	710	482	265	539	584	200	994	800
Wroclaw Cheetahs	280	688	410	842	893	142	994	776
Young Padawans	325	322	317	220	632	0	994	701
Zrakomlati++	270	332	573	305	206	0	710	152

Table 2. Final scores for each task

## Problem A: Battleships

*Authors*

**Abhijith Padmakumar**

*Points: 3000*

Each team will be provided a bot, henceforth called a "battleship". Your task is to program your battleship to shoot down and destroy other team's ships to score points.

### Game Information

The game server updates the state of the game (moves bots, handles collisions, awards points etc) every frame. You can view the GUI and see your battleship in real time on the projector at the front.

### Arena information

The arena is two dimensional with **length: 1920** and **breadth: 1080**. There is a 'wall' at the arena's boundaries and colliding with the boundary will destroy the battleship.

### Battleship Health

A battleship has a shield with health of 100. Each time your shield is hit, its health is reduced by 25. The shield is destroyed when the health reaches 0. Health can be interpreted this way:

- 100% Full shield
- 75% Half of the shield
- 50% No shield
- 0% Ship is dying

### Radius of the ship:

- Radius with Shield: 36
- Radius without Shield: 18

### Battleship Movement

There is no friction or gravitational forces acting on the battleship. The battleship moves forward in the direction it is facing. The maximum speed, acceleration and the turn speed are given below:

- Maximum speed: 5
- Maximum Acceleration: 0.05
- Turn Speed (Maximum degrees that the bot can rotate about it's own axis in an update cycle): 2

Negative acceleration will reduce the speed, but will not make the battleship move in the reverse direction. Decelerating when the speed is 0 will have no effect and similarly, accelerating when the speed is at maximum value will have no further effect.

## Battleship Firing

A battleship can only fire a missile in the direction it is facing.

The speed of a missile is 10.

The firing rate is 1 missile per second. It is up to the player to track when he fired the last missile. If more than 1 missile firing commands are given in 1s by the player, it will be ignored by the engine. There is no penalty for this.

There is unlimited number of missiles for each battleship.

## Networking and How to Control your bot

This game is played over the network using UDP. The server will continuously broadcast a message containing information about the state of the game. All clients should listen for the broadcast on port 9000. Message broadcast by the server is in the format:

```
teamName,Position.x,Position.y,Rotation,speed,shieldHealth;teamName...;teamName...shieldHealth;  
missilePosition.x,missilePosition.y,missileRotation,missileSpeed;missile....;missile...;  
team1Score;team2Score;.....'team10Score;  
team1Name;team2Name;.....;team10Name;
```

First line contains battleship informations, second line is for missiles, third for score and fourth for team names

Team's state information are delimited by ';'.

**Position.x** - Current x coordinate of the the team's battleship.

**Position.y** - Current y coordinate of the the team's battleship.

**Rotation** - Angle between the direction the battleship is facing and the 'y' axis in the clock wise direction.

**speed** - Current speed of the battleship.

**health** - Current health.

Missile's information are delimited by ';'.

**missilePosition.x** - Current x coordinate of the the team's battleship.

**missilePosition.y** - Current y coordinate of the the team's battleship.

**missileRotation** - Angle between the direction the battleship is facing and the 'y' axis in the clock wise direction.

**missilespeed** - Current speed of the battleship.

Team scores are delimited by ';'.

Team names are delimited by ';'.

Each team needs to send messages to the server to control your bot. The port the server is listening is 55678. The message should be in the following format:

```
"teamPassword,acceleration,rotation,shoot"
```

**teamPassword** - Unique Handle provided to your team. Do not share with other teams!

**acceleration** - Value between [-1,1]. A value of 1 indicates that your should increase it's acceleration to max value in the next frame.

**rotation** - Value between [-1,1]. A value of 0.5 indicates that your bot rotate 0.5\*turnSpeed degrees in the

next frame.

**Shoot** - Value in {0,1}. 1 indicates that you bot fires a missile.

When bot is dead and waiting to response its state info won't be sent as broadcast message

Any parameter for which the client sends a value which is not in the domain specified above will be ignored and defaulted to '0' (no movement). Also please note that if a team doesn't send a message in a particular interval, the server will continue executing the last received message.

Sample UDPClient with code for sending and receiving messages is provided below.

### Scoring and GameInfo

The game will be played over the course of the 24h of BubbleRun and will be on display in the main projector. There will be 20 rounds, each 60 minute long, with a 10 minute interval between rounds. In each round, you get points as follows:

Missile hitting another battleship: +100 points

Missile destroying another battleship: +200 points

Colliding into another battleship (and destroying both): 0 points

Colliding with the wall (screen boundary) and self-destructing: -50 points

At the end of a round, the teams will be ranked by the total score they obtained in the round. Score will be allocated based on the rank as follows:

- 1st - 150
- 2nd - 100
- 3rd - 75
- 4th - 50
- 5th - 40
- 6th - 30
- 7th - 20
- 8th - 15
- 9th - 10
- 10th - 5

At the end of the competition (after all the rounds are completed), a team's net score will be calculated by aggregating scores across all rounds.



## Problem B: "Kontra tablic" (Counter-Table) card game

Authors

**Stefan Tarana**

**Saket Bharambe**

Points: 1500

Rules

Serbian: <http://www.igrajkarte.com/kontra-tablic>

Kontra-Tablic is played with a standard deck of 52 cards. It can be played between 2, 3 or 4 players. **For the purpose it will be played between 2 players only.**

The goal is the opposite of game 'Tablic'. Each player strives to take less.

Card evaluation

- 2 Club, 10, 1/11(aces), 12, 13, 14: worth 1 point
- 10 Diamond: worth 2 points
- Others: 0 points

The flow of the game

- Game is played until one player reach 101. First one who reaches 101 loses the game.
- Game consists of rounds. One round will be played until deck is flushed of one of the players reach 101.
- Each round starts by putting 4 cards on the table and giving each players 6 cards.
- In single round one players is always playing the first move. Next round players will be switched.
- When players are out of cards they will be given another 6 cards until deck is empty or one of the players reach 101.
- Move of Player1 consists of:
  - Player1 selecting a card A from cards given to him and shows it to Player2.
  - Player2 picks multiple groups of cards (0 - n) C1, C2, ... Cn, D1, D2, ... Dn, ... from the table that satisfy formula:
    - $Value(A) = Value(C1) + Value(C2) + \dots + Value(Cn) = Value(D1) + Value(D2) + \dots + Value(Dn)$
    - Each group can be single card and there can be arbitrary number of group selected including none.
  - If groups are non-empty than all cards A, C1 - Cn, D1 - DN will be accounted to Player1.
  - If groups are empty than card A is placed on table.
  - Player2 makes a move the move...
- When the deck is exhausted all remaining cards on the table will be accounted to the player that was accounted last (or to the player who played first if all cards are on the table).

Competition

1. Each contestant submits source code on bubble run portal.
2. Code is compiled on portal and run against other team bots.
3. Bot that has less points in each game is a game winner.
4. Match between two bots consists of 21 games. The winner of the match is bot that has more game wins.
5. All bots will play one match with each other every 4 hours. Number of wins will be summed on normalized with scoring function to gain final score.

## Bot Constraints

1. Bot communicates with game engine through standard input and output.
2. After every game bot will be restarted.
3. If bot doesn't reply to server message after 1 second it will lose the game.
4. If bot dies it loses the game.
5. Each bot is accounted for CPU time (aprox 60000ms) and 1GB of memory. When CPU limit is reached bot is killed. When memory cap is reached any memory allocation will fail.
6. If format of response to the server is invalid or action is invalid bot will lose the game. (not the match!)
7. **Bot will lose a game in case of any invalid move.**
8. Messages from the server to the bot are in format:
  - a. <Command> <DATA> <NEW LINE>
9. Each message to the game engine is in format:
  - a. TRUE|FALSE <DATA> <NEW LINE>

## Communication protocol

<number>:= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 12 | 13 | 14

<Suit>:= 0 | 1 | 2 - diamond | 3 - clubs

Command	Parameters	Response	Description	Advise
PUT_ON_TABLE	<number> <suit>	TRUE	Put card on table	Update in-memory state
REMOVE_FROM_TABLE	<number> <suit>	TRUE	Remove card from table. This happens when the deck is empty and cards from the table are accounted to the player who was accounted last	Update in-memory state.
GIVE	<number> <suit>	TRUE	Player is given a card	Update in-memory state.
MOVE		TRUE <number> <suit>	Select a card previously given to a player to show the other player	You can update in-memory state here or in COMMIT command. The same card will be supplied in COMMIT command.
PICK	<number> <suit>	TRUE <num_of_groups> <num_of_el> el el ...	Based on card shown select cards from table	Update in memory state.

		< num_of_el> el el ...	that in sum gives <number>	
COMMIT	<number> <suit> <num_of_groups> <num_of_el> el el ... < num_of_el> el el ... ->	TRUE	Sum of cards previously selected in MOVE phase and PICK phase by the other layer.	Update in memory state.
CLOSE		TRUE	Politely terminate	

Example: Bot1 vs Bot2

	Receives	Sends
Bot1	PUT_ON_TABLE 3 0	TRUE
Bot1	PUT_ON_TABLE 1 1	TRUE
Bot1	PUT_ON_TABLE 13 0	TRUE
Bot1	PUT_ON_TABLE 10 2	TRUE
Bot1	GIVE 3 1	TRUE
Bot1	GIVE 3 2	TRUE
Bot1	GIVE 13 3	TRUE
Bot1	GIVE 10 3	TRUE
Bot1	GIVE 7 3	TRUE
Bot1	GIVE 6 3	TRUE
Bot2	PUT_ON_TABLE 3 0	TRUE
Bot2	PUT_ON_TABLE 1 1	TRUE
Bot2	PUT_ON_TABLE 13 0	TRUE
Bot2	PUT_ON_TABLE 10 2	TRUE
Bot2	GIVE 4 1	TRUE
Bot2	GIVE 4 2	TRUE
Bot2	GIVE 5 1	TRUE
Bot2	GIVE 6 1	TRUE
Bot2	GIVE 7 2	TRUE
Bot2	GIVE 8 1	TRUE
Bot1	MOVE	TRUE 13 3
Bot2	PICK 13 3	TRUE 2 2 3 0 10 2 1 13 0
Bot1	COMMIT 13 3 2 2 3 0 10 2 1 13 0	TRUE

## Problem C: Robo language

Authors

**Andreja Ilić**

Points: 1200



You are given a field where robot can enjoy afternoon walks. Field can be represented as a matrix where each cell can be of form: **empty space** (robot can walk / be on that cell) or **an obstacle**. There are two special empty cells which are **start and finish cells** of a robot walk.

The robot is a very simple one and only commands that he understands are rotations. These commands are:

- “**s**” - stay at this direction
- “**r**” - rotate for 90 degrees on the right
- “**l**” - rotate for 90 degrees on the left
- “**b**” - rotate for 180 degrees

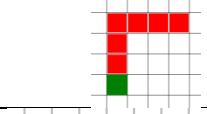
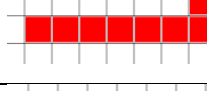
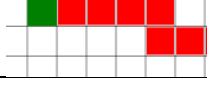
After each of these commands (rotations) robot will make one move in the direction which he is heading at. If other words, he will **first perform rotation and that move one cell in the direction he is facing**. We want to

generate **command list which will take robot from start to finish cell**. Robot can hit an obstacle which will be interpreted as staying in the same cell. You can assume that around the field is a fence which can be interpreted as obstacles (he can't escape).

But there is a small problem ☺. We do not want just give a command list to a robot because that strings can be very long, so we want to compress it as much as we can. We need to **write a program in Robo-language**, in such way that **output of that program is a list of commands** which will be executed by a robot. Idea is to **minimize length of this code**.

Good thing is that the Robo-language is pretty simple. We will have informal description of this language:

- Program contains one command line, one or more method definitions and one or more constant definitions. There should be no empty lines and spaces in Robo program.
- Command line, and each of method and constant definitions should be in separate lines.
- Command line is a list composed from simple commands, methods calls and constant referencing.
- Simple commands are: “**s**”, “**b**”, “**r**” and “**l**”. At the beginning robot is looking up (north).
- Method parameters can be string parameters or integer parameters.
- String parameters are denoted with “**A**”, “**B**”, “**C**” and “**S**”. Integer parameters are denoted with “**N**”, “**M**” and “**K**”.
- Method names are denoted with “**f**”, “**g**”, “**h**”, “**u**”, “**v**” and “**w**”.
- Constant names are denoted with “**a**”, “**c**”, “**x**”, “**y**” and “**z**”.
- Method is defined with method header, after which follows sign “**=**” and after that command list.
- Method can be called with math expressions for integer parameter and command list for string parameters. Math expressions can be created using integers and operators “**+**”, “**-**” and “**\***”.
- Recursive method will stop recursion when at least one of the integer parameters is zero or negative and return empty command list.

Program in Robo language	Output	Code length	Visualization
sssrss	sssrss	6	
x=ll f(N,S)=Sf(N-1,S) xrf(5,s)	llrsssss	28	
f(N)=sf(N-1) g(S)=rSr g(f(3))lf(2)	rsssrlls	32	

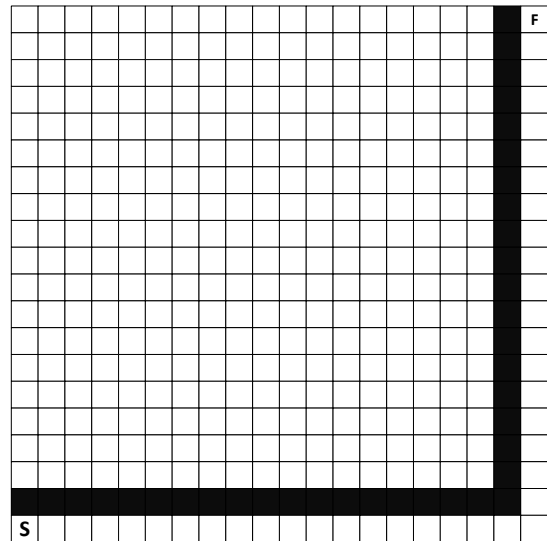
Given program  $p$  we say that it solves given Robo-problem if simulation of its output leads robot from start to end position in less than 10.000 simple commands. If output contains more than 10.000 comands, robot will simulate only first 10.000 simple commands and then he will stop. Also, maximal depth of recursion, stack, should be smaller than 10000.

```
f(N)=sf(N-1)
rf(19)lf(19)
```

Code length is 24, and this code is a solution for example given on the right (black colored cells are obstacles and white ones are empty cells).

Code below is also a solution.

```
f(N)=sf(N-1)
rf(100)lf(100)
```



### Input

First line of input file contains two integers  $n$  and  $m$ , which are dimension of a matrix. Next  $n$  lines contains strings of length  $m$ , which represents matrix cells: "S" for start cell, "F" for finish cell, "#" for obstacle and "." for empty cell.

All inputs are available through our system.

### Output

For each input file, output should be textual file with program written in Robo-language which solves corresponding Robo-problem from input files.

### Scoring

For each test case, score will be equal to the number of characters (code length) used in solution written in Robo-languages. New lines are not counted as characters.

Number of points that you receive for each test case depend on other teams' score on that test case and is calculated as

$$NumPoint = \left(1 - \left(1 - \frac{MIN}{SCORE}\right)^{0.5}\right) \cdot PointPerTestCase$$

In above formula  $MIN$  is minimum score for that test case among all other teams. Each test case is worth the same number of points.



## Problem D: Graph to Grid

Authors

**Aleksandar Tomić**

**Andreja Ilić**

Points: 1000

You are given an undirected graph with  $N$  vertices and  $K$  edges, and a  $M \times L$  dimensional grid. Each point on the grid has a cost  $C_{ij}$ . Your task is to find the projection of the graph to the grid that has the lowest cost. The graph is projected on to the grid by mapping each vertex on to one point on the grid. No 2 vertices may be mapped to the same point.

The cost of a projection is calculated with the following formula:

$$Cost = \sum_{i=0}^{N-1} C_i + \sum_{i=0}^{K-1} MD(E_i.FirstVertex, E_i.SecondVertex)$$

Where:

- $C_i$  is the cost of the grid point where the  $i$  - th vertex is located
- $MD$  is the function for the Manhattan distance between two points on the grid. Manhattan distance between points with coordinates  $(A,B)$  and  $(C,D)$  is equal to  $|A - C| + |B - D|$ .
- $E_i$  the  $i$  - th edge.

You are given 20 input files, and for each input file your task is to submit an output file that contains a description of the mapping needed to project each node to each grid point.

Note that this is an output-only problem, and no program needs to be provided.

### Input files

Files that describe a graph and a grid will be in the following format:

- The first line will contain the non-negative integers  $N$ ,  $K$ ,  $M$  and  $L$  separated by spaces.
- The next  $K$  lines will contain 2 non-negative integers, that will correspond to 2 vertices on the graph that are connected. Vertices are denoted by indexed from the range  $[0, N-1]$ .
- The next  $M$  lines will contain  $L$  non-negative integers between 0 and 1 000 000 000 that correspond to the cost of the grid points.

### Output files

The output files will be expected to be in the next format:

- $N$  lines which contain 2 non-negative integers separated by spaces. The integers in the  $i$  - th line will correspond to the coordinates where the  $i$  - th vertex is located.

Example:

Input file:

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

7 9 9 7 6 7 8  
 8 7 9 6 7 9 6  
 6 7 8 9 5 3 9  
 7 1 9 8 4 5 7

Output File Example:

5 3  
 4 3  
 5 4  
 4 4  
 1 4

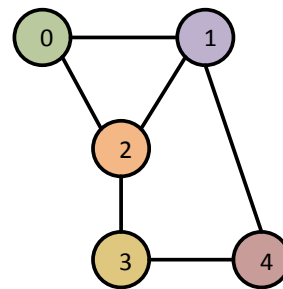
Solution Cost:

30

Explanation:

The input file corresponds to the following grid and graph:

8	5	6	7	5	9	8
7	9	9	7	6	7	8
8	7	9	6	7	9	6
6	7	8	9	5	3	9
7	1	9	8	4	5	7



The solution corresponds to the following layout:

8	5	6	7	5	9	8
7	9	9	7	6	7	8
8	7	9	6	7	9	6
6	7	8	9	5	3	9
7	1	9	8	4	5	7

The cost for the node positions are:

Node0 : 3  
 Node1 : 5  
 Node2 : 5  
 Node3 : 4  
 Node4 : 1

The cost for the edges are:

Edge(0-1) : 1  
 Edge(0-2) : 1  
 Edge(1-2) : 2  
 Edge(2-3) : 1

Edge(3-4) : 3

Edge(1-4) : 4

### Scoring

For each test case, your score is calculated by the following formula:

$$Score = 1 - \left(1 - \frac{MinimumCost}{TeamCost}\right)^{0.5} * 50$$

Where:

- *MinimumCost* is the minimum cost that among all participating teams.
- *TeamCost* the minimum cost that the current team achieved on the given test case

# Problem E: Kinect object recognition

Authors

**Filip Panjević**











**Srđan Rilak**















Points: 2000

Given a short recording of depth images from a Kinect camera, your task is to determine which object (from a given set) the camera is looking at.

### Objects

Kinect camera will be looking at one of the following objects:

Object (ID #)	Mug shots	
Floor/No object (0)		
		
Cube (1)		
		
Ball (2)		

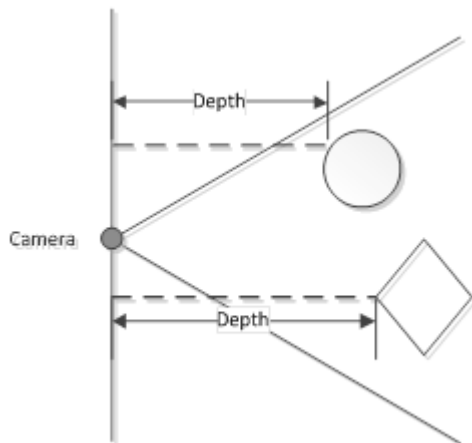
					
Cup (3)					
					
Toilet paper (4)					
					
Shoe (5)					
					

Object will always lie on a flat surface (floor) and there will always be one or no objects in field of view of the Kinect.



## Kinect depth sensor

Kinect depth sensor outputs a 640x480 frame where each pixel represents distance from object in view to camera plane, in millimeters (Figure 1). Depth value 0 indicates unknown distance.



**Figure 1. Depth stream values**

For pixel  $(u, v)$  with depth value  $Z$ , it's position in the real world is (in millimeters):

$$(X, Y, Z) = \left( \frac{(u - 320)}{f} Z, \frac{(v - 240)}{f} Z, Z \right)$$

Where  $f$  is the focal length of the depth sensor (in pixels), and has the value  $f = 571.26$ .

## Problem format

During the course of 24 hours, server will continuously broadcast a stream of depth video over UDP. The stream is composed of 5 second clips looking at a single object. For each 5 second clip you can respond with the ID of the object being shown. If the ID is correct you will receive points, if it's incorrect you will receive negative points. In case the server doesn't receive a response, you will receive 0 points for the given clip. Set of possible objects will change during the course of the competition, as well as the scoring, according to the following scheme:

Time	Objects	Correct answer	Incorrect answer
0 h – 1 h	Floor, Cube	1 point	-1 point
1 h – 6 h	Floor, Cube, Ball	2 points	-1 point
6 h – 12 h	Floor, Cube, Ball, Cup	3 points	-1 point
12 h – 18 h	Floor, Cube, Ball, Cup, Toilet paper	4 points	-1 point
18 h – 24 h	Floor, Cube, Ball, Cup, Toilet paper, Shoe	5 points	-1 point

## Communication

Clients should listen for depth frame broadcast on port 9090. Message broadcasted by the server has the following format:

Data	Size [bytes]
<code>unsigned int video_clip_id</code>	4
<code>unsigned int frame_id</code>	4
<code>unsigned int frame_chunk_id</code>	4

<code>unsigned short pixels[640 * 480 / 10]</code>	$2 * 640 * 480 / 10$
--	----------------------

Each video clip will be 5 seconds long, recorded at 20 frames per second. Each frame will be split into 10 pieces of size 640 \* 48, ordered from top to bottom.

Your response should be as single string in the following format:

**“object <video\_clip\_id> <object\_id> <password>”**

For example, string “object 10 5 tortilla” means there is a shoe in clip number 10, and team password is ‘tortilla’.

**Your response must be received no later than 5 seconds after the clip has completed streaming.** The response should be sent to port 9091. You may send multiple answers for one video clip, but only the first one will be scored. Please don’t spam the server.

**Note: Sensor data is stored in little-endian format. X86 machines are little-endian, but be carefull if you’re using Java.**

## Problem F: Integram

Authors

**Miloš Todić**

Points: 800

Albert Einstein wrote a riddle and said that at most 2% of the world population is able to solve it. In this puzzle, there are 5 houses marked with numbers 1 to 5. Each house has a different color. In each house lives a person with a different nationality. Each person drinks a different kind of beverage, smokes a different brand of cigars and has a different pet.

There is a number of clues given, revealing relationships between items from described categories, like:

- The British person lives in red house
- The owner of the dog doesn't drink tea
- The number on the yellow house is by 2 less than a bird's owner's house number
- ...

The goal is to match each person with his house number and color, beverage he drinks, cigars he smokes and a pet he owns.

Let's make this more general:

Let there be N categories, each with M items. The first category is always a number category, containing numbers from 1 to M. Other categories are marked with letters: A, B, C... Items in each category are marked like this:

1	2	3	...	M	• Items in the first category
A1	A2	A3	...	AM	• Items in category A
B1	B2	B3	...	BM	• Items in category B
...					

Hints are given in these forms:

$Item_1 = Item_2$	Means that items are a match (e.g. A3=C2)
$Item_1 \neq Item_2$	Means that items are not a match (e.g. B2! $\neq$ D4)
$Item_1 \neq Item_2 \neq Item_3 \dots$	Neither pair of the items is a match (shorter for: Item <sub>1</sub> ! $\neq$ Item <sub>2</sub> Item <sub>1</sub> ! $\neq$ Item <sub>3</sub> Item <sub>2</sub> ! $\neq$ Item <sub>3</sub> ...)
$Item_1 < Item_2$	The number from the first category that matches Item <sub>1</sub> is smaller than the one that matches Item <sub>2</sub>
$Item_1 - Item_2 = K$	The number from the first category that matches Item <sub>1</sub> is greater by K than the one that matches Item <sub>2</sub> . K > 0.
$Item_1 \sim Item_2$	Numbers from the first category that match Item <sub>1</sub> and Item <sub>2</sub> , are next to each other.
$Item_1 = Item_2   Item_3$	Item <sub>1</sub> matches exactly one of the items from the right

Item <sub>1</sub>  Item <sub>2</sub> =Item <sub>3</sub>  Item <sub>4</sub>	Item <sub>1</sub> matches exactly one of the items from the right, Item <sub>2</sub> matches the other
--	---

#### Input:

First line of the input holds the values N and M separated by space ( $3 \leq N \leq 9$ ,  $4 \leq M \leq 8$ ).

Second line holds a number of hints, H.

The next H lines contain hints.

Write a program that finds and prints matching items.

#### Output:

Each line of the output should contain matching items separated by space, like:

1 A2 B1 C3

2 A1 B3 C2

3 A4 B4 C4

4 A3 B2 C1

Each item should be printed at most once. Order of the items is not important. Partial solutions are accepted.

#### Example input:

3 4

4

A1=2

B4-A2=1

A3-B3=2

A1!=1!=B1!=B4

#### Example output:

1 A4 B2

2 A1 B3

3 A2 B1

4 A3 B4

## Problem G: Message decoding

Authors

**Vuk Jovanović**

**Andrija Jovanović**

Points: 1000

In beautiful meadows of Iron Hills resided two prosperous kingdoms of Elves on adjacent hills. Their skilled archers successfully defended their kingdoms from attacks. However, this time they face the mighty dwarfs who have already conquered the valley. For defeating the dwarfs, they need a coordinated attack from both the kingdoms. Communicating by sending messengers was out of question as the dwarfs would conquer them. Hence, they decided to communicate by blowing trumpets in tune of the Morse encoding of the message. Knowing that dwarfs also have a copy of the Elves language dictionary, they increased security by using a substitution cipher.

You are now being contacted by the dwarfs to help them decode the message. They have provided you with the recordings of the trumpets and the dictionary they have. It is known that Elves only use the 26 lowercase Latin letters and put spaces between the words.

Input:

Wav file containing recording of coded message. Speed of Morse code can vary across input files. Input file will contain only lowercase letters and a space between words. Message starts with a letter and ends with one.

There will be a text file (dictionary) containing one word per line and it is shared for all inputs.

Output:

Decoded message written only in lowercase letters with single space between words.

Example:

Sample dictionary:

and, at, be, not, or, to

Sample input:

Listen to in.00.example.wav.

Morse code representation of wav file:

.-... ..-. / -.- -.- / ..-. -.. / .. ..-. .-... / .-... ..-. / -.- -.-

Text written inside the encrypted message:

lf xq fd ifl lf xq

Sample output:

Decoded text message:

to be or not to be

Permutation used in substitution cipher:

b	x
e	q
n	i
o	f
r	d
t	l



[Help \(from Wikipedia\):](#)

International Morse code is composed of five elements:

- short mark, dot or "dit" (·) — "dot duration" is one time unit long
- longer mark, dash or "dah" (–) — three time units long
- inter-element gap between the dots and dashes within a character — one dot duration or one unit long
- short gap (between letters) — three time units long
- medium gap (between words) — seven time units long

International Morse code letters:

A ●—	J ●— — —	S ●●●
B —●●●	K —●—	T —
C —●—●	L ●—●●	U ●●—
D —●●	M — —	V ●●●—
E ●	N —●	W ●— —
F ●●—●	O — — —	X —●●—
G — — ●	P ●— — ●	Y —●— —
H ●●●●	Q — — ● —	Z — — ●●
I ●●	R ●— ●	

## Problem H: DNA Alignment

Authors

**Nikola Puzović**

Points: 800

DNA is a molecule that encodes the genetic instructions for development and functioning of all living organisms. The information in DNA is stored in as a code that is made of four nucleotides (chemical bases): Guanine (G), Adenine (A), Thymine (T), or Cytosine (C). The order (sequence) in which these bases are organized is important, since the sequence itself is the information that is used for building an organism. In order to determine if two living organisms are similar, we can align the sequences of DNA and determine if there is evolutionary relationship between them. When two sequences are aligned, identical characters or gaps must be aligned in the same column. A gap is a special symbol ('-') that is used when a match cannot be found in the column. An example alignment is given here:

	Original	Aligned
Sequence 1:	ACACACTA	A - C A C A C <b>T</b> A
Sequence 2:	AGCACACA	A <b>G</b> C A C A C - A

In the example above, we inserted one gap (-) into each sequence so that the remaining characters match. If we want to find evolutionary relationship (or lack of it) in a sample that contains more than one sequence, then we have to perform multiple sequence alignment according to the same rules: identical characters or gaps must be aligned in the same column. The following example shows the alignment of three sequences:

	Original	Matched
Sequence 1:	ACACACTA	A - <b>C</b> A C A C <b>T</b> A
Sequence 2:	AGCACACA	A <b>G</b> <b>C</b> A C A C - A
Sequence 3:	AGACACA	A <b>G</b> - A C A C - A

When constructing the alignment of multiple DNA sequences that are related to each other, the number of gaps is going to be smaller than when unrelated sequences are aligned. Hence, the goal of multiple sequence alignment is to use as little gaps as possible when aligning the sequences.

### Problem and scoring

You are given  $N$  sequences of maximum length  $L$  ( $1 \leq N, L \leq 10000$ ). You need to output the aligned sequences that have the following characteristics:

- All sequences in the output must have the same length.
- When gaps are removed from the output sequences they must be identical to the input sequences with the same name.
- Aligned sequences must have the same symbol or a gap in each column of the output.

Each correct solution will be awarded points according to the following rules:

- Score for each solution is the number of gaps in the solution.
- Solution with the minimal number of gaps will score maximum number of points for the test case.
- Every other solution is scaled according to the formula  $points = A \cdot \left(1 - \left(\frac{Min}{Score}\right)^{0.5}\right)$ , where  $A$  is the maximum number of points for the test case.

## Input and output

Input for one test case is provided in a single input file that contains all sequences. Each sequence is represented with two lines:

- First line contains the name of the sequence. This line must start with a symbol '>'.
- Second line contains the sequence itself and can contain only symbols A, G, C, T and '-'.

An example of valid input and output file is:

### Input file:

```
>sequence Foo  
ACACACTA  
>Bar sequence  
AGCACACA
```

### Output file:

```
>Bar sequence  
AGCACAC-A  
>sequence Foo  
A-CACACTA
```

## Bubble Cup

Bubble Cup finals were held on September 6<sup>th</sup> 2014 in Mikser House. Competition remained in the traditional format, similar to ACM ICPC. Top 14 high school teams tried to solve 9 problems. Three problems remained unsolved, with no submissions for two of them. Every team solved at least one task.

Winners of Bubble Cup were **Me[N]talci Inc.** (Serbia) for the second time in a row. They solved 6 tasks before the four hours mark and start of scoreboard freeze time, which was enough to hold the top position until the end. Second place went to team **.deathSatanBunny** (Serbia), that solved their sixth task during freeze time, getting in front of team **Aroni** (Croatia) that ended the competition third, with 5 solved tasks.

Place	Team	Score	Penalty	Total submissions	Accepted submissions	Rejected submissions
1	Me[N]talci Inc.	6	751	12	6	6
2	.deathSatanBunny	6	891	6	6	0
3	Aroni	5	595	11	5	6
4	The Falcons	4	538	8	4	4
5	Neprelazni B.V.	4	653	11	4	7
6	B Cup	4	743	11	4	7
7	Desarrolladores	4	974	21	4	17
8	Team 17	3	476	7	3	4
9	Even so we are on IOI, we are here.	2	226	4	2	2
10	Cosa Nostra	2	450	4	2	2
11	ExponentialComplexity	2	488	7	2	5
12	Shtepsel	2	554	7	2	5
13	Gimnazija Sombor	1	110	1	1	0
14	NaN	1	313	4	1	3

Table 2. Final results of Bubble Cup

## Problem A: Forest Snake

Authors

**Lazar Milenković**

Implementation and analysis

**Lazar Milenković**

**Dušan Zdravković**

Forest Snake lives somewhere in the big forest of Rudnik and he loves to travel a lot. This summer he decided to visit the famous Micro forest which has many tourist attractions. The most popular among those attractions is Soft tree because of its interesting property. This tree contains a lot of different types of fruits and every fruit has a letter on it. Forest Snake loves palindromes and he decided to make palindrome from the letters of Soft tree. Amount of his happiness is equal to the length of palindrome he makes.

Soft tree can be represented as a connected acyclic graph with  $N$  nodes and  $N - 1$  edges where each node is one fruit and some fruits are connected with edges. Forest snake can choose some node and walk to some other node, but he can visit every edge exactly once. Help him and determine the tour with maximum amount of happiness.

### Input

The first line contains number of nodes  $N$ . The second line contains string of length  $N$  where  $i^{th}$  character is written on node with index  $i$ . Each of the next  $N - 1$  lines contains two integers  $u$  and  $v$ , indicating that there is an edge between nodes  $u$  and  $v$ .

### Output

Output should contain a single integer which represents the maximum amount of Forest Snake's happiness.

### Constraints

- $1 \leq N \leq 5000$
- $1 \leq u, v \leq N$
- All characters are lowercase English letters

### Example input

```
6
badbca
1 2
1 3
1 4
4 5
4 6
```

### Example output

```
4
```

**Time and memory limit: 3s / 256MB**

---

### Solution and analysis:

---

#### Solution 1

The first solution is using trie data structure.

Suppose that the longest palindrome has odd length and its middle is at the current node. Now problem consist of finding two node-disjoint chains which are adjacent to the current node and the strings they form

are the same. We build trie from every subtree rooted at node adjacent to the current node. The trie is built such that it contains all strings which start at the root and end at some leaf. If two tries contain the same strings they are candidates for the solution because they are node-disjoint. We can check whether a string occurs in more than one trie by merging all tries and keeping track of how many times each node occurred. This can be done efficiently by keeping only current trie and union of all tries so far. When the current trie is built we merge it with union. The whole procedure has linear time complexity and when we pick every node as middle overall complexity is quadratic on number of nodes. Memory complexity is linear. Solution is similar for the strings with even length: for every edge build two tries from its endpoints, merge them and check if some string occurs in both tries.

### Solution 2

The second solution is using dynamic programming.

For every two nodes  $u$  and  $v$  calculate

$$dp_{u,v} = \begin{cases} 1, & \text{if } u \text{ and } v \text{ are equal} \\ 0, & \text{if corresponding letters differ} \\ dp_{f(u,v),f(v,u)} + 2 & \text{if corresponding letters are equal and } dp_{f(u,v),f(v,u)} > 0 \end{cases}$$

where  $f(u, v)$  is the first node on path from  $u$  to  $v$ . Calculating all values of  $f$  has quadratic time complexity, and all  $dp$  values can be calculated in quadratic time using memoization. Overall, this solution has quadratic memory and time complexity.

### Summary

Although the first solution has better memory complexity the second one has smaller time constant and it is far easier to implement.

## Problem B: Calculator

Authors

**Dušan Zdravković**

Implementation and analysis

**Andrija Jovanović**

Your task is to implement a special calculator. Like an ordinary calculator, the user can type in a mathematical expression, but that is where similarities end. This calculator lacks some features compared to ordinary calculators - for example, it cannot do subtraction or division. But it can also do some things ordinary calculators are not able to do: after the user types in an expression, she can choose to calculate only a part of it between a given starting and ending point, and she can do it as many times as she likes for the same expression.

The calculator supports the following elements of input:

- Non-negative integers
- Arithmetical operators: +, \*
- Brackets: (, )

### Input

The first line contains one integer  $p$  – the number of elements in the expression. The second line contains the expression  $E$ , comprised of elements listed in the text above. Each element of the expression is separated by a single space character on both sides.

The third line contains one integer  $N$  – the number of requests for calculation on  $E$ . Each of the next  $N$  lines contains two integers, representing the starting and ending element in the expression that should be calculated. (Each number, operator or bracket is a single element. It is not important how many key presses the user needs to make to obtain it).

### Output

The output contains  $N$  lines – in every line there should be one integer, representing the result of the calculation. Since the numbers can get very large, the output should be calculated modulo  $10^9 + 7$ .

### Constraints

- $1 \leq p \leq 1\,000\,000$
- For each integer  $k$  in the expression  $E$ ,  $1 \leq k \leq 10000$ .
- $1 \leq N \leq 100\,000$ .
- $1 \leq a_i \leq b_i \leq p$ , for each  $i \in \{1 \dots n\}$
- It is guaranteed that  $E$  will be a valid mathematical expression.
- It is guaranteed that all subexpressions of  $E$  that need to be calculated will be valid. In particular, none of the subexpressions will begin or end with a + or \* sign, and all brackets will be properly matched.

### Example input

```
17
99 + ( 25 * ( 3 + 7 ) * 10 + 50 ) * 2
2
6 14
3 17
```

### Example output

```
150
5100
```

**Time and memory limit: 2s / 128MB.**



The obvious algorithm that solves the problem is the following:

For each query:

1. Extract the subexpression that needs to be calculated
2. Calculate the value of the subexpression

Step 2 is not completely trivial and we won't go into details on how exactly to implement it, but it should be clear that it requires  $O(p)$  time, and the solution as a whole then requires  $O(N \cdot p)$  time in the worst case, which is clearly not fast enough to finish under the time limit.

Intuitively, we should be able to do this faster because the algorithm described above computes certain expression fragments over and over again. But how do we make use of this insight?

Let's first solve an easier subset of the problem – we'll assume that our expression does not contain any brackets. This means that the expression can be written as a sum of products:

$$E = a_{11} \cdot a_{12} \cdot \dots \cdot a_{1k_1} + a_{21} \cdot \dots \cdot a_{2k_2} + \dots + a_{l1} \cdot a_{l2} \cdot \dots \cdot a_{lk_l}$$

When the expression comes in but before we answer any queries, we can precompute some things.

For each number  $a_{ij}$  we will keep track of the following data:

1.  $i$  and  $j$
2.  $PL_{ij} = a_{i1} \cdot \dots \cdot a_{ij}$  – the left part of the product  $a_{ij}$  belongs to (including  $a_{ij}$ )
3.  $PR_{ij} = a_{ij} \cdot \dots \cdot a_{ik_i}$  – the right part of the product  $a_{ij}$  belongs to (including  $a_{ij}$ )
4.  $SL_{ij} = a_{11} \cdot a_{12} \cdot \dots \cdot a_{1i_1} + \dots + a_{(i-1)1} \cdot a_{(i-1)2} \cdot \dots \cdot a_{(i-1)k_{i-1}}$  – the left part of the total sum, up to the product  $a_{ij}$  belongs to
5.  $SR_{ij} = a_{(i+1)1} \cdot a_{(i+1)2} \cdot \dots \cdot a_{(i+1)k_{i+1}} + a_{l1} \cdot a_{l2} \cdot \dots \cdot a_{lk_l}$  – the right part of the total sum, starting with the product *after* the product  $a_{ij}$  belongs to.

We will also calculate the value of the entire expression  $E$  (which is just  $SL + PL$  for the last element in the expression).

How does this help us? We get a calculation request for the subexpression between elements  $a_{ij}$  and  $a_{pq}$ . Assuming that  $i \neq p$ , the formula

$$E - SL_{pq} - SR_{ij} + PR_{ij} + PL_{pq}$$

gives us the correct result. This is easy to verify:  $E - SL_{pq} - SR_{ij}$  calculates the sum of all products fully contained in the subexpression, and  $PR_{ij} + PL_{pq}$  adds the parts of the two products split by  $a_{ij}$  and  $a_{pq}$  respectively.

Since this is not correct if  $i = p$ , we need a different formula for that case:

$$PR_{ij} \cdot a_{pq} / PR_{pq}$$

Again, it is not difficult to see why this is correct.

How fast is this solution? The precompute step can be performed in  $O(p)$  time. We can go through the expression from left to right, saving cumulative sums and products and arrays as we go along to get  $PL$  and  $SL$ . For  $PR$  and  $SR$ , we do the same thing from right to left. Answering queries can now be done in constant time – we only need to do a couple of array lookups and arithmetic operations. (Note: not all constant-time operations are created equal. The division modulo  $10^9 + 7$  needs to be implemented through [exponentiation by squaring](#). You could write a naïve implementation that makes  $10^9 + 7$  steps, which is still theoretically  $O(1)$  but the time limit checker will not be convinced by that argument ☺). The overall time complexity of

the solution is  $O(p + N)$ .

Let us now solve the full problem. The presence of brackets changes the complete structure of the task, right? Well, not really. Notice that, for each pair of brackets, it is impossible to construct a valid query that contains only one of the brackets and not the other. This means that each query has to either fully reside within the pair of brackets, or fully cover the brackets and everything inside them. This immediately gives us a way to reduce the problem to the solved case:

1. If the subexpression is fully contained within the brackets, we can safely ignore everything outside the brackets
2. If the subexpression generated by the query covers the brackets, we can replace the contents of the brackets with a single number – the value of the expression delimited by the brackets

Let's do this on the example from the problem statement:

$$99 + ( 25 * ( 3 + 7 ) * 10 + 50 ) * 2$$

6 14

The subexpression is completely inside the outer pair of brackets, so we ignore everything else:

$$25 * ( 3 + 7 ) * 10 + 50$$

The remaining pair of brackets is covered by the subexpression, so we calculate it:

$$25 * 10 * 10 + 50$$

Now we have reduced the problem to the already solved case and we can solve it using the same algorithm. The only remaining question is how to do the bracket handling work without impacting the algorithmic complexity. A straightforward way to do this is with the help of a stack structure:

1. Set the number of encountered brackets so far to  $b = 0$ .
2. Set the values we want to keep track of (current accumulated sum, current product, current values of  $i$  and  $j$ ) to their initial values.
3. Go through the expression from left to right, keeping track of cumulative sums and products and saving values of  $PL, SL, i, j, b$  for each element to an array.
  - a. If the current element is an open bracket, take the current state (current sum, current product, current values of  $i$  and  $j$ ), put it on the stack, then reinitialize the state. Increase the number of encountered brackets by 1.
  - b. If the current element is a closed bracket, record the value of the whole expression  $SB$  within the bracket (which should already be calculated as the accumulated sum within the bracket), then pop the previous state from the stack. Keep calculating  $PL$  and  $SL$  for the remaining elements, as if everything within the brackets was a single element with the value  $SB$ .
4. Repeat steps 1-3, going from right to left this time and filling in values of  $PR$  and  $SR$ .

The formulas for answering queries remain the same as in the previous case. The only difference is that, due to  $i$  and  $j$  being "bracket-relative" now, we also have to check whether  $b_{ij} = b_{pq}$  to know when to use the first formula and when the second.

The complexity of this algorithm is the same as the complexity of the algorithm described earlier. We still keep track of just one set of values for each expression element. The only additional operations during the precompute step are related to handling of the stack, but there can be at most  $O(p)$  of them. Answering queries is still done in  $O(1)$  time per query.

## Notes

1. The memory limit is 128MB, so there is more than enough memory to store all data. The memory complexity of the solution is  $O(p)$ , but some amount of care has to be taken to not let the constant factor get out of hand.
2. All inputs and outputs can be handled as 32-bit integers, but using 32-bit numbers everywhere can overflow during multiplication. So you either need to be careful and cast to 64-bit when it's needed, or work with 64-bit numbers all the way and try not to hit the memory limit.

## Problem C: ForEST

Authors

**Dušan Zdravković**

Implementation and analysis

**Dušan Zdravković**

**Lazar Milenković**

This year, ForEST (traditional festival for inhabitants of forests) is organized in Forest Snake's forest. Best beer manufacturers present their products and top jungle musicians have live performances in the middle of wood. Traditionally, monkeys take care of security, and this year they decided to organize ForEST slightly different – there will be multiple fan pits in the forest. The first pit is nearest to the stage, the second is immediately after the first and so on... Also it is known that first pit is inside the second, which is inside the third... Fan pits are separated by long straight ribbons which are fixed to some trees in the forest. Ticket price for the first fan pit is  $N$  coins (forest coin is official currency in every forest in the world), for the second  $N - 1$ ,... For  $i^{th}$  fan pit price is  $N - i + 1$ . Being outside of any fan pit is free.

Position of every tree in forest can be represented with two integer coordinates, and every security ribbon can be represented as straight line segment starting at one tree and ending at some other tree. So, one fan pit is actually convex polygon with trees as vertices and ribbons as edges. Forest Snake is interested in following problem: given coordinates of some animals in the forest, what price each of them paid for being at that place during ForEST?

### Input

The first line contains one integer  $N$  – the number of fan pits. Each of the next  $N$  lines start with the number of nodes of fan pit  $M$ , followed by  $M$  pairs of integers  $x$  and  $y$  – coordinates of nodes. The next line contains number  $Q$  – number of Snake's question. Each of the next  $Q$  lines contain pair of integers  $x$  and  $y$  representing the position of animals.

### Output

For each of the  $Q$  animals output a single integer per line which is equal to its ticket price.

### Constraints

- There is at least one fan pit
- Sum of number of nodes of all polygons doesn't exceed  $3 \cdot 10^5$
- Every fan pit has at least three nodes
- $1 \leq Q \leq 3 \cdot 10^5$
- $-10^9 \leq x, y \leq 10^9$
- It is guaranteed that the first polygon is inside the second, the second inside the third...  $N - 1^{th}$  inside the  $N^{th}$
- All polygons are convex and there is no animal standing on any ribbon or tree
- Nodes of the polygons are given in counterclockwise order

### Example input

```
2
3 -3 2 2 -3 3 5
4 10 10 -10 10 -10 -10 10 -10
3
0 0
100 100
6 3
```

### Example output

```
2
0
1
```

Time and memory limit: 3s / 64MB

For every convex polygon make upper and lower chain, such that upper chain contains all points on clockwise path from the leftmost point of the polygon to the rightmost one. Similar, lower chain contains all points on counterclockwise path from the leftmost to the rightmost point. Make array of all points from input (points from queries and polygons) and sort them by  $x$  coordinate ascending. Also, make two stacks, one for upper chains and the other for lower chains. Stack should keep only index of the last point visited so far. Traverse the array and check if current point is from polygon or from some query:

1. If the point is from polygon:
  - a) If it's the leftmost point of some chain push its polygon on stack
  - b) If it's the rightmost point of some chain pop its polygon from stack
  - c) Otherwise make current point be the last point of its chain
2. If the point is query point:

Check if it's under all upper chains and then do binary search on lower chains, otherwise do the binary search on upper chains. Binary search finds between which two chains the current point is located, and from that we can easily calculate the number of polygons in which the point is located inside of.

Sorting all the points requires  $O(K \log K)$  where  $K$  is total number of points including those on polygon and from queries. Every binary search requires  $O(\log N)$ , which leads to total complexity  $O(K \log K + Q \log N)$ .

## Problem D: Search

Authors

**Dušan Zdravković**

Implementation and analysis

**Danilo Vunjak**

Alice is looking for a job and she has heard that one software company is hiring experts in *run-length encoding* (RLE). It is a very simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. For example, you can compress “`WWWWWWWWWWWWBWWWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWW`” into “`12W1B12W4B14W`”, “`11522666`” into “`21152236`”. Alice is playing around with encoding of positive integers. One unit of data is a single digit, so she encodes each sequence of the same digit as that digit and its count.

We can define a function Alice uses for encoding as  $RLE(A) = B$ , where  $A$  and  $B$  are positive integers. However, since this is a compression algorithm, Alice doesn't consider an encoding valid if  $B$  has more digits than  $A$ .

Alice encodes a number multiple times in a row, until the encoding is not valid. For example, number 333 can be encoded two times in a row before the encoding gets invalid:  $RLE(333) = 33$ ;  $RLE(33) = 23$ ;  $RLE(23) = 1213$  – number 1213 has more digits than number 23 so the last encoding is not valid.

After playing with RLE she found one interesting number - 22. Number 22 can be encoded infinitely many times, because  $RLE(22) = 22$ .

Now she wants to find a positive integer different than 22, with no more than 100 digits, that can be encoded at least 5 times in a row before the encoding gets invalid. Help her!

### Input

There is no input for this problem.

### Output

Output a single positive integer Alice is looking for.

### Constraints

- Number of digits of the positive integer in the output must be  $\leq 100$ .
- Output must not be 22.

### Example input

No example input

### Example output

No example output

**Time and memory limit: 0.1s / 64 MB**

---

*Solution and analysis:*

---

The idea is to find the number which can be encoded 4 times, and then manually construct the number which can be encoded into that number (so ultimately it can be encoded 5 times). To find the first number which can be encoded 4 times, we can use brute force solution. Straight forward brute force solution can be optimized with some heuristics. For example, we don't need to consider numbers that contain digits 5, 6, 7, 8 and 9 in itself, since they will not satisfy both condition to have less than 100 digits, and to be encodable 5 times.

Smallest number different than 22 that can be encoded 4 times is 2233322211. Now, number 22333333333333333333333333333333222222222222222222221 (two times 2, thirty-three times 3, twenty-two times 2 and one 1) can be encoded 5 times. Since this number has less than 100 digits, it is one of the solutions. Another solution can be obtained from number 22333222112, and it will have less digits, 31 total. Number is  $[2x2, 3x3, 3x2, 22x1, 1x2]$ .

## Problem E: Cycles

Authors

**Luka Milićević**

Implementation and analysis

**Luka Milićević**

**Lazar Milenković**

You are given a graph  $G$  with two spanning trees that share no edges. A cycle in  $G$  is a connected subgraph whose vertices have degree 2. Your task is to find a collection of cycles in  $G$  such that every edge is in precisely two of your cycles. Such a collection will always exist.

### Input

The first line contains two integers separated by an empty space:  $n$  – the number of vertices and  $m$  – the number of edges. Every of the next  $n$  lines contains 3 integers  $u, v, l$ , separated with empty spaces, which represent the edge between vertices  $u$  and  $v$ . The remaining integer  $l$  can take values 0, 1, 2. Value 0 means that it is not in either of the spanning trees, 1 means that it is in the first one, and 2 that it is in the second tree.

### Output

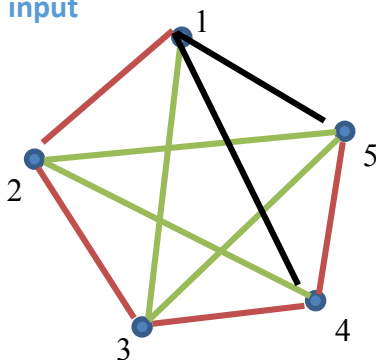
The first line of the output should contain a single integer  $C$  – the number of cycles you produced. Every line that follows should describe one of the  $C$  cycles and should start by integer  $m$  which is the size of the cycle and should then contain  $m$  integers that specify the cycle (so that the edges are between the 1<sup>st</sup> and 2<sup>nd</sup> vertex, 2<sup>nd</sup> and 3<sup>rd</sup> vertex, etc., and between  $m^{\text{th}}$  and 1<sup>st</sup>). Any collection of cycles that contains every edge precisely twice is considered to be a valid solution.

### Constraints

- $1 \leq n \leq 500000, 1 \leq m \leq 1000000, 2n - 2 \leq m$ .
- There are no loops or repeated edges.

### Example input

```
5 10
1 2 1
2 3 1
3 4 1
4 5 1
1 5 0
1 3 2
3 5 2
5 2 2
2 4 2
1 4 0
```



### Example output

```
4
5 1 2 3 4 5
5 1 3 5 2 4
5 2 3 1 4 5
5 5 3 4 2 1
```

Time and memory limit: 10s / 256MB



Since we're already given spanning trees in the graph and we're looking for cycles, it is natural to recall that given a tree  $T$  and an edge  $e$  not in the tree, there is a unique path in  $T$  that joins the endpoints of  $e$ . In turn, this gives a cycle that contains edge  $e$ . Write  $T_1$  for the first spanning tree that is given, and  $T_2$  for the second one. By applying this procedure to edges not in  $T_1$  and the tree  $T_1$ , we get a collection of cycles that contain each edge not in  $T_1$  precisely once, while the edges of  $T_1$  can possibly be contained in multiple cycles. If an edge is contained in more than two cycles, this is certainly a problem, given our goal. The key observation now is that we can actually turn the current collection of cycles into a useful one by applying the symmetric difference to the cycles. In other words, we pick the edges in an odd number of cycles in the current collection. It is easy to see that this gives us an even subgraph  $H$  (i.e. all vertices have even degrees), where we may perform Euler tour to produce a new collection of cycles, that contain each edge of  $H$  precisely once. Crucially, the new collection of cycles contains each edge not in  $T_1$  precisely once, while the edges of  $T_1$  are contained at most once.

We can repeat the same procedure to tree  $T_2$ . As the trees have no common edges, by taking a union of the two collections produced so far, we obtain a collection of cycles that contains each edge of  $G$  once or twice. Finally, we observe that taking all edges that appear precisely once gives another even subgraph, and performing Euler tour once more and adding these cycles, produces a solution – a collection of cycles containing each edge precisely twice.

As far as the implementation is concerned, there is an important point – how to find the cycles coming from a spanning tree efficiently? Recall that we actually do not need all the cycles given by paths along the tree, but actually only the resulting even subgraph given by symmetric difference. For this it is actually sufficient to write at each node of a tree the number of non-tree edges that are adjacent to it and then to pick tree edges whose subtrees have odd sum of numbers at nodes. Thus, a simple tree traversal suffices.

To sum up, the algorithm looks as follows:

- For  $T_1$ , for each node  $v$ , write the number of non-tree edges adjacent to  $v$ , and then perform a DFS traversal that sums the numbers in subtrees and chooses the edges with odd subtree sum.
- Take the edges chosen above and add edges not in  $T_1$  to form an even subgraph.
- Perform Euler tour on this subgraph, and add the resulting cycles to solution.
- Repeat all the steps above to  $T_2$ .
- Finally, form an even subgraph of edges appearing only once the cycles chosen so far.
- Perform another Euler tour to complete the solution.

The algorithm has linear time and memory complexity.

## Problem F: Compression

Authors

**Vanja Petrović Tanković**

Implementation and analysis

**Vanja Petrović Tanković**

**Aleksandar Ivanović**

A software company is making tools for data compression. One group of engineers is working on algorithms for compression of really long textual data. Currently, they are implementing some variations of compression algorithm called *run-length encoding*. *Run-length encoding* is a technique where consecutive runs (sequences) of same data are stored as a run value and count. In this case specifically, algorithm takes a string  $S$  as an input and compresses it into a new string  $C_S$  using *run-length encoding*.  $C_S$  is of the form

$$\langle run_1 \rangle \langle count_1 \rangle \langle run_2 \rangle \langle count_2 \rangle \dots \langle run_k \rangle \langle count_k \rangle$$

where  $\langle run_i \rangle$  is a non-empty string value and  $\langle count_i \rangle$  is a positive integer value for each  $i = 1..k$ .

Decompressing  $C_S$  back to  $S$  works by repeating value of  $\langle run_i \rangle$  exactly  $\langle count_i \rangle$  times for each  $i = 1..k$ .

Obviously,  $C_S$  may not be unique because it might be possible to split  $S$  into runs in many different ways, so engineers are experimenting with different approaches of splitting  $S$  into runs. All of these approaches are optimized for compression speed and low memory consumption of the algorithm, and not the compression efficiency, because in reality input data can be several terabytes large and it is more important that compression is fast and does not take a lot of resources. To measure compression efficiency, engineers will use smaller inputs and compare their approaches of splitting  $S$  into runs to the optimal way of splitting  $S$  into runs – one that results in  $C_S$  of minimum length. Help them by writing a program that will calculate the minimum length of  $C_S$ .

### Input

The first line contains one integer  $N$  – length of string  $S$ . The second line contains the string  $S$ .

### Output

Output contains one integer – minimum length of string  $C_S$ .

### Constraints

- $1 \leq N \leq 3000$
- All letters of  $S$  are lowercase letters of English alphabet

#### Example input

12  
aaaababababc

#### Example output

7

### Example explanation

Compressed string of the smallest length is  $a3ab4c1$ . Some other possibilities for  $C_S$  are  $a4ba3bc1$ ,  $aa2ba2babc1$ ,  $aaaababababc1$  etc. but none of them have length less than 7.

**Time and memory limit: 1.5s / 128MB**

We will start by describing a **dynamic programming** part of the solution.

Let  $DP[i]$  represent the minimum compression length for the string  $S[1..i]$  (substring of  $S$  starting at the position 1 and ending at the position  $i$ ). The solution is then  $DP[N]$ .

So, how do we calculate  $DP[i]$ , for  $i = 1..N$ ? We initialize by setting  $DP[0] = 0$ . When we are at the position  $i$ , we take any substring of  $S$  ending at the position  $i$  as a run. There are exactly  $i$  such substrings:  $S[i..i], S[i-1..i], S[i-2..i], \dots, S[1..i]$ . For each run, we will try to repeat it  $k$  times, where  $k = 1..M$ , and  $M$  is  $\lfloor \frac{i}{length(run)} \rfloor$ .

Then, if the substring  $S[i - k \cdot length(run) + 1..i]$  is equal to the substring  $S[i - length(run) + 1..i]$  repeated exactly  $k$  times, one possible value for  $DP[i]$  would be  $DP[i - k \cdot length(run)] + length(run) + number\_of\_digits(k)$ .  $DP[i]$  obviously takes a minimum of all such values.

We do not need to check the whole first substring – if the equality of substrings holds for  $k - 1$  and  $S[i - k \cdot length(run) + 1..i - (k - 1) \cdot length(run) + 1]$  is equal to the  $S[i - length(run) + 1..i]$ , then it holds for  $k$  also. Actually, we can also see that when we get to  $k$  that doesn't satisfy the equality, we can stop for this run, because the condition would not be satisfied for any number larger than  $k$  either.

Let's take a string  $S = 'aabab'$  as an example. For  $i = 5$ , the runs can be  $'b', 'ab', 'bab', 'abab'$  and  $'aabab'$ . Taking  $'b'$  as a run and  $k = 1$ , we take  $DP[4] + length('b') + number\_of\_digits(1)$  as one possible value for  $DP[5]$ . For  $k = 2$ , we see that substring  $S[4..4]$  is not equal to  $'b'$  so we can stop. For  $'ab'$ , one possible value for  $DP[5]$  is when  $k = 1$ , and the value is  $DP[3] + length('ab') + number\_of\_digits(1)$ . When  $k = 2$ ,  $S[2..3] = S[4..5]$  so possible value for  $DP[5]$  is  $DP[1] + length('ab') + number\_of\_digits(2)$ . We cannot try with  $k = 3$  because of length of  $S$ . Similarly, we can check runs  $'bab', 'abab'$  and  $'aabab'$ , with  $k$  only equal to 1, due to their length.

We haven't mentioned the complexity of the algorithm yet, but so far we can see that this doesn't seem efficient enough in the worst case. We can speed this up by precomputing if the substring  $S[i - j..i] = S[i + 1..i + j + 1]$  for all possible values of  $i$  and  $j$  and storing the results in a matrix of size  $O(N^2)$ . This will help us improve the dynamic programming part of the solution by avoiding checking if two substrings are equal character by character every time, and instead have a  $O(1)$  check. We can precompute this in several ways, such as using **hashing** or **trie** data structure. When using hashing, we can compute the hash values of all substrings in  $O(N^2)$ , using *rolling hash*, as in *Rabin-Karp algorithm*.

Let's see what the total complexity of the solution is. For each index  $i$ , we check all the runs ending at  $i$ . This has time complexity  $O(N^2)$ . Then, for each run, we try to repeat it  $k$  number of times. In the worst case, where the characters of the input string are all the same, we would need to try for each  $k = 1.. \lfloor \frac{i}{length(run)} \rfloor$ . For each  $k$ , we need to check if the substrings are the same, and without any precomputation this has time complexity of  $O(length(run))$ , so the total complexity of the algorithm in that case would be  $O(N^3)$ . However, with  $O(1)$  check, the total complexity is  $O(H_N N^2)$ , where  $H_N$  is  $N^{th}$  harmonic number, and  $H_N = \sum_{i=1}^N \frac{1}{i}$ . Harmonic series grows very slowly, so for  $N = 3000$ ,  $H_N \leq 10$ . With the memory complexity of  $O(N^2)$ , this algorithm is efficient enough, given the constraints in this task.

## Problem G: Sticks

*Authors*

**Dušan Zdravković**

*Implementation and analysis*

**Vanja Petrović Tanković**

You are given an array of  $N$  sticks. First stick is at the position 1, second at the position 2, etc. Last one is at the position  $N$ .

You can pair two sticks if they are not paired already and there are exactly two sticks between them, either paired or unpaired. When you pair two sticks, you move one to the position of other – you can choose which one you move. Only the stick that is moved changes its position, all the other sticks remain at their previous position.

Given  $N$ , determine if it is possible to pair all the sticks and if it is, output the pairing process. If there are multiple solutions, output any solution.

### Input

The first and only line contains one integer  $N$  – number of sticks.

### Output

If it is not possible to pair all the sticks, output  $-1$ . If it is possible, output the pairing process, one pairing step per line. Format of each step is " $a b$ " (without quotation marks), which means that the stick at the position  $a$  is paired with the stick at the position  $b$  and that the stick at  $a$  is moved to  $b$  ( $1 \leq a, b \leq N$ ).

### Constraints

- $1 \leq N \leq 50$

#### Example input

5

#### Example output

-1

### Example explanation

It is not possible to pair all the sticks. We can pair 4 sticks in different ways, but one stick would end up without a pair. One way to pair 4 out of those 5 sticks would be

1 4

5 3

After the first step, stick at the position 1 is paired with the stick at the position 4 and moved to that position. Notice that there are now two sticks between position 3 and 5. In the next step, stick at the position 5 is paired with the stick at the position 3 and moved to that position. Stick at the position 2 is left without a pair.

**Time and memory limit: 0.1s / 16MB**

Let's make a few observations first.

If  $N$  is odd, it is obvious that there is no solution because at least one stick would end up without a pair.

If  $N$  is even, we can reduce the problem to  $N - 2$  sticks by pairing the fourth and first stick and moving the fourth to the first position. We are then left with two sticks less, with the two paired sticks being at the beginning of the array and having no effect in the further pairing process. That means that if the pairing of all sticks is possible when  $N$  equals some even number  $K$ , it is also possible when  $N$  equals any even number larger than  $K$ . Now we just need to find minimum such number.

We can see that it is impossible to pair all the sticks when  $N$  equals 2, 4 or 6 by manually trying all the possibilities (there aren't many of them). We can also try to solve the problem when  $N = 8$  manually. It is not hard to come up with the full pairing process for  $N = 8$ , so we can conclude that there is a solution when  $N$  is even and  $N \geq 8$ .

One solution when  $N = 8$  is to first pair  $5^{th}$  and  $2^{nd}$  stick, moving  $5^{th}$  to the  $2^{nd}$  position. Then, we can pair  $3^{rd}$  and  $7^{th}$  stick, moving the  $3^{rd}$  to  $7^{th}$  position, since there is exactly two unpaired sticks between them after the first step. Third step should be pairing and moving the  $8^{th}$  stick to the  $6^{th}$  position (there is exactly one pair between them from the second step). Final step is pairing the  $1^{st}$  and  $4^{th}$  stick.

So, the final algorithm is to pair the first and the fourth unpaired stick from the beginning of the array in each step, by moving the fourth unpaired stick to the position of the first, until we are left with exactly 8 unpaired sticks. Then those 8 sticks can be paired as previously described. All of this can be done in linear time, so the final algorithm has complexity  $O(N)$ .

## Problem H: Vectors

Authors

**Luka Milićević**

Implementation and analysis

**Luka Milićević**

**Aleksandar Ivanović**

A set of  $m$  vectors  $\{v_1, v_2, \dots, v_m\}$  in  $\mathbb{R}^d$  (the set of  $d$ -tuples of real numbers) is said to be *linearly independent* if the only reals  $\lambda_1, \lambda_2, \dots, \lambda_m$  that satisfy  $\lambda_1 v_1 + \lambda_2 v_2 + \dots + \lambda_m v_m = 0$  are  $\lambda_1 = \lambda_2 = \dots = \lambda_m = 0$ . For example, in  $\mathbb{R}^2$  the set of vectors  $\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$  is linearly independent. However,  $\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right\}$  is not since  $1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} + (-1) \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ .

In this task, you are given  $n$  vectors in  $\mathbb{R}^d$ , and every vector has some weight. Your job is to find a linearly independent set of vectors with maximal sum of weights.

### Input

The first line contains two integers  $d$  and  $n$ . The next  $n$  lines contain  $d + 1$  integers each, separated with one empty space between any two integers. The first  $d$  numbers in the line  $i + 1$  are coordinates of the  $i^{\text{th}}$  vector, and the last number is its weight.

### Output

The output should consist a single integer: the sum of weights of vectors in your set.

### Constraints

- $1 \leq d \leq 200$
- $1 \leq n \leq 500$
- The coordinates of the vectors are integers in the range  $[-10^3, 10^3]$ .
- The weights of the vectors are integers in the range  $[-10^6, 10^6]$ .

#### Example input

```
4 4
1 0 0 0 30
0 0 1 0 30
1 0 1 0 100
0 0 0 1 1
```

#### Example output

```
131
```

**Time and memory limit: 0.5s / 16MB**

---

### Solution and analysis:

---

We claim that the greedy algorithm works, i.e. that it suffices to sort the vectors, start with an empty set of vectors and then at each step add the heaviest vector to the set if the set remains linearly independent.

Observe the following basic lemma from linear algebra.

**Lemma 1.** Suppose that vectors  $\{v_1, v_2, \dots, v_n\}$  are linearly independent, and that  $\{u_1, u_2, \dots, u_{n+1}\}$  are also linearly independent. Then we can find some  $k$  such that  $\{v_1, v_2, \dots, v_n, u_k\}$  is also linearly independent.

We postpone the proof for later, the fact above should at least be intuitively obvious.

**Proof that the greedy algorithm is correct.** We prove by induction on  $s \geq 1$  that the greedy algorithm produces a set of  $s$  vectors of maximal weight. For  $s = 1$  this is clear, as we choose a non-zero vector of maximal weight.

Suppose now that  $s \geq 2$  and that the statement holds for smaller values of  $s$ . Suppose however that the statement fails for  $s$ , that is, the greedy algorithm finds  $\{v_1, v_2, \dots, v_s\}$ , but  $\{u_1, u_2, \dots, u_s\}$  has higher weight. Still, by induction hypothesis,  $\{v_1, v_2, \dots, v_{s-1}\}$  is optimal for  $s - 1$ . This means that any subset of  $s - 1$  elements of  $\{u_1, u_2, \dots, u_s\}$  has weight at most that of  $\{v_1, v_2, \dots, v_{s-1}\}$ , and in particular,  $v_s$  has smaller weight than any  $u_i$ . By Lemma 1 applied to  $\{v_1, v_2, \dots, v_{s-1}\}$  and  $\{u_1, u_2, \dots, u_s\}$ , we have some  $k$  such that  $\{v_1, v_2, \dots, v_{s-1}, u_k\}$  is linearly independent, and  $u_k$  has higher weight than  $v_s$ , so it would have been added to our set before  $v_s$ , which is contradiction. This finishes the proof that the algorithm works.  $\square$

**Proof of Lemma 1.** Suppose contrary, so every  $u_k$  can be written is in the span  $V$  of  $\{v_1, v_2, \dots, v_n\}$ . But  $\{u_1, u_2, \dots, u_{n+1}\}$  are linearly independent in  $V$  which contradicts Steinitz exchange lemma. (See [http://en.wikipedia.org/wiki/Steinitz\\_exchange\\_lemma](http://en.wikipedia.org/wiki/Steinitz_exchange_lemma)).  $\square$

Finally, we may observe that a simple way to implement the algorithm above is to sort the vectors by weights and put them as rows in a matrix in the sorted order. Then one pass of Gaussian elimination gives the solution, by picking the non-zero rows. This gives an algorithm of time complexity  $O(dn^2)$  and memory complexity  $O(dn)$ .

## Problem I: Queries on an array

Authors

**Aleksandar Ivanović**

Implementation and analysis

**Aleksandar Ivanović**

**Petar Veličković**

You are given an array  $a$  of  $N$  elements. Array  $a$  is 0-indexed. There are two types of queries that you should perform on the array.

- *INVERT*  $i j k$ : Invert the  $k^{\text{th}}$  bit on each element in the range  $[a_i, a_j]$
- *SUM*  $i j$ : Output the sum of the elements in the range  $[a_i, a_j]$

Note that  $0^{\text{th}}$  bit is the least significant bit and  $31^{\text{st}}$  bit is the most significant bit.

### Input

The first line contains one integer  $N$  – size of the array. Second line contains  $N$  integers that are initial values of the elements in the array. Third line contains one integer  $Q$  – number of the queries. Following  $Q$  lines contain one query per line.

### Output

Output contains  $Q_{\text{sum}}$  lines, where  $Q_{\text{sum}}$  represents the number of *SUM* queries, and each line contains the answer to the corresponding query.

### Constraints

- $1 \leq N \leq 100.000$
- $1 \leq Q \leq 100.000$
- $0 \leq i \leq j \leq N - 1$
- $0 \leq k \leq 31$
- Elements of the array are 32 bit unsigned integers.

#### Example input

```
4
1 2 3 1
5
SUM 0 2
INVERT 0 2 0
SUM 0 2
INVERT 3 3 10
SUM 3 3
```

#### Example output

```
6
5
1025
```

**Time and memory limit: 4s / 16MB**

---

### Solution and analysis:

---

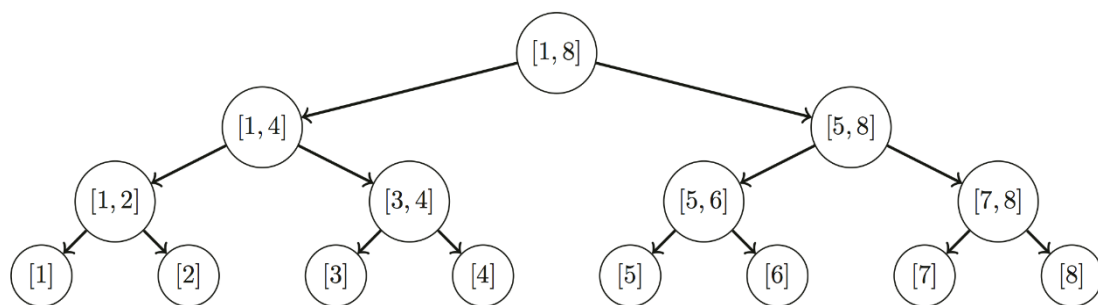
Here we are faced with an interesting and relatively straightforward problem, assuming familiarity with the required data structure. The obvious brute-force solution would involve manually updating each array element upon an inversion command, and performing the summing up in a similar fashion. This has a worst-



case complexity of  $O(N)$  per operation, which is too slow for the given query time. Clearly, we need to utilise a more clever approach here.

A common pattern of thought when up against a problem involving dynamically updating an array and then querying over its intervals is to try to get the complexity down to  $O(\log N)$ ; one of the most common data structures to consider for achieving this is called a **segment tree**, which we can use for this problem as well. A segment tree is a binary tree constructed over an array (a more general version is constructed over a set of points on the real number line) where each node is “responsible” for a certain subinterval of the array – the root is responsible for the entire array, leaves are responsible for each of the individual elements, while nodes in between are responsible for the union of the intervals held responsible by their children – the figure below represents an example layout of the structure for an array of size 8, with the intervals noted in each node:

The operations of updating and querying a single element are clearly of logarithmic time complexity, as they require recursively descending down the tree, halving the interval being considered in each step. Querying



over a range is also of logarithmic cost – if we query by aggregating the values stored in the minimal set of nodes covering the entire range; the proof that there will always be  $O(\log N)$  nodes in this minimal set is left as an exercise to the reader. Updating a range requires us to be more clever; we can only update the minimal set of nodes covering the range – but as we might need to propagate this to the nodes deeper in the tree, we use a technique called **lazy propagation**, where we store pending updates in nodes and propagate them to their children whenever they are accessed.

To see how a segment tree could be used for solving this problem, let us consider an easier variant: *assume we are only dealing with 1-bit values* (as in, all members of the array are lesser than 2). This problem can easily be solved by using a segment tree, having each node store the **sum** of the array values in the interval it’s responsible for (i.e. each node’s value is equal to the sum of the values of its children). Once we have a structure like this, querying for the sum of a range simply involves summing over the nodes in the minimal set as discussed in the previous paragraph. Inversions are first performed by “inverting” all the nodes in the minimal set, then propagating the operation to their children when necessary, and so on. “Inverting” a node is simple: if a node is responsible for an interval of size  $l$ , and it had stored a sum of  $k$  previously, then after inverting that interval, the sum stored will become  $l - k$  (as all the 1s in the interval become 0s and vice versa). Hence, we have successfully reached a logarithmic-time solution per operation for this version of the problem.

To expand this to  $d$ -bit values, we just need to construct  $d$  segment trees of 1-bit values as outlined in the previous paragraph, each responsible for an individual bit of the integers. An inversion operation involves updating the appropriate segment tree (given to us in the problem with the input parameter  $k$ ). When summing up, we can just add up all the powers individually with separate summations on each segment tree:

$$SUM(l, r) = \sum_{i=0}^{d-1} segTree[i].SUM(l, r) * 2^i$$

This gives us an overall time complexity of  $O(\log N)$  per invert operation and  $O(d \log N)$  per summation – overall the worst-case time complexity of the algorithm (when only performing sum queries) of  $O(Q \cdot d \log N)$  and the space complexity is  $O(dN)$ . As in this version of the problem we have fixed  $d = 32$ , we can ignore it from our analysis, giving us the required  $O(Q \log N)$  behaviour.

Another thing to note is that the query result may overflow a 32-bit integer, and as such, using a 64-bit type (long long in C++/Int64 in Pascal) is necessary to completely solve this problem.

## Qualifications

As previous years, the qualifications were split into two rounds, with ten problems in each round. Challenge problems that were introduced last year were present this year as well. Non-challenge problems were worth 1 point in the first round and 2 points in the second round. Challenge problem in the first round was worth maximum of 4 points, while each of the two challenge problems in the second round were worth maximum of 8 points.

The problems for both rounds were chosen from the publicly available archives at the Spoj site ([www.spoj.com](http://www.spoj.com)).

This year, 92 teams managed to solve at least one problem from the qualifying rounds. The competition has long exceeded its regional character, with teams from all over the world participating in the qualifications. Teams that qualified for the finals were from Serbia, Croatia, Bulgaria, Poland, Lithuania, United Kingdom and Ukraine.

Num	Problem name	ID	Accepted solutions
01	Segment Tree	6578	37
02	TRIVIADOR	10328	93
03	Greens Land	10454	55
04	Chemistry	7692	53
05	Eight Directions Crossword	9857	80
06	Another understanding of Super Dice Game	2877	41
07	Snakes and Ladders Again	13092	51
08	Pythagorean triples (medium)	14542	84
09	Foxtic Expressions	14975	65
10	[CH] Japan Crossword	316	69

Table 1. Statistics for Round 1

Num	Problem name	ID	Accepted solutions
01	Digital Image Recognition	3360	17
02	FLING1	13884	41
03	One Instruction Computer Simulator	2023	30
04	Fight with functions	3902	8
05	Soccer Choreography	850	4
06	Yet Another Assignment Problem	6819	30
07	Illumination	2661	3
08	Moebius	3647	Not solved
09	[CH] Guess The Number With Lies v2	17308	50
10	[CH] Colour Brick Game	18073	31

Table 2. Statistics for Round 2

We continued with our tradition that the contestants are the ones who are writing the solutions for qualifications problems. You should note that these solutions are not official - we cannot guarantee that all of them are accurate in general. (Still, a correct implementation should pass all of the test cases on the Spoj site.)

**The organizers would like to express their gratitude to qualification task authors and everyone who participated in writing the solutions.**

---

## Problem R1 01: Segment Tree (ID: 6578)

---

Time Limit: 1-3.5 second

Memory Limit: 256 MB

It was Arbor Day. Alice implemented an RB-tree, Bob composed a segment tree, I made a binary tree - we all have a bright outlook.

Lambda is always making mistakes while implementing segment trees (See his history of submissions). He then decides to draw a "segment tree". He puts  $n$  points on a plane, link certain pairs of them to form segments and all the segments form a tree. As a normal tree, it satisfies the following conditions:

1. Consider points as vertices, segments as edges, it forms a rooted tree.
2. Each node  $u$  is strictly higher than its parent, namely  $y_u > y_{parent\_of\_u}$ .
3. Segments may only intersect on their endpoints.

Lambda wants to minimize the total length of segments. The tree can be rotated to satisfy above conditions.

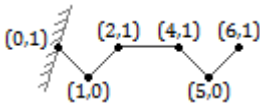
### Input

First line of input contains single integer  $n$  ( $1 \leq n \leq 500$ ). Next  $n$  lines each contain two integers  $x_i, y_i$  denoting the coordinate of  $i_{th}$  point ( $0 \leq x_i, y_i \leq 1000$ ). Points are distinct.

### Output

The one and only line contains a real number representing the minimum length. Your answer must be rounded up to 4 digits after the decimal point.

### Sample

input	output
6 0 1 1 0 2 1 4 1 5 0 6 1	7.6569 

---

### Solution:

---

Prerequisites: stack, two pointers.

The first naive thought helps to understand that we can rotate points about the point  $(0, 0)$  at all possible angles and then to calculate the minimum length of the tree. The length of the tree is the sum of lengths of edges between the vertex and its parent. For each vertex pick the parent which is higher and is the nearest from all the possible candidates. There must be exactly one vertex with no parent at all, if there is more or less, then it is not possible to get a tree.

Additionally, it is easy to see that in this way of connecting vertices there can be no intersections. Because if there are vertices  $v_1$  with the parent  $p_1$  and  $v_2$  with the parent  $p_2$  and edges  $(v_1, p_1)$  and  $(v_2, p_2)$  intersect, then we can swap parents and receive a better answer.

There is a problem in trying all possible angles - their number is infinite. However, we need to consider only  $O(n^2)$  number of angles, because there are  $\binom{n}{2}$  vectors from one vertex to the other. When we try an angle there will be a vector which, when rotated, has the positive  $y$  coordinate and it lies at the smallest angle with the  $OX$  axis. So we can go through each vector and assume that it lies at the smallest angle with the  $OX$  axis

(it is best to assume that it makes as small as possible positive angle with the  $OX$  axis). Then we know which vectors we can consider and to calculate one possible length of a tree.

Firstly, let's sort all possible vectors (from one vertex to all others) by their pointing direction.

There is a structure of the vector:

```
struct vec {
    int x, y;
    int typ;
    int ind;
    vec(int gx = 0, int gy = 0, int own = 0) {
        x = gx; y = gy;
        if (y == 0) typ = x > 0? 0: 4;
        else if (x == 0) typ = y > 0? 2: 6;
        else if (x > 0 && y > 0) typ = 1;
        else if (x < 0 && y > 0) typ = 3;
        else if (x < 0 && y < 0) typ = 5;
        else if (x > 0 && y < 0) typ = 7;
        else assert(false);
        ind = own;
    }
};
```

The vector has coordinates  $(x, y)$  and has its type by the place which it is in. Notice, that types makes it easy to compare vectors by their orientation. The attribute "*ind*" shows which vertex is a beginning point for the vector. Let's move each vector to the  $(0, 0)$  and to sort them with this function:

```
bool Less(const vec &a, const vec &b)
{
    if (a.typ != b.typ) return a.typ < b.typ;
    int cr = cross(a.x, a.y, b.x, b.y);
    if (cr != 0) return cr > 0;
    return a.ind < b.ind;
}
```

To check if the vectors make the same angle with the  $OX$  axis we use the function:

```
bool Equal(const vec &a, const vec &b)
{
    return !Less(a, b) && !Less(b, a);
}
```

We can compare vectors by their type, and if their types are equal we can use the cross product. Because the positive sign of the cross product shows that the first vector is to the right of the second one and vice versa.

The solution uses these functions with vectors:

```
int cross(int ax, int ay, int bx, int by) { return ax * by - ay * bx; }
ld len(ld ax, ld ay) { return sqrt(ax * ax + ay * ay); }
```

When we have sorted the vectors we can apply the "two pointers" strategy to quickly find an interval of vectors which point upwards.

Let's introduce the operations made when we go with the right pointer to the right and when we go with the left pointer to the right. When we go with the right pointer - we add a vector to our data structure, when we go with the left pointer we remove a vector from our data structure.

Each vertex has its own stack where its outgoing vectors are ordered by the length (from the lowest to the greatest). The stack is implemented as an array with two pointers: "Sl" - the left pointer and "Sr" - the right pointer. The first elements of each stack are added to get the current length of a tree - "cur", also the number of vertices which has at least one element is counted - variable "cnt". If  $cnt = n - 1$ , where  $n$  is the number of all vertices, then we have a new candidate. Let's pick the candidate with the minimal length. It is an answer to the given task.

The operations with stacks are handled with these functions:

```
// Insert(vertex, vector_index)
void Insert(int v, int ind)
{
    while (Sl[v] < Sr[v] && lens[S[v][Sr[v] - 1]] >= lens[ind]) {
        if (Sr[v] - Sl[v] == 1) { cur -= lens[S[v][Sr[v] - 1]]; cnt--; }
        Sr[v]--;
    }
    if (Sl[v] == Sr[v]) { cur += lens[ind]; cnt++; }
    S[v][Sr[v]++] = ind;
}

// Erase(vertex, vector_index)
void Erase(int v, int ind)
{
    if (Sl[v] < Sr[v] && S[v][Sl[v]] == ind) {
        cur -= lens[S[v][Sl[v]]]; cnt--; Sl[v]++;
        if (Sl[v] < Sr[v]) { cur += lens[S[v][Sl[v]]]; cnt++; }
    }
}
```

There is a small trick in "Erase" function, we only need to check and to delete the first element. Because we delete the leftmost element, if it had been greater than the smallest one, it would have been already deleted when inserting other elements after it.

The main function which initializes the data and implements the "two pointers" strategy:

```
int main()
{
    scanf("%d", &n);
    if (n == 1) { printf("0.0000\n"); return 0; }
    for (int i = 0; i < n; i++)
        scanf("%d %d", &X[i], &Y[i]);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) if (j != i)
            vecs[vlen++] = vec(X[j] - X[i], Y[j] - Y[i], i);
    }
    sort(vecs, vecs + vlen, Less);
    for (int i = 0; i < vlen; i++)
        lens[i] = len(vecs[i].x, vecs[i].y);

    int pnt = 0;
```

```
    for (int i = 0; i < vlen; i++) {
        while (Equal(vecs[i], vecs[pnt]) || cross(vecs[i].x, vecs[i].y,
vecs[pnt].x, vecs[pnt].y) > 0) {
            Insert(vecs[pnt].ind, pnt); pnt = (pnt + 1) % vlen;
        }
        if (cnt == n - 1) res = min(res, cur);
        Erase(vecs[i].ind, i);
    }
    cout << fixed << setprecision(4) << res + eps << endl;
    return 0;
}
```

The complexity of the algorithm is  $O(n^2)$ .

---

**Added by:** Lox

**Solution by:**

*Name:* Karolis Kusas

*School:* Kaunas University of Technology

*E-mail:* karolis.kusas@gmail.com

---



---

## Problem R1 02: TRIVIADOR (ID: 10328)

---

Time Limit: 5.0 second

Memory Limit: 256 MB

Triviador is a war between two Kings. A king can attack an enemy region at each step. When a king attacks a region, he conquers all the enemy regions connected to it (not just the immediate ones). All the 8 regions around any region are connected to it. The kings get alternate chances to attack. King1 gets the chance to attack first. Assume both kings are intelligent and find who will conquer the whole territory at the end of the war. It can be proved that one of the Kings can win for sure if he is intelligent!

### Input

The first line is an integer  $t$  ( $1 \leq t \leq 100$ ), denotes the number of test cases. In each test case the first line consists of two integers  $m$  and  $n$  ( $1 \leq m, n \leq 10$ ) denoting the number of rows and columns in the territory (each cell is a region). Then the description of each cell follows. Every region contains a character 'X' if it is owned by King1 or 'O' otherwise.

### Output

For each test case output the result in a single line 'X' if King1 wins or 'O' if King2 wins.

### Sample

input	output
3	O
3 3	X
XOX	X
XXX	
XOX	
3 5	
XXXXX	
XXXOO	
XXXOO	
4 4	
XXXX	
OOOO	
XXXX	
OOOO	

---

### Solution:

---

First of all, let us define a component. A component is a set of same-colored regions such that when one of them is attacked by a king, all of them are conquered.

It is obvious that, unless one of the kings has won, each component has at least one adjacent component (a neighbor). In case of a component having exactly one neighbor, we call it simple.

It is also fairly easy to notice that a non-simple- component with  $K$  neighbors divides the grid into  $K$  sections – sets of components that are separated from all other components by either a wall, or a single component.

It will here be proven that the king that wins the game will be the king with more components at the start of the game, or, in case of a tie, the first king to make a move.

We will try to prove that if a king has a greater or equal number of components than his opponent, after both of them make move, the former one will still have a greater or equal number of components. So, if King1 has  $C1$  components, and King2 has  $C2$  components, if both of them play optimally, at the start of every move that king1 is supposed to play, he will be able to maintain his advantage ( $C1 \geq C2$ ), or will NOT be able to negate his opponent's advantage ( $C1 < C2$ ).

Suppose King1 makes a move. If King2 has a simple component, by conquering that component, King1 will not change the number of his components, and he will decrease the number of King2's components by 1. Thus, he will maintain his advantage, since King2 can only reduce the difference of  $C1$  and  $C2$  (in his advantage, of course) by at most one.

In case of King2 not having a simple component, let's pick a random component of his – name it *TheChosenOne*. If *TheChosenOne* has  $K$  neighbors ( $K > 1$ ), it divides the grid into  $K$  sections. Inside each of these sections, King1 has at least one component more than King2. This is easily proven recursively, since either the section consists of one simple component belonging to King1, or King2 has a component in that section, dividing it into further subsections, for which the same rule can apply. Thus,  $C1$  has at least  $K - 1$  components more than  $C2$  (at least one for each section and  $-1$  for *TheChosenOne*).

If King1 conquers *TheChosenOne*, the number of his components will decrease by  $K - 1$ , since he will merge his  $K$  components into a single one, and  $C2$  will be decreased by 1. Thus, no matter which component King1 picks (as long as King2 has no simple components), he will still have at least 1 component more than his opponent ( $C1 - (K - 1) > C2 - 1$ ).

As said before, no matter what move King2 plays, he will not end up having more components, since before his move King1 had strictly more components than him.

This proves that if one of the kings has a greater or equal number of components at the start of his turn, he will maintain his advantage until the start of his next turn. The process repeats until the losing king has no components remaining.

Since we need to count the number of components belonging to each king, we can simply use BFS for this. The complexity is  $O(m * n)$ , where  $m$  and  $n$  are the numbers of rows and columns, respectively, so this solution is quite easy to code, although relatively difficult to prove.

---

**Added by:** [cegprakash](#)

**Solution by:**

**Name:** [Stefan Velja](#)

**School:** [Gimnazija Jovan Jovanović Zmaj](#)

**E-mail:** [stefanvelja96@gmail.com](mailto:stefanvelja96@gmail.com)

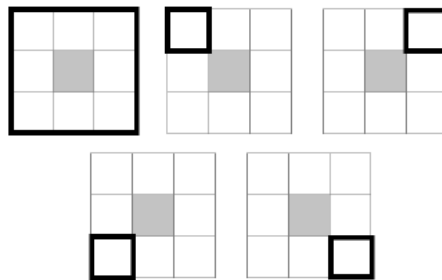
---

**Problem R1 03: Greens Land (ID: 10454)**

Time Limit: 2.0 second

Memory Limit: 256 MB

Mr. Green has a large portion of land divided into square units that are either field or lake areas. He wants to fence a rectangular portion of his lands to use for livestock. The lake areas have a very soft soil and any fence built near those areas have a chance to fall (and then the animals could escape), so no fence should be built near a lake area.



Mr. Green wants to know of how many ways he can fence a rectangular area of his lands without any portion of the fence having a common border with a lake area. In the example above, for a 3x3 land with a lake area in the center, we have 5 possibilities of fence.

**Input**

On the first line a positive integer: the number of test cases, at most 100. After that per test case: One line with a integer  $N$  ( $1 \leq N \leq 300$ ): the size of the land ( $N \times N$ ).  $N$  lines, each with  $N$  characters. Each character is either '.' or 'X'. The  $j$ -th character on the  $i$ -th line is a 'X' if position  $(i, j)$  is a lake area, and '.' if it is a field area.

**Output**

For each test case output a line with the number of different valid ways with Mr. Green can fence his lands.

**Sample**

input	output
3	5
3	8
...	441
.X.	
...	
3	
X..	
...	
X..	
6	
.....	
.....	
.....	
.....	
.....	
.....	

---

**Solution:**

---

We're looking for number of rectangular areas such that we can surround them with fence that does not have common border with lake area. Row  $r$  is good to be top side of rectangle if we can build fence between row  $r - 1$  and row  $r$ , and it's good to be bottom side of rectangle if we can build fence between row  $r$  and row  $r + 1$ . Column  $c$  is good to be left side of rectangle if we can build fence between column  $c - 1$  and column  $c$ , and similar with the right side.

\*Assuming that top left field is  $(0, 0)$  and bottom right  $(n - 1, n - 1)$

Fix two opposite sides of rectangle, we'll fix top and bottom. Now, for each column we want to know whether that column is good to be left and/or right side of rectangle with fixed top and bottom sides. We need fast way to do this.

Column  $c$  is good to be left side if there is no lake area in column  $c - 1$  from fixed top to fixed bottom row (including them) and same for column  $c$ . It also similar for right side, we need all field areas in column  $c$  and column  $c + 1$ .

Let  $dp[i][j]$  be number of lake areas in column  $j$  from row 0 to row  $i$  (including them). We can calculate this matrix at the beginning using simple rules:

- if cell  $(i, j)$  is lake area then  $dp[i][j]$  is equal  $dp[i - 1][j] + 1$
- otherwise  $dp[i][j]$  is equal  $dp[i - 1][j]$

Now, we can simply check if  $dp[down][c] - dp[up - 1][c]$  is 0 and we know that in column  $c$  from row up to row down there is no lake areas. Later, when we say check if column is good to be left/right side, it means use  $dp$  matrix to check if corresponding two differences are 0.

Lets get back to the solution. After fixing top and bottom side, say we fixed  $x_{top}$  and  $x_{bottom}$ , iterate through columns from left to right. When on column  $c$  follow next steps:

- Check if cells  $(x_{top} - 1, c)$  and  $(x_{top}, c)$  are field areas. If not, this means that we can not use any of previous columns nor column  $c$  to be left side, so just reset some counter, for example  $number_{of\_left\_sides}$  to 0 and continue to column  $c + 1$ . Do similar things for cells  $(x_{bottom}, c)$  and  $(x_{bottom} + 1, c)$ .
- If all four cells are field areas, we can use  $number_{of\_left\_sides}$  as number of good columns to be left side of our rectangle. Additionally, check if column  $c$  is good to be left side of our rectangle (one column can be both, left and right side) and if it can increase  $number\_of\_left\_sides$  by 1.
- Finally, check if column  $c$  is good to be right side of our rectangle. And if it is, add  $number_{of\_left\_sides}$  to the solution. Otherwise, continue to column  $c + 1$  (do not reset counter  $number_{of\_left\_sides}$ ).

Time complexity of this solution is  $O(n^3)$  where  $n$  is size of input matrix. Memory complexity is  $O(n^2)$ .

---

**Added by: Paulo Costa****Solution by:***Name: Marko Baković**School: School of Computing, Union University, Belgrade, Serbia**Faculty of Mathematics, University of Belgrade, Belgrade, Serbia**E-mail: markobakovic95@gmail.com*

---

---

## Problem R1 04: Chemistry (ID: 7692)

---

Time Limit: 60.0 second

Memory Limit: 256 MB

The story started some 5'000 years ago in Ancient Egypt, was continued by the Greeks and Arabs, reached France, Europe, and finally conquered the world. The studies on the compositions of waters, the humans' greed for purified materials, millions of experiments and many brilliant minds made chemistry what it is today: No more the quest of the Philosopher's stone, but the study of matter and the changes it undergoes.



There remain nevertheless still groups of stout-hearted followers of ancient believes, so-called alchemist. Keeping their research top-secret, they meet once a year for a conference where they share their recent findings. This year's location is Lausanne and Extremely Purified Fluorescent Liquids (EPFL) is the topic. The idea is that the chemists brew together some new EPFLs. As we speak about state of the art EPFLs, it is necessary that certain chemists put their very specific knowledge together. Thus for a certain EPFL  $E_1$ , the presence of chemists  $C_1, C_2$  and  $C_3$  may be required. For another  $E_2$ , chemists  $C_1$  and  $C_4$  might be necessary.

Although chemists are generally very patient people, as their reactions might take long times, they get very impatient if they are to observe experiments in which they are not involved. As an example, chemist  $C_4$  would go crazy if he had to assist to the brewage of  $E_1$ . To ensure a pleasant stay in Lausanne to every chemist, you are to arrange their departure and arrival dates so that each chemist is available whenever his knowledge is required, but is not in Lausanne when other EPFLs are created.

To this purpose, you are given a schedule with ones and zeros. Each column stands for one EPFL, each row for one chemist. There is a 1 at position  $(C_i, E_i)$  if chemist  $C_i$  is needed for EPFL  $E_i$ , and a zero otherwise. Your task boils now down on rearranging the columns in a way that all ones are consecutive in every line. For traditional reasons, the organizers' EPFL is always brewed first and corresponds to the first column of the input schedule ( $E_1$ ).

### Input

The input consists of several test-cases separated by an empty line. Each test-case starts with the number of chemists  $C$  ( $1 \leq C \leq 400$ ), followed by the number of EPFLs  $E$  ( $1 \leq E \leq 400$ ). Then follow  $C$  lines of  $E$  characters, '1' or '0'. You may assume that there exists exactly one order of EPFLs (initiated by  $E_1$ ) that meets the above constraints. Input terminates on a test-case with  $C = E = 0$ , which must not be processed.

### Output

Print each output on a line by itself, which holds  $E$  numbers, corresponding to the initial position in the planning, arranged such that all chemists are available when necessary and away from Lausanne otherwise. (the first number must always be 1 as a tribute to the host).

### Sample

input	output
6 5 00010 01000 10101 10100 00011 00101 0 0	1 3 5 4 2

---

### Solution:

This is simple graph theory problem. Let's represent EPFLs as nodes in graph. Two nodes are connected by edge if there is a chemist which attends both of their corresponding EPFLs. Number of edges between two nodes is the same as the number of chemists attending their corresponding EPFLs. Degree of node is the number of different nodes adjacent to it. The algorithm which solves this problem is very simple:

*Push the first node on queue*

*While queue is not empty*

*Current node is the first node in queue*

*Mark current node as visited*

*Append current node to the solution*

*Find unvisited nodes with maximal number of edges in common with current node*

*If there is more than one such node*

*Choose one with minimal degree*

*If there is only one such node*

*Push it on queue*

*If there is an unvisited node append it to the solution*

At the beginning we push the first element on queue because of the task statement. In every step of while loop current node is being marked as visited and appended to the solution. After that, the next node is chosen as the node with maximum number of edges in common with current node. This is because one node have more or equal edges in common with its neighbor node (in final arrangement) than with other nodes. If there are more nodes with maximum number of edges in common with current node, then it is easy to show that we should choose the one with minimum degree. At the end of the while loop there can be only one unvisited node and we put it on the end.

---

**Added by:** *Christian Kauth*

**Solution by:**

*Name: Lazar Milenković*

*School: School of Computing*

*E-mail: milenkovic.lazar@gmail.com*

---

---

## Problem R1 05: Eight Directions Crossword (ID: 9857)

---

Time Limit: 1.0-4.0 second

Memory Limit: 256 MB

What is an Eight Directions Crossword? It's a filled crossword in which all the words are hidden in eight directions (up, down, left and right and also up-left, down-right etc.) You have to find these hidden words in each crossword.

Đuro has made an  $N \times N$  eight-directions-crossword. His crossword is a bit strange: you are given only one word and you have to find it in a crossword. To make things more difficult, you can skip some letters in the crossword while looking for the given word. More precisely, the given word is the subsequence of not necessarily consecutive letters in a row, column or a diagonal of the crossword in one of the eight directions.

Now you discover that, under these conditions, you can read the given word in the crossword in multiple ways. How many?

### Input

In the first line of the input there is an integer  $N$  ( $2 \leq N \leq 1000$ ), the crossword dimension, followed by space and the given word you are to find. This word has  $2 - 10$  letters.

$N$  lines follow, representing the crossword. All letters in the crossword and in the given word are small letters of the English alphabet.

### Output

Print the required number of ways. (This number will fit into *int64* in Pascal or *long long* in C/C++)

### Sample

input	output
8 silba siolobba oooaoooo oooboooo aoooooooo oboloooo oolooooo oooioooo oooss000	4

---

### Solution:

---

This was one of the easiest problems in this year's BubbleCup qualification rounds.

Let's see how to solve this task on one dimensional crossword. This is well known dynamic programming task. For each position  $i$  in one dimensional crossword we need to calculate number of ways to match first  $j$  letters in given word. We store that in  $dp[i][j]$  and final solution is stored in  $dp[size\ of\ crossword][length\ of\ given\ word]$ .

Transitions are following:

- $dp[i][j] = dp[i][j] + dp[i - 1][j]$  // we skip  $i_{th}$  letter in crossword
- $if (a[i] == s[j]) dp[i][j] = dp[i][j] + dp[i - 1][j - 1]$  // the letter in crossword is same as the next letter in given word that we need to match so we match  $j_{th}$  letter

Now we get back to the original problem. We notice that solution to original problem on two dimensional crossword is sum of solutions to a lot of one dimensional problems. We can implement this task nicely by

writing only one function solve that returns solution to one dimensional problem given starting row of original matrix, starting column of original matrix and direction. We also have two arrays *delta\_row* and *delta\_column*. *Delta\_row* and *delta\_column* help us determine which cell in two dimensional crossword is next cell to be considered in one dimensional problem we are solving at the moment. For example if direction 0 = *down* then *delta\_row*[0] = 1 and *delta\_column*[0] = 0. We easily get new row and column by adding *delta\_row* and *delta\_column* to current row and column. By implementing function solve in this way we can call it by any direction and any starting position. For direction down we call solve with every cell of first row, for direction right with every cell in first column, for down-right with every cell in first column and with every cell in first row and so on. For every direction we call  $O(n)$  times function solve. Time complexity of function solve is  $O(n * l)$ ,  $l$  = length of given word. Overall time complexity is  $O(n * n * l) = O(l * n^2)$ .

---

**Added by:** [Adrian Satja Kurdija](#)

**Solution by:**

*Name:* [Mislav Bradač](#)

*School:* [V. Gimnazija](#)

*E-mail:* [mislav.bradac@gmail.com](mailto:mislav.bradac@gmail.com)

---



---

## Problem R1 06: Another understanding of Super Dice Game (ID: 2877)

---

Time Limit: 2.0-4.0second

Memory Limit: 256 MB

When we were trying to solve the problem SDGAME, we got a misunderstanding of it. We didn't get AC until we were told the original meaning. But we think our kind of understanding is also interesting and is worthy of doing. So enjoy the problem.

Alice and Bob are playing a game. The game consists of a circular track of  $M$  ( $2 \leq M \leq 1,000,000,000$ ) cells labeled 0 through  $M - 1$ . Initially both players start at cell 0. The game progresses by having each player take turns rolling one of  $N$  ( $1 \leq N \leq 10,000$ ) 'super - dice' labeled 0 through  $N - 1$ . The actual mechanics of the 'super-dice' is not very well understood; however, it is known that they will only ever turn up a number between 0 and 1,000,000,000 inclusive after a roll. After rolling the super-dice the number of spaces a player moves is determined by the product of a contiguous subsequence of the values shown on the dice (which are available)(There are special rules for determining the range that vary each move that will not be discussed).If all the values are unavailable, the player moves one space. Iff the number on the dice is more than 1,000,000,000 or less than 0, the dice is unavailable.

To make matters more complicated, after any turn if Alice and Bob land on the same cell the value shown on all dice(neither available nor unavailable) is multiplied by the label of the cell they are on. Note in this way it is possible for some dice to show numbers greater than 1,000,000,000.

After playing this game for a while, Alice and Bob have grown frustrated because the calculations became too difficult. Given the series of  $R$  ( $1 \leq R \leq 100,000$ ) dice rolls and ranges, help Alice and Bob determine their position after each move. Assume that all dices start out showing 1 and all dices are available.

### Input

The first line contains  $R, N$  and  $M$  each separated by a space.  $R$  lines follow. Each line will contain  $d v a b$  separated by a space.  $d$  indicates the label of the dice rolled.  $v$  indicates the value shown on the dice.  $a$  and  $b$  indicate the range of dice used to determine the move distance.

### Output

$R$  lines containing the position of the player that just rolled after their roll.

### Sample

input	output
6 4 4	1
0 1000000000 1 1	2
1 999999998 1 1	2
2 500000000 3 3	2
0 1 2 2	0
3 1 0 3	0
0 6 0 3	

---

### Solution:

---

Prerequisites: set, segment tree.

The segment tree was used to solve this task. You can read about it here:

[http://en.wikipedia.org/wiki/Segment\\_tree](http://en.wikipedia.org/wiki/Segment_tree) Basically, it includes these operations (used in this task):

- void Create(int vertex, int leftindex, int rightindex) - initializes the segment tree.
- void Update(int vertex, int leftindex, int rightindex, int modifiedindex) - modifies the value of the required member.
- int Get(int vertex, int leftindex, int rightindex, int queryleftindex, int queryrightindex) - the value

obtained in the given interval.

- void Union(int vertex) - the information to "vertex" is written from its children.
- void Down(int vertex) - the information from "vertex" is written to its children.

The following data is stored in each interval:

- the result of multiplication of all available members in this interval; (1)
- the constant of multiplication with which members of smaller intervals should be multiplied to get the current correct result; (2)
- the number of available (defined in the task) members in the interval. (3)

This segment tree can quickly update one member and to get the result of multiplication in the given interval (each query is done in  $O(\log n)$  time).

However, it cannot handle situations when members become unavailable or becomes available again. When these situations arise?

Members become unavailable: when multiplied by the positive constant, they become greater than 1,000,000,000. Furthermore, we multiply all members then. How to list these elements quickly? Well, if we keep them sorted from the greatest to the lowest, we can take some amount of members from beginning until they don't exceed 1,000,000,000 when multiplied. We change these members to infinity and update them in the segment tree. Also we don't need to multiply other members one by one. We can have a constant of multiplication and to multiply it. Notice, that to multiply all members in the segment tree we need to modify only "vertex" = 1.

Members become available: only multiplying by 0 can do that. When multiplying all members by 0, we change all infinity members to 0 and we change all available members which are positive to 0. It is done with looping through all members.

You may wonder why the solution is effective while it can go through all members each query. This assumption is not right. When members become unavailable, only multiplying by 0 can change that (otherwise, we may ignore them). When we multiply by 0, we may ignore the members until they are modified one by one with each query (because multiplication cannot change members equal to 0).

So three sets are maintained: *nil*, *inf* and *S* (with available members sorted from the greatest to the lowest).

Also, notice, that we can perform calculations modulo  $m$ , so ints are usually enough and long longs are used to avoid overflows in expressions.

Data types and constants:

```
typedef long long ll;
typedef pair <int, int> ii;

const int Maxn = 10005;          // Maximum number of 'super-dice'
const int Maxm = 65536;        // Maximum number of vertices in the segment
tree
const ll Inf = 1000000001;     // Infinity (defined in the task)

// fraction + index of the member
struct frac {
    int a, b, ind;
    frac(int a = 0, int b = 0, int ind = 0): a(a), b(b), ind(ind) { }
    bool operator <(const frac &f) const {          // Bigger gets to the
beginning
        ll gota = ll(a) * f.b, gotb = ll(b) * f.a;
        if (gota != gotb) return gota > gotb;
        return ind < f.ind;
    }
};
```

```
// the node of the segment tree
struct node {
    int res, cnt; // described above (1), (3)
    int flag;    // described above (2)
    node(int res = 0, int cnt = 0, int flag = 0): res(res), cnt(cnt),
flag(flag) { }
};
```

Required variables:

```
int r, n, m;
ii has[Maxn]; // The fraction of the member (actual value is mult
/ has[i].second * has[i].first)
ll mult; // the constant of multiplication (described above)
set <frac> S; // described above
set <int> nil, inf; // described above
node st[Maxm]; // the segment tree
```

Additionally, it is important to know how to multiply all available members (number of which is "cnt") in the interval by the constant "C". The answer is  $\text{current\_result} * \text{Power}(C, \text{cnt}) \% m$ , where  $\text{Power}(C, \text{cnt}) = C^{\text{cnt}}$ .  $a^p$  (in  $O(\log p)$  time) is calculated this way:

```
int Power(int a, int p)
{
    if (a == 0) return p == 0? 1: 0;
    int res = 1;
    while (p) {
        if (p & 1) res = ll(res) * a % m;
        p >>= 1; a = ll(a) * a % m;
    }
    return res;
}
```

Operations with the segment tree:

```
void Create(int v, int l, int r)
{
    st[v] = node(l, r - l + 1, 1);
    if (l < r) {
        int m = l + r >> 1;
        Create(2 * v, l, m); Create(2 * v + 1, m + 1, r);
    }
}
```

```
void Down(int v)
{
    if (st[v].flag != 1) {
        if (st[2 * v].cnt) {
            st[2 * v].res = ll(st[2 * v].res) * Power(st[v].flag, st[2
* v].cnt) % m;
            st[2 * v].flag = ll(st[2 * v].flag) * st[v].flag % m;
        }
        if (st[2 * v + 1].cnt) {
            st[2 * v + 1].res = ll(st[2 * v + 1].res) *
Power(st[v].flag, st[2 * v + 1].cnt) % m;
            st[2 * v + 1].flag = ll(st[2 * v + 1].flag) * st[v].flag
% m;
        }
    }
}
```

```

        }
        st[v].flag = 1;
    }
}

void Union(int v)
{
    st[v].res = ll(st[2 * v].res) * st[2 * v + 1].res % m;
    st[v].cnt = st[2 * v].cnt + st[2 * v + 1].cnt;
}

void Update(int v, int l, int r, int x)
{
    if (l == r) st[v] = has[l].first >= Inf? node(1, 0, 1): node(ll(mult)
/ has[l].second * has[l].first % m, 1, 1);
    else {
        int mid = l + r >> 1;
        Down(v);
        if (x <= mid) Update(2 * v, l, mid, x);
        else Update(2 * v + 1, mid + 1, r, x);
        Union(v);
    }
}

int Get(int v, int l, int r, int a, int b)
{
    if (l == a && r == b) return st[v].res;
    else {
        int mid = l + r >> 1;
        Down(v);
        int res = 1;
        if (a <= mid) res = ll(res) * Get(2 * v, l, mid, a, min(mid,
b)) % m;
        if (mid + 1 <= b) res = ll(res) * Get(2 * v + 1, mid + 1, r,
max(mid + 1, a), b) % m;
        return res;
    }
}

```

The initialization of the data structures:

```

void Init()
{
    for (int i = 0; i < n; i++) {
        has[i] = ii(1, 1); // 1/1 = 1
        S.insert(frac(1, 1, i));
    }
    mult = 1; // we can say that all members were
multiplied by 1
    Create(1, 0, n - 1);
}

```

The function which changes the value of one dice:

```

void Change(int d, int v)
{
    // deleting

```

```

    if (has[d].first >= Inf) inf.erase(d);
    else if (has[d].first == 0) nil.erase(d);
    else S.erase(frac(has[d].first, has[d].second, d));
    // inserting
    has[d] = ii(v, mult);
    if (v) S.insert(frac(v, mult, d));
    else nil.insert(d);
    Update(1, 0, n - 1, d);
}

```

The function which multiplies all the numbers by the constant "mmult":

```

void Multiply(int mmult)
{
    if (st[1].cnt) { // multiply all elements in the segment tree
        st[1].res = ll(st[1].res) * Power(mmult, st[1].cnt) % m;
        st[1].flag = ll(st[1].flag) * mmult % m;
    }
    if (mmult == 0) { // modify all members to 0
        for (set <int>::iterator it = inf.begin(); it != inf.end(); )
        {
            has[*it] = ii(0, 1); Update(1, 0, n - 1, *it);
            nil.insert(*it); inf.erase(it++);
        }
        for (set <frac>::iterator it = S.begin(); it != S.end(); ) {
            has[it->ind] = ii(0, 1); Update(1, 0, n - 1, it->ind);
            nil.insert(it->ind); S.erase(it++);
        }
    } else {
        mult *= mmult;
        for (set <frac>::iterator it = S.begin(); it != S.end() && mult
/ it->b * it->a >= Inf; ) {
            has[it->ind] = ii(Inf, 1); Update(1, 0, n - 1, it-
>ind);
            inf.insert(it->ind); S.erase(it++);
        }
    }
    if (S.empty()) mult = 1; // to avoid overflows in the long run
}

```

All that is left to implement the main function which can solve the actual task:

```
int main()
{
    scanf("%d %d %d", &r, &n, &m);
    Init();
    int pos[2] = {0, 0};    // the positions of the first and the second
player
    int cur = 0;
    while (r--) {
        int d, v, a, b; scanf("%d %d %d %d", &d, &v, &a, &b);
        Change(d, v);
        pos[cur] = (pos[cur] + Get(1, 0, n - 1, a, b)) % m;
        printf("%d\n", pos[cur]);
        cur = !cur;        // 0 -> 1, 1 -> 0
        if (pos[0] == pos[1]) Multiply(pos[0]);
    }
    return 0;
}
```

---

**Added by:** [Zhang Taizhi](#)

**Resource:** [Based on Super Dice Game \(ID: 2833\)](#)

**Solution by:**

*Name:* [Karolis Kusas](#)

*School:* [Kaunas University of Technology](#)

*E-mail:* [karolis.kusas@gmail.com](mailto:karolis.kusas@gmail.com)

---

---

## Problem R1 07: Snakes and Ladders Again (ID: 13092)

---

Time Limit: 1.0 second

Memory Limit: 1536 MB

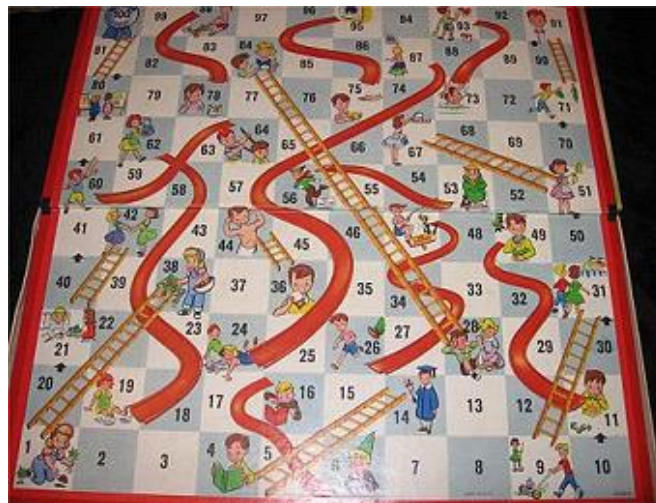
Snakes and Ladders (or Chutes and Ladders) is an ancient Indian board game regarded today as a worldwide classic. It is played between two or more players on a game board having numbered squares (fields) on a grid. A number of "ladders" and "snakes" (or "chutes") are pictured on the board, each connecting two specific board squares. The object of the game is to navigate one's game piece from the start (Bottom square) to the finish (Top Square), helped or hindered by ladders and snakes, respectively. The historic version had root in morality lessons, where a player's progression up the board represented a life journey complicated by virtues (ladders) and vices (snakes). If, after throwing a dice, a player's token lands on the lower-numbered end of a "ladder", the player moves his token up to the ladder's higher-numbered square. If he lands on the higher-numbered square of a "snake" (or chute), he must move his token down to the snake's lower-numbered square. If any of those cases takes place, we will call a square unstable. Otherwise it is stable.

The game is a simple race contest lacking a skill component, and is popular with young children.

In this problem you're required to calculate the expected number of 6-sided die throws to move your game piece from the start (bottom square) to the finish (top square).

### Formal game description

Fields are arranged on an  $N \times M$  grid and numbered from 1 to  $N * M$ . Last field, indicated by  $N * M$ , is referred to as Top Square. Each player starts with a token on a square at position "0" (the imaginary space beside the "1" grid field; Bottom Square), which is always stable. So in total we have  $N * M + 1$  fields. In every turn player throws the die and moves up by the given number of squares. If that would result in a field higher than Top Square, then token is not moved. If the square that token ends on is unstable, it is moved as indicated by ladder or snake. This is repeated until token is placed a stable field. You can assume that a stable field can be reached from any field on the board. If this final, stable field is the Top Square, game ends and player wins.



### Input

Input contains multiple test cases. First line of each test case contains integers  $N, M, S, L$ , where  $n$  and  $m$  are the board dimensions,  $N$  ( $0 < N \leq 10$ ),  $M$  ( $0 < M \leq 10$ ), and  $S$  and  $L$  are number of snakes and ladders respectively. Next  $S$  lines describes snakes. Each line contains two integers:  $h$  and  $t$ , where  $h$  is the snake's head position and  $t$  is the snake tail position. ( $0 < t < h \leq N * M$ ). Next  $L$  lines describes ladders. Each line contains two integers:  $p$  and  $q$  where  $p$  is the ladder's bottom and  $q$  is the ladder's top ( $0 < p < q < N * M$ ).

The input will be terminated by the end of file.

**NOTE!** There could be more snakes and/or ladders leading from a single field. In such a case use the last snake/ladder specified in the input.

## Output

Print one number per test case (each in separate line), expected number of dice throws needed to reach the Top Square. It's guaranteed that the Top is always reachable. You round the result to exactly 3 decimal places.

## Sample

input	Output
5 10 3 5 16 6 47 26 49 11 1 38 4 14 9 31 40 42 36 44	30.198

---

## Solution:

In the beginning, let's notice that only the stable fields are interesting for us. Why? Because we always roll a die from a stable field and move (possibly through a number of unstable positions) to another stable field. As it is guaranteed that a stable field can be reached from any field on the board, we can easily precompute for each stable field  $s$  and thrown number  $x$ :  $M(s, x)$  = where you have to advance from field  $s$  after getting  $x$  at the dice.

To solve the problem we'll need to compute  $E(s)$  = expected number of dice throws needed to reach the Top Square from every field  $s$  and print  $E(0)$ . Obviously, from the Top Square we don't move at all, so  $E(N * M) = 0$ , but what about the other fields?

From the definition of expected value we have:

$$E(s) = 1 + \frac{1}{6} \sum_{x=1}^6 E(M(s, x))(*)$$

Before we say how to compute it, let's consider a slightly easier version of the problem, when there are no snakes, only ladders. That means that we always go ahead or (in the worst case) stay in place, never move back. Then we have  $M(s, x) \geq s$  and can compute desired values from the definition, using dynamic programming approach in the order of decreasing values of  $s$ .

What about the general case? We see that the definitions (\*) give us system of linear equations and we can solve it in  $O((N * M)^3)$  time using standard Gaussian elimination algorithm.

---

*Added by: abdelkarim*

*Resource: The First Palestinian Collegiate Programming Contest*

*Solution by:*

*Name: Bartłomiej Dudek*

*School: University of Wrocław*

*E-mail: bardek.dudek@gmail.com*

---

---

## Problem R1 08: Pythagorean triples (medium) (ID: 14542)

---

Time Limit: 4.0 second

Memory Limit: 256 MB

---



Pythagoras is credited, by tradition, for the first proof of the relation  $a^2 + b^2 = c^2$  in any right angled triangle where  $c$  is hypotenuse and  $a$  and  $b$  are the catheti. We define a Pythagorean triple as a set of three positive integers  $a$ ,  $b$ , and  $c$  which satisfy the above equation, ie,  $a^2 + b^2 = c^2$ .

$\{3,4,5\}$  is the most common example of such triples.

### Input

The first line of input contains an integer  $T$ , the number of test cases. Each of the next  $T$  lines contains two integers  $N, M$ .

### Output

For each test case, print on a single line the number of Pythagorean triplet  $\{a, b, c\}$  such that  $N \leq a, b, c \leq M$ .

### Sample

input	output
3	1
1 5	1
4 10	45
10 100	

### Constraints

$$0 < T < 100$$

$$0 < N < M$$

$$0 < T \times M < 1.21 \times 10^8$$

---

### Solution:

---

First, recall Euclid's formula for generating Pythagorean triples given a pair of integers  $m, n$  where  $m > n$ :

$$a = m^2 - n^2$$

$$b = 2mn$$

$$c = m^2 + n^2$$

The above formula generates all primitive Pythagorean triples. A primitive Pythagorean triple is a Pythagorean triple  $\{a, b, c\}$  where  $GCD(a, b, c) = 1$ . A Pythagorean triple generated by the above formula is primitive if  $m$  and  $n$  are coprime and  $m - n$  is odd. Every Pythagorean triple where  $GCD(a, b, c) = d > 1$  can, of course, be derived from a primitive triple by multiplying some triple  $\{a', b', c'\}$  by  $d$ . The solution works by generating all primitive Pythagorean triples and then for each such triple checks in  $O(1)$  time how many of the derived Pythagorean triples fit into the given constraints. It is clear that for a given value of  $M$ , the number of primitive Pythagorean triples satisfying  $a, b, c < M$  is  $O(M)$ , since the number of possible values of  $m$  is  $O(\sqrt{M})$  and  $n < m$ , so we have  $O(\sqrt{M}\sqrt{M}) = O(M)$ .

The outer loop of the solution iterates over all values of  $m$ . After that, all positive integers less than  $m$  and coprime with  $m$  can be sifted out by the following algorithm:

```

Set V[] to false;
for each i from 2 to m do
    if (i divides m) and (V[i] = false) then

```

```
        for each j from i to m in steps of i do
            V[j] = true
        endfor
    endif
endfor
```

This algorithm is slightly faster than directly computing  $GCD(m, n)$  for all values of  $n$ . Afterwards, all the values of  $n$  coprime with  $m$  will have  $V[n] = false$ . The inner loop iterates over values of  $n$  where  $GCD(m, n) = 1$  and  $m - n$  is odd. Then, we compute  $a, b, c$  and find  $L = \min(a, b, c)$  and  $H = \max(a, b, c)$ . The latter is clearly equal to  $c$ . Then, we just need to find the number of positive integers  $d$  such that  $N \leq ad, bd, cd \leq M$ , that is,  $N \leq Ld$  and  $Hd \leq M$ . This can, of course, be done in  $O(1)$  time by finding the minimum and maximum value of  $d$ . We add the newly found number to the total number of triplets and in the end we just print that value.

---

*Added by: Francky*

*Solution by:*

*Name: Ivan Stošić*

*School: Gimnazija Svetozar Marković Niš*

*E-mail: ivan100sic@gmail.com*

---

---

## Problem R1 09: Foxic Expressions (ID: 14975)

---

Time Limit: 1.0 second

Memory Limit: 1536 MB

Let's talk about some definitions, shall we?

- An uppercase letter is a character between "A" and "Z", inclusive. You knew that.
- A string is a sequence of characters. You probably knew that.
- A Foxic letter is a superior uppercase letter - namely, one of "F", "O" or "X". You probably didn't know that.
- A Foxic string is a superior string, consisting only of Foxic letters. You didn't know that.

Finally, a Foxic expression is a special string, with each of its characters being either a Foxic letter, or an "n" immediately following a Foxic letter. A Foxic expression can be translated into a Foxic string by a three-step process. First, up to one character can be added, removed, or modified, provided that the resulting string is still a valid Foxic expression. Next, every Foxic letter immediately preceding an "n" is replaced by zero or more occurrences of that same letter. Finally, each "n" is removed. You most certainly did not know that.

There are  $T$  ( $1 \leq T \leq 100$ ) scenarios to consider, as described above. In each scenario, given a Foxic string  $S$  of length  $N$  ( $1 \leq N \leq 100$ ) and a Foxic expression  $E$  of length  $M$  ( $1 \leq M \leq 100$ ), you'd like to determine whether or not  $E$  can be translated into  $S$ .

### Input

Line 1: 1 integer,  $T$

For each scenario:

Line 1: 1 integer,  $N$

Line 2: 1 string,  $S$

Line 3: 1 integer,  $M$

Line 4: 1 string,  $E$

### Output

For each scenario: The string "Yes" (without quotes) if  $E$  can be translated into  $S$ , or "No" otherwise.

### Sample

input	output
2	Yes
5	No
OOOFO	
7	
OXnFOXn	
3	
FOX	
7	
OFnOXnO	

### Explanation

In the first scenario, one possible course of action is to erase the second character of  $E$ , leaving the Foxic expression "OnFOXn". Next, we may choose to replace the first "O" with three copies of "O", and the remaining "X" with zero occurrences of "X", since each of these precedes an "n" - this yields the string "OOOnFOn". Finally, after removing each "n", we are left with "OOOFO", which matches  $S$ . Replacing the second character with an "O" would have also been possible.

In the second scenario, it is impossible to translate E into S through any valid steps.

---

**Solution:**

---

This is a problem which could be easily summarized in a single sentence as “regular expressions with a twist”. It is a simple problem for anyone who’s familiar with regular expressions (regex for short from now on), so it would be rather useful to start off with precisely defining a few key terms about regexes:

An **alphabet** is specified as any *finite set of symbols*,  $\Sigma$ .

For example, the set of lowercase English letters,  $\Sigma = \{a, b, c, \dots, z\}$  is an alphabet, while the set of natural numbers,  $\mathbb{N} = \{1, 2, 3, \dots\}$  is not.

A **string of (non-negative) length  $n$**  over an alphabet  $\Sigma$  is an ordered  $n$ -tuple of elements of  $\Sigma$  written without punctuation. There is a unique string for which  $n = 0$ ; we call it the **null string** and usually denote it with  $\varepsilon$ . The **set of all strings** over an alphabet  $\Sigma$  is denoted as  $\Sigma^*$ .

If we use the set of lowercase English characters as our alphabet  $\Sigma$ , some members of  $\Sigma^*$  would be *abcdef*, *jjj*, *x*,  $\varepsilon$ , *zzz*, etc.

The **concatenation** of two strings  $u, v \in \Sigma^*$  is a string  $uv \in \Sigma^*$  obtained by joining the two strings together end-to-end.

For example (still using the English characters as our alphabet), if  $u = abc$  and  $v = xyz$ , then  $uv = abcxyz$ .

A **regular expression** over an alphabet  $\Sigma$  can be defined as follows:

- each symbol of the alphabet,  $a \in \Sigma$ , is a regular expression;
- $\varepsilon$  is a regular expression;
- $\emptyset$  is a regular expression;
- if  $r$  and  $s$  are regular expressions then so is  $(r|s)$ ;
- if  $r$  and  $s$  are regular expressions then so is  $rs$ ;
- if  $r$  is a regular expression then so is  $(r)^*$
- every regular expression over  $\Sigma$  can be built by applying these rules finitely many times.

Regular expressions represent a language for representing patterns, and as such are commonly used for pattern-matching; here are **rules** based on which we can **match** a string  $u \in \Sigma^*$  to a regex:

- $u$  matches  $a \in \Sigma$  iff  $u = a$ ;
- $u$  matches  $\varepsilon$  iff  $u = \varepsilon$ ;
- nothing matches  $\emptyset$ ;
- $u$  matches  $r|s$  iff  $u$  matches  $r$  or  $u$  matches  $s$ ;
- $u$  matches  $rs$  iff it can be expressed as a concatenation of two strings,  $u = vw$ , such that  $v$  matches  $r$  and  $w$  matches  $s$ ;
- $u$  matches  $r^*$  iff either  $u = \varepsilon$ , or it can be expressed as a concatenation  $u = aaaaaaaaa \dots a$  such that  $a$  matches  $r$ .

Now it should be fairly clear to notice that the “foxic expressions” referred to in the problem statement can be expressed as regular expressions over  $\Sigma = \{F, O, X\}$ , with the string  $an$  (where  $a \in \Sigma$ ) representing  $(a)^*$ . The problem is now reduced to generating all possible regexes with the allowed rules, and checking if any of them matches the given string.

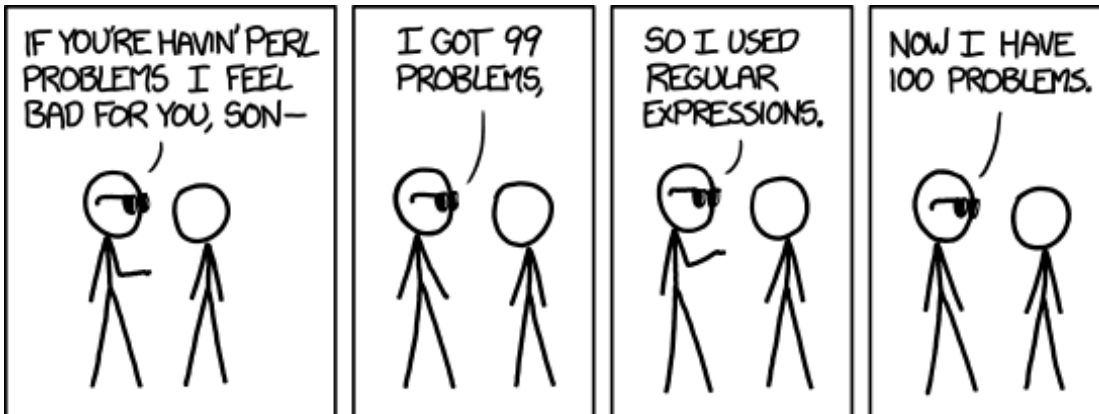
Once we have a regex we want to use for pattern-matching, we can use it to construct a graph called a **deterministic finite automaton**, which, once constructed, determines in time  $O(n)$  whether a given string of length  $n$  matches that regex. The construction methods for this graph will not be covered in this solution, as several programming languages (including Java, which is one of the allowed languages in the qualifications) have regex pattern-matching as an in-language feature. Interested readers are advised to look up *Automata*

and computability by Kozen, D.C. (1997) for further insight into these algorithms and in general into the very exciting field of computation theory.

Regexes are a very powerful tool, commonly utilized in terminals (Unix shell/Windows cmd); a simple example would be the shell command

```
$ ls *.txt
```

which will extract all files with a plain text extension from the current working directory. The concept is central to a few programming languages as well – it likely explains why the only accepted solution to this problem before appearing at BubbleCup qualifications was in Perl.



<http://xkcd.com/1171/>

---

**Added by:** *Jacob Plachta*

**Solution by:**

*Name: Petar Veličković*

*School: University of Cambridge*

*E-mail: pv273@cam.ac.uk*

---

## Problem R1 10: [CH] Japan Crossword (ID: 316)

Time Limit: 21.0 second

Memory Limit: 256 MB

Japan crossword is a very popular game. It represents encoded picture which consists of filled block of cells. At the start of game you see empty grid. Each row (column) has some numbers in beginning of the row (column). Each number means how many continuous cells are filled in a hidden picture (length of the filled blocks). Filled blocks of cells are arranged from left to right and from top to bottom. Between filled blocks must be at least one empty cell. For example, numbers are 4, 2, 7 mean that there are three groups with 4, 2, and 7 filled cells in it. Your task is decode hidden picture using hints.



### Input

The first line of input contains a single positive integer  $t \leq 300$  - the number of test cases. Then for every test case first line specifies integer numbers  $R$  and  $C$  (number of rows and columns) of the picture ( $1 \leq R \leq 50, 1 \leq C \leq 100$ ). Below  $R$  lines are follow. Each line consists of any integers for horizontal hints. The very last number for every line is 0. Then  $C$  lines are follow. Each line consists of any integers for vertical hints. And again every line ends with 0.

### Output

For every test case you should write decoded picture in the form of rectangle with  $R$  rows and  $C$  characters in each line. Symbol '#'(sharp) means filled block and symbol '.'(point) means empty cell.

### Score

The score awarded to your program is the total of all scores obtained for its individual test cases. The score for a test case is calculated so that for each 'right' row or column you get 1 points. The row(column) is counted as a 'right' if there is a group of filled cells for every number in beginning of the row(or column) and length of every cell is equal corresponded number. If All rows and columns are 'right' your score multiply by 1.5 for this test case.

### Sample

input	output
1	.###.
10 5	##.##
3 0	#####
2 2 0	#####
5 0	.###.
5 0	..#..
3 0	..#..
1 0	..###
1 0	..##.
3 0	..###
2 0	
3 0	
3 0	

5 0	
1 8 0	
5 3 0	
3 1 1 0	

**Score:**

$$(10 + 5) * 1.5 = 22.5$$

**Solution:**

**Line solver:**

When you start to solve this problem first thing that comes to mind is to code a method that will extract all known fields (fields that can be only black or only white) of a single line. A fast way to do this is with dynamic programming. (We check for each  $i$  and  $j$  if  $i - th$  field of the line can be the last field of  $j - th$  black run. A field  $i$  can be black if there is such  $j$  and  $k$  that  $j - th$  field can be the end of  $k - th$  run and it contains field  $i$ . Similar to previous conclusion a field  $i$  can be white if there is such  $j$  so that you can fit first  $j$  runs in fields before  $i$  and the rest in fields after  $i$ .) This can be done in  $O(N * K)$  where  $N$  is the length of the line and  $K$  is the number of black runs in the line.

**Using line solver:**

Now that we have foundation we can start to really solve the puzzles. If we iterate the "line solver" on all rows and columns while we are getting changes on the table we can solve 173/252 test cases. (A good thing to notice is that there is no need to "line solve" a row or column if nothing has changed in it. (ie. we have not marked any more fields in it black or white). This can save major time.)

**1-contradiction:**

In the other 79 cases looking at a single line at a time will not help so we need to make some guesses. First we can pick a field of the table and mark it black or white. If the crossword now has no solution we can be sure that the field we picked is the opposite color of the one we chose so we can color it and go to solving the new table. (We run the "line solver" on each line again.) This solution will give us 232/252 test cases.

**Boards with many solutions:**

So far our solution seems to be doing good on finding a solution if the crossword has unique solution. But what about a case where you have only 1 black field in each row and column. The board has  $N!$  solutions. ( $N$  being the size of the table.) To deal with this we can make multiple guesses. We can do a bit of recursion: we pick a field and color it see what it gives us. (We run the line solver on the new table) There are couple of cases:

1. It leads us to a solution :)
2. It gives us a contradiction in which case we go back a step and color that field the other color.
3. It only makes a few more fields known. In that case we make more guesses.

Now we need a way to pick which field we try to color. A way that worked for me is to pick one where there are the least unknown fields in its row and column. (It turns up that you only need to make 4 – 5 guesses at maximum.)

This solution will find a valid configuration for all given test cases.

---

*Added by: Maxim Sukhov*

*Solution by:*

*Name: Marko Stanković*

*School: Gimnazija Svetozar Marković, Niš*

*E-mail: markostankovic996@gmail.com*

---



---

## Problem R2 01: Digital Image Recognition (ID: 3360)

---

Time Limit: 3.0 second

Memory Limit: 256 MB

According to Wikipedia, image processing is any form of signal processing for which the input is an image, such as photographs or frames of video; the output of image processing can be either an image or a set of characteristics or parameters related to the image. Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it.

The task you are facing here is a relatively easy one (compared to our general conception of image processing!). Given a black-and-white image of size  $R * C$  with some digits (and possibly other shapes) on it, your program needs to figure out the digits written on the image. Specifically, the digits drawn on the graph will adhere to the following rules:

- 1) Digits are drawn with a series of strokes. A **stroke** can be regarded as a **rectangle** of any size on the image, and its edges will always be parallel to either **x-axis** or **y-axis**. The number of strokes required to draw each digit will be exactly as follows:

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

Refer to the **figure below** if you are unclear about how the digits are drawn.

- 2) Although the **width** of strokes used to draw a digit might be **different**, the **outer shapes of digits** will strictly follow those specified in the **figure below**.
- 3) In order for a digit to be recognizable, **all** parts (**strokes** and **joints**) presented in the graph below must also be clearly **distinguishable** in the image.
- 4) You may assume that the image is not rotated, and there is **no noise** in the input.



Please output the sum of digits recognizable in the graph. In the case that no characters is recognizable, please output 0 instead.

### Input

There are multiple test cases in the input file.

Each test case starts with two integers,  $R$  and  $C$  ( $1 \leq R, C \leq 500$ ), specifying the number of rows / columns of the graph. Each of the following  $R$  lines contains consecutive  $C$  characters ("0" or "1"), describing the image to be processed.

Two successive test cases are separated by a blank line. A case with  $R = 0, C = 0$  indicates the end of the input file, and should not be processed by your program.

### Output

For each test case, please print a single integer, the sum of recognizable numbers. See the sample output for format details.

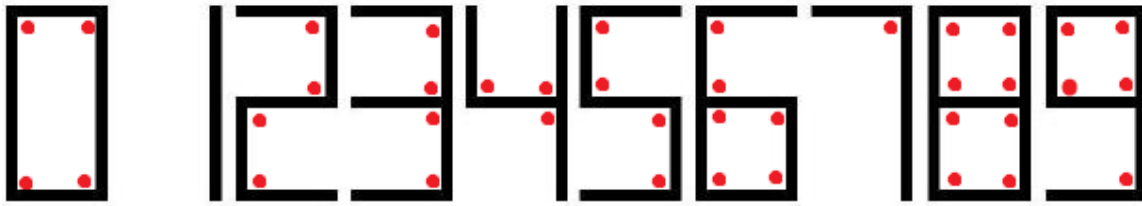
### Sample

input	output
5 12 001101011111 000101000011 000101001111 001101000011 000000000111	Case #1: 4 Case #2: 0 Case #3: 15 Case #4: 3 Case #5: 2
5 3 111 010 110 010 110	
6 14 11111000011111 11001000000011 11111001000000 11111001001110 11001011001010 11111000001110	
5 2 11 01 11 01 11	
6 9 111100111 000100001 000100011 011100010 010000011 011110000	
0 0	

---

### Solution:

Our task here is to differentiate between 10 digits (0 – 9), so we don't need a fully-scalable solution. Thus, we can focus on *almost* hard-coding each digit. By far the easiest way to go about it would be to count the number of inner edges (pic 1), and determine the position of each edge relative to other edges. Inner edges are given in red.



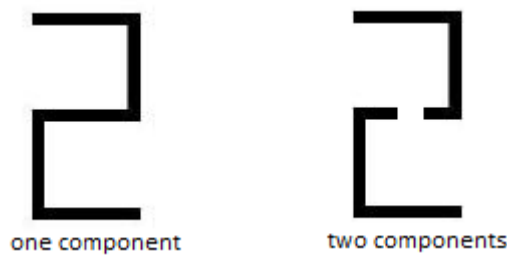
pic 1

First, we can do **BFS (breadth-first search)** to isolate our figure. Next, we count the number of inner edges for that figure. If the number of inner edges matches the number of inner edges in any of our digits (code 1), we can then continue to check the relative positions of those edges.

```
char innerEdgesCount[] = { 4, 0, 4, 4, 3, 4, 6, 1, 8, 6 };
```

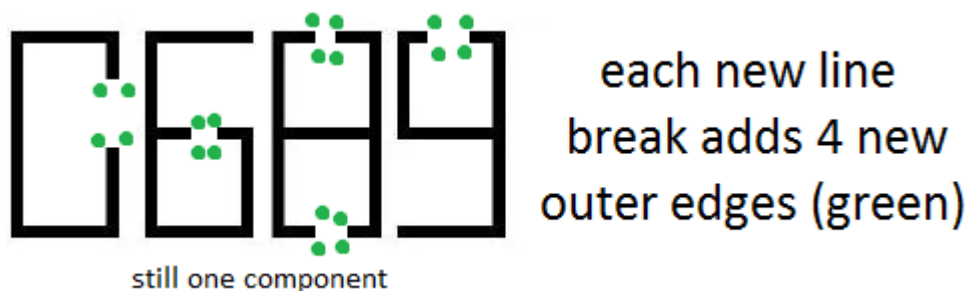
code 1

If their relative positions are correct, we know that we have a match. The reason why this solution works is simple. We can define each digit with their number of points (dots, digit zero has 4 dots, digit one has 2 dots, etc.), their relative positions and lines which connect those dots. With BFS we make sure that the component is connected (pic 2).



pic 2

With this in mind, we can notice that we can still have errors. Digits 0, 6, 8 and 9 can still be one component, but have their lines broken (pic 3). Therefore, we can count the number of **outer edges**, since breaking a line adds 4 new outer edges. Thus, we have defined each number with the **number of outer edges and relative positions of inner edges**.



pic 3

---

*Added by: [Trichromatic] XilinX*  
*Resource: ACM/ICPC Asian Regional Contest, Hangzhou 2008*  
*Solution by:*  
*Name: Saša Vučković*  
*School: School of Computing*  
*E-mail: sashans13@hotmail.com*

---

---

## Problem R2 02: FLING1 (ID: 13884)

---

Time Limit: 2.0 second  
Memory Limit: 1536 MB

Fling! is a popular puzzle game created by the well-known developers at CandyCane LLC.



The premise of the game is simple. You are given a certain number of balls on the screen to start. The goal is to fling one into another in order to knock the other off the screen. The puzzle is considered solved if you can do so while leaving only one ball remaining on the screen. Some might read this and think that it might not be too difficult, but the game gets challenging quickly. The problem is that you cannot fling two balls that are adjacent (i.e. next to) each other.

The first ball you choose can fling the 2nd ball if and only if:

- 1) The two balls exist in same row or same column
- 2) The two balls are not adjacent
- 3) There is no other ball in between the two balls

If there exist a 3rd ball after the 2nd ball in the same line of action, the 2nd ball takes the position just before the third ball, pushes the 3rd ball and the 3rd ball gets flinged. (This continues till a ball gets knocked off the screen. Note that 2nd ball and 3rd ball can be adjacent).

Given a Fling! puzzle, just print "Yes" if it is a valid puzzle(solvable) or "No" otherwise.

For better understanding of gameplay you may have a look at this video. (optional)

### Input

The first line of the input consists of an integer  $t$ , the number of test cases. For each test case, the first line consists of two integers  $m$  and  $n$ , the number of rows and columns of the puzzle. Then follows the description of the board.  $A[i][j]$  is '.' if the cell is empty or 'B' if the cell has a ball.

### Output

For each test case print "Yes" if the puzzle is valid or "No" otherwise. (case-sensitive).

### Constraints

- $1 \leq t \leq 100$
- $1 \leq m, n \leq 10$
- You can assume that the number of balls in the board is approximately equal to  $(m * n) / 10$

### Sample

input	output
4	Yes
5 5	Yes
....	No
....	Yes
..B..	
....	
.B..B	
5 4	
....	
B...	
B...	
....	
B...	
3 4	
BB..	
....	
.B..	
1 1	
B	

---

### Solution:

---

FLING1 was the easiest task of Round 2. The solution is very simple: to find if the given puzzle is valid or not, we just have to try all the possible combinations. If, at any time, we end up with just one ball on the board, we have a valid puzzle. We can achieve this by using **DFS (depth-first search)**.

We might encounter a problem with memory, as saving the whole matrix for each move is expensive. Since every move alters only one row/column, and the rest remains the same, we can reduce memory usage by saving only a part of the board. Another valid optimization exploits the fact that we are given approximately  $M * N / 10$  balls in each test case, and saving only the positions of those balls would suffice. Furthermore, in each iteration, instead of saving the positions of all the balls, we can save each move made, thus reducing memory usage even more.

Worst case complexity of this solution is  $O(B! * 4^B)$ , where  $B$  stands for the number of balls. We derive this, the number of possible moves until the end of the game (in the worst case), by using well-known combinatorial formulas. Since there are  $T$  test cases, and there are approximately  $M * N / 10$  balls, the solution, at first sight, doesn't seem good enough. Upon closer inspection, we see that the theoretical worst case complexity can, in fact, never be achieved. The fact that in the majority of cases the game tree has a lot less than  $B! * 4^B$  nodes, which means that a lot of moves are impossible, leads us to understand that this, supposedly naive solution, fits the time limit, and is correct.

---

*Added by: cegprakash*

*Solution by:*

*Name: Nikola Jovanović*

*School: Matematička gimnazija*

*E-mail: nikolajovanovic96@yahoo.com*

---

---

## Problem R2 03: One Instruction Computer Simulator (ID: 2023)

---

Time Limit: 1.0 second

Memory Limit: 256 MB

Daniel is building towers out of blocks. He has many black and white blocks. He has built  $n$  towers out of those. Now he suggests Max playing the following game. Black block will belong to Daniel and white blocks will be Max's blocks. During his turn the player can take any of his blocks from any tower and remove it and all the blocks above it. As usual the player who can't make the move loses. Daniels make the first move. Determine who will win if both players play optimally.

### Input

The input starts with number  $t$  - the amount of test cases. The first line of each test is number  $n$  - the number of towers. Then  $n$  strings follow. Each string is formed of 'B' and 'W' characters, where 'B' means black block and 'W' - white block. Each string describes one tower from bottom to top.

### Output

For each test case print 'Win' if Daniel wins and 'Loss' if Max wins given both players play optimally.

### Sample

input	output
1 5 BBWWB BWBB BB WWW WB	Win

---

### Solution:

---

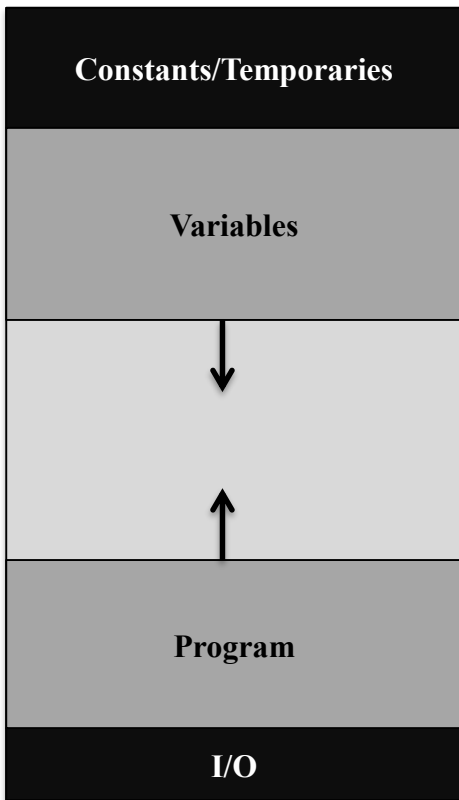
Here we are faced with a relatively standard problem of writing assembly code for a (single-instruction) ISA, to calculate expressions written in Reverse Polish Notation. This does not necessarily require an ingenious idea, but exploiting the fact that the assembly should be generated by a higher-level language can allow us to come up with subroutines that will make our job much easier. As such, the solution will first describe a few of those simple ideas, and then work on combining them in a complete solution for the problem.

The first thing to be discussed is the memory model for our processor. It is split into two regions: one reserved for I/O and the other serving as a regular Von Neumann general purpose memory. Since the same memory is used both for storing programs and data, care has to be taken not to have data corrupting the instructions and vice-versa. As such we will further partition the main memory region into three sub regions:

- the program region, which will contain the instructions;
- the constants and temporaries region, which will contain constants and other temporary storage registers used by our algorithms internally;
- the variables region, containing the variables we utilize as we unwind the expression.

The program region will start where instructions normally start (right after the I/O region), while the register regions will start on the opposite side – first the constants and temporaries as we can figure out exactly how much we will need. The program and variables regions then expand in opposite sides as the assembly is generated, thus minimizing the possibility of an overlap.

A good illustration of the concept can be seen on the image to the right, with the bottom representing the memory address 0, and the top representing memory address 9998.



Now that the memory model is sorted out, it is time to discuss the various subroutines we might find useful in simplifying our code. Here are a few that should be useful – note that from now on, ZR is referring to a special memory location in the constants/temporaries region, that we will always assert to be zero between subroutine calls. Additionally, when a SUBLEQ instruction is provided without a third argument, it is implied that the (possible) jump is to the location of the next instruction (i.e. current line + 3). TA, TB, TC etc. will refer to registers used to store temporary variables.

```
RESET (X) /* X := 0 */
      SUBLEQ X X
```

```
INC (X) /* X := X + 1 */
      SUBLEQ -1 X
```

```
SET (X) /* X := 1 */
      RESET X
      INC X
```

```
INV (X, Y) /* X := -Y */
      RESET X
      SUBLEQ Y X
```



```

ASSIGN(X, Y) /* X := Y */
    SUBLEQ Y ZR
    INV X ZR
    RESET ZR

```

```

IFLEQ(X, C) /* IF X <= 0 JMP C */
    SUBLEQ ZR X C

```

```

IFLEQ(A, B, C) /* IF A <= B JMP C */
    ASSIGN TD A
    ASSIGN TE B
    SUBLEQ TE TD C

```

```

JMP(C) /* JMP C */
    SUBLEQ ZR ZR C

```

```

DBL(X) /* X := X + X */
    SUBLEQ X ZR
    SUBLEQ ZR X
    RESET ZR

```

To completely generate our assembly, we will need core subroutines for the four main arithmetic operations. Addition and subtraction are simple to implement; here we demonstrate a possible implementation of them:

```

ADD(A, B, C) /* C := A + B */
    SUBLEQ A ZR // ZR := -A
    SUBLEQ B ZR // ZR := -(A + B)
    INV C ZR // C := A + B
    RESET ZR // ZR := 0

```

```

SUB(A, B, C) /* C := A - B */
    SUBLEQ A ZR // ZR := -A
    INV C ZR // C := A
    SUBLEQ B C // C := A - B
    RESET ZR // ZR := 0

```

Multiplication and division require significantly more work – this is due to the fact we must take into account the possibility of negative numbers as operands, and that our assembly will likely be too slow if we employ a naïve algorithm. Luckily, we can use the concepts from exponentiation by repeated squaring to produce fast algorithms – only this time we use doubling (addition) instead of squaring. This is how a simple multiplier would look (assuming we already know the operands are both positive), with the help of labels (which can simply be supported by f.ex. mapping instruction locations to strings before compiling a subroutine into assembly), and four additional temporary registers IP, II, BP and BI:

```

FASTMULP(A, B, C) /* C := A * B, A >= 0, B >= 0 */
    ASSIGN TA A // TA := A
    ASSIGN TB B // TB := B
    SET TC // TC := 1

```

```

    RESET C           // C := 0
LP:  IFLEQ TB EXIT   // LP: IF TB <= 0 JMP EXIT
    RESET IP         // IP := 0
    SET II           // II := 1
    RESET BP         // BP := 0
    ASSIGN BI TA     // BI := TA
NXT: INC TB          // NXT: TB := TB + 1
    IFLEQ TB II CNT  // IF TB <= II JMP CNT
    SUBLEQ TC TB     // TB := TB - 1
    ASSIGN IP II     // IP := II
    DBL II           // II := II + II
    ASSIGN BP BI     // BP := BI
    DBL BI          // BI := BI + BI
    JMP NXT         // JMP NXT
CNT: ADD C BP TD     // CNT: TD := C + BP
    ASSIGN C TD      // C := C + BP
    SUBLEQ IP TB     // TB := TB - IP
    SUBLEQ TC TB     // TB := TB - 1
    JMP LP         // JMP LP
EXIT: // EXIT: end of subroutine

```

This can be easily further extended to handle operands of various signs -- a 'flag' register would probably be useful here, set to be the XOR (exclusive disjunction) of the operands' signs, and if it was positive, we would invert the result obtained from running FASTMULP on the absolute values of A and B. Division can be handled in a similar way (omitted and left as an exercise to the reader).

With all the arithmetic subroutines in place, everything we need to do is perform the usual stack-based RPN evaluation algorithm, allocating a new variable in the variables region for each computation performed, and storing the variable locations in the stack. Finally, we do something along the lines of

```

FINISH() /* store result and exit */
    ASSIGN 0 (Stack.top())
    JMP 10000

```

...and we're done!

For me, this was a very interesting problem which required significant time spent in front of the whiteboard, and far less time spent actually coding the solution, which is always a good thing. One of the past exam papers at my university actually featured a full question on SUBLEQ, so if you find such material engaging, you should check it out on this link: <http://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2004p5q2.pdf>. Have fun!

---

*Added by: Chen Xiaohong*

*Resource: Changed and Enhanced from Colombian National Contest*

*Solution by:*

*Name: Petar Veličković*

*School: University of Cambridge*

*E-mail: pv273@cam.ac.uk*

---

---

## Problem R2 04: Fight with functions (ID: 3902)

---

Time Limit: 12.0 second

Memory Limit: 256 MB

Multiplicative functions are defined as functions such that  $f(m * n) = f(m) * f(n)$ . Now, we put an extra constraint on multiplicative functions that if  $m$  and  $n$  are coprime, then  $f(m)$  and  $f(n)$  are also coprime. Additionally it is also provided that  $f(1) = 1$ .  $f(x)$  is defined for positive integers and it returns positive integers.

Now, you are provided with some  $x$  and corresponding  $f(x)$ . Your task is to find out, if you can uniquely determine the value of  $f(y)$  given  $y$  and if yes, find the value.

### Input

The first line of input contains a number representing the number of test cases. For each test case, the first line contains a number  $N$  representing the number of  $(x, f(x))$  pairs to be provided.  $N$  Lines follow, each line containing a pair of space separated numbers : the first one corresponding to  $x$  and second one to  $f(x)$ . Next line contains  $q$ , the number of queries.  $q$  lines follow, each containing a number  $y$ .

### Output

For each test case output  $q$  lines, one corresponding to each query. The output should contain "YES  $f(y)$ " where  $f(y)$  is replaced by the integer denoting  $f(y)$  with no leading zeroes if given the data, we can uniquely determine  $f(y)$ , or "NO" if the input data is inconsistent with the properties of the function or with the given information provided about the function, we cannot uniquely determine  $f(y)$ .

### Sample

input	output
3	NO
3	YES 6
2 2	YES 12
3 2	
7 19	
1	
7	
1	
6 6	
1	
6	
2	
2 2	
3 3	
1	
12	

### Constraints

The number of test cases are less than 20.  $N \leq 50$ .  $x$  and  $f(x) \leq 10^{50}$ .  $x$  and  $f(x)$  do not have a prime factor greater than 100005.

The number of queries are less than or equal to 50. Each number in the query is less than  $10^{50}$ . You are guaranteed that if the answer is unique, it contains less than 400 digits.

---

**Solution:**

---

First, notice that because our function is multiplicative we only care about values it attains for prime numbers. We will represent our function as infinite weighted bipartite graph. On both sides we are going to have nodes enumerated as prime numbers.

For each prime number  $p$  on the left side and for each number  $q$  on the right side of the graph such that  $q$  divides  $f(p)$  we will add edge going from  $p$  to  $q$  with weight equal to largest power of  $q$  that divides  $f(p)$ . For example, if  $f(3) = 16$ , then there is edge from 3 to 2 with weight 4.

Now, if we know that  $f(x) = y$ , we are going to factorize both  $x$  and  $y$  and iterate over all pairs of prime number  $p, q$  such that  $p$  divides  $x$  and  $q$  divides  $y$ .

Let's say that  $p$  divides  $x$  with power of  $a$  and  $q$  divides  $y$  with power of  $b$ . If  $a$  divides  $b$  then we add edge from  $p$  to  $q$  with weight  $b / a$ , but if there already is edge from  $p$  to  $q$  with weight other than  $b / a$  or  $a$  doesn't divide  $b$  then we delete this edge and allow no more operations on this edge.

For each pair  $(p, q)$  we keep counter which we increase each time we encounter pair  $(p, q)$ . We also keep counter how many times we had  $p$  in factorization of  $x$  and how many times we had  $q$  in factorization of  $y$ . Next, we iterate over all edges  $(p, q)$  again and check if counters of  $p$  and  $q$  have the same value and if they don't then we erase edge  $(p, q)$ . After this we check if there is node  $q$  on the right side which has positive counter but no edges coming to it. If there exists such node then our function is invalid and we output 'NO' for each query.

Otherwise, our function is valid and we proceed to read queries.

To answer query for some  $x$ , we need two arrays. First array will be counter for each node on right side and second array will represent power to which we raise the value of this node in final result. We also need boolean variable which will be flag that we know what is value of our function. It is initialized as true. We factorize  $x$  and we iterate each prime  $p$  that is divisor of  $x$ , let's say that  $p$  divides  $x$  with power of  $c$ . If  $p$  has no outgoing edges we set our flag as false.

Now we iterate over all outgoing edges from  $p$ , let's say we have edge  $(p, q)$  with weight  $w$ . If counter of  $q$  is not zero and power value of  $q$  is not  $c * w$  flag is false, otherwise power of  $q$  must be  $c * w$ . Last, we initialize our result as one, and we iterate over all nodes on right side which have positive counter. For each such node we check that its counter is equal to its degree (otherwise we set flag as false) and we multiply results by value of that node raised to its power.

If flag is true we output our result, otherwise we output 'NO'.

---

**Added by:** *Race with time*

**Resource:** *Code Craft 09*

**Solution by:**

*Name:* **Mislav Balunović**

*School:* *Gimnazija Matija Mesić, Slavonski Brod*

*E-mail:* *mislav.balunovic@gmail.com*

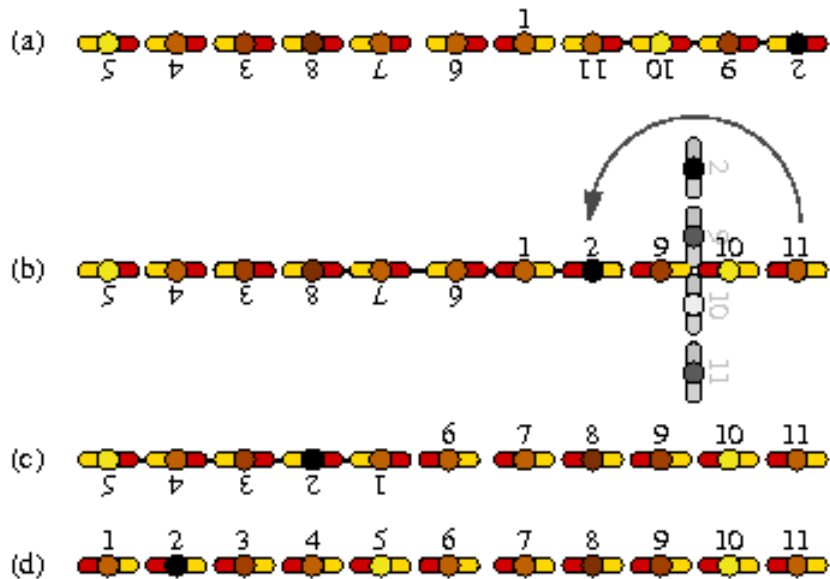
---

## Problem R2 05: Soccer Choreography (ID: 850)

Time Limit: 20.0 second

Memory Limit: 256 MB

Mr. Bitmann, the coach of the national soccer team of Bitland, is a perfectionist. He taught his players optimal tactics and improved their endurance and shape. So they qualified for the soccer world cup this year. Due to his perfectionism the coach attaches importance not only to the performance in the game but also before the game. So he told the team captain in what formation the team should assemble before the national anthem is played. Since each of the 11 team members has a unique number between 1 and 11 on his shirt, he can represent the formation as a permutation of numbers.



"Hmmm... I'll let my players dance!" A great idea! He took his notebook and started to create a choreography which leads to his expected formation. Due to the fact that no one of the players took dancing lessons he restricts his dance to one basic move: One player or more players who stand side by side can turn 180 degrees around the center of the move. Picture (b) contains an example: The players

$$-11 - 10 - 9 - 2$$

(we mark players which stand in the wrong direction with a minus) can do one move to

$$2 9 10 11$$

As perfect as he is he calculated a dance with a minimum number of moves. It works perfectly and now he's planning to do dancing performances with teams with more than 11 members. So he needs your help to find optimal dancing moves...

### Input

Each test case starts with the number of team members  $n$  ( $0 \leq n < 2200$ ). The next lines represent the formation at the beginning and the expected formation at the end of the choreography.

### Output

For each test case output  $m$ , the minimal number of moves which are necessary to reach the expected formation. The next  $m + 1$  lines should represent one possible scenario of moves.

### Sample

input	output
11	3 Steps
-5 -4 -3 -8 -7 -6 1 -11 -10 -9 -2	-5 -4 -3 -8 -7 -6 +1 -11 -10 -9 -2
1 2 3 4 5 6 7 8 9 10 11	-5 -4 -3 -8 -7 -6 +1 +2 +9 +10 +11
11	-5 -4 -3 -2 -1 +6 +7 +8 +9 +10 +11
1 2 3 -4 -5 -6 -7 -8 -9 10 11	+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11
11 9 8 7 6 5 4 3 2 10 1	5 Steps
0	+1 +2 +3 -4 -5 -6 -7 -8 -9 +10 +11
	+1 +2 -3 -4 -5 -6 -7 -8 -9 +10 +11
	+1 -2 -3 -4 -5 -6 -7 -8 -9 +10 +11
	+1 -2 -3 -4 -5 -6 -7 -8 -9 -11 -10
	+11 +9 +8 +7 +6 +5 +4 +3 +2 -1 -10
	+11 +9 +8 +7 +6 +5 +4 +3 +2 +10 +1

---

### Solution:

This problem is based on the “Sorting signed permutations by reversals” problem, which is well-studied and has application in some fields of biology.

In the problem we are provided with a signed permutation. A signed permutation is a permutation in which each number has a sign, either + or – . We also have a reversal operation. A reversal operation takes a sequence of consecutive elements, reverses their order, and changes the signs of the numbers in the sequence. The problem is to reach some specific permutation (with all signs +) starting from our initial signed permutation and using the minimum amount of reversals.

To solve the problem we must simplify it. Suppose we are given signed permutation  $A$  and we want to transform it to permutation  $B$ . Denote the  $i$ -th element in  $A$  by  $A[i]$ . We now build a new signed permutation  $A'$ . For each element  $A[k]$  we find the position its absolute value occurs in  $B$ . Suppose  $|A[k]| = B[p]$ , then  $A'[k] = p$  if  $A[k] > 0$ , and  $A'[k] = -p$  otherwise . It is easy to see that sorting  $A'$  is equivalent to transforming  $A$  to  $B$ , i.e. it is done using the exact same reversals.

For example take  $A = \{-3, -2, +1, +4\}$  and  $B = \{2,3,1,4\}$ , we get  $A' = \{-2, -1,3,4\}$ . Then doing a reversal in the interval  $[1; 2]$  will change  $A$  to  $\{+2, +3, +1, +4\}$  which is precisely  $B$ , and it will change  $A'$  to  $\{+1, +2, +3, +4\}$  which is a sorted signed permutation.

After this transformation we need an algorithm to sort a signed permutation with minimum amount of reversals and more importantly, give us a list of those reversals. This problem, while not easy, is well studied and there are a few known algorithms that run in polynomial time. The algorithm required for this problem is described in paper by Kaplan, Shamir and Tarjan – “Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals” ([paper](#)).

The algorithm runs in  $O(N^2)$  time and generates all reversals required to sort a signed permutation which is fast enough to solve the problem. Once we have a list of those reversals we can naively apply them one by one to the initial permutation and print the result after each reversal.

**Note:** There is a public [applet](#) that can help you visualize the process of the algorithm on custom examples.

---

*Added by: Simon Gog*

*Solution by:*

*Name: Encho Mishinev*

*School: МГ "Атанас Радев"*

*E-mail: encho.mishinev@gmail.com*

---

---

## Problem R2 06: Yet Another Assignment Problem (ID: 6819)

---

New term is coming. Our monitor Cathy Yin is going to make necessary preparations. Now she has  $m$  jobs to do, and  $n$  classmates are going to help her. Each job requires some classmates working on it for certain time, say the  $i$ -th classmate must work on the  $j$ -th job for  $A[i][j]$  minutes. As an Oler of great responsibility she wishes to finish all jobs as soon as possible. But a classmate can do only one job at a time, and two classmates can not do the same job at the same moment. For example, to decorate the classroom, Alpha must work on it for 3 minutes plus Beta works on it for 4 minutes, then one possible assignment will be *ABABBAB*, taking 7 minutes in total.

Now she is going to make a detailed schedule specifying who is doing what at each moment. Jobs are independent, i.e. they may be done in arbitrary order. Also for each job anyone can do it for arbitrarily long, but not longer than the required time  $A[i][j]$ . Anyone can be free at any time. Time for certain classmate doing certain work need not be consecutive.

As her friend, you are to help her to work out the schedule minimizing the time needed. (The time of this assignment itself does not count!)

### Input

First line of the input contains two positive integers  $m, n$  ( $1 \leq m, n \leq 2000$ ), number of jobs and classmates.

$m$  lines follow, each describing a job.  $i$ -th line contains  $n$  non-negative integers ( $\leq 106$ ), where the  $j$ -th number is  $A[i][j]$ , meaning that the  $j$ -th classmate has to work on the  $i$ -th job for  $A[i][j]$  minutes as described above.

### Output

First line contains single integer  $T$ , minimum time needed. Next line contains  $n$  non-negative integers ( $\leq m$ ), giving one possible schedule for the first minute, where the  $i$ -th number specifying the job for the  $i$ -th classmate to do, and 0 denotes that the corresponding classmate is free.

If there are multiple solutions, any one is accepted.

### Sample

input	output
2 2	7
2 5	1 0
5 1	

### Explanation

Two jobs are assigned to two classmates, say Lambda and Mu. To tidy up the classroom Lambda needs to work for 2 minutes and Mu 5 minutes; and to move desks for new comers Lambda needs 5 minutes and Mu 1 minute.

One optimal schedule is:

T	Lambda	Mu
0	Tidy	Free
1	Move	Tidy
2	T	M
3	M	T
4	M	T
5	M	T
6	M	T

7 minutes in total. It is obvious that it is impossible to finish it in less than 7 minutes.

**Solution:**

Even though, in the title of this problem there is a word assignment, this task is in fact a variation of Open-shop scheduling problem. Variation, because it allows interruption of any job at any moment with the intention of resuming said job at the later time. In Computer Science this is also called preemption.

General Open-shop scheduling problem is NP-hard, however if you allow preemption, the minimum time required to complete all the jobs is simply the highest row or column sum. More precisely:

$$\text{Max}(\sum_{i=0, j=0}^{m-1, n-1} A_{i,j} , \sum_{i=0, j=0}^{n-1, m-1} A_{j,i})$$

Let's denote the said maximum sum with  $\alpha$ .

The fact that  $\alpha$  is indeed the optimal answer can be proven if you consider the two cases:

1. A row has the maximum sum, then that row has to be processed at least  $\alpha$  units of time, since only one row can be assigned to one column at a time, achieving less than  $\alpha$  time is impossible. Now all there's left to do is to show that it is possible to achieve the time of exactly  $\alpha$  units. This will be done through the reconstruction phase of our algorithm.
2. A column has the maximum sum. Since only one student can work on only one job at a time, and only one job can be worked on by only one student at a time, students and jobs are interchangeable. So just "swap" students and jobs and refer to Case 1).

To reconstruct the answer, first create a matrix of size  $(n + m) * (n + m)$ . Now, fill in the columns from  $n$  to  $n + m - 1$  such that the sum of each row is  $\alpha$ , and fill in rows from  $m$  to  $n + m - 1$  such that the sum of each column is  $\alpha$ . Also fill in the lower right side of the matrix such that the sum of each row and column in the entire matrix is exactly  $\alpha$ .

2	2	4
1	2	0

*initial matrix of jobs  
 $\alpha = 8$*

2	2	4	<u>0</u>	0
1	2	0	0	<u>5</u>
<u>5</u>	0	0	0	<u>3</u>
0	<u>4</u>	0	<u>4</u>	0
0	0	<u>4</u>	<u>4</u>	0

*After expanding the matrix, added numbers are underlined*

Now, create a bipartite graph with  $2 * N$  vertices, where every vertex represents either a row or a column. Create an edge between row  $i$  and column  $j$  if  $A_{i,j}$  is nonzero. In said graph find a maximum matching. If row  $i$  and column  $j$  are matched, that means that in the first second, student  $j$  works on job  $i$ . If  $j \geq n$  then no student is assigned to job  $i$ .

For each row  $i$  and column  $j$  that are matched decrease  $A_{i,j}$  by one and repeat the entire process until you are left with matrix of all zeroes.

Note here that, at each step, a perfect matching is performed. All rows get matched to some column, and since the matrix is square, every column gets matched to some row. The sum of the entire matrix is  $N \cdot \alpha$  and at each step you reduce that sum by  $N$ , so total number of steps taken is  $\alpha$ .



Hopcroft-Karp algorithm which runs  $O(|V| \cdot \sqrt{|E|})$  time, can be used to reconstruct the first second of the solution. Number of edges  $|E|$  is at most  $(n + m)^2$  and the number of vertices  $|V|$  is  $2 * (n + m)$ , so the overall complexity is  $O((n + m)^2)$ .

---

*Added by: Tony Beta Lambda*

*Solution by:*

*Name: Dragan Marković*

*School: School of Computing*

*E-mail: dragan224@gmail.com*

---

---

### Problem R2 07: Illumination (ID: 2661)

---

Time Limit: 2.0 second

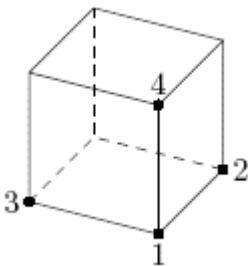
Memory Limit: 256 MB

Two cubes and a light bulb are placed in a three-dimensional Euclidean space. You are expected to find out if one of them casts shadow on the other one and if so, calculate the area of this shadow.

#### Input

Multiple test cases. For each test case:

The first line of the input contains the coordinates of the bulb. It is followed by two groups of four lines each that describe the cubes. Each line of the cube description contains the coordinates of a vertex (see the figure where the vertices are marked and labeled in the same order as they are given in the input).



All the coordinates are given with 5 digits after decimal point. It is guaranteed that the cubes do not intersect, the light bulb is outside both of them, and doesn't lie on any of the planes that contain their faces. A light bulb should be regarded as a point light source.

Input terminates by EOF.

#### Output

For each test case:

The output should contain a single line with two numbers separated with a space character. The first one is the number of the cube that has a shadow on it (1 or 2). The second is the area of the shadow. If none of the given cubes casts shadow on the other the output should contain a single number  $-1$ .

Note: if your output has an error with absolute value less than  $10^{-2}$ , it will be judged as Accepted. i.e. You may output any number of digits after decimal point.

#### Sample

input	output
-1.00000 1.00000 1.00000	2 4.000
0.00000 0.00000 0.00000	-1
2.00000 0.00000 0.00000	
0.00000 2.00000 0.00000	
0.00000 0.00000 2.00000	
5.00000 0.00000 0.00000	
7.00000 0.00000 0.00000	
5.00000 2.00000 0.00000	
5.00000 0.00000 2.00000	
0.00000 0.00000 0.00000	
1.00000 1.00000 1.00000	
2.00000 1.00000 1.00000	
1.00000 2.00000 1.00000	
1.00000 1.00000 2.00000	
-1.00000 -1.00000 -1.00000	

-1.00000 -2.00000 -1.00000	
-2.00000 -1.00000 -1.00000	
-1.00000 -1.00000 -2.00000	

**Solution:**

In this task we're given two non-intersecting cubes and a single point light source outside of them. We have to find out whether this source casts a shadow on any of the solids. If it does, we need also to output which cube has a shadow on it and how large this shadow is.

On the first sight (and probably a few more, too), we may have completely no idea what to do; there are countless combinations of positioning and rotation of the solids and the source. It most likely means that we have to take steps to simplify the problem a bit. Let's call the cubes  $A$  and  $B$  and the source  $S$ .

First of all, we can check the area of shadow  $B$  casts on  $A$  (if there is any). Then we can swap the cubes and use exactly the same routine to compute the area of shadow cast on  $B$ . Now we focus on checking area cast on  $A$ .

We know that  $A$  consists of 6 square faces. We can process them one by one. For each face we need to check if there would be any light on it without  $B$  (that is, if this face isn't hidden from the light source) and then how much shadow there is on it.

**Processing a single face**

However, we still find it extremely difficult to compute the result even for a single face. Fortunately for us, we can do a trick. Let's translate and rotate our 3D space so that:

- if length of  $A$ 's edge is equal to  $a$ , then the coordinates of face are  $(0,0,0)$ ,  $(a, 0,0)$ ,  $(a, a, 0)$ ,  $(0, a, 0)$
- the interior of cube fulfills  $z > 0$ .

This transform is convenient for us for a few reasons. Firstly, we don't shrink or enlarge the space, so the result remains unchanged. Secondly, we get a very easy test if the face is in front of the light source or behind it (we just need to check if  $z' < 0$  for the light source). Moreover, the face now lies on  $z = 0$  plane and all the computations become easier.

The question is, how to find such transform? We know that there is a matrix  $M$  which converts previous coordinates  $(x, y, z)$  into new ones  $(x', y', z')$ :

$$(x, y, z, 1)^T M = (x', y', z', 1)^T$$

(and  $\det M = 1$  because we don't change the size of the space, but that isn't important). How can we compute the matrix? It's easy — we can take four vertices of the cube which will become  $(0,0,0)$ ,  $(a, 0,0)$ ,  $(0, a, 0)$  and  $(0,0, a)$  after the transform (call them  $(x_1, y_1, z_1), \dots, (x_4, y_4, z_4)$ ). Then we get

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} M = \begin{pmatrix} 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & a \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

## $z = 0$ problem

We're still left with problem how to compute shadow. We assume that light is cast on the selected face after removing  $B$  (that is,  $z'_s < 0$ ). We may find it easier to compute the shadow cast by  $B$  on the plane  $z = 0$  instead of the face. After some experiments, we find out that the following are true:

**Statement 1.** *Shadow (if it exists) is a convex subset of  $R^2$  (maybe unbounded) which is an intersection of half-planes.*

**Statement 2.** *Let's find the shadow cast by all the edges of  $B$  on the plane  $z = 0$ . Then the shadow is the smallest convex figure containing all such cast edges.*

Note that the edges don't have to form the convex figure. It happens for example if  $z = 0$  cuts off only one corner with three parts of edges coming out of it (figure 1).

It means we need to find the shadow cast by  $B$ 's edges on plane  $z = 0$ , compute the convex hull  $C$  of such set of edges (which can be converted to set of points) and compute the common area of  $C$  and the processed face of  $A$ . Now note that:

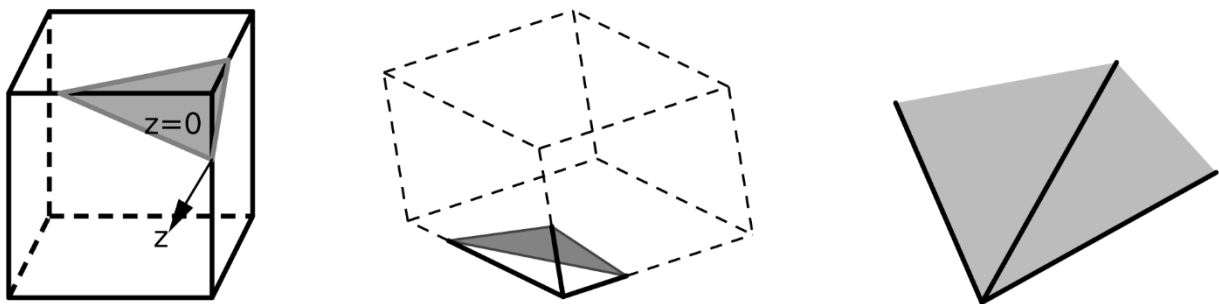


Figure 1: An interesting example. Left:  $B$  cube and section made by  $z = 0$  plane. Vector shows  $z > 0$ . Middle: the same cube and (bold) visible parts of edges from  $z = 0$  perspective. Right: the "light projections" of the visible edges and the region that is shaded by  $B$ . Note that not all edges of convex hull are the projections.

- The second can be easily done by any convex hull algorithm (e.g. Graham's scan) because all the points lie on  $z = 0$  plane.
- Intersection of two convex polygons on the plane can also be done quite easily using Sutherland-Hodgman algorithm; that is, convert one polygon into a set of half-planes and for each half-plane cut off the part of the second polygon that is outside the half-plane.
- Computing the area of resulting polygon can be done using a very known formula

$$A = \frac{1}{2} \left| \sum_{i=1}^n (x_i y_{i+1} - x_{i+1} y_i) \right|$$

We're only left with one problem.

## Casting edges on $z = 0$ plane

Here comes a thing we've been avoiding for such a long time — a case study. Fortunately, it won't hurt that much. Let's call  $z'_s$  a converted  $z$ -coordinate of light source and  $z'_1, z'_2$  the converted  $z$ -coordinates of the edge endpoints (and call these endpoints  $X, Y$ ).

1. If  $z'_1, z'_2 \geq 0$ , no shadow will be cast on  $z = 0$  for sure ( $z = 0$  completely covers this edge).
2. If  $z'_1, z'_2 \leq z'_s$ , also no shadow will be cast on  $z = 0$  as the light beam goes straight and after

- reaching any point of the edge it cannot hit  $z = 0$  anymore.
3. If  $z'_1, z'_2 \in (z'_s, 0)$ , then the shadow will be cast. It's very easy to compute it: cast both endpoints on  $z = 0$  plane (cut  $SX \rightarrow$  and  $SY \rightarrow$  rays with  $z = 0$ ); they both are obviously the endpoints of the cast edge.
  4. If nothing above happens, we can partition the edge into at most three parts: the first one fulfills  $z \in (-\infty, z'_s + \varepsilon)$ , the second one  $z \in (z'_s + \varepsilon, -\varepsilon)$ , and the third one  $z \in (-\varepsilon, +\infty)$ . Only the second one can create a shadow so we need to consider only this one (and this way we reduced this problem to case 3).

The  $\varepsilon$  is very helpful for us as it allows to approximate the unbounded figure as a polygon in which "the vertices at infinity" are just usual points, though with large values. If we considered an edge for which  $z \in (z'_s, 0)$  the beam going through an endpoint at  $z = z'_s$  would cut  $z = 0$  plane at infinity — that is what we want to avoid.

### Summary

It is quite a long solution, so we need to sum it up:

1. Read cubes  $A, B$  and light source  $S$ .
2. Select a cube (say  $A$ ) on which we'll try to cast the shadow.
3. Take each of 6 faces of  $A$  and then try to compute the result.
4. Transform the space so that the selected face lies on  $z = 0$ .
5. If the light source fulfills  $z'_s \geq 0$ , return no shadow on this face.
6. Cast each of 12  $B$  edges onto  $z = 0$  plane.
7. Compute the 2D convex hull of the endpoints of these edges (if there are any).
8. Find an intersection of this convex hull and the selected face.
9. Compute the area of intersection and add it to the result.

---

*Added by: [Trichromatic] XilinX*

*Resource: ACM/ICPC NEERC Moscow Subregional Contest 2007*

*Solution by:*

*Name: Marek Sokółowski*

*School: I LO w Lomży*

*E-mail: mnbvmar@gmail.com*

---

---

## Problem R2 09: [CH] Guess The Number With Lies v2 (ID: 17308)

---

Time Limit: 20.0 second

Memory Limit: 1536 MB

Judge has chosen an integer  $x$  in the range  $[1, n]$ . Your task is to find  $x$ , using query as described below. But be careful, because the Judge is a liar. Judge is allowed to lie up to  $w$  times in single game and only in reply for query.

Single query should contains set  $S = \{a_1, a_2, \dots, a_k\}$ . The reply for query is "YES", if  $x$  is in  $S$ . Otherwise the reply is "NO".

$$1 \leq k < n$$

$$1 \leq a_1 < a_2 < \dots < a_k \leq n$$

### Communication

You should communicate with Judge using standard input and output.

**Attention: the program should clear the output buffer after printing each line. It can be done using `fflush(stdout)` command or by setting the proper type of buffering at the beginning of the execution - `setlinebuf(stdout)`.**

The first line of input contains single integer  $T$ , the number of games. Then  $T$  games follow.

At the beginning of each game you should send to the Judge a line with command "START\_GAME". The Judge will answer you with numbers  $n, w, m$ , where  $n, w$  are as described above and  $m$  is the maximum number of queries that you can use in this game.

Then you should send some queries, every query is a line with "QUERY" keyword, then single-space separated values  $k a_1 a_2 \dots a_k$ . After each query the Judge will answer "YES" or "NO".

At the end of game you should give answer: "ANSWER  $y$ ", where  $0 \leq y \leq n$ ;  $y = 0$  means, that you skip this game without the correct answer. Otherwise  $y$  is the answer for the game. When  $y \neq x$ , the solution will got WA.

Then start the next game (if there is any).

### Sample

Communication
The example of communication. J=Judge, P=Player.
J: 3
P: START_GAME
J: 2 2 10
P: QUERY 1 1
J: YES
P: QUERY 1 1
J: YES
P: QUERY 1 1
J: YES
P: ANSWER 1

```

P: START_GAME
J: 2 4 10

P: QUERY 1 2
J: YES
P: QUERY 1 2
J: YES
P: QUERY 1 1
J: YES
P: QUERY 1 1
J: YES
P: QUERY 1 2
J: YES
P: QUERY 1 2
J: YES
P: QUERY 1 2
J: NO
P: QUERY 1 1
J: NO
P: ANSWER 2

P: START_GAME
J: 12345 7 100

P: ANSWER 0

```

### Explanation

In 1st game Judge said 3 times, that his number is 1 and he didn't lie. The answer is 1, because he can lie only 2 times.

In 2nd game the Judge lied in 3rd, 4th and 7th query.

In 3rd game the Player gave up. The score is  $4 * 1002$ .

The score is  $32 + 82 + 4 * 1002 = 9 + 64 + 40000 = 40073$

### Note

Be careful. The Judge will try to maximize the number of queries that you will ask. If necessary, the Judge can also replace chosen value  $x$  with the other one. But don't worry too much - at the end of the game, the value  $x$  chosen by Judge will satisfy all except at most  $w$  of your queries.

### Note2

If you got SIGXFSZ error, you probably use unnecessary numbers in queries. Let's see at the example:

```

P: START_GAME
J: 16 2 14

P: QUERY 2 1 2
J: NO
P: QUERY 3 1 2 3
J: NO
P: QUERY 4 1 2 3 4
J: NO

```

P: QUERY 5 1 2 3 4 5  
J: NO  
P: QUERY 6 1 2 3 4 5 6  
J: NO

In 4th query, there are unnecessary numbers 1 and 2. This numbers cannot be the answer for this game, because the Judge said three times (in 1st, 2nd and 3rd query's reply) "1 and 2 are not OK!", but the Judge can lie only 2 times. From the same reason in 5th query, the unnecessary numbers are 1, 2 and 3. When  $n$  is big enough, the profit from this optimization is huge (and probably *SIGXFSZ* won't appear).

---

**Solution:**

---

The problem **[CH] Guess The Number With Lies v2** on this year's Bubble Cup qualifications is known in information theory as Ulam's game.

It is a game with two players, named Paul and Carole and three parameters  $n, q$  and  $k$ , known to both players. Carole thinks of an integer  $x \in [1, n]$ . Paul has  $q$  questions with which to determine  $x$ . The questions must be of the form "Is  $x \in A$ ?", where  $A \subseteq \{1, \dots, n\}$ . He (Paul) may use previous answers before deciding his next question. Carole is permitted to lie but she (Carole) may lie at most  $k$  times through the entire course of the game. Paul wins if at the end of the  $q$  questions there is a unique possible value for  $x$ . Carole is allowed to play an adversary strategy, i.e., Carole does not actually pick an  $x$  but answers all questions so that there is at least one  $x$  that she could have picked. Now the game is one of perfect information and so we can say for given  $n, q, k$  that either Paul or Carole will win the game. The question is: Who wins? Note that when  $k = 0$  the game reverts to the classical "Twenty Questions" and Paul wins if and only if  $n \leq 2^q$ .

In this particular problem we take the role of Paul and are allowed to ask a much higher than optimal number of questions to determine  $x$ . Of course, asking fewer questions awards a higher score for each test case. Note that during the initial analysis we will consider  $k$  to be a fixed positive integer, and later use the derived conclusions to develop a strategy for playing the game.

Consider a generalization of this game with the single parameter  $n$  replaced by a sequence of nonnegative integers  $x_0, x_1, \dots, x_k$ . Let  $A_i, 0 < i < k$ , be disjoint sets, with  $|A_i| = x_i$ ; these sets known to both players. Now Carole selects  $x \in A_0 \cup \dots \cup A_k$ . If  $x \in A_i$  then Carole is permitted to lie at most  $k - i$  times about it. Again, Carole can play an adversary strategy so that either Paul or Carole will win the game. The  $n, q, k$  games correspond to  $x_0 = n, x_1 = \dots = x_k = 0$ . This representation is useful for analyzing "middle positions" of the  $n, q, k$  game. In this sense  $x_i$  gives a count on those  $x$  for which if  $x$  is the answer then Carole has already lied  $i$  times about it.

Another way of thinking about this problem is in terms of chips. Imagine a board with positions marked (from left to right)  $0, 1, \dots, k$ . There is one chip for each possible answer  $x$ . A chip is placed on position  $i$  if  $x$  is an answer Carole can lie about at most  $k - i$  more times. Thus, the  $x_0, x_1, \dots, x_k$  game starts with  $x_i$  chips on position  $i$ , for each  $i$ . In this context, each round ( $q$  is now the number of rounds) Paul selects a set  $A$  of chips, corresponding to asking the question "Is  $x \in A$ ?". A "No" answer by Carole would mean that, for each  $x \in A$ , if  $x$  is the answer then it has been lied about one more time. This corresponds to moving all chips in  $A$  one position to right. Chips that were in position  $k$  are removed from the board. A "Yes" answer by Carole corresponds to moving all chips not in  $A$  one position to the right. That is, Paul selects a set  $A$  of chips and Carole selects whether to move all chips in  $A$  or all chips not in  $A$  one position to the right. Carole is not permitted to move all the chips off the board (although this would not occur in actual play). Paul wins if at the end of the game there is precisely one chip remaining on the board. We define the state of our board to be the vector  $P = (x_0, x_1, \dots, x_k)$ . This state changes during the game as the chips are moved.

Now, we imagine Carole announcing a random strategy-whatever set  $A$  Paul selects, Carole will then flip a fair coin to decide whether to move the chips of  $A$  or the chips not in  $A$  one position to the right. (If by this strategy all chips are removed we will agree that Carole has lost.) The coin flips are done separately each round. Now a strategy for Paul has a probability of winning. For each chip  $c$  let  $X_c$ , be the indicator random variable for  $c$  to remain on the board at the end of the game. Regardless of Paul's strategy, each chip will



move forward with probability 0.5 each turn - if the coin flip “matches” whether  $c \in A$  - and the movements on the different turns are mutually independent. If  $c$  starts at position  $j$ , its position at the end of the game is given by  $j + B(q, 0.5)$ , or “off the board” if this is larger than  $k$ , where  $B(q, 0.5)$  is the standard Binomial distribution, the number of heads in  $q$  flips of a fair coin. We will consider

$$E[X_c] = \Pr[B(q, 0.5) \leq k - j],$$

the probability of remaining on the board, to be the weight of the chip  $c$ . Let  $X = \sum X_c$  the sum over all chips  $c$ . Linearity of expectation gives  $E[X] = \sum E[X_c]$ , which we consider to be the weight of the whole state of the board, and the expected number of remaining chips on it. This weight being greater than 1 means that the expected number of remaining chips is also greater than 1. In particular, this implies that we cannot have  $X < 1$  always, so that with positive probability Carole will win. However, this is a perfect information game and so with perfect play either Paul or Carole will always win. Since no strategy allows Paul to always win, there is a strategy (not randomized) so that Carole always wins! This conclusion allows us to check whether a certain board state is unwinnable for us (Paul) in a given number of moves and thus allows us to put a lower bound on the required number of moves to win.

$E[X]$  is not overly time consuming to compute (caching binomial coefficients makes this faster), which allows us to quickly determine if a state is possibly winnable in  $q$  moves. Since  $E[X]$  is monotonous (if it might be possible to win in  $q$  moves, it isn't impossible in  $q + 1$ ) we can use binary search to determine the lower bound for the number of moves required to win a certain board –  $s(P)$ . Note that this does not guarantee that a strategy that always wins in  $s(P)$  moves exists, but has proved to be an acceptable heuristic to use when developing a strategy for this problem.

For any board state  $P$ , our strategy for winning in  $s(P)$  moves involves finding the subset of chips  $A$  to play in the next move. This provides us with two possible future states  $P_{YES}$  and  $P_{NO}$ , depending on Carol's answer. The strategy is considered valid if  $s(P_{YES}) \leq s(P) - 1$  and  $s(P_{NO}) \leq s(P) - 1$ . Since chips in each stack have the same individual weight, and the weight associated with the stacks decreases from left to right, a greedy algorithm is a great candidate for finding the subset  $A$ . Basically, we can use a descending sort of the chips by slightly modified weight (only consider the change in the chip's weight for both possible future boards) to add the chips one by one to either  $A$  or  $\tilde{A}$  and keep the weights of  $P_{YES}$  and  $P_{NO}$  balanced. This allows us to (almost) evenly split the chips and make close to optimal moves.

---

*Added by: miodziu*

*Solution by:*

*Name: Uglješa Stojanović*

*School: School of Computing*

*E-mail: ugljesas@gmail.com*

---

---

## Problem R2 10: [CH] Colour Brick Game (ID: 18073)

---

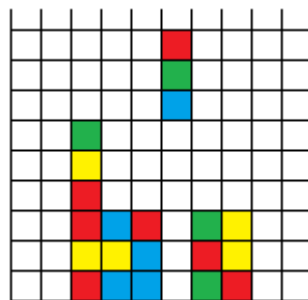
Time Limit: 50.0 second

Memory Limit: 256 MB

Let's consider the tetris like game. The rules are very easy, there is board of width 10 and height 20. Color bricks fall down the board. Every brick contains 3 colors. Player can rotate colors in the brick and determine the column, in which to place current brick. The goal is to construct horizontal, vertical or diagonal lines of length 3 or more in the same color. After every brick falls down, the cleaning up algorithm is executed.

Single step of cleaning up algorithm searches for lines constructed from cells with the same color (there can be more than one such line). If no line is found, the algorithm finishes. Otherwise all cells in all found lines disappear, then the cells above this disappeared cells fall down. The cleaning up algorithm executes single steps as long, as there are any changes on the board.

The brick can be rotated. When the brick contains colors  $A, B, C$  (from top to bottom), then single rotation makes the brick with colors  $C, A, B$ , and double rotation makes the brick with colors  $B, C, A$ .



Your task is to write the program, which can play in Colour Brick Game. You will receive the consecutive bricks and for each of them have to determine the correct column and rotation. For each cleaned line You got points (according to scoring section). The goal is to maximize the score.

### Input

The first line of input contains single-space separated integers  $n$  and  $k$  ( $3 \leq k \leq 9, 1 \leq n * k \leq 10^6$ ), where  $n$  is the number of bricks in the game and  $k$  is the number of different colors, that can appear in the game. The next  $n$  lines contain bricks description, one line per brick. Each brick is represented as string of length 3, containing digits (every digit is from 1 to  $k$ ).

### Output

For every brick from the input, You should write a line with two single-space separated integers  $x$  and  $r$  ( $1 \leq x \leq 10, 0 \leq r \leq 2$ ), where  $x$  is the number of column to place the brick and  $r$  is the rotation of the brick (0 for no rotation, 1 for single rotation, 2 for double rotation).

If after falling down the brick is located outside the board (the higher brick cell is higher than the highest board row), the game finishes immediately. The judge doesn't read the remaining moves, so you don't need to output them. You can also stop, when there is no valid move at the board and you don't need to output this last move.

### Score

Base score for cleaned line of length  $m$  is  $m^2$ . The base score is multiplied by the number of lines cleaned in single cleaning up step and then multiplied by the cleaning up step number (see example for clarify).

The brick score is sum of scores received from cleaned lines after the brick fallen down.

Total score is sum of brick scores.

### Sample

input	output
11 5	1 1
211	2 0
321	3 0
232	3 2
233	4 0
345	5 0
245	6 0
451	6 0
332	7 0
451	7 0
332	8 1
312	

input	output
11 5	1 0
211	1 0
321	1 0
232	1 0
233	1 0
345	1 0
245	1 0
451	
332	
451	
332	
312	

### Explanation

In first example, for the first 10 bricks there is no line to clean. The last brick makes lines to clean in 4 cleaning up steps.

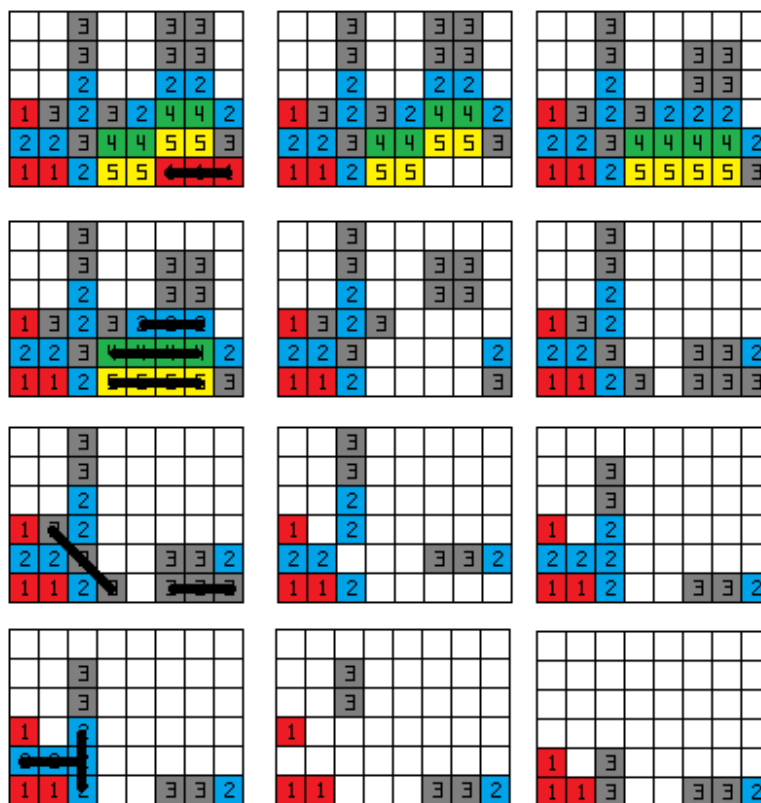
In *1st* step there is only one line of length 3 to clean. The score is  $3^2$ , multiplied by 1 (number of lines cleared in the step) and multiplied by 1 (number of step).  $3^2 * 1 * 1 = 9$ .

In *2nd* step there are 3 lines (one of length 3, two of length 4). The score is  $3^2 + 4^2 + 4^2$ , multiplied by 3 (number of lines cleared), multiplied by 2 (number of step).  $(3^2 + 4^2 + 4^2) * 3 * 2 = 41 * 6 = 246$ .

In *3rd* step there are 2 lines (both of length 3). The score is  $(3^2 + 3^2) * 2 * 3 = 18 * 6 = 108$

In *4th* step there are 2 lines of length 3. The score is  $(3^2 + 3^2) * 2 * 4 = 18 * 8 = 144$ . Total score is  $9 + 246 + 108 + 144 = 507$ .

In second example player placed all bricks in *1st* column. There is no cleaned line, so the *7th* brick after falling down is located outside the board. The game finishes with score 0. Note, that there is only 7 lines in output.




---

**Solution:**

---

This is one of well-designed challenge problems. After some analysis, we conclude several things:

- Big score is clearing as much bricks as possible in as many clearing as possible
- It is harder to fit bricks for big scores when number of colors ( $K$ ) is big
- When  $K$  is small, bricks tend to get cleared very easily and we need to avoid this

These observations bring us to winning strategy. We concluded that time shouldn't be wasted on  $K$  bigger than 4. We concentrate only on cases of 3 and 4 colors, which alone brings about 20 million points.

Roughly, algorithm works as follows:

- 1) We calculate score for all 10 possible positions and 3 rotations
- 2) For maximum of 30 possibilities we calculate number of points it would bring
- 3) Next thing to be done is sorting of scores, in following fashion: primary - Number of points, secondary and even tertiary we sort using other parameters like number of same adjacent bricks, number of holes, etc.
- 4) Complete state of playing field and scores are remembered after every new insertion of new brick
- 5) We choose the worst score (Which is very likely 0) and repeat it until some condition is satisfied (For example, for  $K = 3$  it turned out it is in about 60 insertions or full table)
- 6) In that moment, we search for the best score out of the remembered scores, we go back to the corresponding state and clear the blocks for the best score.
- 7) After clearing, we remove the remembered states and scores and start the algorithm all over again from this new state.

The falling bricks win points in following fashion: About 50 bricks makes 0 or just a little above 0 points, then just one brick produces  $12K$ . Then about 60 bricks make about 0, then the next makes about  $15K$ . Then about 40 bricks make about 0, then just one more produces about  $18K$ . The average value of points per clear

is  $15K$ , when moments are well chosen. The largest score in random generated input is worth about  $50K$  points. That means that number of points per brick is average 400 points for  $K = 3$ , and 250 points for  $K = 4$ . It is easily seen that it is impossible to do so if every clear is done as soon as possible. For the case of  $K = 5, 6, 7$  we managed to play the game until the very end, but number of points is drastically smaller compared to strategy we described, which cannot be implemented in these cases. For the case of  $K = 8$  or  $9$  we couldn't finish the game, but to be honest, we didn't have to, because the idea above indicated that big  $K$  won't bring us much points.

Per test cases:

Number of colors	Correct moves	Score
3	3333	1333534
4	2498	646033
5	2000	204065
3	33333	11997210
4	25000	6168573
5	20000	1685123
6	16664	521511
7	14285	215244
8	1007	14740
9	370	4062

---

*Added by: miodziu*

*Solution by:*

*Name: Aleksandar Ogrizović*

*School: Gimnazija Sombor*

*E-mail: xxxgrizxxx@gmail.com*

---

*The scientific committee would like to thank everyone  
who did important behind-the-scenes work.  
We couldn't have done it without you!*

*We will prepare a lot of surprises  
for you again next year!  
Stay with us!*

***Bubble Cup Crew***





bubble  
cup



student coding competition

# ProblemSet booklet

