Bubble Cup 2019

Microsoft Development Center Serbia

Problem set & Analysis from the Finals and Qualification rounds

Belgrade, 2019

Scientific committee:
Aleksa Milisavljević
Aleksandar Damjanović
Bojan Vučković
Janko Šušteršič
Kosta Bizetić
Kosta Grujčić
Miloš Šuković
Nebojša Savić
Nikola Smiljković
Nikola Nedeljković
Predrag Ilkić
Slavko Ivanović

Qualification analysts:
Aleksa Miljković
Ivan Stošić
Katarzyna Kowalska
Konrad Majewski
Krzysztof Maziarz
Magdalina Jelić
Marko Šišović
Michal Zawalski
Mihailo Milošević
Miloš Purić
Mladen Puzić
Nikola Jovanović
Nikola Pešić
Pavle Janevski
Pavle Martinović
Petar Vasiljević
Tadija Šebez
Uroš Berić
Uroš Maleš
Veljko Radić

Cover:
Sava Čajetinac

Typesetting:
Marijana Prpa
Anja Pantović

Volume editor:
Dragan Tomić

Contents

Preface

Dear Bubble Cup finalists,

I would like to thank you for taking part in the twelfth edition of Bubble Cup in Belgrade, Serbia.

This year there are two divisions: The Premier League – the division opened for high school and university teams which meet the eligibility requirements of Bubble Cup; and Rising Stars – the division opened only for eligible Serbian high school teams. By creating the Rising Stars division, we want to encourage Serbian high schools to compete in Bubble Cup and motivate young programming talents in Serbia as well.

Bubble Cup 12 has gathered more than 80 competitors in 28 teams (The Premier League consists of 16 teams and Rising Stars consists of 12 teams). The Finalists come from Belarus, Bulgaria, Poland, Ukraine and Serbia. The Rising Stars finalists, composed of high school teams from Serbia, come from Belgrade, Niš, *Novi Sad and Sombor.*

For twelve years, Microsoft Development Center has been hosting Bubble Cup, a competition designed for high school and university students, as a form of a complete preparation for some of the most significant and prestigious coding tournaments – such as the ACM competition. In addition to that, the importance of the participation in the Bubble Cup competition in preparing young talents for professional development is also reflected in the fact that many of more than 300 employees of Microsoft Development Center were once Bubble Cup finalists.

We are glad to have you all as a part of our Bubble Cup team for a better future of discovering new coding models and creating new friendships.

Sincerely,
Dragan Tomic
MDCS PARTNER Engineer manager/Director

About Bubble Cup

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 overgrown and the vision was expanded to attract talented programmers from the entire region and promote the values of communication, companionship and teamwork.

This edition of Bubble Cup is special because the format of the competition has changed this year. Now we have two divisions:

1. **Premier League** – Division opened to high school and university teams that satisfy eligibility rules
2. **Rising Stars** – Division opened only to eligible Serbian high schools' teams.

The best 16 teams from the Premier League division and the best 8 teams from the Rising Star division will have chance to compete in the Finals. Additionally, 4 best teams from specialized IT departments were invited to compete in Rising Stars division for total number of 12 teams in Rising Stars division.

This year, all Bubble Cup finalists had a chance to visit Microsoft Development Center Serbia and an opportunity to hear about the Center, PSI:ML machine learning seminar, MDCS initiative, to try demos and to talk with engineers who shared their experience.

Microsoft Development Center Serbia (MDCS) was created with a mission to take an active part in the conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of the Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 13 years a vast number of products came out as a result of a 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 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, Japan and China), and Microsoft researchers from Redmond, Cambridge and Asia.

Microsoft | Development Center Serbia

Bubble Cup Finals

The Bubble Cup 12 Finals were held on September 14, 2019, at the VIG Plaza in Belgrade (New Belgrade).

Dražen Šumić officially opened the competition by giving an inspiring speech dedicated to all finalists, encouraging them to give their best and to make this competition even more challenging and exciting for all participants!

The competition started at 10.00am and lasted until 3.00pm. In the evening, at Microsoft Development Center Serbia, the award ceremony was held and was later followed by a lounge party organized in honor of all participants.

During the finals the teams in Premier League and Rising Stars divisions solved slightly different problem sets which had overlapping problems. The problem distribution between leagues is represented by the table below:

	A	B	C	D	E	F	G	H	I	J	L	K
Premier League												
Rising Stars												

The rules of the contest were the same for both divisions, which remained in the classical ACM ICPC five-hour format. Prizes were given to the top 3 Premier League teams, as well as the top 3 Rising Stars teams. Also, this year special awards were given:

- **Bubble Racer** for the team who got the fastest solution;
- **Bubble Stamina** for the team most persistant in trying to resolve the problem-s
- **Bubble Finals++** award for the individual with the largest number of the finals he/she attended.

Bubble Cup Finals Results – Premier League

This year we had less problems than usual, however they were tough and less ordinary compared to the previous Bubble Cup finals. It was one of the closest competitions in Bubble Cup history. Heading to final 1-hour freeze time 3 teams:

1. **Warsaw Akabats** (Antoni Żewierżejew, Konrad Czapliński, Marek Sokolowski)
2. **Los Estribos** (Jan Tabaszewski, Maciej Hołubowicz, Mateusz Radecki)
3. **Uncle Bogdan** (Alexander Kernozhitsky, Fedar Karabeinikau, Yahor Dubovik)

had the same score with 7 solved problems with only penalty separating them apart. Until the very last second it was open who will win this year's finals. In the end **Los Estribos** came on top as the only team to solve all the problems in a thriller finish.

Scoreboard

Team name	Score	Penalty
Los Estribos	9	1405
Warsaw Akabats	8	908
Uncle Bogdan	8	1196
Ultimate Fried Cookies	6	955
I want in house dva	5	605
Budućnost PMF-a	5	640
MaxFun	4	355
KhNURE_NRG	4	565
Danonki	4	576
Jagiellonian Hedgehogs	4	591
Yang and Fibs	3	233
fufel	3	339
Cogito ergo sudo	3	365
KNU_Is_Not_Unix	3	443
Suckcessful team	3	465
AIM for gold	2	161

Bubble Cup Finals Results – Rising Stars

The competition for the first three places and prizes in the Rising Stars division was just as exciting. All 3 places where open for grabs to a lot of different teams, and one correctly solved problem could make a big difference. When the competition ended the **Infinity (Nikola Pešić, Miroslav Grubić, Tadija Šebez)** team came out as a winner with **Gii Klub (Pavle Martinović, Lazar Kostić, Mladen Puzić)** and **Inspekcija (Igor Pavlović, Marko Grujčić, Uroš Maleš)** coming 2nd and 3rd respectively.

Scoreboard

Team name	Score	Penalty
Infinity	5	462
Gii Klub	5	674
Inspekcija	4	376
Шми клуб	4	575
SouthSerbia DžB	4	603
Gassivity	4	631
_dujm	3	299
Gimnazija Sombor	3	573
To je dinamicko...	2	89
TryCatch and Exceptions	2	122
Beskonacna_petlja	2	153
ЂЕВРЕК	1	160

Finals problems

Problem A: Harvester

Rising Stars only

Author:

Nikola Smiljković

Implementation and analysis:

Nikola Smiljković

Janko Šušteršič

Statement:

It is Bubble Cup finals season and farmer Johnny Bubbles must harvest his bubbles. The bubbles are in a rectangular bubblefield formed of $N \times M$ square parcels divided into N rows and M columns. The parcel in i^{th} row and j^{th} column yields A_{ij} bubbles.

Johnny Bubbles has available a very special self-driving bubble harvester that, once manually positioned at the beginning of a row or column, automatically harvests all the bubbles in that row or column. Once the harvester reaches the end of the row or column it stops and it must be repositioned. The harvester can pass through any parcel any number of times, but it can collect bubbles from the parcel only once.

Johnny is a very busy farmer, so he is available to manually position the harvester at most four times per day. Johnny is also impatient, so he wants to harvest as many bubbles as possible on the first day.

Please help Johnny calculate what is the maximum number of bubbles he can collect on the first day.

Input:

The first line contains two integers N and M — the bubblefield size.

Each of the next N lines contains M integers. The j^{th} element in the i^{th} line is A_{ij} — the yield of the parcel located in the i^{th} row and the j^{th} column.

Output:

The output contains one integer number – the maximum number of the bubbles Johnny can harvest on the first day.

Constraints:

- $1 \leq N, M \leq N \cdot M \leq 10^5$
- $0 \leq A_{ij} \leq 10^9$

Example input 1:

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

Example output 1:

```
10
```

Explanation 1:

Farmer Johnny can harvest all the bubbles by positioning the harvester on the first and the second row.

Example input 2:

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

Example output 2:

```
80
```

Explanation 2:

One way Johnny can harvest the maximum number of bubbles is to position the harvester in the second row, the fourth row, the second column and the fourth column.

```
Time and memory limit: 0.5s / 256 MB
```

Solution and analysis:

Let rs_i be the sum of yields in the i^{th} row, and cs_i be the sum of yields in the j^{th} column.

First, we can notice that if there are less than five columns or rows in the field, we can just sum up all the elements of the matrix A, e.g. sum up all the elements in rs or sum up all the elements in cs. Otherwise, since all of the numbers in A are non-negative, we'll always want to select exactly four rows or columns of A. There are five options to consider when selecting a total of four rows or columns: four columns, one row and three columns, two rows and two columns, three rows and one column, or four rows.

It is obvious that to achieve the maximum sum by selecting four columns, we must sum up four largest elements of the cs array. Similarly, to achieve the maximum sum by selecting four rows we must sum up four largest elements of the rs array. The time complexity for both operations is $O(N + M)$.

Let's assume we want to select one row and three columns, and that we have selected the i^{th} row. To maximize the sum, we have to select three columns with the largest value $cs_j - A_{ij}$. Then for each row we can find three columns that add up to the largest sum and pick the maximum. Similarly, we can find the maximum sum by selecting three rows and one column by applying this process on the transpose matrix of A. The time complexity for both operations is $O(N + M)$.

When selecting two rows and two columns we can assume $N \leq M$. If that is not the case, we can consider a transpose of the matrix A. For each pair of rows (i_1, i_2) we find two columns with the largest value $cs_j - A_{i_1 j} - A_{i_2 j}$, and pick the maximum sum of selected rows and columns. The time complexity for this operation is $O(N \cdot M \cdot min(N, M))$.

We can consider all five cases and pick the case with the maximum solution, giving the overall time complexity of $O(N \cdot M \cdot min(N, M))$.

Problem B: Periodic integer number

Rising Stars only

Author:

Aleksandar Damjanović

Implementation and analysis:

Aleksandar Damjanović

Nebojša Savić

Statement:

Alice became interested in periods of integer numbers. We say that a positive X integer number is periodic with length L if there exists a positive integer number P with L digits such that X can be written as $PPPP \ldots P$. For example:

$X = 123123123$ is a periodic number with length $L = 3$ and $L = 9$
$X = 42424242$ is a periodic number with length $L = 2, L = 4$ and $L = 8$
$X = 12345$ is a periodic number with length $L = 5$

For the given positive period length L and positive integer number A, Alice wants to find the smallest integer number X strictly greater than A that is periodic with length L.

Input:

The first line contains one positive integer number L representing the length of the period. The second line contains one positive integer number A.

Output:

One positive integer number representing the smallest positive number that is periodic with length L and is greater than A.

Constraints:

- $1 \leq L \leq 10^5$
- $1 \leq A \leq 10^{100\,000}$

Example input 1:

```
3

123456
```

Example output 1:

```
124124
```

Example input 2:

```
3

12345
```

Example output 2:

```
100100
```

Explanation

In the first example 124124 is the smallest number greater than 123456 that can be written with period $L = 3$ ($P = 124$).

In the second example 100100 is the smallest number greater than 12345 with period $L = 3$ ($P = 100$)

Time and memory limit: 0.5s / 256 MB

Solution and analysis:

Let N be the number of digits of the given number A. Solution to this problem can be divided into 6 different cases that need to be covered:

1) $\boldsymbol{L > N}$
   In this case where L is greater than N we don't have enough digits in A for given period so we have to append digits to satisfy minimal length L. The solution is to generate smallest number with L digits. (E.g. for $L = 3$ solution is 100)
2) $\boldsymbol{N \% L \neq 0}$
   For this situation where we have that N is not divisible by L, the number A cannot be partitioned into smaller numbers of length L (We cannot represent A as $A = XYZ$ … where all the parts have same the length of L). Solution for this case is similar to previous one, we first have to append digits to A so that N becomes divisible by L. Now that we can partition A into $\frac{N}{L}$ equal parts of length L we just take smallest numbers with length L for each part. (E.g. for $N = 4$ and $L = 3$ we first append 2 digits so that we get $N = 6$ which is divisible by L and then we take smallest number of length L which is 100 and append it $\frac{N}{L}$ times so the solution is 100100)
3) $\boldsymbol{N \% L = 0}$
   If N is divisible by L number A can be represented as $A = XYZ$ … where X, Y and Z are parts of number A with length L. So we can partition number A into $\frac{N}{L}$ parts of length L. Lets denote first part as X. We have 3 subcases to take into consideration:
   1) **All digits of A are 9s :**
      In this case we don't have larger number with equal number of digits so we must append one more part of length L to number. We do this to ensure that number of digits stays divisible by L . To make it smallest larger than A for each part we take smallest number with length L. (E.g. for $A = 999999$ and $L = 3$ solution is 100100100)
   2) **All parts are equal or first different part from X is larger than X:**
      We iterate through all parts starting from second (from left to right) and compare it with X (first part). If all parts are equal with X or first different part is larger than X solution is to increase X by 1. By doing this we reset all parts to the right from X (parts which digits have smaller weight) to 0 so we can adjust them to $X + 1$ also. Final solution is $X + 1$ repeated $\frac{N}{L}$ times.
   3) **First different part from X is smaller than X:**
      Lets denote that first smaller part with Y. Now if we increase Y to be equal with X we reset all parts that are right from Y to 0. By doing this we can just adjust all parts to be equal to X and that's final solution to this case.

Problem C: Workout plan

Premier League and Rising Stars

Author:

Aleksandar Damjanović

Implementation and analysis:

Aleksandar Damjanović

Predrag Ilkić

Statement:

Alan decided to get in shape for the summer, so he created a precise workout plan to follow. His plan is to go to a different gym every day during the next N days and lift $X[i]$ grams on day i. In order to improve his workout performance at the gym, he can buy exactly one pre-workout drink at the gym he is currently in and it will improve his performance by A grams permanently and immediately. In different gyms these pre-workout drinks can cost different amounts $C[i]$ because of the taste and the gym's location but its permanent workout gains are the same.

Before the first day of starting his workout plan, Alan knows he can lift a maximum of K grams. Help Alan spend a minimum total amount of money in order to reach his workout plan. If there is no way for him to complete his workout plan successfully output -1.

Input:

The first one contains two integer numbers, integers N and K – representing the number of days in the workout plan and how many grams he can lift before starting his workout plan respectively.

The second line contains N integer numbers $X[i]$ separated by a single space representing how many grams Alan wants to lift on day i.

The third line contains one integer number A representing permanent performance gains from a single drink.

The last line contains N integer numbers $C[i]$, representing the cost of the performance booster drink at the gym he visits on day i.

Output:

One integer number representing the minimal amount of money spent to finish his workout plan. If he cannot finish his workout plan, output $-\mathbf{1}$.

Constraints:

- $1 \leq N \leq 10^5$
- $1 \leq K \leq 10^5$
- $1 \leq X[i] \leq 10^9$
- $1 \leq C[i] \leq 10^9$
- $1 \leq A \leq 10^9$

Example input 1:

```
5 10000
10000 30000 30000 40000 20000
20000
5 2 8 3 6
```

Example output 1:

```
5
```

Explanation 1:

After buying drinks on days 2 and 4 Alan can finish his workout plan.

Example input 2:

```
5 10000
10000 40000 30000 30000 20000
10000
5 2 8 3 6
```

Example output 2 :

```
-1
```

Explanation 2:

Alan cannot lift 40000 grams on day 2.

Time and memory limit: 0.5s / 256 MB

Solution and analysis:

On day i Alan will be able to accomplish his goal if $X_i \leq K_{cur} + A * d$, where K_{cur} is the current weight he can lift and d is the number of additional pre-workout drinks he takes before the i^{th} day. Based on this, we can formulate greedy algorithm to solve the problem.

The solution is to iterate for each day while maintaining binary min-heap storing pre-workout drink prices. When current performance (K_{cur}) is not enough to accomplish the goal for a given day, calculate the minimum number of drinks he needs to take (d) in order to be able to lift X_i. To minimize amount of money spent, we take d smallest prices from the min-heap and add them to the result. Now we need to update current performance ($K_{cur} = K_{cur} + A * d$) and continue to iterate. In case there are not enough items in the heap, output -1. Time complexity for this is $O(n * \log n)$.

Problem D: The Light Square

Premier League and Rising Stars

Author:

Kosta Grujčić

Implementation and analysis:

Kosta Grujčić

Aleksandar Damjanović

Statement:

For her birthday Alice received an interesting gift from her friends – *The Light Square*. The Light Square game is played on an $N\ x\ N$ lightbulbs square board with a magical lightbulb bar of size $N\ x\ 1$ that has magical properties. At the start of the game some lights on the square board and magical bar are turned on. The goal of the game is to transform the starting light square board pattern into some other pattern using the magical bar without rotating the square board. The magical bar works as follows:

1. It can be placed on any row or column
2. The orientation of the magical lightbulb bar must be left to right or top to bottom for it to keep its magical properties
3. The entire bar needs to be fully placed on a board
4. The lights of the magical bar never change
5. If the lightbulb light on the magical bar is the same as the light of the square it is placed on it will switch the light on the square board off, otherwise it will switch the light on
6. The magical bar can be used an infinite number of times

Alice has a hard time transforming her square board into the pattern Bob gave her. Can you help her transform the board or let her know it is impossible? If there are multiple solutions print any.

Input:

The first line contains one positive integer number N representing the size of the square board.

The next N lines are strings of size N consisting of 1's and 0's representing the initial state of the square board starting from the top row. If the character in a string is 1 it means the light is turned on, otherwise it is off.

The next N lines are strings of size N consisting of 1's and 0's representing the desired state of the square board starting from the top row that was given to Alice by Bob.

The last line is one string of size N consisting of 1's and 0's representing the pattern of the magical bar in a left to right order.

Output:

Transform the instructions for Alice in order to transform the square board into the pattern Bob gave her. The first line of the output contains an integer number $0 \leq M \leq 10^5$ representing the number of times Alice will need to apply the magical bar.

The next M lines are of the form "col $\boldsymbol{X}$" or "row $\boldsymbol{X}$", where $\boldsymbol{X}$ is 0-based index of the matrix, meaning the magical bar should be applied to either row $\boldsymbol{X}$ or column $\boldsymbol{X}$.

If there is no solution, print only -1. In case of multiple solutions print any correct one.

Constraints:

- $1 \leq N \leq 2 \cdot 10^3$

Example input 1:

```
2
10
00
00
00
10
```

Example output 1:

```
1
row 0
```

Explanation 1:

You can apply magical bar on the first row or column. Valid orientations for magical bar to keep its magical properties are:

1 0 and 1
0

Example input 2:

```
2
11
11
00
01
11
```

Example output 2:

```
-1
```

Explanation 2:

For the second example it is not possible to transform the board into the pattern Bob wants.

Time and memory limit: 2.0s / 256 MB

Solution and analysis:

Considering that bulbs can be in one of two states and how magical bar operations, which we are allowed to apply, work, we can deduce that such operations are nothing else but bitwise XOR on rows or columns with the given number X.

The first obvious observation is that only parity matters, so we'll apply operations not more than once per row and column.

Let's take an arbitrary entry in the first matrix and its' corresponding entry in the second – $a_{i,j}$ and $b_{i,j}$. In order to make them equal, we only have to check i-th and j-th digits in the number x, since only these two digits can affect the value in $a_{i,j}$. There are several cases to consider:

1. $a_{i,j} = b_{i,j}$
    a. $x_i = 1$ and x_j=1
    b. $x_i = 1$ and $x_j = 0$
    c. $x_i = 0$ and $x_j = 1$
    d. $x_i = 0$ and $x_j = 0$
2. $a_{i,j} \neq b_{i,j}$
    a. ...

Now we'll simplify the first case when $a_{i,j} = b_{i,j}$. Two corresponding entries are equal, so there's nothing to change. However, if $x_i = 0$ then XOR won't change anything. Other cases are similar – if both x_i and x_j are equal to 1, we can choose to XOR both i-th row and j-th column or to not XOR any of them. If we write the first case in a logic expression we'll get something like this:

$$\left[(r_i \wedge c_j) \vee (\neg r_i \wedge \neg c_j)\right] \wedge \left[(r_i \wedge \neg c_j) \vee (\neg r_i \wedge \neg c_j)\right] \wedge \left[(\neg r_i \wedge \neg c_j) \vee (\neg r_i \wedge c_j)\right]$$

where r_i and c_j are logical variables assigned for i-th row and j-th respectively. If r_i is true, then we will XOR i-th row with x. Similar for c_j. One should note that disjunctions above are in fact exclusive.

We can further transform such an expression using well-known indetities in order to achieve full conjunctive normal form. In the following case we'll end up with:

$$(\neg r_i \vee c_j) \wedge (r_i \vee \neg c_j) \wedge \neg c_j \wedge \neg r_i.$$

Such form can be used to solve the problem as a two satisfability problem, that is known to be solvable in linear time.

For every disjunction of the form $(p \vee q)$ we'll add two directed edges $(\neg p, q)$ and $(\neg q, p)$. That will result in $2 \cdot N$ nodes for both rows and columns. If p and $\neg p$ happen to be in the same strongly connected component, then there is no solution. Otherwise, we can find topological sort in the condensation graph of our initial graph and based on the order of strongly

connected components assign values to each of the logical variables – if p is before $\neg p$ then p should be true.

Overal time complexity is $O(N^2)$ as well as space complexity.

Problem E: BubbleReactor

Premier League and Rising Stars

Author:

Kosta Bizetić

Implementation and analysis:

Kosta Bizetić

Miloš Šuković

Statement:

You are in charge of the BubbleReactor. It consists of N BubbleCores connected with N lines of electrical wiring. Each electrical wiring connects two distinct BubbleCores. There are no BubbleCores connected with more than one line of electrical wiring.

Your task is to start the BubbleReactor by starting each BubbleCore. In order for a BubbleCore to be started it needs to be receiving power from a directly connected BubbleCore which is already started. However, you can kick-start one BubbleCore manually without needing power. It is guaranteed that all BubbleCores can be started.

Before the BubbleCore boot up procedure its potential is calculated as the number of BubbleCores it can power on (the number of inactive BubbleCores which are connected to it directly or with any number of inactive BubbleCores in between, itself included)

Start the BubbleReactor so that the sum of all BubbleCores' potentials is maximum.

Input:

The first line contains one integer N, the number of BubbleCores.

The following N lines contain two integers U, V denoting that there exists electrical wiring between BubbleCores U and V.

Output:

Single integer, the maximum sum of all BubbleCores' potentials.

Constraints:

- $1 \leq N \leq 15000$
- $1 \leq U \neq V \leq N$

Example input:

```
10
0 1
0 3
0 4
0 9
1 2
2 3
2 7
4 5
4 6
7 8
```

Example output:

```
51
```

Explanation:

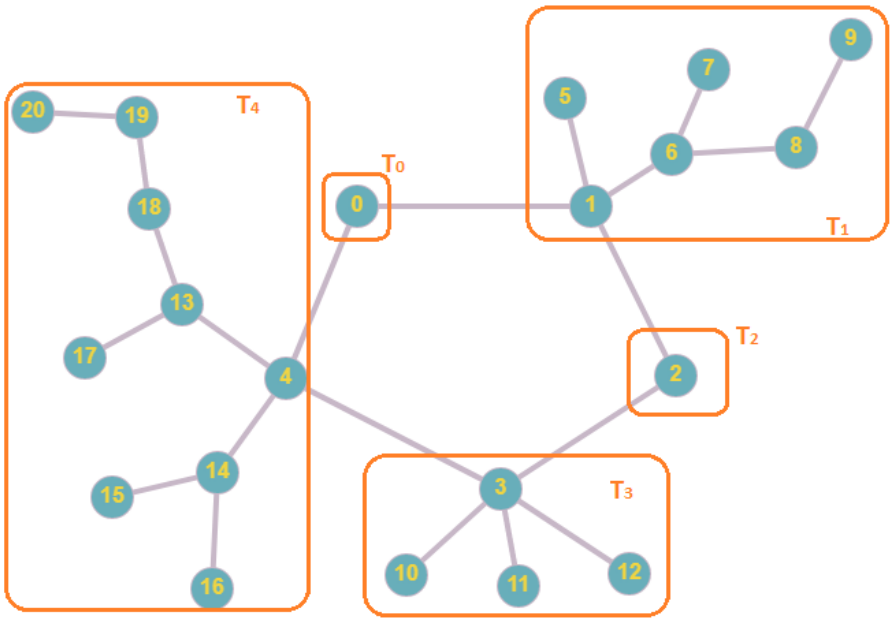
If we start by kickstarting BubbleCore 8 and then turning on cores 7, 2, 1, 3, 0, 9, 4, 5, 6 in that order we get potentials 10 + 9 + 8 + 7 + 6 + 5 + 1 + 3 + 1 + 1 = 51

```
Time and memory limit: 1.5s / 256 MB
```

Solution and analysis:

From the formulation, we can conclude that the BubbleReactor is a modification of the tree. It's a tree with additional edge (it's connected to the graph with exactly one cycle).

We can find the cycle using DFS. Let's focus on the cycle.



Let's look at trees $T_1, T_2, \ldots T_k$ and try solving a problem on a single tree. For one tree, the sum of node potentials is uniquely determined by selection of the starting node. When we pick a node, it will break the tree to K components - one for each edge from the node core. In each component, we know which node we have to pick next, when we decide to pick from that component. We can choose from which component to pick first, but that will not impact the sum of potentials - we will just add the same potentials in different order.

Now, how can we find the node which will generate the maximal sum of potentials? We can calculate the sum of potentials when starting from a randomly chosen node. We can do that using DFS, by tracking the number of nodes in every subtree created by removing the edge and potential generated in each subtree which we calculated during the process of powering the whole tree.

After we've calculated the sum of potentials for the fixed starting node, we can calculate the sum of potentials when starting from any adjacent node.

We want to track two values for each edge/node (e, v):

$node\_cnt$ – The number of nodes in the subtree created by removing the edge. We are interested in the subtree which contains the node v.

$potential\_in\_subtee$ – Te potential which will be generated by that same subtree if we entered v through the edge e.

If we calculated the sum of potentials for v_0, and we want to calculate for v_1 (the edge which we are focusing on is $v_0 \to v_1$).

$potential\_in\_subtee(v_1 \to v_0)$ =

$$= sum\_of\_potentials(v_0) - N - potential\_in\_subtee(v_0 \to v_1) + node\_cnt(v_1 \to v_0)$$

$$sum\_of\_potentials(v_1) = N + \sum_{v:adj(v_1)} potential\_in\_subtee(v_1 \to v)$$

$potential\_in\_subtee(v_1 -> v)$ should be calculated before or during the evaluation of $sum\_of\_potentials(v_0)$.

Let's go back to the cycle. For node c on the cycle, we have two options: we will enter it from another node on cycle or from the tree which coresponds to that node (tree which is "hanging from the cycle" and c is root of the tree). Our goal is to calculate overall maximal sum of potentials in case we started from subtree T_i with root c_i.

Let's say that we started from subtree T with root c. While powering the graph, at one point, we are in c. We should go through the cycle, but we have to choose in which order we want to consume the values on the cycle, since the order can impact the overall potential. We can calculate the optimal way for traversing the cycle from all nodes in the cycle in N^2 using DP. For node c_1, we must see in which order we want to traverse $c_2, c_3 \dots c_k$, which will be $dp[c_2, c_k]$.

$$dp[c_i, c_i] = node\_cnt(T_i)$$

$$dp[c_i, c_j] = max\left(\sum_{l=i}^{j} node\_cnt(T_l) + dp[next(c_i), c_j], \sum_{l=i}^{j} node\_cnt(T_l) + dp[c_i, previous(c_j)]\right)$$

We can easily calculate the potential P_i which we will get from the coresponding tree after entering from the cycle. So, we know the overall potential which we will get after entering c with the potential in subtree P is: $dp[previous(c),\ next(c)] - P + \sum_{i=1}^{k} P_i$.

By adding $N + P$ to that value, we get the overall sum of potentials in case we started graph powering from the c. Now we can calculate the sum of potentials for every node in subtree T using the algorithm for the tree, and find the maximum.

Problem F: Function Composition

Premier League and Rising Stars

Author:

Aleksandar Damjanović

Implementation and analysis:

Aleksa Milisavljević

Kosta Grujčić

Statement:

We are not going to bother you with another generic story when Alice finds about an array or when Alice and Bob play some game. This time you'll get a simple, plain text.

First, let us define several things. We define function F on the array A such that $F(i,1) = A[i]$ and $F(i,m) = A[F(i,m-1)]$, $m > 1$. In other words, function $F(i,m)$ represents composition $A[...A[i]]$ applied m times.

You are given an array of length N with non-negative integers. You are expected to give an answer on Q queries. Each query consists of two numbers – m and y. For each query determine how many x exist such that $F(x,m) = y$.

Input:

The first line contains one integer N – the size of the array A.

The next line contains N non-negative integers – the array A itself.

The next line contains one integer Q – the number of queries.

Each of the next Q lines contain two integers m and y (in this order).

Output:

Output exactly Q lines with a single integer in each that represent the solution. Output the solutions in the order the queries were asked in.

Constraints:

- $1 \leq N \leq 2 \cdot 10^5$
- $1 \leq A_i \leq N$
- $1 \leq Q \leq 10^5$
- $1 \leq m_i \leq 10^{18}$
- $1 \leq y_i \leq N$

Example input:

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

Example output:

```
3
0
1
1
0
```

Explanation:

For first query we can notice that $F(3,10) = 1$, $F(9,10) = 1$ and $F(10,10) = 1$. For second query no x satisfies condition $F(x,5) = 7$. For third query $F(5,10) = 6$ holds. For fourth query $F(3,1) = 1$. For fifth query no x satisfies condition $F(x,10) = 8$.

Time and memory limit: 1.0s / 256 MB

Solution and analysis:

We should first notice that this problem can be modeled as a graph theory problem. We define a graph in which there exists a directed edge from x to y iff $y = A[x]$. Now we can make a few observations about this graph:

1. Every connected component can be observed independentely,
2. Each node has exactly one outgoing edge,
3. Queries now ask for the number of x such that their m^{th} ancestor is y,
4. This graph has exactly one cycle per connected component.

When answering to given queries we have two cases:

1. **y is the node which doesn't belong to any cycle**. In this case we actually have to find the number of descendants of y on depth m from y. We'll do this offline in the DFS traversal of the forest we get when we remove the edges that belong to the cycle. At each point we remember how many nodes exist on certain depth. When we first enter node the y, we remember how many nodes we have discovered so far on the depth m from the y. When we finish processing the y, we can find out how many additional nodes we have discovered on the corresponding depth, based on the previously saved data.
2. **y is the node on the cycle**. This case is a bit trickier. We'll define an arbitrary node z on the cycle as the first node. Let the distance from the y to the z be t and the length of the cycle be l. If we had a query with the distance m on y, we could now analyze the query for $m + t$ on the node z. We can easily answer the modified query by observing a tree we get when we remove the outgoing edge from the z. We get the tree in which a node on the depth d is an answer to the query iff $d \geq m + t \wedge d \equiv m + t \ (mod\ l)$. All nodes which could be the answer for the modified query, but not for the original one are nodes which are on the distance $m + t$ from z, but the path from them to the z doesn't pass through the y. This can be answered in a similar manner as in the first case.

Problem G: Jumping Transformers

Premier League and Rising Stars

Author:

Nikola Nedeljković

Implementation and analysis:

Nikola Nedeljković

Slavko Ivanović

Statement:

You, the mighty *Blackout,* are standing in the upper-left $(0, 0)$ corner of NxM matrix. You must move either right or down each second.

There are K transformers jumping around the matrix in the following way. Each transformer starts jumping from position (x, y), at time t, and jumps to the next position each second. The x-axes grows downwards, and y-axes grows to the right. The order of jumping positions is defined as $\{(x, y), (x + d, y - d), (x + d, y), (x, y + d)\}$, and is periodic.

You want to arrive to the bottom-right corner $(N - 1, M - 1)$, while slaying transformers and losing the least possible amount of energy. When you meet the transformer (or more of them) in the matrix field, you must kill them all, and you lose the sum of the energy amounts required to kill each transformer.

After the transformer is killed, he of course stops jumping, falls into the abyss and leaves the matrix world. Output the minimum possible amount of energy wasted.

Input:

In the first line, integers N, M, representing the size of the matrix, and K, the number of jumping transformers.

In the next K lines, for each transformer, the numbers x, y, d, t, and e, represent the starting coordinates of the transformer, the jumping positions distance in the pattern described above, the time when the transformer starts jumping, and the energy required to kill it.

Output:

Print a single integer, the minimum possible amount of energy wasted, for *Blackout* to arrive at the bottom-right corner.

Constraints:

- $1 \leq N, M \leq 500$
- $0 \leq K \leq 5 \cdot 10^5$
- $0 \leq t \leq N + M - 2$
- $0 \leq e \leq 10^9$
- $d \geq 1$
- It is guaranteed that all 4 of the jumping points of the transformers are within the matrix coordinates

Example input:

```
3 3 5
0 1 1 0 7
1 1 1 0 10
1 1 1 1 2
1 1 1 2 2
0 1 1 2 3
```

Example output:

```
9
```

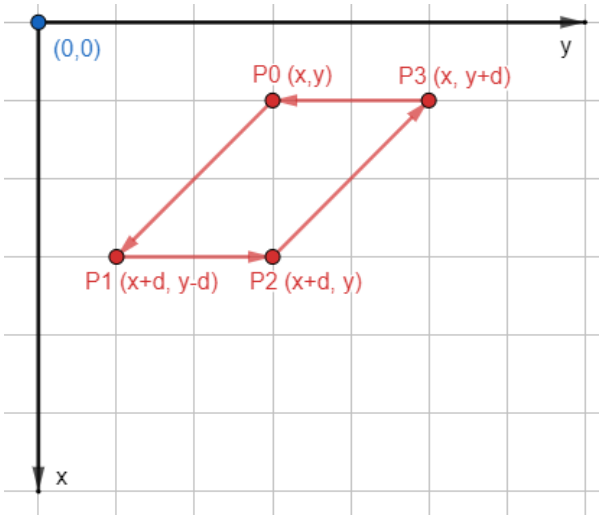
Explanation:

If Blackout takes the path from $(0, 0)$ to $(2, 0)$, and then from $(2, 0)$ to $(2, 2)$ he will need to kill the first and third transformer for a total energy cost of 9. There exists no path with less energy value.

```
Time and memory limit: 4.0s / 128 MB
```

Solution and analysis:

Every transformer is defined with (x, y, t, d, e), where t is starting time, d – the characteristic length for jump lengths (the length between $\{P0, P2\}, \{P1, P2\}, \{P0, P3\}$ on the picture below) and e the cost of destroying a transformer.



If we define Q as $Q = x + y$, it represents the position/length/time that you travelled on the matrix when you are in point (x, y). Now we can define the requirement for the meeting between Blackout and the transformer (meetup point) on every of the transformer's positions $(P0, P1, P2, P3)$. It is:

$$P0: Q = T \bmod 4, Q >= T$$
$$P1: Q = T + 1 \bmod 4, Q >= T + 1$$
$$P2: Q + d = T + 2 \bmod 4, Q + d >= T + 2$$
$$P3: Q + d = T + 3 \bmod 4, Q + d >= T + 3$$

Now we could populate the matrix with all of the meetup points with all the transformers with their cost of destroying and find the best road on that matrix. But it is important to note that Blackout could meet with the transformer zero, once or "two times" (the second meetup actually is imaginary since Blackout destroyed the transformer on the first one). So, the cost of the second meetup point of the same transformer is zero. It is important to notice that both meetup points for all situations to have two meetups with the same transformer, have one coordinate constant. These are such situations: $P1$ and $P2$; $P0$ and $P2$; $P0$ and $P3$.

All of the above allows us to find the best route and the minimum cost solution using dynamic programing. To represent this minimum cost we have matrix $dp[x][y][k][dir]$, where x, y represents the position in the matrix, k – represents the column/row on which Blackout gets into the current row/column and dir represents the direction from which Blackout entered the current position (x, y), from 0-up or from 1-left.

The recursion for such defined dp is:

$dp[i][j][k][0]$
$= dp[i-1][j][k-1][0]$
$+ Cost\ of\ destroying\ all\ transformers\ from\ (i,j)\ meetup\ point\ which\ are\ not\ already\ destroyed\ in\ some\ of\ the\ previous\ k\ steps\ up\ (or\ left).$

So the final solution is $min(dp[N-1][M-1][0][0], dp[N-1][M-1][0][1])$.

Time complexity $O(n^3)$. Memory complexity: $O(n^2)$ with dp memory optimizations.

Problem H: Guarding warehouses

Premier League and Rising Stars

Author:

Nikola Smiljković

Implementation and analysis:

Slavko Ivanović

Nikola Smiljković

Statement:

Bob Bubblestrong just got a new job as a security guard. Bob is now responsible for the safety of a collection of warehouses, each containing the most valuable Bubble Cup assets - the high-quality bubbles. His task is to detect thieves inside the warehouses and call the police.

Looking from the sky, each warehouse has the shape of a convex polygon. The walls of no two warehouses intersect, and of course, none of the warehouses are built inside of another warehouse.

Little did the Bubble Cup bosses know how lazy Bob is and that he enjoys watching soap operas (he heard they are full of bubbles) from the coziness of his office. Instead of going from one warehouse to another to check if the warehouses are secured, the plan Bob has is to monitor all the warehouses from the comfort of his office using the special X-ray goggles. The goggles have an infinite range, so a thief in any of the warehouses could easily be spotted.

However, the goggles promptly broke and the X-rays are now strong only enough to let Bob see through a single wall. Now, Bob would really appreciate if you could help him find out what the total area inside of the warehouses monitored by the broken goggles is, so that he could know how much of the warehouses area he needs to monitor in person.

Input:

The first line contains one integer N – the number of warehouses.
The next N lines describe the warehouses.
The first number of the line is integer c_i – the number corners in the i^{th} warehouse, followed by c_i pairs of integers. The j^{th} pair is (x_j, y_j) – the coordinates of the j^{th} corner. The corners are listed in the clockwise order.
Bob's office is positioned at the point with the coordinates $(0, 0)$. The office is not positioned inside any of the warehouses.

Output:

Print a single line containing a single decimal number accurate to at least four decimal places – the total area of the warehouses Bob can monitor using the broken X-ray goggles.

Constraints:

- $1 \leq N \leq 10^4$
- $3 \leq c_i \leq 10^4$
- $\sum_{i=1}^{N} c_i \leq 5 \cdot 10^4$
- $|x_j|, |y_j| \leq 3 \cdot 10^4$

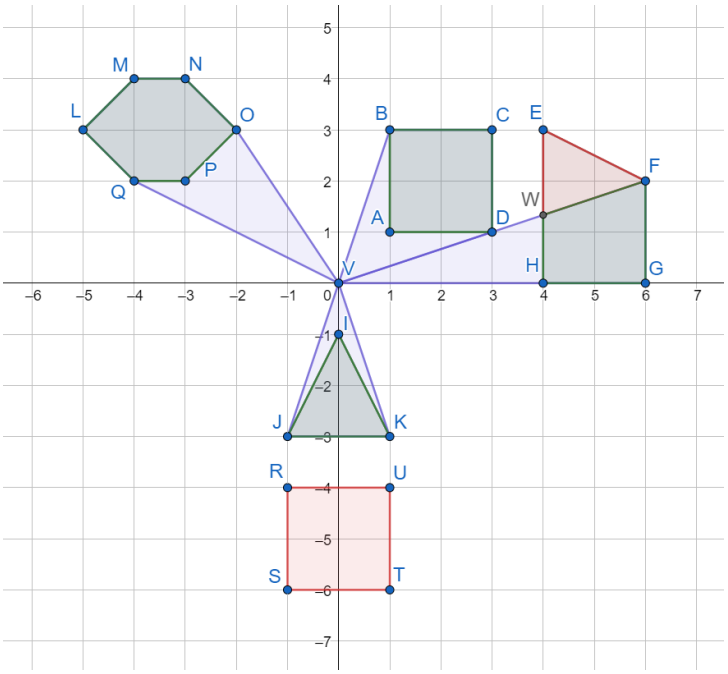
Example input:

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

Example output:

```
13.333333333333
```

Explanation:



Areas monitored by the X-ray goggles are colored green and the areas not monitored by the goggles are colored red.

The warehouses $ABCD$, IJK and $LMNOPQ$ are completely monitored using the googles.

The warehouse $EFGH$ is partially monitored using the goggles: part EFW is not monitored because to monitor each point inside it, the X-rays must go through two walls of warehouse $ABCD$.

The warehouse $RUTS$ is not monitored from Bob's office, because there are two walls of the warehouse IJK between Bob's office and each point in $RUTS$.

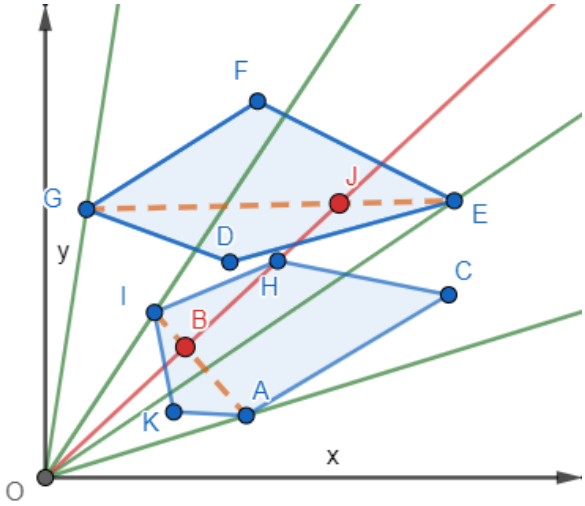
The total area monitored by the goggles is $P = P_{ABCD} + P_{FGHW} + P_{IJK} + P_{LMNOPQ} = 4 + 3.333333333333 + 2 + 4 = 13.333333333333$.

Time and memory limit: 1.0s / 256 MB

Solution and analysis:

The main idea for solving the problem is doing angular sweep line. Simply, Bob Bubblestrong will have to go through all the points of all polygons ordered by an angle that point makes with the Ox axis and update several data structures to calculate the observable area.

The ordered set of polygons that are intersected with the current position of the sweep line (the red line in the pictures below) should be maintained at every step. The first polygon in the set should be the one nearest to Bob (which is stationed in point O). It is easy to prove that two polygons cannot change the relative order in the set based on the sweep line position. For comparison between two polygons for the current sweep line position for example we could use the distance between O and the intersection point (points B and J in the picture below) between the sweep line (the red line on the picture) and the line made by polygon's points which made the smallest and the largest angle with the Ox axis.

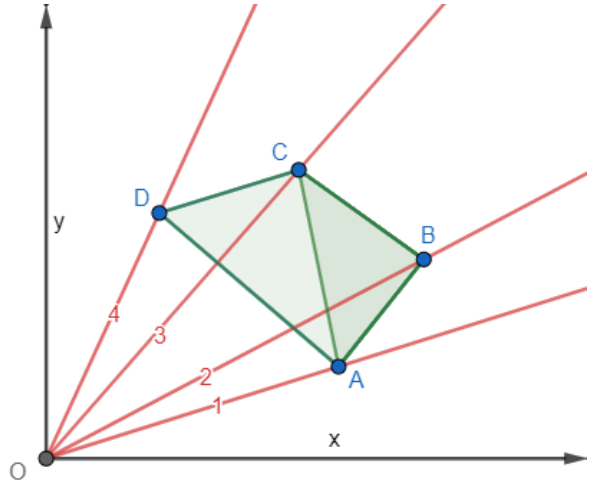


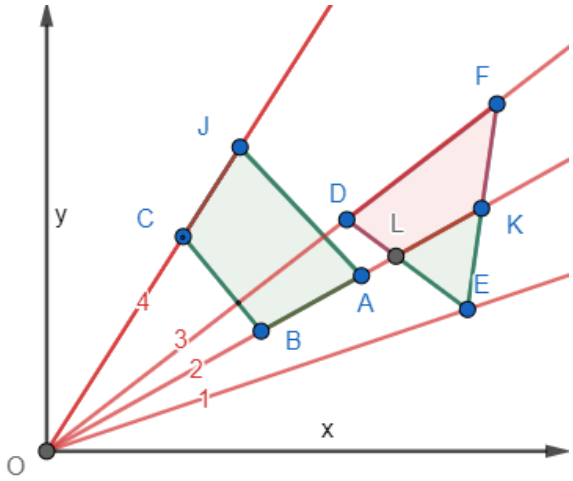
We move the sweep line from one point to another in increasing angle it makes with the Ox axis. When the sweep line visits each of the points, we update last visited points for that polygon. Total area of visible polygons should be updated only if:

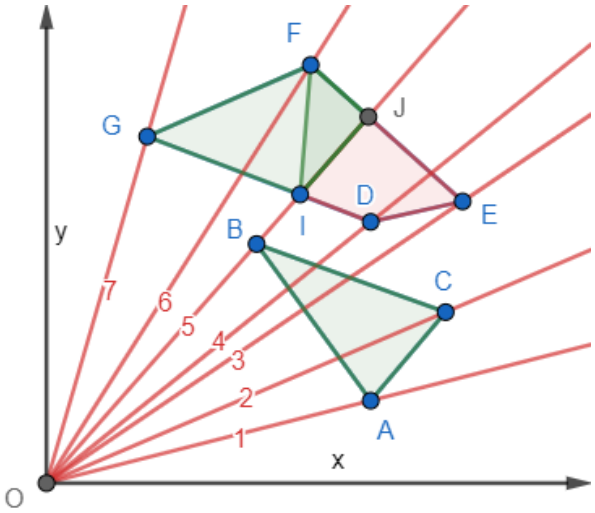
a) the sweep line visits the vertex of the nearest polygon – we should add the area of the triangle the current point makes with the last two visited points of the same polygon, one on the visible part of the convex hull and one of the invisible part of the convex hull,
b) the current point is the first visited point of a new nearest polygon – we should add the area of the triangle or quadrilateral on the next visible polygon (the points A or B on the second image below).

There are several possible situations demonstrated in the pictures below. The green areas are added to sum up the observable warehouses areas after noted sweep line steps.

Based on all of the above, the overall complexity is $O(n \log n)$.


C
D
B
y
4
3
2
1
A
x
O


F
J
D
y
C
L
K
E
A
4
3
B
2
1
x
O


F
G
J
D
E
y
B
I
C
7
6
5
4
3
2
A
1
x
O

Problem I: Xor Spanning Tree

Premier League and Rising Stars

Author:

Aleksa Milisavljević

Implementation and analysis:

Aleksa Milisavljević

Janko Šušteršič

Statement:

In a galaxy far, far away is the ancient interplanetary republic of Bubbleland, consisting of N planets. Between them, there are M bidirectional wormholes, each connecting a pair of planets. Bubbleland is a very centralized republic, with a capital planet Whiteplanet, from which any another planet can be reached using these wormholes. It is also guaranteed that no wormhole connects the planet to itself and that no two different wormholes connect the same pair of planets.

We call a path that begins at one planet, visits other planets and each of them at most once and returns to the starting point a *tour*. Interplanetary Safety Regulations guarantee that each planet belongs to at most one *tour* and that there are at most 42 *tours*.

After many eons of usage, wormholes need to be repaired and each wormhole has the cost W_i which needs to be payed for reparation. Unfortunately, the Senate of Bubbleland is short on budget. Therefore, they have decided only to fix as many wormholes as they need in order to have all the planets reachable from the capital and to pay as little money as they have to for this repair. However, the way in which the Senate calculates the cost is different. The cost of the set of reparations is binary xor of costs of each individual reparation, that is if reparations to be made have costs $A_1, A_2, \ldots, A_k$, the cost of the entire set is $A_1 \oplus A_2 \oplus \ldots \oplus A_k$.

Now the Senate would like to know how much money do they have to pay and also the number of different ways to achieve cost modulo 1000000007.

Input:

The first line of input contains two numbers N, the number of planets and M, the number of wormholes.

The following M lines contain three numbers U_i, V_i and W_i, meaning that there exists a wormhole connecting the planets U_i and V_i, with the repair cost of W_i.

Output:

Output two numbers, the cost of entire reparation and the number of different valid reparations with that cost modulo 1000000007.

Constraints:

- $1 \leq N \leq 100\,000$
- $1 \leq M \leq 100\,041$
- $1 \leq U_i \neq V_i \leq N$
- $0 \leq W_i \leq 100\,000$

Example input:

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

Example output:

```
1 1
```

Explanation:

We can repair wormholes $1, 2, 3, 5$ and 6, paying $5 \oplus 1 \oplus 2 \oplus 3 \oplus 4 = 1$, one can check that this is the cheapest repair in which all of the planets are connected and the only valid repair with that cost.

```
Time and memory limit: 2.0s / 128 MB
```

Solution and analysis:

We should first notice that there exists an upper bound of any solution cost as it is computed as binary xor of constrained numbers. Let us declare this minimum upper bound as A, ($A = 2^{17}$). The goal of this task is to compute minimal total cost of repairs as well as the number of different ways to achieve this price modulo $m = 1000000007$. This problem can be viewed as finding such edges of a connected graph which we can remove to create a minimum spanning tree so that the binary xor of weights of remaining edges gives minimal possible number. We can calculate binary xor of remaining edges as xor of weights of all edges and the ones we are removing.

We can store number of ways to remove such edges for every possible cost in an array. Length of this array will be A. Given that the graph has maximum of $k = 42$ cycles and that no node belongs to more than one cycle, we can find all cycles simply by using *Depth First Search,* for example. For each cycle we know we have to exclude exactly one edge so we can iterate over edges and count the appearance of every weight. Lets assume we have arrays of numbers of appearances of weights calculated for two different cycles. If we look at those arrays as polynomial coefficients of polynomials P and Q respectively, we need to calculate such product of polynomials so that for every $a_k x^k$ from P and $b_r x^r$ from Q we add $a_k b_r x^{k \oplus r}$ to the resulting polynomial $P \cdot Q$. We can easily calculate $P \cdot Q$ in $O(A^2)$ complexity, but we can do this quicker using ***Fast Walsh-Hadamard transform (FWHT)*** which transforms coefficient representation of polynomial into point-value representation with the property that above mentioned multiplication of polynomials maps to point-wise multiplication of transformed arrays. We can calculate FWHT of an array in $O(A \cdot logA)$ time and point-wise multiplication in $O(A)$, where A is length of the arrays.

After going through all of the cycles and multiplying transforms of their respective arrays, we need to compute inverse WHT of the resulting array. Inverse Walsh-Hadamard transform is calculated in the same manner as the regular one, but divided by the length of the array. We can compute division modulo some prime number p using Fermat's little theorem which states that $a^p \equiv a \, (mod \, p)$. Hence we multiply FWHT with A^{m-2} calculated using fast modular exponentiation to aquire inverse transform. Therefore, time complexity of this part of the problem is $O(k \cdot A \cdot logA)$.

In this way, we can calculate the number of different ways to achieve any price. But in order to know the smallest price that can be achieved, we need another array in which we only remember 1if price can be achieved or 0 if it can't. We recalculate this array for each cycle. We need additional array because of the case in which the number of possible ways to achieve some price is divisible by m. Similarly, time complexity of this part of the problem is $O(k \cdot A \cdot logA)$ and therefore total time complexity is $O(k \cdot A \cdot logA)$.

Problem J: Product tuples

Premier League only

Author:

Nikola Nedeljković

Implementation and analysis:

Aleksa Milisavljević

Nikola Nedeljković

Statement:

While roaming the mystic areas of Stonefalls, in order to drop a legendary loot, an adventurer was given a quest as follows. He was given an array $A = \{a_1, a_2, \ldots, a_N\}$ of length N, and a number K.

Define array B as $B(q, A) = \{q - a_1, q - a_2, \ldots, q - a_N\}$. Define function F as $F(B, K)$ being the sum of products of all K-tuples of elements in array B. For example, if the array B is [2,3,4,5], and with $K = 3$, the sum of the products of all 3-tuples is $2 * 3 * 4 + 2 * 3 * 5 + 3 * 4 * 5 + 2 * 4 * 5 = F(B, 3)$.

He was then given a number Q, the number of queries of two types:

- Type 1: Given q, i, and d calculate $F(B(q, A), K)$ where we make a change to the initial array as $A[i] = d$**.**
- Type 2: Given q, L, R, and d calculate $F(B(q, A), K)$ where we make a change to the initial array as $A[i] = A[i] + d$ for all i in range $[L, R]$ inclusive.

All changes are temporarily made to the initial array, and don't propagate to the following queries.

Help the adventurer calculate the answer to the quest, and finally get that loot!

Input:

In the first two lines, the numbers N and K, the length of the initial array A, and tuple size, followed by $a_1, a_2, a_3, \ldots, a_N$, elements of array A**,** in the next line. Then follows the number Q**,** the number of queries. In the next Q lines come queries of the form:

- $1\ q\ i\ d$**,** for type 1
- $2\ q\ L\ R\ d$, for type 2,

as explained above

Output:

Print Q lines, the answers to the queries, modulo 998244353.

Constraints:

- $1 \leq N \leq 2 \cdot 10^4$
- $1 \leq K \leq N$
- $1 \leq L \leq R \leq N$
- $1 \leq Q \leq 10$
- $1 \leq i \leq N$
- $0 \leq A_i, q, d \leq 10^9$

Example input:

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

Example output:

```
85
127
63
```

Explanation:

In the first query array $A = [1, 2, 3, 4, 5], B = [5, 4, 3, 2, 1]$, the sum of products of 2-tuples = 85.

In the second query array $A = [1, 2, 3, 4, 2]$, $B = [5, 4, 3, 2, 4]$, the sum of products of 2-tuples = 127

In the third query array $A = [1, 3, 4, 4, 5], B = [5, 3, 2, 2, 1]$, the sum of products of 2-tuples = 63

Time and memory limit: 9.0s / 128 MB

Solution and analysis:

Let us first solve following two tasks:

1. Given an array a of N elements find sum of products of all K-tuples for each K with following queries: What would be the result if we would change a_i to d?
   Let us construct segment tree above the given array. Let's consider node above interval $[L,\ R]$. In it we will for each K from 0 to $R - L + 1$ remember sum of products of all K-tuples of elements in interval $[L, R]$ in an array s, where we define $s_0 = 1$. Then we calculate values for some non-leaf node. If array l is the array with results for left child of current node and array r is array with results for right child of current node, we can notice that equation $s_i = \sum_{j=0}^{i} l_j r_{i-j}$ holds, therefore we can apply fast polynomial multiplication to calculate result for current node faster. Now for each query we need only to recalculate $O(\log N)$ nodes. Time complexity is $O(N \log N)$ per query.

2. Given an array a of N elements find sum of products of all K-tuples for each K with following queries: What would be the result if we increased all elements for d?
   Now we can calculate an array which contains all the results for original array in the same manner we did it in first task. Let's name that array t. We can notice that following equation holds: $s_i = \sum_{j=0}^{i} \binom{N-j}{i-j} t_j d^{i-j}$, where s is the name of array which contains results for array if all elements would be increased for d. We can also apply fast polynomial multiplication for this case. We multiply these two polynomials: $(N-j)!\, t_j$ and $\frac{d^j}{(i-j)!}$. Time complexity is $O(N \log N)$ per query. This is actually slightly modified version of a known Schönhage multiplication algorithm for polynomials.

Using these two ideas we can solve original task. We will consider negative values of array elements instead of original, and after that we will construct segment tree above the modified array as we did in the first task. When we get query of first type, we'll change an array element as we did in first task and after that use modification we used for increasing all elements for d in the second task. When we get query of second type, we'll use constructed segment to split array in intervals, for which condition that they either entirely belong to query interval $[L,\ R]$ or they are entirely out of it holds, rather than changing elements and building segment tree once more. After that we will greedily multiply intervals of same type (intervals are of the same type if they are either both inside query interval or both outside of it) ascending by size. Then, we will have two arrays, in one result for interval $[L,\ R]$ in another result for rest of the array. We can now apply reasoning from the second task to decrease all elements of first array by t. Then we can again apply fast multiplication to get results for entire array and yet once more apply modification of Schönhage algorithm to increase all elements for d.

Problem K: Alpha planetary system

Premier League only

Author:

Bojan Vučković

Implementation and analysis:

Bojan Vučković

Kosta Bizetić

Statement:

Three planets X, Y and Z within the Alpha planetary system are inhabited with an advanced civilization. The spaceports of these planets are connected by interplanetary space shuttles. The flight scheduler should decide between 1, 2 and 3 return flights for every existing space shuttle connection. Since the residents of Alpha are strong opponents of the symmetry, there is a strict rule that any two of the spaceports connected by a shuttle must have a different number of flights.

For every pair of connected spaceports, your goal is to propose a number 1, 2 or 3 for each shuttle flight, so that for every two connected spaceports the overall number of flights differs.

You may assume that:

- every planet has at least one spaceport
- there exist only shuttle flights between spaceports of different planets
- for every two spaceports there is a series of shuttle flights enabling traveling between them
- spaceports are not connected by more than one shuttle

Input:

The first row of the input is the integer number N, representing overall number of spaceports. The second row is the integer number M representing number of shuttle flight connections.

Third row contains N characters from the set $\{X, Y, Z\}$. Letter on I^{th} position indicates on which planet is situated spaceport I. For example, "XYYXZZ" indicates that the spaceports 0 and 3 are located at planet X, spaceports 1 and 2 are located at Y, and spaceports 4 and 5 are at Z.

Starting from the fourth row, every row contains two integer numbers separated by a whitespace. These numbers are natural number smaller than N and indicate the numbers of the spaceports that are connected. For example, "12 15" indicates that there is a shuttle flight between spaceports 12 and 15.

Output:

The same representation of shuttle flights in separate rows as in the input, but also containing a third number from the set {1, 2, 3} standing for the number of shuttle flights between these spaceports.

Constraints:

- $3 \leq N \leq 100\,000$
- $2 \leq C \leq 100\,000$

Example input:

```
10
15
XXXXYYYZZZ
0 4
0 5
0 6
4 1
4 8
1 7
1 9
7 2
7 5
5 3
6 2
6 9
8 2
8 3
9 3
```

Example output:

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

Time and memory limit: 2.0s / 128 MB

Solution and analysis:

Let $G = (V, E)$ denote the graph with vertices (spaceports) V partitioned into three subsets X, Y and Z, and with E denoting the edges (shuttle flight connections) between vertices. Denote by the edge weight the number of return flights between vertices (spaceports). Proper coloring of graph is assigning integer values to graph vertices so that any two adjacent vertices obtain different color (value).

Here is a description of the construction of an algorithm leading to a solution of the problem.

1. Form an arbitrary spanning of tree T of graph G (for example using BFS).
2. Check if the graph G contains an odd cycle:
   Properly color the vertices of T with colors from $\{1, 2\}$. Depending on whether there exists an edge in E connecting two vertices of T with the same color, we have two cases:
   a. For every $uv \in E$ vertices u and v are assigned different colors in T.
   b. There exists $uv \in E$ such that the color of u equals the color of v in T. That means that the path from u to v has an even length and together with uv it forms an odd length cycle C in graph G.
3. Depending on 2, we assign weights to the edges of E to satisfy the conditions of the problem:
   a. Assign weight 3 to every edge in E. Let $\sigma(u)$ denote the sum of weights of the edges incident to u modulo 3. Take arbitrary vertex v that is colored 1 in T and calculate the distance to v for every vertex of V in tree T. Let's say that the maximal distance is k. We implement an algorithm from the following pseudo-code:

```
for n := k down to 1:
    for every vertex u whose distance to v in T is n, and whose
    first vertex on the path uv is w, we choose a weight from
    {1,2,3} for uw is such a way that σ(u) equals the color of u
    in T.
        if (σ(v) ≠ 2) or (v has only one adjacent vertex in
        graph G):
            return assigned weights
        else:
            Let u and w denote vertices with
            uv ∈ E and wv ∈ E.
            Add 1 modulo 3 to weights of uv and wv
              (but use 3 instead of 0 for weight).
```

return assigned weights

b. Assign weight 3 to every edge in E. Let $\sigma(u)$ denote the sum of weights of edges incident to u modulo 3.

Assign to every vertex of T color 0 if it belongs to set X, color 1 if it belongs to set Y, and color 2 if it belongs to set Z.

Take an arbitrary vertex v on the odd cycle C that is obtained in 2a. Next, for every vertex of V calculate its distance to v in tree T. Let's say that the maximal distance is k. We implement an algorithm similar to the first one:

for $n := k$ down to 1:

for every vertex u whose distance to v in T is n, and whose first vertex on the path uv is w, choose a weight from $\{1, 2, 3\}$ for uw is such a way that $\sigma(u)$ equals the color of u modulo 3 in T.

if ($\sigma(v)$ equals the color of v in T):

return assigned weights

else:

Let m be the value from the set $\{1, 2\}$ such that $\sigma(v) + m$ equals the color of v modulo 3 in T. Denote vertices of cycle C with u_0, u_1, …, u_{2n}, u_{2n+1} where $v = u_0 = u_{2n+1}$.

for weight of each edge $v_{2i}v_{2i+1} \in C$:

Add $2m$ modulo 3 (use values from $\{1, 2, 3\}$ for weights)

for weight of each edge $v_{2i+1}v_{2i+2} \in C$:

Add m modulo 3 (use values from $\{1, 2, 3\}$ for weights)

return assigned weights

Any of the steps described above can be done in at most $O(|E|)$ steps, thus the complexity of the solution equals $O(|E|)$.

Qualification problems

Round 1: Ada and Tic-Tac-Toe

Statement:
Ada the Ladybug was playing 3D Tic-Tac-Toe against her good friend Velvet Mite Vinit. They were playing on a $3 \times 3 \times 3$ field. Sadly, Vinit had to go home earlier so the game was left unfinished. Ada wants to know who would win the match (if they both played optimally). You also know that Ada started.

Input:
The first line contains a single integer $1 \leq T \leq 1000$, the number of test-cases.

Each test-case will contain 9 lines with three character each, representing the field ('**.**' for an empty place, '**x**' for a move of Ada and '**o**' for a move of Vinit). You are assured that the game is unfinished and valid.

First three lines represent the top field, the next three lines represent the middle field and the last three lines represent the bottom field.

There will be a blank line before each test-case.

Output:
For each test-case, print the name of the winner (Ada or Vinit).

Example input:
```
7

..o
o.x
xox
x.x
...
x.o
...
o.o
.x.

.x.
...
..o
o.x
x..
..o
..x
...
...

o..
...
o.x
.x.
...
.x.
```

```
o.o
.xx
oox

..x
..x
.o.
..x
..x
xo.
o..
..o
...

...
..x
..x
.o.
..x
o..
...
...
x.o

...
..x
..o
...
...
xo.
ox.
...
.ox

o..
x..
...
oxx
..o
.ox
..o
.x.
.x.
```

Example output:

```
Vinit
Vinit
Ada
Vinit
Ada
Vinit
Vinit
```

Example input:

```
1

xox
x.o
oox
o.o
x.x
```

```
xxo
x.o
oxo
ox.
```

Example output:

```
Ada
```

Time and memory limit: 1s / 64 MB

Solution:

The solution is based on minimax algorithm. We will call a state winning if the player that is to make a move has the winning strategy or call it losing otherwise. The state is winning if and only if there is at least one move that leads to a losing state or to an end state. When you are to decide whether a state is winning or losing first check if there is a move that leads to an end state and then consider other moves. For many states the player that has the cell in the middle has the winning strategy. Because of that if the cell in the middle is empty first check if placing a marker in it leads to a losing state and if that's not the case then check other moves in any order. In order not to solve the same state more than once we can store the results in a hash table.

Problem source: SPOJ
Solution by:
Name: Tadija Šebez

Round 1: Building Subways

Statement:
The department of progress of a city of Orbita Interconnected Advanced (OIA) has decided to address the complex construction of an underground rapid transit. The subway network, as it is called, will link zonal centers of consolidation of the city, which today is connected with tram lines, following the same route as these, which it will replace.

The problem is that initially budget that is account could be not enough to perform all the work. Part of the tour will continue by tram. Then a set of sections will be chosen to become subway so that the greatest distance traveled by tram from any center not modernized to the subway network is minimal. A center is considered modernized when a section of subway has one end in that center. The subway network must be done so that it links all modernized centers.

You must write a program that, given a description of the $N-1$ current sections of trams that connect all N centers (so that it is possible to travel between any of the two) including its length in kilometers L_i, and the budget K with which the department has expressed in the same unit, calculate what will be the maximum distance mentioned above, leaving engineers to determine the detailed list of sections to be built.

Input:
You receive:

- A line with numbers N and K ($2 \leq N \leq 200000$, $K \leq 1000000000$)
- $N-1$ lines with X_i, Y_i, L_i, centers connected by the section i and its length. ($1 \leq X_i$, $Y_i \leq N$, $1 \leq L_i \leq 10000000$).

Output:
You must output a line with an integer number representing the maximum distance traveled by tram to reach the underground.

Example input:
```
5 14
1 2 3
1 3 5
5 2 7
4 2 6
```

Example output:
```
6
```

Time and memory limit: 1s / 256 MB

Solution:

Let's solve another problem first.

Is there a valid construction so that the maximum distance traveled is at most D, for some fixed D.

For this we will need to precompute two values first. We will root a tree in any node. After that we will use DFS to precompute the depth of the nodes. Now we will run another DFS to calculate the deepest node in a subtree of the node. We will do this for each node. It can be done by simply picking the maximum of $children + length$. Also, we need BFS to calculate the maximum distance for each node to some other that isn't in its subtree. It is done in the same way, more or less.

Now let's just try all posibilities to be the root of our construction.

We will use DFS. But only if the distance from the chosen node to the farthest that isn't in its subtree is less than D.

We should keep two values. One will be the global variable, which will show us how much budget we can spend. Another one will be forwarded in DFS function, showing us what depth we have to cut in that subtree, so that the construction will be valid for value D, also let's call this value z.

So let's say we are in some node v. We will run DFS to its child if the deepest node in subtree of child is on a bigger depth than z. In case we go to that child, we will substract the length of edge from the budget. Also we will forward $z - length$ of edge to DFS function of that child. We will check this for every child. If we can cut all with this budget there exists a valid solution.

So because we are using DFS only, time complexity of this checking is $O(n)$. And because there are $O(n)$ nodes to choose as a root it is $O(n^2)$.

But let's make one important observation. Let's say we are trying to root some node v. If there is some node in subtree of v, a node for which we know for a fact that can't be the root, then v also can't be the root, because the price for rooting v is at least that much.

So if we iterate from deepest nodes, to those with less depth, we won't check anything more times than needed. So we will be doing DFS in order of depths. Therefore, we get complexity $O(n)$.

Instead of trying for every possible value of D, we can binary search for it. Because we are not looking for the exact value of D, but we want to make it at most D.

Finally we get $O(n * \log_2(10^{18}))$.

Problem source: COJ
Solution by:
Name: Aleksa Miljković

Round 1: Input

Statement:

In a recent programming contest, one of the problems was about tiling floors with rectangular tiles. The input specification reads like this:

The input contains several floors. The first line of the input gives the number of floors. Each floor is described in several lines. The first line contains two positive integers: the length and width of the floor, in millimeters. A floor is at most $40000\,mm$ long or wide. The next line contains a single number: the number t of tiles ($1 \leq t \leq 100$).

The following t lines each contain the description of a tile. A tile is given as four integers: $x_1\ y_1\ x_h\ y_h$
where $(x_1\ y_1)$ are the coordinates of the lower left corner of the tile, and $(x_h\ y_h)$ are the coordinates of the upper rightmost corner of the tile. A tile always has a positive area. The order of the coordinates of the floor and those of the tile coincide, of course. You may assume that the tiles are mutually disjoint, and cover the floor, the whole floor, and nothing but the floor.

The last line of this specification raised some problems. Not for the contestants, but for the judges. Some of the test cases consist of many tiles. How can we be sure that our input file meets this condition? What we need is a checking program that verifies this condition.

Problem

Given an input file in the above format, find out for each floor whether the tiles

1. are disjoint,
2. do not lie outside the floor,
3. do cover the floor.

Input:

The input contains several floors. The first line of the input gives the number of floors. Each floor is described in several lines. The first line contains two positive integers: the length and width of the floor, in millimeters. A floor is at most $40000\,mm$ long or wide. The next line contains a single number: the number t of tiles ($1 \leq t \leq 100$).

The following t lines each contain the description of a tile. A tile is given as four integers: $x_1\ y_1\ x_h\ y_h$
where $(x_1\ y_1)$ are the coordinates of the lower left corner of the tile, and $(x_h\ y_h)$ are the coordinates of the upper rightmost corner of the tile. A tile always has a positive area. The order of the coordinates of the floor and those of the tile coincide, of course.

Output:

For each floor the output contains a single line, containing one of the following words:

- **NONDISJOINT** if overlapping tiles occur;
- **NONCONTAINED** if no overlapping tiles occur, but some tiles go outside the floor;
- **NONCOVERING** if no overlapping tiles occur, and no tiles go outside the floor, but some parts of the floor are not covered;
- **OK** if none of these are true.

Example input:

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

Example output:

```
NONDISJOINT
NONCONTAINED
NONCOVERING
OK
```

Time and memory limit: 1s / 64 MB

Solution:

The problem does not require any higher knowledge, but just a little bit of reasoning. We have to determine at most 3 things (if one is true, then we don't need to determine the next one):

1. Are there any two rectangles which overlap with one another;
2. Is there is any rectangle which has a part (or more) out of the bounds of the floor;
3. Do all rectangles cover the entire floor (without overlapping and parts outside the bounds).

For N number of rectangles, there is $\binom{N}{2}$ pairs of rectangles. Because the number of rectangles is at most 100, we are able to go through every pair of rectangles there is under the given time limit. Note the fact that every side of the rectangle is parallel to either abscissa or ordinate. It makes the determination of overlapping not that difficult. Two rectangles (a, b) with their respective given coordinates $(x_{la}\ y_{la}\ x_{ha}\ y_{ha})$ and $(x_{lb}\ y_{lb}\ x_{hb}\ y_{hb})$ **DO NOT** overlap if at least one of the following conditions is satisfied:

1. $x_{la} \geq x_{hb}$;
2. $x_{ha} \leq x_{lb}$;
3. $y_{la} \geq y_{hb}$;
4. $y_{ha} \leq y_{lb}$.

(touching of two rectangles is not considered overlapping)

If neither of the previous conditions were satisfied for at least one pair of rectangles, then overlapping occurs. The answer is "**NONDISJOINT**".

If overlapping does not occur, we have to move on. We can easily check if a rectangle goes outside the floor simply by going through every rectangle once and asking one simple question. Coordinates of the lower left corner of the floor are $(0, 0)$ and of the upper right corner of the floor are (L, W). $\boldsymbol{L}$ is the length, while W is the width of the floor. Thus, if one of the following conditions is true for a rectangle with its given coordinates $(x_l\ y_l\ x_h\ y_h)$, the rectangle **IS NOT** on the floor with its whole surface. The conditions are as follows:

1. $x_l < 0$;
2. $y_l < 0$;
3. $x_r > L$;
4. $y_r > W$.

If at least one of the previous conditions is true, then there are some parts outside the floor. The answer is "**NONCONTAINED**".

And lastly, if there is no overlapping and no parts outside the floor, we have to determine if all rectangles cover the whole floor's surface. We will do that by calculating a sum of surfaces of every rectangle. The surface of the floor is equal to $L \times W$. If the sum of surfaces of rectangles is less than the floor's surface, the floor is not entirely covered. The answer is "**NONCOVERING**". Otherwise, if the sum is equal to the floor's surface, the answer is "**OK**".

And that is it, the problem is solved.

Problem source: COJ
Solution by:
Name: Uroš Berić

Round 1: First to meet the spaceship

Statement:

A spaceship has been sighted heading towards Earth. For the entire time that humanity has been monitoring it, it has not altered its course, it only changed speed for unknown reasons. As such, all the possible places on Earth where the spaceship might end up landing form a straight line; depending on how much it changes speed, it will land at different times, meaning a different point on this line due to Earth's rotation.

N people want to be the first to meet the aliens - they picked a point x_i on this line and wait at that point in their vehicle with speed v_i. Now they are all anxiously waiting for the spaceship's arrival. NASA has given a list of Q most likely locations where the spaceship might end up landing - and everyone wants to know who would get to be the first to meet the aliens if the spaceship landed at each of the given points. They turned to you for help!

Input:

The first line contains two integers $1 \leq N, Q \leq 300000$ - the number of people wishing to meet the aliens and the number of possible points where the spaceship might land.

The following N lines contain two integers $0 \leq x_i < 10^9$ and $0 < v_i \leq 10^9$ - the point on the line where the i^{th} person is waiting and the speed of his vehicle. Additionally, $x_i = x_j \longrightarrow v_i \neq v_j$.

The last line contains Q distinct integers $0 \leq q_i < 10^9$ - the points on the line where the spaceship might end up landing. You may assume the spaceship will not land at any point containing a person waiting in a vehicle.

Output:

For each query q_i output the number of people who will arrive at the spaceship first if it lands at that point. A person at x_j with speed v_j will arrive at q_i in time $\frac{|x_j - q_i|}{v_j}$. The people who will arrive at the spaceship first are those j for whom the fraction is minimal out of all people. Then, in the same line, output the 1-based indices of these people as they were given in the input, sorted in ascending order.

Example input:

```
4 7
10 5
30 1
20 4
100 1
5 31 22 15 85 60 61
```

Example output:

```
1 1
```

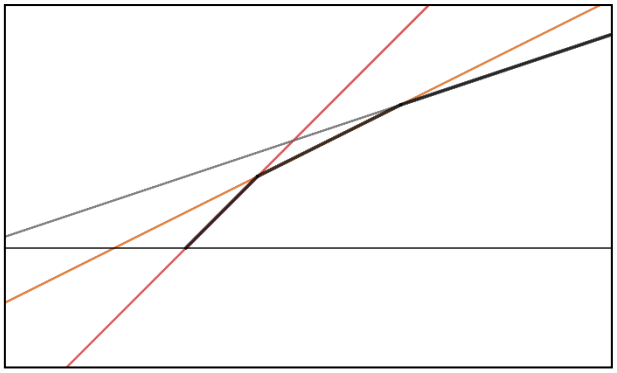
```
1 2
1 3
1 1
2 1 4
2 1 3
1 1
```

Time and memory limit: 3s / 64 MB

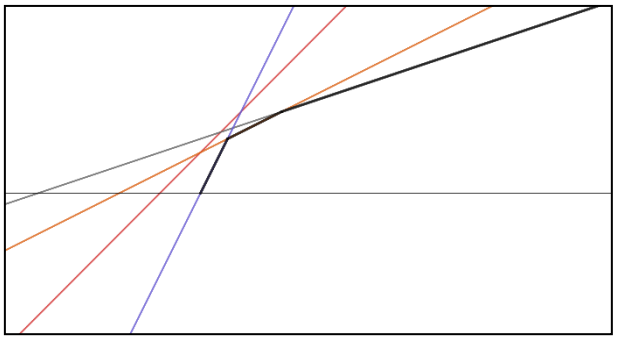
Solution:

The first thing to note is that we can process the queries in an offline manner. This means that we will first input all of them and then solve in them in some order we deem most appropriate to find a quick solution. The main idea in this problem is that we solve each query as two different queries, in a different offline manner each, and combine these answers to find the answer to each query.

For every query, we will find which people are closest to it from the right and from the left independently. Combining these answers will trivially let us answer the full problem.



Here we will describe how to find what person is closest to it from the left. The other case is obviously completely analogous. We will sort the people and the queries into one single array sorting them according to their X value. Now we are interested in how we want to process the incoming information of two types: "New person" or "Answer query". The trick here is to visualize a number line. We imagine each person as a linear function of time over distance, meaning the y-coordinate of some line is the time required for him to get to this point. Now we are interested in finding the minimum of our currently added lines. This is, of course, reminiscent of finding a convex hull. We want to find the lowest currently on the set of added lines in a certain point on a number line.



To do this, we maintain a stack of lines. Now, obviously, if some line is no longer optimal at some point, it will never be the optimal line again, so we can remove it from our stack. Also, our newest line will be optimal for some time, and then be overtaken by an older line that is steeper. This means our stack will be always increasing in the steepness of its lines. Noting this, we can solve by maintaining the stack, sorting them by "overtaking points". This means that when adding our new line, and if the top 3 lines from the stack are l_1, l_2, l_3, and l_2 overtakes l_1 at $x = d_1$, and l_3 overtakes l_2 at $x = d_2$, then we pop l_2 from the stack if $d_2 < d_1$. We do this because if l_3 overtakes l_2 before l_2 overtakes l_1, this means that l_2 has already lost its turn as the dominant line when it overtakes l_1, and we don't need it in the stack any more. When answering one of the queries, we simply pop the lines that are overtaken by the next one in the stack up to this point as they will never be optimal again, and answer with the according line at the top of the stack. Some extra care is needed

when the query is exactly an overtaking point, but this can be effectively resolved with a bit of care where we put $<$ and where we put $\leq$.

Also, a thing to note is that, as queries are always in different points, the max cardinality of the answer sets we need to memorize for the "closest right" part of the algorithm is $O(N+Q)$, as every query that isn't in an overtaking point requires us to memorize only one answer, while the answers in the overtaking points are at most $2N$, as every person overtakes and is overtaken exactly once.

Now merging the answers is easy. If the optimal answer from the left and the right is equal, then the answer to this query is the union of the two answer sets, if not we just choose the larger one.

This algorithm gives us something that is essentially a $O(N+Q)$ solution, but a logarithmic factor is added with the sorting at the beginning, meaning that this solution has $O((N+Q)\log(N+Q))$ time complexity, and $O(N+Q)$ memory complexity.

Problem source: SPOJ
Solution by:
Name: Pavle Martinović

Round 1: Mysteriousness of a Logical Expression

Statement:

Cho decided that it is time to find the love of his life. After conducting intensive online research, he found out that to stand a chance, he has to be intelligent, handsome, kind, charismatic, confident, funny, responsible, reliable, straightforward, mysterious, a gentleman.....

Cho was puzzled. He thought he already had all of these characteristics. After taking a definitely legitimate online personality test, he realized he lacked just one of them - he was not mysterious enough.

He decided he will only say statements with as many interpretations as possible; that way it will always be unclear what he actually means, and that should grant him an aura of unpredictability and mysteriousness.

Cho prepared some pickup lines, and now he wants to know how mysterious they sound - that is, in how many ways they can be interpreted to still make sense.

Input:

Each pickup line can be represented by a logical expression. A valid logical expression exp can be

- $exp = X$
- $exp = OR(exp, exp)$
- $exp = AND(exp, exp)$
- $exp = NOT(exp)$

where X is a boolean variable, and OR, AND, NOT are basic boolean operations.

The first line contains an integer $1 \leq T \leq 6$, the number of expressions.

Each of the T subsequent lines contains one valid expression as described above (i.e. it is correctly parenthesized, with no additional spaces).

$|exp| \leq 700000$ and the sum of $|exp|$ in an input file ≤ 1850000

Output:

For each expression, find the number of ways that True/False values can be (independently) assigned to the variables X in the expression so that the whole expression evaluates to True.

Since this value could be huge, output it modulo $10^9 + 7$.

Example input:

```
2
AND(X,OR(X,X))
NOT(OR(X,X))
```

Example output:

```
3
1
```

Time and memory limit: 1s / 64 MB

Solution:

Let's call all expressions of the first type basic (just a boolean variable), and let's call all the others complex expressions.

Let the function $true(l, r)$ be the number of ways true/false values can be (independently) assigned to the variables X in the expression, which begins at position l and ends at position r, so that the expression evaluates to true. Similarly with $false(l, r)$.

The solution of the whole problem will be: $true(0, |exp| - 1)$.

Notice that if we know the value of $true(l, r)$ and the number of characters X between l and r we can easily get $false(l, r)$ by the following formula:

$$false(l, r) = 2^{(number\ of\ variables\ X\ between\ l\ and\ r)} - true(l, r).$$

Precomputation:

We can use stack to find out for every complex expression where it begins and where it ends (bracket parsing). When the program iterates to the next '(' character, it's position is pushed to the stack. When the program finds ')' character, it pops one integer from the stack - the position of the begining of the expression which ends at the current possition.

In order to find out the number of variables X between any two characters we can use partial sums. Also, we need to maintain an array $p2$ of powers of two modulo $10^9 + 7$.

Now, when we precalculated partial sums, and found a pair for each bracket we can solve a problem using recursion.

Recursion description:

Function $true(int\ l, int\ r)$ returns 1 if the expression is basic, otherwise it finds the ',' character and positions l_1, r_1, l_2 and r_2 - begining and end of exp_1 and exp_2.

If the current expression is:

1. $NOT(exp)$ function returns: $false(l + 4, r - 1)$
2. $AND(exp1, exp2)$ function returns: $(true(l_1, r_1) * true(l_2, r_2)) \% (10^9 + 7)$
3. $OR(exp1, exp2)$ function returns: $(true(l_1, r_1) * true(l_2, r_2) + true(l_1, r_1) * false(l_2, r_2) + false(l_1, r_1) * true(l_2, r_2)) \% (10^9 + 7)$

Complexity analysis:

We can do precomputation in linear time, and the function itself requires $O(|exp|)$ time so the total complexity is $O(\sum |exp|)$.

Problem source: SPOJ

Solution by:
Name: Magdalina Jelić

Round 1: N..K..Divide

Statement:

Mona and her brother Alaa are very smart kids, they love math a lot and they are passionate about inventing new games.

So after they came back from school they invented a new game called "N..K..Divide".

First of all, let's define a function $D(m)$, such that $D(m)$ is the number of distinct primes in the prime factorization of m.

For Example $D(9) = 1$ and $D(12) = 2$.

The rules of the game are simple:

- The game consists of rounds.
- In each round there are two integer numbers N and K.
- Each round consists of multiple turns.
- Each player alternately takes a turn (starting with the player who won the last round / by Mona in the first round).
- In each turn the player chooses some positive integer M, where $2 \leq M \leq N$ such that N is divisible by M and $D(m) \leq K$, then divides N by the chosen M.
- The first player who cannot make any valid move loses.
- The player who wins more rounds is declared as the winner of the game (if it's a tie, then Mona is considered the winner).

So the kids asked their father to run the game for them.

For each of the R rounds their father gives them two integer numbers N, K and wants to know who will be the winner of this round if they both play optimally (to win the current round).

Input:

The first line consists of a single integer $1 \leq R \leq 10$ the number of rounds the kids are going to play.

Each of the next R lines contains two space-separated integers N, K where $2 \leq N \leq 10^{14}$, $1 \leq K \leq 10$.

Output:

Print $R + 2$ lines.

For the first R lines, the i^{th} line of them should contain the name of the winner of i^{th} round if both players play optimally "Mona" or "Alaa" (without quotation marks).

The line number $R + 1$ is an empty line.

In the last one print the name of the winner, print "Mona" if Mona is the winner of the game otherwise print "Alaa" (without quotation marks).

Example input:

```
5
3 4
6 2
6 1
9 1
18 2
```

Example output:

```
Mona
Mona
Alaa
Alaa
Alaa

Alaa
```

Time and memory limit: 1s / 64 MB

Solution:

Let's analyze a single round. Suppose that Mona plays first and let the numbers be N and K.

We will try to translate this problem to some variation of the famous game Nim, since our game is impartial and therefore we know that it is equivalent to Nim, as stated by Sprague Grundy theorem.

Let's factor our number N. Each prime divisor of N will represent a heap and the number of objects in this heap (size of heap) will be exponent of prime divisor in factorization of N.

Notice that now our game translates to the following:

We have $D(N)$ heaps and in each move the player is allowed to take objects from at most K heaps (as opposed to original Nim, where the player is allowed to take objects from exactly one heap).

Fortunately, the solution for this is just a slight extension for the solution of a regular Nim (where we take XOR-sum of sizes all piles and if it is positive the first player wins, otherwise, if it equals to 0, then the second player wins).

In this extension, as well as in the solution for the regular Nim, we consider the binary representation of sizes of heaps, only now we don't consider XOR-sum of sizes (which is actually sum modulo 2) but we consider the sum of each binary digit modulo $K + 1$.

So, the conclusion is that we write binary expansion of each size of heap, and then digit by digit we sum the modulo $K + 1$ and if the number we got is positive then Mona wins, otherwise Alaa wins (This is called Moore's variation of Nim).

Now that we know who wins in each round, we just remember the number of wins for each player and in the end we output the final winner.

Problem source: SPOJ
Solution by:
Name: Uroš Maleš

Round 1: Rational Numbers

Statement:

In mathematics, a rational number is any number that can be expressed as the quotient or fraction $\frac{p}{q}$ of two integers, with the denominator q not equal to zero. Since q may be equal to 1, every integer is a rational number. Furthermore a set S is called countable if there exists an injective function f from S to the natural numbers $N^* = \{1, 2, 3, \dots\}$.

The set of all real numbers is not countable, but the set of all rational numbers is infinite and countable. In order to prove that let's put all rational numbers in a table with the following rules: in the first row are placed all integer numbers ordered by their absolute value, and placed after each natural number (including zero) are its oposite:

$0, 1, -1, 2, -2, \dots, n, -n, \dots,$

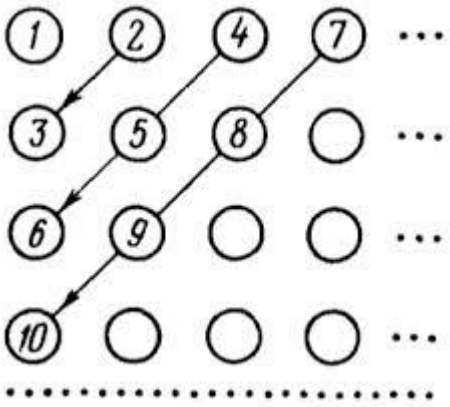
In the second row all irreducible fractions whose denominator is 2 are placed, ordered by their absolute value and again after each positive number its oposite follows:

$\frac{1}{2}, -\frac{1}{2}, \frac{3}{2}, -\frac{3}{2}, \frac{5}{2}, -\frac{5}{2}, \dots$

In general, in the n^{th} row all irreducible fractions with the denominator n are placed, ordered by their absolute value and after each positive number its oposite follows. The rational number table that has been obtained possess an infinite number of rows and columns:

$$\begin{array}{cccccc}
0 & 1 & -1 & 2 & -2 & \dots \\
\frac{1}{2} & -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{5}{2} & \dots \\
\frac{1}{3} & -\frac{1}{3} & \frac{2}{3} & -\frac{2}{3} & \frac{4}{3} & \dots \\
\dots\dots & \dots\dots & \dots\dots & \dots\dots & \dots\dots & \dots\dots \\
\frac{1}{n} & -\frac{1}{n} & \dots\dots & \dots\dots & \dots\dots & \dots\dots \\
\dots\dots & \dots\dots & \dots\dots & \dots\dots & \dots\dots & \dots\dots
\end{array}$$

Clearly every rational number is placed in the table. Let's enumerate the elements of the table according to the following schematic (arrows indicate the direction of the increasing numeration):



As a result, every rational number receives a unique natural number. Can you tell which rational number occupies the i^{th} position in the table?

Input:

There is a single integer T in the first line of input ($1 \leq T \leq 100$). It stands for the number of test cases to follow. Each test case consists of a single line, which contains an integer number i ($1 \leq i \leq 10^8$).

Output:

For each test case print a single line with the i^{th} rational number r_i. For r_i being an integer number print r_i in the usual single integer representation, otherwise (r_i is not an integer number) r_i must be printed in the format $\frac{p}{q}$ as an irreducible fraction (Greatest Common Divisor of p and q is 1 and $q > 0$). Check the example output for clarification.

Example input:

```
4
1
2
3
9
```

Example output:

```
0
1
1/2
-1/3
```

Time and memory limit: 1s / 64 MB

Solution:

First, let's find the position of the fraction in the table, in other words, let's find the row (which is also the denominator of the fraction) and the column of the cell in which our fraction is. In the 1^{st} diagonal there is 1 fraction, in the 2^{nd} there are 2 fractions … in k^{th} fraction there are k diagonals. So, in first k diagonals there are $\frac{k(k+1)}{2}$ fractions. Therefore, we can find in which diagonal our fraction is by finding the biggest number k such that $\frac{k(k+1)}{2} < p$ where p is the number from input. Then, the number of the diagonal that we are looking for is $D = k + 1$.

Once we have the number of the diagonal we can also find the number of the row in which our fraction is positioned and let's call it R. R is easily calculated by formula: $R = p - \frac{k*(k+1)}{2}$.

Now, we can also find the number of the column in which our fraction is and let's call it C. C it can be easily calculated too. Note that $r + c = D + 1$ holds for every cell of the diagonal, so $C = D + 1 - R$.

Once we have the column of the cell, we can find the numerator of the fraction. We can do that by finding $\left(\frac{C}{2}\right)^{th}$ coprime number to R if C is even (if C is even, we have to put a negative sign too) or $\left(\frac{C+1}{2}\right)^{th}$ coprime number to R if C is odd. We can do that by simply iterating over numbers from 1 to ∞ and incrementing the counter when the greatest common divisor of the iterator and R is 1.

There are some corner cases when the number from input is 1 (we should output '0') and when the denominator is 1 (we should output the numerator only and put a negative sign if C is odd because everything in the first row is shifted by 1 position because of '0').

Problem source: COJ
Solution by:
Name: Petar Vasiljević

Round 1: Toby and the Frog

Statement:
Toby the dog is on the cell 0 of a numbered road, TJ the frog is on the cell number X of the same road. Toby wants to catch the frog, but he is not smart enough to count the distance from his current position to X, so he performs the following algorithm:

- Let pos be Toby's current position.
- Jump an integer distance d, where d is uniformly distributed over $[1, min\,(X - pos, 10)]$.
- If the new position is the frog's position, catch it and send it as tribute to the queen.
- In other case start the algorithm again.

Note that the length of Toby's jump cannot be infinite, in fact, it must be less than or equal to 10. Besides this, he will never jump over the frog, in other words, he will never reach a position greater than X. TJ the frog does not want to be caught, due to this, TJ wants to compute the expected number of jumps that Toby needs in order to reach cell number X.

Help TJ to compute this value.

Input:
The input starts with an integer $1 < T \leq 100$ indicating the number of test cases.

Each test case contains one integer $10 \leq X \leq 5000$ denoting the frog's cell.

Output:
For each test case print in one line the expected number of jumps that Toby needs to reach cell number X.

Answers with absolute error less than 10^{-6} will be considered correct.

Example input:
```
2
10
20
```

Example output:
```
2.9289682540
4.8740191199
```

Time and memory limit: 1s / 64 MB

Solution:

First of all, computing a mathematical solution for this task is the easier part of this task. Let us define X_i as the probability of Toby reaching the frog in i moves. After that, the solution for this task is:

$$result = \sum_{i=1}^{x} i * Xi$$

where x is the position of the frog.

Calculating the probability for every scenario would result in a lot of time consumption and probably wouldn't pass in the given time limit. Now the same thinking as for calculating the n^{th} Fibonacci number leads us to a dynamic programing solution.

Let $dp[i] = probability\ of\ getting\ to\ position\ i$. Now $dp[0] = 1.0$ because Toby will always be able to get to that position. Now the recurrence formula would look like this:

$$dp[i] = \sum_{j=1}^{\max(0,i-10)} \frac{dp[i-j]}{\min(10, x-(i-j))}$$

So now we got the $dp[]$ matrix ready and since we need to calculate the expected number of jumps it will be equal to the sum of all the probabilities in the $dp[]$ matrix (except for $dp[0]$ because that is not a jump probability). This is because the probability of getting to some position is equal to the probability of getting one more jump to that position.

When it comes to memory and time limits, they are much lower than the task wanted them to be, because time complexity is $O(T * 10 * x)$ and memory complexity is $T(x)$.

Problem source: COJ
Solution by:
Name: Mihailo Milošević

Round 1: Walk home together

Statement:

A group of f friends gets together to have a coding problem-solving party at their university computer lab every weekend. Unfortunately, it closes at 22:00, at which point they have no other option but to walk home. The city they live in can be described as n junctions connected by m bidirectional roads. Since these friends have been interrupted from coding their solutions, each one of them wants to get home as fast as possible to finish his and submit it. At the same time, they each have a few problems that they want to discuss with the others, so they will pick such a path that the entire group can walk together for as long as possible. On top of that, so that this walk would feel fresh every time, they would like to take a different path each weekend, for as long as possible.

The group of friends were able to figure out both the length of the longest path they could all efficiently walk together, as well as how many such distinct paths there are - do you think you can do it, too?

Input:

The first line contains an integer T - the number of test cases. T test cases follow, each in the given format. Test cases are separated by a blank line.

The first line of a case contains four integers $n\ m\ c\ f$ - the number of junctions, the number of two-way roads connecting the junctions, the junction at which the computer lab is located, and the number of friends. Junctions are numbered from 1 to n.

The second line contains f distinct integers $F_1, \ldots, F_f$ $-$ F_i is the junction where friend number i lives. None of them are equal to c.

The following m lines contain three integers $x\ y\ z$, denoting a two-way road connecting the junctions x and y, of length z. Each unordered pair x, y will be present at most once.

You can assume that the city is connected.

Output:

Output two integers L and W.

A 'path' is a sequence of junctions $a_1 \ldots a_k$ where for all $1 \leq i < k$ the junctions a_i and a_{i+1} are connected by a road.

L is the length of the longest path, such that for all $1 \leq i \leq f$ if after walking this path friend number i can get home the soonest at some time $L + t_i$, there exists no path from c to F_i shorter than $L + t_i$. In other words, none of the friends could have gotten home sooner even if they had chosen a different path which did not include the one of length L.

W is the number of such paths, modulo $10^9 + 7$.

Two paths are considered distinct if either

- they contain a different number of junctions, or
- they contain the same number of junctions (let's call that k), where one path is described by $a_1 \dots a_k$ and the other by $b_1 \dots b_k$, and there exists some $1 \leq i \leq k$ such that $a_i \neq b_i$.

Constraints:

- $1 \leq f < n \leq 5000$
- $n - 1 \leq m \leq min\,(10^6, \frac{n*(n-1)}{2})$
- $1 \leq c \leq n$
- $1 \leq F_i \leq n$
- $1 \leq x < y \leq n$
- $1 \leq z \leq 4 * 10^5$
- $1 \leq T \leq 500$

Additionally, if $T > 1$, $n \leq 50$.

The largest input file is under 16MB.

Example input:

```
3
9 12 2 3
9 8 1
2 5 3
2 3 7
2 4 5
3 6 4
6 7 2
5 7 20
4 7 8
4 8 50
7 9 10
7 8 30
1 8 15
1 9 15

9 12 2 3
9 8 1
2 5 3
2 3 7
2 4 5
3 6 4
6 7 2
5 7 20
4 7 8
4 8 10
7 9 10
7 8 30
1 8 15
1 9 15
```

```
9 12 2 3
9 8 1
2 5 3
2 3 7
2 4 5
3 6 4
6 7 2
5 7 20
4 7 8
4 8 10
7 9 10
7 8 30
1 8 4
1 9 4
```

Example output:

```
13 2
5 1
15 1
```

Explanation:

In the first case, the paths are $2, 3, 6, 7$ and $2, 4, 7$. Friends number 1 and 2 can then go straight home, and friend number 3 can go home either through junction 8 or 9. For each friend, this is an optimal path.

In the second case, friend number 1 can get home the fastest the same ways as before, friend number 2 can get home the fastest by going down $2, 4, 8$ and friend number 3 by going down $2, 4, 8, 1$. Hence the longest path which all friends are willing to take is $2, 4$ of length 5, and no other such path exists.

In the third case, the first friend can get home the fastest by paths $2, 3, 6, 7, 9$ or $2, 4, 7, 9$ or $2, 4, 8, 1, 9$; the second friend's only fastest path is $2, 4, 8$, and the third friend is only willing to take the path $2, 4, 8, 1$. Hence the longest path they are all willing to take has length 15 and it is $2, 4, 8$.

Time and memory limit: 4s / 64 MB

Solution:

Firstly, we make a DAG using Dijkstra's algorithm which will contain all vertices and edges needed for any path of shortest length to any node. This graph will be useful because in this problem we are looking for a path which we can extend with a few extra edges to be the shortest path to any "home" node, which itself is the shortest path too. To construct this graph we simply add every edge connecting node A to node B (order matters, we are making a DAG) such that $dist(A) + length(edge) = dist(B)$, where $dist()$ represents the length of the shortest path from the source node to a node we obtained using Dijkstra's algorithm. It is easy to prove that every path of the shortest length from the source node to each other node can be made using edges from this graph, using contradiction. Also, every path in this graph is the shortest path which is apparent since we used Dijkstra distances. Now that we have this graph we will sort it topologically, or since we already have the data from Dijkstra's algorithm, we can simply sort it using the order in which we "confirmed" the nodes while running Dijkstra (these two methods yield the same sorting). We would also get the same ordering if we sorted by the shortest distance from the source. Now we will move from the back of the array to the front (from the node the furthest away from the source to the closest one) and look for the node the furthest from the source and from which we can get to each of the "home" nodes using only edges in the graph we constructed. Since this sorting is also a topological sorting, all nodes to which we can get to when starting from node A are closer to the end of the array than node A. For each node we will have a bitmask representing to which of the "home" nodes we can get to. So, while moving from the back of the array we will update bitmasks of all of the parental nodes (i.e. nodes from which we can get to that node using only one edge) by using bitwise AND. The first node we get to (remember we started from the end) such that the bitmask contains 1 in all positions will be one of (there could be multiple such nodes) the furthest nodes from the source from which we can get to each "home" node. Now we just check if any of the other nodes of the same $dist()$ have a good bitmask. We now have the L value of the output ($dist()$ to one of these nodes), we still need to calculate W, which in a DAG can be done using simple combinatorics (the only thing we need to keep in mind is that we could have multiple nodes to which we are calculating the number of ways to, and just add them at the end). Let's discuss the time complexity of this solution. The time complexity of Dijkstra's algorithm is $O(m * \log n)$ which is good enough. Construction of the DAG and sorting of nodes are also fine. The only part where we need to be careful is traversing of the array while updating bitmasks. The bitmask could have up to n bits and we do a bitwise AND for each edge in the DAG which we could have up to m. So in order to achieve a good time complexity we need to use a C++ bitset (or a smart custom bitmask using an array of integers) which would reduce the time complexity of this part down to $O(\frac{n*m}{32})$ which is good enough.

Problem source: SPOJ
Solution by:
Name: Marko Šišović

Round 1: [Challenge] Guess The Number With Lies v5

Statement:

The Judge has chosen an integer x in the range $[1, n]$. Your task is to find x, using at most m queries as described below. But be careful because there are 2 important things:

1. The Judge can lie. The Judge is allowed to lie w times in a single game and only in reply for query.
2. You must prepare all queries at once and send all of them to the Judge before you get any reply.

Query:

A single query should contain a string $s_1 s_2 s_3 \dots s_n$, where s_i is '0' or '1'. The reply for the query is "YES", if s_x = '1', or "NO" otherwise.

For example, when $n = 8$, $x = 5$ and the query is "00101011", the Judge will reply "YES" (if telling the truth) or "NO" (if lying), because the 5^{th} character in the query is '1'.

Real task and scoring:

The real task is not to find the number x chosen by the Judge, but only to prepare queries, which allow you to guess the x value for every possible Judge's reply. The Judge won't give you a reply for your queries.

If the Judge will find such a reply for your queries, that there will be more than one x value possible, you will fail the test case (see example for detail explanation). Otherwise you succeed.

If you succeed, your score is q^2, where q is the number of queries, that you use. If you fail, you get a penalty score equal to $4 * m^2$. The total score is $\lfloor \frac{the\ sum\ of\ scores\ of\ single\ games}{100} \rfloor$. The smaller score is the better score.

Input:

The first line of input contains one integer T - the number of test cases. Then T test cases follow.

Each test case contains one line with three single-space separated integers n, w and m, where n is the size of the game, w is the maximal number of lies and m is the maximal number of queries, that You can use.

Output:

For each test case print a line with one integer q - the number of queries that you want to ask.

Then print q lines with your queries. Each query should be string of length n, with all characters '0' or '1'.

Constraints:

- $1 \leq T \leq 2^7$

- $2 \leq w \leq 2^4$
- $2 \leq n \leq 2^{12}$
- $(2 * w + 1) * \lceil \log_2 n \rceil \leq m$
- $0 \leq q \leq m$

Example input:

```
4
2 2 6
2 3 8
2 16 34
5 2 18
```

Example output:

```
6
01
01
01
01
01
01
6
01
01
01
01
01
01
0
15
00011
00011
00011
00011
00011
01010
01010
01010
01010
01010
00100
00101
00110
01100
01111
```

Explanation:

Notation: reply "YYN..." means, that the Judge replied "YES" for the first and the second query, "NO" for the third query, and so on...

In the 1st test case there are only two numbers. The Judge can reply "YYYYYY", "NNNNNN" (without a lie), "YYYYYN", "YYYYNY", "YYYNYY", "YYNYYY", "YNYYYY", "NYYYYY", "NNNNNY", "NNNNYN", "NNNYNN", "NNYNNN", "NYNNNN", "YNNNNN", (with one lie) or "NNNNYY", "NNNYNY", "NNNYYN", "NNYNNY", "NNYNYN", "NNYYNN", "NYNNNY", "NYNNYN", "NYNYNN", "NYYNNN", "YNNNNY", "YNNNYN", "YNNYNN", "YNYNNN", "YYNNNN", "NNYYYY", "NYNYYY", "NYYNYY", "NYYYNY", "NYYYYN", "YNNYYY", "YNYNYY", "YNYYNY",

"YNYYYN", "YYNNYY", "YYNYNY", "YYNYYN", "YYYNNY", "YYYNYN", "YYYYNN" (with 2 lies). The other replies ("NNNYYY", "NNYNYY", "NNYYNY", "NNYYYN", "NYNNYY", "NYNYNY", "NYNYYN", "NYYNNY", "NYYNYN", "NYYYNN", "YNNNYY", "YNNYNY", "YNNYYN", "YNYNNY", "YNYNYN", "YNYYNN", "YYNNNY", "YYNNYN", "YYNYNN", "YYYNNN") are not ok, because for every one of them and for every possible x value, the Judge lies more than two times. For each good reply, there is only one integer from range $[1, 2]$, that matches this reply ("match" means match all except at most two of the queries). The score is $6^2 = 36$. Notice, that You can use the same query more than once.

In the 2^{nd} test case the player tried to give the solution, but the solution was wrong. The Judge can reply for example "YYYNNN" and then for both possible values of x the judge lied 3 times. The player needs more queries in this case. The score is the penalty score $4 * 8^2 = 256$.

In the 3^{rd} test case the player decided to skip. The player got the penalty score $4 * 34^2 = 4624$.

In the 4^{th} test case the nubmer of queries given by the player is smaller than the maximal possible number. For every possible Judge's reply there is only one possible value of x, so the test succeeded. The score is $15^2 = 225$.

The total score is $\lfloor \frac{36 + 256 + 4624 + 225}{100} \rfloor = 51$.

Time and memory limit: 1s / 64 MB

Solution:

In "Guess The Number With Lies v5", we imagine a scenario when the judge has chosen a number in $\{1, \dots, n\}$, and we have to guess it, taking into account the fact that the judge can lie some number of times.

The questions we're allowed to ask consist of selecting any subset of $\{1, \dots, n\}$; in response, we learn if our subset contains the secret number. The judge can lie at most w times, and we have to design a small set of queries, which allows us to guess the secret number no matter what it is and which questions the judge decides to lie to. Note that we have to select the queries beforehand, so we cannot depend on the answers to the initial queries when choosing the subsequent ones.

To slightly transform the problem, consider encoding a strategy as a binary matrix M of size $q \times n$. Each of the q rows will correspond to a single query, containing n bits denoting which of the numbers from $\{1, \dots, n\}$ are present in the query.

Notice that if the secret number is k, then the true answers to our queries correspond to the k^{th} column of M. As the judge can lie at most w times, it can respond with a different sequence of answers – more precisely, any sequence inside a Hamming ball of radius w around the k^{th} column. If two different columns of M have Hamming distance at most $2w$, then it's easy to see that there exists a binary sequence b with distance at most w to both columns. If the judge responds to our queries according to the sequence b, we see that we won't find out the exact value of the secret number; hence the strategy encoded by M is incorrect. Conversely, if any two columns of M have Hamming distance at least $2w + 1$, any sequence of answers can match at most one choice of the hidden number, so M gives a valid strategy.

The analysis above gives us a different look at the problem: we need to find a matrix M of size $q \times n$, such that every two columns of M have Hamming distance at least $2w + 1$, and q is as small as possible.

We can start with M being empty, and add rows one by one, as long as there are columns at distance less than $2w + 1$.

The easiest solution is to iterate for i from 0 to $ceil(\log n)$, and for each such i, create a row with ones for numbers that have the i^{th} bit set and zeros for the other ones. It's easy to see that if we duplicate each such row $2w + 1$ times, we get a valid matrix M. This gives us a solution with $q = (2w + 1) * ceil(\log n)$. Note the constraint on m, which means that for our simple solution we're already asking at most m queries, and we don't even need to read the value of m from the input. If we want to explore a more general class of solutions, it becomes non-trivial to know whether we should continue adding rows to M or not. However, if we keep all pairwise distances between columns, and update them after each row is added, it's easy to know when to stop.

One possible class of sequences which we can consider as rows for M is the following. Take any number $x \leq 2^{(ceil(\log n))}$, and consider an assignment from $\{1, \ldots, n\}$ to $\{0, 1\}$ defined as:

$$f(i) = NumberOfOnes(i\ AND\ x)\ mod\ 2$$

Note that the class of sequences above also contains the sequences we used as rows in our initial solution. On the other hand, this class is not very big as it has only $O(n)$ sequences.

In our next solution, we will extend M, each time by greedily picking an optimal row that can be constructed according to the process above. We can score a candidate row in many different ways; for example, we can consider the minimum Hamming distance between a pair of columns at the moment, take all such "worst" pairs of columns, and count for how many of them we're increasing the distance by adding the candidate row.

The above idea gives a much better solution, but it's pretty slow. The next idea is to calculate the solutions locally and hardcode them in the code sent to the judge system.

To fit within the code length limit, we need to resolve a few issues. First, it's not possible to hardcode the solutions themselves, as each question corresponds to sequence of n bits. Fortunately, our questions come from a narrow family of sequences which can be described with just one number x less than $2n$. Thus, the solution to a particular input (n, w) takes not that much space, but there are many possible values of n. Now, we can notice that an answer for some n gives also an answer for all smaller n. If we precompute the answers for n being the powers of two (and all possible values of w), then given some arbitrary n we can round it up to the closest possible power of n. Finally, even after these reductions, the resulting code can exceed the length limit. In that case, we can further compress the solutions before hardcoding them, while adding an additional code that decompresses them at runtime.

This solution gives already pretty good results. In order to squeeze a few more points out of it, we can notice that the precomputation time is not bounded by the judge's time limit. If we randomly perturb our greedy construction process so it's not deterministic, we can try many different random seeds (or different variants of the algorithm itself, e.g. with a different greedy selection rule) and hardcode the best one we find. Checking these different seeds/variants is trivially parallelizable, so we can scale it to multiple threads, and even multiple machines. Obviously, using more compute power quickly starts to yield diminishing returns, but it allows to obtain a few more points into the final score. This solution now places within the top 3 places on the BubbleCup leaderboard.

Problem source: SPOJ
Solution by:
Name: Krzysztof Maziarz

Round 2: Ada and Scarecrow

Statement:

As you might already know, Ada the Ladybug is a farmer. She has multiple farmhouses and some fields which always connect two adjacent farmhouses. She plans to buy some scarecrows to scare crows!

A scarecrow placed on a farmhouse scares all crows from all adjacent fields. A crow on a field can be disastrous, so Ada has to arrange the scarecrows in such a way that they cover all of the fields. As scarecrows are pretty expensive, she wants to minimize the number of them. Can you count it for her?

Note: Even thought it might look like that from description, the formed "graph" doesn't have to be planar! Also, multi-fields and self-field are not allowed.

Input:

The first line contains an integer $1 \leq T \leq 100$.

Each of the next T test-cases begins with two integers $1 \leq N, M \leq 150$, the number of farmhouses and the number of fields.

Each of next M lines contains two numbers $0 \leq A_i, A_j < N$, the farmhouses, which are connected by a field.

Output:

For each test-case output the minimal number of scarecrows Ada has to buy, to cover all fields.

Example input:

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

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

Example output:

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

Example input:

```
2
17 7
12 9
11 8
9 16
11 2
5 14
7 6
8 14
16 23
10 13
11 3
7 14
14 3
9 11
7 8
0 9
8 14
10 8
0 4
6 15
6 12
2 5
10 2
7 11
13 12
15 13
5 3
15 0
6 2
12 9
5 1
1 4
```

Example output:

```
4
```

9

Time and memory limit: 1s / 64 MB

Solution:

This problem is a well-known NP-complete graph theory problem – Minimum Vertex Cover. A good strategy to solve such problems is to use a Branch and Bound approach along with a good heuristic.

A state is defined as a mapping of the set of nodes of the graph to the set $\{0, 1, X\}$, 0 denoting that a node is not taken into the vertex cover, 1 denoting that it is taken into the vertex cover, and X denoting that this node may or may not be included into the vertex cover. A state s is valid if no edge of the graph has both its endpoints marked with 0. A state is final if it does not map any node to X.

Let's start with a pure brute-force recursive solution and then improve it by eliminating unnecessary recursive calls. We will use a recursive function whose parameter is a valid state. If the state is final, we count the number of ones and store that number. Otherwise, we find any node p such that $s(p) = X$ and then recursively call the function twice, once for a state such that this node has been mapped to 0 instead and once where it has been mapped to 1.

Here, we can make the first improvement. When we replace $s(p)$ with 0, we can also immediately change $s(q)$ to 1 for all neighbors q of p. If we do this each time we assign 0 to an undecided node, it's not hard to see that we will always end up with a valid state.

If the number of nodes mapped to 1 already exceeds the minimum solution found so far, we can return immediately. We will improve this by finding a lower bound on the number of undecided nodes we need to map to 1 in order to complete a vertex cover. Let M be a matching of the graph considering only the edges whose both endpoints are mapped to X. Then, it's not hard to see that we need to map at least $|M|$ more nodes to 1. Thus, we want to find a large matching of that graph – the bigger, the better. Unfortunately, finding the exact size of the maximum matching is too slow. Instead, we can find any matching using a greedy algorithm. This approach results in lower overall running time.

The order of recursion matters. When we pick a node on which to recurse, we will pick the one having the most neighbors not mapped to 1. Also, first we will recurse setting $s(p) = 1$ and then setting $s(p) = 0$.

Another improvement is to process the state by greedily assigning values to certain nodes which are currently mapped to X in such a way that we don't worsen the solution. If a node has no neighbors mapped to X, we can freely map that node to 0. Otherwise, if it has exactly one neighbor mapped to X, then, for this edge, at least one of the nodes has to be set to 1, so let's set this node to 0 and its neighbor to 1.

Finally, the following trick seems to slightly improve running time. After reading the graph from input, we can permute the labels of nodes and permute their adjacency lists. Then, we

will randomly pick a node as a root, run BFS from it, and reorder the nodes in based on their distance from the root, again permuting their adjacency lists.

Problem source: SPOJ
Solution by:
Name: Ivan Stošić

Round 2: Ada and Prime

Statement:

As you might already know, Ada the Ladybug is a farmer. She grows many vegetables and trees and she wants to distinguish between them. For this purpose she bought funny signs, which contain a few digits. The digits on the sign could be arbitrarily permuted (yet not added/removed). Ada likes prime numbers so she wants the signs to be prime (and obviously distinct). Can you find the number of signs with prime number which could be obtained?

NOTE: The number can't have leading zero!

Input:

The first line of input will contain $1 \leq T \leq 10000$, the number of test-cases.

The next T lines will contain $1 \leq D \leq 9$, the number of digits on a sign, followed by D digits $0 \leq d_i \leq 9$.

Output:

For each test-case, output the number of distinct primes which could be generated on a given sign.

Example input:

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

Example output:

```
0
0
11
283
15
```

Time and memory limit: 4s / 64 MB

Solution:

Let's present an offline solution. First, we precompute the answer for every possible input set and save that in an array ans, such that $ans[to\_key(input)]$ (initialized to 0) represents the number of possible primes that can be generated using the digits from the input set. The to_key function is used to map the input digits to an integer number that can be used as an index for the ans array. Since the input digits are treated as a set, the solution remains the same when they are permuted, and to_key should map all those permutations to the same integer key. One possible choice is to map every set of digits $\{d_1, d_2, \dots, d_n\}$ to $10^{d_1} + 10^{d_2} + \cdots 10^{d_n}$.

The most naive way to precompute the solutions for all inputs is to iterate through all the numbers in $[1, 10^9]$ (this is enough since $D \leq 9$) and for every number check if it's a prime. For every prime found we treat its digits as a set and do $sol[to\_key(prime)]{+}{+}$. By applying to_key to both primes and input sets we avoid needless computation and duplication of results.

However, this solution is too slow. The main optimization we have to introduce is to use the Sieve of Eratosthenes to find all primes in $[1, 10^9]$ instead of doing it manually. On top of that, we have to use the segmented variant with clever usage of bitmasks to save on time even further. The sieve and the segmented variant are well known and their description can be easily found in many places, so we will omit it in this editorial. One of those places, that also hosts a heavily optimized C++ implementation of segmented bit sieve is https://github.com/kimwalisch/primesieve/wiki/Segmented-sieve-of-Eratosthenes. Working with bitmasks can be tricky when we need to extract actual values of primes and map them to keys, but apart from that, the segmented bit sieve fits our use case perfectly.

This, along with the special handling of some edge cases, is good enough to pass on SPOJ. For the BubbleCup judge, one final optimization was needed: Instead of sieving all primes in $[1, 10^9]$ we use $[1, 10^{maxD}]$ where $maxD$ is the largest value of D in the current test case.

Problem source: SPOJ
Solution by:
Name: Nikola Jovanović

Round 2: Just a Palindrome

Statement

A palindrome is a symmetrical string, that is, a string read identically from left to right as well as from right to left.

Chiaki has a string s and she can perform the following operation at most once:

- choose two integers i and j ($1 \leq i, j \leq |s|$),
- swap s_i and s_j.

Chiaki would like to know the longest palindromic substring of string after the operation.

Input

There are multiple test cases. The first line of input contains an integer T, indicating the number of test cases. For each test case the first line contains a non-empty string s ($1 \leq |s| \leq 10^6$) consisting of lowercase and uppercase letters.

It is guaranteed that the sum of all $|s|$ does not exceed 10^6.

Output

For each test case, output an integer denoting the answer.

Example input

```
10
a
xxxx
ssfs
aaabbacaa
missimxx
abababababgg
dfsfsdgdg
asdsasdswe
chiaki
teretwer
```

Example output

```
1
4
3
8
6
9
6
9
3
6
```

Time and memory limit: 1s / 256 MB

Solution

Let the given string be S and its length be n. We refer to the substring $S_l S_{l+1} \dots S_r$ as $S_{l,r}$ for $1 \leq l \leq r \leq n$. To represent the reversed string A we use the notation A^R.

Let's traverse S from left to right and find the longest palindrome centered at each position. We will first consider the palindromes of even length.

For a fixed index i $(1 \leq i < n)$, we assume that the center of the palindrome is situated between positions i and $i+1$. We can use binary search to find the minimum number k such that $S_{i-k+1,i} \neq S^R_{i+1,i+k}$ or $i-k+1 < 1$ or $i+k > n$. If the mentioned indexes are out of bounds, we can just update our answer with $2(k-1)$ and continue. Otherwise, there has happened a mismatch, (that is, $S_{i-k+1} \neq S_{i+k}$) which we can fix using one swap if the character S_{i-k+1} appears in range $[1, i-k] \cup [i+k, n]$ or if S_{i+k} appears in range $[1, i-k+1] \cup (i+k, n]$.

Suppose we somehow managed to fix the first mismatch. In that case, we might be able to extend our palindrome further beyond it. Thus, we binary search the next mismatch, i.e., the minimum number l greater than k such that $S_{i-l+1,i-k} \neq S^R_{i+k+1,i+l}$ or $i-l+1 < 1$ or $i+l > n$. If there is no second mismatch, we can just update our answer with $2(l-1)$. Otherwise, we have found the second mismatch: $S_{i-l+1} \neq S_{i+l}$. We can fix both of them using one swap if and only if the logical statement A is true:

$$A \equiv (S_{i-l+1} = S_{i-k+1} \wedge S_{i+k} = S_{i+l}) \vee (S_{i-l+1} = S_{i+k} \wedge S_{i-k+1} = S_{i+l})$$

If the first operand of disjunction is true, we can perform a swap on S_{i-l+1} and S_{i+k}. If the second operand is true, we can perform a swap on S_{i-l+1} and S_{i-k+1}.

If we were able to fix both of the mismatches using one swap, then we should try and lengthen our palindrome even more with yet another binary search. Finally, we update our answer with $2(m-1)$, where m is the minimum number greater than l such that $S_{i-m+1,i-l} \neq S^R_{i+l+1,i+m}$ or $i-m+1 < 1$ or $i+m > n$ ($S_{i-m+1} \neq S_{i+m}$ is the third mismatch).

When dealing with the palindromes of odd length, we assume that their center is exactly the fixed index i $(1 \leq i \leq n)$ and use the largely unchanged approach for even length palindromes. The only difference is that an additional way to fix the first mismatch is to swap S_{i-k} with S_i if $S_{i+k} = S_i$ or swap S_{i+k} with S_i if $S_{i-k} = S_i$ (the indexes are adapted to suit the odd length palindromes).

The implementation can be carried out efficiently using rolling hash (p is a chosen constant and $t_1, \dots, t_k$ are characters):

$$H = t_1 p^{k-1} + t_2 p^{k-2} + t_3 p^{k-3} + \cdots + t_k p^0$$

Having precomputed H_i and H_i^R (the value of hash of $S_{1,i}$ and $S_{i,n}^R$ respectively) we can obtain the value of hash of a certain substring of S and therefore check substrings for equality in constant time:

$$H_{l,r} = H_r - H_{l-1} \times p^{r-l+1}$$

$$H_{l,r}^R = H_l^R - H_{r+1}^R \times p^{r-l+1}$$

We also need to be able to check if a certain character f appears in a given range $[l, r]$, which can be accomplished by precomputing $C_{i,j}$, the number of times character j appears in $S_{1,i}$.

On the whole, we get a solution that works in $O(n \times \log n)$, with a constant factor depending on the tidiness of one's implementation.

Problem source: SPOJ
Solution by:
Name: Miloš Purić

Round 2: Tjandra 19th birthday present (HARD)

Statement:
This game/puzzle is about matches, given N matches, your task is to arrange the matches (not necessarily all) such that the number of rectangles (any size) is maximum.

Input:
The input begins with the number T of test cases in a single line.

In each of the next T lines there is one integer N.

Output:
For each test case, on a single line, print the required answer (maximum number of rectangles).

Constraints:
- $1 < T \leq 100$
- $1 < N \leq 10^{18}$

The T numbers N are uniform-randomly chosen in the range.

Example input:
```
6
3
4
8
12
15
987654321123456789
```

Example output:
```
0
1
3
9
12
609663161271147689131481595715032 06
```

Explanation:
First test case:

No rectangle can be formed with only 3 matches.

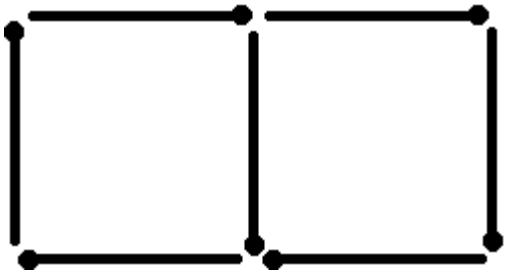
Second test case:

Only one rectangle can be formed with 4 matches.

Third test case:

There are max 3 rectangles.

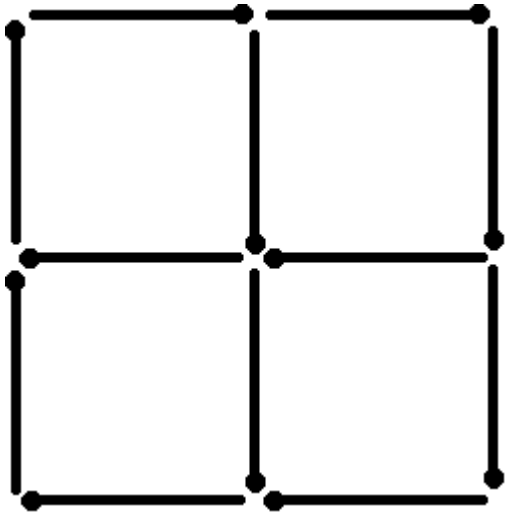
(2 size 1×1, 1 size 2×1) can be formed with a number of matches ≤ 8, here is one of the matches formations:



Fourth test case:

There are max 9 rectangles.

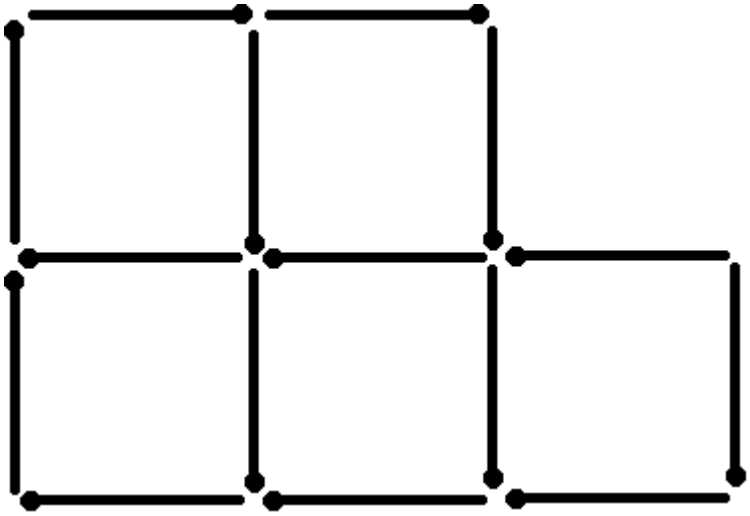
(4 size 1×1, 2 size 2×1, 2 size 1×2, 1 size 2×2) can be formed with a number of matches ≤ 12, here is one of the formations:



Fifth test case:

there are max 12 rectangles.

(5 size 1×1, 3 size 2×1, 1 size 3×1, 2 size 1×2 , 1 size 2×2) can be formed with a number of matches ≤ 15, here is one of the formations:



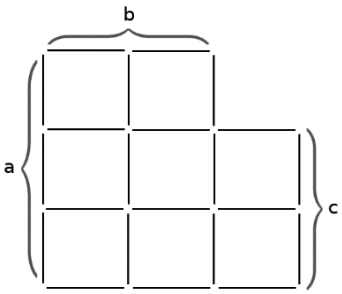
Sixth test case:

You have to figure out by yourself how to compute that in the required time.

Time and memory limit: 0.5s / 64 MB

Solution:

The key observation is that for any number of matches, the optimal solution has a form of grid-aligned rectangle, possibly with some additional unit cells on one side. This lemma is rather intuitive and its proof is technical. Therefore, now I will show how to solve the problem assuming the lemma and then I will prove the lemma itself. Henceforth I will call such structures extended rectangles. Note that this observation suffices to solve the problem for small values of n, since there is only $O(\sqrt{n})$ possible *extended rectangles*.



Consider any extended rectangle. Denote its dimensions by a, b and the number of additional cells as c. Without loss of generality we can assume $a \leq b$, since it is always better to put additional cells along the shorter side. Then clearly the number of simple rectangles in it is equal to

$$\binom{a+1}{2}\binom{b+1}{2}+\binom{c+1}{2}(b+1)$$

Since we have only n matches, we know that

$$n \geq a(b+1)+b(a+1)$$

which comes from the number of matches in the central rectangle. This gives us

$$\frac{n-b}{2b+1} \geq a$$

And therefore

$$a = \left\lfloor \frac{n-b}{2b+1} \right\rfloor$$

Also, the remaining matches give us

$$c = \left\lfloor \frac{n-1-a(b+1)-b(a+1)}{2} \right\rfloor$$

in case there are any, otherwise $c = 0$. Now let's observe that the following inequality holds:

$$\binom{a+1}{2}\binom{b+1}{2} + \binom{c+1}{2}(b+1) =$$

$$\binom{\left\lfloor \frac{n-b}{2b+1} \right\rfloor + 1}{2}\binom{b+1}{2} + \binom{\left\lfloor \frac{n-1-a(b+1)-b(a+1)}{2} \right\rfloor + 1}{2}(b+1) \leq$$

$$\binom{\left\lfloor \frac{n-b}{2b+1} \right\rfloor + 1}{2}\binom{b+1}{2} + \binom{\frac{n-1-a(b+1)-b(a+1)}{2} + 1}{2}(b+1) \leq$$

$$\binom{\frac{n-b}{2b+1} + 1}{2}\binom{b+1}{2} = \frac{b(b+1)(n-b)(n+b+1)}{4(2b+1)^2}$$

It can be obtained with simple algebraic transformations, for clarity I will omit those. This way we have an upper bound on the optimal result. Note that whenever $2b + 1$ divides $n - b$, which means that $c = 0$, the sides are equal, so this bound is tight. This function also flattens quickly as b approaches its maximal value $\sqrt{2n}$.

Now I will argue that in order to find the optimal value of b, it is sufficient to consider only those for which c is either *close* to 0 or to a. Consider any b for which the value of c is neither close to 0 nor to a. If we decrease n by a small value, there should be another b' which for this lower value n' satisfies $2b' + 1$ divides $n' - b'$ and thus its score hits the bound, so it is not smaller than for b. The same way we can increase n and look for another b'. This reasoning can be easily formalized. Precise meaning of close and will be explained later. Note that we can expect an even distribution of c values, since in this problem input is chosen randomly.

Finally, we came to a conclusion that allows us to build an algorithm – since peak score values appear for such b that the remainder of $\frac{n-b}{2b+1}$ is small, it suffices to iterate over such b and take the highest score. Clearly

$$2b + 1 | n - b \quad \Rightarrow \quad 2b + 1 | 2n - 2b \quad \Rightarrow \quad 2b + 1 | 2n + 1$$

and thus, we should iterate over the divisors of numbers *close* to $2n + 1$ and one of the obtained values for b should give the optimal result.

The algorithm is as follows. We iterate over numbers between $2n + 1 - k$, $2n + 1 + k$ for a certain value of k. Given this number, m, we factorize it with any algorithm, such as Pollard's rho. Then we iterate over odd divisors of m and assuming it is equal to $2b + 1$, we calculate the score using the formula shown before. From all the values obtained this way, we choose the highest one.

The choice of range for remainders, k, is rather hard to be determined in a formal way. This value should be chosen experimentally. Since the inputs are random, we should expect it to be rather low. Indeed, for this particular test set $k = 16$ is sufficient.

Now I will prove the key lemma. Firstly, we shall observe that all matches should be aligned as edges of a single grid – if there is more than one aligned component there can be no rectangle that contains a pair of matches from different aligned components and thus if we embed both structures in one large grid, the score cannot decrease.

Assume we have any grid-aligned set of matches. In order to prove the lemma, it suffices to show that there exists an extended rectangle that consists of non-greater number of matches in which we can find at least that many rectangles as in the starting set.

Let's transform the matches in the following way: for every column of matches (subset with fixed x-coordinate of the center) we shift it to remove the empty spaces inside and place the lowermost match on arbitrary fixed axis. Formally, match with endpoints $(x, y) - (x + dx, y + dy)$ where $0 \leq dx, dy \leq 1$ moves to $(x, N) - (x + dx, N + dy)$ where N denotes the number of matches placed lower in this column.

I will show that this operation does not decrease the number of rectangles. Fix two different columns a, b of vertical matches. All the rectangles with sides in these columns are determined by the two remaining sides, which are horizontal paths linking the columns. Denote the number of such paths by k. Clearly the number of considered rectangles cannot be greater than $\frac{k(k-1)}{2}$. Let x, y be the numbers of matches in columns a, b respectively. All the considered rectangles are also determined by intervals of consecutive matches in columns a, b, which are not greater than $\frac{x(x+1)}{2}$ and $\frac{y(y+1)}{2}$. If we align all the columns, for these two we obtain a fully connected structure of height $min(x, y, k - 1)$. And it gives at least that many rectangles as minimum of $\frac{k(k-1)}{2}$, $\frac{x(x+1)}{2}$ and $\frac{y(y+1)}{2}$. If we aggregate the result for every pair of columns, we get that the overall count of rectangles does not decrease.

After performing such shift vertically and similar horizontally we get a corner of a grid. Clearly if the solution contains a match with an endpoint that is not an endpoint of any other match, it can be removed without affecting the score, since it cannot be a part of any rectangle. After applying this operation exhaustively, we are left with a structure that consists of edges from a subset of unit cells of the grid, forming a corner. Let's denote the heights of consecutive columns of unit squares as $a_1, a_2, \ldots, a_k$. We know that $a_1 \geq a_2 \geq \cdots \geq a_k$.The number of rectangles in this set is

$$1 * \frac{a_1(a_1 + 1)}{2} + 2 * \frac{a_2(a_2 + 1)}{2} + \cdots + k * \frac{a_k(a_k + 1)}{2}$$

It can be proved by means of mathematical analysis that such function is optimized by a constant sequence (possibly except for the last element). And this is indeed the extended rectangle. Note that as long as we do not increase the first element and do not increase sequence length, keeping a fixed sum of the sequence is equivalent to using a fixed number of matches. This completes the solution.

Problem source: SPOJ
Solution by:
Name: Michał Zawalski

Round 2: Game Simulator

Statement:

"Tractor" is a very popular poker game in China. There are four players in the game. Suppose their names are Alice, Bob, Charles and David, in clockwise order. A judge is needed for this game. The players are divided into two teams, Alice and Charles are in team 1, and the other two are in team 2. The prop they use to play the game are two decks of pokers, including 108 cards in total. A simplified rule of the game is described below.

The whole game contains a number of rounds. In each round, one team is called **"Declarers" (CT)**; the other team is called **"Defenders" (FT)**. Each team has a **current rank (CR)**. The goal of the player is to increase his own team's CR as much as possible.

A certain round has a **Main Suit** (Heart - H, Spade - S, Club - C, Diamond - D, or None - no main suit in this round) and its CR. The CR in this round is the CR of the CT, and the Main Suit will be given. The Main Suit and CR will be used to determine the order of the cards.

Cards ranked 5, 10, King value 5, 10, 10 pts (points) respectively, all other cards value 0 pts. In one round, we only consider the FT's pts. The rules of getting pts for FT will be discussed later.

If the FT gets less than 80 pts in one round, they will hold the FT in the next round. This situation is called **"make"**. Otherwise, they become CT in the next round and the original CT become FT instead. This situation is called **"down"**.

If the FT gets 0 pts, the CR of the current CT will be increased by 3, for example, if the CR of the CT is 9, it will become Queen (12). Otherwise, if the FT gets less than 40 pts, the CR of the CT will be increased by 2. Otherwise, if the FT gets less than 80 pts, the CR of the CT will be increased by 1. Otherwise, if the FT gets not less than $80 + k * 40$ pts and less than $120 + k * 40$ pts, the CR of the current FT will be increased by k. For example, if the FT gets 255 pts in a round, the CR of the current FT will be increased by 4; and if the FT gets 80 pts, both teams' CR remain unchanged. If a team's CR becomes beyond Ace, this team is considered the WINNER of the whole game.

During a round, one of the players in CT is called the **dealer**. If "make", the pard (teammate) of the dealer becomes the next round's dealer. Otherwise ("down"), the player on the dealer's right-hand side becomes the dealer of the next round. For example, if the dealer of the current round is Alice and her team (CT) is "down", the dealer of the next round should be Bob (on Alice's right-hand side).

At the start of a round, each of the players except the dealer gets exactly 25 cards; the dealer gets all the remaining 33 cards. After that, the dealer chooses 8 of his cards and gives them to the judge, and these cards are called **"hidden cards"**.

Now each player has exactly 25 cards. A round consists of several **tricks**. In the first trick, the dealer plays one or more cards (called **"lead"**), then, in clockwise order, players play the same number of cards as the first player one by one (called **"follow"**). The winner of the current trick leads cards during the next trick, and so on. If the winner of the current trick is a member of FT, then the FT gets the sum of the cards' pts played in this trick.

Now we start to describe how to determine the winner of a trick. After the main suit and the CR of the current round are fixed, we can determine the **"trumps"** which are cards with the main suit or CR (Current Rank), and the jokers. All other cards are **"not-trumps"**.

We can have an order among all the cards according to the following rules:

1. "Trumps" are ordered higher than "not-trumps".
2. For the trumps, the order is listed below:
   - Red Joker
   - Black Joker
   - card with main suit and CR (if exists)
   - other card with CR
   - other trumps ordered by their ranks (i.e., A, K, Q, J, T, 9, 8, 7, …, 3, 2)
3. For the "not-trumps", they are ordered by their ranks.

Assume in all the description below, in the current round, the CR of the CT is 7.

Suppose the main suit is H, the cards can be arranged in this order (as an example):

$\boldsymbol{S2}, \boldsymbol{C2}, \boldsymbol{D2}$

$< \boldsymbol{S3}, \boldsymbol{C3}, \boldsymbol{D3}$

$< \boldsymbol{S4}, \boldsymbol{C4}, \boldsymbol{D4}$

$< \boldsymbol{S5}, \boldsymbol{C5}, \boldsymbol{D5}$

$< \boldsymbol{S6}, \boldsymbol{C6}, \boldsymbol{D6}$

$< \boldsymbol{S8}, \boldsymbol{C8}, \boldsymbol{D8}$

$< \boldsymbol{S9}, \boldsymbol{C9}, \boldsymbol{D9}$

$< \boldsymbol{ST}, \boldsymbol{FT}, \boldsymbol{CT}\ (\boldsymbol{T} - \boldsymbol{10})$

$< \boldsymbol{SJ}, \boldsymbol{CJ}, \boldsymbol{DJ}\ (\boldsymbol{J} - \boldsymbol{Jack})$

$< \boldsymbol{SQ}, \boldsymbol{CQ}, \boldsymbol{DQ}\ (\boldsymbol{Q} - \boldsymbol{Queen})$

$< \boldsymbol{SK}, \boldsymbol{CK}, \boldsymbol{DK}\ (\boldsymbol{K} - \boldsymbol{King})$

$< \boldsymbol{SA}, \boldsymbol{CA}, \boldsymbol{DA}\ (\boldsymbol{A} - \boldsymbol{Ace})$

$< H2$

$< H3$

$< H4$

$< H5$

$< H6$

$< H8$

$< H9$

$< HT$

$< HJ$

$< HQ$

$< HK$

$< HA$

$< S7 = C7 = D7$

$< H7$

$< BJ\ (the\ Black\ Joker)$

$< RJ\ (the\ Red\ Joker)$

If "None" during this round, then the pokers can be arranged in this order:

$\boldsymbol{H2, S2, C2, D2}$

$< \boldsymbol{H3, S3, C3, D3}$

$< \boldsymbol{H4, S4, C4, D4}$

$< \boldsymbol{H5, S5, C5, D5}$

$< \boldsymbol{H6, S6, C6, D6}$

$< \boldsymbol{H8, S8, C8, D8}$

$< \boldsymbol{H9, S9, C9, D9}$

$< \boldsymbol{HT, ST, FT, CT}$

$< \boldsymbol{HJ, SJ, CJ, DJ}$

$< \boldsymbol{HQ, SQ, CQ, DQ}$

$< \boldsymbol{HK, SK, CK, DK}$

$< \boldsymbol{HA, SA, CA, DA}$

$< H7 = S7 = C7 = D7$

$< BJ$

$< RJ$

In these two tables, cards written in italic are "trumps", and cards written in boldface are "not-trumps".

In each trick, the **lead cards** (played by the player leading this trick) must be either all "trumps", or all "not-trumps" with the same suit.

The possible **structures** of the cards are listed below (assume the main suit is H and main rank is 7 for the example):

- **Single.** A single card, such as D9.
- **Pair.** Two same cards, such as D9D9. But D7S7 is not a pair although their orders are the same.
- **Tractor.** Two or more consecutive-ordered pairs, satisfying the condition that they are all "trumps", or all "not-trumps" with the same suit, such as SJSJSQSQSKSKSASA, H7H7S7S7HAHA or RJRJBJBJ. But, these are not tractors: S7S7C7C7 (their orders are the same), C7C7C6C6 (they are not consecutive-ordered), DADAD2D2 (Ace is not "one", so they are not consecutive-ordered), H2H2H4H4, or D2D2D3. Be careful: if "None" in this round, H7H7S7S7HAHA is not a tractor (H7 and S7 are same-ordered because of "None").
- **Throw.** The combination of the structures above, satisfying the condition that they are all "trumps", or all "not-trumps" with the same suit. Each of the Single, Pair or Tractor in a Throw is one of the Throw's component. In the original tractor game, in some situation, the throw will be rejected. But, to keep the rule simple, we assume in this problem that all the throws are accepted. For example, RJRJBJBJH7H7HQHQHJHJH9H9H6H6HAH2 contains six components: two tractors, two pairs and two single cards (RJRJBJBJH7H7-HQHQHJHJ-H9H9-H6H6-HA-H2); CACAC8C8CK contains three components: two pairs and one single card (CACA-C8C8-CK).

A throw can be treated as a different list of components, for example, H2H2H3H3H4H4H5H5H6H6 can also be treated as H2H2H3H3-H4H4H5H5H6H6, or H2H2-H3H3H4H4H4H5H5-H6H6, and so on. For the lead cards, each time we choose the longest component (choose the one with the highest order to break the tie) to construct a list of components, this list is the structure of the lead cards, also **the structure of the trick**. So that the structure of the trick is unique.

After the first player leads his or her cards, other players follow the cards one by one in clockwise order, as mentioned above.

An important part of the game is to determine the winner of a trick:

If one's follow cards contain both "trumps" and "not-trumps", or all "not-trumps" but with different suits, this player can't be the winner of this trick.

Otherwise, if the lead cards are all "not-trumps" and one's follow cards contain "not-trumps" with a different suit from the lead cards, this follow player can't be the winner of this trick.

Else, if one's follow cards can't be constructed as the same structure of the lead cards, this player can't be the winner of this trick either.

Otherwise, if the structure of this trick is not "throw", the one who played the highest-ordered card wins this trick. If more than one player played the same highest-ordered card, the winner of this trick will be the one who plays the highest-ordered card first.

Now let's consider the "throw" situation. We construct the follow cards into the structure of the lead cards, so that the order of the highest-ordered card in all the longest components of the "throw" is as high as possible (this card is called **"honor card"**). Note that tractor can be treated as several pairs or shorter tractors, and pair can be treated as two single cards. The winner of this trick is the one who plays the highest-ordered "honor card". Similarly, if more than one player played the same highest-ordered "honor card" the winner of this trick will be the one who plays the highest-ordered "honor card" first.

There are many hair-raising rules about lead and follow cards; fortunately, they're not related to this problem, the only thing we care about is: when someone leads a "not-trump" "throw" the only possible way to beat it is to "throw" the same structure of "trumps". And it's impossible to beat a leading "trump" "throw".

Special attention on the examples below. In these examples, Alice always leads the cards. And assume in all the following examples, the CR is 7, and the main suit is H.

Alice	Bob	Charles	David	Winner	Comments
SA	S2	ST	S5	Alice	Alice plays the highest-ranked card SA.
SA	S2	ST	SA	Alice	Alice plays the first SA.
SA	S2	ST	H2	David	David plays the first only "trump".
SA	H2	C7	D7	Charles	Charles plays "trump" C7, while David plays D7

					with the same order of C7.
C2C2	C3C4	C7D7	RJBJ	Alice	The structure of this trick is "pair", while all players except Alice play two single cards.
D3D3	DTDT	SKSK	H2H3	Bob	Bob plays a pair with an order higher than Alice's, while Charles discards a pair with the wrong suit.
D3D3	DTDT	SKSK	H2H2	David	David plays the only "trump" pair.
D6D6D8D8	DJDJDKDK	DTDTD2D3	HTHTBJBJ	Alice	Alice plays the only tractor (because the CR is 7).
H6H6H8H8	H7H7BJBJ	C2C2C3C4	HKHKRJRJ	Bob	Bob's tractor is higher-ordered than Alice's.
H6H6H8H8	H7H7D7D7	C2C2C3C4	HKHKRJRJ	Bob	Bob also plays a tractor!
SASK	STST	C2H3	S7SK	Alice	Alice makes a throw.
SASK	HKH3	HAH2	S7SK	Charles	Both Bob and Charles can beat Alice.
SASK	HAH2	HAH3	S7SK	Bob	Both Bob and Charles can beat Alice, but Bob's HA comes first.

S2S2S3S3SA	H3H3H4H4RJ	D7D7H7H7H2	S7S7SQSJS6	Charles	Both Bob and Charles can beat Alice.

There is a special rule about "hidden cards": if the winner of the last trick of a certain round is a member of FT, then, in addition, the FT gets the sum of the hidden cards' pts, multiplied by 2^w(2 to the power of $\boldsymbol{w}$). When the structure of the final trick is not "throw", then $\boldsymbol{w}$ is the number of lead cards of the last trick of this round. If the structure is "throw" instead, $\boldsymbol{w}$ is the length of the longest components, in the example RJRJBJBJH7H7HQHQHJHJH6H6HA, the $\boldsymbol{w}$ is 6 because the length of RJRJBJBJH7H7 (the longest components of the "throw") is 6.

To make the problem easier, you are only to write a single-round tractor game simulator.

Input:

Multiple test cases, the number of them $\boldsymbol{T}$ is given in the very first line. For each test case:

The first line contains the main suit of this round (H, S, C, D, O; O denotes "None" in this round), the dealer of this round, the CR of team 1, the CR of team 2, separated by single spaces. Each of the rest lines contains 4 strings: the lead cards and the cards played by the second, third and last player. In one string, the cards can be given in any order. Each player will play exactly 25 cards in one round.

You may assume the input is always valid.

There is a blank line between consecutive test cases, and a blank line also appears between $\boldsymbol{T}$ and the first test case.

Output:

For each test case:

The first line contains the case number.

The second line contains the pts gotten by the FT in this round.

If a team wins the whole game after this round, output "Winner: Team **X**"(without quotes, **X** should be either 1 or 2) in the second line. If no team wins, output the new CR of team 1, the new CR of team 2 after this round, followed by the name of the dealer of the next round, separated by single spaces.

See the example for further details.

Example input:

```
1

O Charles 2 2
S6S6S7S7 SASKSJST STS8S4S4 S3S5SJSQ
S9S9 H3D3 C3DT SAD3
DA DQ DK D4
SKS8S5S3 RJC2D2H2 C6C8CJD9 H3CKDTD5
```

```
H7H7 H6H4 HJHQ H9H9
DJDJ DKH5 D5D4 D6D6
D8D8 C4C3 HTH5 D9D7
C5C5 C6CT H8HQ C7C4
H8 C7 HA HA
H2 RJ BJ CK
DA BJ C8 HK
S2S2C2 CQCAD2 HTHJHK C9CQCA
```

Example output:

```
Case #1:
50
3 2 Alice
```

Time and memory limit: 1s / 64 MB

Solution:

The hardest parts of this problem are reading the statement and parsing the input. However, there are two algorithmic sections we should focus on:

1) to determine the structure of the trick we can use a greedy algorithm: we will always pick the longest tractor from the remaining cards to add to the structure of the trick. If there are more tractors of that length, we pick the one with the highest valued card
2) to check whether follow cards can be constructed as the trick structure, we might be tempted to use the same greedy algorithm, however this does not work. A counterexample can be easily found. A solution that passes is to have a vector of the lengths of tractors in the trick structure. Then for every permutation of that vector, we greedily get the largest value tractor of that length we can.

Using those two algorithms, we simulate the game.

Problem source: SPOJ
Solution by:
Name: Mladen Puzić

Round 2: Union Laser

Statement:

Doctor Y, a leading expert in the field of boxology, is investigating the properties of boxen with his expensive super-precision laser. In particular, he plans to position the laser inside the union of a bunch of boxen and see where the beam collides with itself.

Since Doctor Y blew all his cash on the laser, he can't actually afford to purchase boxen to conduct his experiment – instead, he will simulate it on his old computer, which uses an integer grid system. He will give his computer an integer N, the number of boxen ($1 \leq N \leq 10^4$), as well as their descriptions. Each box is described by 4 integers, x_1, y_1, x_2, and y_2, and is an axis-aligned rectangle with coordinates (x_1, y_1) and (x_2, y_2) describing a pair of its opposite corners. Boxen may overlap with one another. He will also input the location of the laser (A, B), such that it will be positioned strictly within at least one of the boxen. The laser points down and right are at a $45°$ angle. All coordinates have absolute values of at most 3000.

As it turns out, boxen are made of a perfectly reflective material (a fact which Doctor Y will discover upon completion of his experiment) – as such, whenever the laser beam hits the edge of the union of the boxen, it bounces off. For example, if it hits a vertical edge from the left, it bounces to the right, with the up-down direction preserved. If it hits a corner head-on, it will reverse the direction, hence colliding with itself right at the corner. Normally, light travels quite fast, but due to the mysterious properties of boxen, the speed of light when inside a box union is only $\sqrt{2}$ units/second.

Doctor Y plans to use his computer to simulate the path of the laser and see when and where it first collides with itself, but there's one problem – he doesn't know how to program! Offering non-cafeteria food, he lures a desperate Waterloo student into his lab, where he forces him (or her?) to write the laser simulation program. That's you.

Input:

Line *1*: 1 integer, N

Line *2*: 2 integers, A and B

Next N lines: 4 integers, x_1, y_1, x_2, and y_2, describing the coordinates of each box.

Output:

If the laser beam never collides with itself or the laser, simply output "Time limit exceeded".

If the laser beam collides with the actual laser (at coordinates (A, B)) before it collides with another location through which the beam has already passed, the first line of output should consist of a single number – the amount of time that passes before the laser beam collides with the laser, in seconds. The second line should read "Broken equipment".

Otherwise, the first line of output should consist of a single number – the amount of time that passes before the laser beam first collides with itself, in seconds. The second line should contain a pair of numbers – the x and y coordinates of where this takes place.

Each number should be rounded to 1 decimal place.

Example input:

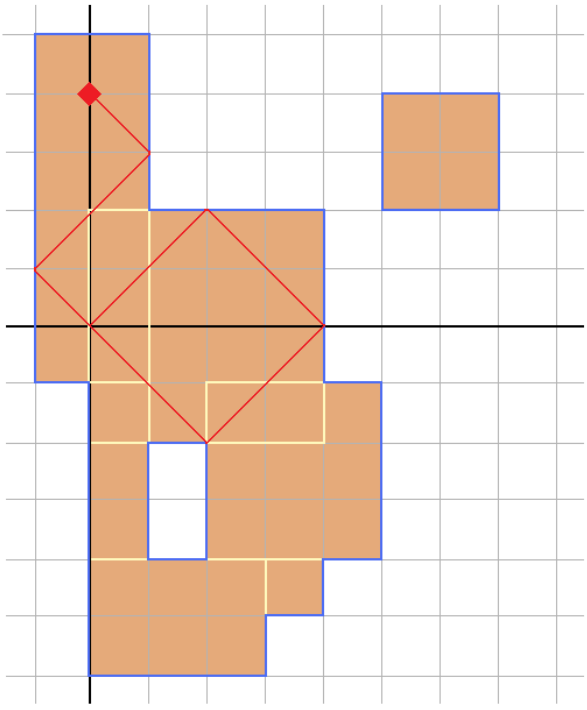
```
7
0 4
-1 5 1 -1
0 -2 4 2
0 -4 1 2
2 -4 5 -1
5 2 7 4
3 -5 4 -4
0 -6 3 -4
```

Example output:

```
12.0
0.0 0.0
```

Explanation:

The grid looks like this:



The filled-in squares represent squares that are part of at least one box – together, they make up the box union. Note that the union can have holes in it, and that it can be disjoint.

The blue lines denote the boundaries of the union – these are the lines that the laser can bounce off of.

The yellow lines are the edges of boxen that do not contribute to the union – the laser can go through these freely.

The red diamond is the position of the laser, and the red line is the path that the laser beam follows. Note that when it bounces off the corner at $(2, -2)$, since it didn't hit it head-on, it is the same as bouncing off of a horizontal edge.

Following the path of the laser beam, it can be seen that it collides with itself at $(0, 0)$. Before this, it has travelled $12\sqrt{2}$ units, which takes exactly 12 seconds.

More examples:

If the laser were positioned at $(0, 3)$, the output should be:

```
12.0
0.0 -1.0
```

If the laser were positioned at $(0, 1)$, the output should be:

```
5.0
5.0 -4.0
```

If the laser were positioned at $(0, 0)$, the output should be:

```
8.0
Broken equipment
```

Time and memory limit: 1s / 256 MB

Solution:

Since the maximum absolute value of a coordinate is 3000, it is apparent that we can just simulate the process. However, before doing that we have to know which squares belong to at least one box. We will use a 2D int array $num[6005][6005]$ that in $num[x][y]$ saves how many boxes contain the square that has the point (x, y) as the lower left corner. In order not to have negative coordinates we will add 3005 to every coordinate in input. For each box with the lower left and upper right corner at (x_1, y_1) and (x_2, y_2), we will do the following:

$$num[x_1][y_1] + +;$$

$$num[x_1][y_2] - -;$$

$$num[x_2][y_1] - -;$$

$$num[x_2][y_2] + +;$$

Now, it is not hard to notice that the required number of boxes for the square with the lower left corner (x, y) is just the sum of $num[i][j]$ such that $0 \leq i \leq x$ and $0 \leq j \leq y$. We can easily compute it using the following recurrent relation:

$$num[x][y] += num[x-1][y] + num[x][y-1] - num[x-1][y-1]$$

Now, it only remains to simulate the process. Firstly, we can notice that the beam cannot collide with itself after a non-integer number of seconds (just look at the parity of the sum of coordinates at which the beam is located). So, it will always collide with itself at a lattice point. We will have a 2D bool array that indicates whether the beam has already passed through a particular square. It is also needed to have two variables dx and dy that indicate the way at the beam moves at x and y-axis, respectively. Now, it only remains to consider the reflections of the beam. This only requires a lot of casework (probably it is possible to be done without that much casework, but firstly before bashing all the cases I got WA, so I will present only that casework solution, anyway it is just an implementation detail, but this whole task is mostly about implementation). So, I will separately consider each distinct pair of (dx, dy). There are four such pairs, but here I will cover only one because they are very similar to each other. So, if $dx = 1$ and $dy = 1$, then there are a few subcases:

1) $num[x-1][y], num[x][y-1] > 0$
- The beam hits the corner and it collides with itself.
2) $num[x-1][y] > 0 \;\; num[x][y-1] < 0$
- The beam starts going left instead of right, so dx becomes -1
3) $num[x-1][y] < 0 \;\; num[x][y-1] > 0$
- The beam starts going down instead of up, so dy becomes -1
4) $num[x-1][y], num[x][y-1] < 0 \;\; num[x][y] < 0$
- The beam hits the corner and collides with itself.
5) $otherwise$

- The beam just continues to go in the same direction along the both axes.

After checking all those cases we just have to mark the current position of the beam as visited and to update it.

Problem source: SPOJ
Solution by:
Name: Veljko Radić

Round 2: The Zebra Crossing

Statement:

Have you ever wondered why people collide with each other at pedestrian crossings? The reasons are probably difficult to analyze from a scientific point of view, but we can hazard a guess. A part of the problem seems to be that the statistical pedestrian, when faced with a red light, will either cross at once (this category of pedestrians doesn't really interest us, since they tend to collide with cars rather than with each other), or will stop dead and stand still until the light changes. Only when the light turns green does he begin to act rationally and heads for his destination using the shortest possible path. Since this usually involves crossing the road slightly on the bias, he will inevitably bump into someone going across and heading another way.

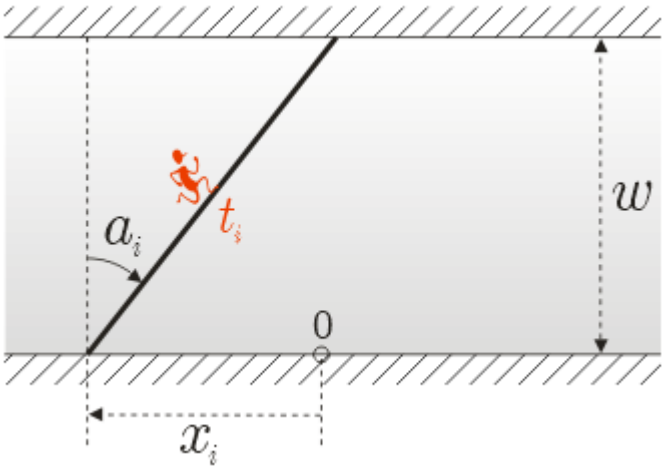
One day, while you are approaching the traffic lights you usually cross at, you begin to wonder how many other people you could possibly collide with if you really wanted. All the people are standing at different points on the same side of the street as you are. From past observations you can predict the exact angle and speed at which individual pedestrians are going to cross. You can decide at which point along the side of the street you will wait for the green light (any real coordinate other than a place where some other pedestrian is already standing) and at what angle and at what speed you intend to cross. There is an upper bound on the speed you may cross at.

Assume that once the light turns green, all pedestrians start moving along straight lines, at constant speed, and that collisions, however painful they may be, have no effect on their further progress. Since you wouldn't like to arouse anyone's suspicions, you also have to cross in accordance with these rules. A collision only occurs if at a given moment of time you have exactly the same x and y coordinates as some other pedestrian.

Input:

Input starts with a single integer t, the number of test cases. t test cases follow.

Each test case begins with a line containing three integers $n\ w\ v$, denoting the number of people other than you who wish to cross the street, the width of the street expressed in meters, and the maximum speed you can walk at given in meters per second, respectively. Each of the next n lines contains three integers $x_i\ t_i\ a_i$, which describe the starting position of the i^{th} pedestrian measured in meters, the time (in seconds) he takes to cross the street, and the angle at which he is walking with respect to the line normal to the sides of the street, expressed in $\frac{1}{60}$ parts of a degree.



Output:

For each test case output a single integer - the maximum number of people you can collide with by the time you reach the opposite side of the street.

Constraints:

- $t \leq 100$
- $1 \leq n \leq 10000$
- $1 \leq w \leq 100$
- $1 \leq v \leq 10000$
- $-10000 \leq x_i \leq 10000$
- $1 \leq t_i \leq 10000$
- $-5000 \leq a_i \leq 5000$

Example input:

```
1
5 20 2
-20 10 2700
20 10 -2700
-5 1 4000
-4 1 4000
5 1 -400
```

Example output:

```
2
```

Explanation:

In the example, due to the imposed speed limit, it is only possible to collide with the first two pedestrians while crossing the street, at the last possible moment.

Time and memory limit: 1s / 64 MB

Solution:

First of all, we will split the velocity of each pedestrian into its horizontal and vertical components. The vertical component for pedestrian i will be $v_{iv} = \frac{w}{t_i}$ and the horizontal component will be $v_{ih} = \frac{\mathrm{w} * \tan\frac{a_i}{60}}{t_i}$.

To solve the problem we need a few observations:

- Since we can't start at the same point as some other pedestrian, we can only collide with a pedestrian if we have the same vertical velocity as him.
- When choosing at which speed we want to move at, it's always optimal to move at the maximum possible speed.

We can split the pedestrians into groups that have the same vertical velocity (the same t_i), and calculate the answer for each group separately. Let's say that the time that every pedestrian needs to cross the street from the current group is t_g. If our maximum speed is less than $\frac{w}{t_g}$ (vertical velocity of the group), we can't collide with anyone from that group. Otherwise, we can calculate the maximum horizontal velocity we can move at with the following formula:

$$v_h = \cos(\arcsin(\frac{w}{t_g * v})) * v = \sqrt{1 - \left(\frac{w}{t_g * v}\right)^2} * v$$

Since we can move in either direction, we have two cases to consider for out horizontal velocity (v_h or $-v_h$).

For each pedestrian from the group we can calculate the interval of positions we need to start at if we want to collide with that pedestrian. One border will be x_i and the other will be $x_i + (v_{ih} - v_h) * t_i$. If we calculate the intervals for all the pedestrians from the group, the solution for the group will be the maximum number of intervals covering a point which is not already occupied by a pedestrian. We can calculate this using the line sweep method. The final solution is the maximum of the solutions for each group.

Complexity: $O(t * n * \log n)$

Problem source: SPOJ
Solution by:
Name: Nikola Pešić

Round 2: Connect the Points

Statement:

I'll give you a challenge! You will receive an array of size N by M, with the characters '-', '#' and '*'. See an example below:

```
*-#-*
-----
--*--
####-
*----
```

The character '#' means obstacles, the characters '-' mean empty spaces and the characters '*' mean points. Your job is to put the minimum amount of additional points needed to make all the points connected. You can only put more points in empty spaces. For example, for the matrix above, you need at least 7 additional points to connect all the points, as shown in the following figure:

```
*-#-*
-*-*-
--*-*
####*
****-
```

Do you accept the challenge?

Input:

The input contains several test cases. Each test case starts with a line containing two integers N and M indicating the dimensions of the matrix ($1 \leq N * M \leq 100$). After the first line, the next N lines describe the array in the same manner shown in the statement.

Output:

For each test, the output consists of one line containing the minimum number of additional points that need to be added to the matrix to connect all the points. If it is impossible to connect all the points, print 'impossivel'.

Example input:

```
5 5
*-#-*
-----
--*--
####-
*----
3 4
---*
----
*---
1 5
*-#-*
2 2
-#
#-
```

Example output:

```
7
2
impossivel
0
```

Time and memory limit: 3s / 64 MB

Solution:

The simplest possible solution checks every of at most $2^{\{nm\}}$ possibilities of placing a dot or not. Then we can find all the connected components in the graph of the dots using DFS or Find&Union algorithm. At the end, among the graphs with only one connected component we choose the one that needed the smallest number of dots.

Without loss of generality, we can assume that the number of rows is no less than the number of columns ($n \geq m$).

We can design a dynamic programming algorithm, where the state contains: the number of rows already processed, the assignment of dots in the last row, and how they are split into connected components.

First, let's show a way to compute this dynamic programming which is a little too slow. Fix the number of already processed rows and consider all the states of the dynamic programming corresponding to this number of processed rows. For every such state, we iterate over all 2^m possibilities of placing a dot or not in a next row and compute the new states. Note that the upper bound for the number of the states per row is Bell number (the number of ways of splitting the set into any number of nonempty subsets of any size).

In order to improve this solution, we have to refrain from creating transitions for adding a full row, since that creates $O(2^m)$ transitions per state. Rather than that, we want to create transitions which consider one field at a time. Given a state, we can either decide to stop putting any more dots in its last row and move to the next row or add a new dot in the last row. This brings down the number of transitions per state to $O(m)$.

Additionally, the above solution can be sped up even further, considering the states of the dynamic programming as vertices of a graph, and transitions as weighted edges. Typically, solving the DP involves computing the value of all states by considering them in a topological order with respect to the transition graph. However, we can run Dijkstra algorithm, which allows to skip the states, for which the answer is larger than the cost of the optimal solution.

Note that some care must be taken in properly designing the dynamic programming transitions and defining final states. For example, we cannot accept a transition which "orphans" an existing component, making it impossible for it to connect to some other component or the dot which wasn't considered yet.

Problem source: URI
Solution by:
Name: Katarzyna Kowalska

Round 2: Forest

Statement:

Bruce Force is standing in the forest. He wonders what is the tree trunk that is the farthest away, which is not blocked from his view by other tree trunks.

Bruce has made a map of the trees in the forest. The map shows his current position as the origin of a cartesian coordinate system. Tree i is shown on the map as a circle with the center (x_i, y_i) and radius r_i. You may assume that a tree trunk is visible if and only if there exists a line segment on the map from the origin $(0, 0)$ to a point on the border of the circle representing the tree trunk, where the line segment does not intersect or touch another circle.

Input:

The input contains several test cases. The first line of each test case contains one integer n, where n specifies how many trees are on the map. The following n lines contain 3 integers x_i, y_i, r_i where (x_i, y_i) is the center of the circle representing tree trunk i, and r_i is the radius of the circle. You may assume that no two circles in the input intersect, i.e., for any of the two circles, the distance between their centers is more than the sum of their radii. Moreover, you may assume that no circle contains the origin. The last test case is followed by a line containing one zero.

Hint: In the second test case, the first four trees block the view of all trees farther away than these four trees.

Output:

For each test case, print one line with the maximum Euclidean distance from the origin to a visible tree. The distance to a tree should be measured using the point of the tree closest to the origin, no matter if this point is in fact visible or not.

Round the answer to 3 digits after the decimal point.

Constraints:

- $1 \leq n \leq 1000$
- $-10000 \leq x_i, y_i \leq 10000$
- $1 \leq r_i \leq 1000$

Example input:

```
3
10 10 11
1 1 1
-20 -10 20
5
1 2 2
-2 1 1
2 -1 1
-1 -2 2
10000 -10000 1000
```

```
0
```

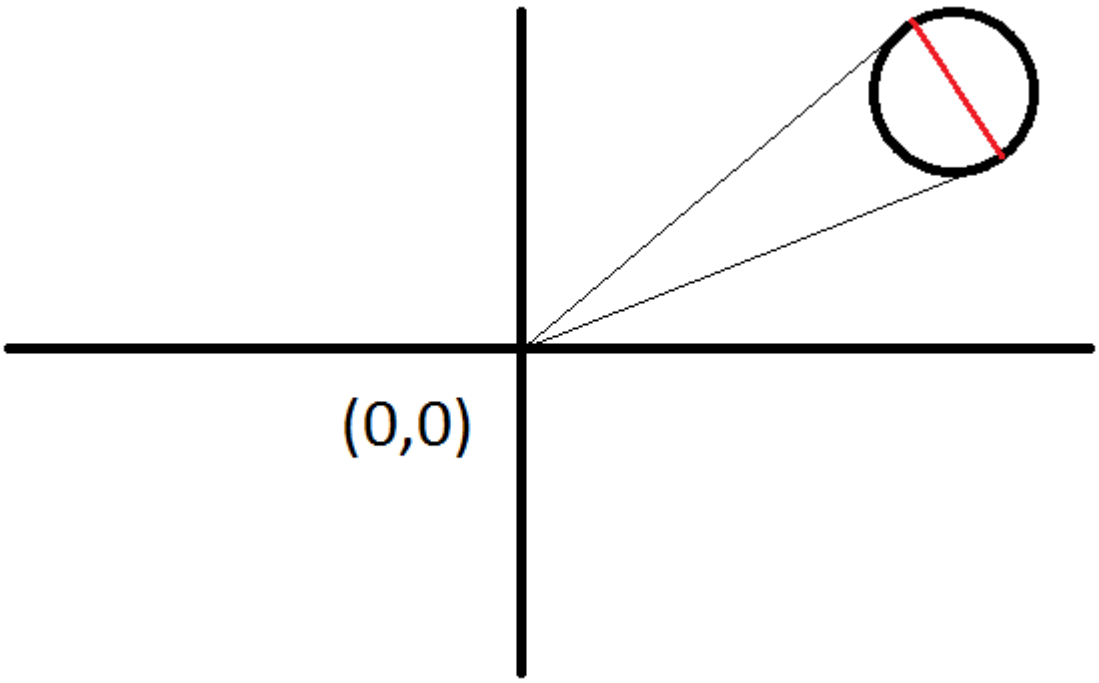
Example output:

```
3.142
1.236
```

Time and memory limit: 1s / 64 MB

Solution:

Imagine and think about what exactly you see when you look at a certain circle from point $(0, 0)$ and can you represent it differently in your code? When you think about it, it's the same as if you are looking at the line whose endpoints are the points of intersection between two tangents and a circle (tangents intersect at point $(0,0)$) . Now the task seems a little bit easier. We convert circles to lines by using analytical geometry. We find the intersection points between the tangents and the circle and those 2 points are later used instead of a circle. Also, we will need the angles between the tangents and the x-axis in order to sort the circles later.



Now we have a set of lines on a plane and we want to analyse all of the lines we can see and then find the furthest circle which is represented by a certain line. Another useful observation is that we only need to look at the endpoints because only at those endpoints we may see another circle overlap with some other circles.

Let's sort all the endpoints by a polar angle, which we define as a number from segment $[0, 2\pi]$. Now we iterate through all endpoints and we have to maintain our current solution and the circle which is the nearest one at the moment. After we update our nearest circle, we check whether it is better than our current solution. The overall complexity of the algorithm is $O(N \log N)$ per one test-case.

Just a few minor things you should be careful about:

- The Endpoint should have information about the distance from point $(0, 0)$ and which circle it represents. That way you know how to compute the distance to the nearest point on a circle.

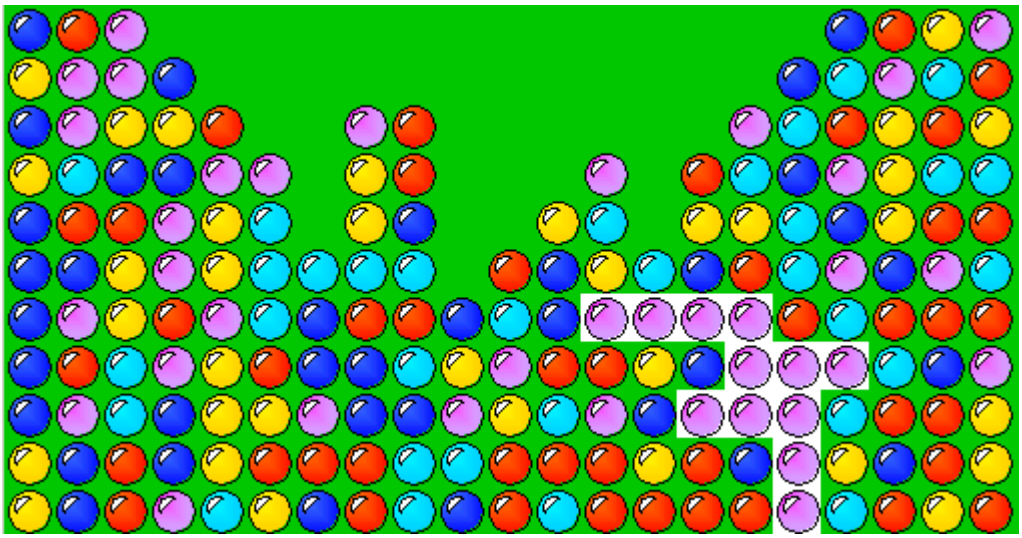
- When sorting the endpoints if they have the same polar angle you sort them by distance to the point $(0,0)$.
- Before iterating you should check all of the lines which intersect positive direction of x-axis and also find the closest one. That could be your initial solution.
- Be careful when rounding the numbers to a smaller number of decimals. I kept the solution rounded to 8 decimals until the end and only round it to less on output. You can round the results you use in computations for endpoints.

Problem source: URI
Solution by:
Name: Pavle Janevski

Round 2: [Challenge] JawBreaker Game

Statement

The goal of this popular game is to get the maximum number of points, by removing aligned pieces of the same color. The pieces can only be removed if they touch each other by an entire side. The more pieces you remove in one turn, the more points you get. The number of points in one turn is described by the following formula: $N * (N - 1)$, where N is the number of pieces (for example $2 * (2 - 1) = 2$, $10 * (10 - 1) = 90$). If you remove pieces from the middle of the field, then all the pieces located higher fall down and occupy the empty spaces. The game is finished when no pieces which can be removed from the field remain.



In this problem you will be given a field and pieces on it. Your goal is to obtain the maximum number of points.

Note: You can practice a little and plan your strategy with this on-line game [the on-line game is slightly different from the one described above]: **http://www.bigfrog.net/jawbreaker/**

Input

t – the number of tests, then t tests follow.

Each test starts with 3 integers: H - the number of rows of the playing field, W - the number of columns of the playing field and C - the number of different colors of the pieces. Then follow H rows with W numbers in each, separated by spaces. Each number is in the range from 0 to $C - 1$ and describes the color of a piece.

Output

For each test you must output the letter "Y" if you want to solve this test, or the letter "N" otherwise. If you output Y, you must output a set of lines with 2 integers x, y in each. These integers define rows and columns in the field $[0 \leq x < H], [0 \leq y < W]$. Coordinates are counted from the upper left corner of the field. After your last move output the line $-1\ -1$.

You'll receive the status Wrong Answer if your coordinates are outside the field, or point to an empty space, or to a single piece.

Score:

The score received for this problem is the sum of the points received for each playing field. The score for a playing field is calculated as $\left\lfloor \frac{100 * C * C * base\_score}{H * W} \right\rfloor$, where C – number of different colors, H – field height, W – field width, $base\_score$ – the number of points calculated as in the description of the problem.

Constraints:

- $t \leq 500$
- $4 \leq H, W \leq 50$
- $3 \leq C \leq 20$

Example input:

```
1
4 4 3
0 0 1 1
1 1 2 2
0 1 2 0
0 1 1 2
```

Example output:

```
Y
1 0
1 0
3 2
-1 -1
```

Explanation:

Initial field:

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

After the first turn (removed 5 "ones"):

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

After the second turn (removed 4 "zeros"):

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

After the third turn (removed 3 "twos"):

```
. . . .
. . . 1
```

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

Score:

In this case $base\_score = 5 * (5 - 1) + 4 * (4 - 1) + 3 * (3 - 1) = 20 + 12 + 6 = 38$, so $score = \lfloor (100 * 3 * 3 * 38)/(4 * 4) \rfloor = \lfloor 2137.5 \rfloor = 2137$.

Time and memory limit: 4s / 256 MB

Solution:

It's quite clear that there is no chance of finding an optimal solution within the given input constraints and time limits. Thus, we must rely on heuristic algorithms.

Moreover, our board is changing during the play so we cannot apply typical optimization techniques such as local search. For example, assuming we have a sequence of moves, we cannot simply change a move in the middle of the sequence as it will often make some of our further moves invalid. Hence, we will focus on fully greedy approaches.

It's worth playing this game for a while to observe that there are two basic strategies in this game:

1) Creating and collecting as many groups of more than 1 piece as possible.
2) Building a large block in one color by eliminating the other colors and hitting it when it cannot be enlarged.

Although the second strategy leads to much higher base score when there are few colors available, it is not very useful when there are many colors on the board – then it is very difficult to make a large one-colored block. Indeed, after taking experiments with randomly generated boards, we concluded that the second strategy should be applied only if the board contains less than 15 colors.

Now, let's discuss some details of implementing both approaches. Firstly, we divided available blocks of one-colored pieces into vertical and horizontal groups depending on how they are connected. Let's observe that if a few pieces of one color form a vertical group, they cannot become disconnected after removing other pieces. However, this property is clearly false for horizontal groups – it is possible that after removing some pieces below the horizontal group, it may become disconnected.

Our implementation of the first strategy is based on the observations above. The algorithm consists of many steps. In each step we look for horizontal blocks at first. If there are many such groups, we choose the uppermost one and collect it. Finally, if there is no horizontal block of pieces, we hit the lowest vertical group.

For the second strategy the only difference from the previous approach is, that we choose a special color which should be omitted (i.e. not collected) unless we cannot hit a block of any other color. We considered a few most frequent colors as candidates for such special color.

Summing up, our solution consists of two greedy algorithms – we decide which of them to run based on the number of colors on the board. During the competition we tried many other minor improvements but the key to obtaining a good score was to apply the described methods.

Problem source: SPOJ
Solution by:
Name: Konrad Majewski