Bubble Cup 2018

Microsoft Development Center Serbia

Problem set & Analysis from the Finals and Qualification rounds

Belgrade, 2018

Scientific committee:
Aleksandar Damjanović
Aleksandar Lukač
Aleksandr Milanin
Andrija Jovanović
Balša Knežević
Branko Fulurija
Daniel Paleka
David Milićević
Filip Vesović
Ibragim Ismailov
Kosta Grujičić
Miloš Šuković
Nebojša Savić
Nikola Smiljković
Ognjen Tošić
Predrag Ilkić
Radoica Draškić
Slavko Ivanović

Qualification analysts:
Aleksa Miljković
Aleksandar Maksimović
Andrei Valchok
Andrijana Dejković
David Milinković
Emil Indzhev
Filip Ćosović
Ivan Avirović
Ivan Stošić
Jovan Pavlović
Konrad Majewski
Krzysztof Maziarz
Mark Karneichyk
Michal Zawalski
Mladen Puzić
Nikola Jovanović
Nikola Pešić
Pavle Martinović
Piotr Nawrot
Stjepan Požgaj
Tadija Šebez
Vladimir Milenković
Vladyslav Hlembotskyi

Cover:
Sava Čajetinac

Typesetting:
Ana Trajković

Volume editor:
Dragan Tomić

Contents

Preface

Dear Bubble Cup finalists,

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

This year, we have introduced two divisions: 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 schools teams. By incorporating this change and 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 11 has gathered more than 70 competitors in 25 teams (Premier League consists of 17 teams and Rising Stars consists of 8 teams). Finalists come from Belarus, Bulgaria, Croatia, Poland, the UK, Ukraine, Poland and Serbia. The Rising Stars finalists, composed of high school teams from Serbia, come from Belgrade, Kragujevac, Niš, Novi Sad, Sombor and Šabac.

We are more than glad to see some of you coming back to Bubble Cup as competitors, judges or crew! We appreciate your trust in this competition and we are proud to have you as contributors to its growth and development. Except for the challenging moments you will encounter while solving the problems, we hope that you are going to take away with you some fun memories, new friendships and other opportunities from the Bubble Cup Finals.

Having all these things in mind, we hope you will join us next year and contribute to making Bubble Cup an inspiring, productive and enjoyable event.

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. Serbian high school teams can choose to compete in Premier league by contacting Bubble Cup organizers to change their default division by the start of Round 2.

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. Teams competed in two divisions in the finals: Premier League and Rising Stars.

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 XI Finals were held on September 22, 2018, at the VIG Plaza in Belgrade (New Belgrade).

Andreja Ilić, a participant of the first Bubble Cup and later a Bubble Cup organizer, officially opened the 11th Bubble Cup. He stressed the importance of the competitions based on the algorithimic problem solving and the fact that they develop a cognitive way of thinking, which is imporant for every part of our lives, in his speech for the finalists.

This initiative inspired a vast number of Serbian high schools to participate in this competition. This confirms the significance and the positive influence of this competition on young talents. In addition to that, it is important to mention that a lot of people empleyed in the MDCS today were the finalists of Bubble Cup. They, along with the other participants, are the people who have the potential to change this world for the better.

The competition started at 11.00am and lasted until 4.00pm. 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 qualification phase Premier League gathered the very best teams in Europe, and in the end, the challenge problem made all the difference in terms of who will qualify for the finals. This intense fight inspired us to expand the number of teams coming to the finals to 17. The Rising Stars league attracted a record number of Serbian high school teams during qualifications. Compared to Bubble Cup X, a 110% more high school teams from Serbia participated in the qualifications, fighting to become one of the 8 teams in the finals. While we are delighted to see so many new faces coming to Bubble Cup, we are also very pleased to see so many returning competitors.

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

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

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 2 Rising Stars teams.

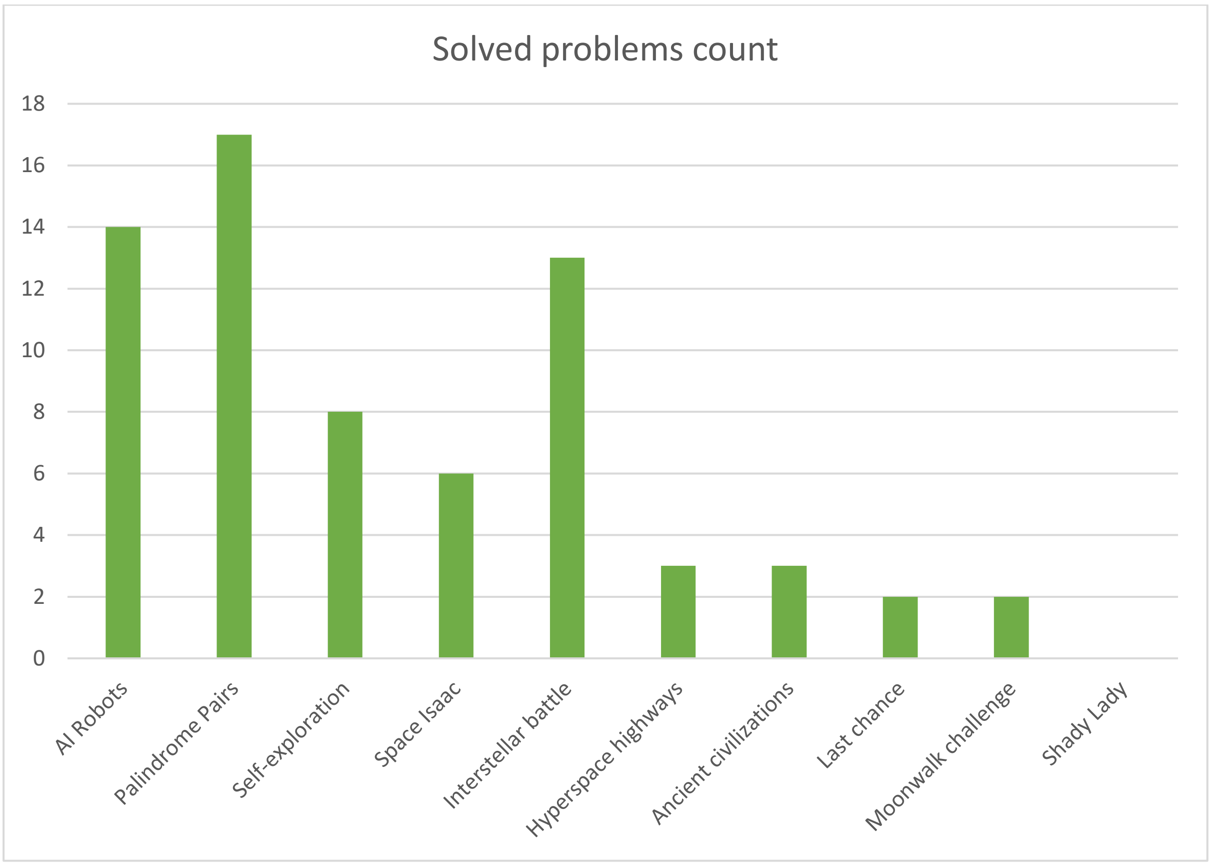
Bubble Cup Finals Results – Premier League

Problems for this year's edition of Bubble Cup were extremely difficult, in order to match the very best teams we had on the finals. This was likely the thoughest problem set in the Bubble Cup history. **Warsaw Igloos (Marek Sommer, Kamil Dębowski, Karol Kaszuba)** stellar performance, as the only team to solve 9 problems out of 10, convincigly won this year's Bubble Cup. They took a lead after 90 minutes and only increased it as the time went by. **Pchnąć w tę łódź jeża lub ośm skrzyń fig (Michał Zawalski, Konrad Majewski, Konrad Paluszek)** stayed in the top 4 places for the most of the competition and solving the problem "Self-exploration" in the last 30 mins gave them 2nd place. Biggest change on the standings during the 1-hour freeze time was made by **BSUIR2 (Andrei Valchok, Aliaksei Vistiazh, Andrei Sobal)**, who went from the 8th place and 3 solved problems before the freeze to 6 problems. Their last correct submission was made 5 minutes before the end of the contest.

Scoreboard

#	Team name	Score	Penalty
1	Warsaw Igloos	9	1678
2	Pchnąć w tę łódź jeża lub ośm skrzyń fig	6	1119
3	BSUIR2	6	1256
4	Jagiellonian Owls	5	796
5	Zato što volimo PMF	5	970
6	Selski Babi	5	1234
7	Danonki	4	408
8	fufel	4	560
9	Les Misérables	4	771
10	BooleИaNs	4	821
11	Slaven me lupo	3	441
12	nikva tikva	3	459
13	φφ team	3	493
14	øjhøjh	3	508
15	RAF_101	2	655
16	CornerCase	1	-46
17	lemi lemi sve zalemi	1	12

Problems solved distribution for Premier League



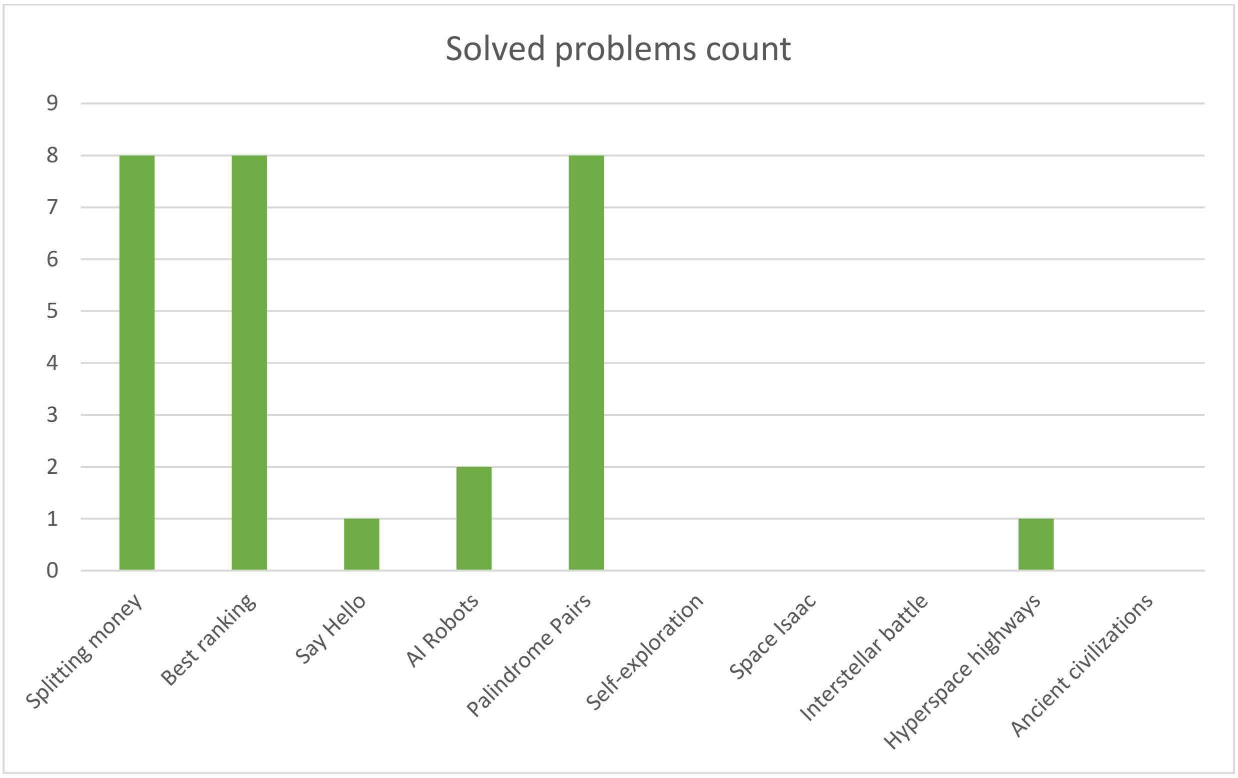
Bubble Cup Finals Results – Rising Stars

Bits please (Nikola Pešić, Tadija Šebez, Jovan Pavlović) started strong and kept their lead throughout the competition. Their last correct submission was a couple of seconds before the end of the contest for the problem Say „Hello!“. The next three teams were fighting for 2nd place award by trying to be the first team to solve the problem „AI Robots“. In the end **Inspekcija (Igor Pavlović, Marko Grujčić, Uroš Maleš)** came on top with the best penalty out of teams with four solved problems.

Scoreboard

#	Team	Score	Penalty
1	Bits please	6	911
2	Inspekcija	4	349
3	Gii Klub	4	554
4	Gimnazija Sombor	4	908
5	Nepohvaljeni	3	344
6	greattor	3	368
7	VR1.s	3	443
8	The_GodFathers	3	509

Problems solved distribution for Rising Stars



Problem A: Splitting money

Rising Stars division only

Author:

Aleksandar Damjanović

Implementation and analysis:

Aleksandar Damjanović

Andrija Jovanović

Statement:

After finding and moving to the new planet that supports human life, discussions started on which currency should be used. After long negotiations, Bitcoin was ultimately chosen as the universal currency.

These were the great news for Alice, whose grandfather got into Bitcoin mining in 2013, and accumulated a lot of them throughout the years. Unfortunately, when paying something in bitcoin everyone can see how many bitcoins you have in your public address wallet.

This worried Alice, so she decided to split her bitcoins among multiple different addresses, so that every address has at most x satoshi (1 bitcoin = 10^8 satoshi). She can create new public address wallets for free and is willing to pay f fee in satoshies per transaction to ensure acceptable speed of transfer. The fee is charged from the address transaction is sent from. Tell Alice how much total fee in satoshi she will need to pay to achieve her goal.

Input:

First line contains number N representing total number of public addresses Alice has. Next line contains N integer numbers $a[i]$ separated by a single space, representing how many satoshi Alice has in her public addresses.

Last line contains two numbers x and f representing maximum number of satoshies Alice can have in one address, as well as fee in satoshies she is willing to pay per transaction.

Output:

Output one integer number representing total fee in satoshi Alice will need to pay to achieve her goal.

Constraints:

- $1 \leq N \leq 200\,000$
- $1 \leq a[i] \leq 10^9$
- $1 \leq f < x \leq 10^9$

Example input:

```
3

13 7 6

6 2
```

Example output:

```
4
```

Explanation 1:

Alice can make two transactions in a following way:

0. 13 7 6 (initial state)
1. 6 7 6 5 (create new address and transfer from first public address 5 satoshi)
2. 6 4 6 5 1 (create new address transfer from second address 1 satoshi)

Since cost per transaction is 2 satoshi, total fee is 4 satoshi.

```
Time and memory limit: 1s/ 256 MB
```

Solution and analysis:

It’s useful to notice that since we can create multiple addresses for free we never need to transfer bitcoin to an existing address. Therefore, we can solve the problem for each public address separately and sum the fees at the end.

In order to minimize the number of fees we will need to minimize the number of transactions, because the cost per transaction is constant. Therefore, we will need to make transactions as large as possible – in other words $f + x$. So, if number of satoshies in the address is $a[i]$, we might think we need exactly $\left\lfloor \frac{a[i]}{f+x} \right\rfloor$ transactions in order to make them $\leq x$. Unfortunately, this isn’t entirely true because the remainder $a[i] \,\%\, (f + x)$, can still be larger than x. If it is, then we need one extra transaction of size $f + 1$. This solves the problem per one public address.

Another important observation is that even though all numbers on the input are 32-bit integers, given the constraints, sum of the fees per public address can be larger than 32-bit integer allows. So we will need to use 64-bit integer for summation.

Problem B: Best Ranking

Rising Stars division only

Author:

Stefan Velja

Implementation and analysis:

Slavko Ivanović

Miloš Šuković

Statement:

Formula 1 officials decided to introduce new competition. Cars are replaced by space ships and number of points awarded can differ per race.

Given the current ranking in the competition and points distribution for the next race, your task is to calculate the best possible ranking for a given astronaut after the next race. It's guaranteed that given astronaut will have unique number of points before the race.

Input:

The first line contains two integer numbers - N representing number of F1 astronauts, and current position of astronaut *D* you want to calculate best ranking for (no other competitor will have the same number of points before the race).

The second line contains N integer numbers $S_k, k = 1 \ldots N$ separated by a single space, representing current ranking of astronauts. Points are sorted in non-increasing order.

The third line contains N integer numbers $P_k, k = 1 \ldots N$, separated by a single space, representing point awards for the next race. Points are sorted in non-increasing order, so winner of the race gets the maximum number of points.

Output:

Output contains one integer number – the best possible ranking for astronaut after the race. If multiple astronauts have the same score after the race, they all share the best ranking.

Constraints:

- $1 \leq N \leq 200000$
- $1 \leq D \leq N$
- $0 \leq S_k \leq 10^8, k = 1 \ldots N$
- $0 \leq P_k \leq 10^8, k = 1 \ldots N$

Example input:

```
4 3

50 30 20 10

15 10 7 3
```

Example output:

```
2
```

Explanation:

If the third ranked astronaut wins the race, he will have 35 points. He cannot take the leading position, but he can overtake the second position if the second ranked astronaut finishes the race at the last position.

```
Time and memory limit: 1s / 256 MB
```

Solution and analysis:

Let's name our competitor Kimi and label S as his starting position. To maximize Kimi's final position, we will give him max number of points. After that, the competitor is on position C. It is safe to conclude that the best position of competitor X meets the required equation $C \leq X \leq S$. L is a list of all competitors who should have less or the same number of points as Kimi after all the points have been given and P is the list of points we should give to this competitor. Now we need to give points to other competitors in such a way to maximize length of L. Let's start from the position $C + 1$ (the first astronaut after Kimi) and proceed to give to all the competitors points to achieve our goal. Lists L and P are empty at the beginning. We should try to give the competitor the least number of points from remaining set of unassigned points to keep them below Kimi.

- If we can keep them below Kimi we need to add them to list L and we need to add current minimum number of remaining points to list P. Note: Do not give points to competitors yet.
- If we cannot keep them below Kimi we need to:
    - If L is not empty: remove the one with the most points from the list and add the current one or
    - If L is empty: try with next competitor

When we reach the end of the scoreboard we have both lists ready, we just need to give points: the least number of points to the best competitor,...

Time: $O(n)$ Memory: $O(n)$

Problem C: Say "Hello!"

Rising Stars division only

Author:

Aleksandar Damjanović

Implementation and analysis:

Miloš Šuković

Slavko Ivanović

Statement:

Two friends are travelling through Bubble galaxy in their spaceships. They say "Hello!" via signals to each other if their distance is smaller or equal than d_1 and

- it's the first time they speak to each other or
- at some point in time after their last talk their distance was greater than d_2.

We need to calculate how many times friends said "Hello!" to each other. For N moments, you'll have an array of points for each friend representing their positions at that moment. A person can stay in the same position between two moments in time, but if a person made a move we assume this movement as movement with constant speed in constant direction.

Input:

The first line contains one integer number N representing number of moments in which we captured positions for two friends.

The second line contains two integer numbers d_1 and d_2.

The next N lines contains four integer numbers Ax, Ay, Bx, By representing coordinates of friends A and B in each captured moment.

Output:

Output contains one integer number that represents how many times friends will say "Hello!" to each other.

Constraints:

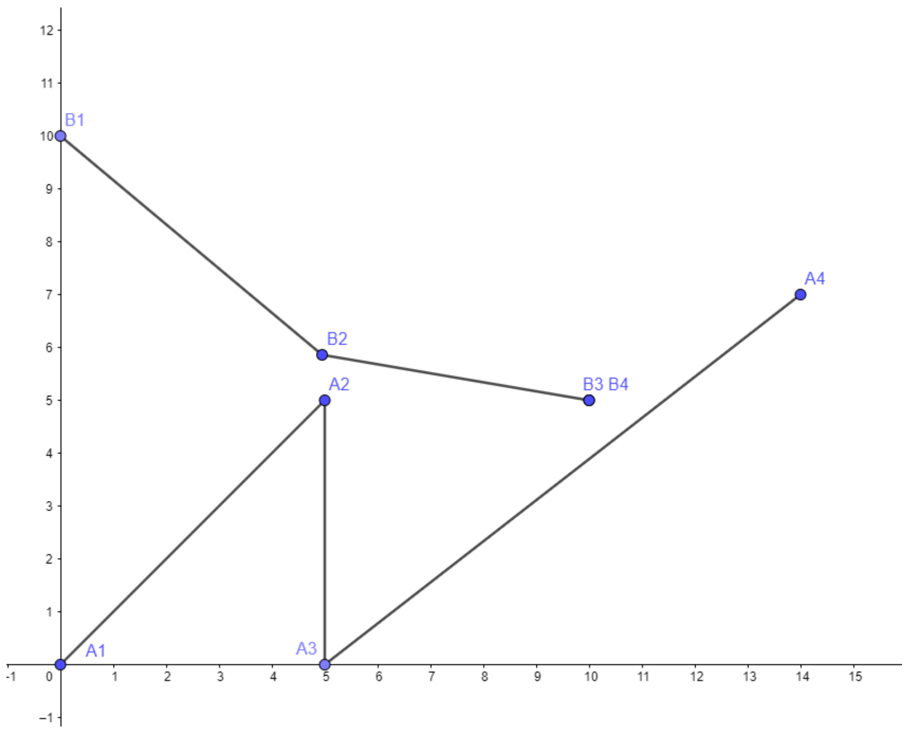
- $2 \leq N \leq 100\,000$
- $0 < d_1, d_2 < 1000$
- $0 \leq Ax, Ay, Bx, By \leq 1000$

Example input:

```
4

2 5

0 0 0 10

5 5 5 6

5 0 10 5

14 7 10 5
```

Example output:

```
2
```



Explanation:

Friends should send signals 2 times to each other, first time around point $A2$ and $B2$ and second time during A's travel from point $A3$ to $A4$ while B stays in point $B3 = B4$.

```
Time and memory limit: 2s / 256 MB
```

Solution and analysis:

First, we must find minimum distance between two moving points on segments (possibly a dot, if there is no movement) for each iteration.

One way to do this is to parametrize moving points on segments with parameter $t \in [0, 1]$. If we define first segment with endpoints *A* and *B*, and second segment with endpoints *C* and *D*, we can define moving points $M = A + t\,(B - A)$ on first segment and $N = C + t\,(D - C)$ on second segment.

We are looking for minimum value of $|MN|$ as function of *t*:

$$\min_{t\in[0,1]} \sqrt{[Mx - Nx]^2 + [My - Ny]^2} =$$

$$\min_{t\in[0,1]} \sqrt{[(Ax - Cx) + t\,((Bx - Ax) - (Dx - Cx))]^2 + [(Ay - Cy) + t\,((By - Ay) - (Dy - Cy))]^2}$$

We can omit the square root function, since minimum for functions with and without square root will be in the same t_0. After derivation on variable t and comparison with 0 (This method is used for finding local extremum of the function. In our case, we know it will be minimum, because two lines which are not parallel in space are getting further, so quadratic function will be convex) from equitation, we have two cases for finding t_0 where function has minimum value.

Let's define $Kx = (Bx - Ax) - (Dx - Cx)$ and $Ky = (By - Ay) - (Dy - Cy)$.

1. $Kx = 0$ and $Ky = 0$, segments are parallel and then for every *t*, $|MN|$ has the same value.

2. Otherwise $t_0 = -\frac{(Ax-Cx)*Kx+(Ay-Cy)*Ky}{Kx^2+Ky^2}$. If $t_0 < 0 \; or \; t_0 > 1$ we search for minimum in $t = 0$ and $t = 1$, because function is monotonic on $[0,1]$, because quadratic function has one local extremum, and if it's not on $[0,1]$, function is monotonic on that interval.

Now we should just track the distance and increase the counter when conditions from statement are met. We can do this by defining two states in which points after every move:

1. Should get closer than d_1.

2. Should get further than d_2.

Starting state is calculated by distance of the points A and B in starting positions:

```
int state = start_dist < d1 ? shouldGetFurther : shouldGetCloser;
if (state == shouldGetFurther) counter++;
```

For each segment, we will change states by looking into previous state and following two variables:

1. Closest distance between two moving points on segments we are currently comparing.
2. Distance between moving points at end of the segments.

```
if (state == shouldGetCloser) // current distance from A is greater than d1
{
	if (closestDistance <= d1)
	{
		counter++; // saying “Hello!”

		// It's possible that in the same move, we were closer than d1, but
		after, again, further than d2. In that case, we don't change the state.
		// We change the state only if we stayed closer than d2.
		if (endPointDistance <= d2)
			state = shouldGetFurther;
	}
}
else // state == shouldGetFurther (current distance from A is smaller than d2).
{
	if (endPointDistance > d2)
		state = shouldGetCloser;
}
```

Problem D: AI robots

Premier League and Rising Stars

Author:

Kosta Grujčić

Implementation and analysis:

Kosta Grujčić

Filip Vesović

Statement:

In the last mission, MDCS has successfully shipped N AI robots to Mars. Before they started exploring, they were arranged in a line. Every robot can be described with **three** numbers: position (x_i), radius of sight (r_i) and IQ (q_i). Since they are intelligent robots they are very talkative. Two robots can talk only if they can see each other. Radius of sight is inclusive, so the i^{th} robot can see all other robots in a range $[x_i - r_i, x_i + r_i]$. However, they don't walk to talk with any robot, but only with robots who have similar IQ. By similar IQ we mean that absolute difference of their IQs **isn't bigger than K.**

Help us and calculate how many pairs of robots are going to talk with each other, so we can timely update their software and avoid any potential quarrel.

Input:

The first line contains two integers, number N and K. Next N lines contain three numbers each – x_i, r_i and q_i respectively.

Output:

Output contains only one number – solution of the problem.

Constraints:

- $1 \leq N \leq 10^5$
- $0 \leq x_i, r_i, q_i \leq 10^9$
- $0 \leq K \leq 20$

Example input:

```
3 2

3 6 1

7 3 10

10 5 8
```

Example output:

```
1
```

Explanation:

The first robot can see the second, but not vice versa. The first robot can't even see the third. The second and the third robot can see each other and their IQs don't differ more than 2, so only one pair of robots will have a talk.

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

Solution and analysis:

Let's translate statement into mathematical language. We have to find number of sets $\{i, j\}$ so that $\min(r_i, r_j) \geq |x_i - x_j|$ and $|q_i - q_j| \leq K$. First of all, we have to find for each robot which robots it can see. To do that we perform line sweeping:

- Add $x_i - r_i$ as event of type 0
- Add x_i as event of type 1
- Add $x_i + r_i$ as event of type 2

After we sort all these events, we iterate them from left to right. Whenever we face event of type 0 we *save information* that a robot stands on position x_i. When we face event of type 2, we *delete information* that there is a robot on position x_i. And finally, when we face event of type 1 we have to count how many robots stand in positions $[x_i - r_i,\ x_i + r_i]$. It's easy to see that all these robots see ith robot as well. We will use segment tree to store all these information about robots, so complexity of counting how many robots each of them can see is $O(\log X_{MAX})$, as well as updating part. But not all robots will talk to each other, so we have to take their IQ in count. Since K is small number, we can iterate through valid IQs, and perform queries individually. We have segment tree for every possible IQ and we update corresponding tree everytime we face events of type 0 or 2. Since no more than $2 \cdot K$ queries are going to be performed, total complexity of processing one robot is $O(K \cdot \log X_{MAX})$.

Problem is that we don't have enough memory to store all these segment trees because IQ can be up to 10^9. So we compress these values and map them in range $[1, N]$. Since consecutive compressed IQs don't necesserally differ by 1, we have to care about that. Same problem applies to coordinates – we don't have memory to build complete segment tree on array of length 10^9. But since we have to store N segment trees, we need to use their implicit variant and save only what we really need, so we can use original values of positions, which will result in complexity of $O(\log X_{MAX})$ per query, or also use value compression and make it $O(\log N)$.

Overall time complexity is $O(K \cdot N \log N)$ or $O(K \cdot N \log X_{MAX})$ and space complexity is $O(NlogN)$ or $O(N \log X_{MAX})$.

Note that we can use policy based data structures and completely avoid segment trees and keep the same complexities.

Problem E: Palindrome Pairs

Premier League and Rising Stars

Author:

Branko Fulurija

Implementation and analysis:

Branko Fulurija

Balša Knežević

Statement:

After learning a lot about space exploration, a little girl named Ana wants to change the subject.

Ana is a girl who loves palindromes (string that can be read the same backwards as forward). She has learned how to check for a given string whether it's a palindrome or not, but soon she grew tired of this problem, so she came up with a more interesting one and she needs your help to solve it:

You are given an array of strings which consist only of small letters of the English alphabet. Your task is to find **how many** palindrome pairs there are in the array. A palindrome pair is a pair of strings where the following condition holds: **at least one** permutation of the **concatenation** of the two strings is a palindrome. In other words, if you have two strings, let's say aab and $abcac$, and you concatenate them into $aababcac$, you have to check if there is a permutation of this new string such that it is a palindrome (in this case there is the permutation $aabccbaa$).

Two pairs are considered different if the strings are located on **different indices**. The pair of strings with indices (i, j) is considered **the same** as the pair (j, i).

Input:

First line contains a positive **integer** N, representing the length of the input array.

Next N lines contain a string (only letters 'a' – 'z'), the elements of the input array.

Output:

Output one number, representing how many palindrome pairs there are in the array.

Constraints:

- $1 \leq N \leq 100\ 000$
- $The\ total\ number\ of\ characters\ in\ the\ input\ array\ will\ be\ less\ than\ 1\ 000\ 000$

Example input 1:

```
3
aa
bb
cd
```

Example output 1:

```
1
```

Explanation 1:

```
aa + bb => abba
```

Example input 2:

```
6
aab
abcac
dffe
ed
aa
aade
```

Example output 2:

```
6
```

Explanation 2:

```
aab + abcac = aababcac => aabccbaa
aab + aa = aabaa
abcac + aa = abcacaa => aacbcaa
dffe + ed = dffeed => fdeedf
dffe + aade = dffeaade => adfaafde
ed + aade = edaade => aeddea
```

```
Time and memory limit: 2s / 256 MB
```

Solution and analysis:

For each string we need to remember the parity of the occurrences of each letter

If permutation of the concatenation of two strings makes a palindrome, this means that only one letter appears an odd number of times.

To have a permutation of the concatenation of two strings that make a palindrome, the parity of the occurrence of each letter in both strings can be distinguished only in one position.

It is necessary to find for each string how many strings there are, so that their series of parity differ in only one position.

Make a mask where the i-th position signifies the parity of the i-th letter in the alphabet. Memorize each mask as many times as it appears. We will use a map.

Iterate through all the strings, add the number of masks that differ in only one position from the current mask of string to the solution.

As each two strings have evolved twice, the solution will be divided by 2.

Time complexity: $O(N \log N)$.

Problem F: Self-exploration

Premier League and Rising Stars

Author:

Aleksandar Damjanović

Implementation and analysis:

Aleksandar Damjanović

Radoica Draškić

Statement:

Being bored of exploring the Moon over and over again Wall B decided to explore something he is made of – binary numbers. He took a binary number and decided to count how many times different substrings of length two appeared. He stored those values in c_{00}, c_{01}, c_{10} and c_{11}, representing how many times substrings $00, 01, 10$ and 11 appear in the number respectively. For example:

- $10111100 \rightarrow c_{00} = 1,\ c_{01} = 1,\ c_{10} = 2,\ c_{11} = 3$
- $10000 \rightarrow c_{00} = 3,\ c_{01} = 0,\ c_{10} = 1,\ c_{11} = 0$
- $10101001 \rightarrow c_{00} = 1,\ c_{01} = 3,\ c_{10} = 3,\ c_{11} = 0$
- $1 \rightarrow c_{00} = 0,\ c_{01} = 0,\ c_{10} = 0,\ c_{11} = 0$

Wall B noticed that there can be multiple binary numbers satisfying the same $c_{00}, c_{01}, c_{10}, c_{11}$ constraints. Because of that he wanted to count how many binary numbers satisfy the constraints c_{xy} given the interval $[A, B]$. Unfortunately, his processing power wasn't strong enough to handle large intervals he was curious about. Can you help him? Since this number can be large print it modulo $10^9 + 7$.

Input:

First two lines contain two positive **binary** numbers A and B, representing the start and the end of the interval respectively.

Next four lines contain four integer numbers $c_{00}, c_{01}, c_{10}, c_{11}$ in **decimal** form representing the count of two-digit substrings $00, 01, 10$ and 11 respectively.

Output:

Output one integer number representing how many binary numbers in the interval $[A, B]$ satisfy the constraints $mod\ 10^9 + 7$.

Constraints:

- $1 \leq A \leq B < 2^{100\,000}$
- $0 \leq c_{00}, c_{01}, c_{10}, c_{11} \leq 100\,000$
- *A, B are valid binary numbers and have no leading zeroes*

Example input 1:

```
10

1001

0

0

1

1
```

Example output 1:

```
1
```

Explanation 1:

The binary numbers in the interval $[10, 1001]$ are $10, 11, 100, 101, 110, 111, 1000, 1001$. Only number 110 satisfies the constraints: $c_{00} = 0$, $c_{01} = 0$, $c_{10} = 1$, $c_{11} = 1$.

Example input 2:

```
10

10001

1

2

3

4
```

Example output 2:

```
0
```

Explanation 2:

No number in the interval satisfies the constraints.

```
Time and memory limit: 1s/ 256 MB
```

Solution and analysis:

Before we start thinking more about the problem let's make couple of simple observations:

1. Constraints $c_{00}, c_{01}, c_{10}, c_{11}$ define exact number of digits in the number. $digitsCount = c_{00} + c_{01} + c_{10} + c_{11} + 1$
2. If function $F(x)$ counts the number of binary numbers that satisfy constraints $c_{00}, c_{01}, c_{10}, c_{11}$ less than x modulo $MOD = 10^9 + 7$, than the answer is $(F(B + 1) - F(A) + MOD)\ \%\ MOD$. Notice that we must add MOD because otherwise answer could be negative and thus wrong.
3. If number x has less digits that $digitsCounts$, than $F(x)$ will be 0.

Let's first solve easier problem by calculating $F(x)$, when number x has more digits than $digitsCount$, or in other words all binary numbers that satisfy constrains.

When observing the number that satisfies given constraints $c_{00}, c_{01}, c_{10}, c_{11}$, we can reduce every group of ones to just single 1 and do the same for all groups of zeroes. By doing this, we obtain a string that has interleaving ones and zeroes and has the same c_{01} and c_{10} count as the starting string.

Because valid binary number starts with 1 we conclude that $c_{10} - c_{01}$ can be either 0 or 1. So, the numbers c_{10} and c_{01} tell us about number of groups of zeroes and ones in the binary number that satisfies given constraints. Number of different binary numbers that satisfy given constraints is the number of ways to insert 0 and 1 into corresponding groups of zeroes and ones that are mentioned before. Let $\#zeroesGroups$ be the number of groups of zeroes and $\#onesGroups$ the number of groups of ones. Then number of different binary numbers is equal to:

$$\binom{c_{00} + \#zeroesGroups - 1}{c_{00}} \binom{c_{11} + \#onesGroups - 1}{c_{11}}$$

Now we have figured out number of all binary strings that satisfy given constraints.

In order to solve general problem, let x have same number of digits as $digitsCount$.

Function $F(x)$ is implemented to count the number of binary strings that satisfy constraints for each prefix of x and then add them all up. We will smartly choose which prefixes to consider by iterating through A from the most significant bit and in case that the current bit is 1 we count the number of binary numbers that have all bits till the current bit the same as A and 0 in that position and it satisfies constraints.

For example if $x = 11001010$, we need to calculate all the numbers that satisfy constrains with prefixes: $10yyyyyy$, $11000yyy$, $1100100y$. There will be at most at most $O(N)$ prefixes, where N is the number of digits. For each prefix we can count two digit substring in the prefix in order to determine total count of numbers that satisfy constraints with given prefix. Since all groups of prefixes are distinct, we can get the result by summing all of them.

For calculating binomial coefficients we can precalculate factorials and inverse factorials using fast exponentiation. After that, for given n and k $\binom{n}{k}$ can be calulated in $O(1)$ time. Because of it, total time complexity of this solution is $O(N\log(MOD))$ where N is the number of digits.

Problem G: Space Isaac

Premier League and Rising Stars

Author:

Daniel Paleka

Implementation and analysis:

Balša Knežević
Ognjen Tošić
Daniel Paleka

Statement:

Everybody seems to think that the Martians are green, but it turns out they are metallic pink and fat. Ajs has two bags of distinct nonnegative integers. The bags are disjoint, and the union of the sets of numbers in the bags is $\{0, 1, \ldots, M-1\}$, for some positive integer M.

Ajs draws a number from the first bag and a number from the second bag, and then sums them modulo M.

What are the residues modulo M that Ajs **cannot** obtain with this action?

Input:

The first line contains two positive integers N and M, denoting the number of the elements in the first bag and the modulus, respectively.

The second line contains N nonnegative integers $a_1, a_2, \ldots, a_N$, the contents of the first bag.

Output:

In the first line, output the cardinality K of the set of residues modulo M which Ajs cannot obtain.

In the second line of the output, print K space-separated integers greater than zero and less than M, which represent the residues Ajs cannot obtain. **The outputs should be sorted in increasing order of magnitude**. If $K = 0$, do not output the second line.

Constraints:

- $1 \leq N \leq 200\,000$
- $N+1 \leq M \leq 1\,000\,000\,000$
- $0 \leq a_1 < a_2 < \ldots < a_N < M$

Example input 1:

```
2 5

3 4
```

Example output 1:

```
1

2
```

Explanation 1:

The first bag and the second bag contain $\{3, 4\}$ and $\{0, 1, 2\}$, respectively. Ajs can obtain every residue modulo 5 except the residue 2: $4 + 1 \equiv 0,\ \ 4 + 2 \equiv 1,\ \ 3 + 0 \equiv 3,\ \ 3 + 1 \equiv 4$ modulo 5.
One can check that there is no choice of elements from the first and the second bag which sum to 2 modulo 5.

Example input 2:

```
4 1000000000

5 25 125 625
```

Example output 2:

```
0
```

Explanation 2:

The contents of the first bag are $\{5, 25, 125, 625\}$, while the second bag contains all other nonnegative integers with at most 9 decimal digits. Every residue modulo $1\,000\,000\,000$ can be obtained as a sum of an element in the first bag and an element in the second bag.

Example input 3:

```
2 4

1 3
```

Example output 3:

```
2

0 2
```

Time and memory limit: 1.5s / 256MB

Solution and analysis:

Solution 1

Let's assume that we cannot obtain residue X modulo M.

Then if there is a number y in the first bag, there has to be a number $X - y$ modulo M, also. Otherwise, a magician could draw numbers y and $X - y$, and obtain number X

For our solution it is important to notice that if $y < X$, then $(X - y \text{ modulo } M) < X$. Also, if $y > X$, then $(X - y \text{ modulo } M) > X$.

The smallest member is paired with the largest member that is less or equal than X. The second smallest member is paired with the second largest member that is less or equal than X, etc.

The smallest member that is larger than X is paired with the greatest member of an array.

We need to find a boundary of the input array and check if all the pairs bring the same result.

How can we do that?

We can make another array b of $n - 1$ length, where $b_i = a_{i+1} - a_i$.

We can iterate by boundary and check if the both left and right side are palindrome. If they are, sum of any pair is the residue which magician cannot obtain. We can hash the input array and in $O(1)$ check if some consecutive subset is palindrome.

Time Complexity: $O(n)$.

Solution 2

Let A and B be the sets of numbers in the first and second bags, respectively. We need to find all the residues that are not in the sumset $A + B = \{a + b : a \in A, b \in B\}$. We start with the following lemma:

$$x \notin A + B \iff A = x - A,$$

where $x - A = \{x - a : a \in A\}$.

Proof: If $a \in A$ and $x \notin A + B$, then $x - a \in B$, for otherwise $a + (x - a) = x$ would be in $A + B$. That gives $x - A \subseteq A$, and symmetry gives the equality.

As in the previous solution, we consider the array of modulo M differences between the numbers a_i. If we denote $b_i = a_i - a_{i-1}$ (with $b_1 = a_1 - a_N$), we can state the following lemma:

$$A = x - A \implies \text{The string } b_N b_{N-1} \dots b_1 \text{ is a cyclic shift of the string } b_1 b_2 \dots b_N.$$

Proof: Left as exercise. Hint: try placing the numbers on a discrete circle, which Z/Z_M indeed is.

It is a well-known exercise to find all matches of a string in another string – one particularly nice way here is the Z-Algorithm, which we use in the second official solution: it is a standard trick to run it on the string $b_N b_{N-1} \dots b_1 (-1)\ b_1 b_2 \dots b_N b_1 b_2 \dots b_N$. (It will work because -1 doesn't appear among the b_i-s.)

Now, it's intuitive that each found match gives an unique x that we should output. With a little bit of math, one can prove a bijection:

$$b_N b_{N-1} \dots b_1 = \ b_j b_{j+1} \dots b_{j-1} \ <==> \ a_N + a_{j-1} \notin A + B.$$

We can keep track of the "overflow" point to avoid sorting, so the complexity here can be $O(N)$. Of course, we allowed sorting to pass.

Problem H: Interstellar battle

Premier League and Rising Stars

Author:

Ognjen Tošić

Implementation and analysis:

Kosta Grujčić

Daniel Paleka

Statement:

In the intergalactic empire Bubbledom there are N planets, of which some pairs are directly connected by two-way wormholes. There are $N-1$ wormholes. The wormholes are of extreme religious importance in Bubbledom, a set of planets in Bubbledom consider themselves one intergalactic kingdom if and only if any two planets in the set can reach each other by traversing the wormholes. You are given that Bubbledom is one kingdom. (In other words, the network of planets and wormholes is a tree.)

However, Bubbledom is facing a powerful enemy also possessing teleportation technology. The enemy attacks every night, and the government of Bubbledom retakes all the planets during the day. In a single attack, the enemy attacks every planet of Bubbledom at once, but some planets are more resilient than others. Planets are number $0,1,\dots,N-1$ and the planet i will fall with probability p_i. Before every night, (including the very first one) the government reinforces or weakens the defenses of a single planet.

The government of Bubbledom is interested in the following question: what is the expected number of intergalactic kingdoms Bubbledom will be split into, after a single enemy attack (before they get a chance to rebuild)? (In other words, you need to print the expected number of connected components after every attack.)

Input:

The first line contains one integer number N denoting the number of planets in Bubbledom (numbered from 0 to $N-1$).

The next line contains N different real numbers in the interval $[0,1]$, specified with 2 digits after the decimal point, denoting the probabilities that the corresponding planet will fall.

The next $N-1$ lines contain all the roads in Bubbledom, where a wormhole is specified by the two planets it connects.

The next line contains a positive integer Q, denoting the number of enemy attacks.

The next Q lines each contain a non-negative integer and a real number from interval $[0, 1]$, denoting the planet the government of Bubbledom decided to reinforce or weaken, along with the new probability that the planet will fall.

Output:

Output contains Q numbers, each of which represents the expected number of kingdoms that are left after each enemy attack. Your answers will be considered correct if their absolute or relative error does not exceed 10^{-4}.

Constraints:

- $1 \leq N \leq 10^5$
- $1 \leq Q \leq 10^5$

Example input:

```
5
0.50 0.29 0.49 0.95 0.83
2 3
0 3
3 4
2 1
3
4 0.66
1 0.69
0 0.36
```

Example output:

```
1.68040
1.48440
1.61740
```

```
Time and memory limit: 1s / 256 MB
```

Solution and analysis:

Let G be the tree representing the planets of Bubbledom. Suppose that the survival probability of a vertex v is $p(v)$. After the enemy strike and before the government rebuilds, what used to be Bubbledom will no longer be a tree, but a forest. If the forest has V vertices and E edges, it has $V - E$ connected components, i.e. new intergalactic kingdoms.

Hence the expected number of connected components is

$$\mathbb{E}[V - E] = \mathbb{E}[V] - \mathbb{E}[E] = \sum_{v \in G} p(v) - \sum_{(u,v) \in G} p(u)p(v)$$

by linearity of expectation (the second sum denotes the sum over all (u, v) such that u and v are connected in G). Now root the tree at any vertex. Let

$$f(v) = \sum_{u \text{ is a child of } v} p(u)$$

It is easy to see that

$$\mathbb{E}[\text{connected components}] = \sum_{v \in G} p(v) - \sum_{v \in G} p(v)f(v)$$

After the government changes $p(v)$, only $p(v)$ and $f\big(\text{parent}(v)\big)$ are changed, which can be easily updated in $O(1)$. The total time complexity of this algorithm is $O(N + Q)$.

Problem I: Hyperspace™ highways

Premier League and Rising Stars

Author:

Daniel Paleka

Implementation and analysis:

Ognjen Tošić

Daniel Paleka

Statement:

In an unspecified solar system, there are N planets. A space government company has recently hired space contractors to build M bidirectional Hyperspace™ highways, each connecting two different planets. The primary objective, which was to make sure that every planet can be reached from any other planet taking only Hyperspace™ highways, has been completely fulfilled. Unfortunately, lots of space contractors had friends and cousins in the Space Board of Directors of the company, so the company decided to do much more than just connecting all planets.

In order to make spending enormous amounts of space money for Hyperspace™ highways look neccessary, they decided to enforce a strict rule on the Hyperspace™ highway network: whenever there is a way to travel through some planets and return to the starting point without travelling through any planet twice, every pair of planets on the itinerary should be directly connected by a Hyperspace™ highway. In other words, the set of planets in every simple cycle induces a complete subgraph.

You are designing a Hyperspace™ navigational app, and the key technical problem you are facing is finding the minimal number of Hyperspace™ highways one needs to use to travel from planet A to planet B. As this problem is too easy for Bubble Cup, here is a harder task: your program needs to do it for Q pairs of planets.

Input:

The first line contains three positive integers N, M and Q, denoting the number of planets, the number of Hyperspace™ highways, and the number of queries, respectively.

Each of the following M lines contains a highway: highway i is given by two integers u_i and v_i, meaning the planets u_i and v_i are connected by a Hyperspace™ highway. It is guaranteed that the network of planets and Hyperspace™ highways forms a simple connected graph.

Each of the following Q lines contains a query: query j is given by two integers a_j and b_j, meaning we are interested in the minimal number of Hyperspace™ highways one needs to take to travel from planet a_j to planet b_j.

Output:

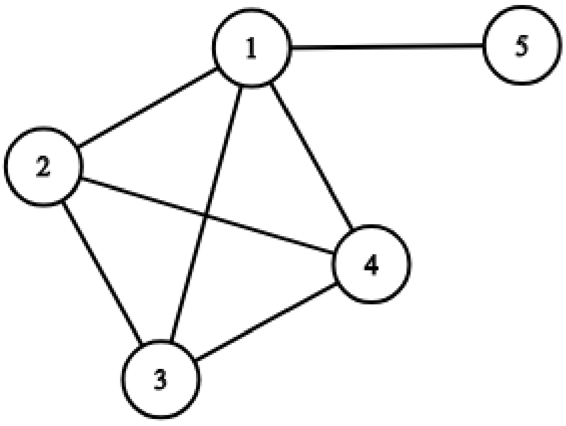
Output Q lines: the j-th line of output should contain the minimal number of Hyperspace™ highways one needs to take to travel from planet a_j to planet b_j.

Constraints:

- $1 \leq N \leq 100\,000$
- $1 \leq M \leq 500\,000$
- $1 \leq Q \leq 200\,000$
- $1 \leq a_j < b_j \leq N$
- $1 \leq u_i < v_i \leq N$

Example input 1:

```
5 7 2

1 2

1 3

1 4

2 3

2 4

3 4

1 5

1 4

2 5
```

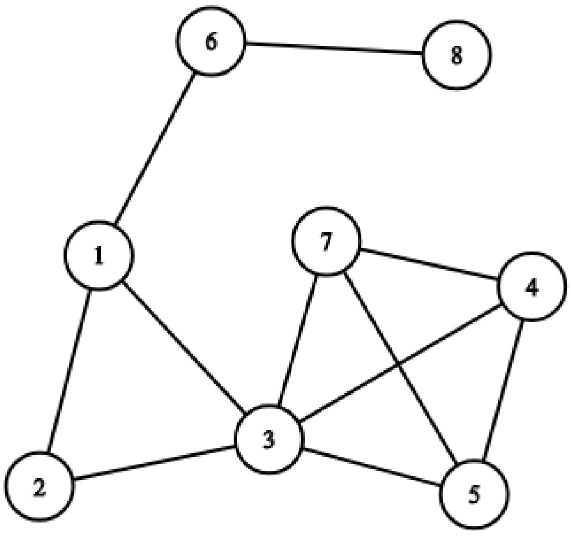


Example output 1:

```
1

2
```

Example input 2:

```
8 11 4

1 2

2 3

3 4

4 5

1 3

1 6

3 5

3 7

4 7

5 7

6 8

1 5

2 4

6 7

3 8
```



Example output 2:

```
2

2

3

3
```

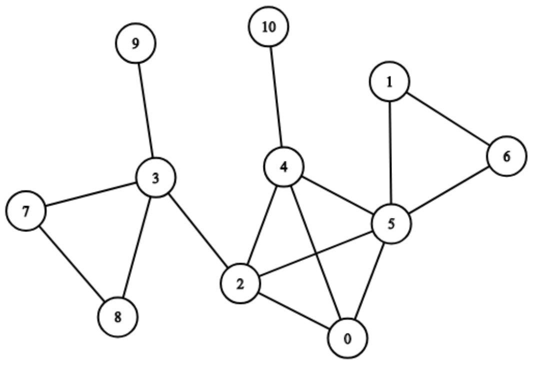
```
Time and memory limit: 1s / 256MB
```

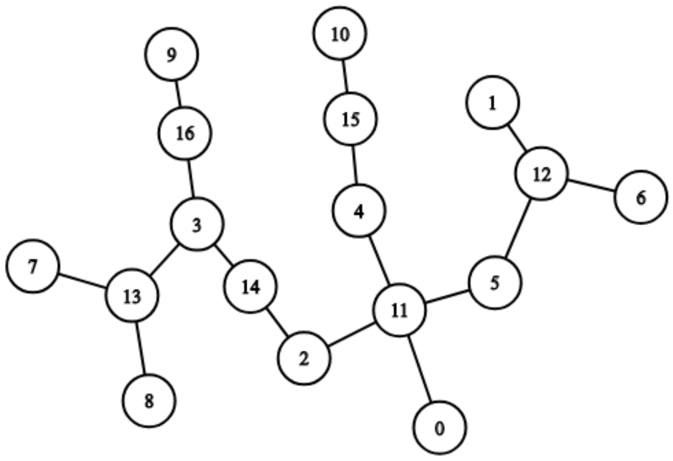
Solution and analysis:

Let $G = (V, E)$ be a graph where each simple cycle is a clique.

The idea is the following: for every maximal clique (in the sense that no other clique contains it), remove all the edges between its vertices, and add a single vertex that is adjacent to all of its vertices.

This is well-defined, since we can do this for one clique of the original graph at a time (and the order of cliques doesn't matter), because if two maximal cliques intersect in an edge, one is contained in the other. The graph obtained by applying this transformation as long as possible is a tree (if it had a cycle, the cycle would correspond to a cycle of the original graph, hence would be a clique, hence the vertices of the new graph wouldn't correspond to maximal cliques). Let the resulting graph be T'. For example, if the graph G is as in the first picture below, graph T' would be the second picture.





If $u, v \in G$, the claim is that $d_G(u,v) = \frac{1}{2} d_{T'}(u,v)$. This is easiest to see by considering the following operation on G: for each maximal clique C, remove the edges of C, and add a new vertex connected to each vertex in C, but with edges of weight $\frac{1}{2}$. Then this operation doesn't change lengths of shortest paths between vertices. Suppose that after applying this operation as many times as possible, we get a graph T''. But T'' can't have any edges of weight 1, since they are themselves cliques, so must be contained in some maximal clique. Hence all the edges in T'' are of weight $\frac{1}{2}$. Hence we can just remove the weights from the edges of T'' to obtain the unweighted graph T'.

We can easily prove that T' is, in fact, always a tree: a cycle in T' would induce a bigger clique in the original graph G. Also, note that the number of vertices in T' is $O(V)$. Now our solution splits in two parts: creating T' from G, and answering shortest path queries on the tree T'.

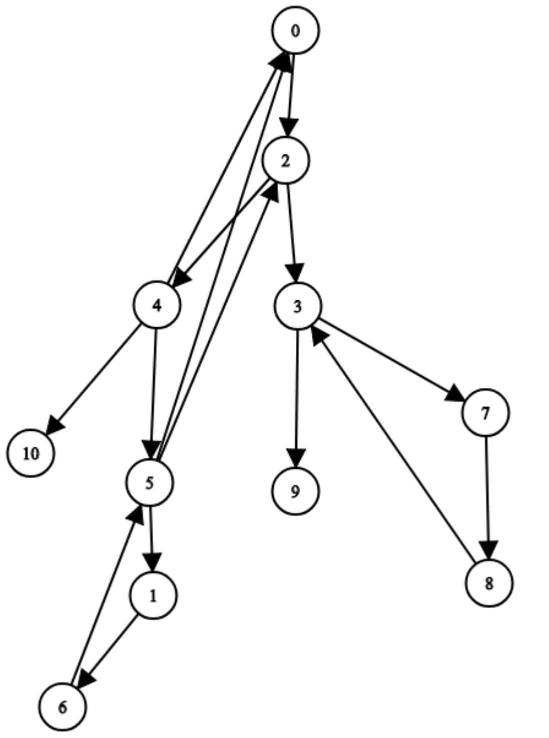
Let us first address the second part. Answering "the length of the shortest path from A to B in a tree" is the most frequent application of the lowest common ancestor (LCA) algorithms. In the official solution, we used the standard approach of preprocessing the tree in $O(V \log V)$ and answering every query in $O(\log V)$.

The problem is now reduced to finding an efficient construction of T'. Suppose T is the directed subtree of G generated by a depth-first search starting at any vertex $r \in G$. We need a few observations.

Observation 1: Consider any pair of adjacent vertices u, v. Either u is an ancestor of v or v is an ancestor of u in T. Suppose u is reached first in the depth-first search. Then v will either be a direct child of u or will be explored from a direct child of u, hence will be a descendant of u.

Observation 2: Let C be a set of pairwise adjacent vertices of G (i.e. a clique). Let u be a vertex in C of maximal depth. Suppose that $u \neq v \in C$. Then, because of observation 1, u must be an ancestor of v . Hence all the vertices of C lie on the path from r to u.

For the graph given above, the depth-first-search tree along with other edges would look like the picture below (the edges going upward are the edges of G but not in T).



Now direct the edges of G so that there is a directed edge $u \rightarrow v$ only when v is a descendant of u. The idea is to construct the tree T' recursively along T: we want to have, for each vertex $v \in G$ the solution of the problem for the induced subgraph on the descendants of v in T.

Define a function $solve(v)$ that is supposed to replace all maximal cliques in the induced subgraph of G on the descendants of v, and for each descendant of v find the value $f(v)$ defined as follows:

If v is not the vertex of the biggest depth in some clique, then $f(v) = -1$

Otherwise, $f(v)$ =the vertex of the smallest depth belonging to the clique where v is the vertex of the biggest depth.

Suppose that $v \in G$ and $v_1, \dots, v_k$ are the children of v in T, and that $solve(v_1), \dots, solve(v_k)$ have already been called. Then iterate over the edges of v in G, and check if the other end of that edge is the vertex of the biggest depth of some clique. If some vertex u adjacent to v is, this clique must contain v (since $u, v, \text{parent}(v), \text{parent}\big(\text{parent}(v)\big), \dots, u$ is a cycle). Hence $f(u)$ is currently one of the children of v. Mark that this child has been "visited", and set $f(u) = v$. Now iterate over all children v_i of v that were not "visited" and set $f(v_i) = v$. By observation 2, this must find all cliques that v belongs to.

This algorithm is $O(V + E)$ for the construction of T' (since each edge of G is processed at most once), $O(V \log V)$ for LCA preprocessing, and $O(\log V)$ per query due to a call to the lowest common ancestor function. Hence the total time complexity is $O(E + (Q + V)\log V)$.

Remark: It is also possible to construct T' by just finding maximal cliques a vertex belongs to, removing them and doing the same procedure on the connected components of the remaining graph. To find maximal cliques vertex v belongs to, just check for each neighbour u of v, all the common neighbours of u and v. This can be done in $O(\operatorname{deg}(u))$ if before that we just mark all the neighbours of v as currently active (this takes $O(\operatorname{deg}(v))$). After removing all the cliques v belongs to, in the resulting graph, there can't be any paths between some two (former) neighbours of v. Then just repeat this operation for each such neighbour. Since each neighbourhood of each vertex is searched at most twice, this runs in $O(E)$. Naive implementation of this is $O(E \log V)$ (because of edgde deletions) but by reusing the marks, it is possible to bring this down to $O(E)$.

Problem J: Ancient civilizations

Premier League and Rising Stars

Author:

Aleksandr Milanin

Implementation and analysis:

Ibragim Ismailov

David Milićević

Statement:

On the surface of a newly discovered planet which we model by a plane, explorers found remains of two civilizations in number of different locations. They would like to learn more about those civilizations and to explore the area they need to build roads between some of the locations. But as always, there are some restrictions:

1) Every two locations of the same civilization are connected by a unique path of roads
2) No two locations from different civilizations may have road between them (explorers don't want to accidentally mix civilizations they are currently exploring)
3) Roads must be **straight line segments**
4) Since intersections are expensive to build, they don't want any two roads to intersect (that is, only common point for any two roads may be at some of locations)

Obviously, all locations are different points in the plane, but explorers found out one more interesting information that may help you – no three locations lie on the same line!

Help explorers and find a solution for their problem, or report it is impossible.

Input:

In the first line, integer N – the number of locations discovered.

In next N lines, three integers x, y, c – coordinates of the location and number of civilization it belongs to (0 or 1).

Output:

In first line print number of roads that should be built.

In the following lines print pairs of locations (their **0-based** indices) that should be connected with a road.

If it is not possible to build roads such that all restrictions are met, print „Impossible“. You should not print the quotation marks.

Constraints:

- $1 \leq N \leq 10^3$
- $0 \leq x, y \leq 10^4$
- $c \in \{0, 1\}$

Example input:

```
5

0 0 1

1 0 0

0 1 0

1 1 1

3 2 0
```

Example output:

```
3

0 3

1 4

2 4
```

Explanation:

By connecting locations 0-3 all civilization-1 locations are connected, and by connecting locations 1-4 and 2-4 all civilization-0 locations are connected and there are no intersections.

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

Solution and analysis:

Let's call 0 points „white" (W) and 1 points „black" (B). What should be clear from the statement is that we need to create two trees of black and white color by connecting some of the nodes (locations) without any edges intersecting. Firstly, let's create a convex hull for all points. Then, there are three cases:

1. If black and white points alternate on the hull (if the circle of points consists of more than two color segments, like WBBBWWBBBW – 4 segments) it is obvious that we won't be able to make two trees without intersections, so print „Impossible".

2. If all points on convex hull are of the same color (let's say color X) then connect all but one consecutive pairs. After that find any Y colored point inside the hull and for each XXY triangle (where X points are consecutive on the hull) apply the following algorithm. Given two points of color X and one point of color Y check if there is any Y colored point inside the triangle. If there is not, just connect all X colored points in any way. Otherwise, pick random Y colored point inside the triangle. Then you have three triangles: XXY, YYX and YYX. Apply the same algorithm recursively on each of those.

3. If convex hull is like BB...BWW...W (it consists of two color segments) then connect all consecutive B and W pairs and create following triangles: All consecutive W pairs with first B and all consecutive B pairs with first W (respective to the convex hull orientation). Then run the algorithm described in case 2) recursively on each of those triangles.

Calculating convex hull is $O(NlogN)$. The other part of algorithm is trickier. We should notice that each point „generates" at most three triangles, which means there are $O(N)$ triangles. We search for the points inside triangles by having set of points not yet processed and trying to find if there is any point that lies inside the current triangle. For all triangles that sums up to $O(N^2)$. When we find a point inside the current triangle we remove it from the set, and for all points that is $O(NlogN)$. Based on that, overral complexity is $O(N^2)$.

Problem K: Last chance

Premier League division only

Author:

Filip Vesović

Implementation and analysis:

Filip Vesović

Kosta Grujčić

Statement:

It is the year 2969. 1000 years have passed from the moon landing. Meanwhile, the humanity colonized the Hyperspace™ and lived in harmony.

Until we realized that we were not alone.

Not too far away from the Earth, the massive fleet of aliens' spaceships is preparing to attack the Earth. For the first time in a while, the humanity is in real danger. Crisis and panic are everywhere. The scientists from all around the solar system have met and discussed the possible solutions. However, no progress has been made.

The Earth's last hope is YOU!

Fortunately, the Earth is equipped with very powerful defense systems made by MDCS. There are N aliens' spaceships which form the line. The defense system consists of three types of weapons:

- SQL rockets – every SQL rocket can destroy at most one spaceship in the given set.
- Cognition beams – every Cognition beam has an interval $[l, r]$ and can destroy at most one spaceship in that interval.
- OMG bazooka – every OMG bazooka has three possible targets, however, each bazooka can destroy either zero or exactly two spaceships. In addition, due to the smart targeting system, the sets of the three possible targets of any two different OMG bazookas are disjoint (that means that every ship is targeted with at most one OMG bazooka).

Your task is to make a plan of the attack which will destroy the largest possible number of spaceships. Every destroyed spaceship should be destroyed with exactly one weapon.

Input:

The first line contains two numbers. Number of your weapons N and number of spaceships M. In the next N lines, each line has one integer that represents type (either 0, 1 or 2). If the type is 0, then the weapon is SQL rocket, the rest of the line contains strictly positive number K and array k_i of K integers ($1 \le k_i \le M$). If the type is 1, then the weapon is Cognition beam, the rest of the line contains integers l and r. If the type is 2 then the weapon is OMG bazooka, the rest of the line contains distinct numbers a, b and c.

Output:

The first line should contain the maximum number of destroyed spaceships – X. In the next X lines, every line should contain two numbers A and B, where A is an index of the weapon and B is an index of the spaceship which was destroyed by the weapon A.

Constraints:

- $1 \leq N \leq 5000$
- $1 \leq M \leq 5000$
- $\sum K \leq 100\,000$
- $1 \leq k_i \leq M$
- $1 \leq l \leq r \leq M$
- $1 \leq a, b, c \leq M$

Example input:

```
3 5

0 1 4

2 5 4 1

1 1 4
```

Example output:

```
4

2 1

3 2

1 4

2 5
```

Explanation:

SQL rocket can destroy only 4th spaceship. OMG Bazooka can destroy two of 1st, 4th or 5th spaceship, and Cognition beam can destroy any spaceship from the interval [1,4].

The maximum number of destroyed spaceship is 4, and one possible plan is that SQL rocket should destroy 4th spaceship, OMG bazooka should destroy 1st and 5th spaceship and Cognition beam should destroy 2nd spaceship.

```
Time and memory limit: 2s / 128 MB
```

Solution and analysis:

This problem can be modeled as a maximum matching problem and can be solved using a standard flow algorithm. However, there are few details which we need to take into consideration.

First of all, we will create a bipartite graph with weapons in one set and spaceships in another. Every spaceship will be connected to the sink vertex with the edge of unit capacity. From the source, there will be an edge to every weapon with the unit capacity if the weapon is SQL rocket or Cognition beam and 2 if the weapon is OMG bazooka. Every SQL rocket will be connected with spaceships which it can destroy with an edge of unit capacity.

If we connect Cognitive beams with every spaceship they can destroy, that will result with $O(N \cdot M)$ edges and a solution will TLE or MLE. In order to reduce the number of edges for Cognition beams, we can construct the segment tree on the top of spaceships nodes and connect Cognition beams with the minimum number of nodes which will cover the whole interval. It's similar to querying a standard RMQ segment tree. Edges in segment tree have infinite capacity.

And finally, we need to connect every OMG bazooka with three possible targets. In addition, we need to ensure that all OMG bazookas destroy zero or two spaceships. After we find maximum flow for every OMG bazooka that has destroyed only one spaceship, we can easily find which other weapon has destroyed other two and assign one of them to the bazooka. That assignment is valid since there is no target overlaping between bazookas and total number of destroyed spacehips remains the same.

Overall time complexity is $O(\min(N, M) \cdot \max(N, M) \log M)$ and space complexity is $O(\max(N, M) \log M)$.

Problem L: Moonwalk challenge

Premier League division only

Author:

Ibragim Ismailov

Implementation and analysis:

Ibragim Ismailov

David Milićević

Statement:

Since astronauts from BubbleCup XI mission finished their mission on the Moon and are big fans of a famous singer, they decided to spend some fun time before returning to the Earth and hence created a so called "Moonwalk challenge" game.

Teams of astronauts are given the map of craters on the Moon and direct bidirectional paths from some craters to others that are safe for "Moonwalking". Each of those direct paths is colored in one color and there is a unique path between each two craters. Goal of the game is to find two craters such that given array of colors appears most times as continuous subarray on the path between those two craters (overlapping appearances should be counted).

To help your favorite team of astronauts win, you should make a program that, given the map, answers queries of the following type: For two craters and array of colors answer how many times given array appears as continuous subarray on the path from the first crater to the second.

Colors are represented as lowercase English alphabet letters.

Input:

In the first line, integer N – number of craters on the Moon. Craters are numerated with numbers 1 to N.

In next $N-1$ lines, three values u, v, L denoting that there is a direct path with color L between craters u and v.

Next line contains integer Q – number of queries.

Next Q lines contain three values u, v, S where u and v are two craters for which you should find how many times array of colors S (represented as a string) appears on the path from u to v.

Output:

For each query output one number that represents number of occurrences of array S on the path from u to v.

Constraints:

- $2 \leq N \leq 10^5$
- $1 \leq u, v \leq \mathrm{N}$
- $1 \leq Q \leq 10^5$
- $|S| \leq 100$

Example input:

```
5

1 2 a

1 3 b

1 4 a

4 5 c

4

2 4 a

2 5 aa

2 5 ac

3 5 c
```

Example output:

```
2

1

1

1
```

```
Time and memory limit: 6s / 256 MB
```

Solution and analysis:

The key idea behind this solution is to split the tree into chains using Heavy-Light Decomposition.

When the tree is decomposed, we notice that the query string length is at most 100. It means that for each chain in decomposition we can make dictionary containing all substrings of lengths from 1 to 100 in the chain as keys and positions of occurrences of those substrings in the chain as values.

There are few notes about building the dictionary. Firstly, since having strings as keys will take up too much memory, we should have their hash values as keys. Next, we can optimize dictionaries by adding only substrings that are equal to some string from the set of query strings, meaning we should solve queries offline afterwards. And lastly, indices of occurrences of substrings inside chains that are stored in the dictionary should be sorted in order to improve counting later on.

For each query we find Lowest Common Ancestor of the two given nodes and split solving into two parts: *(u to LCA)* and *(v to LCA)*. Notice that in one of the two parts we should search for the reverted query string since we are going up the tree and we built the dictionary by going down the tree. Both parts are solved in the same manner. For each chain on the path from *node* to *LCA* we are counting number of matching substrings that start in that chain. We do so by checking values in dictionary entries (using binary search to find range of appropriate starting indices inside the chain). Separately, we should consider strings that start in one chain, but end in some other chain. This problem is solved by taking last 100 characters from the chain and concatenating next 100 characters on the path to the *LCA* (because query string is at most 100 characters long). Also note that if chain is shorter than 100 characters then we should take needed characters from previous chains (if there are some) and so on… Now it is easy to find number of occurrences of query string in the string we built (by using KMP algorithm, for example). After both paths are processed, it remains to check for occurrences that contain *LCA* and that is done similarly. Take 100 adjacent characters from both paths and build a string of at most 200 characters. In that string search for occurrences of our query string (like we did earlier).

For Heavy-Light Decomposition complexity is $O(N)$. Saving query strings is done in $O(Q|S|)$ because for each string we need to hash it and save the hash value in some hash map. Building dictionary is $O(N|S|)$ because for each node we add at most $|S|$ strings to the dictionary and we can hash longer strings based on shorter ones. Sorting all dictionary entries is for sure less than $O(N|S|\log(N|S|))$ since that would be the complexity if all of the entries were stored in single array. So, the previous two sum up to $O(N|S|\log(N|S|))$. For each query finding *LCA* is $O(logN)$. Then, for each chain (we have at most $logN$ chaing on the path from *node* to *LCA*) there is $O(logN)$ for binary search for correct start indices in dictionary values and $O(|S|)$ for finding occurrences that end outside the current chain. Total complexity is then $O(Q|S| + N|S|\log(N|S|) + QlogN(logN + |S|))$.

Problem M: Shady Lady

Premier League division only

Author:

Daniel Paleka

Implementation and analysis:

Daniel Paleka

Aleksandar Lukac

Statement:

Ani and Borna are playing a short game on a two-variable polynomial. It's a special kind of a polynomial: the monomials are fixed, but all of its coefficients are fill-in-the-blanks dashes, e.g.

$$\_xy + \_x^4y^7 + \_x^8y^3 + \ldots$$

Borna will fill in the blanks with positive integers. He wants the polynomial to be bounded from below, i.e. his goal is to make sure there exists a real number M such that the value of the polynomial at any point is greater than M.

Ani is mischievous, and wants the polynomial to be unbounded. Along with stealing Borna's heart, she can also steal parts of polynomials. Ani is only a petty kind of thief, though: she can only steal **at most one** monomial from the polynomial before Borna fills in the blanks.

If Ani and Borna play their only moves optimally, who wins?

Input:

The first line contains a positive integer N, denoting the number of the terms in the starting special polynomial.

Each of the following N lines contains a description of a monomial: the k-th line contains two space-separated integers a_k and b_k which means that the starting polynomial has the term $\_x^{a_k}y^{b_k}$.

It is guaranteed that for $k \neq l$, either $a_k \neq a_l$ or $b_k \neq b_l$.

Output:

If Borna can always choose the coefficients such that the resulting polynomial is bounded from below, regardless of what monomial Ani steals, output „Borna". Else, output „Ani".

You shouldn't output the quotation marks.

Constraints:

- $2 \leq N \leq 200\,000$
- $0 \leq a_k, b_k \leq 1\,000\,000\,000$

Example input 1:

```
3

1 1

2 0

0 2
```

Example output 1:

```
Ani
```

Explanation 1:

The initial polynomial is $\_xy + \_x^2 + \_y^2$. If Ani steals the $\_y^2$ term, Borna is left with $\_xy + \_x^2$.

Whatever positive integers are written in the blanks, $y \to -\infty$ and $x := 1$ makes the whole expression go to negative infinity.

Example input 2:

```
4

0 0

0 1

0 2

0 8
```

Example output 2:

```
Borna
```

Explanation 2:

The initial polynomial is $\_1 + \_x + \_x^2 + \_x^8$. One can check that no matter what term Ani steals, Borna can always win.

```
Time and memory limit: 1s / 256MB
```

Solution and analysis:

Let's call a set of monomials (in other words, a polynomial with blanks for coefficients) boundable if the blanks can be filled with positive integers such that the resulting polynomial is bounded.

The problem can now be rephrased as follows: does there exist an element in a given set of monomials, which when removed, leaves a set that is not boundable?

To solve the problem, we use a geometric reinterpretation of the setting.

Define the Cartesian representation of a set of monomials to be a set of points in the Cartesian the coordinate plane, constructed as follows: $x^a y^b \to (a, b)$ for all monomials in the set.

Call a lattice point even if both of its coordinates are even, and odd otherwise.

The key lemma, which crushes the problem, is:

A set is boundable if and only if its Cartesian representation, together with the origin, has a convex hull with all vertices even.

Proof: We will first prove the backward direction, i.e. convex hull even implies boundable.

Each point $\boldsymbol{Q} = (a, b)$ that is not a vertex of the convex hull can be expressed as a weighted mean of the vertices of the convex hull:

$$\boldsymbol{Q} = \alpha_1 \boldsymbol{P_1} + \alpha_2 \boldsymbol{P_2} + \dots + \alpha_m \boldsymbol{P_m},$$

where $\boldsymbol{P_1}, \boldsymbol{P_2}, \dots, \boldsymbol{P_m}$ are the vertices of the convex hull, and $\alpha_1 + \alpha_2 + \dots + \alpha_m = 1$.

Now the inequality between weighted arithmetic and geometric means gives us:

$\alpha_1 x^{\boldsymbol{P_1}.x} y^{\boldsymbol{P_1}.y} + \alpha_2 x^{\boldsymbol{P_2}.x} y^{\boldsymbol{P_2}.y} + \dots + \alpha_m x^{\boldsymbol{P_m}.x} y^{\boldsymbol{P_m}.y} \geq |x^a y^b|$,

where we omitted the moduli on the right side because $\boldsymbol{P_i}$-s are even.

We claim that we can obtain a bounded polynomial by setting the coefficients to 1 for all monomials corresponding to the non-vertices of the convex hull, and setting the coefficients to the vertex-monomials to N, the total number of monomials. The boundedness follows by summing the inequalities for all points $\boldsymbol{Q}$.

Now we prove the forward direction by contradiction: assume there is an odd vertex $\boldsymbol{Q} = (a, b)$ of the convex hull. Without losing generality, we can assume a is odd.

It's well known one can choose a line which intersects the convex hull of points only in $\boldsymbol{Q}$. Let the normal vector to the line from the origin (which is unique, as $\boldsymbol{Q}$ is not the origin because a is odd) be (p, q).

We claim that putting $(x, y) = (-t^p, t^q)$ and sending $t \to \infty$ makes the monomial $x^a y^b$ corresponding to the point $\boldsymbol{Q}$ go to negative infinity at a rate faster than the growth of any other monomial.

That is true because each monomial $x^c y^d$ becomes $\pm t^{pc+qd}$, and the exponent in $-t^{pa+qb}$ is maximal among all exponents, because every exponent is now the length of the projection of the point vector to the normal vector of the line, and the point $\boldsymbol{Q}$ has the maximal projection due to the way we chose the line. Also, the exponent $pa + qb$ is positive, because in the other case $p \cdot 0 + q \cdot 0 > pa + qb$. End of proof.

The problem is now reduced to the question:

Given a set of points in the plane, can we remove a point (bar the origin) such that the convex hull of the remaining points has at least one odd vertex?

That problem can be solved with a smart modification of one of the standard convex hull algorithms. We will describe a more straightforward approach here: first find the non-strict convex hull of the set of points, and colour the points on the hull alternately black and white. (Don't colour the origin.) Now find the strict convex hulls of the uncoloured+white points, and the uncoloured+black points. If any of these convex hulls has an odd point, Ani wins, otherwise Borna wins.

We leave the proof of the algorithm to the reader.

There is also a second type of solution, which can be thought as a dual to the one previously described: for each point (a, b) as in the previous solution, draw the line $y = ax + b$. Now it can be seen that a set is boundable if and only if (bar some cases) the upper convex envelope contains only even lines, i.e. lines with both a and b even. One then proceeds similarly as in the first solution to find Ani's best move in $O(n)$ or $O(n \log(n))$.

This second solution, while being slightly harder to implement, is somewhat easier to come up with: when one substitutes $(x, y) = (-t^p, t^q)$ in the polynomial, the monomial $x^a y^b$ with the maximal $ap + bq$ becomes the dominant term. By a suitable transformation, the problem is now just a variant on the „convex hull trick" technique.

Qualification problems

Round 1: Big Snowfall

Statement:
Anders the cat has been hired to clean the snow of the streets of Heavy Metal City. He drives a cleaning machine of the well-known brand the Blue-White Tree, and when he gets his machine stuck, he removes the snow by using a shovel made of the best wood around extracted from Quad Segment Trees, and the best steel out of Manowar factories. His laziness doesn't allow him to work too much; therefore, he does not want to pass the same street more than once. He has a map of the neighborhood he needs to clean: it is mainly built of streets and intersections.

Due to the fierce traffic, the streets of Heavy Metal City can be traversed only in their original direction with the aim to avoid any accident. In addition to that, due to Anders' aim, the Mayor of the city is strongly thinking about stopping the traffic for a certain amount of time which will be long enough to clean the city, allowing Anders to traverse streets in both directions.

Given the number of intersections and the streets between them, tell Anders if he can clean all the streets without passing any of them more than once. In addition to that, you must tell him that if the traffic stopping is needed or not.

Input:
The first line contains an integer number $1 \leq T \leq 100$ representing the amount of cases. For each one:

- The first line contains two space-separated integer numbers $1 \leq N \leq 50$ and $0 \leq M \leq N * (N - 1)$: the amount of intersections and streets respectively. The intersections are conveniently numbered between 1 and N. And the cleaning machine can start and finish in arbitrary intersections.
- The following M lines contain two space-separated integer numbers A and B $(1 \leq A, B \leq N, A != B)$, to describe a street going from the intersection A to intersection B.

Output:
If Anders can clean the city normally as he wants, the output is "YES". If the traffic stopping is needed in order to complete Anders' aim, the output is "$TRAFFIC\ STOPPING\ NEEDED$". Otherwise the output is "$WAKE\ UP\ EARLIER$"

Example input:
```
3
2 2
1 2
2 1
4 3
1 2
1 3
1 4
3 3
1 2
1 3
2 3
```

Example output:
```
YES
WAKE UP EARLIER
TRAFFIC STOPPING NEEDED
```

Time and memory limit: 2s / 256 MB

Solution:

Let's interpret the problem as a graph and recall the concept of a Eulerian path/cycle.

A Eulerian cycle is a cyclic path (meaning it starts and ends at the same vertex) that passes through every edge exactly once (it can go through any vertex as much as it needs to). A Eulerian path is essentially the same thing sans the requirement of it being cyclic (although a Eulerian cycle is still a Eulerian path).

It's easy to see that the question is equivalent to testing whether the given directed graph has a Eulerian path, and if it does not, does it have one if we strip the directions from the edges. There is a well-known test for these two questions, which states the following:

Given an undirected graph, it admits a Eulerian cycle if and only if it's connected (the isolated vertices aren't taken into consideration here, as they are irrelevant to visiting the edges) and every vertex has an even degree.

Given a directed graph, it admits a Eulerian cycle if and only if it's connected (isolated vertices aren't taken into consideration here, either) and every vertex has an indegree equal to its outdegree.

We shall now present a short outline of the proof of the directed case (the undirected case easily follows analogously):

We shall induct on the number of edges in the graph G, the case of 0 edges being trivial.

Since every vertex which we don't ignore has an outdegree of at least 1, it is not a directed acyclic graph, hence it contains a cycle C. Now take the cycle out of the graph (we'll call this graph G/C). Since we have decreased the indegree and outdegree of every vertex by the same amount the "equal indegree and outdegree" condition holds true, and the graph is split into more (maybe 1) connected components that have this condition. Now take the cycle that traverses C and every time it reaches a new connected component in G/C it traverses its Eulerian cycle, which exists given the inductive hypothesis. This construction gives the Eulerian cycle for graph G.

These easily generalize into finding whether a Eulerian path exists, by adding a phantom edge between the starting and finishing vertices of the path and we get that we want to have exactly:
0 or 2 vertices of an odd degree, in the undirected case.

Every vertex having its indegree equal to its outdegree; or having exactly one vertex whose outdegree is larger than its indegree by exactly one, one vertex whose outdegree is smaller than its indegree by exactly one, and the rest have their indegree equal to its outdegree.

(in both cases the connectivity argument needs to hold true)

Using this theorem, using Depth-First-Search (for checking whether the graph is connected) and degree counting arguments, we can derive an easy algorithm implementing it in $O(n + m)$ time.

Problem source: COJ
Solution by:
Name: Pavle Martinovic

Round 1: Bono

Statement:
Kang and Kung are board games enthusiasts. However, they like to play only deterministic games, such as chess. Since there are only a few deterministic games, they decided to create a new one. This game is called 'Bono'.

The rule of Bono is simple. The board consist of a 3x3 grid. The game is turn-based for 2 players. On each turn, the current player must fill an empty cell with a piece of water spinach. Players may not move any water spinaches that have been placed. The board will destroy itself if there is a row or column or diagonal consisting of 3 pieces of water spinach. A player loses if he can't make a move in his turn.

Of course, the first player will always win the game if he plays optimally. That is why they created Bono v2. In Bono v2, the number of boards used in a game is N ($N \leq 1000$). On each turn, the current player must fill an empty cell on an undestroyed board with the same ruling as Bono. A player loses if he can't make a move in his turn.

Bono v2 is still too easy since if they both play optimally, player 1 will always win if N is odd, and player 2 will always win if N is even. To solve this matter, Bono v3 is created. In Bono v3, the initial state of each board might not be empty. Some cells might already be occupied with a piece of water spinach. Other than that, the rules are same as Bono v2.

Kang and Kung decide to play Bono v3 T times ($T \leq 1000$). Kang always moves first and they both play optimally. Who will win each game?

Input:
The first line of input is T, the number of games is ($T \leq 1000$). For each game, the first line is N, the number of boards ($N \leq 1000$). Next N lines consist of the starting boards. A board is represented with a 9-digit binary string. Cell in (r, c) position is represented by $((r-1)x3 + c)^{th}$ character in the string. 0 means the cell is empty, 1 means the cell is filled.

Output:
For each game, output a line containing the winner's name.

Example input:
```
3
1
000000000
2
000000000
000000000
2
100010000
001010000
```

Example output:

```
Kang
Kung
Kung
```

Time and memory limit: 2s / 256 MB

Solution:

Firstly, let's notice that this game is impartial (for a single board). An impartial game is such a game that both players have the perfect information, the same moveset in a given state and that the losing player is the one who can't make a move. This means that we can apply the Sprague-Grundy theorem to help us solve the problem.

The theorem tells us that every state in an impartial game (and as such every table in this game) is equivalent to a Nim heap of a certain size. Size of that heap for a certain state is often called that state's $Nimber$. $Nimber$ of a losing state is 0. We can find the $Nimber$ of a state using the formula given by the theorem: $Nimber[state_0] = mex\{Nimber[state_1], Nimber[state_2], \dots Nimber[state_N]\}$

where $state_1, \dots state_N$ are all the states you can get to from $state_0$ in a single move, and mex meaning minimum excludant (the smallest nonnegative integer not included in the set). We use recursion with memorization to find the $Nimbers$ for all possible states of the board. Complexity is $O(2^K \log K)$, where K is the number of spaces on the board. In this problem, $K = 9$.

Now that we know $Nimbers$ for all N boards and that those $Nimbers$ are equivalent to Nim heaps, we can just solve a regular Nim problem with N heaps, which has a well-known solution. Nim is a game with multiple coin heaps where players take turns, taking any amount of coins from a single heap of their choosing. The player who plays the first- wins (considering optimal play) if and only if the bitwise xor of all heap sizes is greater than zero.

Problem source: SPOJ
Solution by:
Name: Mladen Puzić

Round 1: Changu Mangu in a Football Team

Statement:
Changu and Mangu are a part of a football team which is going to participate in a tournament. There are n teams in total in the tournament. Each team plays twice against every other team (home and away fixture). The team that wins, is awarded 3 points. The team that draws, gets 1 point, while the team that loses gets no points.

At the end of the tournament, the teams are ranked 1 to n according to total points. The rank of each team t having p points is one plus the number of teams having more than p points. It is possible that more than one team have the same ranks.

In addition to the team that has rank 1, the $Lucky$ team is also awarded, if it exists. The $Lucky$ team is the one that has absolutely the highest number of wins (absolutely means that no other teams have the same number of wins), absolutely the highest number of goals scored, and absolutely the lowest number of goals conceded, is called the $Lucky$ team. ($Lucky$ Team should have all these properties.)

Changu keeps dreaming about being a part of the Lucky team. Your task is to find out the worst possible rank for the $Lucky$ Team.

Input:
The first line contains T, the number of test cases. The next T ($T \le 10^5$) lines contain a number n ($1 \le n \le 10^{18}$), the number of teams participating in the tournament.

Output:
For each test case, print on a separate line, the worst possible rank for the $Lucky$ Team

Example input:
```
2
1
3
```

Example output:
```
1
1
```

Time and memory limit: 2s / 256 MB

Solution:

Let's prove that if $n > 4$ the answer is n. The Lucky team will win once against teams A and B and they (both) will beat the Lucky team once in their second match. Every other team will beat the Lucky team once.

All other matches on the tournament will end as a draw. This way every team except teams A, B and the Lucky team will have $3 * 1 + 1 * (2n - 3) = 2n$ points.

Teams A and B will have one less so $2n - 1$, and the Lucky team will have $3 * 2 + (n - 3) = n + 3$,so for every $n > 4$the Lucky team is the last. The Lucky team has 2 wins and every other team only 1.

We can easily handle the condition about goals, let's say that the result of Lucky's team victories against A and B is $L: 0$, let’s say that every lost game of the Lucky team is by $1: 0$. Let every draw of the Lucky team be $0: 0$ and let every draw between teams that are not the Lucky team be by $2: 2$. Now the Lucky team gave $2 * L$ goals (L is some very large number) and conceded $n - 1$,every other team conceded at least $(2 * n - 3) * 2$ goals so all conditions are fulfilled since we can pick Lthat’s large enough.

If $n = 2$ it's obvious that the Lucky team must finish first.

If $n = 3$ and number of wins of the Lucky team is 1 it's obvious that the Lucky team finishes first (because all other games on the tournament ended as a draw).

If it's 2, other teams can get only one win so there are 3 more games where they can get max 3 points (all draws) so they can't have more then 6 pts.

Finally, if $n = 4$ we will prove that the answer is 2. Again, if the number of wins of the Lucky team is 1 he will obviously end as first.

If the number of wins is 3 or more it's same as case where $n = 3$ and number of wins is 2.

So, the only possible number of wins of the Lucky team where it won't finish first is 2. Lucky team will have 2 wins and the other teams will have in sum 3 wins, so from 6 matches the Lucky team will have at least 7 points. If the Lucky team beats the same team 2 times (let's say team C) he can lose only 2 times so it can have at least 8 points, which is the maximum number of points that the other 2 teams can have. So, it's obvious that the Lucky team must beat 2 different teams (let's say B and C) and they can't have as much points as he has and team A will have more points so Lucky team will finish second.

So finally, if $\boldsymbol{n} > \mathbf{5}$ answer is $\boldsymbol{n}$, if $\boldsymbol{n} = \mathbf{4}$ answer is $\mathbf{2}$, otherwise it's $\mathbf{1}$.

Problem source: SPOJ
Solution by:
Name: Jovan Pavlović

Round 1: Crazy LCP

Statement:
You are given an array of strings in this problem; these strings are given unique indexes from 1 to N (in the same order as in the input). Then you are given Q queries, each query consists of 2 integers L and R, to answer the query you need to find a pair of strings with different indexes in the range from L to R (inclusive), where the length of the longest common prefix for these 2 strings is the maximum among all other possible pairs.

Input:
Your program will be tested on one or more test cases. The first line of the input will be a single integer T $(1 \leq T \leq 20)$ representing the number of test cases. Followed by T test cases. Each test case starts with a line containing an integer N $(2 \leq N \leq 10^5)$ representing the number of strings followed by a line containing N non-empty strings of lower case English letters separated by a single space, representing the list of strings. The sum of lengths of the strings in each test case is not greater than 200,000.

Followed by a line containing an integer Q $(1 \leq Q \leq 10^5)$ representing the number of queries followed by Q lines, each line will contain 2 integers separated by a space, L R, which represent a query as described above $(1 \leq L < R \leq N)$.

Output:
For each query print a single line containing an integer which is the maximum length of the longest common prefix as described above.

Example input:
```
1
4
aab abc aac xba
3
2 3
1 3
3 4
```

Example output:
```
1
2
0
```

Time and memory limit: 2s / 256 MB

Solution:

The main idea is that we can find an answer to a query with binary search. Let's sort strings by lexicographical order, but also remember their indexes in the original array. Now we calculate the LCP of each two consecutive strings. As an example, add these values to set. Iterate over these values in an increasing order. Now observe that when we are at some value *v* we are going to have some mutually non-intersecting groups consisting of consecutive elements. In each of those groups, any two strings are going to have LCP of at least *v*. So, if we sort their indexes and pair consecutive, we have just found the nearest right for each of them so that they have LCP at least *v*. We can add these pairs to vector and sort them by left value. Now to check if the answer exists for some value, we can just check if there is a pair within range in the vector corresponding to that value. But we have our pairs sorted, so it is easier for us to check. We just need to find the position of a pair with at least L in left value with the lower bound. In addition to that, just check the minimum value over right values in pair in suffix that starts from this position. For example, we can do this with the segment tree.

Let's analyze the complexity of this approach now. Iterating through LCP values costs us $O(S\ logN)$ where S is sum of strings' lengths. That's due to the fact that each string cannot be in an iterator more times than its length. And for the query it is $O(\log V * \log N)$ where V is the number of different LCP values (SQRT(S)) which is easily provable using the $n * (n+1)/2$ formula. We can also easily prove why these are the only possible values for the solution by verifying the fact that LCP of strings at positions i and j in the sorted order is the minimum value over LCPs of consecutive strings, and the fact about our groups being consecutive is a consequence of this fact.

But we can do this even faster and without the advanced data structures like the segment tree, sparse table, etc...

Firstly, notice that we can just check the suffix. So, we can get this in $O(1)$. But there is still the lower bound. This is not a big problem if we sort the queries by L. So, we can just move the pointers to "delete" the strings that are never going to be checked (the ones with the index less than L). Thus, we can again simulate the lower bound and get $O(1)$.

Problem source: A2OJ
Solution by:
Name: Aleksa Miljković

Round 1: Find the Next Letter

Statement:
Anders the cat is playing with their party´s partners Vinagrito and Klaus, playing with letters this time; he has a list of N lowercase letters in an arbitrary order and not particular distribution. Letters are conveniently numbered between 1 and N. The game is simple; each time one of them selects some letter (their respective position) of the list, the other cats must find (if it exists) the first letter (their respective position), to the right in the given list which is greater than the selected letter.

Anders hates losing, so he needs your help to find the solution rapidly.

Input:
In the first line an integer number $T \leq 1000$ will be given corresponding to the number of test cases. The next T lines contains an integer number $1 \leq N \leq 10^5$ representing the number of letters in the list, followed by N lowercase letters as a whole string of size N which are the elements of the list itself. You can safely assume that the sum of the sizes of all the strings will not exceed 10^5. The first line contains T, the number of test cases.

Output:
For each case output a line with N space-separated integer numbers not greater than N representing the respective position of the solution for each letter in the list. You must print the solution for the first letter first, then the solution for the second letter, and so on. If there is no solution for some letter/position in the list, you must print -1 instead (note that the last number must be always -1).

Example input:
```
3
1 r
5 abcde
9 uprdesoft
```

Example output:
```
-1
2 3 4 5 -1
-1 3 6 5 6 9 9 9 -1
```

Time and memory limit: 2s / 256 MB

Solution:

For every test case we will have a stack in which we will keep a letter and its position. We just need to go through the list of letters and whenever we get to a new letter i, until the letter on top of the stack is greater than letter i, we need to remove that letter and set the solution for the removed letter to i. When we are finished, we just need to set the solution of all the letters that are left in the stack to -1.

Problem source: COJ
Solution by:
Name: Ivan Avirović

Round 1: Oil Skimming

Statement:

Thanks to a certain "green" resources company, there is a new profitable industry of oil skimming. There are large slicks of crude oil floating in the Gulf of Mexico just waiting to be scooped up by enterprising oil barons. One such oil baron has a special plane that can skim the surface of the water collecting oil on the water's surface. However, each scoop covers a 10m by 20m rectangle (going either east/west or north/south). It also requires that the rectangle is completely covered in oil, otherwise the product is contaminated by pure ocean water and thus unprofitable!

Given a map of an oil slick, the oil baron would like you to compute the maximum number of scoops that may be extracted. The map is an NxN grid where each cell represents a 10m square of water, and each cell is marked as either being covered in oil or pure water.

Input:

The input starts with an integer K $(1 \leq K \leq 100)$ indicating the number of cases. Each case starts with an integer N $(1 \leq N \leq 600)$ indicating the size of the square grid. Each of the following N lines contains N characters that represent the cells of a row in the grid. A character of '#' represents an oily cell, and a character of '.' represents a pure water cell.

Output:

For each case, one line should be produced, formatted exactly as follows: "Case X: M" where X is the case number (starting from 1) and M is the maximum number of scoops of oil that may be extracted.

Example input:

```
1
6
......
.##...
.##...
....#.
....##
......
```

Example output:

```
Case 1: 3
```

Time and memory limit: 2s / 256 MB

Solution:

We want to fill oil cells in the matrix with as many rectangles of the dimension 2×1 as possible. One oil cell can share a rectangle with only one neighboring oil cell from the same row or column.

If we color cells of matrix like on a chessboard (cell in position (i, j) is black if $(i + j) \% 2 = 0$ and white otherwise) every rectangle would be on exactly one black and one white square.

We can create bipartite graph of oil cells. The first partition consists of black squares, and second of white squares. There is an edge between every two neighboring oil cells with different color.

Then we should find maximum bipartite matching. Every pair of matched vertex represents one placed rectangle. Maximum matching is equal to the maximum number of rectangles that are possible to place.

Every vertex can be connected with most 4 other vertices so that there are not so many edges in the graph and we can use the Ford-Fulkerson algorithm to find the maximum bipartite matching.

Problem source: A2OJ
Solution by:
Name: Filip Ćosović

Round 1: Optimum Click

Statement:

An electronic apparatus consists of a display and two buttons S and M. When connecting the machine, zero is displayed. If the S key is pressed, the number on screen increases to 1 and if the M key is pressed, the number that is displayed is multiplied by n. Find the least amount of clicks needed to display a number k and the string formed by S and M with minimal sequence in which keys must be pressed to display k on the computer screen.

Input:

The input consists of lines with two numbers separated by a single space k $(1 \leq k \leq 10^{12})$ and n $(2 \leq n \leq 100)$. The inputs end with a line containing 0 0.

Output:

For each input line output the minimum amount of clicks needed to display k, separated by a space, the minimum string formed by S and M in the order in which you must press these keys to achieve objective. If you have more than one string to return, use the one which minimizes the number of M.

Example input:

```
4 3
0 0
```

Example output:

```
3 SMS
```

Time and memory limit: 2s / 256 MB

Solution:

We will first consider a dynamic programming solution. Let's $dp[i]$ denote minimum number of clicks to obtain number i starting from 0.

Then $dp[k]$ is the solution. It is easy to see that $dp[i]$ is equal to $dp[i-1]+1$ if i is not divisible by n. Otherwise, $dp[i]$ is $1 + minimum\ between\ dp[i-1]\ and\ dp[i/n]$. This approach requires $O(k)$ time and memory which is not acceptable for the given constraints.

We can avoid calculating all the values in dp, and use greedy approach.

We claim that $dp[i/n]$ is always lower than $dp[i-1]$ when i is divisible by n.

$dp[i*1] = 1 + \min(dp[i], dp[i*n-1]) =$

$1 + \min(dp[i], dp[(i-1)*n+n-1]) =$

$1 + \min(dp[i], dp[(i-1)*n]+n-1) =$

$1 + \min(dp[i], dp[i-1]+1+n-1) =$

$1 + \min(dp[i], dp[i-1]+n) =$

$1 + \min(dp[i], dp[i]-1+n]) =$

$1 + dp[i]\ (because\ n > 1)$

Now we know that there is a unique sequence that leads to value n in minimal amount of clicks. Initial value is k. When current value is divisible by n, 'M' is appended to the sequence and current value is divided by n. If it is not divisible, 'S' is appended. The algorithm ends when value becomes zero. Note that we need to reverse the string at the end.

There could be maximum $\log(k)$ 'M'-s in the sequence, and for each 'M' maximum of $n-1$ 'S'-s. Because of that, overall complexity is $O(n * \log(k))$.

Problem source: COJ
Solution by:
Name: Andrijana Dejković

Round 1: Origami

Statement:

You would like to mail to your mom some origami you have made.

The price of mailing depends on the area of the envelope: the smaller the envelope area, the less cost to ship. You cannot fold the origami shape to make it smaller. Of course, the envelope you are shipping the origami in must be rectangular.

Consider the vertices which represent the points along the boundary of the paper in order, such that the edge of the paper may fold over itself. Given the vertices describing the origami shape, what is the area of the smallest envelope that you can use to mail the origami?

Input:

The first line contains the integer N $(3 \leq N \leq 100\,000)$ which is the number of vertices describing your origami. The next N lines contain two integers, x y, the x-coordinate and y-coordinate of that particular vertex $0 \leq x \leq 10^7$; $0 \leq y \leq 10^7$. You should assume all vertices are distinct, and that there is no line which contains all vertices.

Output:

Output the area of the smallest envelope that will contain the origami, rounded to the nearest integer. You can assume that no test case will have the area of the smallest envelope containing the given vertices that have a fractional part between 0.49 and 0.51.

Example input:

```
6
4 9
8 13
8 9
0 13
4 0
0 3
```

Example output:

```
104
```

Time and memory limit: 2s / 256 MB

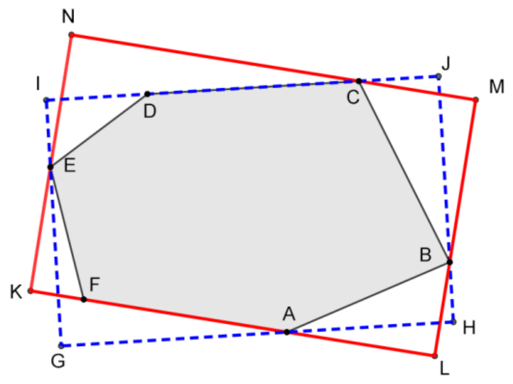
Solution:

First of all, we should remove every vertex from set of vertices of polygon that isn't contained in convex hull of that polygon, because if all vertices from convex hull are contained in some rectangle, then obviously all given points are also contained in that rectangle.

The key observation that is used in this solution is that the minimum area rectangle containing all of the vertices must have one of its edges coincident to one of the edges of the polygon. Using this observation, we can calculate area of every rectangle that has stated property.

Here is the brute-force approach for this problem: for each edge of polygon, we calculate the area of minimal-area rectangle that has one of its edges (call it bottom edge) coincident to that edge of the polygon. This is being done by scanning through all vertices of the polygon and finding extreme vertices that will determine the rectangle (in other words, we are looking for the leftmost, the rightmost and the uppermost vertex in coordinate system rotated so that x-axis is parallel to the bottom edge). Time complexity of this solution is $O(n^2)$, because for each edge of the convex hull we iterate through all vertices and look for the three remaining vertices needed to determine the size of rectangle.

Time complexity of $O(n)$ can be accomplished by using *rotating calipers* approach. Instead of iterating through all remaining vertices of convex hull when looking for extreme vertices that determine the rectangle, we can instead rotate some rectangle by minimal angle such that one of its edges becomes coincident to some other convex hull edge.



For each rectangle that has one of its edges coincident to one of the edges of convex hull, we define set of supporting points as set of four points of convex hull where each of those points touches one of the edges of the rectangle (if 2 vertices touch the same edge of the rectangle, we take one that appears later when iterating through vertices counterclockwise). In example provided here, for rectangle $KLMN$, set of supporting points is $\{A, B, C, E\}$. When we rotate this rectangle counterclockwise keeping the set of supporting points the same, the next rectangle that will have one of its edges coincident to one of the edges of convex hull is rectangle $GHJI$, and the corresponding set of supporting points will be $\{D, E, A, B\}$. We notice that when we rotate some rectangle by minimal angle with stated constrains, the set of supporting points will differ by only one vertex.

Using this observation, the solution becomes much more efficient. At the beginning, we compute the initial rectangle whose one edge is coincident to some arbitrarily chosen edge of polygon. This is being done by using idea from above mentioned brute-force approach. Then, once we have the initial rectangle computed, at each step, we compute all angles that the edges of the rectangle form with the corresponding edges of the polygon. We choose the smallest of those angles and rotate the rectangle by that angle. This rotation is being done in $O(1)$ time complexity. After each rotation, new edge of polygon will become coincident to an edge of rectangle, and after n rotations, the area of smallest rectangle for every edge of polygon will be calculated. Of all those rectangles, the one with smallest area is the one we are looking for.

Problem source: DMOJ
Solution by:
Name: David Milinković

Round 1: Real Phobia

Statement:
Bert is a programmer with a real fear of the floating-point arithmetic. Bert has quite successfully used rational numbers to write his programs, but he does not like it when the denominator grows large.

Your task is to help Bert by writing a program that decreases the denominator of a rational number, whilst introducing the smallest error possible. For a rational number A/B, where $B > 2$ and $0 < A < B$, your program needs to identify a rational number C/D such that all of the following are true:

- $0 < C < D < B$
- the error $|A/B - C/D|$ is the minimum over all possible values of C and D
- D is the smallest such positive integer

Input:
The input starts with an integer K $(1 \leq K \leq 1000)$ that represents the number of cases on a line by itself. Each of the following K lines describe one of the cases and consists of a fraction formatted as two integers, A and B, separated by "/" such that:

- B is a 32 bit integer strictly greater than 2, and
- $0 < A < B$

Output:
For each case, the output consists of a fraction on a line by itself. The fraction should be formatted as two integers separated by "/".

Example input:
```
3
1/4
2/3
13/21
```

Example output:
```
1/3
1/2
8/13
```

Time and memory limit: 2s / 256 MB

Solution:

We will solve each query separately.

We can assume that a/b is given in the lowest terms because otherwise, the solution is trivial.
We want to find the nearest neighbor c/d of a/b such that $d < b$ and $\gcd(c, d) = 1$.

Let's assume that a/b < c/d. Consider c/d. It is the element after a/b in the Farey sequence F_b, which means that $cb - ad = 1$. In particular, $da + 1 = 0\ (mod\ b)$, i.e. $d = -1/a\ \ (mod\ b)$. So, let

$r = 1/a\ \ (mod\ b)$. We can calculate it using extended Euclidean algorithm.
Now make d as big as possible, such that $d = -r\ (mod\ b)$ and $d \leq b$. Then just put c = $(da + 1)/b$.

In the case $a/b >\ c/d$ we can calculate it in a similar way.

Problem source: A2OJ
Solution by:
Name: Stjepan Požgaj

Round 1: Rocks

Statement:

Nikita has a profound collection of rocks, all of which have names. In his spare time, Nikita loves to play with these rocks. He does this by arranging the rocks in a line. He occasionally adds more rocks to the end of the line. Unfortunately, he has so many rocks, that he often forgets which rocks he already added. He is not allowed to add a rock that is already in the line (because they're all unique!). He may also switch the position of two rocks. More importantly, the most fun part is finding the mass of a few consecutive rocks! Obviously, you get the mass of each rock from its name. The name of each rock is unique and consists only of lowercase letters. The longest name Nikita will assign a rock is 100 letters long. The mass of a rock is the sum of the letters in its name, where a = 1, b = 2, c = 3 ... z = 26. E.g. rock will have a mass of 47.

Input:

You will execute ($1 \leq C \leq 100\,000$) commands. There are 5 types of commands, in the following format:

- $A\ R$ - add rock R to the end of the line, R is a string – the name of the rock, output "$Can't\ add\ R$" if rock R already exists in the line.
- $S\ X\ Y$ - swap the position of rocks X and Y – X and Y are both strings – the name of the two rocks, it is guaranteed that both X and Y exist in the line.
- $M\ X\ Y$ - output the mass of rocks in between (inclusive) the rocks X and Y. Both the rocks are guaranteed to exist in the line.
- $R\ X\ Y$ - replace rock X with new rock Y – X and Y are both strings. X is guaranteed to exist in the line and Y is guaranteed to not exist in the line.
- N - output the number of rocks currently in the line.

The next C lines contain these 1 lined commands. There will be at most $1 \leq N \leq 10\,000$ rocks in the line at a time. The longest name Nikita will assign a rock is 100 letters long.

Output:

Output depends on the commands in input. See the input specification. All output for each command goes on its own separate line.

Example input:
```
12
A a
A b
A c
M a c
M b c
S a c
M b a
R c d
M d b
A c
A d
N
```

Example output:
```
6
5
3
6
Can't add d
4
```

Explanation:
After the first 3 commands, the rocks that exist in the line are a, b, c, in that order. The mass of a to c is a + b + c = 1 + 2 + 3 = 6. The mass of b, c is 2 + 3 = 5. The position of a and b are swapped so that the line is now c, b, a. The mass of b and a is 2 + 1 = 3. Rock c is taken out and replaced with rock d so the line becomes d, b, a. The mass of d and b is 4 + 2 = 6. Rock c gets added successfully into the line, which is now d, b, a, c. Rock d can't get added since it already exists. Finally, there are 4 rocks in the line at the end.

Time and memory limit: 2s / 256 MB

Solution:

To solve this problem, we need to somehow maintain the order in which rocks are placed in line. Let's create associative array $Position[rock\_name]$ in which we'll store position of every rock already placed in the line. Also, let's create dynamic array $Mass[rock\_position]$ in which we will store for every position in the line mass of rock on this position.

Now let's describe how to perform all five types of queries using these two arrays.

1) "Add rock to the end of line". This one is quite easy: just check whether new rock R is present in associative array Position and depending on it either add R to this array and extend Mass by one element, or print "Can't add R".

2) "Swap positions of rocks X and Y". We need to swap values $Mass\big[Position[X]\big]$ and $Mass\big[Position[Y]\big]$, then swap values $Position[X]$ and $Position[Y]$.

3) "Output the mass of rocks in between (inclusive) the rocks X and Y". To be able to perform this type of queries one needs to quickly compute sum of values in some subsegment of array Mass. This is standard task called *Range Sum Query* and it can be solved by using many different techniques: *Fenwick trees*, *Segment trees* or *SQRT-decomposition*. To get bounds for range sum query we need to examine values $Position[X]$ and $Position[Y]$.

4) "Replace rock X with new rock Y". Just assign $Mass\big[Position[X]\big] = Mass(Y), Position(Y) = Position[X]$ and remove key X from associative array.

5) "Output the number of rocks currently in the line" - output size of array Position.

We can use various data structures to represent associative array Position. It can be a *Trie* (lookup time is $O(|S|)$, where S is a lookup key), some *Balanced Search Tree* (lookup time $O(|S| \log N)$ or *Hash Table* (lookup time is $O(1)$ on average and $O(Sum_i|S_i|)$ to precompute hashes of all strings in input).

Problem source: DMOJ
Solution by:
Name: Andrei Valchok

Round 1: Fire Evacuation Plan

Statement:

At RHHS, the school is on fire every other day, because safety is our number one priority. After administration has decided enough was enough, they decided to place detailed evacuation plans in each and every classroom to ensure everyone's safety during the fire evacuations.

An evacuation plan consists of a list of movements in the cardinal directions (North, South, East, West), which detail the exact movements a student must make in order to make it to safety. One such evacuation plan may be NNWS, which describes an evacuation plan where the student must move one meter north, one meter north, one meter west, and then one meter south in order to reach safety.

However, the administration has decided that these evacuation plans were too easy to follow and did not foster the academic atmosphere the RHHS is so famously known for having. As a result, the administration decided to encode the evacuation plans with a different encoding for each room. With this encoding, each cardinal direction is replaced with a string of letters, and the final evacuation plan is the concatenation of this string of letters. For example, the following mapping represents a possible encoding:

N → AA

S → A

E → B

W → AB

With this encoding, our original evacuation plan of NNWS becomes AAAAABA.

A side effect of this encoding is that it may represent multiple possible original evacuation plans. With the above encoding, AAAAABA could not only represent NNWS, but also SSSSSES.

Given an encoding and an evacuation plan, determine the number of possible distinct destinations the evacuation plan could lead to.

Input:

The first four lines will contain the encoding for north, south, east, and west respectively. Each encoding will not exceed 128 characters. The next line will contain an evacuation plan encoded with the given encoding, of length L $(1 \leq L \leq 2500)$

Output:

Output a single integer representing the number of possible distinct destinations the evacuation plan may lead to.

Example input:

```
AA
A
B
AB
AAAAABA
```

Example output:

```
6
```

Time and memory limit: 2s / 256 MB

Solution:

First, we hash the strings that represent north, south, east and west (cardinal directions).

Then we hash string L using the following formula:

$$h[i] = \left(L_1 p^0 + L_2 p^1 + \ldots + L_i p^{i-1}\right) \mod M$$

Where p and M are prime numbers. For example: $p = 31$ and $M = 10^9 + 7$. Array h contains hash values of every prefix of L.

For matching the strings, we will use the Rabin-Karp algorithm. Destinations are represented by the Cartesian coordinate system.

Using dynamic programming, we will go over the string L and for every position i calculate the number of distinct destinations. We will also have for each position i a set or an unordered set in which we will store all possible distinct destination (pairs of coordinates). The reason we use the term set is because it can't have any duplicates.

We will call this array of sets S. $S[0]$ contains only pair $(0,0)$.

The program does the following things: For each position i it checks for every cardinal direction, if it matches to any suffix of this prefix of L (which ends at position i). If it matches and the starting position of that suffix is j then we shall insert in $S[i]$ every destination that is present in $S[j-1]$ but changed by 1 (it depends which cardinal direction is in question). In the end, answer is the size of $S[size(L)]$.

Problem source: DMOJ
Solution by:
Name: Aleksandar Maksimović

Round 1: PIEK

Statement:

Ms. Magda likes cookies very much. She decided to organize an expedition during summer which would consist of tasting pastries from each bakery. Our heroine has been thinking about finding the track length that's as minimal as possible; which passes through each bakery exactly once and returns to the start point. She managed to get the table of distances between every two bakeries. You, as a good friend of Magda's, decided to help her with the problem and here is our task: you have to write a program, which calculates the minimal length of the track in exchange for cookies. The shorter your track is, the more cookies you get, and Magda is more satisfied.

Example:

We've got 4 bakeries: 1, 2, 3, 4. The table of distances looks as follows:

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

The shortest length equals to 18. An example way how the track may look: $1 -> 4 -> 3 -> 2 -> 1$

Input

In the first line the number of bakeries is n. Next, you should follow n lines, each consisted of

n $(n \leq 400)$ numbers (which are the distances between bakeries). Distance d between any two bakeries a and b is always between 0 and 100 000 $(0 \leq d(a,b) = d(b,a) \leq 100\,000)$.

Output:

In the first line the calculated length of your track. In the second line $n + 1$ numbers separated by whitespaces (from 1 to n including) where the first and the last must be the same, it's the order of visiting bakeries.

Example input:

```
4
0 4 7 3
4 0 5 8
7 5 0 6
3 8 6 0
```

Example output:

```
18
1 4 3 2 1
```

Score:

It's the number of cookies that Magda gives to you, she will give you more if the track is shorter.

Exact formula per test case is $[x/(length - y)]$, where the $length$ is your answer, while x and y are Magda's constants per test case. Your score will be considered wrong if $length > x + y$. Constants per test case are created such to allow a wide variety of solutions.

Time and memory limit: 1s / 256 MB

Solution:

After reading the problem statement, it is clear that it asks for solving the, quite standard, Travelling Salesman Problem (https://en.wikipedia.org/wiki/Travelling_salesman_problem). As this problem is NP-complete, it cannot be solved exactly, and we must rely on approximate approaches.

The TSP problem can be approximated with various approximation ratios, however, only if we assume that the edge weights satisfy the triangle inequality. In PIEK, the edge weights can be arbitrary, and it can be easily proven that no approximation algorithm exists in that case. Hence, we must resort to heuristics, hoping that they will work well in practice, or at least on the data in the judge system.

One of the most useful ideas that work both for TSP with and without triangle inequality are processes which maintain a candidate solution, which is a valid tour, and perform improvement steps, in each decreasing the total length of the tour. Usually, it is possible to use such an approach to make use of the time limit to its fullest.

Let us fix some solution and consider a connected interval of the visited cities. If we reverse the order of the cities in that interval and keep the positions of all the other cities intact, we can see that only two edges that were previously used are no longer used, and that the other two edges are used in their place. In particular, given an interval to reverse, in $O(1)$ time we can compute the length of the tour after the change.

With the abovementioned idea in mind, I will present the first algorithm present in my solution, which is the so-called Lin-Kernighan heuristic. In each phase, the algorithm starts with some candidate solution and ends which a solution that is at least as good as the initial one. Each phase works as follows. First, we randomly rotate the tour. Then, we consider prefixes of the rotated tour and compute what would be the length of the tour if a given prefix was reversed. Here we can either check all prefixes, or only some of them, chosen heuristically. Now, we take the prefix which results in the shortest possible tour after the reversal and reverse it. However, it is important that we perform that move even if it increases the length of the tour. Then, we continue to consider the prefixes of the tour after the change. After some number of iterations, say 15, we end up with a sequence of tours, with not necessarily decreasing lengths. Now, we pick the shortest one (which can also be the initial tour), and this tour is our new candidate solution after the end of the phase. Within a single phase, some care should be taken to ensure that the process does not "flip" between just two solutions and does some exploration of other solutions instead. Some speedup can be attained by using a balanced binary search tree to store the tour (treap, splay etc.) – then interval reversals can be executed in $O(\log N)$ time – although with the length of the tour being only 400 it is not necessary to get an acceptable performance. This concludes the description of the Lin-Kernighan heuristic, which my program performs in a loop as long as it still has time.

It is left to discuss what tour to supply as an initial solution to LK. One can start with a random tour (possibly a few different random tours), a tour created by some greedy process, or use a simple algorithm such as Christofides' algorithm. Each of those propositions is viable, and will likely converge to a sensible tour, after sufficiently many LK iterations. A better starting tour will often lead to a better final solution, although this is not a strict rule. Intuitively, solutions that are "locally optimal", and are only "globally entangled" in a few places, will be easier to be "uncross" for LK, than solutions that contain many "local" errors. This intuition stems from the fact, that many iterations of LK would be needed in order to fix many, arbitrarily placed, "local" errors.

My solution uses a rather sophisticated process to select the initial solution. This algorithm has a theoretical basis in a Lagrange relaxation of the linear program formulation of TSP, however, understanding that basis is by no means necessary to understand the resulting algorithm. I will describe the algorithm in the following paragraph.

First, take the minimum spanning tree of the given graph. MST's are present in many algorithms for the TSP – indeed, the correspondence between the two is clear: a TSP tour can be thought of as a specialized spanning tree (after adding one additional edge).

If we imagine that our MST has many vertices of degree 2, then we can split it into a small number of paths, those paths will form "reasonable" parts to be used in a tour. Hence, we can greedily order those paths to get a short tour. Note that the intuition from one of the previous paragraphs holds: a path consisting of edges from our spanning tree is "locally good", and "the largest errors" are introduced when our greedy approach tries to glue the paths together, possibly using some expensive edges in the process. Thus, a tour of such form is a good candidate for an initial solution, and also a candidate that should work well with LK.

Of course, it will be rarely the case that many vertices of our MST have degree 2. Some of them will be leaves (degree 1), and some will have degree 3 or more. Intuitively, the former are incident to expensive edges, while the latter to cheap edges. We want to ensure that the minimum spanning trees will contain more edges incident to vertices that currently are leaves and fewer edges incident to vertices that currently have a degree 3 or more. If we increase the weights of all edges incident to some vertex by a fixed value, then we will likely decrease its degree in the subsequent minimum spanning trees, analogously we can increase the degree by decreasing the edge weights. Thus, we perform an iterative process of changing the weights and trying to "fix" the degrees of vertices, rebuilding the MST after each change. We can use the final obtained MST as a basis for creating our paths, and then the initial solution. More information on this, as well as other algorithms for TSP, can be found in the works of K. Helsgaun.

This concludes the description of my algorithm. To wrap up, note that there are many possible approaches to this problem, and it’s impossible to exactly compare them to each other without trying them out on the provided test data. Hence, many different solutions are possible. Well-established algorithms with provable guarantees in the metric case will often still perform well on the non-metric instances.

Problem source: SPOJ
Solution by:
Name: Krzysztof Maziarz

Round 2: NEO

Statement:
You are given an array a with n integer elements. You can divide it into several parts (may be 1), each part consisting of consecutive elements. The NEO value in that case is computed by: Sum of value of each part. Value of a part is the sum of all elements in this part multiple by its length.

Example: We have array: [2 3 -2 1]. If we divide it, so it looks like: [2 3] [-2 1], then NEO = (2 + 3) * 2 + (-2 + 1) * 2 = 10 - 2 = 8.

Because there are many ways to divide an array into several parts, we can get many different NEO values. Your task is to find the NEO with the maximum value.

Input:
First line: T (number of test cases, T$\leq$ 10) For each of testcase:

- First line: n $(1 \leq n \leq 10^5)$
- Second line: $a[1], a[2], \dots, a[n] (-10^6 \leq a[i] \leq 10^6)$

Output:
For each test case, print the maximum NEO value.

Example input:
```
1
4
1 2 -4 1
```

Example output:
```
3
```

Time and memory limit: 4s / 256 MB

Solution:

In the problem NEO we are given an array $A = [A_1, A_2, \dots, A_N]$. Now we want to divide the array into parts, so that each part consists of consecutive elements. The value of the part $[A_l, A_{l+1}, \dots, A_r]$ equals to $(A_l + A_{l+1} + \dots + A_r)(r - l + 1)$. The value of a division of the array equals to the sum of values of the parts. We want to find out what is the maximum possible value of a division of the array.

The problem looks like a simple and standard dynamic programming problem. But it's by far trickier than it looks at first sight. Let's assume that $A_0 = 0$. Now let's define the array of partial sums $S = [S_0, S_1, \dots, S_N]$ so that $S_i := A_0 + A_1 + \dots + A_i$. Let's apply a dynamic programming approach to the problem. Let F_i be the maximum possible value of a division of the array $[A_0, A_1, \dots, A_i]$ and let's assume that we already know all the values $F_0, F_1, \dots, F_{i-1}$. How can we compute F_i? It's pretty straightforward. Let us fix the last part. We know that the last element of the last part is i. Let's iterate over the last element of the previous part (and call it j). Then $F_i = max_{0 \le j < i}\{F_j + (S_i - S_j)(i - j)\}$. Now we've got an easy approach to solve the problem in $O(N^2)$. It looks like this approach doesn't have any chance to get accepted. It's true, the solution is too slow, but we can optimize it drastically with the following optimizations:

- We can use a subset of optimizations from here: https://codeforces.com/blog/entry/56101?#comment-398797
- It can be proven that if we have two consecutive elements x, y and $x \ge 0, y \ge 0$, then we can replace these two elements with the element $x + y$ (we also need to remember that the new element should be counted as several elements, not just one). Now we can assume that among every pair of consecutive elements, at least one of them is negative and elements have "weights" (remember, some elements are counted as several elements). We can use the dynamic programming approach to solve this problem as well (it's extremely similar to the one described above, so it's left as an exercise to the reader). The number of states can be reduced by this trick.

Unfortunately, these optimizations allow us reduce the hidden constant factor of our solution, not the complexity. But applying them was enough to get the solution accepted by the judge system. It's also possible to apply a kd-tree data structure (https://en.wikipedia.org/wiki/K-d_tree) to solve this problem, but it wasn't faster than the previously mentioned approach.

It's also worth noting that here (http://codeforces.com/blog/entry/59694) you can find a discussion about this problem and the possible approaches. It's possible to solve the problem in $O(N\ log^2 N)$ time, but the solution is extremely complicated, so we won't present it here (you can find it under the link).

Problem source: SPOJ
Solution by:
Name: Vladyslav Hlembotskyi

Round 2: Ada and Homework

Statement:
Ada the Ladybug came home with some difficult homework. Since she is a very skilled mathematician, she already deduced how to count the answer for N. Consider all numbers K (in range $2 \leq K \leq N$), for which it is true that $gcd(N, K) == 1$ and add $gcd(N, K-1)$ to sum. What is the sum?

More formally put, find: $\sum gcd(K-1, N)$, for $K \in [2, N]$ where $gcd(N, K) == 1$.

Anyway, the numbers are too large, so she can't do that without your help. Can you help her?

Input:
The first line contains $1 \leq T \leq 1000$, number of test-cases. Each of following T lines contains

$2 \leq N \leq 10^{18}$, number for which Ada wants the answer.

Output:
For each test case, print the sum of deduced formula.

Example input:
```
11
2
5
6
7
8
10
50
100
1000
524288
945406969379503350
```

Example output:
```
0
3
2
5
8
6
70
260
5400
4718592
13819669753990598336l0
```

Time and memory limit: 6s / 256 MB

Solution:

Let $f(n) = \sum_{2 \le i \le n, \gcd(n,i)=1} \gcd(n, i-1)$. Our task is to compute the value $f(n)$ for several (up to one thousand) values of n which may be as large as 10^{18}. First, we will show that $f(n) = \phi(n)d(n) - n$, where $\phi(n)$ is the totient function and $d(n)$ is the number of divisors of n. Then, we will give an efficient method to compute $\phi(n), d(n)$.

Part 1

Define $g(n) = f(n) + n = \sum_{1 \le i \le n, \gcd(n,i)=1} \gcd(n, i-1)$. This is true since $\gcd(n, 1) = 1$ and $\gcd(n, 0) = n$. Now we only need to show that $g(n) = \phi(n)d(n)$. We will do this by showing that $g(n)$ is a multiplicative function and that $g(p^k) = \phi(p^k)d(p^k) = (p-1)p^{k-1}(k+1)$, for all primes p and positive integers k. In number theory, a function f is said to be multiplicative if $f(nm) = f(n)f(m)$ for all coprime positive integers $n, m, \phi(n)$ and $d(n)$ are well known to be multiplicative.

Let's compute $g(p^k)$ according to the definition above. The value of $gcd(p^k, i)$ can only be one of the following: $\{1, p, p^2, \dots, p^k\}$. Let's count the number of occurrences of p^j in the sum for some $j, 0 < j < k$. Whenever $i-1$ is divisible by p^j, $gcd(p^k, i) = 1$. So, the value p^j will divide exactly p^{k-j} summands. Since each summand is a power of p, the number of summands which are exactly equal to p^j can be found by subtracting the number of summands divisible by p^{j+1}, so the number is $p^{k-j} + p^{k-j-1}$. So, the values $j, 0 < j < k$ contribute$\sum_{j=1}^{k-1} p^j(p^{k-j} - p^{k-j-1})$ to the total. The sum can easily be computed and is equal to $(p-1)p^{k-1}(k-1)$. The value $p^j, j = k$ appears exactly once (when $i = 1$) and contributes p^k to the sum. The values 1 appear whenever we have an adjacent pair $i, i+1$ which are both coprime with p^k, and, in any p consecutive values of i there are exactly $p-2$ such pairs. So $j = 0$ contributes $(p-2)p^{k-1}$ to the sum. Overall, the sum is equal to:

$$(p-1)\,p^{k-1}(k-1) + p^k + (p-2)\,p^{k-1}$$

$$= p^{k-1}((p-1)(k-1) + p + (p-2))$$

$$= p^{k-1}(pk - p - k + 1 + 2p - 2)$$

$$= p^{k-1}(pk + p - k - 1)$$

$$= (p-1)\,p^{k-1}(k+1)$$

as we supposed. All that remains now is to show that g is multiplicative. Let n, m be two coprime integers. Let δ be the map from E_{nm} to $E_n \times E_m$, where E_k is the set of all integers x between 0 and $k-1$ coprime with k, defined as $\delta(x) = (x \bmod n, x \bmod m)$. It can be shown that this map is well defined and is in fact a bijection. Since n, m are coprime, $\gcd(nm, x) = \gcd(n, x)\gcd(m, x)$ for all integers x. Let's expand $g(nm)$:

$$g(nm) = \sum_{i \in E_{nm}} \gcd(nm, i-1)$$

$$\sum_{i \in E_{nm}} \gcd(n, i-1)\gcd(m, i-1)$$

$$= \sum_{(x,y)\in E_n \times E_m} \gcd(n,\ \delta^{-1}(x,y)-1)\gcd(m,\ \delta^{-1}(x,y)-1)$$

$$= \sum_{(x,y)\in E_n \times E_m} \gcd(n, x-1)\gcd(m, y-1)$$

$$= \sum_{x \in E_n} \sum_{y \in E_m} \gcd(n, x-1)\gcd(m, y-1)$$

$$= \sum_{x \in E_n} \gcd(n, x-1) \sum_{y \in E_m} \gcd(m, y-1)$$

$$= g(n)g(m)$$

as we supposed. $g(n) = \phi(n)d(n)$ for all $n > 1$ follows from the base case (prime powers) and the multiplicativity of $\phi(n)d(n)$.

Part 2

There are simple closed form formulas for $\phi(n)$ and $d(n)$ based on the prime factorization of n. If $n = \prod p_i^{\alpha_i}$, where p_i are primes and α_i are positive integers:

- $\phi(n) = \prod \frac{p_i - 1}{p_i} p_i^{\alpha_i}$
- $d(n) = \prod(\alpha_i + 1)$

The hardest part is to find an efficient factorization algorithm which works well for numbers up to 10^{18}. For this, we can use Pollard's rho algorithm. This algorithm only works when its input is a composite number, so to obtain the factorization of a number we have to be able to stop when we reach a prime. This can be done efficiently using the Miller-Rabin primality test. Pollard's rho has expected time complexity of $O(n^{1/4})$ while the Miller-Rabin primality test has time complexity of $O(log^2 n)$ for each so-called witness. For numbers up to 10^{18}, there is a fixed list of 12 witnesses of primality. We have to call the Miller-Rabin test and Pollard's rho algorithm exactly once for every inner node of the factorization tree, and the Miller-Rabin test for every leaf. Since the factorization tree is a proper binary tree and it has no more than 59 leaves, it can have no more than 58 internal nodes. Informally, we can estimate the running time as $117 \cdot 12\ log^2 n + 58\ n^{1/4}$, which, for $n \leq 10^{18} \leq 1.92 \times 10^6$. In practice, caching the results of each factorization can greatly improve performance.

Remark

Both $\phi(n)$ and $d(n)$ are no greater than n, however, their product may not fit into an unsigned 64-bit integer. For this, we can implement bignums or use the builtin type __int128 which is supported by newer versions of GCC on 64-bit platforms.

Problem source: SPOJ
Solution by:
Name: Ivan Stošić

Round 2: Count the Graphs

Statement

First, let's define an undirected connected labeled graph, it's a graph with N nodes with a unique label for each node and some edges. There's no specific direction for each edge, and in addition to that, duplicate edges and edges from a node to itself aren't allowed. You can reach any other node from any node.

A bridge in such a graph is an edge which will, if we remove it, disconnect the graph (there will exist nodes which aren't reachable from each other).

In this problem you are given N and K, and your task is to count the number of different undirected connected labeled graphs with exactly N nodes and K bridges. Since that number can be huge, print it modulo M.

An edge is defined by using the labels of the nodes it connects, for example we can say (X, Y) is an edge between X and Y, also (Y, X) is considered the same edge (since it's undirected). Two graphs are considered different if there's an edge which exists in one of them but not in the other.

Input

Your program will be tested on one or more test cases. The first line of the input will be a single integer T $(1 \le T \le 30)$ representing the number of test cases. Followed by T test cases.

Each test case will be just one line containing 3 integers separated by a space, N $(1 \le N \le 50)$, K $(0 \le K < N)$ and M $(1 \le M \le 10^9)$, which are the numbers described in the statement. It's guaranteed that N will not be more than 25 in 80% of the test cases.

Output

For each test case, print a single line with the number of graphs as described above modulo M.

Example input

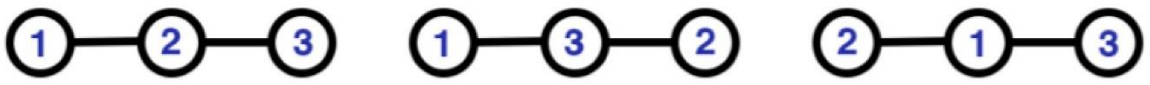
```
4
3 2 10
3 0 10
6 3 10000
6 3 1000
```

Example output

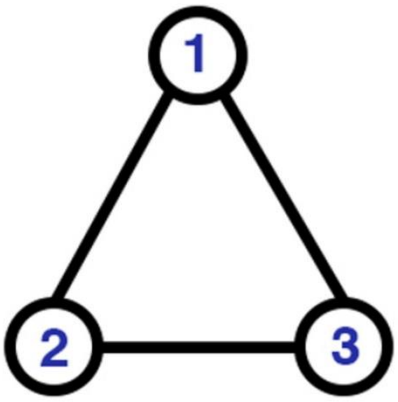
```
3
1
2160
160
```

Explanation

The following are the 3 graphs for the first test case:



The following is the only graph for the second test case:



Time and memory limit: 5 s / 256 MB

Solution

We're going to use dynamic programming to compute the number of graphs with n nodes and exactly k bridges for all values of n, k up to 50. We will store the result as bigints and take it modulo M only when a query arrives. So, the language of choice for this problem is Python since it has native bigint support. Let $d_{n,k}$ be the number of such graphs and let $k > 0$. We are going to count such graphs by observing one bridge. After removing this bridge, the graph becomes disconnected (by definition) and the two resulting components have n nodes and $k-1$ bridges in total. So, we're just going to iterate over all ways of partitioning the set of nodes into an ordered pair of sets of size a, b, respectively $(a+b=n, a, b>0)$. This can be done in $\binom{n}{a}$ ways. After that, we're going to choose the number of bridges in the first and the second component, let these numbers be equal to $i, j (i+j=k-1, i, j \geq 0)$, and, since we're using dynamic programming, we will already have computed the values $d_{a,i}, d_{b,j}$. Finally, we're going to choose two nodes, one from a and one from b where we will put our final bridge. Note that we will count each graph with k bridges exactly $2k$ times, once for each bridge and the factor of two comes from the fact that we're counting ordered partitions into two subsets. So, the final recurrence relation is:

$$dp_{n,k} = \frac{1}{2k} \sum_{1 \leq a,b; a+b=n; i+j=k-1; i,j \geq 0} \binom{n}{a} \cdot \mathrm{a} \cdot \mathrm{b} \cdot d_{a,i} \cdot d_{b,j}$$

Note that this only works for $k > 0$. For $k = 0$ we're going to have to use a different strategy. We can use that fact that $\sum_{k=0}^{n-1} d_{n,k}$ is equal to the number of connected graphs on n nodes. Let's call that value c_n, so, if we find a way to compute c_n, we can compute $d_{n,0}$ as $c_n - \sum_{k=1}^{n-1} d_{n,k}$. Let's find a recurrence relation for c_n. Assume we have computed c_i for $i < n$. Let's count the number of disconnected graphs on n nodes now instead (let that be y_n) and then use the fact that $c_n + y_n = 2^{\frac{n(n-1)}{2}}$. Let's observe the connected component of node n, let that be i. It can have any size from 1 to $n-1$. So, for each such i, the number of ways to choose nodes from the set $\{1, \ldots n-1\}$ which will be in the same connected component as n is $\binom{n-1}{i-1}$. Now, there are exactly c_i ways to add edges to this component to make it connected. We must not add any edges between these ii nodes and the rest of the graph. Finally, within the rest of the graph (the remaining $n-i$ nodes) we can add edges arbitrarily, in $2^{\frac{(n-i)(n-i-1)}{2}}$ ways. So, the recurrence relation for c_i is:

$$c_n = 2^{\frac{n(n-1)}{2}} - \sum_{i=1}^{n-1} \binom{n-1}{i-1} \cdot c_i \cdot 2^{\frac{(n-i)(n-i-1)}{2}}$$

Overall, the time complexity for precomputation is $O(n^4)$ bignum operations, plus one bignum modulo operation per query. The hidden constant is quite small, so this solution is more than fast enough to pass.

Problem source: A2OJ
Solution by:
Name: Ivan Stošić

Round 2: Fox Girls

Statement:
You woke up this morning and realized that some foxes have turned into girls! In the light of this exciting event, you have kindly invited Izuna, a fox-girl, over to your house (with the purest of intentions, of course). You have some anime that you want to share with Izuna, but your list of anime is very long. Therefore, you will create a strategy which consists of watching some anime with Izuna while she's at your house, and then leaving her with a list of recommendations.

For each anime on your list, you will recommend another anime from the list to Izuna, meaning that if she has watched one at your house, she should definitely watch the other sometime later. Being an obedient fox-girl, Izuna will always follow your recommendations.

Of course, you have to start watching some anime with Izuna at your house in order for her to follow all the recommendations and eventually watch them all, but since you are short on time, you need to determine the minimum number of minutes you spend watching anime at your house before Izuna can follow some of the recommendations and eventually watch all the anime from the list.

Input:
The first line of input will have N $(2 \le N \le 100\,000)$, the number of anime on your list, numbered from 1 to N. The second line of input will have N space-separated integers, the time it takes in minutes to watch the ith anime $(1 \le time_i \le 10^9)$. The third line of input will have N space-separated integers, meaning that you will recommend the i^{th} anime if Izuna watched the anime with the i^{th} number in this list at your house, and vice versa $(1 \le recommendationi \le N, recommendationi \ne i)$.

Output:
The first and only line of output should be the minimal number of minutes spent watching anime so that Izuna will eventually finish watching all the anime on your list.

Example input 1:
```
3
20 23 42
2 3 1
```

Example output 1:
```
20
```

Explanation 1:
By taking 20 minutes to watch the first anime, Izuna will definitely watch the other two anime due to the recommendations.

Example input 2:
```
4
```

```
100 23 23 24
2 3 1 1
```

Example output 2:

```
47
```

Explanation 2:

Although watching solely the first anime is enough for Izuna to watch the rest of the anime on her own, that will take 100 minutes. It is much more efficient to watch the fourth anime and either the second or the third anime at your house for a total of 47 minutes.

Time and memory limit: 2s / 256 MB

Solution 1:

First, let's reformulate the given problem in terms of graph theory. We're given n vertices and n undirected edges, and there is at least one edge going out of each vertex. Each vertex has a cost assigned to it (the time needed to watch the anime). Our task is to choose a subset of vertices, with the minimum possible cost, and mark all the vertices in this subset, so that each vertex is marked itself or there exists an edge from this vertex to some marked vertex. Let's call all vertices satisfying these criteria "good" (the vertices that are not good are called "bad").

The straightforward algorithm finds an optimal selection by checking every subset of vertices and returns a correct one with the least cost. Unfortunately, this solution works in $O(2^{n*n})$ time which is unacceptable, considering the input data limits.

In order to solve this task more effectively, we must observe a distinctive feature of the given graph. It consists of one or several connected components, each one being a tree with a cycle. This is the case because a connected graph with $n-1$ edges is a tree and adding one more edge creates a cycle in this graph.

Vertices that belong to different connected components obviously don't affect one another so we can calculate the answer for each component independently. The final answer will be the sum of the answers for all the connected components.

Now, let's analyze the problem for a single component. First, let's ignore the edges that belong to the cycle in this component. That is, we'll consider trees rooted in a vertex from the cycle.

For each vertex in the tree we want to calculate the following 3 values:

1. The minimum cost, so that all the vertices in its subtree are good and the vertex itself is bad.
2. The minimum cost, so that all the vertices in its subtree are good and the vertex itself is marked.
3. The minimum cost, so that all the vertices in its subtree and the vertex itself are good, but the vertex itself isn't marked (that means at least one of its children has to be marked).

We can easily calculate these values using depth-first search and simple dynamic programming over subtrees. One possible C++ implementation of this approach is attached below.

```
void dfs(int v, int parent){
	long long a1 = 0, a2 = 0, a3 = INF;

	for(auto it:Graph[v]){
		if (it != parent){ // it - vertex adjacent to v
			dfs(it,v);
			// we are looking for good, not marked vertices
			a1 += dp[it][3];
			// we are looking for the cheapest option because they are all
			// good (because v is marked)
			a2 += min(dp[it][1],min(dp[it][2],dp[it][3]));
			// we are looking for  the „cheapest” marked son
			a3 = min(a3,dp[it][2] - min(dp[it][2],dp[it][3]));
		}
	}
	dp[v][1] = a1;
	dp[v][2] = a2 + t[v]; // t[v] - cost of marking vertex v
	dp[v][3] = a1 + a3;

	return;
}
```

Let's get back to the cycle. We will simplify our problem once again and assume that we know what is going on with one arbitrary vertex from the cycle. As described above, there are only 3 possibilities
— it is either marked, covered by its marked neighbor on the cycle, or covered by its marked child in the tree rooted in it.

Having made such assumptions, we can theoretically forget about the edge connecting the chosen vertex with one of its neighbors and analyze the rest of the cycle as a path. Next, let's calculate the answer for this component by using dynamic programming along the path, which is very similar to that which we used before. The details are left as an exercise for the reader.

To sum up, we choose an arbitrary vertex from the cycle and we consider its three possible initial states. The rest of the cycle forms a path for which we can calculate the answer by using dynamic programming. The only question left is what to do with that "forgotten" edge connecting the vertex
to its other neighbor on the cycle. It turns out that one additional if statement in our algorithm is enough to solve this problem (depending on what's the initial state of the vertex we've assumed).

Complexity:

To separate the cycle vertices from the tree vertices we used the set from the STL — $O(n * \log n)$. Each tree vertex has been visited once and processed in constant time, which gives us $O(n)$ in this phase of the algorithm. Each cycle vertex has been processed in constant time which also gives us an $O(n)$ complexity.

The final complexity is $O(n * \log n) + n + n) = O(n * \log n)$, which easily fits the time limits.

Problem source: DMOJ
Solution by:
Name: Piotr Nawrot

Solution 2:

Firstly, we construct a graph with N nodes (representing N anime) and N edges (representing connections between every anime and the one that we recommend after watching it). Because there are only N edges, the graph can either be a tree or a graph that has exactly one cycle. Note that the graph can be made up of multiple connected components that all have the properties described above. Some terms I will use:

To watch a node – to take the time to watch the anime which that node represents at your house,
Recommended node – the anime that that node represents is recommended for Izuna to watch later,
$DP[i][2]$ – firstly, to call the function for node i and state 2 and then take the value from the DP table.

1. Let's first consider the case when the connected component has no cycle.

We can use the dynamic programing on trees to solve this case, we will do a recursive tree traversal (just like dfs) and fill up our DP table using memorization (we can imagine the tree as if it was rooted at the starting node of our traversal). Our DP table will have 3 different states for every node in the graph:

State 0: the last node that we visited isn't watched or recommended,
State 1: the last node that we visited is recommended,
State 2: the last node that we visited is watched.

$DP[i][0]$ will store the minimum amount of time required to watch/recommend all the nodes in the subtree of node i when the state of node i is, in this case, 0.

We have the following transitions:

From $state\ 0$, we only have the option to watch the current node. Then add the values of $DP[all\ children\ nodes][2]$ to the solution.

From $state\ 0$, we have the option to watch the current node and do the same as in $state\ 0$ (first solution) or not to watch the current node but instead watch one of its children nodes. To do this we need to calculate the values of $DP[all\ children\ nodes][1]$ and $DP[all\ children\ nodes][0]$. Then, for every child node, we find $DP[child\ node][0] - DP[child\ node][1]$. Take the minimum of these values (let's call that node $min\ node$), and our second solution is $DP[min\ node][0] + DP[all\ other\ nodes][1]$. Then we choose our final solution to be the minimum of these two solutions.

From $state\ 2$**,** we again have the option to watch the current node and do the same as in $state\ 0$ or not to watch the current node. The solution of the case when we don't watch the current node is just the sum of $DP[all\ children\ nodes][1]$. And we again take the minimum of these two solutions.

Our final solution is $DP[any\ node\ from\ the\ tree][1]$.

2. For the case when the connected component has a cycle, the solution is almost the same as case 1. just with a few changes.

First we need to find any 2 adjacent nodes in the cycle (let's call them $node\ 1$ and $node\ 2$) and delete the edge between (the graph becomes a tree), then we have the following cases:

We don't watch either of them: we can solve this case with the exact same solution as in the case 1. starting from any one of these 2 nodes with the state of 1 ($DP[node\ 1][1]$ or $DP[node\ 2][1]$),
We watch node1: To solve this we need to make it so if we ever get to $DP[node\ 2][1]$, we do $DP[node\ 2][2]$ instead ($node\ 2$ is already recommended by $node\ 1$). The final solution is $DP[node\ 1][1]$,
We watch node2: just like in the last case, add an exception that if we get to $DP[node\ 1][1]$, we do $DP[node\ 2][2]$ instead. The final solution is $DP[node\ 2][1]$.

Solution by:
Name: Nikola Pešić

Round 2: Alphabetic Rope

Statement:
The Alphabetic Rope is now available in the market. The Alphabetic Rope consists of the alphabetic characters in each stripe, which looks like a string.

You are given an Alphabetic rope, which consists of lowercase alphabetic characters only, you have to perform some operations on the rope and answer some queries within it.

There are 3 types of queries:

- $1\ X\ Y$: Cut the rope segment from X to Y and reverse it, then join at the front of the rope.
- $2\ X\ Y$: Cut the rope segment from X to Y and reverse it, then join at the back of the rope.
- $3\ Y$: Print on a new line of the Alphabet on Y^{th} position of the current rope.

Input:
There is only one input. The input begins with a single line giving the Alphabetic Rope as a string S. The next line containing Q, follows Q lines giving queries as mentioned above. (Index used are 0-based)

Output:
For each type 3 query, print a single character in a new line.

Example input:
```
gautambishal
5
1 3 5
3 0
3 3
2 2 4
3 9
```

Example output:
```
m
g
a
```

Constraints:
- $1 \leq |S| \leq 100\,000$
- $1 \leq Q \leq 100\,000$

Time and memory limit: 2s / 256 MB

Solution:

In this problem, we are given a string of length N and we should be able to handle Q queries of 3 types, two of them being:

Cut out a part of the rope, reverse it, and append it to the beginning/end of the remainder of the string.

The third type of the query we are to handle is to print k^{th} character at any given point.

Now, unfortunately, for this problem, the straightforward brute force $O(NQ)$ solutions accepted this verdict: the solution is to basically remove every element in the given segment and add them to the beginning/end of the string.

In more complexity, there is a pretty straightforward solution with the time complexity of $O(Q \log N)$ using the well-known data structure named Treap.

A very useful link: http://e-maxx.ru/algo/treap, which basically explains the Treap data structure and even provides a very good implementation of one. Using this data structure, we can literally cut the string (our original treap) in $O(\log N)$ time, to set a flag inside it as reversed too, and then to union the two treaps (with the given order, equivalent to concatenating two strings).

Now, to make all of this even more formal and clear, I can provide a short explanation

Assume that we are given these methods:

split(treap whole, treap &left, treap &right, int number), which splits the whole treap into two parts, left and right, with the fact that first cnt elements go into the left part, and all the other elements go into the right part

unite(treap &where, treap left, treap right), where we basically concatenate two strings (a treap represent a string)

reverse(treap &t), where we just reverse one string (it does so by setting some flags, but there is no point in explaining the stuff that's already explained in the link I've provided)

Now, using these three functions, it's easy to do all 3 operations in $O(\log N)$.

If you think I should elaborate some more on treap functions and stuff, I can do so, but I don't think it's necessary (I'd just copy paste them from the link above).

There is also a C++ built-in structure called rope, which supports string cuttings in same complexity as in the abovementioned solution, but I haven't found a way to reverse a string by using that notation and thus to solve the problem. If anybody did, maybe it'd be a nice extra solution to this problem.

Problem source: SPOJ
Solution by:
Name: Vladimir Milenković

Round 2: Winter Is Here

Statement:

Winter is here, and the great wall was destroyed. The night king and his army of dead control the North now, they attacked every place there, except the Castle of Winterfell, because it is protected from the army of the dead and the white walkers by some old magic.

The North can be represented as a directed rooted tree with N nodes (each node gets a unique ID from 1 to N), the nodes are connected by using directed edges, each edge represents a road (the roads can be traversed in just 1 way) with exactly 1 white walker protecting each road, the root of the tree will be node 1, and from the root it will always be possible to follow some roads to reach any other node. Winterfell is placed in some node with the ID v (not necessarily the root). Jon Snow and the rest of the surviving people who are in Winterfell heard that the night king is in some node with an ID in the range from L to R inclusive, and since Jon knows that there is no hope for them to win the war against the white walkers unless they kill the night king, he decided to go on a suicide mission and try to do that.

Daenerys, Tyrion and Sir Davos tried to convince him otherwise, but we all know Jon. When they lost all hope and realized that they will never stop him, Tyrion put together the following plan for him (given the values of v, L and R):

- Jon and Sir Jorah Mormont will go on this mission.
- Each one of them should choose a node with an ID in the range from L to R, that can be reached from v, and go to check that chosen node. Note that they can't choose the same node, but it's okay if any of them chooses v itself.
- For safety reasons, their paths from v to the chosen nodes shouldn't have any common road.
- To increase the profit of the mission, each of them should kill the white walker that protects the road they pass by.

Now, given some possible scenarios for v, L and R, can you find the optimal pair of nodes that they should choose to increase the total number of white walkers which they will kill?

Input:

Your program will be tested on one or more test cases. The first line of the input will be a single integer T $(1 \leq T \leq 10)$ representing the number of test cases. Followed by the T test cases.

Each test case starts with a line containing 2 integers separated by a space, N $(1 \leq N \leq 20{,}000)$ representing the number of nodes and q $(1 \leq q \leq 100{,}000)$ representing the number of scenarios.

Followed by a line which contains $N - 1$ space separated integers $p1, p2, \dots, pN - 1$, which means there's a road from node p_i to node $i + 1$.

Followed by q lines, each line contains 3 space separated integers v, L and R $(1 \le v, L, R \le N)$ representing a scenario $(L \le R)$.

Output:
For each test case print q lines, each of them contains the answer of the corresponding scenario by printing the maximum total number of white walkers that can be killed by choosing the optimal pair of nodes that satisfies the plan or print -1 if you can't find a pair of different nodes that satisfies the plan

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

Example output:
```
4
-1
1
-1
```

Explanation 1:
In the first scenario, one will go to node 5 and the other will go to node 6, each of them will kill 2 white walkers, so the total is 4. In the second scenario, to reach any node in that range, they must go through the road from node 1 to node 2, which isn't allowed according to the plan. In the third scenario, one of them will choose to stay at node 2, and the other one will go to node 3. In the fourth scenario, they can't go to node 1 from node 2 (the road is in the other direction), and they can't both choose node 2.

Time and memory limit: 2s / 256 MB

Solution:

Let's consider a single scenario (v, L, R). Since the task is to find two paths that start in v and have no common edge, we must choose two different subtrees of v with the deepest nodes in each of them. A trivial solution would be to iterate over the subtrees of v, for each of them determine the deepest node from the given range $[L, R]$ and finally pick two best results. This algorithm gives us the complexity $O(q * N)$.

To improve the running time, we should avoid iterating over the tree for every scenario. Since the queries are somehow connected with intervals, it seems that a segment tree can be helpful. In this case, we can use a multidimensional segment tree.

Let's consider a grid $N \times N$, where the dimensions refer to the tree nodes, sorted in two orders: node id and post-order. For every vertex with node id v_1 and post-order id v_2 let's put a value equal to its depth in the cell with coordinates (v_1, v_2). It is known that in the post-order sequence vertices from any subtree of the node graph form a single interval. Now if we want to find the deepest node in the given subtree with the node id from the given range, all we need to do is to find the maximal value in segment $[a, b] \times [c, d]$, where a, b are the node id boundaries and c, d refer to the minimal and maximal post-order label from the subtree. Such queries can be efficiently computed by using a 2D segment tree, built on the grid described above. Since its size is too big to keep it in the memory, it must be constructed dynamically. It should be noticed that from N^2 base fields only N contains information.

The algorithm is as follows: firstly, we create a 2D segment tree over a square $N \times N$, which can put value in the given point and find the maximal value in the given rectangle. Then, for every vertex v_i on depth d_i with post-order p_i we insert value d_i to the tree, on the field (v_i, p_i). Finally, we process our scenarios. For every query (v, L, R) we find the maximal node n_1 by asking the segment tree for range $[L, R] \times [a, b]$, where a, b refer to minimal and maximal post-order in the subtree of v. Let's denote the id of the result by r_1. If there is no such vertex or the answer is v itself, no valid pair of paths can be found. Otherwise, we must find the second path. Since the post-orders are naturally sorted, we can easily binsearch the child of v which contains r_1 in its subtree S. To get a valid second path, we must find the second deepest vertex r_2 in the very same segment, excluding the subtree S. Since the post-orders of vertices in S form a single interval, this splits the next query in at most two - lower and higher post-orders. Finally, if r_2 is found, we can output the answer, which is the sum of the depths found, subtracted by the depth of v. If r_2 was not found and cannot be replaced with v, there is no valid answer.

The solution for a single scenario includes finding the maximum in the segment tree and binsearching through the children of v. This gives us the complexity of $O(log^2 N + \log N)$, which means total $O(q\ log^2 N)$. It should also be noticed that this approach computes all the answers online.

Another solution uses only 1D segment tree. Firstly, we read all the scenarios and group them in the corresponding vertices. Then, traversing the tree bottom-up, we build a segment tree over the subtrees and answer the queries from their roots.

Specifically, suppose we stand in the vertex v. Firstly, we process the children of v recursively. This way, we have a segment tree for every child of v. Since these trees are built on pairwise disjoint sets, we can naturally merge them into one segment tree by copying every meaningful node. Then for every scenario stored in v we get the deepest vertex in the proper range. Finally, we iterate over the children of v. For a child v_i, if there is at least one scenario for which a vertex in v_i's subtree was selected, we temporarily subtract all vertices in this subtree from the merged segment tree and compute the answer for every such scenario. As a result, we return the merged segment tree.

This approach obviously leads to a valid solution but is too slow to be used. However, it can be improved to achieve sufficiently short running time.

Let's look at the subtrees of children of v and let's denote the biggest one as W. When merging the segment trees, we can merge the subtrees excluding W and for every query asked, look for the answer in both. When subtracting the subtrees in the final stage, now there is no need to do it for W. Finally, when returning the structure, we add the merged part to the one obtained from W.

This way we process the vertex v in a time proportional to the sum of v's children subtrees sizes, excluding the biggest one. It is known that for any tree, the sum

$$\sum_{v\in V}\sum_{k\in S(v)}|T_k|$$

where $S(v)$ is the set of children of v excluding the one with biggest subtree and T_k denotes the subtree rooted in k, is $O(N)$. The cost of adding a segment tree of size a to another can be bounded by $a * \log N$. Therefore, the complexity of this solution is $O(q\log N + N\log N)$. This is even faster than the previous one, though the answers are computed offline.

Problem source: A2OJ
Solution by:
Name: Michał Zawalski

Round 2: This Means War

Statement:

An army is going into war, and they want to divide all soldiers into some groups, in a way that maximizes their total strength. Let's consider the N soldiers as points in a 2-dimensional plane, with all soldiers standing on the x-axis at distinct locations, the i^{th} soldier will be standing at x_i on the x-axis (the soldiers are numbered from 1 to N). You will be given the soldiers in a sorted order based on their x value, from left to right.

Your task is to divide them into one or more groups, where each soldier belongs to exactly one group, and all members of any group are next to each other without anyone else from other groups in between them, so each group will be defined using 2 integers, a and b (where $a \le b$), which means this group includes all soldiers from the a^{th} position to the b^{th} position (inclusive).

Each soldier will be given a function f_i to be used (only if that soldier is the left most soldier of a group) to evaluate the strength of the group. You will be given a list of M different values, each value is z_j (they are numbered from 1 to M), which will be used to evaluate all the functions, for each function fi you will be given the value of $f_i(z_j)$. To get the value of any z other than the given M ones, you just consider (z_j, $f_i(z_j)$) as a point, and connect every 2 consecutive points (based on z_j) in each function using a straight line segment, and now you have a function which covers all possible values from z_1 to z_M.

The strength of the whole army is the sum of strengths of each group, the strength of a group from the a^{th} position to the b^{th} position is $f_a(x_b)$, in other words, it's the value of the function for the most left soldier when we pass the x value of the most right soldier to it. **Check the notes at the end for more explanation of the first test case.**

You are given all the required details as described above, and your task is to divide the soldiers into groups to maximize the total strength of the whole army.

Input:

Your program will be tested on one or more test cases. The first line of the input will be a single integer T $(1 \le T \le 10)$ representing the number of test cases. Followed by T test cases.

Each test case starts with a line containing 2 integers separated by a space, N $(1 \le N \le 10^5)$ representing the number of soldiers and M $(2 \le M, N \times M \le 10^5)$ representing the number of z values.

Then follows a line containing N sorted integers separated by a space, which are the positions of the soldiers $x_1, x_2, \dots, x_N (-10^6 \le x_i \le 10^6)$.

A line containing M sorted integers separated by a space, which are the z values as described above $z_1, z_2, \dots, z_M(-10^6 \leq z_j \leq 10^6)$ follows.

Then, they are followed by N lines, where each line contains M integers separated by a space. The j_{th} value from the left in the ith line from the top is the value of $f_i(z_j)$ $(-10^6 \leq f_i(z_j) \leq 10^6)$.

It is guaranteed that $z_1 \leq x_1$ and $x_N \leq z_M$.

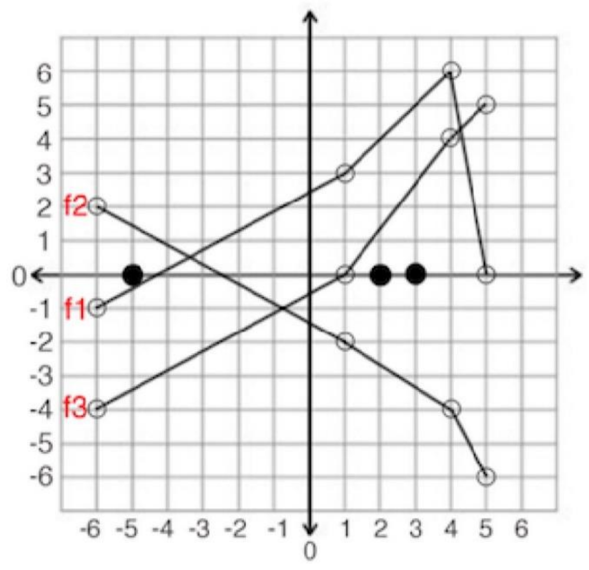
Output:
For each test case print a single line containing a single decimal number rounded to exactly 6 decimal places, which is the maximum strength of the whole army you can get.

Example input:
```
3
3 4
-5 2 3
-6 1 4 5
-1 3 6 0
2 -2 -4 -6
-4 0 4 5
5 2
-2 5 8 9 10
-2 10
-7 -6
-3 -7
0 -8
9 -10
5 -4
2 2
0 7
-2 7
-10 -2
-4 -10
```

Example output:
```
6.666667
-6.000000
-2.000000
```

Explanation:

The following image represents the first test case:



3 solid circlers on the x-axis are the locations of the 3 soldiers, and we have the 3 functions f_1, f_2 and f_3 (one for each soldier) plotted as described above. The best solution here is to put the first 2 soldiers in a group, the strength of that group will be $f_1(x_2)$, which is $f_1(2) = 4$, and the last soldier alone and the strength of that group will be $f_3(x_3)$, which is $f_3(3) = 2.666667$, so the total strength is 6.666667 (everything rounded to 6 decimal places).

Time and memory limit: 5s / 256 MB

Solution:

We will start with a dynamic programming approach. Let's denote DP_r as the maximum strength of the army we can obtain if we use only first r soldiers. For a fixed r we can focus on the last group of soldiers to devise the following recurrence relation:

$$DP_r = \max_{1 \le l \le r}(DP_{l-1} + f_l(x_r))$$
$$DP_0 = 0$$

We want to calculate DP_i for $i \in \{1,2,\dots,n\}$, and output DP_n as our final solution. To find an efficient way to solve this let's consider a simpler case with two assumptions:

1. Each f_l is a linear function
2. DP_{l-1} can be ignored

The expression to maximize becomes just $f_l(x_r)$, where f_l is linear. In other words, for each x_r we are looking for a "highest" linear function from the set $\{f_1, f_2, \dots, f_r\}$. This can be solved in $O(r\log r)$ with a technique based on envelopes, known as *convex hull trick* (CHT). The linked tutorial describes CHT well, but since it's a very popular method among BubbleCup setters, you can find its description in several places in the last year's booklet. To solve the original problem, we have to adapt CHT to a case where the two assumptions stated above don't hold.
Firstly, f_l being piecewise linear instead of linear can be solved easily by rebuilding the whole CHT structure whenever we come across a z value. At any time during our algorithm, we consider exactly one linear piece for each piecewise linear function (the piece above x_r). The total complexity of all rebuilding steps is $O(NM)$, which does not harm the overall performance since the problem statement explicitly states that $NM \le 10^5$.
Secondly, introducing DP_{l-1} to the expression that needs to be maximized merely changes the equation of a line that we add to our CHT structure. Instead of adding $f_l(x) = k_l \cdot x + n_l$ we will add $f_l(x) + DP_{l-1} = k_l \cdot x + n_l + DP_{l-1} = k_l \cdot x + n'_l$. After modifying the line insertion part, we can keep the rest of the algorithm unchanged.
The complexity of the entire solution is $O(T \cdot (MN + N\log N))$.

Problem source: A2OJ
Solution by:
Name: Nikola Jovanović

Round 2: Lannister Army

Statement:
In Jaime's army there is a total N number of warriors. All of them are standing in a single row. Now Jaime wants to convey a message to his warriors. But it's very difficult to convey a message if the warriors are standing in a single row. So, Jaime wants to break that single row into K rows. This is such a formation that in each row at least one warrior should be present. It's important mention that there is an amount of unhappiness associated with each warrior x which is equal to: number of warriors in front of x (in his row) whose height is greater than the height of x. In addition to that, the total unhappiness is a sum of unhappiness of all warriors. Jaime wants his army to be as happy as possible. Now, Jaime wants you to break the single row into K rows so that the total unhappiness of the army is minimum.

Note : You just have to break the row, you are not allowed to change the position of the warriors.

Input:
The first line of input contains two integers N and K. The second line of input contains N number of integers, i^{th} of which denote height of i^{th} warrior standing in that single row (represented as $H[i]$).

Constraints:
- $1 \leq N \leq 5000$
- $1 \leq K \leq N$
- $1 \leq H[i] \leq 10^5$

Output:
Output the minimum possible value of the "total unhappiness".

Example input 1:
```
6 3
20 50 30 60 40 100
```

Example output 1:
```
0
```

Explanation 1:
Break as :

Row 1 : 20 50

Row 2 : 30 60

Row 3 : 40 100

Example input 2:
```
8 3
20 50 30 60 40 100 5 1
```

Example output 2:

```
2
```

Explanation 2:

Row 1 : 20 50 30 60, Unhappiness = 1

Row 2 : 40 100, Unhappiness = 0

Row 3 : 5 1, Unhappiness = 1

Total = 2

Time and memory limit: 2s / 256 MB

Solution:

I will describe two solutions to this problem.

The first solution is based on the divide and conquer optimization in dynamic programing.

Our recurrent formula is $dp[i][j] = min\{0 \leq k < j \mid dp[i-1][k] + cost[k+1][j]\}$, where $dp[i][j]$ is the lowest possible level of unhappiness of the warriors if we break first i of them in j rows, $cost[i][j]$ is unhappiness of the warriors in row formed by taking continuous segment from i to j (both borders inclusive). We also know that $cost[i][j] \leq cost[i][j+1]$ and $cost[i][j] \leq cost[i-1][j]$ because adding one warrior to a row can't decrease the unhappiness. Another important inequation is that for each $r > l$:

(*) $cost[r][j+1] - cost[r][j] \leq cost[l][j+1] - cost[l][j]$

To prove it, let's look at the following equation: $cost[i][j+1] = cost[i][j] + cnt[i][j], cnt[i][j]$is the number of warriors k such that $i \leq k \leq j$ and $H[k] > H[j+1]$. It's obvious that $cnt[l][j] \geq cnt[r][j]$ for each $l < r$. (*) follows from $cnt[l][j] = cost[l][j+1] - cost[l][j]$ and $cnt[r][j] = cost[r][j+1] - cost[r][j]$. When (*) is true we can use divide and conquer optimization in dynamic programing. It's based on fact that $opt[i][j] \leq opt[i][j+1]$, where $opt[i][j]$ is minimum index such that $dp[i][j] = dp[i-1][opt[i][j]] + cost[opt[i][j] + 1][j]$.

I'll prove it before describing why this is helpful. Suppose that $opt[i][j] > opt[i][j+1]$.

From recurrence for $dp[i][j]$ follows:

$dp[i][j] < dp[i-1][opt[i][j+1]] + cost[opt[i][j+1] + 1][j]$

(1) $dp[i-1][opt[i][j]] + cost[opt[i][j] + 1][j] < dp[i-1][opt[i][j+1]] + cost[opt[i][j+1] + 1][j]$

From recurrence for $dp[i][j+1]$ follows:

$dp[i][j+1] <= dp[i-1][opt[i][j]] + cost[opt[i][j] + 1][j+1]$

$dp[i-1][opt[i][j+1]] + cost[opt[i][j+1] + 1][j+1] <= dp[i-1][opt[i][j]] + cost[opt[i][j] + 1][j+1]$

Here we use $<=$ because $opt[i][j] > opt[i][j+1]$ so if two options are equal lower index is chosen to be $opt[i][j+1]$.

But if

$dp[i-1][opt[i][j+1]] + cost[opt[i][j+1] + 1][j+1] = dp[i-1][opt[i][j]] + cost[opt[i][j] + 1][j+1]$

Then

$dp[I-1][opt[i][j+1]] + cost[opt[i][j+1]+1][j] <= dp[I-1][opt[i][j]] + cost[opt[i][j]+1][j]$. Contradiction with (1)!

Now we have:

(2) $dp[i-1][opt[i][j+1]] + cost[opt[i][j+1]+1][j+1] < dp[i-1][opt[i][j]] + cost[opt[i][j]+1][j+1]$

By summing (1) and (2) we get:

$cost[opt[i][j]+1][j] + cost[opt[i][j+1]+1][j+1] < cost[opt[i][j+1]+1][j] + cost[opt[i][j]+1][j+1]$

$cost[opt[i][j+1]+1][j+1] - cost[opt[i][j+1]+1][j] < cost[opt[i][j]+1][j+1] - cost[opt[i][j]+1][j]$

Contradiction with (*)!

Now I will describe algorithm based on above inequation. First, we precompute all costs in $O(N^2)$. After that we will run recursive algorithm K times. Let's say that we process i^{th} run and want to find values of $dp[i][k]$ for $l <= k <= r$. Let s be $opt[i][l-1]$ and e be $opt[i][r+1]$. Then we take element m in the middle between l and r.

Then $dp[i][m] = min\{s <= k <= min\{e, m-1\} | dp[I-1][k] + cost[k][m]\}$.

Now we know values of $opt[i][m]$ and $dp[i][m]$ so we can recursively solve intervals from l to $m-1$ and from $m+1$ to r. Each run has time complexity of $O(NlogN)$ because maximum depth of recursion is at most $logN$ and in each depth we iterate over $O(N)$ elements. So, the total time complexity of this solution is $O(KNlogN)$.

Second solution is based on binary search.

Let's say that $sol[i]$ is minimum total unhappiness if we break warriors in exactly i rows. It's obvious that $sol[i] >= sol[i+1]$ because we can just take rows that are optimal for i and break one of them into two and we get same or lower total unhappiness. Lets see what happens when we add additional unhappiness C for each row in final answer.

Solutions for this modified problem will be $sol2[i] = sol[i] + i * C$. Now solutions with more rows become worse.

We can find minimum total unhappiness if we can split warriors in any number of rows in $O(N2)$.

Recurrent formula is: $dp[i] = min\{0 <= k < i | dp[k] + cost[k][i] + C\}$.

We can also find how many rows are in optimal solution.

Let's define value $f[C]$ as the number of rows in optimal solution for C. If there are more optimal solutions take one with the maximum number of rows. It's obvious that $f[i] >= f[i+1]$.

Maybe it seems that this condition is enough to run binary search and find C for which solutions is $sol[k] + C * k$, but we don't know that there exists such C. Let's look at functions $sol2[i](C)$. They are linear functions with slope equal to i and y-intercept equal to $sol[i]$. $f[C]$ gives us the line that intersects vertical line $x = C$ in the lowest point. Now we know that we can find solution for k with binary search if and only if $sol2[k]$ has some point on the lower convex hull of these functions. If we prove that each function has some integer point on the lower convex hull, then our algorithm is correct in every case. It's true if and only if (3) $sol[i-1] - sol[i] >= sol[i] - sol[i+1]$. I will prove it.

Suppose that (3) is correct. Let $x[i][j]$ be the x coordinate of intersection of i^{th} and j^{th} function. It's obvious that $x[i][i+1] = sol[i] - sol[i+1]$, so from (3) follows that $x[i-1][i] >= x[i][i+1]$.

i^{th} function is on lower convex hull between points $x[i-1][i]$ and $x[i][i+1]$. Because of that each function has at least one point on the lower convex hull.

Suppose that $sol[i-1] - sol[i] < sol[i] - sol[i+1]$ for some i. Then $x[i-1][i] < x[i][i+1]$. $(i-1)^{th}$

function is lower than i^{th} on segment from $x[i-1][i]$ to infinity, and $(i+1)^{th}$ function is lower than i^{th} on segment from negative infinity to $x[i][i+1]$, so i^{th} function has no points on the lower convex hull.

The only thing left is to prove that $sol[i-1] - sol[i] >= sol[i] - sol[i+1]$. I will leave it as an exercise for the readers.

Because $f[0] = N$ and $f[N^2] = 1$ we have to run binary search on interval from 0 to N^2. The total time complexity of this solution is $O(N2log(N2))$.

First solution has slightly better time complexity, but it also has larger constant factor.

Problem source: SPOJ
Solution by:
Name: Tadija Šebez

Round 2: Ada and Cucumber

Statement:

Ada the Ladybug works as farmer. It's the season of cucumbers and she wants to harvest them. There lie many cucumbers all around her house. She wants to choose a direction and follow it until all cucumbers in that direction are collected.

Let's consider Ada's house as centerpiece of whole universe, lying on [0,0]. The cucumbers are defined as lines on plane. No cucumber goes through Ada's house (and no cucumber touches it).

How many cucumbers can Ada pick in one go if she chooses the best direction possible?

Input:

The first line contains an integer T, the number of test-cases.

Each test-case begins with an integer $1 \leq N \leq 10^5$.

Afterward N lines follow, with four integers $-10^6 \leq x1, y1, x2, y2 \leq 10^6$, the beginning and end of each cucumber. Each cucumber has a positive length.

Sum of all N over all test-cases won't exceed 10^6.

Note: Even though cucumber will not go through the house, they might touch, intersect or overlap other cucumbers!

Output:

For each test-case print one integer - the maximal number of cucumbers which could be picked if Ada chooses a direction and picks every cucumber lying in it.

Example input:

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

```
5 1 5 5
6 6 6 -11
3
1 3 4 3
4 2 4 -1
5 1 6 6
```

Example output:

```
3
2
1
4
2
```

Possibly harvested cucumbers:

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

Time and memory limit: 2s / 256 MB

Solution:

Every cucumber is a segment on a plane and we want to find a ray starting at $(0, 0)$ which crosses the biggest number of such segments.

First, let's observe that there exists an optimal ray which passes through an endpoint of some segment. To prove it, let's consider an optimal ray. If it doesn't pass through any of the given endpoints, we can move it clockwise (or counterclockwise) until it reaches some endpoint – after that we obtain a ray which crosses the same segments as initial optimal ray and satisfies our requirement.

Using this observation, we obtain a simple algorithm with complexity $O(N^2)$ – it is enough to consider all possible $2N$ rays and calculate the score for each of them in $O(N)$ time. To solve this task, we need to do it faster.

Let's sort all the endpoints by a polar angle, which we define here as a number from interval $(0, 2\pi]$. We will iterate over them in this order maintaining a counter – how many segments are currently considered ray crosses.

To calculate the initial value of it, we count the segments which one of endpoints lies in upper half-plane and which touches or crosses axis $x > 0, y = 0$. If the current endpoint is a beginning of some segment (i.e. if the segment is directed counterclockwise), we increase the counter by one, otherwise – we decrease it by one. After each change, we check whether we can update the result. If more than one point lies on the same ray, we have to consider the beginnings of the segments before the endings.

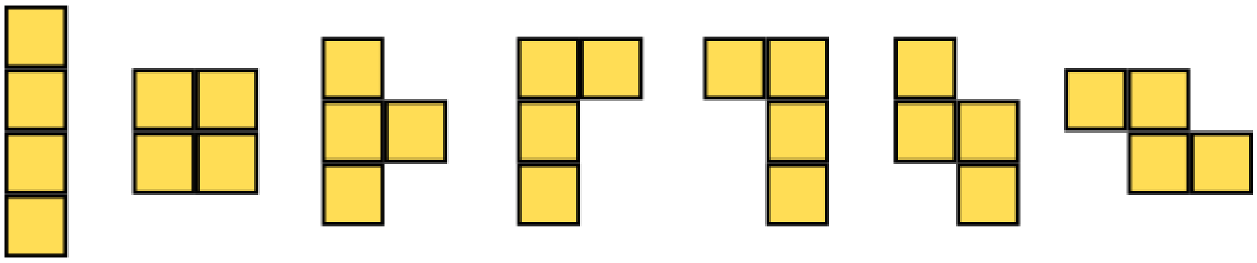
The overall complexity of the algorithm is $O(N \log N)$ per one test-case.

Problem source: SPOJ
Solution by:
Name: Konrad Majewski

Round 2: Seedlings

Statement

Bogdan the Botanist is conducting a research on a certain species of plant, which supposedly has some miraculous healing properties. Thanks to his connections among other scientists, he can get as many seedlings for his research as he pleases. However, plants need proper conditions in order to grow properly. The best method to ensure that the seedlings develop correctly is to place them on specialized shelves, which have proper watering systems, lighting conditions etc. Many different shapes of the shelves are available. Apart from the standard square-shaped 1x1 shelves, it is also possible to obtain shelves that look like this:



Each of the shelves presented above consists of four standard-shaped shelves. Standard shelf can hold one large flowerpot with seedlings. Shelves presented above can hold six flowerpots, thanks to the somewhat better use of available space.

Bogdan already bought a room for storing the seedlings - from the bird's-eye view, it looks like a rectangle, n meters high and m meters wide, consisting of square-shaped fields, each 1 meter wide. You can't place the shelves on some of the fields (due to wiring, water supply systems and other stuff) - we consider such fields blocked. There is a door on the wall above the field in the top left corner - it opens inward, so we can't place anything on this field either. Bogdan plans to fit as many flowerpots in the room as possible, by placing shelves on the remaining fields. Moreover, he wants to do it in such a way that the square-shaped segments of the shelves exactly cover the fields in the room. His research demands constant access to plants - every shelf must be accessible by walking over the non-blocked fields, starting from the field with the door. You can walk between two fields if they share a common edge. Help Bogdan! Place the shelves according to the rules, to fit as many flowerpots in the room as possible.

Input

The first line contains a single integer t, denoting the number of testcases. ($t \leq 10$). Then, the testcases follow. The description of a single testcase begins with two integers n, m $(1 \leq n, m \leq 50)$ - height and width of the rectangle representing the room. Then, the n lines follow, each containing m characters. j^{th} character in i^{th} line denotes a square in i^{th} row and j^{th} column. "." (a dot) denotes a free square, "X" denotes a blocked square. The field in the top left corner is always free.

Output

You should find an arrangement of shelves that satisfies the rules from the problem statement for every testcase. The description begins with two integers p and d $(1 \leq p \leq n * m)$ - the number of shelves and the number of flowerpots on the shelves, respectively. Then, the descriptions of p shelves should follow. Each shelf is described by four integers w_i, k_i, r_i and o_i $(1 \leq w_i \leq n, 1 \leq k_i \leq m, 0 \leq r_i \leq 7, 0 \leq o_i \leq 3)$ - it means that the anchor point of the i^{th} shelf is in the row number w_i and column number k_i, the shelf is of type r_i, and is rotated o_i times 90 degrees to the right, starting from the configuration on the picture. The anchor point of every shape is in the top left corner in the picture (and it is in the same segment after the rotation). Type 0 denotes standard 1x1 square-shaped shelf, types from 1 to 7 correspond to shapes in the picture (counting from left to right).

Example input

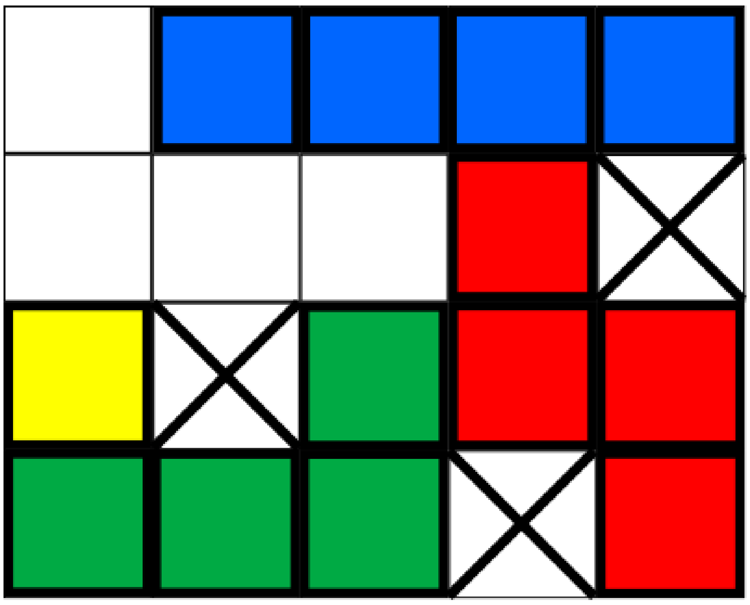
```
1
4 5
.....
....X
.X...
...X.
```

Example output

```
4 19
1 2 1 3
2 4 6 0
3 3 5 1
3 1 0 0
```

Explanation

The arrangement from the sample testcase looks as follows:



You can't place another shelf without violating the rules - for example, by putting the type 0 shelf on the field at (2,3), the red shelf and the green shelf would be unreachable. The arrangement contains three larger shelves and one standard shelf, so the total number of flowerpots is 3*6+1 = 19.

Scoring

If the arrangement of the shelves is correct, and the given number of flowerpots d is correct, it is worth $d/(n * m)$ points. Overall score is equal to the sum of individual scores. Score for the sample output is 19/(4*5) = 0.95.

Time and memory limit: 5 s / 256 MB

Solution:

The solution consists of two main parts. Firstly, it builds a network of paths used for walking. Then the remaining free spaces are filled with shelves. The second part of the solution will be explained first as it is simpler.

On each step we check each free square and try to place each big (consisting of four squares) shelf with each orientation. Every time we find a valid shelf placement we grade it according to a certain heuristic. Then after checking every combination, if we have found at least one valid shelf placement, we place the best (graded highest) shelf and repeat. When no more shelves can be placed, the remaining free squares next to the paths are filled with small shelves. It is important to note that the solution does not regard the not-yet-filled squares as potential paths during this step. Only the squares marked during the first part of the solution are regarded as walkable.

The heuristic used to grade each possible shelf placement consists of two parts. Firstly, a number of points is given for each square filled by the shelf, that does not touch a path. This encourages shelf placement that fills otherwise unreachable squares. Secondly, a number of points is given for each external edge of the shelf that touches an already filled (by a path, shelf or an X) square. This encourages shelf placement that neatly fits into the currently left gaps. Furthermore, the number of potential points is much higher in more complicated areas since a shelf can potentially touch many more filled squares. This way the solution naturally starts filling areas that if left until the end might not get filled. The exact values for each of the two rewards was obtained through testing. They are 13 points for the first type of reward and 9 or 10 for the second. 9 is normally used, but 10 is used when the shelf type is a two by two square because it has fewer external edges.

The first part of the solution is more complicated as it is a combination of two approaches that were tried separately before that. The first one was to build the paths without taking the shelves into much consideration. The solution tried to build a semi-regular structure of parallel paths that were spaced about 3-4 squares from each other. This was done by having a list of potential extensions to the current path network — squares that touched the current path. Then the best one would be selected using some heuristic based on the distance from other squares that are a part of the current path. At each step, the current path network is scored based on the number of the squares that are reachable (or in a later version that are at distance two to the nearest path square). After all squares are processed, the path is rebuilt from the start, but this time up to the point where the score is highest. This approach performed rather poorly when compared to the following ones. This is because it doesn't take into consideration what shelves can fit in the different squares.

The second approach that was tried out consisted of writing a simpler but complete (not split into two parts) solution and after it terminates, to disregard its shelf placement but use the paths it has found (so each free square that is reachable from the start is marked as part of the path). The solution itself was a greedy algorithm that at each step placed the shelf that cuts off the least amount of squares/shelves. This approach performed much better because the paths were suited to the possible shelf placements on the field.

The final approach was a combination of the two. On each step, one of two things is done. Either the path network is extended in a manner similar to the one in the first approach (but with a different heuristic that will be described below), or a shelf is placed in a manner similar to the way they are placed in the second step of the solution (however the values for the rewards are different — the reward for using an unreachable square is 15, and the rewards for touching a taken square are -1 and -0.9, they are actually punishments because we want the initial shelves to be more spaced out to leave room for the path network). Again, at each point, the current path network is graded based on the reachable free squares plus the points from the shelves that are already placed. This would be done until all squares are processed but this exceeds the time limit, so it is done until the current score starts significantly decreasing when compared to the best score. Finally, the path network is rebuilt up to the point where the score is the highest, and the shelves are disregarded.

The heuristic used to pick the square, with which to extend the path network, consists of the following steps: for each unreachable free square find which candidate (for extending the path network) is the closest; for each candidate count the number of free squares to which it is the closest; pick the candidate that is the closest to the freest cells. The only problem is that doing this as described actually exceeds the time limit, so an approximation of this is done. All candidates are added in a BFS queue simultaneously and we should remember where we originally started each square. This gives an unfair advantage to the candidates that are near the beginning of the queue but behave almost the same way as the intended heuristic.

The only thing left to discuss is which of the two things (placing a shelf or extending the path network) is done at each step. Placing a shelf is done (or tried) at every third step and extending the path network is done at the other two. Furthermore, even on those other two steps there is a 10% chance to try placing a square instead. Again, these values were obtained through testing.

Problem source: SPOJ
Solution by:
Name: Emil Indzhev

Round 2: Big Integer

Statement:

Nathan is a big fan of recreational mathematics. For one of his problems, he needs to add together very large numbers. He created a class called BigInteger to help with the adding, but he isn't done yet! Nathan needs to stress test his code, so he devised the following problem.

There will be N instructions (which are given as a string of length N). There are two types of instructions:

- 0 to 9: Add this digit to the end of the current number. Afterwards, add the current number to the total
- - : Remove the last digit from the current number. It is guaranteed that the current number will not be empty after this instruction. Afterwards, add the current number to the total.

At the beginning, the total is 0 and the current number is 0. Nathan wrote a program in Python to solve this problem, but it is slow and drains his battery too much. Can you help Nathan double check his answers?

Input:

The first line will contain the integer N ($N \leq 500\,000$). The second line will contain a string of length N. Every character in this string can be found in 0123456789-.

Output:

Print the total. Leading zeroes will be ignored by the checker.

Example input 1:

```
8
0100---5
```

Example output 1:

```
00000127
```

Explanation 1:

00 + 001 + 0010 + 00100 + 0010 + 001 + 00 + 005 = 127

Example input 2:

```
4
1817
```

Example output 2:

```
2017
```

Explanation 2:

01 + 018 + 0181 + 01817 = 2017

Example input 3:
```
2
0-
```

Example output 3:
```
0 0
```

Time and memory limit: 7s / 256 MB

Solution:

Let’s try to use the sqrt-decomposition.

Firstly, split all the instructions in blocks of length K. We will operate the blocks from the first to the last.

Now we have some new blocks. Let's say that the current number that we have before the block is X. Then divide X into two parts LP and RP. RP will be the last K digits of X and LP will be equal to $(X - RP)$.

Look at RP. The length of it won't be greater than $K * 2$, so, we can simply maintain RP and add it to the answer in $O(|RP|)$.

Now look what is going on with LP. If we process the new instruction, then if it isn't '-' then LP is changing to $LP * 10$, if it’s otherwise, we should delete one digit from the end of LP. If we notice that this digit is always zero, then we can do the following. Delete all zeros from the end of LP, so the new number is $ZipLP$, then add to the answer $ZipLP * (10^{cnt_1} + 10^{cnt_2} + \cdots + 10^{cnt_K})$, where cnt_i is the number of zeros at the end of LP after i instructions. As cnt_i is always positive, all this can be done with FFT in $O(N * log(N))$.

It works in $O(N/K * (N * log(N) + N) + N * K)$. But it's a little bit slow.

Optimization:

Store RP and answer in blocks by $B = 15$ digits. And now it works in $O(N/K * (N * \log N + N/B) + N * K/B$. Take $K = 10000$, add some magic and get AC.

Problem source: DMOJ
Solution by:
Name: Mark Korneychik