# bubble cup 8

## booklet

Microsoft
Development Center Serbia

# BUBBLE CUP 2015

Student programming contest

Microsoft Development Center Serbia

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

Belgrade, 2015

**Scientific committee**

Vanja Petrović Tanković
Andrija Jovanović
Marko Rakita
Mladen Radojević
Ibragim Ismailov
Aleksandar Milovanović
Stefan Stojanović
Andrej Ivašković
Borna Vukorepa
Predrag Ilkić
Andreja Ilić

**Qualification analyses**

Dimitrije Erdeljan
Vladimir Milenković
Dušan Živanović
Nikola Jovanović
Petr Smirnov
Stjepan Požgaj
Nikita Podguzov
Tonko Sabolčec
Ivan Stošić
Jovan Gagrčin
Ivan Dejković
Ivan Šego
Bartłomiej Dudek
Nikola Spasić
Srđan Milaković
Mislav Bradač
Lazar Grbović

**Cover:**

Sava Čajetinac

**Typesetting:**

Predrag Ilkić

**Proofreader:**

Vanja Petrović Tanković

**Volume editor:**

Dragan Tomić

# Contents

# Preface

*Dear Finalist of Bubble Cup 8,*

*Thank you for participating in the eight edition of the Bubble Cup. I hope you had a great time in Belgrade and enjoyed the event.*

*MDCS has a keen interest in organizing Bubble Cup. Many of our engineers participated in similar competitions in the past. One of the characteristics of our team culture in MDCS is passion for solving challenging technical problems.*

*Bubble Cup 8 has attracted the largest number of participants so far, with more than 600 students participating during qualifications. Finalists had a chance to hear world-class experts on our Bubble Cup conference, organized for the first time this year. In addition to countries that have participated in Bubble Cup finals before (Serbia, Croatia, Poland), we had several new countries this year (Latvia, Ukraine, Belarus, Russia). Also for the first time we organized an online mirror of the finals, with over 2800 teams registered for the competition. Bubble Cup continues to grow and increase its popularity year by year.*

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

*Thanks,*<br>
*Dragan Tomić*<br>
*MDCS PARTNER Engineer manager/Director*

# About Bubble Cup and MDCS

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

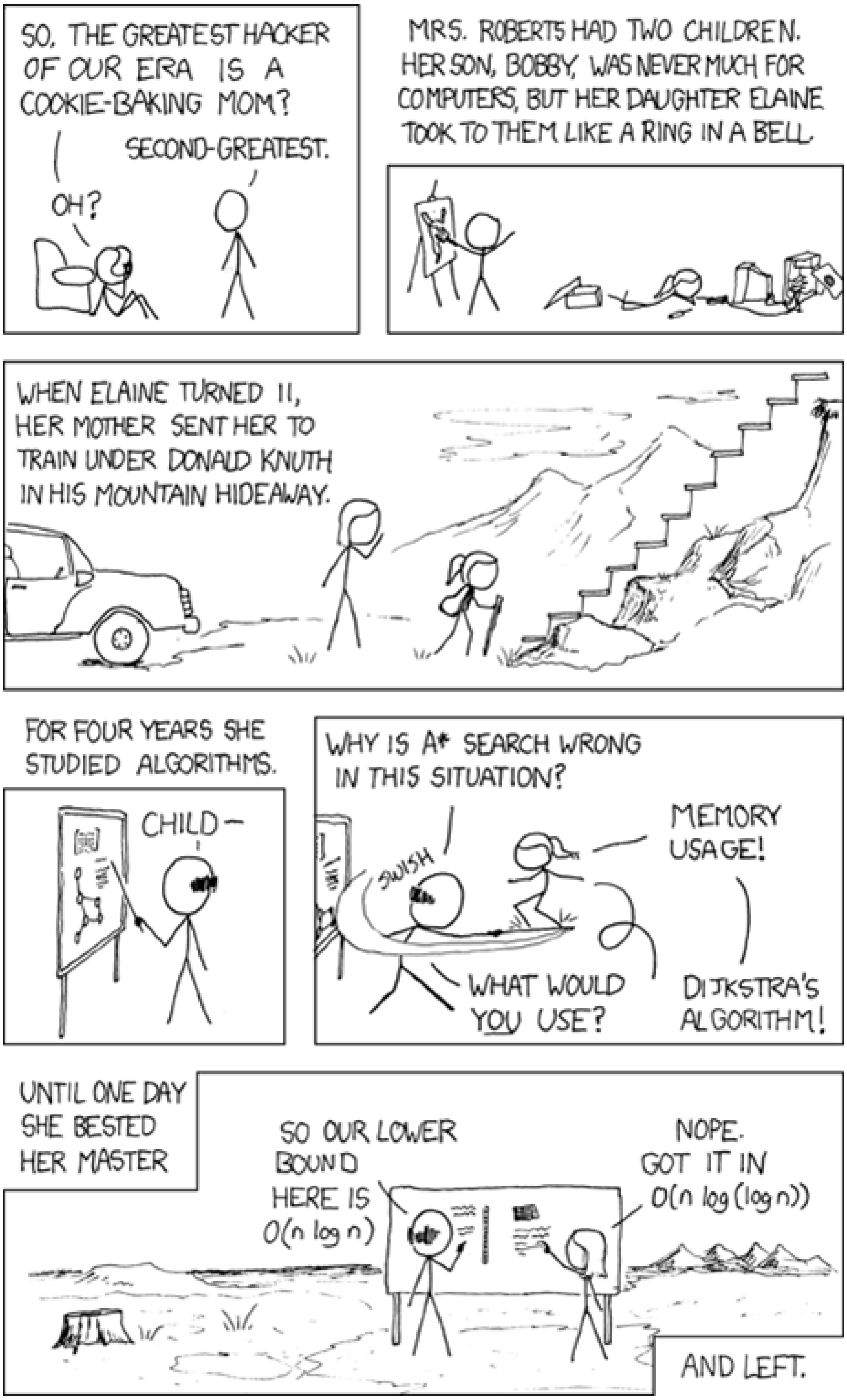
Format of the competition has remained the same this year. All competitors battled for the place in finals during two qualifications rounds. Top high school and university teams were invited to the finals in Belgrade, where they competed in the traditional five hours long contest. Finalists also had the opportunity to listen to experts in competitive programming during the Bubble Cup Conference, which was organized for the first time this year.

**Microsoft Development Center Serbia (MDCS)** was created with a mission to take an active part in conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 10 years a number of products came out as a result of great team work and effort.

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

Microsoft
Development Center Serbia

## *Problem set & Analysis*


SO, THE GREATEST HACKER OF OUR ERA IS A COOKIE-BAKING MOM?
OH?
SECOND-GREATEST.
MRS. ROBERTS HAD TWO CHILDREN. HER SON, BOBBY, WAS NEVER MUCH FOR COMPUTERS, BUT HER DAUGHTER ELAINE TOOK TO THEM LIKE A RING IN A BELL.
WHEN ELAINE TURNED 11, HER MOTHER SENT HER TO TRAIN UNDER DONALD KNUTH IN HIS MOUNTAIN HIDEAWAY.
FOR FOUR YEARS SHE STUDIED ALGORITHMS.
CHILD—
WHY IS A* SEARCH WRONG IN THIS SITUATION?
SWISH
MEMORY USAGE!
WHAT WOULD YOU USE?
DIJKSTRA'S ALGORITHM!
UNTIL ONE DAY SHE BESTED HER MASTER
SO OUR LOWER BOUND HERE IS O(n log n)
NOPE. GOT IT IN O(n log(log n))
AND LEFT.

Taken from xkcd.com – A web comic of Romance, Sarcasm, Math, and Language

# Finals 2015

Bubble Cup finals were held on September 4 and 5 in Mikser House. The event started with Bubble Cup Conference, where **Mike Mirzayanov**, **Michal Forišek** and **Przemysław Dębiak (Psyho)** shared their stories, experience, tips and tricks with the competitors.

The final competition remained in the traditional 5-hours format, similar to ACM ICPC. This year, university and high school students competed in the same category. In one of the most exciting Bubble Cup finals so far, 21 teams were presented with a total of 9 problems.

Final standings were unknown until the very end of the competition. Many successful submissions and changes on the scoreboard happened in the last 15 minutes of the finals. Ultimately, the team **3/4 IOI Polish Team** (Poland, high school) rose to the top, as the only team that solved 7 problems. The second place went to **LNU Penguins** (Ukraine, university), followed by team **unusual** (Latvia, university) with 6 problems solved and only 1 minute penalty behind.

In addition to on-site finals in Belgrade, an online mirror of the finals was organized on Codeforces (www.codeforces.com), with more than 2800 teams registered for the competition. The winner of the online competition was **tourist** (Gennady Korotkevich), solving all 9 problems in 4 hours and 17 minutes.

| Place | Team | Score | Penalty |
|---|---|---|---|
| **1** | 3/4 IOI Polish Team | **7** | 1325 |
| **2** | LNU Penguins | **6** | 1067 |
| **3** | Unusual | **6** | 1068 |
| **4** | Frozen Heart | **5** | 501 |
| **5** | Me[N]talci Corp. | **5** | 593 |
| **6** | Wroclaw Cheetahs | **5** | 841 |
| **7** | Lazo i prijatelji | **4** | 394 |
| **8** | RAF RAF GSM | **4** | 571 |
| **9** | Is there anybunny there? | **4** | 576 |
| **10** | Papirnate maramice | **4** | 613 |
| **11** | Air penguins | **4** | 620 |
| **12** | Nulla dies since AC | **4** | 624 |
| **13** | XYZ | **4** | 708 |
| **14** | Rozochocone Szczerzożerzaczki | **3** | 435 |
| **15** | Booleans | **3** | 498 |
| **16** | Mastovito ime | **3** | 552 |
| **17** | Team2k15 | **1** | -16 |
| **18** | Програмерски Моћан Факултет у Нишу | **1** | 14 |
| **19** | Bubblecup Cunningsnatch | **1** | 54 |
| **20** | Spacemen | **1** | 84 |
| **21** | Gimnazija Sombor | **1** | 159 |

Table 1. Final results of Bubble Cup

# Problem A: Fibonotci

*Authors*

***Vanja Petrović Tanković***

*Implementation and analysis*

***Vanja Petrović Tanković***

Fibonotci sequence is an integer recursive sequence defined by the recurrence relation

$$F_n = c_{n-1} \cdot F_{n-1} + c_{n-2} \cdot F_{n-2}$$

with

$$F_0 = 0,\ F_1 = 1.$$

Sequence $c$ is infinite and *almost cyclic* sequence with a cycle of length $N$. A sequence $s$ is *almost cyclic* with a cycle of length $N$ if $s_i = s_{i\ mod\ N}$, for $i \geq N$, except for a finite number of values $s_i$, for which $s_i \neq s_{i\ mod\ N}$ $(i \geq N)$.

Following is an example of an *almost cyclic* sequence with a cycle of length $4$.

$$s = (5, 3, 8, 11, 5, 3, 7, 11, 5, 3, 8, 11, \dots)$$

Notice that the only value of $s$ for which the equality $s_i = s_{i\ mod\ 4}$ does not hold is $s_6$ ($s_6 = 7$ and $s_2 = 8$).

You are given $c_0,\ c_1, \dots, c_{N-1}$ and all the values of sequence $c$ for which $c_i \neq c_{i\ mod\ N}$ $(i \geq N)$.

Find $F_K\ mod\ P$.

### Input:

The first line contains two numbers $K$ and $P$. The second line contains a single number $N$. The third line contains $N$ numbers separated by spaces, that represent the first $N$ numbers of the sequence $c$. The fourth line contains a single number $M$, the number of values of sequence $c$ for which $c_i \neq c_{i\ mod\ N}$. Each of the following $M$ lines contains two numbers $j$ and $v$, indicating that $c_j \neq c_{j\ mod\ N}$ and $c_j = v$.

### Output:

Output should contain a single integer equal to $F_K\ mod\ P$.

### Constraints:

- $1 \leq N, M \leq 50.000$
- $0 \leq K \leq 10^{18}$
- $1 \leq P \leq 10^9$
- $1 \leq c_i \leq 10^9$, for $i = 0, 1, \dots, N-1$
- $N \leq j \leq 10^{18}$
- $1 \leq v \leq 10^9$
- All values are integers

### Example input:

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

### Example output:

```
4
```

**Time and memory limit: 3s / 64MB**

## *Solution and analysis:*

Let's first solve a simpler problem – when the sequence $c$ is *cyclic*, i. e. when $c_i = c_{i \, mod \, N}$, for $i \geq 0$.

This simpler version is similar to *Fibonacci sequence*. Actually, for $N = 1$ and $c_0 = 1$, it is the Fibonacci sequence. To find the $K^{th}$ number of these kind of recursive sequences fast we should first write them in their matrix form. Matrix form of this sequence is:

$$\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = \begin{pmatrix} c_{i-1} & c_{i-2} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix}$$

Expanding this, we can see that

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{K-1}C_{K-2} \dots C_2C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{where } C_i = \begin{pmatrix} c_i & c_{i-1} \\ 1 & 0 \end{pmatrix}.$$

How do we calculate this efficiently?

For relatively small $K$, and we will take $K < N$ for this case, we can do this just by multiplying all the matrices. For large $K$ $(K \geq N)$, we will take advantage of the fact that $c$ is *cyclic.* Since $c$ is *cyclic* with cycle of length $N$, we know that $C_{N-1}C_{N-2} \dots C_1C_0 = C_{iN+(N-1)}C_{iN+(N-2)} \dots C_{iN+1}C_{iN}$, for $i \geq 0$ (note that $C_0 = \begin{pmatrix} c_0 & c_{N-1} \\ 1 & 0 \end{pmatrix}$). Let's define this product of matrices as $S = C_{N-1}C_{N-2} \dots C_1C_0$.

Now, we can write a formula for $F_K$ that can be calculated quickly:

$$\begin{pmatrix} F_K \\ F_{K-1} \end{pmatrix} = C_{a-1}C_{a-2} \dots C_1C_0S^bC_{N-1}C_{N-2} \dots C_2C_1 \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}, \text{where } b = (K-N) \; div \; N \text{ and } a = K \; mod \; N.$$

We can calculate $S^b$ in $O(logb)$ steps using **exponentiaton by squaring**, and then we can just multiply everything in the expression to get $F_K$ quickly.

Let's get back to the full problem, when $c$ is *almost cyclic*. In this case, we cannot just use $S^b$ in the formula above, because some matrices in $S^b$ may not respect the *cyclic* property. Instead of $S^b$, we will have something like

$$S \cdot S \cdot \dots \cdot S \cdot E_1 \cdot S \cdot S \cdot \dots \cdot S \cdot E_2 \cdot \dots = S^{t_1} \cdot E_1 . \cdot S^{t_2} \cdot E_2 \cdot S^{t_3} \cdot \dots$$

where $E_i$ denotes the product of matrices of the cycle, with some matrices being different than the matrices of the original cycle. Also, $i \leq 2M$ since each of the $M$ values of $c$ different than values of the original cycle appears in exactly two matrices, so at most $2M$ of cycles are affected.

We can still calculate each $S^{t_i}$ quickly, using exponentiation by squaring. Since there are at most $2M$ of those, total complexity of this would be $O(MlogK)$.

Now we only need to calculate each $E_i$. Naive way would be to just multiply all matrices of $E_i$. Since the number of matrices is $N$, this would be $O(NM)$ worst case, which is too slow. To quickly calculate $E_i$, we will initially create a **segment tree** of matrices, with matrices of original cycle in the leaves. Internal nodes of the tree will represent the product of their children. This means that the root will represent the product of all matrices in the cycle. To calculate $E_i$, we will just update our segment tree with those values that are different than the original values of the cycle. We will do this by updating corresponding leaves of the tree, moving up to the root and updating the products in the internal nodes. After we're done updating the tree with all of the matrices that are different than matrices of the original cycle, we will just use the product in the root of the tree. Finally, we will update the tree back with matrices of the original cycle in order to reuse the segment tree for $E_{i+1}$.

Since there are $O(N)$ nodes in the segment tree, the complexity of updating is $O(logN)$. The total complexity is then $O(MlogN + MlogK)$. We should also mention that the constant factor is not very small, since we operate on matrices and not just integers.

Note that we need to find $F_K\ mod\ P$ and $P$ may not even be a prime number. However, this does not affect us since we only deal with operations of addition and multiplication throughout the whole procedure and we can just do them all modulo $P$.

# Problem B: Bribes

*Authors* | *Implementation and analysis*

***Andrej Ivašković*** | ***Andrej Ivašković***

Ruritania is a country with a very badly maintained road network, which is not exactly good news for lorry drivers that constantly have to do deliveries. In fact, when roads are maintained, they become **one-way**. It turns out that it is sometimes impossible to get from one town to another in a legal way – however, we know that all towns are **reachable**, though **illegally**!

Fortunately for us, the police tend to be very corrupt and they will allow a lorry driver to break the rules and drive in the wrong direction provided they receive 'a small gift'. There is one patrol car for every road and they will request 1000 Ruritanian dinars when a driver drives in the wrong direction. However, being greedy, every time a patrol car notices the same driver breaking the rule, they will charge **double** the amount of money they requested the previous time on that particular road.

Borna is a lorry driver that managed to figure out this bribing pattern. As part of his job, he has to make $K$ stops in some towns all over Ruritania and he has to make these stops in a certain order. There are $N$ towns (enumerated from 1 to $N$) in Ruritania and Borna's initial location is the capital city i.e. town 1. He happens to know which ones out of the $N-1$ roads in Ruritania are currently unidirectional, but he is unable to compute the least amount of money he needs to prepare for bribing the police. Help Borna by providing him with an answer and you will be richly rewarded.

### Input:

The first line contains $N$, the number of towns in Ruritania. The following $N-1$ lines contain information regarding individual roads between towns. A road is represented by a tuple of integers $(a, b, x)$, which are separated with a single whitespace character. The numbers $a$ and $b$ represent the cities connected by this particular road, and $x$ is either $0$ or $1$: $0$ means that the road is bidirectional, $1$ means that only the $a \to b$ direction is legal. The next line contains $K$, the number of stops Borna has to make. The final line of input contains $K$ positive integers $s_1, \ldots, s_K$: the towns Borna has to visit.

### Output:

The output should contain a single number: the least amount of thousands of Ruritanian dinars Borna should allocate for bribes, modulo $10^9 + 7$.

### Constraints:

- $1 \le N \le 10^5$
- $1 \le K \le 10^6$
- $1 \le a, b \le N$ for all roads
- $x \in \{0,1\}$ for all roads
- $1 \le s_i \le N$ for all $1 \le i \le K$

### Example input:

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

### Example output:

```
4
```

### Explanation:

Borna first takes the route $1 \to 5$ and has to pay 1000 dinars. After that, he takes the route $5 \to 1 \to 2 \to 3 \to 4$ and pays nothing this time. However, when he has to return via $4 \to 3 \to 2 \to 1 \to 5$, he needs to prepare $3000$ $(1000 + 2000)$ dinars. Afterwards, getting to 2 via $5 \to 1 \to 2$ will cost him nothing. Finally, he doesn't even have to leave town 2 to get to 2, so there is no need to prepare any additional bribe money. Hence he has to prepare $4000$ dinars in total.

**Time and memory limit: 1.5s / 64MB**

## *Solution and analysis:*

Let us first provide a suitable interpretation of the task description. The country can obviously be modelled as an undirected graph, where vertices are towns and edges are roads. We see that it is connected (the description mentions that every city is reachable), but we also know that there are $N - 1$ edges, where $N$ is the number of vertices. From this it follows that the graph is, in fact, a tree. For the sake of simplicity, let us make this tree rooted at node 1.

Let us consider just an $a \rightsquigarrow b$ transfer. From the previous assertion it follows that the cheapest path from $a$ to $b$ will always be the shortest path from $a$ to $b$ – which is, in fact, the only path from $a$ to $b$ that does not have any repeated vertices. Borna's trip is thus uniquely defined by all of his stops. Getting from town $a$ to town $b$ requires that Borna first goes to the lowest common ancestor (LCA) node of $a$ and $b$, and then descends to $b$ (note that the LCA can also be any of the nodes $a$ and $b$!). Computing the LCA of two vertices is a well-known problem and may be solved in several different ways. One possible approach is to use the equivalence between LCA and the range minimum query (RMQ) problem and then compute the LCA of any two vertices in constant time. Another one is based on heavy path decomposition. In any case, we need to be able to compute the LCA in $O(1)$ time.

Let us now define the notion of a **banned (directed) edge**: a directed edge $a \to b$ is *banned* if it requires paying a bribe. If $a$ is the parent of $b$ for a banned edge $a \to b$, then we call $a \to b$ a **down-banned edge**. Similarly, we may define **up-banned edges**. If Borna traveled along a banned edge $p$ times, then he will have to prepare $1 + 2 + \dots + 2^{p-1} = 2^p - 1$ thousands of dinars for bribing the police. Hence we need to determine the number of times every edge was traversed. This depends on whether the edge is down-banned or up-banned.

Before delving into these two cases, we need to compute the following three properties for every town $x$:

- $ends\_down$: the number of times $x$ was the final stop in a path, this is equal to the number of occurrences of $x$ in the array of stops;
- $ends\_up$: the number of times $x$ was the highest stop in a path, this is equal to the number of times $x$ was the LCA of two consecutive stops;
- $gone\_up$: the number of times $x$ was the first stop in a path.

Now we consider the cases:

- If an edge $a \to b$ is up-banned, then the number of times it was traversed is equal to the number of times any vertex in $a$'s subtree was an initial stop, minus the number of times any vertex in $a$'s subtree was the highest stop (i.e. sum of all $gone\_up$'s minus the sum of all $ends\_up$'s). We may compute these parameters for all vertices at once using just one post-order tree traversal. Thus we can compute the 'bribe contributions' of all up-banned edge in linear time.
- If an edge $a \to b$ is down-banned, then the number of times it was traversed is equal to the number of times any vertex in $b$'s subtree was a final stop, minus the number of times any vertex in $b$'s subtree was the highest stop (i.e. sum of all $ends\_down$'s minus the sum of all $ends\_up$'s). Similar to the previous case, we can compute the 'bribe contributions' of all down-banned edges using only one post-order tree traversal.

Hence, by first computing $ends\_down$, $ends\_up$ and $gone\_up$ for every vertex, and then traversing the tree, we are able to compute the answer. The final complexity depends on the implementation of LCA. The asymptotically optimal solution to this problem has $O(N + K)$ time complexity, but even an $O(N \log N + K)$ approach is acceptable given these constraints.

# Problem C: Party

*Authors*

***Ibragim Ismailov***

*Implementation and analysis*

***Ibragim Ismailov***

People working in MDCS (Microsoft Development Center Serbia) like partying. They usually go to night clubs on Friday and Saturday.

There are $N$ people working in MDCS and there are $N$ clubs in the city. Unfortunately, if there is more than one Microsoft employee in night club, level of coolness goes infinitely high and party is over, so club owners will never let more than one Microsoft employee enter their club in the same week (just to be sure).

You are organizing night life for Microsoft employees and you have statistics about how much every employee likes Friday and Saturday parties for all clubs.

You need to match people with clubs maximizing overall sum of their happiness (they are happy as much as they like the club), while **half** of people should go clubbing on Friday and the other **half** on Saturday.

### Input:

The first line contains integer $N$ – number of employees in MDCS.

Then an $NxN$ matrix follows, where element in $i$-th row and $j$-th column is an integer number that represents how much $i$-th person likes $j$-th club's **Friday** party.

Then another $NxN$ matrix follows, where element in $i$-th row and $j$-th column is an integer number that represents how much $i$-th person likes $j$-th club's **Saturday** party.

### Output:

Output should contain a single integer – maximum sum of happiness possible.

### Constraints:

- $2 \leq N \leq 20$
- $N$ is even
- $0 \leq level\ of\ likeness \leq 10^6$
- All values are integers

### Example input:

```
4
1 2 3 4
2 3 4 1
3 4 1 2
4 1 2 3
5 8 7 1
6 9 81 3
55 78 1 6
1 1 1 1
```

### Example output:

```
167
```

### Explanation:

Here is how we matched people with clubs:
Friday: 1st person with 4th club (4 happiness) and 4th person with 1st club (4 happiness).
Saturday: 2nd person with 3rd club (81 happiness) and 3rd person with 2nd club (78 happiness)
4+4+81+78 = 167

**Time and memory limit: 1.5s / 4MB**

## *Solution and analysis:*

This problem is a variation of the well-known *assignment problem.* More about the problem can be found on the *Wikipedia* article - https://en.wikipedia.org/wiki/Assignment_problem.

First, notice that the memory limit is very low. This makes it *almost* impossible to write a dynamic programming solution. So, let's look at a different approach.

The most naïve solution would be going through $\binom{N}{N/2}$ combinations of assignments – half of people on Friday and the other half on Saturday. For every combination, we run *Hungarian algorithm* and find the best answer among all of the combinations. Although the memory complexity of the algorithm is only $O(N^2)$, the time complexity of this solution is $O(\binom{N}{N/2}N^3)$, since *Hungarian algorithm* has $O(N^3)$ time complexity. This is too slow.

To explain the solution of this problem let's describe the scheme of *Hungarian algorithm*:

```
HungarianAlgorithm(…)
{
	for (int i = 0; i < n; i++)
	{
		hungarian_iteration();
	}
}
```

*Hungarian algorithm* allows rows of the assignment matrix to be added one by one, in $O(N^2)$ time complexity each. To solve our problem, we will recursively go through all sets of binary masks with equal number of 0s and 1s, where 0 in $i$th position means that the $i$th person would go partying on Friday, while 1 denotes a person going partying on Saturday. In each recursive call, we will add a row to the solution for the binary mask we are currently generating. Time complexity of the solution is $O(\binom{N}{N/2}N^2)$, because the number of recursive calls is $O(\binom{N}{N/2})$. This is enough to solve the task within the constraints.

Curiosity here is that the number of states we have to visit during our recursion is closely related to problem *H. Bots.* You can find a detailed explanation in the analysis of that problem.

# Problem D: Tablecity

*Authors* | *Implementation and analysis*

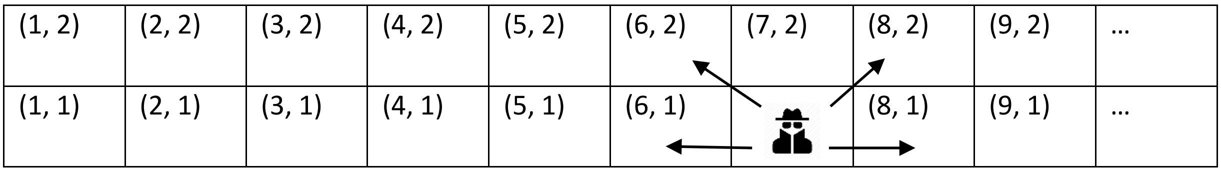
***Aleksandar Damjanović*** | ***Stefan Stojanović***

There was a big bank robbery in Tablecity. In order to catch the thief, the President called none other than Albert – Tablecity's Chief of Police. Albert does not know where the thief is located, but he does know how he moves.

Tablecity can be represented as $1000\ x\ 2$ grid, where every cell represents one district. Each district has its own unique name "$(X, Y)$", where X and Y are the coordinates of the district in the grid. The thief's movement is as follows:

**Every hour** the thief will **leave** the district $(X, Y)$ he is currently hiding in, and **move** to one of the districts: $(X-1, Y)$, $(X+1, Y)$, $(X-1, Y-1)$, $(X-1, Y+1)$, $(X+1, Y-1)$, $(X+1, Y+1)$ as long as it exists in Tablecity.

Below is an example of thief's possible movements if he is located in district $(7, 1)$:



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| (1, 2) | (2, 2) | (3, 2) | (4, 2) | (5, 2) | (6, 2) | (7, 2) | (8, 2) | (9, 2) | … |
| (1, 1) | (2, 1) | (3, 1) | (4, 1) | (5, 1) | (6, 1) | | (8, 1) | (9, 1) | … |

Albert has enough people so that **every hour** he can pick any **two** districts in Tablecity and fully investigate them, making sure that if the thief is located in one of them, he will get caught. Albert promised the President that the thief will be caught in **no more** than 2015 hours and needs your help in order to achieve that.

### Input:

There is no input for this problem.

### Output:

The first line of output contains integer $N$ – duration of police search in hours. Each of the following $N$ lines contains exactly 4 integers $X_{i1}\ Y_{i1}\ X_{i2}\ Y_{i2}$ separated by spaces, that represent 2 districts $(X_{i1},\ Y_{i1})$ and $(X_{i2},\ Y_{i2})$ which got investigated during $i^{th}$ hour. Output is given in **chronological** order ($i^{th}$ line contains districts investigated during $i^{th}$ hour) and should **guarantee** that the thief is caught in no more than 2015 hours, **regardless of thief's initial position and movement**.

### Constraints:

- $1 \leq N \leq 2015$
- $1 \leq X \leq 1000, 1 \leq Y \leq 2$

### Example input:

No example input

### Example output:

```
2
5 1 50 2
8 1 80 2
```

### Explanation:

Example output is not guaranteed to catch the thief and is not correct. There exists a combination of an initial position and a movement strategy such that the police will not catch the thief.

Consider the following initial position and thief's movement:
In the first hour, the thief is located in district $(1, 1)$. Police officers will search districts $(5, 1)$ and $(50, 2)$ and will not find him.
At the start of the second hour, the thief moves to district $(2, 2)$. Police officers will search districts $(8, 1)$ and $(80, 2)$ and will not find him.
Since there is no further investigation by the police, the thief escaped!

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

### *Solution and analysis:*

First notice that parity of thief's $X$ coordinate changes every time he moves.

Assume that at the beginning $X$ coordinate of thief's position is odd, and check districts $(1, 1)$ and $(1, 2)$. The next day check districts $(2, 1)$ and $(2, 2)$ and so on until $1000^{th}$ when you check districts $(1000, 1)$ and $(1000, 2)$. What is achieved this way is that if starting parity was as assumed, thief could have never moved to district with $X$ coordinate $i$ on day $i + 1$, hence he couldn't have jumped over the search party and would've been caught. If he wasn't caught, his starting parity was different than we assumed, so on $1001^{st}$ day we search whatever ($1$ and $1001$ are of the same parity, so we need to wait one day), and then starting on $1002^{nd}$day we do the same sweep from $(1, 1)$ and $(1, 2)$ to $(1000, 1)$ and $(1000, 2)$ and guarantee to catch him.

Shortest possible solution is by going from $(2, 1)$ and $(2, 2)$ to $(1000, 1)$ and $(1000, 2)$ twice in a row, a total of 1998 days, which is correct in the same way. First sweep catches the thief if he started with even $X$ coordinate, and second sweep catches the thief if he started with odd $X$ coordinate.

# Problem E: Spectator Riots

*Authors*

***Borna Vukorepa***

*Implementation and analysis*

***Borna Vukorepa***

It's riot time on football stadium Ramacana! Raging fans have entered the field and the police find themselves in a difficult situation. The field can be represented as a square in the coordinate system defined by two diagonal vertices in $(0,0)$ and $(10^5, 10^5)$. The sides of that square are also considered to be **inside** the field, everything else is **outside**.

In the beginning, there are $N$ fans on the field. For each fan we are given his speed, an integer $v_i$ as well as his **integer coordinates** $(x_i, y_i)$. A fan with those coordinates might move and after one second he might be at any point $(x_i + p,\ y_i + q)$ where $0 \leq |p| + |q| \leq v_i$. $p, q$ are both **integers**.

Points that go **outside** of the square that represents the field are excluded and all others have **equal probability** of being the location of that specific fan after one second.

Andrej, a young and promising police officer, has sent a flying drone to take a photo of the riot from above. The drone's camera works like this:

1) It selects three points with **integer coordinates** such that there is a chance of a fan appearing there after one second. They must **not** be collinear or the camera won't work. It is guaranteed that **not all** of the initial positions of fans will be on the same line.
2) Camera focuses those points and creates a circle that passes through those three points. A photo is taken after **one second** (one second after the initial state).
3) Everything that is **on the circle or inside it** at the moment of taking the photo (one second after focusing the points) will be on the photo.

Your goal is to select those three points so that the **expected number** of fans seen on the photo is maximized. If there are more such selections, select those three points that give the circle with **largest radius** among them. If there are still more suitable selections, **any one** of them will be accepted. If your answer follows conditions above and radius of circle you return is smaller then the optimal one by $0.01$, your output will be considered as correct. No test will have optimal radius bigger than $10^{10}$.

### Input:

The first line contains the number of fans on the field, $N$. The next $N$ lines contain three integers: $x_i, y_i, v_i$. They are the x-coordinate, y-coordinate and speed of fan $i$ at the beginning of the one second interval considered in the task.

### Output:

You need to output the three points that camera needs to select. Print them in three lines, with every line containing the x-coordinate, then y-coordinate, separated by a single space. The order of points does not matter.

### Constraints:

- $3 \leq N \leq 10^5$
- $0 \leq x_i, y_i \leq 10^5$
- $0 \leq v_i \leq 1000$
- All numbers will be integers

### Example input:

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

### Example output:

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

### Explanation:

The circle defined in output will catch all of the fans, no matter how they move during one second.

**Time and memory limit: 0.5s / 128MB**

### *Solution and analysis:*

First we simplify the problem by replacing the initial set of points with the set of all points where some fan might appear after one second, call that set $S$. Every point in $S$ has a probability that a specific player will appear there. Consequently, for every point $P$ in $S$ we know the expected number of fans at it after one second, call it $P_e$.

Now, for some arbitrary circle that passes through some three points of $S$ (which doesn't violate the rules of the problem), the expected number of fans caught on camera is the sum of all $T_e$, where $T$ is a point on or inside the circle. Our goal is to find a circle that maximizes that sum.

After drawing a few examples, we can notice that we can always catch most of the points that are possible locations of some fan or even all of them. We can write a brute-force solution that will increase our suspicion that all fans can be caught, no matter how they move.

Now let's try to find the largest circle of those that surely catch all fans and don't violate the rules in the problem. It is easy to see that three fixed points that determine the circle must lie on the convex hull of $S$ (otherwise we surely wouldn't catch all points of $S$ with that circle).

Convex hull can be computed in $O(|S|log(|S|))$, which might be too slow if unnecessary points are not eliminated from $S$.

Notice that for every fan in input, if his speed is $v$, he might appear at $O(v^2)$ points, so convex hull algorithm would have $O(Nv^2log(Nv))$ complexity, which is too slow.

The trick is to take only convex hull of those $O(v^2)$ points, which will have $O(1)$ points. All other points should be eliminated from $S$ as they don't have a chance of appearing on convex hull of $S$. Contestants need to be careful with edge cases when a fan potentially goes out of the field.

After computing the convex hull of $S$ (call it $H(S)$), we hope to find the circle that will pass through some three points on that hull and contain all other points inside it or on it.

These two claims can be proven geometrically:
1) For a convex polygon, the largest circle among all circumcircles of triangles determined by the polygon

vertices will surely contain all vertices of the polygon on it or inside it.
2) For a convex polygon, the largest circumcircle of some triangle that is determined by vertices of the polygon is a circumcircle of a triangle that contains three consecutive vertices of a polygon.

With 1) and 2) we conclude:

The largest circle among those that are circumscribed around triangles that are composed of three consecutive vertices of $H(S)$ contains all of the points of $H(S)$ (and then obviously of $S$) and no other circle that contains all those points can be larger.

This means that we can finish the problem easily in linear time (with respect to the size of convex hull).

# Problem F: Bulbo

*Authors*

***Vanja Petrović Tanković***

*Implementation and analysis*

***Aleksandar Milovanović***

Bananistan is a beautiful banana republic. Beautiful women in beautiful dresses. Beautiful statues of beautiful warlords. Beautiful stars in beautiful nights.

In Bananistan people play this crazy game – Bulbo. There's an array of bulbs and each player has a position, which represents one of the bulbs. Distance between two neighboring bulbs is 1. Each turn one contiguous set of bulbs lights-up, and players have the cost that's equal to the distance from the closest shining bulb. Then all bulbs go dark again. Before each turn players can change their position with $|pos_{new} - pos_{old}|$ cost, and players know three next light-ups. The goal is to minimize your summed cost. I tell you, Bananistanians are spending their nights playing with bulbs.

Banana day is approaching, and you are hired to play the most beautiful Bulbo game ever. A huge array of bulbs is installed, and you know your initial position and all the light-ups in advance. You need to play the ideal game and impress Bananistanians, and their families.

### Input:

The first line contains number of turns $n$ and initial position $x$. Next $n$ lines contain two numbers $l_{start}$ and $l_{end}$, which represent that all that bulbs from $[l_{start}, l_{end}]$ interval are shining this turn.

### Output:

Output should contain a single number which represents the best result (minimum cost) that could be obtained by playing this Bulbo game.

### Constraints:

- $1 \le n \le 5000$
- $1 \le x \le 10^9$
- $1 \le l_{start} \le l_{end} \le 10^9$

### Example input:

```
5 4
2 7
9 16
8 10
9 17
1 6
```

### Example output:

```
8
```

### Example play:

Before 1. turn move to position 5
Before 2. turn move to position 9
Before 5. turn move to position 8

**Time and memory limit: 1s / 64MB**

## *Solution and analysis:*

First, let's consider solution in $O(n * maxX)$ complexity. It's a simple dynamic programming approach.

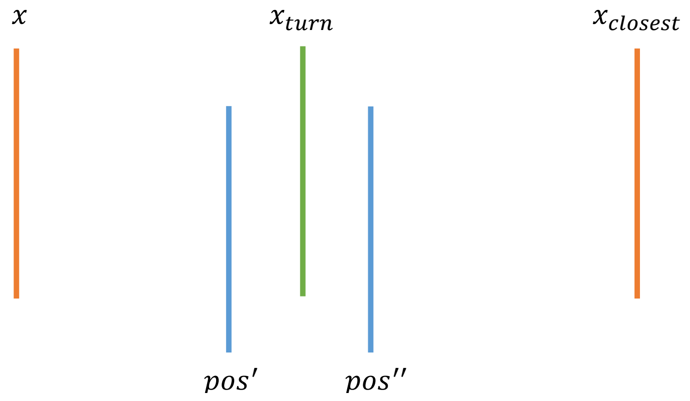
We have an array $dyn$ (size $maxX$), which should say how costly would it be to be in this position at the end of the turn. At initialization step each position is set to $maxValue$ (some big number), except the initial position which is set to be $0$. This should represent that in the initial moment it's impossible to be anywhere except in the initial position.

Before each turn we can change position, so before each turn we could calculate the best cost of being in that position when the bulbs light up. We can do this by passing this array twice – once from the left and once from the right. Consider the case we're passing from left to right (smaller index $i$ to larger index $i$, the other case could be explained in the similar fashion). While passing, we can keep the $bestVal$, which would be initiated to $dyn[0]$, and updated as $bestVal = min(bestVal + 1, dyn[i])$, $dyn[i] = bestVal$ for each $i > 0$. Rationale: if we came from the $i - 1$ position, we have the same cost of that position $+1$. If we didn't, we have the same cost we had before this. Now we only need to add distance to the closest bulb for each position and we finished this turn. When we finish each turn we pick the lowest value in the array, and that's our solution. Simple enough.

But this solution is too slow for us. We want more.

Statement: We never have to move to the position which is not beginning or the end position of one of the light-ups.

Let's consider following situation: We're at the position $x$. If $x$ is going to be inside of the next turn shining bulbs, there's no point of moving at all (it would be the same as moving after the turn). So, consider $x$ is outside those bulbs, and $x_{closest}$ is the closest bulb to $x$ that will be lighten-up. Also, consider $x < x_{closest}$ (the other case could be explained in the similar fashion). Consider all the remaining light-up points (light-up beginning and end positions) are in the array $pos$, which is sorted. Take a look at the following picture:



First thing we could notice is the fact that our cost for this turn is going to be $x_{closest} - x$, if we finish the turn anywhere between $x$ and $x_{closest}$ inclusive. Going left from $x$ or right from $x_{closest}$ doesn't make any sense, because we would have same or bigger cost than staying in $x$ or $x_{closest}$ and moving from it in the next turn.

Next, let's consider we haven't ended our turn on some light-up endpoint, but between two neighboring endpoints, $pos'$ and $pos''$. Let's call that position $x_{turn}$. Let's also introduce $x_{closest2}$, which is the closest bulb from the $x_{turn}$ in the next turn light-up.

If $x_{turn}$ is shining, then $pos'$ is shining as well, so we could have finished our turn there. If $x_{closer2} \leq pos'$ we would be better off or equally well if we finished our turn in $pos'$. In that case we would have $x_{turn} -$

$pos'$ smaller cost for the next turn. If afterwards we need to go to $x_{turn}$, total cost would not exceed the cost of going straight to $x_{turn}$ in the initial turn. If $x_{closer2} \geq pos''$ we would be better off or equally well if we finished our turn in $pos''$, similar to the explanation for $pos'$. So, in each turn we could stay in the place or go to the closest light-up endpoint and we could still get the optimal solution.

We can use this fact to make a $O(n^2)$ solution – instead of each position we should take consider only light-up endpoints and initial position. Everything else is the same as in original solution.

Dynamic programming solution is enough to pass within the constraints for the program, but this problem can be solved in linear time as well.

Let's look at the values of array $dyn$. We can notice that this array actually has only one "local minimum" at each turn. What this means is that we have a range $[l, r]$ and that all of the values from $dyn[0]$ to $dyn[l]$ are monotonically decreasing, all values from $dyn[r]$ to $dyn[10^9]$ are monotonically increasing, while all of the values $dyn[l], dyn[l+1], \dots, dyn[r]$ have the same value and represent the minimum summed cost until this turn. We can use this property to create a linear time algorithm.

Our linear algorithm will be as follows:

At the beginning, our optimal range will be $[x_{start}, x_{start}]$ and minimum cost will be $0$. At each turn, we will update this optimal range and minimum cost.

If the range of shining bulbs in the next turn intersects with our optimal range, we can easily see that our new optimal range will be this intersection. The minimum cost will stay the same as the cost for previous turn, since we don't need to move if we are located somewhere in this intersection, as we will already be located at a bulb that is shining. Anywhere outside of this intersection, the cost would increase since either the distance to the closest shining bulb would be larger than $0$, or because of moving from our optimal range to somewhere outside of it, or both.

If the range of shining bulbs in the next turn does not intersect our optimal range and is left from it, we will set that our optimal range starts from the rightmost shining bulb and ends at the left end of our previously optimal range. Our minimum cost will increase for exactly the distance between these positions – if we don't move from the left end of our previously optimal range, our cost increases for this distance. If we move from the left end of our previously optimal range by one position to left, we decrease our distance in the next turn by $1$, but increase the cost by $1$ because of the move. Same goes for moving two positions to left, and so on until the movement to the rightmost shining bulb in the next turn (at which point our distance to shining bulb will be $0$, but our cost for moving will be the same as distance when we didn't move at all). It is easily seen that the minimum cost for a position that is left from this new optimal range is larger by $1$, and the minimum cost for a position that is right from this new optimal range is also larger by $1$. Moving further to the left or right, this cost increases more and more, resulting in only one "local minimum" as we described before.

If the range of shining bulbs does not intersect our optimal range and is right from it, we can do a similar thing as we do when the range is left from our optimal range.

After all the turns, we have our optimal range and the minimum cost possible. The total complexity is $O(n)$, since in each turn we update the range in $O(1)$.

## Problem G: Run for beer

*Authors*

***Alexandr Lyashko***

*Implementation and analysis*

***Andrija Jovanović***

People in BubbleLand like to drink beer. Little do you know, beer here is so good and strong that every time you drink it your speed goes $10$ times slower than before you drank it.

Birko lives in city Beergrade, but wants to go to city Beerburg. You are given a road map of BubbleLand and you need to find the fastest way for him. When he starts his journey in Beergrade his speed is $1$. When he comes to a new city he always tries a glass of local beer, which divides his speed by $10$.

The question here is what the minimal time for him to reach Beerburg is. If there are several paths with the same minimal time, pick the one that has least roads on it. If there is still more than one path, pick any.

It is guaranteed that there will be at least one path from Beergrade to Beerburg.

### Input:

The first line of input contains integer $N$ – number of cities in Bubbleland and integer $M$ – number of roads in this country. Cities are enumerated from $0$ to $N-1$, with city $0$ being Beergrade, and city $N-1$ being Beerburg.
Each of the following $M$ lines contains three integers $a, b$ ($a \neq b$) and $len$. These numbers indicate that there is a bidirectional road between cities $a$ and $b$ with length $len$.

### Output:

The first line of output should contain minimal time needed to go from Beergrade to Beerburg.
The second line of the output should contain the number of cities on the path from Beergrade to Beerburg that takes minimal time.
The third line of output should contain the numbers of cities on this path in the order they are visited, separated by spaces.

### Constraints:

- $2 \leq N \leq 10^5$
- $1 \leq M \leq 10^5$
- $0 \leq len \leq 9$
- There is at most one road between two cities

### Example input:

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

### Example output:

```
32
3
0 3 7
```

**Time and memory limit: 0.5s / 64MB**

### *Solution and analysis:*

Let's first decipher the text of the problem to help us figure out which set of conditions the solution needs to satisfy.

We'll notice that the maximum direct distance between any two cities is 9, and the speed decrease from each step to the next is 10x. This gives us an important pieces of insight:

If we have two paths, neither of which ends with a 0-length edge, the shorter path will always be faster than a longer path – if the longer path has $k$ steps and the shorter one has $l$, its last step will take at least $10^k$ time, while the entire shorter path cannot take more than $10^{l+1} - 1 \leq 10^k - 1$ time.

From this follows a slightly more general statement:

If we have two paths, the one which is shorter after we trim all trailing zeros from both will be faster. Furthermore, if two paths have the same length after trimming trailing zeros, the one with the smaller distance between the last two cities will be faster than the other. If the two last distances are the same, the one with the shorter second-to-last edge will be faster, etcetera.

It turns out that there is only one case in which we care about trailing zeros: the problem statement says that from two paths with equal time the preferred one should be the one visiting less cities – the one that has a smaller number of zeros at the end. This gives us a way to simplify the problem: we will start from city $n - 1$ (Beerburg) and calculate which other cities can reach it in 0 time and with how many steps. This can be done using a breadth-first search that starts from Beerburg and ignores all non-zero length paths. Let's call this set of vertices $Z$.

After this, we can run another breadth-first search, this time starting from Beergrade, and find the shortest path to a city belonging to $Z$. Having in mind our conclusions above, if there is exactly one such path we can be certain that's the answer to the problem. Unfortunately, life is not so simple and there's no guarantee that there will be just one shortest path. So what do we do now?

First, we can assign to each vertex its shortest distance (in terms of the number of edges) from Beergrade – this is easy to do within the breadth-first search algorithm. Let's denote with $k$ the least number of steps required to reach a vertex in $Z$ when starting from Beergrade. Then we can apply the following reasoning:

Consider all edges connecting a vertex in $Z$ to an edge in the set of vertices $V_{k-1}$ that are exactly $k - 1$ steps away from Beergrade, and then out of them consider the ones that have minimal length. We'll call the set of cities that are connected to $Z$ through one of these edges $S_{k-1}$, and for each city $v \in S_{k-1}$ we'll save two pieces of information:

- $u_v$ – the vertex in $Z$ they are connected to through a minimal-length edge
- $t_v$ - the minimal number of trailing zeros after they've reached a vertex in $Z$.

What is so special about the set $S_{k-1}$? Well, it turns out that the optimal solution has to pass through some vertex $v$ in $S_{k-1}$ and then follow its edge $u_v$ to a vertex that's 0 time away from the goal. This is simple to prove: let's assume the opposite is true. Then we'd have to be able to find a path from Beergrade to $Z$ with less than $k$ steps, or a path with exactly $k$ steps that has a shorter last step than all edges connecting $S_{k-1}$ to $Z$, or a path with the same number of steps, the same final step and a smaller number of trailing zeros. However, we've constructed the set $S_{k-1}$ precisely in a way such that none of these conditions can be satisfied.

We can now construct sets $S_{k-2}, S_{k-3}, \dots S_0$ in a similar manner:

- Let $V_i$ be the set of vertices that are exactly $i$ steps away from Beergrade.
- Take all edges connecting the set $S_{i+1}$ to $V_i$. Among those edges, take the ones with minimal length.
- The set $S_i$ is the set of vertices in $V_i$ on the end of edges in the previous step.
- For each vertex $v$ in $S_i$, we save the city $u_v$ from $S_{i+1}$ they are connected to through a minimal-length edge, and $z_u$ = $z_v$, the number of trailing zeros at the end of its path (if there are multiple candidates for $u$, break ties in favour of the one with the smallest $z_u$).

A simple inductive proof similar to the one we've made for the set $S_{k-1}$ will convince us that the optimal solution has to pass through a city in each $S_i$. Finally, the set $S_0$ contains just one city: Beergrade. This gives us the solution, and we just have to iterate through the trail of cities $u$ to reconstruct it.

Let's calculate the complexity of this solution. Its first two steps are simple breadth-first searches, with complexity $O(m+n)$. How complex is the process of constructing the sets $S_i$? We can do it by performing a constant number of operations on each city in $S_{i+1}$ and each edge that has one end in $S_{i+1}$ – we check whether its other end has distance $i$ from Beergrade, we find the minimum length in the set of edges satisfying the previous condition, and we compare each edge to the minimum. So the complexity of constructing $S_i$ is $O(|S_{i+1}|) + O(\sum_{v\in S_{i+1}} \deg(v))$. Since no vertex can belong to more than one set $Z_i$, the total complexity is $O(\sum_i |S_i|) + O(\sum_{v\in S_{i+1}, i\in 0..k} \deg(v)) \leq O(n) + O(\sum \deg v) = O(n) + O(2m) = O(n+m)$. Recontructing the path can then take up to $O(n)$ time, and the final complexity is $O(m + n)$.

In terms of memory complexity, we only need a constant additional amount of memory per each node, so the memory needed to hold the graph in memory dominates – this is also $O(m+n)$.

# Problem H: Bots

*Authors*

***Ibragim Ismailov***

*Implementation and analysis*

***Ibragim Ismailov***

Sasha and Ira are two best friends. But they aren't just friends, they are software engineers and experts in artificial intelligence. They are developing an algorithm for two bots playing a two-player game. The game is cooperative and turn based. In each turn, one of the players makes a move (it doesn't matter which player).

Algorithm for bots that Sasha and Ira are developing works by keeping track of the state the game is in. Each time either bot makes a move, the state changes. And, since the game is very dynamic, it will never go back to the state it was already in at any point in the past.

Sasha and Ira are perfectionists and want their algorithm to have an optimal winning strategy. They have noticed that in the optimal winning strategy, both bots make exactly $N$ moves each. But, in order to find the optimal strategy, their algorithm needs to analyze all possible states of the game (they haven't learned about alpha-beta pruning yet) and pick the best sequence of moves.

They are worried about the efficiency of their algorithm and are wondering what is the total number of states of the game that need to be analyzed?

### Input:

The first and only line contains integer N.

### Output:

Output should contain a single integer – number of possible states modulo $10^9 + 7$.

### Constraints:

- $1 \leq N \leq 10^6$

### Example input:

2

### Example output:

19

### Explanation:

Start: Game is in state A.

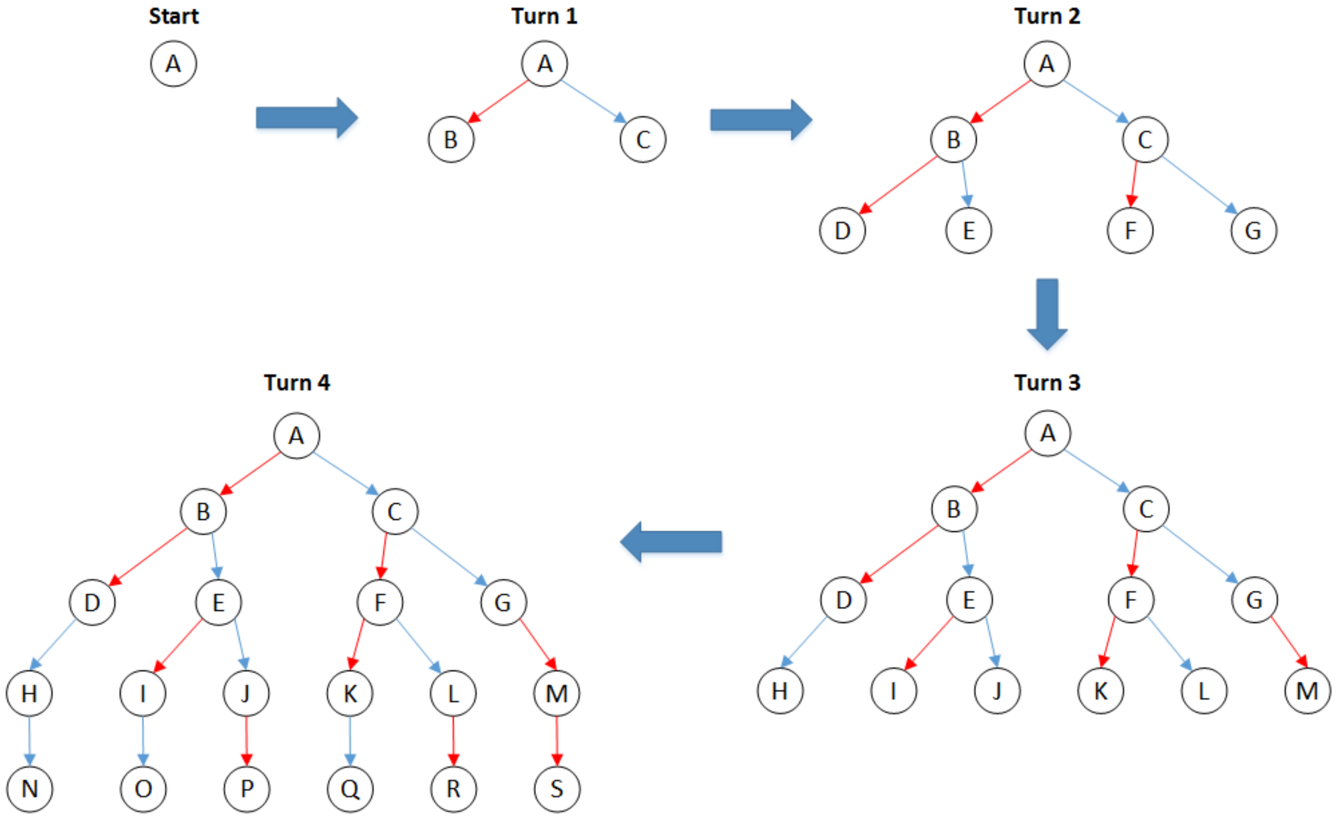
Turn 1: Either bot can make a move (first bot is red and second bot is blue), so there are two possible states after the first turn – B and C.

Turn 2: In both states B and C, either bot can again make a turn, so the list of possible states is expanded to include D, E, F and G.

Turn 3: Red bot already did $N = 2$ moves when in state D, so it cannot make any more moves there. It can make moves when in state E, F and G, so states I, K and M are added to the list. Similarly, blue bot cannot make a move when in state G, but can when in D, E and F, so states H, J and L are added.

Turn 4: Red bot already did $N = 2$ moves when in states H, I and K, so it can only make moves when in J, L and M, so states P, R and S are added. Blue bot cannot make a move when in states J, L and M, but only when in H, I and K, so states N, O and Q are added.

Overall, there are 19 possible states of the game their algorithm needs to analyze.



**Time and memory limit: 2s / 64MB**

## *Solution and analysis:*

Problem can be transformed to a more formal one: how many vertices will a *trie* (data structure) contain, if we add all possible strings of length $2N$ with $N$ zeroes and $N$ ones to this *trie*.

First, it is obvious that the upper half of this tree will be a full binary tree. Let's take a look at $N = 3$:

- level $0$: $1$ vertex
- level $1$: $2$ vertices
- level $2$: $4$ vertices
- level $3$: $8$ vertices.

Starting from level $N + 1$, not every vertex will have two children. Vertices that will have two children are those that have less than $N$ zeroes and less than $N$ ones on the path to the root.

So, here is how we calculate how many vertices there will be on level $i + 1$:

- Let's define $S_i$ as number of vertices on level $i$ which have less than two children
- If $V_i$ is number of vertices on level $i$, then $V_{i+1} = 2V_i - S_i$
- $S_i$ can be calculated easily using binomial coefficients: $S_i = 2\binom{i}{N}$

Everything else is in the task are implementation techniques. We need to calculate some modular multiplicative inverses, which can be done in logarithmic time using exponentiation by squaring. We can calculate binomial coefficients efficiently, by using the values calculated for the previous level.
Time complexity of the solution is $O(Nlog(10^9 + 7))$.

# Problem I: Robots protection

*Authors* *Implementation and analysis*

***Alexandr Lyashko*** ***Predrag Ilkić***

Company “Robots industries” produces robots for territory protection. Robots protect triangle territories – right isosceles triangles with catheti parallel to North-South and East-West directions.

Owner of some land buys and sets robots on his territory to protect it. From time to time, businessmen want to build offices on that land and want to know how many robots will guard it. You are to handle these queries.

### Input:

The first line contains integer $N$ – width and height of the land, and integer $Q$ – number of queries to handle.

Next $Q$ lines contain queries you need to process.

Two types of queries:

1. $1\ dir\ x\ y\ len$ – add a robot to protect a triangle. Depending on the value of $dir$, the values of $x, y$ and $len$ represent a different triangle:
   - $dir = 1$: Triangle is defined by the points $(x, y), (x + len, y), (x, y + len)$
   - $dir = 2$: Triangle is defined by the points $(x, y), (x + len, y), (x, y - len)$
   - $dir = 3$: Triangle is defined by the points $(x, y), (x - len, y), (x, y + len)$
   - $dir = 4$: Triangle is defined by the points $(x, y), (x - len, y), (x, y - len)$
2. $2\ x\ y$ – output how many robots guard this point (robot guards a point if the point is inside or on the border of its triangle)

### Output:

For each second type query output how many robots guard this point. Each answer should be in a separate line.

### Constraints:

- $1 \le N \le 5000$
- $1 \le Q \le 10^5$
- $1 \le dir \le 4$
- All points of triangles are within range $[1, N]$
- All numbers are positive integers

### Example input:

```
17 10
1 1 3 2 4
1 3 10 3 7
1 2 6 8 2
1 3 9 4 2
2 4 4
1 4 15 10 6
2 7 7
2 9 4
2 12 2
2 13 8
```

### Example output:

```
2
2
2
0
1
```

**Time and memory limit: 1s / 512MB**

### *Solution and analysis:*

To solve this problem, we should first take a look at one easier problem. Consider having the same problem statement, but with rectangles instead of triangles. The solution to this problem is pretty straightforward. We will store a matrix representing points in our coordinate system. For each type 1 query, given the rectangle $((x_{ul}, y_{ul}), (x_{ur}, y_{ur}), (x_{ll}, y_{ll}), (x_{lr}, y_{lr}))$, we would add -1 to the points $(x_{ul}, y_{ul} + 1)$ and $(x_{lr} + 1, y_{lr})$, and 1 to the points $(x_{ll}, y_{ll})$ and $(x_{ur} + 1, y_{ur} + 1)$. Notice that we expanded the given rectangle when adding +1s and -1s, this is because the point is considered in the rectangle even when it is on the border! This way, when type 2 query is received, to find the answer we simply sum every value in the rectangle $(0, 0)$ to $(x, y)$. Since simply summing the points in rectangles is $O(n^2)$, we should use binary indexed tree for this, hence getting the sufficient time complexity $O(\log^2 n)$ per query.

We will use this approach with some modification to solve the original problem. For handling the triangles we need to introduce new coordinate systems. Depending on the type of triangle (types 1 and 4 are analogous, so are types 2 and 3) we will make two new coordinate systems, where the corresponding hypotenuses are parallel to one axis. For types 2 and 3, point $(x, y)$ in original coordinate system would map to $(x + y, y)$, and for types 1 and 4, point $(x, y)$ would map to $(x + n - y - 1, y)$, where $n$ is the size of coordinate plane given in the input.

The problem is now somewhat abstracted to the simple rectangle problem. This time we will need three matrices, one representing original coordinate system and two representing the introduced coordinate systems. For each triangle we need to border it with +1s and -1s, similarly as in the rectangle case, only using 2 matrices for each triangle. When adding border for the cathetus we use the original coordinate system and for the hypotenuse we need one of the two introduced coordinate systems, depending on the type of the triangle. Remember, the point belongs to the triangle even when it is on one of catheti or hypotenuse, so be careful.

For calculating the answer for type 2 query, we need to sum the values in the rectangle from $(0, 0)$ to $(x, y)$ in all three matrices (of course, $(x, y)$ needs to be mapped accordingly to the coordinate system each matrix represent). Again, to not exceed time limit we need to use binary indexed trees.

Since we are using binary indexed tree, the time complexity of this solution is $O(Q * \log^2 N)$.

# Qualifications

As in previous years, the qualifications were split into two rounds, with ten problems in each round. Although the finals were in the traditional format with no open problems, the problem set in qualifications contained challenge problems again. Non-challenge problems were worth 1 point in the first round and 2 points in the second round. Challenge problem in the first round was worth maximum of 4 points, while challenge problem in the second round was worth maximum of 8 points.

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

This year, 77 teams managed to solve at least one problem from the qualifying rounds. Competitors from all around the world participated in the qualifications. Teams that qualified for the finals were from Serbia, Croatia, Poland, Latvia, Ukraine, Belarus, and Russia.

| Num | Problem name | ID | Accepted solutions |
|---|---|---|---|
| 01 | Matts Trip | 14886 | 41 |
| 02 | Estimation | 16809 | 40 |
| 03 | Substrings II | 8747 | 34 |
| 04 | Light Cycling | 14981 | 42 |
| 05 | Crossbits | 3484 | 52 |
| 06 | Font Size | 18530 | 90 |
| 07 | Michel and the championship | 10730 | 57 |
| 08 | The Indian Ocean | 16128 | 35 |
| 09 | Breaking Chocolates | 6499 | 43 |
| 10 | [CH] HQNP Incomputable Hard | 10295 | 41 |

Table 1. Statistics for Round 1

| Num | Problem name | ID | Accepted solutions |
|---|---|---|---|
| 01 | Playfair Cracker | 4476 | Not solved |
| 02 | Modern Pocket Calculator | 8542 | 25 |
| 03 | Boa viagem, Roim | 11718 | 30 |
| 04 | Biology | 7490 | 3 |
| 05 | Mickey Mouse Magic Trick v6 | 17946 | 26 |
| 06 | Electrical Engineering | 7680 | 4 |
| 07 | Turn on the lights 2 | 9535 | 5 |
| 08 | Move Marbles | 6299 | 10 |
| 09 | Perfect Maze | 9190 | 19 |
| 10 | [CH] The Secret Recipe | 18901 | 39 |

Table 2. Statistics for Round 2

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

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

## Problem R1 01: Matts Trip (ID: 14886)

Time Limit: 8.0 second

Memory Limit: 1536 MB

Matt finds himself in a desert with $N$ $(2 \le N \le 10)$ oases, each of which may have food, water, and/or a palm tree. If oasis $i$ has food, then $F_i = 1$ – otherwise, $F_i = 0$. Similarly, $W_i = 1$ if and only if oasis i has water, and $P_i = 1$ if and only if it has a palm tree. These 3 values are completely independent of one another.

Some pairs of these oases are connected by desert paths, which each take 1 hour to traverse. There are $M$ $(0 \le M \le 45)$ such paths, with path $i$ connecting distinct oases $A_i$ and $B_i$ in both directions $(1 \le A_i, B_i \le N)$. No pair of oases is directly connected by more than one path, and it's not guaranteed that all oases are connected by some system of paths.

Matt starts at an oasis $S$, and wants to end up at a different oasis $E$ $(1 \le S, E \le N)$.

Both of these oases are quite nice - it's guaranteed that $FS = WS = PS = FE = WE = PE = 1$.

Since he's in a hurry to get out of the desert, he wants to travel there in at most $H$ $(1 \le H \le 10^9)$ hours.

However, he can only survive for up to MF hours at a time without food, and up to MW hours at a time without water $(1 \le MF, MW \le 4)$. For example, if $MF = 1$ and $MW = 2$, then every single oasis he visits along the way must have food (as he would otherwise spend more than 1 hour without it), and he cannot visit 2 or more oases without water in a row.

Since Matt is a computer scientist, before actually going anywhere, he's interested in the number of different paths he can take that will get him from oasis $S$ to oasis $E$ alive in at most $H$ hours.

Note that there may be no such paths.

Being a computer scientist, he of course only cares about this number modulo $(10^9 + 7)$.

### Input

Line 1: 7 integers $N, M, H, S, E, MF$, and $MW$

Next $N$ lines: 3 integers $F_i, W_i$, and $P_i$, for $i = 1..N$

Next $M$ lines: 2 integers $A_i$ and $B_i$, for $i = 1..M$

### Output

For each test case display its case number followed by the length of the shortest path or impossible in one line.

### Samples

| Input | Output |
|---|---|
| 3 3 3 1 2 1 4<br>1 1 1<br>1 1 1<br>0 1 0<br>1 2<br>2 3<br>1 3 | 2 |

| Input | Output |
|---|---|
| 5 5 3 3 2 3 2<br>1 0 0<br>1 1 1<br>1 1 1<br>0 0 1<br>0 1 0<br>1 2<br>1 3<br>1 4<br>3 4<br>4 2 | 2 |

## *Solution:*

In order to make the problem simpler, we will first transform the graph defined by the oases, so that we no longer need to consider food and water constraints.

Each node in the new (transformed) graph represents a state in which Matt can be, defined with three values $(u, f, w)$, where $1 \leq u \leq N, 1 \leq f \leq MF, 1 \leq w \leq MW$. These correspond to the oasis he is currently in, the number of remaining steps he can take without running out of food and of water respectively. Directed edges exist between nodes whose oases are connected and who have the appropriate values for $f$ and $w$ (decreased by one if the destination oasis does not contain food/water, set to $MF/MW$ if it does). This gives us a graph with $M = N \cdot MF \cdot MW \leq 160$ nodes. Obviously, the stated problem is equivalent to finding the number of walks from the node $(S, MF, MW)$ (the "source") to $(E, MF, MW)$ (the "destination" - it is guaranteed that the destination oasis has both food and water) with at most $X$ steps.

Next, we will add a new "sink" node to this graph, with one inward edge from the destination and a self-loop. Each walk from the source to the destination with at most $X$ steps has a corresponding to a walk from the source to the sink with exactly $X + 1$ steps (the original walk + step from destination to sink + as many steps in the self-loop as it takes to reach $X + 1$ steps). Obviously, there is a bijection between these two types of walks, so finding the number of walks described in the statement is equivalent to finding the number of walks from the source to the sink with a length of exactly $X + 1$.

This can be easily solved using dynamic programming. Define $W(u, l)$ as the number of different walks from the source to the vertex $u$ with the length $l$. We have the following formula (calculated modulo $10^9 + 7$)

$$W(u, l) = \begin{cases} 0, & l = 0 \wedge u \neq source \\ 1, & l = 0 \wedge u = source \\ \sum\limits_{edge\ from\ u\ to\ v} W(v, l-1), & l > 0 \end{cases}$$

The answer is the value of $W(sink, X + 1)$.

However, the time complexity of this solution is $O(MX)$. Since $MX$ could be over $10^{11}$, this is not sufficient for solving the problem with the given time constraint.

We can rephrase the formula for calculating the values of $W$ for $l > 0$ to the following, where $A$ is the adjacency matrix of our graph ($A_{u,v}$ is equal to 1 if an edge from $u$ to $v$ exists, and to 0 if it does not):

$$W(u, l) = \sum_{v\ in\ graph} A_{u,v} W(v, l-1)$$

If we view all the values of $W(u, l)$ for one particular value of $l$ as a vector (labelled $W_l$), the above formula is equivalent to matrix multiplication, so $W_l = AW_{l-1}$ for $l > 0$. By induction, it is simple to derive the following:

$$W_l = A^l W_0 = A^l \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Since exponentiation can be done in $O(\log X)$ multiplications, each with a time complexity of $O(M^3)$, this gives us a solution with a total time complexity of $O(M^3 \log X)$.

***Added by: Jacob Plachta***
***Resource:*** *Own problem*
***Solution by:***
*Name:* ***Dimitrije Erdeljan***
*School: Gimnazija "Jovan Jovanović Zmaj", Novi Sad*
*E-mail: dimitrije.erdeljan@gmail.com*

## Problem R1 02: Estimation (ID: 16809)

Time Limit: 6.0 second

Memory Limit: 1536 MB

“There are too many numbers here!” your boss bellows. “How am I supposed to make sense of all of this? Pare it down! Estimate!” You are disappointed. It took a lot of work to generate those numbers. But, you’ll do what your boss asks. You decide to estimate in the following way: You have an array $A$ of numbers. You will partition it into $k$ contiguous sections, which won’t necessarily be of the same size. Then, you’ll use a single number to estimate an entire section. In other words, for your array $A$ of size $n$, you want to create another array $B$ of size $n$, which has $k$ contiguous sections. If $i$ and $j$ are in the same section, then $B_i = B_j$. You want to minimize the error, expressed as the sum of the absolute values of the differences $(\sum|A_i - B_i|)$.

### Input

There will be several test cases in the input. Each test case will begin with two integers on a line, $n$ $(1 \leq n \leq 2{,}000)$ and $k$ $(1 \leq k \leq 25, k \leq n)$, where $n$ is the size of the array, and $k$ is the number of contiguous sections to use in estimation. The array $A$ will be on the next $n$ lines, one integer per line. Each integer element of $A$ will be in the range from $-10{,}000$ to $10{,}000$, inclusive. The input will end with a line with two 0s.

### Output

For each test case, output a single integer on its own line, which is the minimum error you can achieve. Output no extra spaces, and do not separate answers with blank lines. All possible inputs yield answers which will fit in a signed 64-bit integer.

### Sample

| Input | Output |
|---|---|
| 7 2<br>6<br>5<br>4<br>3<br>2<br>1<br>7<br>0 0 | 9 |

### *Solution:*

First, there is one lemma we should expose and prove.

**Lemma statement**

Let's define the absolute deviation of an array A with N elements and number X as the following sum:

$$absDev(A, N, X) = \sum_{i=1}^{n} |A_i - X|$$

For indices where $A_i < X$ holds, the absolute value will be $X - A_i$, and for ones where $A_i > X$ absolute value will be $A_i - X$. The ones where $A_i = X$ $(A_i - X = 0)$ don’t affect the return value so they can be excluded. We can now rewrite initial equation as:

$$absDev(A, N, X) = NL(X) \cdot X - SL(X) + SR(X) - NR(X) \cdot X$$

Where $NL(X)$ and $NR(X)$ are numbers of $A_i$ smaller than and larger than $X$, respectfully, and $SL(X)$ and $SR(X)$ are sums of $A_i$ which are smaller than and larger than X, respectfully. The value of $X$ that minimizes absolute deviation (minimizes return value of $absDev(A, N, X)$) is the median of array $A$.

Just mentioning, median of an odd-sized array is its middle element after sorting the array, and for even-sized arrays it's the arithmetic mean of two middle elements, but for this lemma's purpose, picking any of the two middle elements would suffice. This lemma is important for the problem Estimation, because, in the optimal arrangement, we are estimating some subarray to its median.

**Lemma proof**

Let's call the number that minimizes the absolute deviation $K$. We will represent all the numbers from the array as dots on the number line, and define moving right as increasing $K$ (moving it along x-axis in positive direction), while moving left as decreasing $K$ (moving it along x-axis in negative direction). First, let's prove the lemma for odd-sized arrays.

Divide the number line in two parts, left of $X(median)$ (call it $L$), and right of $X$ (call it $R$). We want to prove that, while being in $L$, moving right only improves your solution (minimizes deviation), while moving left diminishes it.

First, $K$ belonging to $L$ implicates that $NR(K) > NL(K)$. Compare values of absolute deviation for any $K$ that belongs to $L$, and $K + dK$, where $dK$ is infinitesimal value ($dK$ doesn't have to be infinitesimal, but the proof is nicer with that). $K + dK$ should also be in $L$, and the following proof is true until $K + dK$ enters $R$. Also, we will never skip some point, and we will do the following thing until we reach the next element, and then we'll set $K$ as the following element and keep doing the same process-that way we will cover the entire range ($L$)).

$$\begin{aligned}&absDev(A, N, K + dK) - absDev(A, N, K)\\&\qquad = NL(K + dK) \cdot (K + dk) - SL(K + dK) + SR(K + dK) - NR(K + dK) \cdot (K + dK)\\&\qquad - (NL(K) \cdot K - SL(K) + SR(K) - NR(K) \cdot K)\end{aligned}$$

But, $SL, SR, NL, NR$ are same for $K$ and $K + dK$ (written above what kind of $K$ and $dK$ we're picking), so:

$$absDev(A, N, K + dK) - absDev(A, N, K) = dK \cdot (NL(K) - NR(K))$$

Because $dK > 0$ and we are in region $L\left(NL(K) < NR(K)\right)$, we get $absDev(A, N, K + dK) < absDev(A, N, K)$. So, by moving right in $L$ region we improve are solution. Similarly, in region $R$ we should move left in order to get optimal solution. Taking two conclusions together, the most optimal value for K (the peak of the function) is exactly the median of the array, $X$. Now, let's prove the same lemma for even-sized arrays.

In this case, we will have two middle elements of the array, let's call them $X_1$ and $X_2$.
Call the left part $L$, right part $R$, and the part between $X_1$ and $X_2$ as $M$. For the left part and right part, proofs are the same as with the odd-sized arrays. For left part, moving right improves solution, and for the right part, moving left improves solution. And for the middle part, because $NL(K) = NR(K)$, the value of the absolute deviation doesn't change, so it is minimal and constant. For the task, we will take any of the two middle elements for $K$.

There is also one technique that we should know before solving this problem, so-called 'Keeping track of median'. We also need to keep track of $NL$, $NR$, $SR$ and $SL$. That means, if we have an array, by iterating over it once, we should have those variables calculated for every prefix of that array. At some time, if we have $NL$, $NR$, $SR$, $SL$ and $X$ (median), error of estimating given sequence is $NL(X) \cdot X - SL(X) + SR(X) - NR(X) \cdot X$. Now, let's start with the solution of the original problem.

At first glance, it looks like a 2D DP problem, and it is. Also, because $N <= 2000$ we can expect a $N^2$ factor in the overall complexity. (In fact, there is also a $K$ factor in the complexity, but, fortunately, $K$ is small). We will use classic 2D Dynamic programming, where $dp[i][j]$ stands for the minimum error in estimating first i elements by using j contiguous sections.

We will do the following process for every $i$. First, fill in the $mediansum[]$ array, positions from 1 to $i$, where $mediansum[j]$ is the minimum error of estimation for the section $[j..i]$, by using aforementioned technique of keeping track of median and formula for calculating absolute deviation. Now, the DP part starts. Iterate by every number of sections, let's call that $j$. We want to calculate $dp[i][j]$. It's not hard to notice that $dp[i][j] = \min(dp[p][j-1] + mediansum[p+1])\,, \forall p < i$. We will do that by iterating $p$ from 0 to $i - 1$.

Our answer will be $dp[n][k]$, and the time complexity of solution is $O(N^2 \cdot K)$, and memory complexity is $M(N \cdot K)$.

***Added by: Bidhan***
***Solution by:***

*Name:* ***Vladimir Milenković***
*School: Mathematical Grammar School, Belgrade*
*E-mail: vladimirm98@gmail.com*

## Problem R1 03: Substrings II (ID: 8747)

Time Limit: 0.721 second

Memory Limit: 1536 MB

You are given a string T which consists of $40000$ lowercase Latin letters at most. You are also given some integers $A$, $B$ and $Q$. You have to answer $Q$ queries. For $i$-th query you are given a string $S_i$ and you need to output how many times $S_i$ appears in $T$. Immediately after answering the current query you need to add $((A * ans + B)\ modulo\ 26 + 1)$-th lowercase symbol of the English alphabet to the end of $T$ where $ans$ is the answer to this query.

### Input

The first line of input contains a string $T$. The next line consists of three integers $Q$ $(1 \leq Q \leq 40000)$, $A$ $(0 \leq A \leq 27)$ and $B$ $(0 \leq B \leq 26)$. The following $Q$ lines contain $Q$ query strings, $S_{i-2}$ on $i$-th line. Input will not exceed 600 kb.

### Output

Output $Q$ lines. Output the answer to the $i$-th query on the $i$-th line output.

### Sample

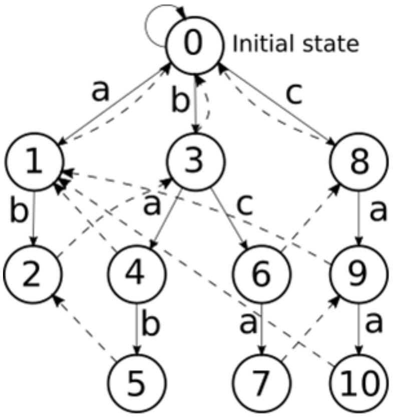
| Input | Output |
|---|---|
| Aaaaa<br>2 0 0<br>a<br>aa | 5<br>5 |

### Solution:

In this problem we need count the number of matches of a set of strings in an input text. This can be done with the Aho Corasick algorithm. The fact that we have to append a character to the input text after each query means that we'll need to modify the algorithm to support counting the matches at any time of the matching process.

**Short description**

The Aho Corasick algorithm is a string searching algorithm used when searching for all occurrences of a finite set of strings in an input text. The main idea is very similar to KMP algorithm, but extended to support multiple keywords. It constructs a finite state automata resembling a trie containing all the keywords.



*Finite state automata, resembling a trie. Dashed lines represent the failure function. This example to the left is built with the following keywords:*

***a b c ab bab bca caa***

The algorithm matches all keywords at once. It processes symbols from the input text, $a_1 a_2 \dots a_n$, one by one. After processing $j$-th symbol the automata should be in a state that represents the longest suffix of $a_1 a_2 \dots a_j$ that is also a prefix of some keyword. When processing the next symbol, $a$, we try to advance the current state by following the edge labeled from the current state (i.e. adding to the longest matched prefix), if that isn't possible we transition to the state representing the longest proper suffix to the current state, and try again. After processing each symbol we output all the matches that end at the current symbol.

The behavior of the automata can be described by three functions: goto function $g$, failure function $f$, and output function $o$. Those functions are computed before the input symbol processing begins.

- goto function $g(s, a)$ returns state advanced to when state $s$ can be advanced with symbol $a$, and $fail$ message otherwise.

- failure function describes the next longest match to use when the current match cannot be advanced. It maps states to states and $f(u) = v$ iff $v$ spells the longest proper suffix to $u$. This function is used for state transitions when goto function emits a $fail$.

- output function returns a set of all keyword that are matched when in state $s$.

**Implementation**

Computing the goto function is done while building a trie of keywords.

Computing failure function for state $u$ which is a son of state $v$ in the trie, with the edge between them being labeled $a$, can be done analog to advancing of automata in state $f(v)$ with symbol $a$. In the process we are not relying on values of failure function of states deeper in trie that v. Because of that we can compute the failure function in BFS order.

Output function can be computed with the following recurrent relation: $output(v) = output\big(f(v)\big) \cup T$, where if state $s$ is associated with a keyword $k$, $T$ is a set containing the keyword $k$, otherwise $T$ is an empty set.

With the three functions precomputed, we can start the symbol processing:

```
s = initialState
for i := 1 to n do
     while g(s,a[i]) = fail do s := f(s)
     s := g(s,a[i])
     if (output(s) != emptySet)
          print(i , output(s))
```

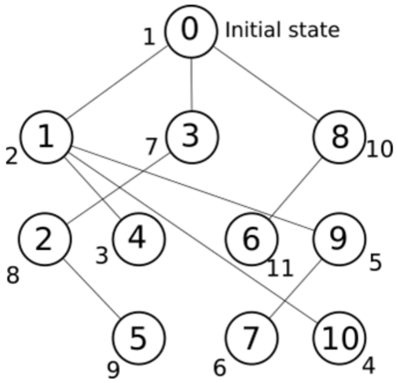
**The modification**

The most obvious way to transform the algorithm described above to support counting the matches is to maintain an array with the number of matches for each keyword, and then after every input symbol go through the result of the output function and update the array. This method is too slow as the number of matches is a factor in complexity. To eliminate that factor we should do away with the output function, and use a more sophisticated way to count the matches.

Notice that the graph constructed of the states of the automata, and failure function edges is a tree. This is because failure function maps $u$ to $v$ only if $u$ is deeper than $v$ in the trie. Let's call that tree fail tree, and the initial state of the automata the root of the fail tree. Also note that for any state $v$, keyword $k$ is in iff state associated with $k$ is on the path from root to $v$ in the fail tree. This means that the number times a keyword $k$ (whose associated state is $v$) was matched is equal to the number of times the current state of the automata was in subtree of $v$ in the fail tree.

To count the number of matches we assign each state $v$ a variable $X[v]$ representing the number of times the algorithm was in a state $v$. When the algorithm processes an input symbol and is in state $v$, we increment $X[v]$. When we're calculating how many times keyword k (whose associated state is $v$) was matched, we return the sum of $X[u]$ for all states $u$ in the subtree of $v$.

This can be implemented efficiently with a data structure that supports modification of a field, and range sum queries in $O(log\ N)$, we'll use Binary Indexed Trees. We traverse the fail tree and assign each state $v$ dfs number, $dfs(v)$. We also compute the largest dfs number in the subtree of $v$, $maxDfs(v)$. When incrementing $X[v]$ we increment the field indexed $dfs(v)$ in BIT. When computing the sum in a subtree of $v$ we return the sum on range $[dfs(v), maxDfs(v)]$ in BIT.



*Fail tree, with a dfs number next to each state*

The complexity of building automata and assigning dfs numbers is $O(M)$, for processing the input symbols it's $O(N+Q)$, and for updating the bit and answering the queries it's $O((N+Q)*logM)$, where M is the sum of the lengths of all keywords.

***Added by: Tooru Ichii***
***Solution by:***
*Name:* ***Dušan Živanović***
*School: Grammar School "Svetozar Marković", Niš*
*E-mail: zdule1998@gmail.com*

## Problem R1 04: Light Cycling (ID: 14981)

Time Limit: 2.0 second

Memory Limit: 1536 MB

Having been sucked into your father's secret computer through a projector in the back of his arcade (or something), you find yourself in the wonderful world of Tron! Here, you play games all day, and if you ever lose, you die.

One such game involves you and an opponent driving around a flat grid on light cycles, which leave behind a permanent trail of...light...wherever they go. This grid can be modeled with the Cartesian plane, and is enclosed by a rectangle of impenetrable walls which ensure that the x-coordinate of each light cycle is always between $1$ and $10^{12}$, while its $y$-coordinate is between $1$ and $10^{6}$ (inclusive). Light cycles always stay on the grid lines, and move at a speed of $1$ square per second.

A match lasts $S$ $(1 \leq S \leq 10^{100})$ seconds. You start at coordinates $(X_A, Y_A)$ and follow a set of $N_A$ $(1 \leq N_A \leq 10^{5})$ instructions, with your ith instruction consisting of moving $L_{A_i}$ squares in the direction given by the character $D_{A_i}$ (with "U", "D", "L", and "R" representing up, down, left, and right, respectively). Similarly, your opponent starts at coordinates $(X_B, Y_B)$ and follows a set of $N_B$ $(1 \leq N_B \leq 10^{5})$ instructions, with their ith instruction described by $L_{B_i}$ and $D_{B_i}$. Of course, neither player's instructions will ever take them beyond the boundaries of the walls, and it will take each player exactly $S$ seconds to execute their instructions. Additionally, for each player, no instruction will have an equal or opposite direction to that of their previous instruction. Finally, if a grid point is ever visited more than once throughout the course of the match, it is guaranteed that one of the path segments intersecting there is passing directly through vertically, while the other is passing directly through horizontally (as such, this cannot happen at either player's starting or ending points).

Whenever both light cycles reach the same grid point at the same time, or a light cycle hits an existing trail of light (in other words, a grid point which either light cycle had previously passed through), a collision occurs. Because you're just playing a practice match for now, neither player dies when this occurs, and, in fact, the collision is not counted in favor of either you or your opponent. Instead, for $T$ $(1 \leq T \leq 20)$ scenarios as described above, you're simply interested in the number of collisions that will occur throughout each match.

### Input

First line: 1 integer, $T$

For each scenario:

First line: 1 integer, $S$

Next line: 3 integers, $X_A$, $Y_A$, and $N_A$

Next $N_A$ lines: 1 character, $D_{A_i}$, and 1 integer, $L_{A_i}$, for $i = 1..N_A$

Next line: 3 integers, $X_B$, $Y_B$, and $N_B$

Next $N_B$ lines: 1 character, $D_{B_i}$, and 1 integer, $L_{B_i}$, for $i = 1..N_B$

### Output

For each scenario:
1 integer: The total number of collisions that will occur.

### Sample

| Input | Output |
|---|---|
| 1<br>12<br>2 5 5<br>R 4<br>U 1<br>L 1<br>D 4<br>L 2<br>3 3 4<br>U 3<br>L 2<br>D 2<br>R 5 | 4 |

### *Solution:*

Even though this problem might seem quite complex, it can easily be transformed into a well-known line segment intersection problem. The first thing we might realize is that each trail is essentially a polygonal chain, where each line segment is parallel to one of the axes. We're interested in counting intersections between those line segments (regardless of the trail they're on), and the constraints given at the end of the problem statement guarantee that there will be no intersections between two vertical or two horizontal lines and that the only single-point intersections are between two line segments that are adjacent on a trail (trail intersections). Once we count all line segment intersections we should subtract the number of trail intersections ($N = N_a + N_b - 2$, because there are $N_a$ segments in the first, and Nb segments in the second trail) since those should not be counted as collisions.

The problem we are facing now can be solved by a line sweep algorithm, an approach based on the concept of a vertical line that is dragged across the plane, stopping to do some computation on a set of valuable points we call events. In this case, we have three events:

- The left endpoint of a horizontal line
- The right endpoint of a horizontal line
- The vertical lines

When the line encounters first two types of events we should add/remove horizontal lines from the **active set** - the set of horizontal lines our sweep line is currently intersecting. Each time it encounters the event of the third type - the vertical line with endpoints $(x, y_1)$ and $(x, y_2)$, where $y_1 < y_2$, we should consider all lines in the active set such that their ordinate is in interval $[y_1, y_2]$. Therefore, we need to be able to insert integers in a set, remove them, and be able to count the number of integers in a range. There are at least two ways to accomplish this: one using an augmented BST (Binary Search Tree), and another one that uses BIT (Binary Indexed Tree, Fenwick Tree). I opted for the latter, and since there is a convenient constraint $y \in [1, 106]$ there are no memory issues. Consider standard BIT functions $Update(idx, val)$ and $GetSum(idx)$. Adding a horizontal line with ordinate $y$ is equivalent to $Update(y, 1)$, removing it would be $Update(y, -1)$, and a third type event can be resolved by a simple range sum query: $GetSum(y_2) - GetSum(y_1 - 1)$.

To sum it up: we parse the input and represent paths as line segment arrays, create events and sort them, and finally perform a line sweep keeping track of the total number of intersections we found. In the very end, we subtract $N$, the number of trail intersections and print the result. The overall complexity of this solution is $O(N\log(N))$. There are still two more things to be careful about:

- the final result and the $x$-coordinates can be too large for 32-bit integer type, so we have to use 64-bit integers (long long in C++)
- there are $T \leq 20$ scenarios in a single test case, which means we have to reset all used variables and structures (e.g. BIT) after each scenario is processed

Of course, you probably noticed that we never used the input variable $S$, the total match length. Since we're dealing with line segments as endpoint pairs, and their length does not really matter, parsing this fairly large $(S \leq 10^{100})$ integer is unnecessary. Let's just suppose its sole purpose is to troll.

***Added by: Jacob Plachta***
***Resource:*** *Own problem*
***Solution by:***

*Name:* ***Nikola Jovanović***
*School: Mathematical Grammar School, Belgrade*
*E-mail: nikolajovanovic96@yahoo.com*

## Problem R1 05: Crossbits (ID: 3484)

Time Limit: 0.133 second

Memory Limit: 1536 MB

Crossbits are like Crosswords; instead of entering words you enter binary bits 01 in a Crossbit under certain given conditions, assuming that a solution exists. An empty Crossbit of size $N$ is an empty grid of size $N \times N$. Given a natural number $N$, consider entering $N^2$ binary bits in an empty Crossbit, satisfying the following conditions:

Each square in the grid contains either a 0-bit or a 1-bit with no 1-bit in two major diagonals.

The total number of 1-bit in each row / column is exactly equal to $K$, $K$ being a given natural number less than $N$.

A 0-bit has at least another adjacent 0-bit either in the same row or in the same column.

The Crossbit represents the $N^2$ -bit binary number B formed by placing bits in the 1st, the 2nd , ... the $N$th row from left to right.

You are required to write a program that enters bits in an empty Crossbit so that the Crossbit represents the least binary number $B$ for given $N$ and $K$.

As an illustration consider the case with $N = 4$ and $= 1$. The Crossbit shown below represents the least binary number $B = 0010100000010100$ of 16 bits satisfying the specified conditions.

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |

### Input

The input may contain multiple test cases. For each test case parameters $N$ and $K$ of the Crossbit are given in one line. Assume that $N$ does not exceed 10.

The input terminates with a line containing 0 as input.

### Output

For each test case, print the Crossbit in $N$ rows; each row contains $N$ bits with a space between two neighbouring bits. Keep a blank line after the last output line of each test case.

### Sample

| Input | Output |
|---|---|
| 4 1<br>6 2<br>0 | 0 0 1 0<br>1 0 0 0<br>0 0 0 1<br>0 1 0 0<br><br>0 0 0 1 1 0<br>1 0 0 1 0 0<br>0 0 0 0 1 1<br>1 1 0 0 0 0<br>0 0 1 0 0 1<br>0 1 1 0 0 0 |

***Solution:***

Since $N$ and $K$ are small, we can precalculate answers for every possible test on local computer and then hardcode answers to source code.

As generating all the possible matrices is extremely slow, we will generate only correct Crossbits. Also, we will generate only one Crossbit and it'll be guaranteed that it'll represent minimal number.

Let's generate matrix row by row from right to left, trying to place to each cell one and if generation fails, placing zero. This order of generation guarantees that ones in first row will be placed at the most right (among all the possible Crossbits), then the same is rule is applied for second row and so on. So, the number represented by generated Crossbit will be as small as possible.

To satisfy the second condition of Crossbit, we need to store two arrays: $row[x]$ and $col[x]$, where $row[x]$ is number of ones in $x$-th row and $col[x]$ is number of ones in $x$-th column. Then it's possible to implement the following cutoff: we'll try to place 1 to position $(i, j)$ if and only if $(row[i] < k \ and \ col[j] \ < \ k)$. Additionaly, if the row is completed and has less than $k$ ones, then we also will cutoff from this branch of bruteforce.

When placing 0 or 1, we also should check that the third rule is satisfied. It's obvious that we should look only at current cell and its neighbours, because all the other cells are unaffected. So, for each cell, in current cell and adjacent to it, we should check if at least one neighbour of given cell is set to zero. Note that uninitialized cells are also considered as zeroes. So, implementing bruteforce for every possible testcase using these two cutoff produces answers in a few seconds.

***Added by: Duc***
***Resource:*** *ACM Kanpur 2006*
***Solution by:***
*Name**:** **Petr Smirnov***
*School: St. Petersburg Academic University*
*E-mail: ComradePetr@gmail.com*

## Problem R1 06: Font Size (ID: 18530)

Time Limit: 0.133 second

Memory Limit: 1536 MB

John bought a billboard. He knows what text he wants to put on it, and he has chosen a monospace font to use, but he doesn't know what the font size should be.

Naturally, John wants to use the largest font size possible.

Sizing works as follows:

- The height of a line is the font size.
- The width of a character is two-thirds of the font size.
- Lines can only break between the space-separated words.
- If a space ever falls at the beginning or end of a line, it is omitted.

Don't worry about details like kerning and line spacing.

### Input

The first line is the width of the billboard in inches, $0 \leq W \leq 7500$, and the height of the billboard in inches, $0 \leq H \leq 7500$.

The third line is the non-empty message, which is comprised of alphanumeric words separated by single spaces. The message is most $10{,}000$ characters long.

### Output

Print the maximum possible font size, in inches. Your answer must be accurate to within 0.001 of the actual value.

### Samples

| Input | Output |
|---|---|
| 60 25<br>The lazy brown dog | 10 |

| Input | Output |
|---|---|
| 50 50<br>The lazy brown dog | 12.5 |

***Solution:***

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

Problem solution (maximum possible font size) can be binary searched, because if it can be written in given font size, it can certainly be done for smaller font. Let the current font size be k. We can determine if we can write text with this font size by following greedy algorithm: we go through all the words from the text and we add the word in the current line as long as it is possible. If when adding the words we move the width of the billboard, then we must increase the number of lines. By doing that, we should pay attention to the width of the space between words. If, after we have processed all the words, the number of the rows time font size is fewer than the width of the billboard, then that font satisfies the conditions, otherwise it doesn't.

***Added by: BYU Admin***
***Solution by:***
*Name****: Stjepan Požgaj***
*School: Grammar School "Josipa Slavenskog", Čakovec*
*E-mail: stjepan.pozgaj1@gmail.com*

## Problem R1 07: Michel and the championship (ID: 10730)

Time Limit: 1.542 second

Memory Limit: 1536 MB

Michel is participating in a championship where each participant has $p_i$ $(0 \le i \le N-1)$ points. He knows for some pairs of participants an inequality between the points of each one, in the form $p_A - p_B \ge C$. Now he wants to know if his data is correct, i.e. if it's possible to assign points for each participant and satisfy all inequalities.

### Input

The input consists of several test cases (at most 150). The first line of each test case consists of two integers $N$ and $M$ $(1 \le N \le 500, 0 \le M \le 5000)$. Then follow $M$ lines of three integers $A$, $B$ and $C$, indicating that $p_A - p_B \ge C$ $(0 \le A, B \le N-1, |C| \le 20000)$.

### Output

Print a single line for each test case with 'y' if the data is valid or 'n' if it's not.

### Samples

| Input | Output |
|---|---|
| 2 2<br>0 1 2<br>1 0 2<br>4 4<br>0 1 1<br>1 2 1<br>2 3 -2<br>3 0 1<br>4 4<br>0 1 1<br>1 2 1<br>2 3 -3<br>3 0 1 | n<br>n<br>y |

***Solution:***

We have some constraints $p_{ai} - p_{bi} \geq c_i$ in input data. This is equivalent to constraints $p_{bi} - p_{ai} \leq -c_i$ (we can just multiply both parts by $-1$). These inequalities remind of distances in directed weighted graph with start vertex $s$: if $d_v$ is shortest distance from vertex $s$ to $v$, then for each edge from $v$ to $u$ with cost $c$ condition $d_u - d_v \leq c$ is satisfied.

So, let's create a graph, each vertex of it corresponds to participant of tournament, and edge from vertex $v$ to vertex $u$ with cost $c$ means, that we have constraint $p_u - p_v \leq c$. Also we have fictive vertex $s$ and edges from $s$ to each vertex of original graph with cost 0.
If we find shortest distances from $s$ to each vertex in such graph, then $p_i = d_i$ will be answer.
It means, that if input data is incorrect, then we can't find shortest distances to each vertex, so in graph there is negative cycle. In other case data is correct. For determining if there is negative cycle in constructed graph (we don't even need to find it) we can use Ford-Bellman algorithm with complexity $O(N \cdot M)$.

Final complexity of solution is $O(T \cdot N \cdot M)$.

***Added by: Marcos Kawakami***
***Resource:*** *Guilherme Souza (guirns)*
***Solution by:***
*Name:* ***Nikita Podguzov***
*School: Saint Petersburg Academic University*
*E-mail: npodguzov@yandex.ru*

## Problem R1 08: The Indian Ocean (ID: 16128)

Time Limit: 1-60 second

Memory Limit: 1536 MB

Salim is a part of THE INDIAN OCEAN BAND. Each musician of the band has already decided what sound will he play (for the sake of simplicity we assume each musician plays only one sound). We say two sounds are in harmony if the frequency of any one of them divides the frequency of the other (that's a pretty restrictive idea of harmony, but THE INDIAN OCEANS are known to be very conservative in music). Salim knows that the notes played by other players are not necessarily in harmony with each other. He wants his own note to improve the music, so he wants to choose his note so that it is in harmony with the notes all the other musicians play.

Now, this sounds simple (as all the frequencies are positive integers, it would be enough for Salim to play the note with frequency 1, or, from the other side, the Least Common Multiple of all the other notes), but unfortunately Salim's instrument has only a limited range of notes available. Help Salim find out if playing a note harmonious with all others is possible.

### Input

The first line of the input gives the number of test cases, $T$. $T$ test cases follow. Each test case is described by two lines. The first contains three numbers: $N$, $L$ and $H$, denoting the number of other players, the lowest and the highest note Salim's instrument can play respectively. The second line contains $N$ integers denoting the frequencies of notes played by the other players.

$1 \leq T \leq 40.$

$1 \leq N \leq 10^4.$

$1 \leq L \leq H \leq 10^{16}$

All the frequencies are no larger than $10^{16}$.

### Output

For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is either the string "NO" (if Salim cannot play an appropriate note), or a possible frequency. If there are multiple frequencies Salim could play, output the lowest one.

### Samples

| Input | Output |
|---|---|
| 3<br>3 2 100<br>3 5 7<br>4 8 16<br>1 20 5 2<br>2 5 10000000000000000<br>9999999999999606 9999999999999822 | Case #1: NO<br>Case #2: 10<br>Case #3: 6 |

### *Solution:*

Given array of frequencies $A$ and range of new frequency $[lo, hi]$ which must satisfy condition that for all numbers in the given array, $A_i$ is its multiplier or divisor.

Assume that $X$ is required value of new frequency. That means that $X$ is the multiplier of all the numbers $A_i$ which are less than $X$ and divisor of all numbers $A_i$ which are greater than $X$. In other words, $X$ is the divisor of *GCD* of all the numbers that are greater than $X$ and the multiplier of *LCM* of all the numbers that are less than $X$.

Let's sort array $A$ and observe *LCM*-prefixes and *GCD*-suffixes:

- $A_1 \leq A_2 \leq A_3 \leq \cdots \leq A_N$
- $G_i = \gcd(A_i, A_{i+1}, A_{i+2}, \dots, A_N)$
- $L_i = lcm(A_1, A_2, A_3, \dots, A_i)$ and $L_0 = 1$

For some fixed position $i$ $(1 \leq i \leq N)$, frequency $X$ is considered good if:

$L_{i-1} \mid X$ and $X \mid G_i$ which means that $L_{i-1} | \; G_i$ holds, then $X$ can be represented as: $X = k \cdot L_{i-1}$ where k is divisor of $\frac{G_i}{L_{i-1}}$

For each $i$ that satisfies $L_{i-1} | \; G_i$, we need to check if there exists a frequency in the given range $[lo, hi]$.

Let's define function $check(x, y)$ which will do just that: $x$ is equal to $(Gi/L_{i-1})$, and $y$ is equal to $L_{i-1}$. Function will pass through all factors $k$ of number $x$, and check whether frequency $L_{i-1} * k$ is within given range. One way to find all factors of a number $x$, which we will use, is to factorize $x$ first. The complexity of whole procedure *check* is described below.

There is one case left when frequency $X$ is the multiplier of all the numbers in the given array.

***The complexity determination***

Sorting and calculating LCM-prefixes and GCD-suffixes can be done in $O(N \; log \; N)$ where $N$ is the size of array $A$. The hard part is to determine complexity for calls of function *check*.

Number of factors of the number $n$ is not greater than $2\sqrt{n}$ (in fact, the greatest number of factors that given frequency may have is less than $10^5$). Factorization of the number $n$ can be done in complexity $O(\sqrt{n})$ which means that the complexity of single call of function $check(x, y)$ is $O(\sqrt{x})$. Let's determine the overall complexity for all the calls of $check$.

Notice that $G_i \leq A_i$, and $L_{i-1} \geq A_{i-1}$, which means that inequality $\frac{G_i}{L_{i-1}} \leq \frac{A_i}{A_{i-1}}$ holds. Let $x_1, x_2, \dots x_k$ be every $x$ that is called in function $check$. Then the following relations holds:

$$x_1 \cdot x_2 \cdot \ldots \cdot x_k = \prod_i \frac{G_i}{L_{i-1}} \leq \prod_i \frac{A_i}{A_{i-1}}$$ for each *i* for which we will call *check*

Since $A_i/A_{i-1} \geq 1$, the above expressions all less than or equal to:

$$\frac{A_2}{A_1} \cdot \frac{A_3}{A_2} \cdot \ldots \cdot \frac{A_N}{A_{N-1}} = \frac{A_N}{A_1} \leq 10^{16}$$

This means that the product of all *x* in the calls of *check(x, y)* is less than or equal to the greatest number $A_i$ (which is less than $10^{16}$). Let $y_i = \log_2 x_i$. Then above inequalities can be written as:

$y_1 + y_2 + \cdots + y_k \leq \log_2 A$ (where $A$ is the greatest value of $A_i$)

Since the complexity for single call of check for some parameter $x_i$ is $O(\sqrt{x_i})$ which is $O(2^{\frac{y_i}{2}})$, the overall complexity for all the calls of check is: $O(2^{\frac{y_1}{2}} + 2^{\frac{y_2}{2}} + \ldots + 2^{\frac{y_k}{2}})$.

Since $2^a + 2^b \leq 2^{a+b} + 1$, then the following relations holds:

$$2^{\frac{y_1}{2}} + \cdots + 2^{\frac{y_k}{2}} \leq 2^{\frac{y_1}{2} + \cdots + \frac{y_k}{2}} + k - 1 = 2^{\frac{y_1 + \cdots + y_k}{2}} + k - 1 = (x_1 \cdot \ldots \cdot x_k)^{\frac{1}{2}} + k - 1$$

Since $(x_1 \cdot \ldots \cdot x_k)^{\frac{1}{2}} \leq \left(\frac{A_N}{A_1}\right)^{\frac{1}{2}}$, the overall complexity for all the calls of function *check* is $O(\sqrt{A})$. Precomputing all the primes below $10^8$ would lead to solution with overall complexity of $O(\sqrt{A}\ /\ log\ A)$ for all the calls of function *check*. Since $k - 1$ is less than or equal to $N$ and doesn't affect complexity, the overall complexity is $O(N\ log\ N\ +\ \sqrt{A}\ /\ log\ A)$ per test case.

***Added by: Gaurav Jain***

***Solution by:***

*Name:* ***Tonko Sabolčec***

*School: XV gimnazija, Zagreb*

*E-mail: tonkosi123@gmail.com*

## Problem R1 09: Breaking Chocolates (ID: 6499)

Time Limit: 0.623 second

Memory Limit: 1536 MB

Bored of setting problems for Bytecode, Venkatesh and Akhil decided to take some time off and started to play a game. The game is played on an R*C bar of chocolate consisting of Black and White chocolate cells. Both of them do not like black chocolate, so if the bar consists only of black chocolate cells, it is discarded (Discarding the bar is not considered as a move). If the bar consists only of white chocolate cells, they do not break it further and the bar can be consumed at any time (Eating the bar is considered as a move). If the bar consists of both black and white chocolate cells, it must be broken down into two smaller pieces by breaking the bar along any horizontal or vertical line (Breaking the bar is considered as a move). The player who cannot make a move on any of the remaining bars loses.

Assuming Venkatesh starts the game and both players are infinitely intelligent, determine who wins the game.

### Input

The first line of input contains a number $T$ $(T \leq 25)$, the number of test cases.

For each testcase, first line contains two space separated integers $R$ and $C$ $(1 \leq R, C \leq 30)$. The following $R$ lines contain $C$ space separated integers which are either 0 (White) or 1 (Black).

### Output

For each testcase output “Venkatesh wins” or “Akhil wins” (quotes for clarity).

### Samples

| Input | Output |
|---|---|
| 4<br>3 3<br>0 0 0<br>0 0 0<br>0 0 0<br>3 3<br>1 1 1<br>1 1 1<br>1 1 1<br>1 2<br>1 0<br>3 3<br>1 0 1<br>0 1 0<br>0 0 1 | Venkatesh wins<br>Akhil wins<br>Akhil wins<br>Venkatesh wins |

### *Solution:*

The solution is straightforward once we understand what *Grundy numbers* (also called *nimbers*) are.

Nim is a mathematical game of strategy in which two players take turns removing pebbles from distinct heaps. On each turn, a player must remove at least one pebble, and may remove any number of pebbles provided they all come from the same heap.

The Sprague–Grundy theorem states that every impartial game is equivalent to a nim heap of a certain size. An impartial game is a game in which the allowable moves depend only on the position and not on which of the two players is currently moving, and where the payoffs are symmetric. The Grundy number of a game of Nim with one heap of $N$ pebbles is $N$.

When two or more impartial and independent games are played in parallel, meaning that each player can choose one of the subgames and make a move in it, the Grundy number of the whole game is equal to the bitwise XOR of the Grundy numbers of the subgames. For example, Nim with $N$ heaps can be understood as $N$ parallel Nim games with one heap. The Grundy number of a game of Nim with $N$ heaps with $a_1, \ldots a_n$ pebbles each is

$$a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

Where $\oplus$ is the bitwise XOR.

Another way to calculate the Grundy number of a game is to consider the set of all Grundy numbers of games resulting from all possible moves from the starting game. Let that set be $A$. Let $Mex(S)$ be the smallest non-negative integer not in $S$. Then, the Grundy Number is equal to $Mex(A)$. Now, it is easy to see why the Grundy number of a game of Nim with one heap of $N$ pebbles is $N$.

A game whose Grundy number is equal to $G$ is won by the first player iff $G > 0$, and by the second player otherwise.

Our game is clearly impartial! We need to use both approaches for calculating Grundy numbers. We'll calculate Grundy numbers for all submatrices of the given matrix. There are three cases to consider:

If a (sub)matrix consists only of zeros, we discard it (this is not considered a move), so it is equivalent to a Nim heap of size 0.

If a matrix consists only of ones, a player cannot break it but can consume it in one move, so it is equivalent to a Nim heap of size 1.

Otherwise, break the matrix into two in all possible ways. This always creates two independent games whose Grundy numbers we can compute recursively. The Grundy number of a game with two matrices whose Grundy numbers are $A$ and $B$ is simply $A \oplus B$. The Grundy number of our original game is equal to

$$Mex(\{A_1 \oplus B_1, A_2 \oplus B_2, \ldots, A_k \oplus B_k\})$$

Where $A_i, B_i$ are the Grundy numbers of two matrices resulting from breaking the original one, and $k$ is the number of ways to do it.

*Remark:* $N = max(R, C)$

We can calculate Grundy numbers for all submatrices of the original matrix (there are $O(N^4)$ of them) in $O(N)$ each - there are $O(N)$ ways to split a matrix. Checking if a matrix contains only zeros or ones can be done in $O(1)$ with $O(N^2)$ preprocessing using 2D prefix sums.

Our solution uses the bottom-up approach to avoid the overhead caused by recursion. The Grundy numbers for the submatrices are stored in a 4-dimensional matrix.

If the Grundy number for the whole matrix is 0, the second player (Akhil) wins, otherwise Venkatesh wins.

Overall, our solution runs in $O(N^5)$ with a very small constant factor, enough to easily pass all test cases.

***Added by: suhash***
***Resource:*** *ByteCode 2010*
***Solution by:***
*Name:* ***Ivan Stošić***
*School: Faculty of Sciences and Mathematics, University of Niš*
*E-mail: ivan100sic@gmail.com*

## Problem R1 10: [CH] HQNP Incomputable Hard (ID: 10295)

Time Limit: 0.2-2s second

Memory Limit: 1536 MB

HQ9+ is an esoteric programming language specialized for certain tasks. For example, printing “Hello, world!” or writing a quine (a program that prints itself) couldn’t be any simpler. Unfortunately, HQ9+ doesn’t do very well in most other situations. This is why we have created our own variant of the language, HQ0-9+−INCOMPUTABLE?!. A HQ0-9+−INCOMPUTABLE?! program is a sequence of commands, written on one line without any whitespace (except for the trailing newline). The program can store data in two memory areas: the buffer, a string of characters, and the accumulator, an integer variable. Initially, the buffer is empty and the accumulator is set to 0. The value of the buffer after executing all the commands becomes the program’s output.

HQ0-9+−INCOMPUTABLE?! supports the following commands:

| command | Description |
|---:|---|
| h, H | appends helloworld to the buffer |
| q, Q | appends the program source code to the buffer (not including the trailing newline) |
| 0-9 | replaces the buffer with *n* copies of its old value – for example, ‘2’ doubles the buffer |
| + | increments the accumulator |
| - | decrements the accumulator |
| i, I | increments the ASCII value of every character in the buffer |
| n, N | applies ROT13 to the letters and numbers in the buffer (for letters ROT13 preserves case; for digits we define ROT13(*d*) = (*d* + 13) mod 10) |
| c, C | swaps the case of every letter in the buffer; doesn’t change other characters |
| o, O | removes all characters from the buffer such that their index, counted from the end, is a prime or a power of two (or both); the last character has index 1 (which is a power of 2) |
| m, M | sets the accumulator to the current buffer length |
| p, P | removes all characters from the buffer such that their index is a prime or a power of two (or both); the first character has index 1 (which is a power of 2) |
| u, U | converts the buffer to uppercase |
| t, T | sorts the characters in the buffer by their ASCII values |
| a, A | replaces every character in the buffer with its ASCII value in decimal (1–3 digits) |
| b, B | replaces every character in the buffer with its ASCII value in binary (exactly eight ‘0’/‘1’ characters) |
| l, L | converts the buffer to lowercase |
| e, E | translates every character in the buffer to l33t using the following table:<br>`ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 0123456789`<br>`48(03=6#\|JXLM~09Q257UVW%Y2 a6<d3f9hijk1m^0p9r57uvw*y2 O!ZEA$G/B9` |
| ? | removes 47 characters from the end of the buffer (or everything if it is too short) |
| ! | removes 47 characters from the beginning of the buffer (or everything if it is too short) |

In this task you have to print the HQ0-9+−INCOMPUTABLE?! program that will give the number n as output.

**Score**: Score is number of test cases correctly solved.

### Limits

Any HQ0-9+−INCOMPUTABLE?! Program (output for each case) must be at most 20,000 commands long. The accumulator is unbounded (it can store an arbitrarily large integer). After each command, the buffer must be at most 20,000 characters long. To prevent code injection vulnerabilities, during the execution of your program (output for each case) the buffer must never contain non-alphanumeric characters, i.e. characters other than A-Z, a-z, and 0-9. If it happens, your score for that particular test case will be 0.

Additionally for this challenge you must use the command '+' at least once for each test case.

### Input

First line has integer $T$ i.e number of test cases. ($T \leq 1000$). Next $T$ lines has a number $n$ ($0 \leq n \leq 10^{1000}$).

### Output

For each $n$, output the required HQ0-9+−INCOMPUTABLE?! Program (having '+' at least once) which will give n as output. If there are multiple solutions, output any one of them. Output of each test case must be in a single line. If you don't want to solve a test case, leave a blank line for it.
PS : No of test cases in 11 test files are respectively -> 10, 10, 100, 1000, 80, 20, 40, 500, 100, 50, 200

### *Solution:*

After a certain time of analyzing this very interesting problem, like most of the contestants, we came to the conclusion that given the additional constraints, we have to find the set of characters which through the transformation with the command "e" result in an array of numbers.

After picking the adequate set of characters, the job is split into two phases:

1. Setting up either one or more characters
2. Pushing the buffer a specific number of places, which is the shortest distance between the inserted character and the next one

Before the final transformation of the characters into digits, one more rotating of the buffer is required. Due to this, during the entire algorithm we follow the state of the buffer, i.e. it's distance from some starting position.

The first phase can most easily be performed by adding the letter "h" to the end, by a set of 40 commands: "hhhhh?hhhhh?hhhhh?hhhh?hhhhh?hhhhh?hhhh?"

The second phase depends from which to which digit we are switching and it can be done with a set of commands "i" and "n". If the letter "z" is not currently in the buffer, the command "i" (increment) does the job, but if it is, we have to do a "nin" to convert the z to an a. This forcing practically brings the larger number of mapping the adequate set of characters for the final operation "e" to only one possibility which is:

| 'D' | 'l' | 'z', 'Z' | 'e', 'E' | 'A' | 's', 'S' | 'b' | 't', 'T' | 'B' | 'P' |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Also, in the case of having a need for the command "c", the state of the upper and lower case characters in the buffer is changed.

Example 1:
input 438
output (159 commands):
**hhhhh?hhhhh?hhhhh?hhhh?hhhhh?hhhhh?hhhh?**iiiiiiniii**hhhhh?hhhhh?hhhhh?hhhh?hhhhh?hhhhh?hhhh?**iiiiiiiiiiniiiiii**hhhhh?hhhhh?hhhhh?hhhh?hhhhh?hhhhh?hhhh?**iiiiiiince+

Bold commands ('h ' and '?') represent phase 1, normal ('I' and 'n') phase 2, and at the end commanc 'c' is to go from lowercase to uppercase, 'e' is to transform letters into digits, an '+' is required by task statement.

The second phase in general takes around ten language commands, which coupled with the simplest implementation of the first phase takes around 50, which means that it's simple to solve problems with the length of around 400 which solves around 1310-1340 cases depending on the implementation.

For an improved score it's necessary to deal with the problem of shortening the length of the command needed to add a letter to the output (phase 1) or to minimize the shifting to the next letter (which represents the next given digit i.e. phase 2)

Optimization 1. - Other than the letter "h" which can be simply put at the end of the buffer, it is possible to add the same one which appeared as the first one in the buffer. That is the case when the next digit in line is the same as the first letter in the given input. This scenario happens in around 10% of the cases.

Optimization 2. - Other than the letter "h" or the first letter in the buffer, we can also add the combination "he". This significantly lowers the length of the output from 100 which is needed for two letter to around thirty. However this combination happens in about 4% of cases.

Essentially adding "h", "he", or the first letter in the buffer to the end of the buffer are the three scenarios that bring additional points. How much, well that depends on the set of commands for adding some of these combinations to the end.

Optimization 3. - If we implement some recursive method for searching the shortest possible combination into the program, it certainly won't always be the best which exist in the given moment but will certainly beat "hhhhh?hhhhh?hhhhh?hhhh?hhhhh?hhhhh?hhhh?". For finding a set of commands shorter than 40, we used the commands "h", "?", "2" - "9". The time constraint of the task doesn't really give us too much room for minimizing the command from phase 1.

Optimization 4. - For phase 2, it is necessary to keep watch on the current rotated state of table 1. , that can be done by rotating the array which takes a lot of time. We did this part with bitmap operations which halved the total time.

Optimization 5. - In the set of commands which determine phase 2, if there are two commands "n" one next to the other, both can removed since "nn" don't change the state of the buffer. It doesn't bring much, but may give a small edge against the other solvers.

Optimization 6. - For phase 2, it's better to precalculate the switching from either digit to another, which once again reduces time of the program execution, since we no longer have to pay attention to the rotated states, but can directly paste the optimal set of commands needed to accomplish phase 2. We started using this optimization only after the appearance of all the digits from 0 to 9.

Optimization 7. - In the case that the digits 2, 3, 5 or 7 are given, it is not necessary to change the CASE state of the letters. Doesn't contribute much either, but can make a difference.

Optimization 8. - If you have done the opt. 1 and opt. 2, then we can pick the shortest command between the improved adding of "h" and the other one.

Optimization 9. - If some of the digits which are a prefix to the given number repeat, it can further shorten the number of output commands if done properly.

Optimization 10. - Before starting the complete algorithm we rotate the input for 0-9 places, e.g. 123 input is rotated to 234, 345, 456, .., 901, 012 and after the algorithm does it's work on all 10 possibilities, we just rotate them by the same amount into their original state. We earned 9 points with this optimization.

Example 2: input 438
output (38 commands): **hhpohh39????????????**niiiiininiiniince+

What most likely contributed to our result for this challenge problem in the Bubble Cup qualifications was the decision to do all the commands for phase one offline. Ready masks were added into the program code as string constants. The 50000 bytes constraint gave us some problems but after concluding that with this ideas we cannot solve inputs longer than 750 digits, we removed some masks which were a character or two shorted than those program generated, we managed to fit into this constraint. Also, adding the three character combination "heo" could also have been done, however it proved to have no effect on the score.

***Added by: XeRoN!X***
***Resource:*** *Variation of IPSC Problem*
***Solution by:***
*Name:* ***Jovan Gagrčin***
*School: Gimnazija "Veljko Petrović", Sombor*
*E-mail: akatsuki555@hotmail.com*

## Problem R2 02: Modern Pocket Calculator (ID: 8542)

Time Limit: 0.623 second

Memory Limit: 1536 MB

Adam likes pocket calculators, not only the early ones, but also the modern ones with a two line LCD-display and mathematically correct operator precedence. The upper (input-)line shows the expression you typed in, the (output-)line below shows the result immediately after the [=] key has been pressed. Given the calculator's input-line, the program's task is to produce its output-line. Using the calculator's [S-D] key, the display switches between fractional and decimal representation of the result, so output must contain both representations.

### Input

Input starts with a positive integer $t$ $(t < 1000)$ in a single line, then $t$ testcases follow. Every testcase consists of either three lines or a single line, representing the expression in the calculator's input-line, followed by a blank line. An expression contains $n$ numbers $(0 < n < 10)$ with exactly one operator $(+, -, x, :)$ between any two numbers and exactly one space to separate number and operator. There will be no invalid expression and no undefined operation.

A number is given as a decimal, a fraction or a mixed number and will be non-negative. If a number is positive, its decimal value is between $10^{-9}$ and $10^{9}$, numerator and denominator are also non-negative and not larger than $10^{9}$ each, given decimals have at most 9 digits overall. These constraints also hold for all calculations, if done properly. The length of the fraction bar depends on the maximum length of the numerator or the denominator respectively. If lengths of numerator and denominator are different, the shorter one will be centered based on the fraction bar. If centering isn't possible, it is set one digit to the right.

### Output

For each testcase print the result the calculator will display in its output-line: first fractional, then decimal, both representations separated by " [S-D] ". If a result is negative, the negative sign has to be printed directly (i.e. no space) in front of the integer part or the fraction bar.
The fractional representation has to be printed in lowest terms either as a proper fraction or mixed number. If the number has an integer representation, that has to be printed instead. Numerator and denominator have to be placed as described in the input section.
After " [S-D] " the exact(!) decimal represenation has to be printed. As the screenshot shows, the calculator is even able to display a repeating decimal in its decimal representation using a vinculum, so that's what the program has to do as well, using underscores in the line above. You can assume that no decimal expansion is longer than 100 digits (it's a calculator with XXL-display).
The number of lines for every output depends on the result. It may be three lines (if fractions appear) or a single line (if result is an integer). Print a blank line after every testcase except the last one. Be careful not to print any trailing spaces.

### Samples

| Input | Output |
|---|---|
| 5<br>  9          50<br>2-- + 0.26 x --<br> 11          15<br><br>       4   9<br>5.88 - -- : -<br>      18   5<br><br>3 - 5.125<br><br>9 + 14<br><br> 1<br>--- x 0.5<br>100 |  113<br>3--- [S-D] $3.6\overline{84}$<br> 165<br><br> 1532<br>5---- [S-D] $5.75\overline{654320987}$<br> 2025<br><br>  1<br>-2- [S-D] -2.125<br>  8<br><br>23 [S-D] 23<br><br> 1<br>--- [S-D] 0.005<br>200 |

***Solution:***

This problem wasn't very difficult, but maybe implementation was a bit tricky. We will split the problem in two parts. Firstly, we have to calculate result of the expression as fraction, and after that, use it to find out the decimal form. Considering that the expression contains both forms, we will represent all numbers in fractional form. Operations between fractions are simple to implement, only we should care that multiplication and division have precedence over addition and subtraction. In order to do that, we should first perform all * and / operations between numbers, and after that add and subtract the results. To keep numerator and denominator coprime, we can use Euclidean algorithm to find GCD of them.

Fraction could contain infinite digits in decimal form, but in that case some digits will repeat. We perform simple division, and save the results. If 0 occurs as a reminder, number of digits is finite, and we stop the algorithm there. Otherwise one of the reminders will repeat, because their number cannot be infinite. Periodic digits are the ones between the first and the second occurrence of the same reminder.

***Added by: numerix***
***Resource:*** *own problem*
***Solution by:***

*Name:* ***Ivan Dejković***
*School: Faculty of Computing, Belgrade*
*E-mail: ivandejkovic95@gmail.com*

## Problem R2 03: Boa viagem, Roim (ID: 11718)

Time Limit: 0.623 second

Memory Limit: 1536 MB

Computer Engineering student Roim is getting ready for a trip to Mexico. For that, he has studied the airplane network, so he knows the details of all $R$ regular flights currently in operation on all of the $N$ available airports. Unfortunately, one of his school mates is very annoying and keeps saying the same stuff to him all the time.

To solve that issue, he will organize two different flight plans: one for the team and one for the annoying guy. The condition is that the flight plans may not contain the same flight (note that it is possible for both to pass through the same airport, and that the same flight may not be used by both even if the times are different). As this may not be possible using only regular flights, he has also considered using some of the $C$ flights chartered by travel agencies, but he'd like to keep those to a minimum as they usually suffer from large delays. Of course, as long as the least number of chartered flights is picked, Roim will pick the plans with the least total cost (defined as the sum of the costs of all flights used).

### Input

The input consists of several test cases. On the first line of a test case are three integers $N$ $(2 \leq N \leq 225)$, $R$ and $C$ $(0 \leq R + C \leq N(N-1)/2)$ separated by spaces. The starting airport is $0$, and the destination is $N - 1$.

The next $R$ lines contain integers $a, b$ $(0 \leq a, b \leq N - 1), c$ $(1 \leq c \leq 100)$, meaning that there exists a one-way regular flight between airports $a$ and $b$, with cost $c$. The following $C$ lines give details for chartered flights in the same manner. There is a blank line at the end of each test case. The last test case is followed by a line containing three zeros.

You may assume that any pair of cities is only connected in at most a single direction by a single flight.

### Output

If it is possible to make the plans, print two integers separated by spaces. The first should be the minimum amount of chartered flights used, and the second is the total cost of the solution.
If it's impossible that both the team and the guy get to their destination, print "Boa viagem, Roim" instead.

### Samples

| Input | Output |
|---|---|
| 4 5 0<br>0 1 1<br>1 3 5<br>0 2 5<br>1 2 1<br>2 3 1<br><br>4 4 1<br>0 1 2<br>1 3 2<br>0 2 2<br>1 2 1<br>2 3 2<br><br>2 1 0<br>0 1 10<br><br>0 0 0 | 0 12<br>1 8<br>Boa viagem, Roim |

***Solution:***

Prerequisites: Dijkstra's algorithm

This was the easiest task of round 2 according to the number of accepted solutions.

This problem is about graph theory. Let's denote airports as nodes, flights as edges and flight's cost as edge's weight. First, let's solve the task without using $C$ flights chartered by travel agencies. We'll use only $R$ regular flights and later combine solution using $R$ regular with $C$ chartered flights.

Let's express our current task (without $C$ flights) in terms of graph theory. We are given $N$ nodes and $R$ edges and we have to find two disjoint paths (they may share nodes, but not edges) in a nonnegatively-weighted directed graph, so that both paths connect the same pair of nodes and have minimum total length. This problem is well known in the world of graph theory and there exists an algorithm called Suurballe's algorithm which solves exactly the same problem which we've stated.

Summary of Suurballe's algorithm (source: https://en.wikipedia.org/wiki/Suurballe%27s_algorithm):

1. *Find the shortest path tree T rooted at node s by running Dijkstra's algorithm. This tree contains for every vertex u, a shortest path from s to u. Let $P_1$ be the shortest cost path from s to t. The edges in T are called tree edges and the remaining edges are called non tree edges.*
2. *Modify the cost of each edge in the graph by replacing the cost w(u,v) of every edge (u,v) by w'(u,v) = w(u,v) − d(s,v) + d(s,u). According to the resulting modified cost function, all tree edges have a cost of 0, and non tree edges have a non negative cost.*
3. *Create a residual graph $G_t$ formed from G by removing the edges of G that are directed into s and by reversing the direction of the zero length edges along path $P_1$.*
4. *Find the shortest path $P_2$ in the residual graph $G_t$ by running Dijkstra's algorithm.*
5. *Discard the reversed edges of $P_2$ from both paths. The remaining edges of $P_1$ and $P_2$ form a subgraph with two outgoing edges at s, two incoming edges at t, and one incoming and one outgoing edge at each remaining vertex. Therefore, this subgraph consists of two edge-disjoint paths from s to t and possibly some additional (zerolength) cycles. Return the two disjoint paths from the subgraph.*

You can find proof of correctness and many more about Suurballe's algorithm in this paper: http://www.eecs.yorku.ca/course_archive/2007-08/F/6590/Notes/surballe_alg.pdf
Let's conclude that we solved our current problem (without $C$ flights) by using only Suurballe's algorithm, but how to solve it with additional $C$ chartered flights which we should use only if we can't find two disjoint paths?
As we can see in the summary, Suurballe's algorithm is just Dijkstra's algorithm with some modifications. As we know that in Dijkstra's algorithm we choose next node greedy (a node with the smallest distance from an initial node, who hasn't been visited yet), we can add some value $INF$ to every $C$ flight's weight ($w'(x,y) = w(x,y) + INF$). Value $INF$ should be bigger than the sum of all weights of $R$ flights because then we secure that in Dijkstra's algorithm we'll always use $R$ flights first, and then $C$ flights.
In the end, let's give quick summary of the task in two steps:

1. Perform $w'(x,y) = w(x,y) + INF$ for all C flights
2. Run Suurballe's algorithm

Time complexity is the same as complexity of Dijkstra's algorithm $O(\,|R + C|\; + \;|N|\; lg\; |N|\,)$.
P. S. "Boa viagem" translated from Portuguese means a good trip.

***Added by: Fernando Fonesca [ITA]***
***Solution by:***

*Name:* ***Ivan Šego***
*School: XV Grammar School, Zagreb*
*E-mail: ivan.sego96@gmail.com*

## Problem R2 04: Biology (ID: 7490)

Time Limit: 2 second

Memory Limit: 1536 MB

It was no later than 1869 that Jules Verne succeeded to vulgarize interest in the depths of the oceans through his science-fiction novel “Twenty thousand leagues under the sea”. On board the Nautilus manoeuvred by captain Nemo, the crew visits the lost city of Atlantis and gets to know strangest kinds of sea dwellers.

Nearly a century later, one of the deepest points on Earth, the Challenger Deep was visited by Piccard and Walsh and a Swiss flag was dibbled at 10.924 metres below sea level. The ridership of the Trieste submarine was amazed by the animal life in these depths.

Recently a team of biologists decided to investigate these depths of the Mariana Trench and especially their So Weird Exotic Rare Citizens (SWERC). To this goal a preliminary study was performed, which showed that the species in the Mariana Trench have very local biotopes, which if projected onto the sea surface, can be described by convex polygons. All the biotopes are located at the same depth and some may overlap. The biologists now want to install racks and cameras in each biotope in order to attract and film them. As delicious as the food at the racks might be, no species would ever take the risk to transgress the borders of its habitat. As these cameras and the associated telecommunication system are extremely expensive, their number is to be minimized. Can you tell the biologists for how many cameras they need to account in their budget planning in order not to miss any species if they choose the locations in a clever way? You may consider each camera-rack couple as a mathematical point which must lie strictly inside the biotope in order to attract the related species.

### Input

The input consists of several test-cases separated by an empty line. Each test-case starts with the number of species $S$ $(0 \leq S \leq 20)$ . Each of the next S lines describes one biotope. The first entry indicates the number $n_i$ of vertices of the convex polygon. Then follow their coordinates in the order $x_1\, y_1\, x_2\, y_2 \ldots x_{n_i}\, y_{n_i}$ $(|x_i|, |y_i| \leq 1000)$. Input terminates on a test-case with $S = 0$, which must not be evaluated.

### Output

For each test-case, output the minimum number of camera-rack couples necessary to screen all the species listed in the input.

### Samples

| Input | Output |
|---|---|
| 3<br>3 11.00 0.00 -2.50 7.79 -2.50 -7.79<br>4 4.00 -3.00 -3.00 4.00 -10.00 -3.00<br>-3.00 -10.00<br>4 4.00 2.00 3.00 3.00 2.00 2.00 3.00<br>1.00<br>10<br>7 -317.00 99.00 -330.55 127.15 -<br>361.01 134.10 -385.43 114.62 -385.43<br>83.38 -361.01 63.90 -330.55 70.85<br>6 -99.00 93.00 -238.50 334.62 -517.50<br>334.62 -657.00 93.00 -517.50 -148.62<br>-238.50 -148.62<br>4 113.00 -134.00 42.00 -63.00 -29.00<br>-134.00 42.00 -205.00<br>3 90.00 -68.00 -261.00 134.65 -261.00<br>-270.65<br>7 218.00 -342.00 147.22 -195.02 -<br>11.83 -158.71 -139.38 -260.43 -139.38<br>-423.57 -11.83 -525.29 147.22 -488.98<br>6 131.00 -286.00 38.00 -124.92 -<br>148.00 -124.92 -241.00 -286.00 -<br>148.00 -447.08 38.00 -447.08<br>4 -170.00 -247.00 -172.00 -245.00 -<br>174.00 -247.00 -172.00 -249.00<br>4 -332.00 -102.00 -395.00 -39.00 -<br>458.00 -102.00 -395.00 -165.00<br>6 -52.00 -224.00 -196.50 26.28 -<br>485.50 26.28 -630.00 -224.00 -485.50<br>-474.28 -196.50 -474.28<br>5 -101.00 163.00 -210.87 314.22 -<br>388.63 256.46 -388.63 69.54 -210.87<br>11.78<br>0 | 2<br>5 |

### *Solution:*

Let's restate the problem statement: we need to divide set S of species (convex polygons) into minimum number of groups s.t. intersection of all polygons in one group has non-zero area.

We would like to calculate an array $good[Q]\ =$ whether subset $Q$ of $S$ can form one group or not.

Firstly, how to compute intersection of two convex polygons? Observe, that polygon is an intersection of half-planes, and intersection of half-plane and a convex polygon is still a convex polygon. So all we need is a classic sub-routine computing intersection of a convex polygon and a half-plane. Polygons consist of less than 40 points (it's not given in the problem statement, but it holds), so this part can be done naively in time $n_1 \cdot n_2$ where $n_i$ – number of vertices of $i$-th polygon.

We can use this approach to compute intersection of every triple of polygons, but for larger sets, it will time out. However, now we can use Helly's theorem (https://en.wikipedia.org/wiki/Helly's_theorem) that gives us:

"Set of polygons has non-zero intersection iff every triple of them has non-zero intersection".

So we compute good[Q] dynamically from the small subsets to the larger ones using the following relation:

$good[Q + \{e\}]\ =\ good[Q]\ AND\ good[\{e, a, b\}]$ for all $a, b$ in $Q$.

Now, similar dynamic approach would be enough to compute the final answer, but here we extend subsets by other subsets, not necessarily singletons:

$$if(good[B])\ dp[A\ sum\ B] = min(dp[A\ sum\ B], dp[A]\ +\ 1);$$

Unfortunately, this particular formula leads to $4^n$ solution, but we can easily speed it up:

1. consider only sets B s.t. they are good and maximal inclusion-wise – one can pre-compute values $reallygood[Q] =\ good[Q]\ AND\ good[Q\ +\ \{a\}]\ =\ false$ for all $a$ not in $Q$

2. for given set $A$ take the smallest element $r$ that doesn't belong to $A$ and consider only sets $B$ that contain $e$ – one can precompute the set $V_r = \{Q{:}\ verygood[Q]\ AND\ r\ in\ Q\}$

It's hard to estimate complexity of the final solution, but those optimizations are enough to get this.

***Added by: Christian Kauth***
***Solution by:***

*Name:* ***Bartłomiej Dudek***
*School: University of Wrocław*
*E-mail: bardek.dudek@gmail.com*

## Problem R2 05: Mickey Mouse Magic Trick v6 (ID: 17946)

Time Limit: 99 seconds

Memory Limit: 1536 MB

Mickey Mouse and Donald Duck love magic. They specialize with card tricks. Mickey invented a new trick and they are going to surprise the world. They've contracted series of shows on whole the globe, worth many millions of dolars $$. The first show is coming up but unfortunately Mickey lost secret of the trick. He remembers only trick description, but it's not enough to satisfy the contract conditions. Help them!

Mickey has n cards with values $1, 2, \dots, n$. He invites a spectator from the audience, Donald is outside the stage and see nothing on the stage. The spectator chooses randomly $k$ cards from the pack and discard the remaining $n - k$ cards. Mickey chooses one card from this $k$ cards, shows it to everyone (except Donald) and hides it to the spectator's pocket. Next Mickey leaves the remaining $k - 1$ cards in some order on the table. Donald is coming back. He is the only person, who doesn't know, what is the number in the hiden card. He can see only $k - 1$ cards on the table. Donald thinks for a while, the drum rumbles, at the beginning very silent, then louder and louder, everyone is waiting, the drum stops, a few seconds of deep silence and... Donald says the number on the hidden card. It's correct, applause! How did he discover the number? It's magic!

Mickey and Donald know, that's not magic only smart math manipulations. They asked You to help them. You have to write computer program, that can help them with the trick. The program should be able to do two things: help Mickey to select one card from given $k$ cards and describe order of remaining $k - 1$ cards on the table, then help Donald to guess the hidden card value basing only on $k - 1$ cards left by Mickey on the table. You can use any strategy that You want, but remember - Donald needs to guess the number during the show, because the huge profit $$ depends on it.

### Input

All integers in the same line are single-space separated (the same concerns problem output).

Values $n, k$ are constant. In this problem $n = 725$ and $k = 6$. There are also problems with different values: MMMAGIC3, MMMAGIC4, MMMAGIC5.

The first line of input contains two integers $M, D$, where $M$ is the number of test cases in which Mickey needs help, $D$ is the number of test cases in which Donald needs help ($M + D < 10^6$).

Every of next $M$ lines contains $k$ distinct integers from range $[1, n]$ - the values on cards given to Mickey. The values are sorted in ascending order.

Every of next $D$ lines contains $k - 1$ distinct integers from range $[1, n]$ - the cards left to Donald on the table. The order is the same, as on the table, from left to right.

### Output

For each Mickey's query write a line with $k - 1$ integers - the values on the cards, that Mickey have to leave on the table, from left to right.
For each Donald's query write a line with one integer - the value of hidden card or (if in Your strategy such situation is impossible) any of remaining values.

### Samples

| Input | Output |
|---|---|
| 3 0<br>1 2 3 4 5 6<br>2 4 6 100 200 500<br>4 7 8 111 222 666 | 5 4 3 2 1<br>6 500 4 100 200<br>4 111 7 8 222 |

| Input | Output |
|---|---|
| 0 3<br>5 4 3 2 1<br>6 500 4 100 200<br>4 111 7 8 222 | 6<br>2<br>666 |

### *Solution:*

In order to help Mickey and Donald one can do the following.

When facing Mickeys query we discard the card with index, in sorted list (0 indexed), equal to the remainder the sum of numbers on all $k$ cards modulo $k$.

Knowing that we did just that, once we are required to reverse the process in Donald's query and are facing $k - 1$ cards, sum of which has remainder $x$ modulo $k$, we know that if the discarded card has the number smaller than the one on card with index 0 in sorted list, then the number on that card must have remainder $y$ modulo $k$ so that sum of $y$ and $x$ has remainder 0 modulo $k$ (because otherwise we wouldn't select the card with the smallest number). Similarly we can write rules for every possible value of discarded card and we get this.

**Notation**

$C$ — value written on discarded card.
$d_i$ — value written on $i$-th card in list of cards Donald is presented to when said cards are sorted ($d_0$ smallest value, $d_{k-2}$ largest) (0 indexed)

If $0 < C \leq d_0 - 1 \rightarrow C \equiv y + 0\ \mathit{modulo}\ k$

If $d_0 < C \leq d_1 - 1 \rightarrow C \equiv y + 1\ \mathit{modulo}\ k$

…

If $d_{k-3} < C \leq d_{k-2} - 1 \rightarrow C \equiv y + k - 2\ \mathit{modulo}\ k$

If $d_{k-2} < C \leq n \rightarrow C \equiv y + k - 1\ \mathit{modulo}\ k$

And because the number of integers $x$ for which $a < x < b$ and $x \equiv k\ (mod\ p)$ is the same as number of integers $x$ such that $a - 1 < x < b - 1$ and $x \equiv k - 1\ (mod\ p)$ (or rather $a - m < x < b - m$ and $x \equiv k - m$, for any integer $m$).

The number of cards with value that satisfies upper constrains is the same as the number of integers $x$ for which

If $0 < x \leq d_0 - 1 \rightarrow x \equiv y + 0 - 0 \equiv y\ \mathit{modulo}\ k$

If $d_0 - 1 < x \leq d_1 - 2 \rightarrow x \equiv y + 1 - 1 \equiv y\ \mathit{modulo}\ k$

If $d_1 - 2 < x \leq d_2 - 3 \rightarrow x \equiv y + 2 - 2 \equiv y\ \mathit{modulo}\ k$

…

If $d_{k-1} - k < x \leq n - (k - 1) = k! \rightarrow x \equiv y + (k - 1) + (k - 1) \equiv y\ \mathit{modulo}\ k$

By combining this we get that the number of cards with the number that satisfies our conditions is the same as the number of integers $x$ such that $0 < x <= k!$ and $x \equiv y\ (mod\ k)$ and that number is same for every integer $y$ and is equal to $k!\ /\ k = (k - 1)!$. And as we know we can arrange our $k - 1$ cards in $(k - 1)!$ ways (by doing that giving the index of the permutation we made as information to Donald) so we just assign the $i$-th permutation to $i$-th card in sorted list of cards which satisfy our conditions.

**Implementation**

We can find the number of integers $x$ for which $a < x < b$ and $x \equiv k\ (mod\ p)$ in constant time by finding $a'$, the largest integer for which it holds true that $a' <= a$ and $a' \equiv k\ (mod\ p)$, and $b'$, the smallest integer for which $b' >= b$ and $b' \equiv k \ (mod\ p)$. The number of said integers is same as number of integers $x$ such that $a' < x < b'$ and $x \equiv k\ (mod\ p)$ which is now easily computable as $(b' - a') / p$.

We can also find the $i$-th (0 based) integer $x$, for which $a < x$ and $x \equiv k\ (mod\ p)$ in constant time by finding $a'$ the same way, and realizing that same $x$ is also $i$-th integer for which $a' < x$ and $x \equiv k\ (mod\ p)$ and now $x$ is computable as $x = a' + (i + 1) \cdot p$.

Now let's assume that the sum of numbers on $k$ cards Mickey has in front of him has remainder $x$ modulo $k$. We need to find out the card index of which would, when the card is put with rest of $k - 1$ cards (and sorted), be same as remainder of sum of numbers from all $k$ cards modulo $k$. Using previously stated conditions and 2 methods stated above we can deduce the index $D$ in time complexity $O(k)$.

With $k$ being 6 in this task we can generate all the permutations of integers 0 through 5 and assign one of them to every integer $x$, $0 <= x < k!$ (we can do that in time complexity $O(k * k!)$). So in Mickeys queries we know that we need to arrange the reminding $k - 1$ cards in such way that their indexes represent the $D$-th permutation, and knowing exactly what is $D$-th permutation and getting the sorted list of cards as input we can easily print reminding $k - 1$ in desired order once again in complexity $O(k)$.

So the overall time complexity of Mickeys query is $O(k)$.

In Donald's queries we need to extract the index of the given permutation in list of sorted permutations, there are many ways to do that, and, with this size of $k$, differences between them are not that important. One way to do that is to realize that all the permutations starting with index 0 are before those who start with 1, etc. on the other side number of permutations which start with $x$ $(0 <= x < k)$ is equal (easily proven using mathematical induction) and is equal to $k! / k = (k - 1)!$. So if the first number in list provided to Donald is $i$-th in the sorted list then the index of the permutation $p$ is $i * (k - 1)! <= p < (i + 1) * (k - 1)!$ so we can represent $p$ as $p = i * (k - 1)! + p_1$ now using similar arguments if the second number in provided list is $i_1$-th in the sorted list of elements excluding the first element we can claim that $i_1 * (k - 2)! \leq p_1 < (i_2 + 1) * (k - 2)!$ so we can represent $p_1$ as $p_1 = i_1 * (k - 1)! + p_2$ and so on until we get to $p_{k-2}$ which is 0 because there is only one permutation of size 1.

We can do all that using sort function and function which gives us number of already used numbers smaller than our number (we need this for calculating the position of our number in shortened sorted list). We can achieve time complexity of $O(k * logk)$ using quicksort and using tournament-tree when calculating positions in sorted lists.

Once we know the index $D$ we can, similarly as in Mickeys queries, find which number is $D$-th in sorted list of numbers which satisfy our conditions and print it in $O(k)$, hence solving Donalds queries in overall time complexity of $O(k * logk + k)$.

Time complexity of this solution is $O(k * k! + M * k + D * k * log\ k)$.

Space complexity of this solution is $O(k * k!)$.

***Added by: miodziu***
***Resource:*** *http://mwpz.poznan.pl*
***Solution by:***

*Name:* ***Nikola Spasić***
*School: Grammar School "Jovan Jovanović Zmaj", Novi Sad*
*E-mail: nnspasic1997@gmail.com*

## Problem R2 06: Electrical Engineering (ID: 7680)

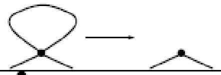
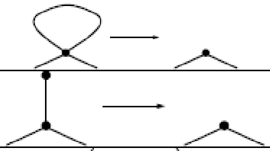
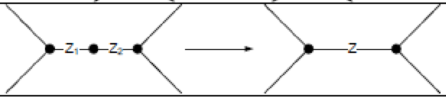
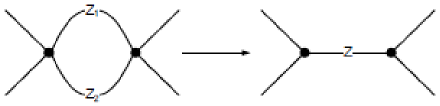
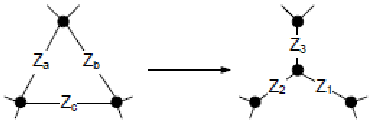
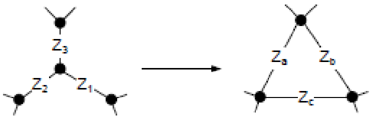
Time Limit: 3 second

Memory Limit: 1536 MB

The electrical engineers' indefatigable strive towards environmentally friendly energy production translated into the recent boom of hydro, solar, wind and geothermal power plants. While the production side seems ready, these ambitious projects have their bottleneck in the transportation and distribution: Besides the energy losses that occur during transportation over long distances, the renewable energy sources cannot provide power on demand – they must be taken as provided by nature. Used at large scale in today's networks, unreliable green energy can disrupt the balance of power grids easily and cause huge damage along with large-scale power outages.

Serious effort is thus put on researching transient and dynamic phenomena in power grids. You are offered a position in the lab for linear and planar distribution networks. Given a description of the distribution network's line impedances $Z_i$ , you are to find the equivalent impedance between some couples of nodes. The knowledge of such equivalent impedances may speed up the network analysis considerably! Impedances are complex number whose real part represents the resistive line behaviour while the imaginary part stands for the capacitive (negative) or inductive (positive) characteristic. Lines are bidirectional, that is impedance(a,b) equals impedance (b,a).

It was proven that any linear and planar graph (can be drawn in a way that its edges intersect only at their endpoints) can be reduced into a single equivalent edge that represents the equivalent impedance between its ending nodes, using the following six transformations:

| | | |
|---|---|---|
| **Empty loop reduction** |  | |
| **Pendant edge reduction** |  | |
| **Series reduction** |  | $\underline{Z} = \underline{Z}_1 + \underline{Z}_2$ |
| **Parallel reduction** |  | $\underline{Z} = \dfrac{1}{\dfrac{1}{\underline{Z}_1} + \dfrac{1}{\underline{Z}_2}}$ |
| **Delta-wye transformation** |  | $\underline{Z}_1 = \dfrac{\underline{Z}_b \cdot \underline{Z}_c}{\underline{Z}_a + \underline{Z}_b + \underline{Z}_c}$<br>$\underline{Z}_2 = \dfrac{\underline{Z}_c \cdot \underline{Z}_a}{\underline{Z}_a + \underline{Z}_b + \underline{Z}_c}$<br>$\underline{Z}_3 = \dfrac{\underline{Z}_a \cdot \underline{Z}_b}{\underline{Z}_a + \underline{Z}_b + \underline{Z}_c}$ |
| **Wye-delta transformation** |  | $\underline{Z}_a = \dfrac{\underline{Z}_1 \cdot \underline{Z}_2 + \underline{Z}_2 \cdot \underline{Z}_3 + \underline{Z}_3 \cdot \underline{Z}_1}{\underline{Z}_1}$<br>$\underline{Z}_b = \dfrac{\underline{Z}_1 \cdot \underline{Z}_2 + \underline{Z}_2 \cdot \underline{Z}_3 + \underline{Z}_3 \cdot \underline{Z}_1}{\underline{Z}_2}$<br>$\underline{Z}_c = \dfrac{\underline{Z}_1 \cdot \underline{Z}_2 + \underline{Z}_2 \cdot \underline{Z}_3 + \underline{Z}_3 \cdot \underline{Z}_1}{\underline{Z}_3}$ |

Now that you have all the necessary operations available, are you able to determine the equivalent impedance between several couples of nodes?

### Input

The input consists of several test-cases separated by an empty line. Each test-case starts with the number of nodes $N$ $(1 \leq N \leq 100)$, the number of bidirectional connections $C$ $(0 \leq C \leq 1000)$ and the number of equivalent impedances to compute $Z$ $(0 \leq Z \leq 10)$ on a line. Then follow $C$ lines, each describing one bidirectional connection in the form ‘EndPoint_1’ ‘EndPoint_2’ ‘Impedance’. ‘EndPoint_1’ and ‘EndPoint_2’ are in the range $1$ to $N$ and impedance has the format ‘re im’ where re and im designate the real and imaginary parts respectively, both being real numbers $d$ such that $10^{-3} < |d| < 10^{3}$. The next $Z$ lines each hold two integers, the indices of the nodes between which you are to compute the equivalent impedance. Input terminates on a test-case with $N = C = Z = 0$, which must not be evaluated.

### Output

For each couple of endpoints, output the equivalent impedance in the form ‘re im’ where re and im designate the real and imaginary parts respectively. If the nodes are not connected, output ‘no connection’. Electrical engineers will consider your result as correct if the absolute error on the real and imaginary parts is below $10^{-2}$. Finish each test-case on a blank line.

### Samples

| Input | Output |
|---|---|
| 5 10 3<br>3 1 12.317 -0.779<br>5 3 30.107 0.289<br>5 1 27.447 -22.649<br>4 2 15.351 24.371<br>5 5 19.63 -3.549<br>2 2 11.841 18.757<br>4 5 4.834 -16.542<br>3 5 5.022 -22.387<br>2 5 24.768 -22.356<br>5 2 27.351 12.053<br>1 2<br>2 3<br>3 3<br><br>10 10 4<br>9 8 6.36 17.411<br>1 3 27.596 -6.484<br>9 10 4.735 -8.282<br>8 8 6.901 27.939<br>8 4 14.894 3.729<br>5 4 14.311 -2.422<br>10 10 11.009 6.225<br>4 4 3.196 -32.703<br>10 9 15.282 -14.799<br>3 9 20.473 27.158<br>10 9<br>8 1<br>2 9<br>9 6 | 23.37 -7.26<br>19.61 -6.97<br>0.00 0.00<br><br>3.79 -5.46<br>54.43 38.09<br>no connection<br>no connection |

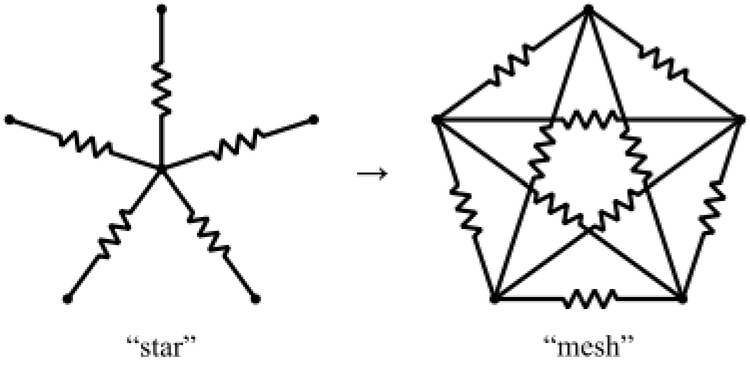
### *Solution:*

Prerequisites: Nodal analysis, Star-mesh transform, Gaussian elimination, Union find

#### Introduction

Main idea of solution is to put current source between query nodes and then determine voltage between nodes. After that, calculating impedance is easy ($Z = U\ /\ I$). Mentioned algorithm was to slow. Its complexity is $O(T * N3 * Z)$, $T$ - number of test cases, $N$ - number of nodes, $Z$ - number of queries. Bottleneck in algorithm is number of nodes so I had to reduce it. I've done that using star-mesh transform.

Star-mesh transform



The equivalent impedance and admittance between nodes $A$ and $B$ are given by:

$$Z_{AB} = Z_A Z_B \sum \frac{1}{Z}, Y_{AB} = Y_A Y_B \frac{1}{\sum Y}$$

Where $Z_A(Y_A)$ is the impedance (admittance) between node $A$ and the central node being removed.

**Detailed description**

First step: Create graph and union find.

Create graph that represents electric circuit. It's easier to work with admittance ($Y = 1 / Z$) than with impedance because parallel reduction is simplified to $Y = Y1 + Y2$. Besides creating graph, you have to create union find data structure. I will use that structure to determine which nodes aren't connected.

```
int N, C, Z;
unordered_map<int, complex<double>> links[MAXN];
UnionFind uf(MAXN);
for (int i = 0; i < C; i++)  {
    fscanf(fin, "%d %d %lf %lf", &a, &b, &re, &im);
    if (a != b) {
        links[a][b] += 1.0 / complex<double>(re, im);
        links[b][a] += 1.0 / complex<double>(re, im);
        uf.connect(a, b);
    }
}
```

Second step: Process queries.

Store all queries in list and mark all nodes that participate in queries (you can skip queries if nodes in query are same or nodes aren't connected).

```
pair<int, int> queries[MAXZ];
set<int> inQuery;
bool skip[MAXN];
for (int i = 0; i < Z; i++) {
    fscanf(fin, "%d %d", &a, &b);
    if (a == b) {
        queries[i] = make_pair(EQUAL, 0);
    } else if (!uf.isConnected(a, b)) {
        queries[i] = make_pair(NOT_CONNECTED, 0);
    } else {
        queries[i] = make_pair(a, b);
        skip[a] = skip[b] = true;
        inQuery.insert({ a, b });
    }
}
```

Third step: Simplify graph

Remove nodes with one connection that aren't mark. After that apply star-mesh transform to other nodes that aren't marked (nodes with lower degree have greater priority).

```
unordered_set<int> simplify;
for (int i = 1; i <= N; i++)  {
    if (!skip[i]) {
        simplify.insert(i);
    }
}
while (!simplify.empty()) {
    int minId = *simplify.begin();
    for (auto nodeId : simplify) {
        if (links[nodeId].size() < links[minId].size()) {
            minId = nodeId;
        }
    }
    simplify.erase(minId);
    // Star - mesh
    complex<double> admSum = 0.0;
    unordered_map<int, complex<double>> &ref = links[minId];
    for (auto node : ref) {
        links[node.first].erase(minId);
        admSum += node.second;
    }

    admSum = 1.0 / admSum;
    for (auto it = ref.begin(); it != ref.end(); it++) {
        auto kt = it;
        for (auto jt = ++kt; jt != ref.end(); jt++) {
            imp = (it->second * jt->second) * admSum;
            links[it->first][jt->first] += imp;
            links[jt->first][it->first] += imp;
        }
    }
}
```

Fourth step: Split nodes

Split nodes to connected components and create matrix for each components.

```
for (auto id : inQuery) {
    components[uf.find(id)].push_back(id);
}
```

Fifth step: Create matrices, solve system of equations and print answer

For each query find corresponding connected component, add current source between nodes, create matrix and finally solve the solve system. Created matrix represents system of equations that is used to apply nodal analysis (each system has at most $2 \cdot Z$ variables).

```
// This array is used for mapping node to its position in matrix
int idToPos[MAXN];
for (int i = 0; i < Z; i++) {
    if (queries[i].first == EQUAL) {
        puts("0.00 0.00");
        continue;
    }

    if (queries[i].first == NOT_CONNECTED) {
        puts("no connection");
        continue;
    }

    vector<int> &ref = components[uf.find(queries[i].first)];
    for (int j = 0; j < ref.size(); j++) {
        for (int k = 0; k < ref.size(); k++) {
            tmpMat[j][k] = 0.0;
        }
    }

    for (int j = 0; j < ref.size(); j++) {
        idToPos[ref[j]] = j;
    }

    // Create matrix
    for (auto id : ref) {
        for (auto link : links[id]) {
            a = idToPos[id];
            b = idToPos[link.first];
            tmpMat[a][b] -= link.second;
            tmpMat[a][a] += link.second;
        }
    }

    // Init B vector
    for (int j = 0; j < ref.size(); j++) {
        B[j] = 0.0;
    }
    // Add current source
    B[idToPos[queries[i].first]] = -1;
    B[idToPos[queries[i].second]] = 1;

    // Solve sytem
    solveGauss(ref.size(), tmpMat, B);
    complex<double> sol = X[idToPos[queries[i].second]] -
                          X[idToPos[queries[i].first]];
    printf("%lf %lf\n", sol.real(), sol.imag());
}
```

**Conclusion**

Complexity of improved algorithm is $O(T * (N^3 + Z^4))$. If $Z$ is lower than $N$, algorithm has better performance.

***Added by: Christian Kauth***
***Solution by:***

*Name:* ***Srđan Milaković***
*School: Faculty of Technical Sciences, University of Novi Sad*
*E-mail: srki.accounts@gmail.com*

## Problem R2 07: Turn on the Lights 2 (ID: 9535)

Time Limit: 1-5 second

Memory Limit: 1536 MB

Recently Zippy received a puzzle. It is an $n \cdot m$ matrix. In most cells of the matrix, there is a light with a switch. However, some cells do not contain a light with a switch, and they are called "blocks". Once he flips the switch in a cell, lights in visible cells from it (including itself) change to its opposite state (which means: on->off, off->on). One cell is visible from another, iff they are in the same row or the same column, and there aren't any blocks between them (and of course the two cells should not be blocks). Zippy wants to turn on all the lights. Please help him to solve the puzzle.

### Input

First line, $n, m$.

The following $n$ lines, each line is a $m$-length string, represting the original state. (0 means on, 1 means off and 2 means a block)

$1 \leq n, m \leq 300$

$number\ of\ blocks \leq\ max(n, m)$

### Output

$n$ lines, each line is a $m$-length string. It's obvious that if a valid solution exists, there exists a solution that every switch is flipped no more than once. So 1 means the switch is flipped once and 0 means the swtich remains unflipped. Of course, a block do not contain a switch, so for the cell, you should always output 0.
It's guaranteed that there always exists a solution. If there are multiple solutions, output any of them.

### Samples

| Input | Output |
|---|---|
| 2 3<br>011<br>121 | 001<br>100 |

### *Solution:*

Let's call all cells that are not blocks „lights” and assume that $m \leq n \leq 300$.

The obvious algorithm that finds a solution is the following:

1. For each light create a boolean variable $x_{i,j}$ – it will determine, whether this switch has to be flipped or not
2. Create a system of boolean equations. For each light $(i, j)$ there is the following equation: xor of all visible lights from it must be 0 or 1, depending on the original state of the board
3. Use Gaussian elimination (https://en.wikipedia.org/wiki/Gaussian_elimination) to solve the system

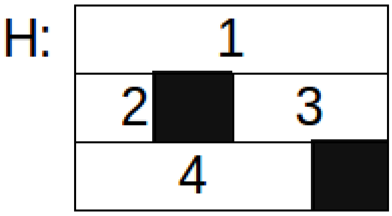
Recall that Gaussian elimination runs in $O(R\text{^}3)$ time, where $R -$ number of variables if there is one equation for each variable, so it is too slow for us as $R \leq n^2$.

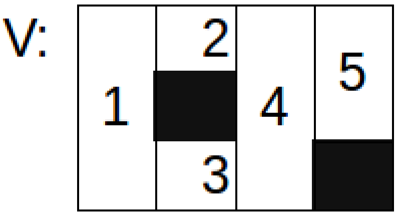
Let's try to construct a smaller system of equations.

Now divide the board into horizontal stripes – non- extendable contiguous sets of lights with common y-coordinate (see the picture). Observe that there are less than $2 \cdot n$ of them, as in an empty board there are

$n$ stripes and every block divides a stripe into two smaller ones. Let's enumerate them: $H_1, H_2, \ldots, H_c$, where $c < 2 \cdot n$.

Similarly, there are vertical stripes: $V_1, V_2, \ldots V_d$, where $d < 2 \cdot n$.





Now let's construct boolean variables $h_i, v_i$ that will count (modulo 2) number of flipped switches in $i$-th region, for example: $h_i =$ xor of all $x_{a,b}$ s.t. $(a, b)$ belongs to the stripe $H_i$. (*)

Does it give us any improvement? Now there are even more variables and equations, but observe that:

$$x_{i,j}\ xor\ h_a\ xor\ h_b\ =\ input_{i,j}$$

where $a$ – number of horizontal stripe that light $(i, j)$ belongs to and $b$ – vertical.

We can rearrange it and obtain:

$$x_{i,j}\ =\ h_a\ xor\ h_b\ xor\ input_{i,j}\ (**)$$

Now we can substitute every occurrence of variable $x_{i,j}$ in equations (*) using formula (**) obtaining system of $c + d < 4n = 1200$ equations. Here we can use Gaussian elimination (if it times out one can use binary operations and achieve $O(R\^{}3/32)$ time) and compute all variables $h_i$ and $v_i$.

Finally, we use equations (**) to reconstruct the final solution.

***Added by: sevenkplus***
***Resource:*** *own problem*
***Solution by:***
*Name:* ***Bartłomiej Dudek***
*School: University of Wrocław*
*E-mail: bardek.dudek@gmail.com*

## Problem R2 08: Move Marbles (ID: 6299)

Time Limit: 0.1 second

Memory Limit: 1536 MB

You have $N$ marbles and $K$ slots. You have to follow the below mentioned rules :

1. You can put in a marble or take out a marble from slot numbered 1 at any time.
2. You can put in a marble or take out a marble from slot numbered $i$ only if there exists a marble at the slot $i - 1$.
3. The game stops when a marble reaches the slot numbered $K$ for the first time.

Your task is to finish the game in minimum number of valid moves.

### Input

The first line contains $t$, the number of testcases. Then on each line is given two numbers $N \le 15, K \le 2^{N-1}$.

### Output

Print two numbers namely the number of "put in" moves and the number of "remove from" moves respectively for all the tests such that you move to the $K$-th slot in minimum number of valid moves. See explanation section below for more details.

### Samples

| Input | Output |
|---|---|
| 1<br>3 6 | 6 3 |

### Explanation:

The following are the valid moves for the given input:
PUT IN 1
PUT IN 2
PUT IN 3
REMOVE FROM 2
REMOVE FROM 1
PUT IN 4
PUT IN 5
REMOVE FROM 4
PUT IN 6

### *Solution:*

Let's first consider a similar game that uses a modified version of the third rule:

> The game stops when there is a marble on the slot numbered $K$ and the board contains no other marbles.

We will call this modified game full game, and the original game (with rules as described in the statement) addition game. We define:

- $A(N, K)$ = The minimum number of valid moves needed to solve the addition game with $N$ marbles and $K$ slots
- $F(N, K)$ = The minimum number of valid moves needed to solve the full game with $N$ marbles and $K$ slots

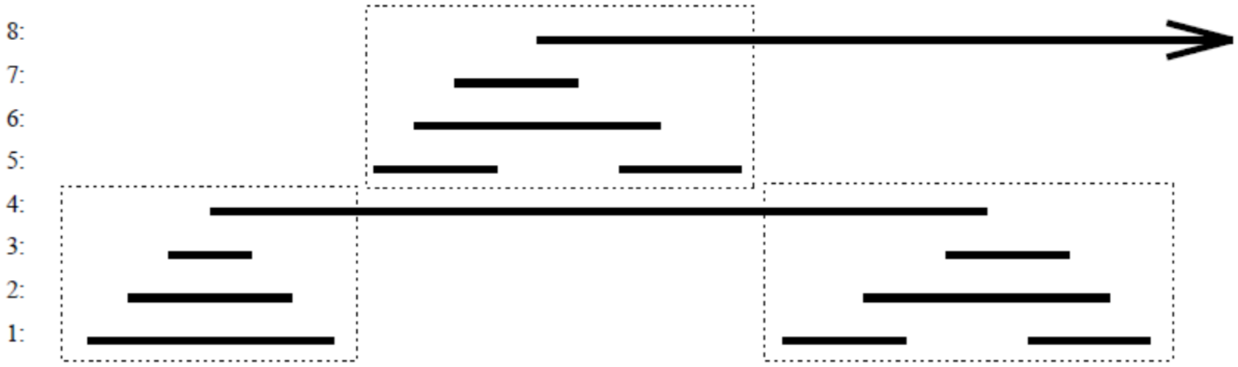
For convenience, we will use the same notation to reference particular games. For example, a full game with parameters $N = 4$, $K = 8$ will be called "the $F(4,8)$ game".

The full game as we defined it is equivalent to Bennett's pebble game [1]. Some proofs that are omitted in this solution can be found in the reference paper.

Let's first focus on the full game. Consider a special slot $M$, such that $1 \le M < K$. It can be shown that, with the right choice of $M$, the full game can be solved in the minimum number of valid moves. In order to do that we will divide the game, with respect to $M$, into the following three parts:

1. Get a single marble to the slot $M$ and then retrieve all marbles used in the process (slots 1 to $M - 1$)
2. Get a single marble to the slot $K$, using the marble we left at the slot $M$, and retrieve all marbles used in the process (slots $M + 1$ to $K - 1$)
3. Remove the marble from the slot $M$ and then retrieve all marbles used in that process (slots 1 to $M - 1$)

As an example, we will take a look at the graphical representation of a solution to $F(4,8)$. The $X$ axis represents the time while the $Y$ axis represents the slots. The sections that can be seen correspond to the three parts we just mentioned.



Dividing the game into smaller parts is a natural step in dynamic programming, so let's try to find an equivalence between these three parts and the "smaller" instances of games, in order to find a recurrence relation:

1. Part one is obviously the same as the $F(N\ ,M\ )$ game.
2. Part two is also a full game, namely $F(N - 1, K - M\ )$. Since we're not using the marble at the slot $M$ we are left with $N - 1$ marbles and we can always place/remove marbles from the slot $K + 1$.
3. Part three is, although this time it's less obvious, still a full game. This equivalence is clearer if we reverse the order of moves and let each addition become a removal, and each removal become an addition. The new sequence of moves represents a solution to the $F(N - 1, M\ )$ game.

These transformations lead us to a recurrence relation:

$$F\ (N,K) = min(F\ (N,M\ ) + F\ (N - 1, K - M\ ) + F\ (N - 1, M\ )\ |\ 1 \le M < K)$$

Of course, there is a base case $F\ (N,1) = 1, \forall N$ and it is easily proven by induction that

$F\ (N,K) = \infty$ for every $K > 2^N - 1$.

The recurrence relation for the addition game can be found in a similar manner. We use the same solution as in the full game except we stop playing once we place a marble on the slot $K$. This means that we go through the whole first part $F(N\ ,M\ )$, and only the first half of the second part which is now equivalent to the addition game $A(N - 1, K - M\ )$:

$$A(N,K) = min(F\ (N,M\ ) + A(N - 1, K - M\ )\ |\ 1 \le M < K)$$

The base case $F\ (N,1) = 1, \forall N$ remains the same but now $F\ (N,K) = \infty$ for every $K > 2^N\ - 1$.

Using these equations, we can compute full $F\ (N,K)$ and $A(N,K)$ tables by iterating over each value of (N,

K) and trying all possible values of M < K. The time complexity of this solution is $O(NK^2)$ which is just too much, so this solution receives a Time Limit Exceeded verdict. We have to find a way to make this computation more efficient, and since iterating over all $(N, K)$ pairs is hardly avoidable the most logical part to try to optimize is the innermost loop, the one in which we try all possible values of $M$ in order to find the minimum number of moves needed to solve the game.

Let's define the function $S(N, K, m)$ as the minimum number of moves needed to complete an addition game with parameters $N$ and $K$ using the slot $m$ as $M$. We're interested in the properties of this function for fixed $N$ and $K$ we could possibly exploit, so we will observe its behaviour on various choices of those parameters and for each value of $m$ such that $S(N, K, m) < \infty$. Here's a random example:

| $M$ | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S(7, 106, m)$ | 537 | 533 | 531 | 529 | 529 | 529 | 531 | 533 | 537 | 541 | 545 | 551 | 557 | 563 | 569 | 575 | 585 | 597 | 609 | 623 | 637 | 653 |

Table 1: The table of results for (N, K) = (7, 106) and m ∈ [43, 64]

Results like these suggest that the function is unimodal, which allows us to use a divide and conquer technique called ternary search to find its minimum more efficiently. Replacing our linear search with this approach yields a time complexity of $O(NK\ log\ K)$, which is good enough to pass all the test cases.

***Added by: Paranoid Android***
***Solution by:***

*Name:* ***Nikola Jovanović***
*School: Mathematical Grammar School, Belgrade*
*E-mail: nikolajovanovic96@gmail.com*

## Problem R2 09: Perfect Maze (ID: 9190)

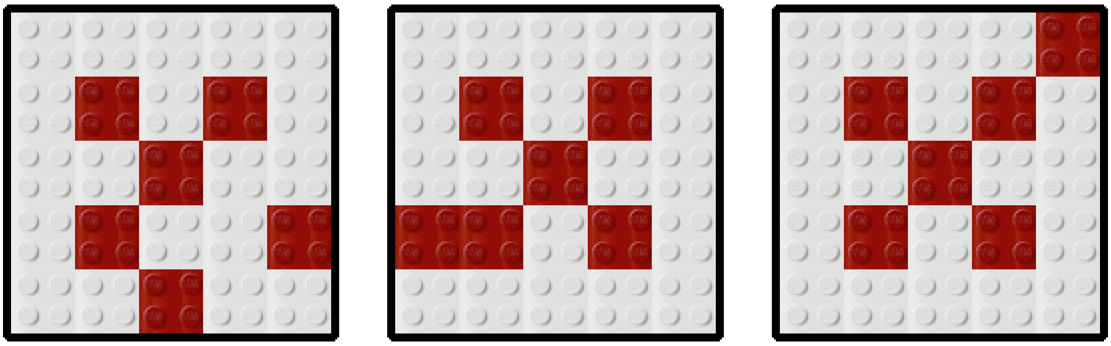
Time Limit: 1 second

Memory Limit: 1536 MB

Hi Guys,

Yesterday my nephew visited me and we played with lego. We took board of 5 x 5 and created different mazes. We decided that we need perfect maze - One that has minimum lego pieces, but there should be exactly one path between each pair of cells without lego. We quickly found out that we needed just 6 pieces and built all 22 configurations using them.

Some of them:



But then he took his board of 13 x 13 and said - let's build perfect maze here! Now we have no idea how many pieces we need and how many configurations we should build. Please help!

### Input

No input

### Output

Two numbers - number of single lego pieces we need to build perfect maze on 13x13 board and number of configurations of them.

### *Solution:*

We will solve the problem using dynamic programming. We fill the grid row by row with legos and empty spaces. When we completely cover first few rows, we don't care exactly how is the upper part of the grid filled. For each cell in the last row, we keep track of whether there is a empty space or lego. To be more precise, we will describe our current state as a tuple with number of elements equal to the number of columns. Each element can be either of:

0 - there is lego in that cell

number larger than 0 - all cells with same number are connected in previously filled rows and are not connected with other cells with different number

For example tuple (1, 0, 1, 0, 2, 2, 0, 3) means that there is lego in second, fourth and seventh cell in the last row we filled. There is connected empty space in first and third cell, there is connected empty space in fifth and sixth cell and there is empty space in eight cell. First and fifth cell are not connected, first and eight cell are not connected, fifth and eight cell are not connected...

To every valid state we try to add every possible row (there are $2^{13}$ of them). When we add a new row, we check whether there is a cycle (i. e. do we connect already connected cells). If there is a cycle we discard that transition, otherwise transition is valid and we update out dynamic programming table.

In order to efficiently implement this solution, we have to normalize our tuples. Notice that the tuples (1, 0, 1, 0, 2, 2, 0, 3) and (3, 0, 3, 0, 1, 1, 0, 2) are equivalent, and we hash them to the same value. For each tuple we memoize smallest number of legos used and number of ways to do that. After filling the whole table we check states with only one component of empty spaces for solution.

It's not easy to do complexity analysis on this problem, but since we only need to submit the result, we can run program locally to calculate 13 x 13 problem.

***Added by: Oleg***
***Solution by:***
*Name**:** **Mislav Bradač***
*School: University of Zagreb*
*E-mail**:** mislav.bradac@gmail.com*

## Problem R2 10: [CH] The Secret Recipe (ID: 18901)

Time Limit: 10 seconds

Memory Limit: 1536 MB

As the Carnival period draws to an end, the remote Kingdom of Byteland is making preparations for the donut feast of Mardi Gras. The King's Court is eagerly anticipating the arrival of notable guests and crowned heads from all round the continent, who will partake of the ceremonial strawberry donut dinner in the Royal Halls. But this year, just before the feast is due to commence, disaster has struck: The Recipe for strawberry donuts, jealously guarded by the Lord Master of the Household, has been destroyed in a kitchen fire caused by an overturned frying pan!

Panic and despair have spread through the King's Court. Without The Recipe, noone is able to fry a half-reasonable donut. The one hope of salvation is in an old, almost forgotten legend, which says that in a far-off part of Byteland, there lives a dynasty of Wicked Old Witches, who hold another copy of The Recipe, treasuring it and passing it on from mother to daughter. In a valiant effort to save the face of the Kingdom of Byteland, you set out on an expedition to find the Witches and their strawberry donut Recipe!

After climbing many hills and crossing many seas, you eventually reach a small village in which all of the legendary Witches dwell. The good news is that one copy of the Recipe is still in existence, and is held by one of the living witches. To make things even better, it turns out that the Witches are not as Wicked as the legend states: in fact, they are quite sociable, and may even help you locate the witch who has the Recipe. However, each witch will insist on telling you her life story first, before you can get any useful information out of her. And they are all Old Witches, so it may take a while, and you really don't want to interrupt a Witch when she is talking. Only after a Witch has recounted her countless tales, will she tell you what she knows about the recipe, which will be something along the following lines:

- "The recipe? Oh dear, I'm sorry, I've never seen it myself..."
- "The recipe? Yes, I do remember... I had it once, many years ago, but I have since passed it on to my most talented daughter (and here comes the name of the daughter in question). Her cooking is so much better than mine, you know!"
- "The recipe? But why of course! Here it is. Just make sure you keep it safe!" (This is the answer you want to hear, but you will only hear it from the one Witch who has the recipe.)

Mardi Gras is drawing near, and you may sadly not have enough time to talk to all the charming Old Witches. Try to figure out a way to obtain the recipe, and to do it as quickly as possible!

### Input

The input starts with a single positive integer integer $k$, denoting the number of Witches in the dynasty ($k < 10^6$). The next line starts with the name of the oldest Witch, from whom all the other Witches are descended, and who of course must have once held the recipe, even if she may perhaps not have it now. The input line describing this Witch is of the form:

***Name*** chats $\boldsymbol{x}$ min.

Each of the next $k - 1$ lines contains a description of a Witch, formatted as follows:

***Name*** is the daughter of ***Mother's-Name*** and chats $\boldsymbol{x}$ min.

The number $x$ describes the number of minutes of your time a Witch will take up if you talk to her: an integer between $1$ and $10^6$. All names of witches are sequences of exactly 5 letters, starting with one upper-case letter (A-Z) and followed by 4 lower-case letters (a-z). The order of Witches at input is such that each Witch is described later than her mother. No two Witches in the dynasty share the same name.

### Output

Your output should describe the strategy you adopt when talking to the witches, with a complete plan of action, stating who to approach in what order, depending on previously obtained answers.

First, print the name of the witch you approach as the first, followed by a list of clauses of the form ANSWER -> FURTHER_ACTION, one clause corresponding to each of the possible answers which may be given by the Witch. For every Witch, you should consider all possible answers the witch may give you, which are feasible given the information you have obtained so far from other Witches. For compactness, assume that in the case when a Witch directs you to her daughter, the ANSWER string is simply to be the name of the daughter in question. If the Witch answers that she has no idea of the recipe, the ANSWER string should be: NEVER. If a Witch tells you she has the recipe, no further action needs to be taken, and you should not write any text corresponding to such an answer in your output file. (In particular, if you can be sure that the Witch you have just asked has the recipe, you should not print text corresponding to any answers given by this Witch.) The ANSWER string should be followed by a description of FURTHER_ACTION to take, representing actions which you will subsequently take, subject to the given answer. The FURTHER_ACTION should be written down following the same grammar rules as those describing the whole of your program's output (i.e., this paragraph is to be read recursively).

White spaces (spaces, newlines, and tabs), numbers, and punctuation marks may be inserted arbitrarily into your program's output to improve its formatting; they will be disregarded when testing your solution. The order in which you consider the possible answers of each Witch is arbitrary.

In your strategy, you may never approach a Witch if, based on information received so far, you can be sure she does not have the recipe. You can assume that Witches always tell the truth, and should not print any text corresponding to the case of an answer which you already know may not occur. You can also safely assume that the recipe does not change hands while you are in the Witches' village.

### Scoring

For each data set, the score of your program will be computed as the ratio of two integer values, $t$ / $T$. Here, $t$ denotes the number of minutes required for a conversation with the most talkative Witch in the dynasty, and $T$ is the total amount of time your strategy takes to find the recipe for a worst-case scenario of answers. Your program will be tested for multiple data files. The overall score of your program will be the average of scores obtained for individual data sets.

### Notes

The time limit of your program for each data set is 10 seconds. Your program must not print more than 30MB of output text (whitespace included!). The number of Witches k will be *approximately* $10^3$ (for half of the tests) and approximately $10^5$ (for the remaining tests).

### Samples

| Input | Output |
|---|---|
| `5`<br>`Alice chats 20 min.`<br>`Betty is the daughter of Alice and chats 10 min.`<br>`Chloe is the daughter of Alice and chats 10 min.`<br>`Marie is the daughter of Chloe and chats 30 min.`<br>`Suzie is the daughter of Marie and chats 10 min.` | `1. Chloe:`<br>`   * "Marie!"`<br>`      2. Suzie:`<br>`         * "NEVER!"`<br>`            3. Marie.`<br>`   * "NEVER!"`<br>`      2. Alice:`<br>`         * "Betty!"`<br>`            3. Betty.` |

## *Solution:*

*The equivalent problem: Tree decomposition*

The problem is equivalent to finding a certain kind of decomposition of a tree while minimizing its height. The decomposition of a tree $T$ is a rooted tree $T'$ such that:

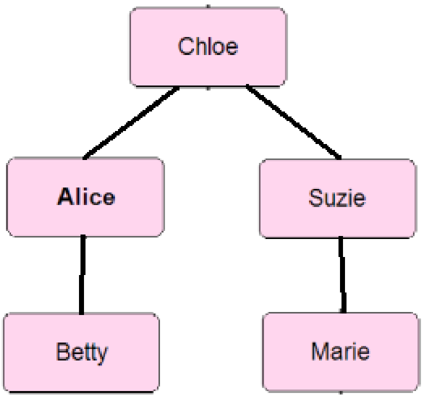
1) Some node $u \in T$ is the root of $T'$.

Let the connected components of $T$ formed after removing $u$ be $T_1, T_2, \dots T_k$. Let $u_1, u_2, \dots, u_k$ be the tree decomposition roots of $T_1, T_2, \dots, T_k$, and let these decompositions be $T_1', T_2', \dots, T_k'$. Then,

2) $u$ is connected with $u_1, u_2, \dots, u_k$ by an edge.

The goal is to minimize the height of $T'$.

For example, the decomposition of the tree given in the original problem's example is:



*The solution*

Our solution uses two tree solving functions ("solvers"), one *Simple* and one *Complex*. Both use another function – an estimator function ("The Estimator"). The Estimator's task is to (you guessed it) estimate the depth of the tree decomposition with some node as its root without actually computing it.

The Simple Solver calls the Estimator and using it finds the best potential root, then recursively calls itself on the connected components formed by removing the root. The Simple Solver has another template parameter telling it whether to print the solution it finds or not - more on this in a minute.

The Estimator works by computing, for each node, the maximum size of a subtree (the number of nodes), the maximum depth (the total weight of all nodes on a path) and the maximum node depth (the number of nodes on the path of maximum depth). The actual estimate is a linear combination of these three factors, with manually tweaked coefficients. This works in linear time and produces an estimate for each node of the tree as the root of the decomposition; therefore, it's easy to not just get the best candidate node, but the best five, or twenty... This is exactly what the Complex Solver does:

The Complex Solver works by first calling the Estimator and then the Simple Solver (without printing) on the best L candidates. The value of L is:

1) Equal to the number of nodes in the (sub)tree if the number of nodes in the original tree is small (up to a few thousand),
2) Equal to the number of nodes if the number of nodes in the current subtree is no more than 19,
3) Equal to 12 otherwise.

The two numbers (12 and 19) were also manually tweaked.

The Complex Solver then chooses the root of a subtree not based on an estimate (unlike the Simple Solver), but instead, based on actual results of calling the Simple Solver. The best of the L candidates is chosen as the root, and the Complex Solver recursively calls itself.

The actual solution is to call the Complex Solver on the entire tree.

***Added by: Adrian***
***Resource:*** *Sphere CEBIT Open Challenge*
***Solution by:***
*Name**:** **Lazar Grbović***
*School: Faculty of Sciences and Mathematics, University of Niš*
*E-mail: lazar96grobarska@gmail.com*

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

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

***Bubble Cup Crew***

# ProblemSet
## booklet

bubblecup.org

Government of the Republic of Serbia
Ministry of Education

INFORMACIONO DRUŠTVO SRBIJE
Društvo za informacione sisteme i računarske mreže