Problem A: Avoiding accidents

You have a collection of four-character words on your desk. However, your friends and their little child are visiting you next week. If the toddler finds and tries to eat your words, it may choke on them and die. To prevent such an accident, you want to hide your words into a single longer string that won't fit into the baby's mouth.

Problem specification

You are given exactly ten strings. Each of them consists of exactly four characters. Construct a new string of length exactly n which contains all ten given words as substrings.

Each substring must be contiguous. For example, `ABXCD` does **not** contain `ABCD` as a substring. The occurrences of the ten given strings may appear in any order and they may overlap arbitrarily.

This problem has two independent subproblems:

- In the **easy subproblem A1** you have to produce a string of length $n = 42$.
- In the **hard subproblem A2** you have to produce a string of length $n = 39$.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case consists of exactly ten space-separated strings. Each string consists of exactly four UPPERCASE English letters (from `A` to `Z`).

Output specification

For each test case, print a single line with a string of UPPERCASE letters. Each of given ten words must appear in your string as a substring at least once. The length of the string must be exactly 42 if you are solving the easy subproblem, and it must be exactly 39 if you are solving the hard subproblem.

The inputs are chosen in such a way that a solution always exists. If there are multiple solutions, you may choose and output any one of them.

What to submit

Do not submit any programs. Your task is to produce and submit the correct **output files** `a1.out` and `a2.out` for the provided input files `a1.in` and `a2.in`. Each line in the file `a1.out` must contain a 42-character string. Each line in the file `a2.out` must contain a 39-character string.

Example

Here is one possible input file:

```
2

TEST INTE RNET PROB ROBL OBLE BLEM SOLV VING TEST

AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA AAAA
```

If n were 29, this would be one possible correct output:

```
INTERNETPROBLEMSOLVINGCONTEST
AAAAAAAAAAAAAAAAAAAHELLOWORLD
```


Problem B: Bounding box

Old Mirko has a son: Mirko junior. A while ago, Mirko senior bought Mirko junior a lot of plastic balls of various sizes. The junior has learned a lot by playing with them.

Now that the junior has grown older, Mirko senior would like to give his son harder challenges. One day after another play session, Mirko senior asked him to put away all balls into a wooden box for easier storage. Unfortunately, Mirko junior is struggling to put everything into the box. Can you help him?

Problem specification

You are given an empty wooden box of dimensions $w \times h \times d$. You are also given a number of plastic balls. Your task is to place all balls into the box. All balls must be completely inside the box. No two balls may intersect. Any such placement will be accepted. In particular, when constructing the placement of balls in the box you may ignore gravity and leave some of the balls floating in the air without support.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a number of lines.

The first line contains floating point numbers $w, h, d, (1 \leq w, h, d \leq 250)$ specifying the dimensions of the wooden box. The box has one corner at x, y, z coordinate $(0, 0, 0)$ and the opposite one at (w, h, d).

The second line contains an integer n, specifying the number of different types of balls. Each ball type is described by a line containing two numbers c and r. The integer $c, (1 \leq c \leq 150)$, specifies the number of copies of the ball type you own and the floating point number $r, (0.001 \leq r \leq 15)$, specifies the radius of the balls. There are **at most 150 balls** in total and all floating point numbers have at most 8 digits after the decimal point.

In the **easy subproblem B1**, $1 \leq n \leq 2$. In the **hard subproblem B2**, $1 \leq n \leq 5$.

Output specification

For each test case, print multiple lines. Print an empty line after each case.

The output for each test case should contain as many lines as there are balls in the box, each line describing one of the balls. The description of each ball has the form "$i\ x\ y\ z$", where i is the type of the ball and (x, y, z) are the coordinates of its center. Ball types are numbered 1 through n in the order in which they appeared in the input.

Balls can be printed in any order, but make sure to use exactly the entire set of the balls you own. All balls must fit inside the box. A solution is guaranteed to exist.

Your answer should have an absolute error of at most 10^{-6}, in particular:

- Two balls with radii r_1 and r_2 respectively are considered intersecting if their centers are closer than $r_1 + r_2 - 10^{-6}$ units.
- A point is outside of the box if the point lies at least 10^{-6} beyond the box's boundaries.

Example

input

```
1

8 8 8
2
1 4
2 0.9
```

output

```
1 4 4 4
2 1 7 1
2 1 7 7

```


Problem C: Counting swaps

Just like yesterday (in problem U of the practice session), Bob is busy, so Alice keeps on playing some single-player games and puzzles. In her newest puzzle she has a permutation of numbers from 1 to n. The goal of the puzzle is to sort the permutation using the smallest possible number of swaps.

Instead of simply solving the puzzle, Alice is wondering about the probability of winning it just by playing at random. In order to answer this question, she needs to know *the number of optimal solutions* to her puzzle.

Problem specification

You are given a permutation $p_1, \ldots, p_n$ of the numbers 1 through n. In each step you can choose two numbers $x < y$ and swap p_x with p_y.

Let m be the minimum number of such swaps needed to sort the given permutation. Compute the number of different sequences of exactly m swaps that sort the given permutation. Since this number may be large, compute it modulo $10^9 + 9$.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the integer n. The second line contains the sequence $p_1, \ldots, p_n$: a permutation of $1, \ldots, n$.

In the **easy subproblem C1**, $1 \leq n \leq 10$.

In the **hard subproblem C2**, $1 \leq n \leq 10^5$.

Output specification

For each test case, output a single line with a single integer: $x \bmod (10^9 + 9)$, where x is the number of ways to sort the given sequence using as few swaps as possible.

Example

input

```
3

3
2 3 1

4
2 1 4 3

2
1 2
```

output

```
3
2
1
```

In the first test case, we can sort the permutation in two swaps. We can make the first swap arbitrarily; for each of them, there's exactly one optimal second swap. For example, one of the three shortest solutions is "swap p_1 with p_2 and then swap p_1 with p_3".

In the second test case, the optimal solution involves swapping p_1 with p_2 and swapping p_3 with p_4. We can do these two swaps in either order.

The third sequence is already sorted. The optimal number of swaps is 0, and thus the only optimal solution is an empty sequence of swaps.

Problem D: Dumb clicking

The Interactive Playable Shovelware Company is about to release a new videogame! You have been chosen to be the tester. Okay, so maybe the developers have just copied the same levels over and over, and maybe the graphics department still hasn't added any images, but so what? Surely the buyers won't mind! It will definitely be a big hit!

In this problem, you are given an in-development prototype of a simple clicking game. The game consists of several levels. As you go through the levels, the game will produce a log of your actions. Finish the game and submit the log as your proof.

Since the game is a prototype, it may lack some features here and there... For example, it won't actually tell you if you do something wrong, so be careful.

JavaScript application

The game is a browser-based JavaScript application. You can either open it from the online problem statement, or open the file `d/easy.html` or `d/hard.html` from the downloadable archive. You'll need a reasonably modern browser to play. Old versions of Internet Explorer probably won't work.

Input specification

There is no input.

Output specification

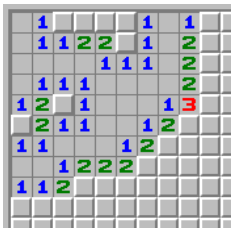
Submit a text file containing the action log of your game. An action log is complete if the last line contains the word "**done**".

Problem E: Evil minesweeper

Kamila loves to torment her friends. Her new diabolic idea is an evil implementation of a well-known single-player game: Minesweeper.

In Minesweeper the player is shown an $r \times c$ grid of cells and told a positive integer m: the number of hidden mines. Each cell contains either a mine or a number. The number written in a cell is the count of mines in adjacent cells (horizontally, vertically, and also diagonally).

Initially, the content of each cell is hidden. The player plays the game by taking actions. In each action the player either asks for a cell to be revealed, or marks a cell that is certain to contain a mine. If the player reveals a cell with a zero, all adjacent cells are revealed automatically – because we know they cannot contain a mine. If any of those cells contain zeros as well, the revealing continues with the neighbors of those cells, and so on. The figure below shows one possible final result of revealing a zero in the top left corner. (In the figure, cells with zeros are the flat cells without a number.)



The player wins the game by locating all m mines. In Kamila's version of the game **each incorrect action immediately loses the game**. That is, there are **two ways to lose the game**: you lose if you reveal a cell with a mine, and you also lose if you mark a cell without a mine as a cell with a mine. Once you take an incorrect action, the game will reveal the locations of all mines as a proof that you made a mistake.

Where's the evil, you ask? Well, this is how Kamila envisions the gameplay:

1. The player chooses the dimensions of the grid and the number of mines.
2. The player reveals some cell.
3. The game pretends to be nice and friendly: it reports that the player **revealed a zero**. Then, the game does the automatic additional reveals described above.
4. What the game actually does is that it carefully choses the revealed area and the digits shown on its boundary in such a way that **the player will be unable to take a second action**. More precisely, regardless of what second action the player takes, the game will always claim that they lost, and show a placement of mines that proves it.

Problem specification

You are given the dimensions r, c and the number m of mines. Rows of the grid are numbered 0 through $r-1$ from top to bottom. Columns are numbered 0 through $c-1$ from the left to the right.

You are also given the coordinates (a_r, a_c) of the first revealed cell.

Check whether this is a situation in which Kamila can carry out her evil plan. If yes, compute and return one possible proof. (This is described in more detail below.)

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line containing five integers: r, c, m, a_r, and a_c.

In the **easy subproblem E1** you may assume the following:

- $10 \leq r, c \leq 20$
- $rc/10 \leq m \leq 7rc/10$ (i.e., mines cover between 10 and 70 percent of the grid)
- $2 \leq a_r < r - 2$ and $2 \leq a_c < c - 2$ (i.e., the first revealed cell is at least 2 from each side)

In the **hard subproblem E2** you may assume the following:

- $1 \leq r, c \leq 20$
- $2 \leq rc$
- $1 \leq m < rc$
- $0 \leq a_r < r$ and $0 \leq a_c < c$

Output specification

If the given values are such that Kamila is unable to carry out her evil plan, output a single line with the string `IMPOSSIBLE`. Otherwise, output the following things:

- First, a line containing the number x of different configurations of mines you want to provide.
- Then, x configurations of mines, each with exactly m mines.

Each configuration of mines is a $r \times c$ matrix of characters, with '`.`' (period) represening a cell without a mine and '`*`' (asterisk) representing a cell with a mine.

The output must have the following properties:

- $x \leq rc$
- For each of the x configurations of mines the cell (a_r, a_c) must reveal a zero when clicked.
- For each of the x configurations of mines the set of cells revealed by revealing (a_r, a_c) in step 3 of the game must be the same, and the numbers shown in those cells must also be the same.
- For each cell that is not revealed in step 3 of the game, there must be at least one of these x configurations that **does** and at least one other configuration that **does not** contain a mine in that cell.

Finding a solution with the smallest possible x is **not** necessary. Any valid solution will be accepted.

The output should not contain any blank lines.

Example

input

```
3

1 2 1 0 0

1 3 1 0 0

2 7 2 0 2
```

output

```
IMPOSSIBLE
IMPOSSIBLE
3
.....**
.......
......*
.....*.
.....*.
......*
```

The first example is too small. When the player reveals the cell $(0, 0)$, we cannot show them a zero because then we won't be able to place the mine anywhere.

The second example is still too small. When the player reveals the cell $(0, 0)$, we can tell them that $(0, 0)$ contains a zero and $(0, 1)$ contains a one. However, we have no chance to be evil. Instead, the player will correctly determine that the only unrevealed cell $(0, 2)$ contains the mine.

In the third example our evil game will reveal the board as follows:

```
00001??
00001??
```

Above, question marks represent unrevealed cells. In this situation the player is unable to make a second move: each unrevealed cell may contain a mine, but at the same time each unrevealed cell may be empty. Regardless of what action the player takes, we can always show them one of the configurations shown in example output as the proof that they lost.

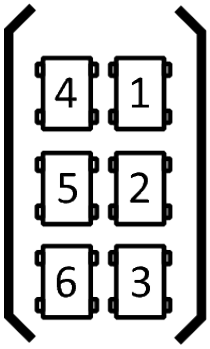
Problem F: Ferries

One day a group of friends jumped into their $2n$ cars and they all went for a road trip into the lake district. The cars were numbered 1 through $2n$. Initially, they were driving in this order.

During the road trip the convoy encountered multiple ferries, one after another. Each ferry in the district was exactly large enough to accommodate the $2n$ cars, and there were no other cars around at any time. Whenever the cars used a ferry, their order got mixed up. This is due to the way the ferries are loaded and unloaded.

A ferry is a long and narrow boat. There is just enough room for two columns of cars, each containing n cars. When loading the ferry, the first n cars form the first column and the last n cars form the second column. When unloading the ferry, all cars are allowed to drive away at the same time, so they have to zip in order to form a single convoy. The first car in the new order will always be the first car from the **right** column, the second car will be the first car from the **left** column, the third car will be the second car from the right column, and so on, alternating right and left.

There are two types of ferries. On type-L ferries the crew instructs the cars to fill the **L**eft column first while on type-R ferries cars fill the **R**ight column first. Note that the ferry type does not influence how the cars leave the ferry. The first car to leave is always the first car in the right column.



Above: Cars 1 through 6, in this order, just boarded a type-R ferry (from the bottom of the figure). The cars will leave this ferry in the following order: 1, 4, 2, 5, 3, 6.

Problem statement

The trip is now over. Your car was car x, which means that at the beginning your car was the x-th car from the beginning. You remember a sequence $a_1, \ldots, a_k$ of positive integers. The number a_i means that at some point in time your car was the a_i-th car from the beginning of the convoy. The numbers in your sequence are in chronological order: your car was a_i-th before it was a_{i+1}-th, for all i.

Compute the smallest possible number f of ferries you encountered during the trip. Produce a proof: one possible sequence of exactly f Ls and Rs such that it is possible that the f ferries you encountered were type-L and type-R ferries in the given order.

Input specification

The first line of the input file contains an integer $t = 100$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the integers n, x, and k, where $1 \leq x \leq 2n$. The second line contains the integers $a_1, \ldots, a_k$.

In the **easy subproblem F1**, $1 \leq n \leq 10^5$ and $1 \leq k \leq 20$.

In the **hard subproblem F2**, $1 \leq n \leq 10^{18}$ and $1 \leq k \leq 10^4$.

Output specification

For each test case, output a single line with the number f, a colon, and one possible shortest string of Ls and Rs. Note that there are no spaces in the output.

Example

input

```
3

10 1 3
1 1 1

10 1 2
2 4

3 2 4
5 3 1 2
```

output

```
0:
2:LL
7:RRLRRLL
```

In the first example there were no ferries, the car simply remained in the first place.

In the second example the first type-L ferry moves the car to position 2 and the second such ferry moves it to position 4.

In the third example `RR` gets us from 2 to 5, `L` from 5 to 3, `RRL` from 3 to 1, and `L` from 1 to 2.

Problem G: Greatest number

Little Peter loves playing at the junkyard. Today he found and brought home a broken scientific calculator. He did his best repairing it, but the calculator is still not in its best shape. Here is what still works:

- There are some expressions stored in the memory of the calculator. Peter can load any of those onto the display.
- The arrow keys and the delete key still work. Using those, Peter can delete any subset of characters from the expression shown on the display.
- The equals sign works. Peter can press it to evaluate the expression shown on the display.

Peter likes big numbers. For each of the expressions in his calculator find the largest number he can produce after pressing the equals sign.

Problem description

You are given a string S that contains a valid arithmetic expression. Remove any (possibly none, but not all) characters from S so that the resulting string T is again a valid arithmetic expression and the value of T is as large as possible.

Valid expressions are defined as follows:

- Any nonnegative integer is a valid expression, as long as it doesn't have unnecessary leading zeros.
- If `a` is a valid expression then `(a)` is also a valid expression.
- If `a` is a valid expression then `-a` and `+a` are also valid expressions.
- If `a` and `b` are valid expressions then `a-b`, `a+b`, and `a*b` are also valid expressions.
- Nothing else is a valid expression.

During the evaluation, the standard operator precedence applies:

- Parentheses are evaluated first, overriding other precedence rules.
- Next, the (right associative) unary `+` and `-` are applied.
- Afterward, multiplication `*` is applied.
- And finally, addition `+` and subtraction `-` are applied from left to right.

There are no other unary or binary operations except for the ones mentioned above. In particular, there is no division and no exponentiation.

Note that the definition allows multiple unary pluses and minuses to follow each other. For example, `1*(2+3*-+-4)` is a valid expression that is evaluated as follows: $-+-4$ is 4, 3×4 is 12, $2+12$ is 14, and 1×14 is 14.

Also note that unnecessary leading zeros are **not** allowed. For example, `4-007` is not a valid expression.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Every test case contains a single line containing a valid arithmetic expression S.

In the **easy subproblem G1**, $t = 100$ and the length of each expression is between 1 and 16, inclusive.

In the **hard subproblem G2**, $t = 1000$ and the length of each expression is between 1 and 1000, inclusive.

Output specification

For each test case, output one line with a valid arithmetic expression T such that T is a subsequence of S and such that T has the maximum possible value. If there are multiple solutions, print any of them.

Example

input	output
3 -10 0*0*1 0+(1*2)	10 1 0+(12)

Problem H: Heavy snowfall

Your town has been hit by an enormous amount of snow this winter. The town council has decided to buy a snow plow to clear the roads.

Problem specification

The town has n intersections, numbered 1 through n. These are all connected together by a network of $n-1$ bidirectional roads. (In other words, the road network is a tree.) All roads have the same length.

At the beginning of winter, there was no snow anywhere. During the winter the snow plow made q journeys. For each journey, we know the path the plow travelled and whether it was snowing during that journey.

During each journey the plow travelled along the only simple path from the origin to the destination of the journey. If it did snow, it always snowed **during the entire journey**, and it always snowed **in the entire town**, not just on the plow's path. All snowfall during the entire winter happened during some of the plow's journeys.

Whenever the plow travels through a road, two things happen in order:

1. All snow is removed from that road.
2. If it's snowing, every road in the town (including the one just cleaned) gains 1 cm of snow.

The destination of one journey might not be equal to the starting point of the next one. In that case, the snow plow isn't active in the meantime – it moves to the next starting point without changing the amount of snow on the roads.

For each journey, compute the total amount of snow the plow removed during that journey.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case starts with a line containing the integers n and q ($1 \le n \le 10\,000\,000; 1 \le q \le 25\,000\,000$). The next $n-1$ lines contain integers p_i, q_i ($1 \le p_i, q_i \le n$) meaning that intersections p_i and q_i are connected by a road. The i-th of the following q lines contains integers a_i, b_i, c_i ($1 \le a_i, b_i \le n; 0 \le c_i \le 1$) describing the i-th journey of the snow plow: The journey started at intersection a_i and ended at intersection b_i. We have $c_i = 1$ if it snowed during this journey, and $c_i = 0$ if it did not.

In the **easy subproblem H1** it is guaranteed that there is a road between intersections i and $i+1$ for every $1 \le i < n$. (The road network is a line.)

Because the size of `h1.in` is about 800 MB and the size of `h2.in` is about 1.2 GB, you cannot download them directly. Instead, we have provided small Python 2 programs `h1gen.py` and `h2gen.py` that will generate `h1.in` and `h2.in` when executed. Each generator should take under 10 minutes to run on average hardware. We recommend running them early – for example, starting them as you start working on your solution for this problem.

Output specification

For each test case: For $i = 1, \ldots, q$, let y_i be the total amount of snow removed during journey i, and let z_i be $i \cdot y_i$. Output one line with a single number: $(z_1 + \cdots + z_q) \bmod (10^9 + 7)$.

Example

input	output
1 5 3 1 2 2 3 3 4 3 5 1 4 1 3 2 1 4 5 0	25

During the journey from 1 to 4, the plow removed $y_1 = 0 + 1 + 2 = 3$ *cm of snow. Afterwards, the roads have* $(3, 2, 1, 3)$ *cm (in the order from the input). During the second journey, we remove* $y_2 = 2$ *cm of snow from the* $2 - 3$ *road, and then every road gains another* 1 *cm, with final amounts* $(4, 1, 2, 4)$*. In the third journey, we remove* $y_3 = 2 + 4$ *cm of snow and no new snow falls, so the final state is* $(4, 1, 0, 0)$*.*

The output is $(1 \cdot y_1 + 2 \cdot y_2 + 3 \cdot y_3) = 3 + 4 + 18 = 25$*.*

Problem I: Intelligence report

TOP SECRET

Secret assignment number 32250:

Agent,

we have obtained several files from a computer of the person of interest. We believe they may contain stolen passwords for our military control systems. Find out what these files are and recover the passwords from them, so that we may verify whether our passwords were stolen.

BURN AFTER READING

Problem specification

You are given a file. Find out what this file is and recover passwords from it.

It might be easier to solve this problem using Linux. If you don't have Linux installed, you might want to download and run an arbitrary live distribution. Or you may choose to do something completely different. Your choice of tools is completely up to you. Whatever works.

Output specification

The file for the **easy subproblem I1** contains 10 passwords for test cases numbered from 0 to 9, each one is a 32-character string containing only alphanumeric characters. Submit one file with one password per line. Passwords should be in the correct order.

The file for the **hard subproblem I2** contains a single password. The password consists of 38 alphanumeric characters. Please make sure that your submission contains a single 38-character string, without any whitespace between the characters of the string.

Problem J: Jumping queues

You probably know this situation really well. You're in a supermarket, standing in a long queue in front of a cashier. Around you, there are several other long queues and it always seems like they're moving much faster than yours. Sure, you can always leave the current queue and move to the end of another one – but after you do so, it still seems like the other queues are moving much faster than your new one. The queues are moving towards the cashiers, but they also grow in time. Still, the longer a queue already is, the slower it tends to grow.

Sometimes, you note that something changed. It may be the speed at which the queues are moving. It may be the rate at which the queues are growing. It may be that a new cashier has just been opened and a queue of a given length has instantly spawned in front of that cashier. (As you can see, this is a very realistic model.)

In order to determine whether jumping queues is worth the effort, you'd like to know the answers to several questions of the following form: "Suppose a person just arrived to the checkouts. They are going to choose a queue and wait in that queue until they reach its cashier. Assuming that the queues will maintain their current speeds, what is the shortest time in which the person can reach one of the cashiers?"

Problem specification

There are n points of sale (POS) numbered 1 through n. Each of them is either open or closed. Initially, POS numbered 1 through m are open. If the POS number i is open, there's a cashier at that POS and a queue in front of the POS. The speed of the queue at POS i is v_i. (More precisely, v_i is the speed at which you're moving towards the cashier if you're standing in that queue.) This queue also has a growth parameter g_i. The length l_i of the i-th queue is a function of the time c: if the POS number i is open, the length $l_i(c)$ grows continuously. The rate of growth is given by the following formula:

$$\frac{\mathrm{d}l_i}{\mathrm{d}c} = \frac{g_i}{l_i}\,.$$

Let $\tau_i(c)$ be the time necessary to reach the cashier when you're standing at the end of queue i at time c, provided that the speed v_i of this queue will remain constant.

You should process q queries of three types:

- "`O` c i v g l": POS number i has opened at time c; there is a new queue of length $l_i(c) = l$ with parameters $v_i = v, g_i = g$ in front of it
- "`U` c i v g": at time c, the parameters of queue i change to v and g
- "`Q` c": find an open queue i for which $\tau_i(c)$ (the waiting time if you joined it at time c) is shortest, and answer that waiting time

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains three space-separated integers n, m and q. Each of the following m lines contains three space-separated *real* numbers v, g and l: the speed, the growth parameter, and the length of queue i at time $c = 0$. Then, q lines follow; each of them contains the description of one query in the format above; c, v, g and l are *real* numbers given with exactly two digits after the decimal point.

In both subproblems, $1 \le m \le n \le 2\,000\,000$, $1 \le q \le 5\,000\,000$, $1 \le c, v, g, l \le 10\,000\,000$. In addition, the time c in successive queries (even queries of different types) will be increasing; in type `U` queries, the speed of each queue will be non-decreasing and the growth parameter non-increasing.

In the **easy subproblem J1**, there will be no queries of types `O` and `U`.
It's guaranteed that the number of queries of type `Q` in each test case will be a multiple of 5 000.

This problem has large input files. You cannot download them directly. Instead, we have provided small Python 2 programs `j1gen.py` and `j2gen.py` that will generate `j1.in` and `j2.in` when executed. Each generator should take under 10 minutes to run on average hardware. We recommend running them early – for example, starting them as you start working on your solution for this problem.

Output specification

For each test case, let q' be the number of queries of type `Q`. Print $q'/5000$ numbers on separate lines: the sum of the first five thousand answers, then the sum of the next five thousand answers, etc. Answers with absolute or relative error up to 10^{-6} will be accepted.

Example

input

```
2

5 2 3
1.00 3.00 50.00
2.00 14.00 15.00
Q 100.00
Q 2000.00
Q 4000.00

5 1 4
8.00 8.00 8.00
O 1.50 4 2.70 3.00 5.44
Q 2.55
U 7.20 1 27.00 2.61
Q 10.00
```

output

```
308.8472630
1.7952699
```

In the above examples the number of queries of type `Q` is not a multiple of 5000. The first number in the example output is the sum of all three answers to the queries in the first example test case, the second number is the sum of both answers in the second example test case.

The answers to the five individual type-`Q` queries in the example are 27.5000000 + 118.5590570 + 162.7882060 in the first test case and 1.2796484 + 0.5156215 in the second test case.

*Note that the second test case **does not satisfy** the additional constraints for subproblem J1.*

Problem K: Kill switch

A kill switch is a mechanism used to shut off a device in an emergency situation.

Jeremy was hired as a contractor by a shady software company. After he finished his work the company pointed to a loophole in the contract and refused to pay him anything for his work. Little do they know that Jeremy suspected foul play and thus he hid a kill switch in his code.

Problem specification

You are given the implementation of a function that *pretends* to sort an array of 32-bit unsigned integers into a non-decreasing order. Find the shortest input the function *fails* to sort.

Input specification

In each subproblem there are two input files: one is a C++ implementation and the other a Python implementation of the same function.

(You may assume that if the answer is n, the two programs behave the same way at least on all valid inputs of size up to $n+47$. Note that huge inputs may cause integer overflows in the C++ implementation. Such inputs are not a part of this problem and they can be safely ignored.)

Each subproblem should be solved separately.

Output specification

Your output file should contain two lines. The first line should contain a nonnegative integer n: the smallest possible length of an array that is not sorted correctly. The second line should contain one possible initial content of the array: the sequence $A[0], \ldots, A[n-1]$. These values must satisfy $0 \leq A[i] < 2^{32}$.

Example

input

```
void example_sort(vector<unsigned> &A) {
    int N = A.size();
    if (N >= 2 && A[0] > A[1])
        swap( A[0], A[1] );
    if (N >= 3 && A[0] > A[2])
        swap( A[0], A[2] );
    if (N >= 3 && A[0] > A[1])
        swap( A[0], A[1] );
}
```

```
def example_sort(A):
    N = len(A)
    if N >= 2 and A[0] > A[1]:
        A[0], A[1] = A[1], A[0]
    if N >= 3 and A[0] > A[2]:
        A[0], A[2] = A[2], A[0]
    if N >= 3 and A[0] > A[1]:
        A[0], A[1] = A[1], A[0]
```

output

```
3
42 47 1
```

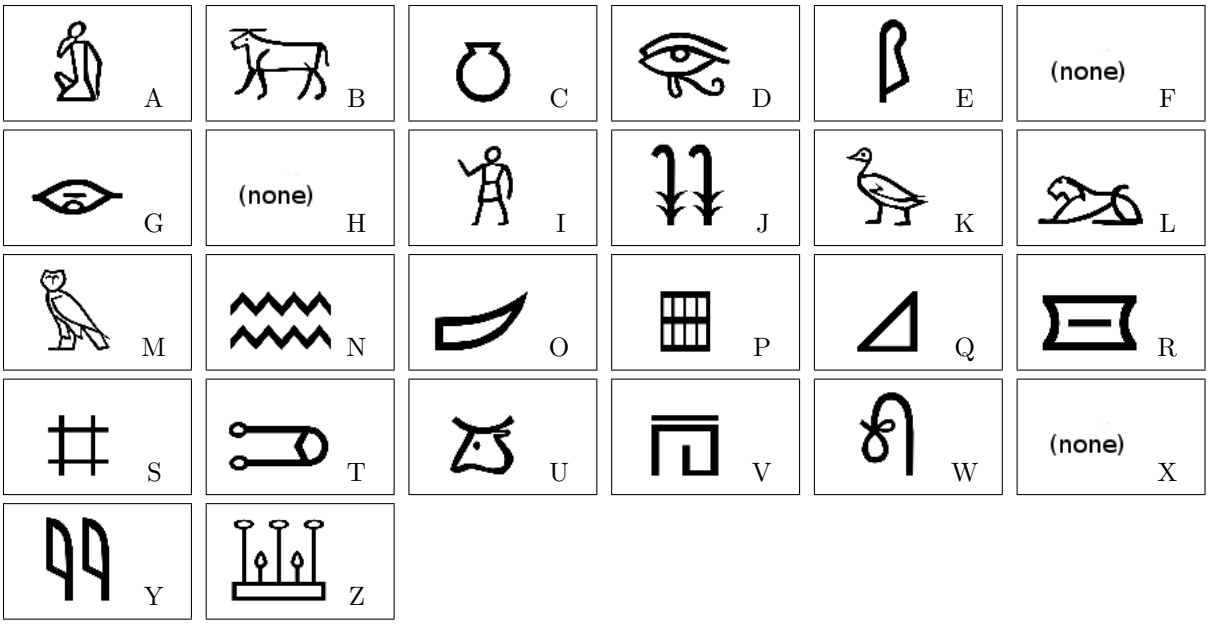
For the input $(42, 47, 1)$ *the provided function will return* $(1, 47, 42)$ *which is not a sorted array.*

Problem L: Luxor Catch-ya!

Egyptian pharaohs were quite wealthy people. Moreover, they brought a lot of this wealth along with them to their afterlife. It is therefore not a big surprise that tomb raiding was a very good way of making a fortune, that is, assuming you lived through the experience. In order to regulate this industry, pharaohs had many traps built into their tombs. One of their latest inventions was an ingenious device called "catch-ya" – the device shows you some random hieroglyphs and you are required to decipher them; if you decipher the symbols wrongly, the device would catch you in a deadly trap. Catch-ya devices were quite successful at the time, mainly because only a few people could read and write hieroglyphs and these people were usually at high state positions anyway.

Problem specification

Because of some unnamed circumstances involving a lot of gambling, you found yourself in a rather intricate situation marked by the fact that you owe a not-really-small amount of money to some not-really-friendly people. In order to resolve this problem you decided to raid a tomb of a wealthy Egyptian pharaoh. The planning paid off and now you are deep inside the pyramid, still alive, standing in front of the catch-ya device. This is the last thing that stands between you and lots of gold. So, the only task left is to decode the provided catch-ya prompts. Fortunately, you are a clever thief and therefore you got a hold of a reference mapping between hieroglyphs and letters you need to enter on the keyboard in front of you.



(Source: the Meroitic Hieroglyphics font by Reinhold Kainhofer.)

You can find both PNG and textual representation of the glyphs in the directory `l/alphabet`. Note that Egyptians did not have sounds for F, H and X so there is no glyph for them.

Input specification

The input files can be found in directories `l/l1` and `l/l2`. Each subdirectory contains exactly $t = 800$ test cases, the i-th of which is stored in two files i`.in` and i`.png`.

Each test case file i`.in` represents a greyscale image with dimensions 600 x 70 pixels – the catch-ya prompt. The file consists of 70 lines, each line containing 600 integers between 0 and 255. i`.png` contains the same image as i`.in`, but in the PNG format.

Each subdirectory also has a file named `sample.out`, which contains the correct decoding for the first 200 test cases of the subproblem.

Output specification

Output a single line for each test case. The line should contain a 6-character lower-case string with the decoded catch-ya. Note that your output should not contain characters "f", "h", or "x".

For the **easy subproblem L1**, your output must be completely correct.

For the **hard subproblem L2**, your output must have the correct format: for each test case you must output one string of 6 allowed lowercase letters. At least 98% of those strings must be correct. In other words, your output may have at most 16 wrong lines.

Example (easy subproblem)

input

```
(799 more catch-yas)
```

output

```
mweaul
(799 more lines)
```


Problem M: Mana troubles

Hello and welcome to a brief excursion into the wondrous world of the trading card game Magic the Gathering. In this episode we will look at *mana* and at *lands*: the cards that produce it.

Problem overview

Mana is the "currency" used to cast spells in the game. The canonical way to produce mana is by having some *lands* on the battlefield. In each turn of the game, each of those lands can be *tapped* once to produce some mana. (At any moment, each land that is on the battlefield is either *untapped*, meaning it can still be used this turn, or *tapped*, meaning it has already been used this turn.)

As the input to your program you will be given two sets of land cards: **untapped** lands that are already on the battlefield and lands that are still in your deck.

Your task is to tap the lands in such a way that you 1. don't die, and 2. produce at least the specified amount of mana of each type.

Below, we will explain in detail how everything works. Note that all the cards shown below will be given to you in a machine-readable format.

Mana

There are five colors of mana: white, blue, black, red, and green. Their symbols are shown below, in this order.

In text, we use the letters W, U, B, R, and G to denote the five colors of mana. Note that U (as in "blUe") is used for the blue color – this is because both "blue" and "black" start with the same letter.

In addition to the five mana colors, there is also colorless mana. In pictures its associated symbol is a grey circle that contains a diamond, as shown below. We use the letter C to represent colorless mana.

All mana produced by lands is of one of the six types mentioned above.

In mana *requirements* there is a seventh possibility: *generic* mana. A requirement of generic mana can be paid using mana of any type. For example, if a spell requires three generic mana, we can satisfy this requirement by producing 2 blue and 1 colorless mana.

Generic mana requirements will be described using digits 1-9. For example, we can write "2WWU" to denote that a spell requires 2 generic, 2 white, and 1 blue mana. We can also write "1C" to denote that a spell requires 1 generic mana and 1 mana that has to be colorless.

Note that we interpret each character of a mana cost separately. For example, "99" is 9+9 = 18 generic mana, not 99 generic mana.

Land types used in this problem

In this section we will describe some of the multitude of land cards that are played in Magic.

Basic land types and basic lands

Each color of mana has an associated *basic land type*. These are Plains (W), Island (U), Swamp (B), Mountain (R), and Forest (G).

The simplest type of a land is a *basic land*. Its name is the same as the corresponding basic land type. Whenever tapped, a basic land produces one mana of the corresponding color. For example, the beautiful

Forest land shown below on the left produces a single G mana when tapped. If you have three untapped Forests on the battlefield, you can tap all three of them to produce GGG (i.e., three green mana).

Multicolored lands

The other two pictures above show lands that are capable of producing multiple colors. When you tap the Mystic Monastery, you get to choose whether it should produce U, R, or W. Similarly, you can tap the Bayou for either B or G.

There is one additional technical difference between these two example lands. Mystic Monastery does not have any basic land type: it is a non-basic land. Bayou has *two* basic land types: it is both a Swamp and a Forest. This distinction will become important soon.

Fetch lands

The Wooded Foothills land shown above is an example of a *fetch land*. This land doesn't produce any mana. Instead, when you tap it, you lose one life and something good happens. The loss of life will be addressed later. The good thing that happens is that you get to search your deck of cards (formally, the *library*: the cards you haven't drawn yet) for any single land card *that has one of the listed basic land types*. You then throw away the fetch land and you replace it by the land you have selected from your library.

For example, the Wooded Foothills can be used to fetch a basic Mountain or a basic Forest (if you have some of them in your library). The Bayou also counts as a Forest. This means that you can use your Wooded Foothills to fetch a Bayou from your library.

Even though the Mystic Monastery can produce R, it is *not* a Mountain. Hence, you *cannot* use the Wooded Foothills to fetch a Mystic Monastery.

The fetch lands themselves do not have any basic land types. Thus, you cannot use a fetch land to fetch another fetch land.

Shock lands

The Steam Vents, shown in the left picture below, are an example of a shock land. If you already have an untapped Steam Vents on the battlefield, it's great for you: you can tap it for either U or R.

However, by default this land comes into play tapped. This means that you cannot use it on the turn when it enters the battlefield. In order to be able to use it right away, you have to pay 2 life. (This is called a "shock" because of a spell of that name that deals 2 damage.) If you pay the 2 life, you'll get the land untapped and you can immediately tap it for one of the two colors it can produce.

In this problem, the shock will matter when using a fetch land. You may note on the card that the Steam Vents count as both an Island and a Mountain. Hence, they can be fetched by an appropriate fetch land such as our old friend the Wooded Foothills. However, if you decide to use the Wooded Foothills to fetch a Steam Vents and then you want to use the Steam Vents in the same turn, you have to pay a total of 3 life: 1 for the fetch land and another 2 to get the Steam Vents untapped.

Pain lands

The Battlefield Forge, shown above, is an example of a *pain land*. When you tap it, you get to choose one of three options: either it produces 1 colorless mana, or you lose 1 life and it produces R, or you lose 1 life and it produces W.

Bounce lands

The last card shown above is Azorius Chancery, an example of a *bounce land*. Bounce lands (also known as Karoos) have an effect that happens when they enter the battlefield. In this problem that effect does not matter. You can simply treat a bounce land as an especially cool land that produces *two mana* when tapped! For example, whenever you tap an Azorius Chancery, you will get *both* a W and a U.

Note that pain lands and bounce lands do not have any basic land types. Therefore, they cannot be fetched by fetch lands.

City of Brass and Gemstone Mine

Up until this point each item in our list was a *category* of lands. In this problem we will have multiple cards from each of those categories – for example, different fetch lands, each with its own two basic land types.

From now on, each card we mention will be a single specific card.

The two different five-color lands we are going to use in this problem are City of Brass and Gemstone Mine (both shown below). For the purpose of this problem we will assume that each Gemstone Mine does have a mining counter. In human words: if you have a Gemstone Mine, you can tap it to produce a single mana of any color for free, and if you have a City of Brass, you can tap it to produce a single mana of any color but you have to pay 1 life for this action. (The mana produced by either of these lands must be a mana of one of the five colors. These lands cannot produce colorless mana.)

Urza's Tron

The last three lands we are going to use produce colorless mana. Individually, each of them produces C (i.e., one colorless mana). However, if you have at least one copy of each of them on the battlefield, they produce more! In that situation, each copy of Urza's Mine and each copy of Urza's Power Plant produces CC when tapped, and each copy of Urza's Tower produces CCC.

All five lands we just mentioned are non-basic lands that cannot be fetched.

Life

At the beginning of the game you have 20 life. Whenever your life total drops to 0 or below 0, you lose the game. Each test case will specify your current life total.

Problem specification

You will be given a situation during your turn. You have some amount of life left. You have some lands on the battlefield and some lands in your library. All lands on the battlefield are currently untapped. If you have some fetch lands on the battlefield, you can use them to fetch some appropriate cards from your library to the battlefield.

You are going to cast a large spell. You are given the mana requirements for this spell. Your program should determine whether it is possible to tap the lands in such a way that you don't die and the mana produced will satisfy the requirements.

Note that you are allowed to produce more mana than you need.

For example, if your goal is to produce "2WWU" and you tap your lands in such a way that they produce "CWWGUU", you satisfied the requirements: you have the two white and one blue, and you can use any two of the three remaining mana to pay the two generic mana.

For another example, if your goal is to produce "RG" and you produce "CCCCCCCGG", you failed: even though you produced 9 mana, none of it is red.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case starts with a line containing a number x: the number of land cards on the battlefield. Then, x lines follow, each containing the name of one card.

The test case continues with a line containing a number y: the number of land cards in your library. Then, y lines follow, each containing the name of one card.

Finally, there are two lines. The first of them contains your current life total (between 1 and 20, inclusive), and the second contains a nonempty string over [1-9CWURBG]: the specification of amounts and types of mana you should produce.

Capitalization and spaces in card names are used consistently everywhere. There are no extra spaces before or after a land name.

You may assume that **the total number of lands** (on the battlefield and in the library together) **is at most 34**.

In the **easy subproblem M1**, you may additionally assume that **there are no fetch lands on the battlefield** and no lands in your library.

Output specification

For each test case, output a single line with the string "YES" if the lands can be tapped accordingly, or "NO" if they cannot.

Be very very glad you do not have to produce a complete log of actions if the answer is YES. And while we are at it, be glad that we left out filter lands, tainted lands, and many other land types that would make this problem computationally much harder. By the way, did you know that Magic the Gathering is Turing-complete? ;)

Card specification

In order to make the implementation more pleasant, we will provide you with a machine-readable file `m_lands.txt` containing the descriptions of all the land cards that can appear in the inputs.

Here is a sample of the file, containing all the cards from the above examples. The other cards only differ in their names and colors of mana they produce.

```
Forest;Forest;G;
Mystic Monastery;;U|R|W;
Bayou;Swamp,Forest;B|G;
Wooded Foothills;;-fetchMountain|-fetchForest;
Steam Vents;Island,Mountain;U|R;--shock
Battlefield Forge;;1|-R|-W;
Azorius Chancery;;WU;
City of Brass;;-W|-U|-R|-B|-G;
Gemstone Mine;;W|U|R|B|G;
Urza's Mine;;1;tron 2
Urza's Power Plant;;1;tron 2
Urza's Tower;;1;tron 3
```

(Notes: This is a semicolon-separated list. The second field lists the basic land types of each land. The third field contains the alternatives you get to choose from, with the symbol - denoting a loss of 1 life. The last, fourth field is used as an extra note whenever it is needed.)

Notes for MtG players

If you do play Magic the Gathering, here are a few quick notes. In this problem there are no cards in hand, and therefore no ability to play an additional land from your hand to the battlefield. There can be more than four copies of nonbasic lands. For example, a test case with 34 Mystic Monasteries is perfectly valid. The rest of the problem should be faithful to the rules of the actual game.

Examples

input

```
4

3
Forest
Forest
Plains
0
20
GGW

3
Forest
Forest
Plains
0
20
2

2
Wooded Foothills
Forest
1
Steam Vents
7
RG

2
Wooded Foothills
Forest
1
Steam Vents
3
RG
```

output

```
YES
YES
YES
NO
```

1. Tap all three lands and you have exactly the mana you need.
2. The requirement is to produce 2 generic mana. Tap any two lands and use the mana they produced.
3. Tap the forest for G. Pay 1 life and use the Wooded Foothills to fetch the Steam Vents. Pay 2 life to get Steam Vents into play untapped. Tap Steam Vents for R. You are still alive with 4 lives left.
4. Here we have too few lives to survive the above sequence of actions.

input

```
4

2
Wooded Foothills
Forest
2
Steam Vents
Mountain
3
RG

4
Steam Vents
Steam Vents
Battlefield Forge
Azorius Chancery
0
1
1RWUU

4
Steam Vents
Steam Vents
Battlefield Forge
Azorius Chancery
0
1
RRWUU

1
Wooded Foothills
4
Forest
Battlefield Forge
City of Brass
Gemstone Mine
20
R
```

output

```
YES
YES
NO
NO
```

1. Instead of the Steam Vents we can now fetch the basic Mountain and tap it for R. As we have only paid 1 life for the fetch, we are still alive with 2 lives left.
2. As the Steam Vents are now on the battlefield (and therefore untapped), we can tap the two of them for R and U without losing life. We can also tap the Azorius Chancery for WU and the Battlefield Forge for C (a colorless mana) and use that C as the generic mana we need.
3. Tapping Battlefield Forge for R costs 1 life and that would kill us.
4. We can fetch the Forest but it doesn't produce R. Even though the other three lands can produce R, they are neither Forests nor Mountains and therefore we cannot fetch them.

input

```
4

1
Wooded Foothills
2
Mountain
Mountain
20
RR

2
Wooded Foothills
Wooded Foothills
1
Mountain
20
RR

4
Urza's Power Plant
Urza's Tower
Urza's Mine
Urza's Tower
0
20
19

3
Tundra
Underground Sea
Volcanic Island
0
20
UUB
```

output

```
NO
NO
YES
YES
```

1. Each fetch land can only be used once. In this test case, you can only fetch one of the two Mountains and that is not enough.
2. You can use one of the two fetch lands to fetch the Mountain from your library to the battlefield. Afterwards, there are no Mountains left in your library, so the second fetch land has nothing to fetch.
3. Behold the power of Tron! We were supposed to produce 1+9 = 10 generic mana. And indeed, if we tap all four lands we have, we get 2+3+2+3 = 10 colorless mana.
4. Tundra, Underground Sea, and Volcanic Island are dual lands of the same type as Bayou, but they produce W|U, U|B, and U|R, respectively. In order to tap these lands for UUB we have to tap Underground Sea for black mana and the other two lands for blue mana.