IPSC 2016

problems and sample solutions

Problem S: Subsequence

Unfortunately, this task has no story because my dog ate some of my notes. Instead, you'll get to help me determine whether I can salvage something out of the scraps left after the dog ran away.

Problem specification

You are given two strings A and B. Find whether B is a subsequence of A.

The subsequence does not need to be contiguous. In other words, your task is to find whether it is possible to change A into B by erasing some (possibly none) of its characters.

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 string A. The second line contains the string B. Both strings consist only of uppercase English letters.

In the **easy subproblem S1** the string A has length at most 100 and the string B has length 3.

In the **hard subproblem S2** the string A has length at most $100,000$ and the string B has length at most $10,000$. Note that you cannot download the input for subproblem S2 directly. Instead, we have provided a small Python 2 program that will generate the file `s2.in` when executed.

Output specification

For each test case, output a single line with a single word: "YES" or "NO" (quotes for clarity).

Note

In the real contest, some large input files may be provided in the same way as the input `s2.in` in this practice problem. Please make sure you are able to generate it.

Example

input

```
3

DAOBCCCCGS
DOG

ABCDEF
ACEG

CAT
CTA
```

output

```
YES
NO
NO
```

Task authors

Problemsetter:	Vlado 'usamec' Boža
Task preparation:	Vlado 'usamec' Boža
Quality assurance:	Peter 'PPershing' Perešíni

Solution

The easy subproblem can be solved by almost any reasonable algorithm. For example, we can try all possible triples of indices into A and check whether those letters form B. This can easily be done using three nested cycles.

To solve the hard subproblem, we can use a greedy algorithm that runs in linear time. We go through all of letters from B and for each letter we find the first possible match in A after the match of the previous letter.

For example, when checking whether "DOG" is a subsequence of some string A, you can look for the first "D", then the first "O" after that "D", and then the first "G" after that "O".

Problem T: Turning gears

Jano the mechanic has a lot of gears (cogwheels) of different sizes. One day, he took n of them and assembled a complicated machine. Now he wants to know whether and how fast will gear number n rotate if he attempts to rotate gear number 1.

Problem specification

There are n gears numbered from 1 to n. All gears are located in the same plane.

Gear i is represented by a circle with center at (x_i, y_i) and radius r_i. All coordinates are integers. The gears are placed so that the center of each circle lies strictly outside all other circles.

Whenever two circles intersect, the corresponding gears are linked to each other correctly. (A single point of contact between two circles **does count** as an intersection. The size of the intersection does not matter.)

We can describe the rotation of a gear by giving its speed (in full revolutions per second) and its direction of rotation (either clockwise or counterclockwise). Suppose we have two linked gears: one with radius r_1 and the other with radius r_2. If the first gear rotates at v_1 revolutions per second, the other gear will also rotate, but in the opposite direction and possibly at a different speed v_2. The speed v_2 can be computed as follows: $v_2 = v_1 r_1 / r_2$.

In a machine with multiple gears the above equation applies to each pair of linked gears. If this would mean a gear has to rotate at two different speeds or directions, it won't rotate at all.

Jano will attempt to rotate gear number 1 clockwise at 1 revolution per second. How will gear n rotate?

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 number n ($2 \le n \le 1\,000$) – the number of gears in the machine. The i-th of the next n lines describes gear i. The description consists of three **integers** x_i, y_i and r_i – its center and its radius, as described above. You may assume that $0 \le x_i, y_i \le 50\,000$ and $1 \le r_i \le 1\,000$.

In the **subproblem T1** you can make the following additional assumptions about each test case:

- The number of gears will not exceed 200.
- All gears will be directly or indirectly linked to gear 1.
- There will be exactly $n - 1$ pairs of directly linked gears.

Output specification

For each test case, output a single line containing the speed and the direction of rotation of gear n. The speed should be expressed as a fraction `p/q` in reduced form. (I.e., the greatest common divisor of p and q should be 1.) The direction should be either "`clockwise`" or "`counterclockwise`".

If the n-th gear will not rotate at all, print "`does not rotate`" instead.

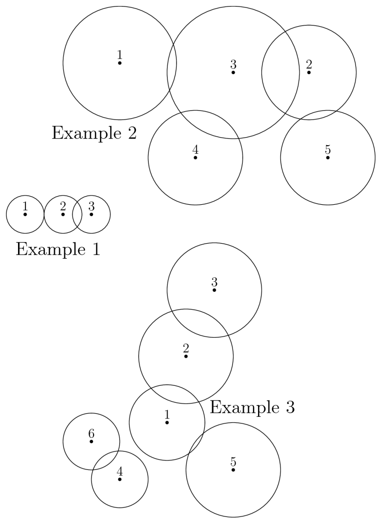
Example

input

```
3

3
0 0 2
4 0 2
7 0 2

5
0 10 6
20 9 5
12 9 7
8 0 5
22 0 5

6
8 6 4
10 13 5
13 20 5
3 0 3
15 1 5
0 4 3
```

output

```
1/1 clockwise
6/5 counterclockwise
does not rotate
```



In the first test case note that gears 1 and 2 are linked, even though they only touch each other at a single point. Also note that the fraction is printed as "`1/1`": the denominator is always printed, even if it is 1. The outputs "`2/2`" and "`1`" would both be rejected by the grader.

In the second test case we can compute the answer as follows:

- *Gear 1 rotates clockwise at 1 revolution per second (rps).*
- *Gear 1 is directly linked to gear 3, thus gear 3 rotates counterclockwise at 6/7 rps.*
- *Gear 3 is also linked to gears 2 and 4. Thus, both of these gears will rotate clockwise. As both of them have the same radius (5), they will both rotate at the same speed: 6/5 rps.*
- *Gear 5 is linked to gear 2. Therefore, gear 5 will rotate counterclockwise. As gears 2 and 5 have the same radius, gear 5 will also rotate at 6/5 rps.*

The third test case ***does not satisfy*** *the additional constraints for subproblem T1. Hence, this test case can only appear in the subproblem T2. This test case illustrates one of the two main reasons why the answer can be "does not rotate".*

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'Žaba' Anderle
Quality assurance:	Jano Hozza

Solution

This problem was inspired by the following "motivational" poster:

The poster actually illustrates a second reason why it might be the case that gear n does not rotate. In some cases, a cluster of gears may become jammed.

But before we analyze that, let's make a quick observation about the rotation speed of gears. The problem statement tells us how the speeds of two joined wheels are related to each other. The formula is pretty straightforward: the longer the radius (or, equivalently, the circumference) of a wheel, the slower it rotates.

We can easily visualize why the formula works as described. Imagine two linked gears: gear A with radius 1 and gear B with radius 3. Gear A is covered in fresh red paint, gear B is all white. As you rotate gear A, a longer and longer part of gear B will become red. By the time gear A makes a full revolution, what part of B's circumference will be red? This is easy: the circumference of B is three times the circumference of A, thus exactly one third of B's circumference will be red. And that, in turn, means that B's speed will be one third of A's speed.

The key to an easy solution is to realize that this also applies if there are more than two gears. Suppose you have a row of gears, each linked to the next one. If you rotate the first gear by some amount, the circumference of each gear will rotate by the same amount. This means that the observation "ratio of speeds is the inverse ratio of radii" holds for *any two gears* in the machine, not just adjacent gears.

In other words, if we want to find the rotation speed of the last gear, all that matters are the radii of the first and the last gear. The sizes of the remaining gears in the row do not matter at all.

This makes the solution to the easy subproblem trivial. We are guaranteed that the gears are all linked and they cannot be jammed, so gear n will definitely rotate. Its rotation speed is r_1/r_n rotations per second. (Remember to reduce this fraction before printing it!)

All that remains is to find its direction of rotation – in other words, to find whether its distance from gear 1 is odd or even. Any simple graph search will answer this.

For the hard subproblem, you first needed to realize that the gears can sometimes become jammed – hence phrases like "Jano will *attempt to rotate* gear number 1" in the problem statement.

But that was not all. You also needed to realize that a cycle in the gear network *does not necessarily mean* that the gears will be jammed.

For instance, take 100 identical gears and place them regularly into a 10×10 grid, so that each gear touches its neighbors. This grid of gears will happily rotate: half of them in one direction, half of them in the other.

In fact, the sizes of gears don't even have to be the same. As we already know, when the gears rotate, their circumferences all shift by the same length. This means that any collection of gears will rotate happily… as long as the directions are correct. The source of the jam is when two different gears try to push a third one to rotate at the same speed but in the opposite direction.

This happens whenever the sequence of rotating gears reaches an *odd cycle* in the gear adjacency graph: a cyclic sequence of $2k+1$ gears in which each gear is linked to both its neighbors. In other words, the jam happens whenever the component that contains gear 1 *is not bipartite.*

This gives us an easy solution for both subproblems. Starting from gear 1, use any graph search (e.g., BFS or DFS) to find the rotation direction for each gear in its component. If, at any moment, you encounter a contradiction, output that the gears are jammed. Otherwise, you already know the rotation direction of gear n, and as we have shown above, you can compute its rotation speed from just the radii r_1 and r_n.

Problem U: Ultimate magic rectangles

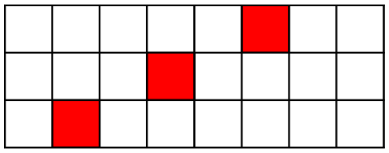
Bob is busy today, so Alice has found a single-player game.

Problem specification

In this game, Alice is given an integer s and an empty table with $r = 3$ rows and c columns. Alice has to fill in some **nonnegative integers** into the cells of the table.

Three cells are called a *triplet* if they lie in different rows and their centers lie on a straight line. The goal of the game is to fill the table in such a way that each triplet will have the same sum.

You are given the number of columns c and the desired sum of each triplet s. Compute the number of ways to fill the table in the desired way. Since this number may be large, compute it modulo $10^9 + 9$.



Above: one of the many triplets on a board with $c = 8$ columns.

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 one line containing two space-separated integers c and s.

In the **easy subproblem U1**, $1 \leq c \leq 50$ and $0 \leq s \leq 50$.

In the **hard subproblem U2**, $1 \leq c \leq 1000$ and $0 \leq s \leq 10^9$.

Output specification

For each test case, print one integer on a separate line – the number of solutions, modulo $10^9 + 9$.

Example

input

```
2

3 1

4 3
```

output

```
5
34
```

In the first test case there are five triplets: each column and both main diagonals of the 3×3 square. The sum of each triplet must be 1, which means that each triplet must contain two 0s and a 1. These are the five solutions:

```
111   000   000   101   010
000   111   000   000   000
000   000   111   010   101
```

In the second test case one of the 34 valid solutions is a 3×4 rectangle full of 1s.

Task authors

Problemsetter:	Jakub 'Xellos' Šafin
Task preparation:	Jakub 'Xellos' Šafin
Quality assurance:	Michal 'mišof' Forišek

Solution

For any $c \geq 2$ the whole table is determined just by 4 of its elements. For example, we can choose the top left 2×2 square, which allows us to uniquely compute the first two elements in the bottom row and successively the top, bottom element and from them the middle element of every column. Bruteforcing the top left 2×2 square, computing the table using this scheme, and checking whether it fits is enough to solve the easy subproblem.

We can now do the above computation symbolically to notice that there are very strict relationships between the elements of the table. Namely, the middle row must be an arithmetic progression, and the top row (and thus also the bottom row) will always contain two interleaved arithmetic progressions – i.e., cells in odd-numbered columns form one arithmetic progression, the remaining cells form the other one.

Here is one way to derive this result formally. Let's focus on a 3×3 subtable of columns $m-1, m, m+1$. We know that the following five equations must hold:

$$\texttt{A[1][m-1]} + \texttt{A[2][m-1]} + \texttt{A[3][m-1]} = s\,, \tag{1}$$
$$\texttt{A[1][m]} + \texttt{A[2][m]} + \texttt{A[3][m]} = s\,, \tag{2}$$
$$\texttt{A[1][m+1]} + \texttt{A[2][m+1]} + \texttt{A[3][m+1]} = s\,, \tag{3}$$
$$\texttt{A[1][m-1]} + \texttt{A[2][m]} + \texttt{A[3][m+1]} = s\,, \tag{4}$$
$$\texttt{A[1][m+1]} + \texttt{A[2][m]} + \texttt{A[3][m-1]} = s\,. \tag{5}$$

Evaluating (1)+(3)-(4)-(5), we get

$$\texttt{A[2][m+1]} + \texttt{A[2][m-1]} - 2\texttt{A[2][m]} = 0\,. \tag{6}$$

In other words, $\texttt{A[2][m+1]} - \texttt{A[2][m]} = \texttt{A[2][m]} - \texttt{A[2][m-1]} = d$ for any m: the numbers in row 2 must form an arithmetic progression; we denoted its difference by d.

Evaluating the differences (1)-(5) and (3)-(4), we can see that in the top row, there's one arithmetic progression with difference $-d$ in even columns and another one in odd columns. If we know the top two rows, we can fill row 3 using (1) (due to symmetry, it will contain arithmetic progressions just like row 1). In addition, it's easy to see that the existence of those three arithmetic progressions automatically satisfies the rule on sums of any three collinear cells. There's just one remaining condition to satisfy: all numbers must be non-negative.

The minimum elements of an arithmetic sequence are at its ends, so we only need to focus on the end of each of our five sequences. Let's denote $\texttt{A[1][1]} = a$, $\texttt{A[2][1]} = b$, $\texttt{A[1][2]} = c$. We can make use of symmetry - if we reverse each of a valid table's rows, d changes sign and the table remains valid. That means we only need to count the case $d < 0$ twice and add the number of solutions for $d = 0$, which is the number of triples such that $a, b, c \geq 0; a, c \leq s - b$: $\sum(s + 1 - b)^2 = (s + 1)(s + 2)(2s + 3)/6$.

If $d < 0$, the sequences in rows 1 and 3 are increasing, so their minimum elements are the first two in rows 1 and 3: $a, c, s - a - b, s - c - b - d$. The sequence in row 2 is decreasing and its minimum element is $b + (c - 1)d$. The conditions which need to be satisfied are

$$\begin{aligned} 0 \le\, & a \le s - b\,, & (7)\\ 0 \le\, & c \le s - b - d = s - b + |d|\,, & (8)\\ (c-1)|d| \le\, & b \le s\,. & (9) \end{aligned}$$

For a fixed b and d, the number of pairs a, c is $(s - b + 1)(s - b + |d| + 1)$. Their sum over all allowed b for a fixed $d < 0$ is an ugly polynomial in $s, |d|$ and its sum over all allowed d (that is, for $|d| \le \lfloor s/(c-1) \rfloor$) is an even uglier polynomial in s only (to be precise, only if we limit ourselves to a fixed value of s mod $(c-1)$, since only that gives a linear dependence of $\lfloor s/(c-1) \rfloor$ on s); the same holds for the final answer. Since each summation increases the degree of the polynomial by 1, the resulting polynomial will have degree 4.

Rather than computing it by hand or even through Wolfram Alpha, it's easier to compute answers for several small values of s (it's sufficient to take s mod $(c-1) + k(c-1)$ for $0 \le k \le 4$) by trying all b, d and summing up the number of choices for a, c; using these values of a polynomial in points $(y_k, x_k = k)$, we can interpolate the answer as

$$\sum_{k=0}^{4} \prod_{j \ne k} \frac{s/(c-1) - j}{k - j} y_k\,, \tag{10}$$

where division by $k - j$ can be handled with precomputed modular inverses. Time complexity: $O(c^2)$.

You can try finding the polynomial in x from the decomposition $s = x(c-1) + r$, e.g. using WolframAlpha. It allows us to compute the answer in $O(1)$ and it only has 25 terms!

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
AAAAAAAAAAAAAAAAAAAAHELLOWORLD
```

Task authors

Problemsetter:	the result of brainstorming during the task selection meeting
Task preparation:	Jano Hozza
Quality assurance:	Michal 'Mio' Hozza

Solution

In the easy subproblem we have to produce a new string which is longer then all ten words together. Therefore we can just concatenate all ten words and append any two characters to the end.

In the hard subproblem it is obvious that in order to fit the words into a 39-letter string two of the words will have to overlap. As we are guaranteed that a solution exists, we know that each input has to contain two words that can overlap.

There are four possibilities how a pair of words can overlap: they can share 1, 2, 3, or 4 common letters. For each pair of words we will check these four possibilities, until we find a pair that fits together.

As soon as we find an overlapping pair of words, we will combine them together into a new word that has fewer than 8 characters. We can then append the remaining 8 words to get a valid solution of length at most 39. If we got a solution that is too short, we fix it by appending arbitrary characters.

Some common pitfalls in the hard subproblem:

- "For each pair of different words" fails if all ten input words are the same
- Trying only overlaps by a single letter sometimes does not work, for example for the input `ABCD BCDE FZZZ GZZZ HZZZ IZZZ JZZZ KZZZ LZZZ MZZZ` the only solution is to merge `ABCD` and `BCDE` into `ABCDE`.

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
```

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Lukáš 'lukasP' Poláček
Quality assurance:	Peter 'PPershing' Perešíni

Solution

Easy subproblem

A ball of diameter 1 meter has volume $\pi/6$ m^3, which is about 0.5236 m^3. A cube with a 1-meter long side has volume 1 m^3, less than twice more than the ball. We notice that in the easy inputs there is plenty of room inside the box, sometimes three times as much as the total volume of the balls. What if we could use treat every ball as smallest cube surrounding it and pack the cubes instead?

We also notice that there are always 2 balls with radii in ratios of $2:1$ and $3:1$, so we can exchange a group of $2\times2\times2$ small cubes or $3\times3\times3$ small cubes for a big cube. We check the dimensions of the box and realize a simple greedy approach will work: we pack all big cubes first and then fill the rest of the space with small ones.

One of the easiest way to implement this greedy approach is the following:

- Create a 3D grid of small cubes inside the box, starting at $(0,0,0)$. Store the coordinates of all small cubes inside a set data structure.
- Create a 3D grid of big cubes inside the box, starting at $(0,0,0)$. Pick as many cubes as needed and place all copies of the big ball inside those cubes. Remove all small cubes inside the used big cubes from the set data structure.
- Place all small balls into the remaining small cubes in the set data structure.

Hard subproblem

We can start by observing that the limits on the box size are much stricter than in the easy subproblem. Moreover, the ratios between the cubes sizes are smaller here. As such, we really need to perform a "ball-packing". Fortunately, you already know how from the experience how to efficiently pack balls of roughly the same sizes – when was the last time you saw a bucket full of apples or strawberries? Could you do much better than the state to which the fruits arrived by themselves with the help of gravity? This suggest a "straightforward" approach – just throw all of the balls into the box and let them settle down. The real technicality is the implementation of this process. There is an easy and a hard way. An easy way is to download some physics engine, add balls, gravity and let it play. The hard way is to code a physics-like dynamics with a collision algorithm yourself. Fortunately, collisions of balls are detected very easily and the more challenging part of the task is to keep balls from "squishing" or just being accelerated to extreme speeds (https://what-if.xkcd.com/1/) which can happen if the simulation starts diverging.

Finally, here are few notes you might find useful:

- To avoid the quite strict overlap check in the grader, it is best to enlarge the balls by a few percent.
- Previous bullet applies applies also to solutions using a physics engine. They tend to keep some collision overlaps.
- The initial positions of balls affect the result, sometims even by over 10%. If the current stable state does not fit into the box, we just need to reeated the experiment with new randomized positions of balls. Only a small number of such trials should suffice.

But what if I hate physics engines

Well, if you don't want to use real physics engine and you are fed up with coding your own (though, you should be glad that you can safely ignore rotation and angular momentum of balls), there is an easier way – let's call it no-physics engine. The idea is very simple.

1. distribute balls randomly in the box
2. repeat until solution is ok or maximum steps reached

- there will be no velocity of the balls. Everything is instant-time teleport ;-)
- "solve" collisions with the box. If a ball is too close to the box, just move it to the correct distance + random epsilon.
- "solve" collisions of balls. If two balls are too close to each other, change their positions such that their *midpoint* is fixed but their distance is now $r_1 + r_2$ + random epsilon.
- Note that random epsilons are quite important – as in a naive implementation the collision check is likely to be in the deterministic order, small random variations allow the algorithm to escape some nasty cycling which could happen if the result after processing of collisions is the same as before.

3. if we did not find a valid solution, repeat with step 1.

This simple "no-physics" engine algorithm is enough to solve both easy and hard subtasks, avoids long simulation times (because everything is instantly teleported) and is quite elegant on its own.

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**, $1 \le n \le 10$.

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

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.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Jakub 'Xellos' Šafin
Quality assurance:	Michal 'mišof' Forišek

Solution

For the easy subproblem, we can view all permutations as vertices in a graph and swaps as edges between them. We're looking for the number of shortest paths from the identity permutation to the input permutation (or vice versa), which can be computed using a single breadth-first search.

In the hard subproblem we can use a different graph idea – the graph of a permutation. Consider a graph with vertices $1, \dots, n$ and n directed edges: from each i to the corresponding p_i. Clearly, for any permutation this graph is a collection of disjoint cycles. We can easily verify that swapping two numbers from different cycles merges those cycles together. By symmetry, swapping two numbers from the same cycle breaks the cycle into two smaller ones.

The identity permutation consists of n single-element cycles. From this, it's apparent that if the input permutation consists of c cycles, the minimum number of swaps needed to sort it is $n - c$. Furthermore, a sequence of $n - c$ swaps is a solution if and only if each of them swaps two numbers that currently lie on the same cycle.

Obviously, each cycle is completely independent from the others. Also, we do not care about the particular elements a cycle contains. Hence, the answer to our problem is completely determined by the sequence of cycle lengths. The cycle lengths are the only thing that matters.

Suppose we have a cycle of length l and we want to make a swap that splits it into two cycles of lengths l_1 and l_2. How many such swaps exist? Clearly, the answer is l, excepf for one special case. If l is even and $l_1 = l_2$, there are only $l/2$ such swaps. We will use $S(l, l_1)$ to denote this number of swaps.

How many ways are there to split one cycle of length l into l cycles of length 1 using a sequence of exactly $l - 1$ swaps? Let's use $C(l)$ to denote the answer to this question.

We can use dynamic programming to compute $C(l)$. We have $C(1) = 1$. Now let $l > 1$. In the first step we will produce two cycles of lengths l_1 and $l - l_1$. We can try all l_1 from 1 to $\lfloor l/2 \rfloor$. For each l_1 we can then continue splitting each smaller cycle separately. There are $C(l_1)$ ways to split the first cycle, $C(l - l_1)$ ways to split the second cycle, and $\binom{l-2}{l_1-1}$ ways to interleave those two independent sequences of swaps. The above gives us the following recurrence:

$$C(l) = \sum_{l_1} S(l, l_1) \cdot C(l_1) \cdot C(l - l_1) \cdot \binom{l-2}{l_1-1}.$$

We can precompute the binomial coefficients and then evaluate this recurrence in $O(l^2)$. That is still too slow for the hard subproblem, but it is already fast enough to tell us a lot of values of our sequence – more than enough to guess the pattern, or to make a lookup in the Online Encyclopedia of Integer Sequences (OEIS). It turns out that for $l \geq 2$ we have $C(l) = l^{l-2}$. This can be computed quickly using modular exponentiation.

Now that we can solve our question for a single cycle, all that remains is to count the ways in which we can interleave the solutions to all cycles of the given permutation. This is very simple: the answer is the appropriate multinomial coefficient.

Imagine that we have a permutation with cycle lengths $l_1, \dots, l_c$ and that we already chose one specific solution for each cycle. The number of ways to interleave these solutions is always the same:

$$\binom{n-c}{l_1-1,\ l_2-1,\ \ldots,\ l_c-1} = \frac{(n-c)!}{(l_1-1)!\ldots(l_c-1)!}\,.$$

The modulus is a prime larger than n, so the factorials in the denominator can be replaced by their multiplicative inverses $inv[(l_k-1)!]$ in the numerator. Thus, the complete answer can be computed as follows:

$$(n-c)!\prod_{k=1}^{c} inv[(l_k-1)!]\cdot l_k^{l_k-1}\,.$$

Note that $C(l)$ is also Cayley's formula giving the number of labeled trees on n vertices. As an exercise, try finding a combinatorial bijection between labeled trees and our sequences 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**".

Task authors

Problemsetter:	task theme by Michal 'Žaba' Anderle, specific ideas by Vlado 'usamec' Boža
Task preparation:	Vlado 'usamec' Boža
Quality assurance:	Tomáš 'Tomi' Belan

Solution

There are two main ways how to solve this task. You can either generate the log using your own program (which requires understanding of the inner workings of the game) or put a small program into the javascript console in your browser to make it play the game for you much faster.

The easy subproblem is pretty straighforward, since the rectangles and squares do not move. But there are several gotchas which you could encounter:

- The instruction for the first level is not in easy-data.js but in the html source.
- Some of the later levels have two commands instead of one.
- After the last level you should write "done" *instead of* "next" (not *after* "next").

The hard subproblem adds some more tricks on top:

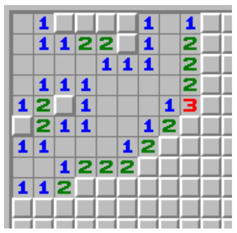
- The source code for game is minified.
- There is a timestamp in the logs. And also instructions about waiting between clicks. And on top of that, you are required to produce valid timestamps that all lie within the duration of the contest (more precisely, between the beginning of the contest and the moment when you submit). The grader rejected submissions with timestamps outside this range. One side effect of this check: it was impossible to submit D2 during first half hour of the contest.
- The squares and rectangles are changing positions. To handle this, you can either read the game's source code to find the pseudorandom algorithm, or let it generate the levels and then query the generated image to find which shape is where. We find the second option much easier.

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.

Task authors

Problemsetter:	Michal 'mišof' Forišek, expanding on an original idea by Jaro Petrucha
Task preparation:	Monika Steinová
Quality assurance:	Michal 'mišof' Forišek

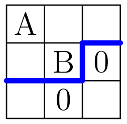
Solution

The easy subproblem can be solved by finding a single valid solution and then padding it with an appropriate number of mines. Each instance that satisfies the constraints for the easy subproblem is solvable.

Below, we give a solution to the hard subproblem. We aim to present a solution that reduces the need for a manual case analysis to a minimum.

Theorem 1. In any valid solution the revealed zeros must form a full rectangle.

Proof: All revealed zeros form a connected component. Imagine walking around the outer boundary of this component (along the grid lines between cells) in counter-clockwise direction. If we only take left turns, the outer boundary must be a rectangle. If we ever take a right turn, we must have the situation shown in the figure below. However, this cannot happen in a valid solution: the revealed non-zero cell B must contain a 1, and thus the player can now determine that the hidden cell A must contain a mine.



Above: a right turn on the ccw boundary of a component of zeros. The thick line shows one possible continuation of this boundary. Cell B must be a 1, and cell A must be a mine.

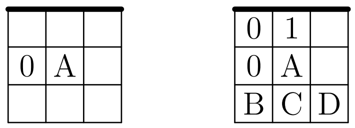
Thus, the only possibility is that the outer boundary of the component of zeros is a rectangle. Furthermore, there can be no non-zero cells in this rectangle, otherwise we could take the non-zero cell with the lexicographically smallest coordinates and we would have exactly the same situation that was shown in the figure above.

In the next step of our solution we will solve the special cases with $r \le 2$ or $c \le 2$. These cases are discussed in detail at the end of this solution, for now we'll skip them and we'll focus on the big picture instead. Without loss of generality, we now may assume that $3 \le r \le c$.

Theorem 2. If $r, c \ge 3$, a valid solution cannot contain a revealed 0 that is exactly one cell away from the border of the board.

Proof: Suppose we have the situation shown in the first figure below: a revealed cell with a 0 that is one cell away from the border, and a revealed cell A that contains a *positive* number. A situation like this one cannot occur in a valid solution.

Why is that? The cell above the 0 must obviously contain a 0 as well. The cell above cell A cannot be a 0, otherwise we would have the situation shown above once again. It cannot be a 2, because then the cell to the right of it would have to contain a mine. Thus, we must have the situation shown in the second figure below.



Above: the deduction from seeing a zero and a nonzero one cell away from the border. (The thick line denotes the border of the game board.)

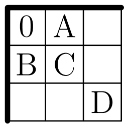
However, this last possibility will also lead us to a contradiction. We know that cell B cannot contain a mine, hence A is either 1 or 2. If $A = 1$, the player will be able to conclude that cell D is empty. And if $A = 2$, the player will be able to deduce that cell D must contain a mine. In either case, our evil plan fails.

Thus, whenever we see a revealed 0 in the second row of the grid, the only remaining possibility is that the entire second row are 0s. However, this then produces a 0 in the second column. By repeating the same argument, the second column must be all 0s as well. From Theorem 1 we know that revealed zeros form a rectangle – but given what we know, the only possibility is that the entire board is a rectangle of zeros, and that cannot happen.

This concludes our proof: a revealed 0 in a cell that is one cell away from the border can never appear in a solution.

Theorem 3. If $r, c \geq 3$, there is no valid solution if the player clicks a corner cell.

Proof. Let's label the cells as shown in the figure below. From Theorem 2 we know that A, B, and C must all contain positive numbers.



Cells A and B must both be 1s, otherwise the player would be able to determine the contents of their hidden neighbors. Then, C must be either 2 or 3. If $C = 2$, the player can be sure that D is empty, and if $C = 3$, the player can be sure that D contains a mine. Thus, there is no valid solution in which we reveal the corner.

At this moment we know that there are only two types of valid solutions left:

- The revealed zeros all lie along a single border of the board (without coming too close to either corner).
- The revealed zeros form a rectangle somewhere inside the board, at least two cells away from each border.

There are valid solutions of both these types. How can we analyze them? By using a simple brute force approach. Suppose we fix the rectangle of revealed zeros. We know that the cells at distance 1 from the rectangle are precisely all the revealed cells with positive integers. We can now do the following:

1. Construct the set of cells that are in distance 2 from the rectangle.
2. Try all possible placements of mines onto these cells.
3. For each placement of mines compute the numbers revealed in the cells around the rectangle of zeros.
4. Divide the placements of mines into "equivalence classes" according to the revealed numbers.

5. For each possible sequence of revealed numbers, look at all placements of mines in that "equivalence class" and check whether they have the desired property. (I.e., whether each cell at distance 2 from the revealed zeros *can, but does not have to* contain a mine.)

Clicking a cell at the border, smallest number of mines.

Suppose that $r \geq 3$ and $c \geq 6$. (Smaller cases can be solved by examining all possibilities.)

If the player clicks a cell at the border, we will need at least 4 mines. If $m < 4$, there is no solution. Why? If we reveal a single zero, the best we can do is shown in the figure below.

	1	0	1	
	2	1	2	

The cells at distance 2 from the zero contain either 3 or 4 mines, and each of those cells can, but does not have to contain a mine. If we now have an arbitrarily large board, we can place three mines at distance 2 from the zero, and the fourth mine anywhere we like. This shows that 4 mines are sufficient.

We can easily show that 4 mines are also necessary. We know (by trying all possibilities) that we cannot use fewer if we only reveal a single zero. If we reveal at least two zeros, we'll have a situation like the one shown below.

	1	0	0	1	
	A	B	C	D	

There are at least two mines – each adjacent to one of the 1s. The values A and D cannot be 1s, so there are at least two other mines, each adjacent to one of those numbers.

Clicking a cell at the border, largest number of mines.

Again, we'll start by looking at what happens if we just reveal a single zero. The best we can do now (i.e., the most mines) is the configuration shown below. There are either 5 or 6 mines at distance two from the 0.

	1	0	1	
	3	2	3	

In other words, among the cells that are at distance *at most* two from the 0, there are 9 or 10 cells that are not mines. Thus, if we reveal this pattern to the player, we can produce valid solutions with up to $rc - 10$ mines.

Can we do better than that? Clearly we cannot. If we reveal more than one zero, we can easily show that there will always be at least 10 cells that don't contain a mine.

Thus, we have shown the following results for the click on a border:

- We cannot have fewer than 4 mines.
- If we use the first pattern, we can create solutions with $4 \dots rc - 12$ mines.
- If we use the second pattern, we can create solutions with $6 \dots rc - 10$ mines.
- We cannot have more than $rc - 10$ mines.

And as for $r \geq 3$ and $c \geq 6$ those two ranges overlap, we are done – all the counts of mines from 4 to $rc - 10$ are possible.

Clicking a cell inside the board.

Suppose that $r \geq 5$ and $c \geq 6$. (Smaller cases can be solved by examining all possibilities.)

Now we are going to do the same thing again, but this time for a clicked cell that is at least 2 cells from each border. We'll skip the details and we'll just show the patterns. (We constructed them by brute force, just to be sure we didn't miss anything, but it's perfectly feasible to construct them by hand as well – as soon as you realize that you just want a 3×3 square.)

The pattern for the fewest mines is shown on the left. There are either 4 or 5 mines at distance two from the 0. The pattern for the most mines is shown on the right. The cells at distance two contain either 11 or 12 mines – in other words, there are either 13 or 14 non-mine cells at distance at most two from the click. As before, it is very easy to show that we cannot have fewer/more mines if we reveal more than one 0.

	1	1	1	
	1	0	1	
	1	1	2	

	3	2	3	
	2	0	2	
	3	2	3	

Hence, the conclusion for clicking inside a large board is that we can have between 5 and $rc - 14$ mines, inclusive. All that remains are the special cases we skipped before: the very narrow boards.

Boards of height 1.

So simple: there is never a solution. The first non-zero digit you'll reveal must be a 1, and the player will know that the next cell has a mine.

Boards of height 2.

Most of this was spoiled in the example in the problem statement. The clicked cell will be a 0. The cell above/below that cell will also be a 0. Thus there are only two possible types of revealed cells, as shown below.

```
0001..    ..1001..
0001..    ..1001..
```

For each of these we can easily find the smallest and largest number of mines we can use.

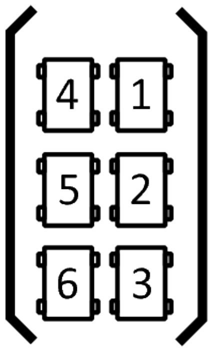
Problem F: Ferries

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 $a_1, \dots, a_k$ 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 $1 \le x \le 2n$. The second line contains the integers $a_1, \dots, a_k$.

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

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

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.

Task authors

Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Michal ‘mišof’ Forišek
Quality assurance:	Tomáš ‘Tomi’ Belan

Solution

The algorithms from this problem have originally been devised for another task: card shuffling. There are two ways to perform a *perfect riffle shuffle* of a deck of cards. (In one of those ways the top card remains on the top, in the other way it becomes the second card from the top.) These two ways of shuffling change the order of a deck of cards in exactly the same way our ferries change the order of the cars in the convoy.

The question from the problem statement then becomes a very natural question: what is the smallest number of (carefully chosen and perfectly executed) riffle shuffles to get a particular card from position p to position q in the deck?

A nice solution to this problem is described by Sarnath Ramnath and Daniel J. Scully in their paper “Moving card i to position j with perfect shuffles”.

The general idea of this solution is very simple. Let’s use zero-based indexing, i.e., cards have numbers 0 through $2n-1$. Counting modulo $2n$, a card from position i can go to either $2i$ or $2i+1$. If we start to draw a tree of possibilities, we will quickly realize that after k shuffles the reachable numbers are precisely those in the interval $[i \cdot 2^k, (i+1) \cdot 2^k - 1]$. (Of course, still counting modulo $2n$. As soon as the length of this interval reaches $2n$, any value is possible.)

Thus, we have a very simple solution:

- Increment k until our interval hits the desired destination. (This takes $O(\log n)$ steps.)
- From the binary value of the offset your destination has within the interval (i.e., from the value “destination minus left boundary of the interval”), reconstruct the right sequence of “times two” and “times two plus one” that gets you from your source to your destination.
- For each individual operation, determine whether it is done by an out-shuffle or an in-shuffle (i.e., an R-ferry or an L-ferry).

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

```
3

-10

0*0*1

0+(1*2)
```

output

```
10
1
0+(12)
```

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'Mio' Hozza
Quality assurance:	Peter 'PPershing' Perešíni

Solution

The solution to this problem is surprisingly simple. So simple that some people actually solved this task *in a text editor*. Here is one possible complete implementation using sed:

```
sed -e '1d' -e '/^$/d' -e 's/[^0-9]//g' -e 's/^0*//' -e 's/^$/0/'
```

The solution in human words: to produce the largest possible value, just keep all the digits (except for unnecessary leading zeros) and erase everything else.

Why does this work? The intuition behind this claim is that the numbers $40 - 7$, $40 + 7$, and even $40 \cdot 7$ are all smaller than the simple 470.

Now for a more formal argument.

First of all, note that if we take a valid expression and change all minuses into pluses, its value will not decrease. This is not a valid change in terms of our problem statement, but we will later erase all of those characters anyway, so it does not matter.

Next, the unary pluses can be safely erased. Thus, we are left with an expression with binary `+`s, binary `*`s and parentheses. Both `+` and `*` are nondecreasing functions on nonnegative integers: if we increase an argument, the result does not decrease.

The concatenation of nonnegative integers a and b has the value $a \cdot 10^l + b$, where l is the number of digits in b. We can now argue as follows:

$$\begin{aligned} a \cdot 10^l + b &\geq a \cdot b + b \geq a \cdot b \\ a \cdot 10^l + b &\geq a \cdot 1 + b = a + b \end{aligned}$$

For example, instead of $a \cdot 935$ it is always strictly better to do $a \cdot 1000 + 935$.

Consider any expression that still contains some non-digit characters. Whenever you see an operator between two numbers, you may remove it without decreasing its value. Whenever you see parentheses around a single number, you may remove them as they have no effect anyway. After a finite number of such steps you will be left with no operators and no parentheses, just a single number.

Summary: We can start with any expression, apply the above sequence of steps, and end with just the string of digits. During those changes the value of the expression never decreased. Hence, the string of digits must be the expression with the largest possible value.

One final comment: If you did not see the trick, a valid solving strategy is to use brute force to solve the small subproblem – e.g., in Python you can easily generate all substrings and call `eval` on each of them (but you need to be careful because `**` is exponentiation in Python). Once you get it accepted (to check that your answers are indeed correct), you can examine them and discover what's going on.

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 ($1 \leq n \leq 10\,000\,000; 1 \leq q \leq 25\,000\,000$). The next $n-1$ lines contain integers p_i, q_i ($1 \leq p_i, q_i \leq n$) 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 ($1 \leq a_i, b_i \leq n; 0 \leq c_i \leq 1$) 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 $1 \leq i < n$. (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

```
1

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

output

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

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Jakub 'Xellos' Šafin

Solution

How will the town change after a journey from a to b that travels through r roads? If it's not snowing, the roads between a to b are simply set to zero snow. If it is snowing, the roads from a to b are set to $r, \dots, 3, 2, 1$ cm of snow, and every other road gains additional r cm of snow.

Updating every road during every journey would be too slow. Instead, let's keep a global timer. For every road, we remember the last time we visited it, and we only advance the timer when we clean a road while it's snowing. The amount of snow on a road is simply the current time minus the last visited time. This way, we only need to update the roads we actually visit.

In the easy subproblem, we now effectively have an array of roads, and we need to quickly find the sum of a range, or update a range to contain an arithmetic progression, such that the difference between consecutive terms being either 0 if it's not snowing, or -1 or 1 if it's snowing depending on the travel direction. This can be done with a *lazy-propagation segment tree*, modified so that instead of simply setting a range to a constant value, we lazily remember the arithmetic progression that belongs to the current node and propagate the correct half to each child.

To extend this to the hard subproblem, we can use *heavy path decomposition* to split the initial tree into a set of disjoint paths such that going from any a to any b only intersects $O(\log n)$ of those paths. Then we build a segment tree for each heavy path, allowing us to process a journey in $O(\log^2 n)$.

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.

Task authors

Problemsetter:	Adam Dej and Michal 'mišof' Forišek
Task preparation:	Adam Dej
Quality assurance:	Michal 'Mio' Hozza

Solution

Once again, we decided to test the boundaries of the comfort zone of our contestants.

There were several ways how to solve this problem. The easy subproblem was easily solvable on any platform. You just had to realize that you are holding a full disk image and you had to attach it to your OS. (For some, such as Windows, this requires the installation of a custom driver, but it should be quick and painless if you have a clue about what you are doing.)

Below we will describe one possible solution, using standard Linux utilities. First we need to find out what the input files are. If we don't already have an educated guess (e.g., from opening one in a text editor and seeing the string `/mnt`), we can ask the useful `file` utility.

```
$ file I1.in
I1.in: Linux rev 1.0 ext2 filesystem data, UUID=58a84427-92cc-4538-b61f-8f984f07f4bd (large files)
$ file I2.in
I2.in: Linux rev 1.0 ext2 filesystem data, UUID=ef5d870a-7ba3-4a5b-a3d3-7c29626efdcf (large files)
```

We can see that we are dealing with images of `ext2` filesystem. To access files inside we need to mount it. However, we can only mount block devices, not files. So we need to create a block device from this file, so-called `loop` device. Mount will do this for us when we specify the `loop` option. And just to be on the safe side, let's mount the filesystem read-only (`ro` option).

```
# mount -o loop,ro I1.in /mnt
# cd /mnt
```

In the easy subproblem, we can see that there is a structure of folders with symlink loops. Do we have any files?

```
# find -type f
./11/not-password-5
./27/8/not-password-13
./27/10/not-password-12
./27/not-password-12
./14/not-password-7
./26/password-9
...
```

OK, so let's filter out only `password-x` files:

```
# find -name 'password*'
./26/password-9
./12/password-5
./19/password-8
```

```
./9/21/password-4
./9/4/password-3
./8/password-2
./0/password-0
./15/password-6
./4/6/password-1
./17/password-7
```

There are 10 of them, so we just print them in the correct order:

```
# for i in ‘seq 0 9‘; do find -name "password-$i" -exec cat {} +; done
Password: uBUJpO7fWB5IGTUQ7z4JE32V69W22d3u
Password: Zk5kvD87CeiZfCUbFsp0cjHH4xo9dW6g
Password: y18Sq2V94lixg1OoZWyHYDbEnGYEAT91
Password: CP0MuBxNPmKUl3zGGeUIEAZxZR44oibF
Password: 9fPZZb06YAr0xlRip9f3DWdxYCizyDwF
Password: XCNSWRY8NBh9DC3BQ6pmnr0nDzJfDxwh
Password: W0cV0ItSiB3GdJmuZpvwamdVpUX18Ydh
Password: 3AGgg26w2YOSXhHt8aeGAg742GxVA34H
Password: yUboh0Zw712exxfjA1pMWQ4la2YT3p3l
Password: cRI2hCO8aCJhZnbYe2pUM1gVCjDoHzMD
```

Since we should only submit the 32-character strings let’s strip ‘Password:’.

```
# for i in ‘seq 0 9‘; do find -name "password-$i" -exec cat {} +; done | cut -c 11-
uBUJpO7fWB5IGTUQ7z4JE32V69W22d3u
Zk5kvD87CeiZfCUbFsp0cjHH4xo9dW6g
y18Sq2V94lixg1OoZWyHYDbEnGYEAT91
CP0MuBxNPmKUl3zGGeUIEAZxZR44oibF
9fPZZb06YAr0xlRip9f3DWdxYCizyDwF
XCNSWRY8NBh9DC3BQ6pmnr0nDzJfDxwh
W0cV0ItSiB3GdJmuZpvwamdVpUX18Ydh
3AGgg26w2YOSXhHt8aeGAg742GxVA34H
yUboh0Zw712exxfjA1pMWQ4la2YT3p3l
cRI2hCO8aCJhZnbYe2pUM1gVCjDoHzMD
```

(Yeah, we know, you probably stopped two steps ago and did the rest in a text editor.)

In the hard subproblem we have just one file `_YOU_NEED_TO_LOOK_DEEPER` and `lost+found` folder. What’s deeper than files in an ext filesystem? After a bit of reading Wikipedia we can find out that files and directories are represented by `inode`s. And directories contain list of filenames and inode numbers of files in them. Now what is the `lost+found` folder for? After a bit of googling we will find that this folder is used by a filesystem check utility, which can recover some lost information, like inodes not attached to any directory.

So let’s check whether we are really dealing with a corrupted filesystem:

```
# e2fsck I2.in
e2fsck 1.42.13 (17-May-2015)
I2.in: clean, 13/1280 files, 199/5120 blocks
```

Well, the `e2fsck` thinks the filesystem is clean. That just means that all pending writes were successfully made before the system was unmounted, not that the system is not corrupted. Let's do a forced check:

```
# e2fsck -f I2.in
e2fsck 1.42.13 (17-May-2015)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Unattached inode 214
Connect to /lost+found<y>? yes
Inode 214 ref count is 2, should be 1.  Fix<y>? yes
Pass 5: Checking group summary information

I2.in: ***** FILE SYSTEM WAS MODIFIED *****
I2.in: 13/1280 files (0.0% non-contiguous), 199/5120 blocks
```

Ha!

```
# mount -o loop,ro I2.in /mnt
# cd /mnt/lost+found/
# ls
'#214'
# cat \#214
OR0
xVv
1nM
fyD
XoN
bgW
erL
VgI
wz4
7JK
qtb
F8o
zt
```

The last thing that remains is to join the lines.

```
# paste -s -d'\0' \#214
OR0xVv1nMfyDXoNbgWerLVgIwz47JKqtbF8ozt
```

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, $1 \leq m \leq n \leq 2\,000\,000$, $1 \leq q \leq 5\,000\,000$, $1 \leq c, v, g, l \leq 10\,000\,000$. 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.*

Task authors

Problemsetter:	Jakub 'Xellos' Šafin
Task preparation:	Jakub 'Xellos' Šafin
Quality assurance:	Tomáš 'Klokan' Dzetkulič

Solution

First of all, let's find out how the length of a queue depends on time. That is, we need to solve the differential equation

$$\frac{\mathrm{d}l}{\mathrm{d}t} = \frac{g}{l} \tag{11}$$

with an initial condition $l(t_0) = l_0$ if the queue's parameters were last updated at time t_0. Simple rearranging and integration gives $(l^2 - l_0^2)/2 = g(t - t_0)$, so $l = \sqrt{l_0^2 + 2g(t - t_0)}$ and

$$\tau_i = \frac{1}{v_i}\sqrt{l_i^2(t_0) + 2g_i(t - t_0)}\,. \tag{12}$$

There are two main observations to be made here:

- two non-identical functions τ_i and τ_j will intersect at most once
- any function τ_i at any time after it's updated will never exceed τ_i from before that update

The first observation is true because τ_i is just a square root of a linear function, the second one holds thanks to non-decreasing v_i and non-increasing g_i. What do they tell us? Thanks to the first one, if we know that one function became smaller than another after some time, we can discard the latter; this allows us to construct an algorithm where we maintain a set of "interesting" functions with the only operations being "move forward in time" and "add a function". The second observation lets us keep old functions without removing them (they can stop being interesting, though), since that doesn't decrease the minimum, so we can treat U-queries as O-queries.

In the easy subproblem, we can compute time intervals when each function gives the minimum. Let's sort the functions non-increasingly by their values at an infinite time - by $\sqrt{g_i}/v_i$. We'll push them into a stack in that order. Before pushing in a function f, we can remove all functions at the top of the stack which are never smaller than f - those which can become minimum (either at time 0 when they're added or when they intersect with a previously added function) only after intersecting with f. Afterwards, only the last function before f in the stack determines when f can become minimum and we can compute their intersection point (if it exists).

For each function that's left in the stack at the end, we now have a non-empty interval when it's minimum. Answering a type Q query is just a matter of finding the right time interval. Overall, this takes $O((M + Q)\log M)$ time. Note the similarity to the convex hull algorithm; in fact, the full solution in case the functions are linear and not sqrt(linear) is known under the name "convex hull trick".

With U/O queries, we can still parse the input, find all functions with times when they're added and sort everything by $v_i/\sqrt{g_i}$. We'll keep a set of interesting functions (those which can yet become minimum - haven't been intersected by an added function that comes later in the sort order) and a priority queue of events - intersections between them, sorted by time. We don't need to keep track of all intersections, only those between functions adjacent in the sort order; this way, each function has to gradually intersect all those before it to become the minimum.

There are two types of operations to implement: add a function and process events occurring before a given time. When adding a function, we should look at its nearest ≤ 2 interesting neighbours and add corresponding two events; when removing a function (making it non-interesting) with 2 neighbours, we should add one event between them. When processing an event, we need to check if it's still valid, since one of the functions between which it would've happened could have been removed, and if it is, remove the function earlier in the sort order. Afterwards, the first interesting function gives the minimum at the given time.

Since we only add 1 event when removing a function and 2 when adding it, there can only be $O(M+Q)$ events to process. Processing one takes $O(\log M + Q)$ time, so the time complexity of the full algorithm is still just $O((M + Q) \log M + Q)$.

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.*

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Jakub 'Xellos' Šafin

Solution

Easy subproblem

In the easy subproblem we are given a sorting routine that takes an n-element array and applies the following steps:

- Compute some a, b, c that are guaranteed to be in $[0, n-1]$.
- Make a recursive call to sort the first a elements.
- Make a recursive call to sort the last b elements.
- Make a recursive call to sort the first c elements.

There is a special case that makes the solution work for $n \leq 2$, but that's it. On the first glance this seems pretty suspicious: does this *ever* work? But if we do a few experiments, we'll discover that it does seem to work for all short arrays we throw at it.

Before we explain why it sorts and where it stops sorting correctly, we'll describe a really simple approach that can solve this task without a full understanding of the implemented algorithm.

A trick that uses inductive reasoning

We will start by taking an educated guess that the worst possible input for an algorithm of this type will be an array sorted in reverse order. We could try this particular input for each n and see where it fails.

Well, that's a good idea, but it's not enough. The trouble here is that the algorithm is very inefficient and it would take ages to terminate for larger values of n.

Luckily for us, there is a very simple optimization. Suppose we already tried all input sizes smaller than the n we are now verifying. This means that whenever we call our function on an array of length smaller than n, the function will sort the array correctly. We can do the same thing much faster – simply by calling a standard $O(n \log n)$-time sort instead of making the recursive call. This change makes the whole routine run in $O(n \log n)$, and therefore our search for the smallest counterexample will run in $O(n^2 \log n)$, which turns out to be fast enough.

Here's a summary of this solution:

- Take the code you were given in `k1.{cc,py}`.
- Change the three recursive calls to three calls to `sort()`.
- For each $n = 1, 2, \ldots$, feed a reversed array into the modified function.
- Stop once you find the smallest n where the modified function fails.
- That n is the answer.

A solution in which we understand what's going on

OK, that was actually pretty cool, but some of us prefer to be sure that our solution is correct. What does the magic algorithm actually do?

The algorithm is a generalization of an esoteric sorting algorithm known as StoogeSort. In StoogeSort we always have $a = b = c = \lceil 2n/3 \rceil$, and with this setting the algorithm is correct – but its time complexity is much worse than the quadratic time complexity of a dumb BubbleSort.

When exactly do these StoogeSort-like algorithms work correctly? Let's draw a picture and use it to do some reasoning.

```
                            a
 first pass: |=============================|--------|

                                       b
second pass: |------------|=========================|

                          c                n-c
 third pass: |========================|-------------|
```

Imagine that we are doing a proof by induction (on n) that our algorithm correctly sorts any n-element array. The base case of the proof works: the algorithm has a special branch that makes it correct for $n \leq 2$. Now let's try proving the inductive step.

After the third pass we want to have a sorted array. According to the induction hypothesis, the third pass correctly sorts the first c elements. The output will be correct if and only if the last $n - c$ elements were already the correct ones, and in their proper places. Let's call these elements *the large elements*.

The second pass sorts the last b elements. This pass will put the large elements into their proper places if and only if all of them are among the last b elements of the array at the beginning of the second pass.

At the beginning some, possibly even all, large elements may be among the first a elements of the array. By sorting the first a elements we bring those large elements to the last few positions of the sorted part. All of them must belong to the part that will be sorted in the second pass. This means that the overlap between the first and the second pass must be at least as large as the total number of large elements.

(A more precise proof of the previous step would first show that a must be strictly greater than the number of large elements. Do you see why we should do that?)

And that's it. The entire proof will work, as long as that last condition is satisfied. How does the condition look like in symbols? The length of the overlap between the first two passes is $a + b - n$. This must be at least as large as the number of large elements, which is $n - c$. In other words, we must have $a + b + c \geq 2n$.

And that's all there is to this problem. As long as we have $a + b + c \geq 2n$, the routine sorts properly. And for the smallest n such that $a + b + c < 2n$, it is clearly enough to take a reversed array as a counterexample, because this array puts as many large elements as possible into the first a positions of the array.

Appendix

Did you wonder why is there a modulo operator in the definition of `f3`? The main purpose of this operator was to make life more difficult for solvers who would somehow try to binary search for the optimal counterexample. This tweak with `%97` is there to make sure that the property we examine *is not* monotonous: after the correct counterexample there are still some larger input sizes that *are* sorted properly.

Oh, and the optimal counterexample modulo 97 gives remainder 95, not 96 as you might guess. Yes, we are evil :)

Hard subproblem

The program for this subproblem does two things. First, it generates an awful lot of swaps. This may seem scary, but the composition of all those swaps is simply some permutation of the input. Note that this permutation is the same for all arrays of the same length.

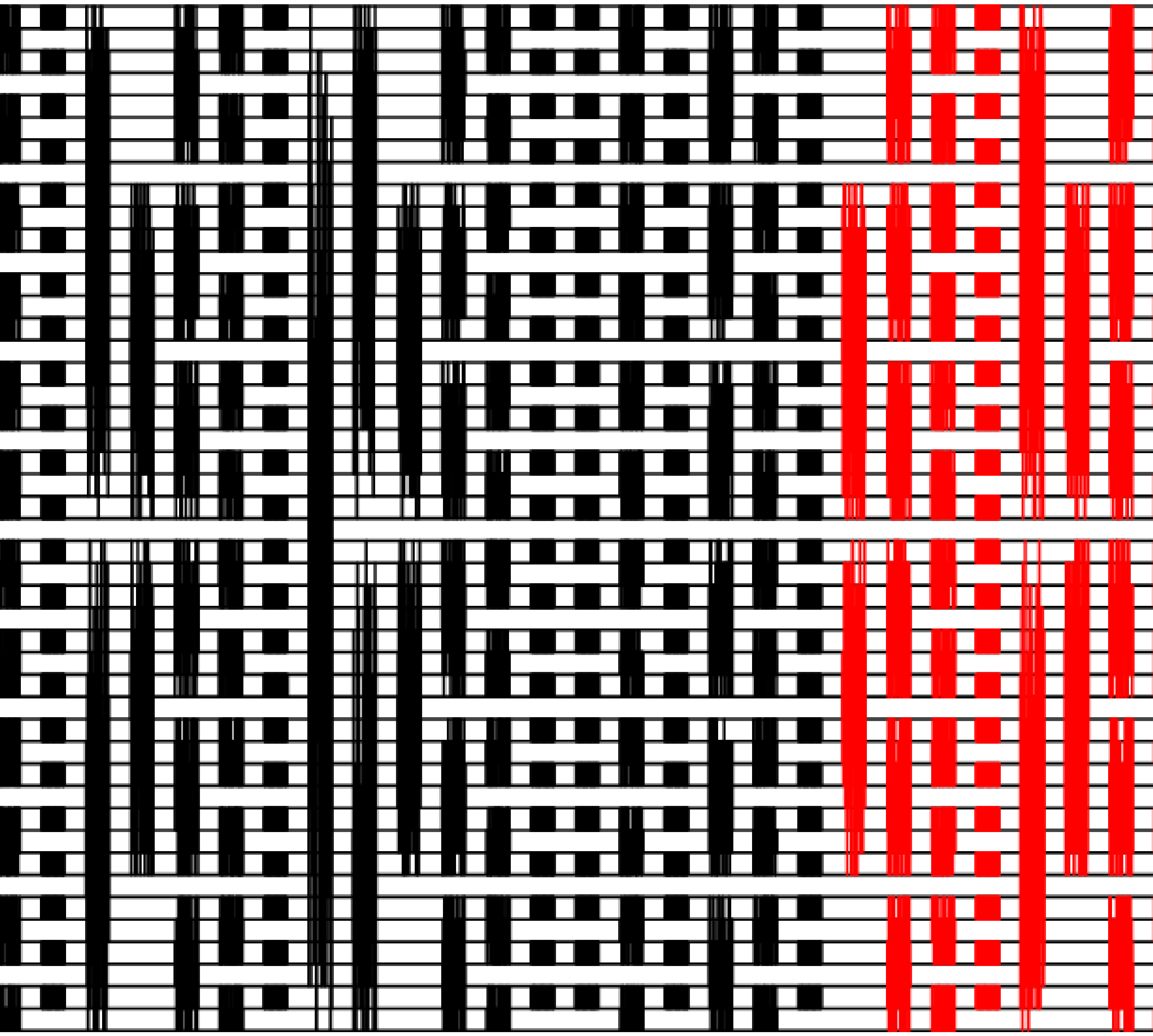
Then, the program generates a slightly smaller number of *conditional swaps*, i.e., instructions of the following form: "given x and y such that $x < y$, if $A[x] > A[y]$, swap $A[x]$ and $A[y]$".

These instructions are called *comparators*, and they are the key ingredient in sorting networks: a simple class of oblivious parallel sorting algorithms.

Evaluating whether a sorting network sorts is a hard problem, but there is a nice algorithmic trick to it called the 0-1 principle: instead of trying all possible *permutations* of the input, it is enough to try all possible vectors of zeros and ones. This brings the time complexity of the test from the trivial $\hat{O}(n!)$ down to $\hat{O}(2^n)$.

Of course, this still means that in practice we can afford testing an algorithm for $n = 20$, or maybe $n = 30$ if we are willing to wait for a while. If we do this to our algorithm, we can verify that it correctly sorts up to $n = 30$. That's nice to know, but not exactly what we wanted.

How can we proceed? Let's try examining our sorting network. We can easily modify the program to *print* all the swaps it performs, and then we can *draw* those comparators. If you do that, you should see something that closely resembles a bitonic sorter. And that might give you an idea how to proceed.



A plot of the swaps (black) and comparators (red) performed by our algorithm for $n = 47$.

In the comparator section of the plot there is only one single part (called "phase l, subphase l in our code) when the top half and bottom half get to mix. Let's call this part"the big round".

We can now divide the "wires" into the top half and the bottom half; for each half separately we can determine the possible outputs of the network before the big round. We will then get a very restricted set of possibilities for the input of the big round. And there, we have to check only two things:

- After the big round, if you have a 1 in the top half and a 0 in the bottom half, you just found that the algorithm does not sort, because it will never swap these two.
- If not, we can take the half that contains both zeros and ones (if any) and simulate only that particular half to verify whether it gets sorted correctly.

Doing the above will allow us to find that the smallest n for which the network fails is $n = 47$.

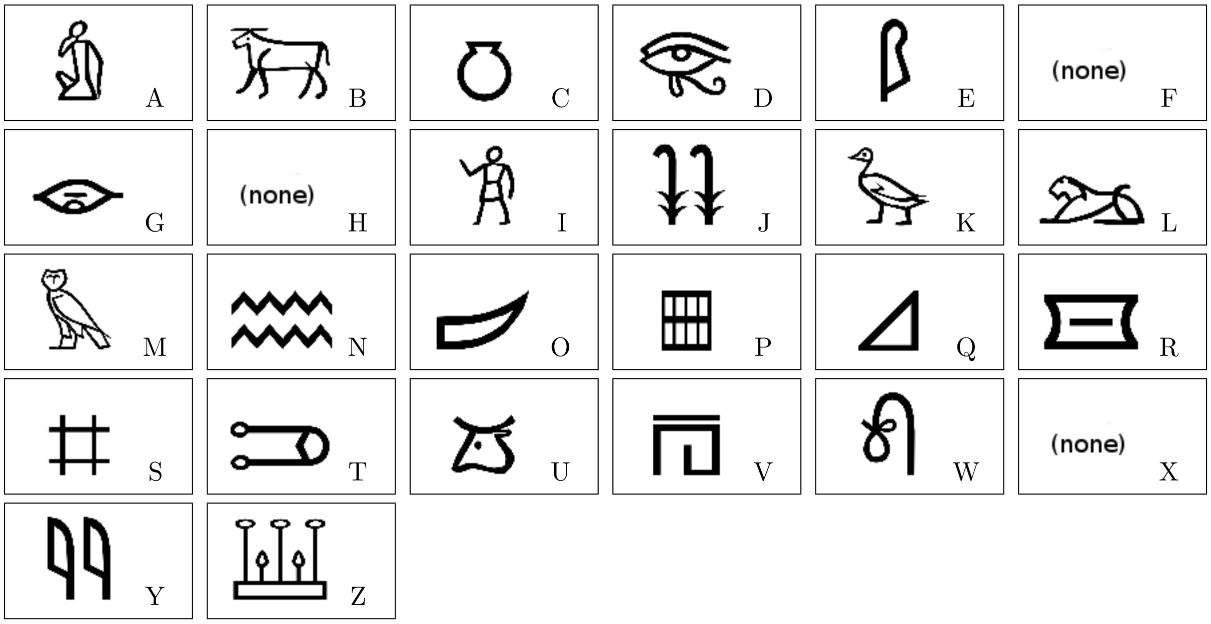
For an alternate solution we may come up with an "educated guess" that a bad input for a network of mostly-random comparators could once again be a reversed array, or something similar to that. We can precompute (once) the permutation done by the first part of our algorithm, and then we can compute its inverse. That tells us the order in which to provide inputs in order to get a reversed array just before the swaps turn into comparators. And then it's just a question of generating lots of inputs that resemble the reversed array, and feeding them into the implementation until you hit one that triggers the bug.

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.


A
B
C
D
E
(none) F
G
(none) H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
(none) X
Y
Z

(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

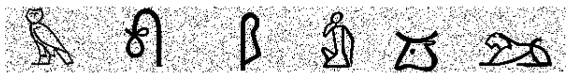
Output a single line for each test case. The line should contain a 6-character lower-case string with the decoded catch-ya. Note that your output should not contain characters "f", "h", or "x".

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)
```

Task authors

Problemsetter:	Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni
Quality assurance:	Vlado 'usamec' Boža

Solution

Easy subproblem

Looking at the pictures of chaptchas from the easy subproblem we can quickly see that they look rather simple – they are just slightly noisy versions of the reference alphabet. Moreover, the captcha letters are clearly separated, each in its own 100x70 block.

This hints of a simple idea — just count have many pixels have the same color in both images. Unfortunately, looks might be deceiving — if you wrote the simple comparison code, you would quickly discover that the accuracy is not very good. The problem is that the images are also slightly shifted by a few pixels. Fortunately, the inputs are small enough that it is possible to check all the reasonable shifts just by a brute force.

If you think you have a solution that's almost correct, there is a neat trick you can do: use your solution's outputs to *generate* a large bitmap of the pictures it *thinks* are correct. Generate a second such bitmap by concatenating the actual inputs. Then, open a graphics editor and place one on top of the other. You will quickly see the differences, and if there are only few of them, you can easily fix the output by hand.

Hard subproblem

Captchas in the hard subproblem pose a more of a challenge. By observing a few of them, we can conclude that the letters are still separated. Unfortunately, unlike the easy subproblem, captchas here are heavily distorted in both x and y directions. Also, the problem statement hints that we do not have to get all of them correct (though, the tolerance is not really big). Finally, we have provided an example set of 200 correctly labelled captchas, i.e., 1200 symbols in total. As you may have guessed, this was quite intentional and the easiest way to solve this problem was to apply machine learning techniques.

There are several different possible machine learning techniques. However, for the domain of image/letter recognition, probably the best one is convolutional neural networks (although SVMs with handcrafted features can also score very high). In our case, we used a fairly standard setup for a convolutional network slightly finetuned to quickly downsample the data. Our layers are

- an 100x70 input layer
- a 7x7 convolution layer with 10 filters
- a maxpool layer pooling 5x5 blocks into a single value
- a 5x5x(10 channels) convolution layer with 10 filters
- a maxpool layer pooling 2x2 blocks into a single value
- the result from the previous layer is 10x7x(10 channels)
- a fully connected 700x70 layer. Note that we use a dropout technique on this layer during the learning step
- a fully connected 70x26 layer with softmax activation function (one-hot encoding of which letter of alphabet it is)

With this setup, the network can learn pretty quickly the given labelled dataset and we were able to obtain >99.5% per-captcha accuracy.

Even the example convolution network from Keras.io works quite well.

But what if I hate machine learning?

Use Amazon Mechanical Turk! Actually, sorry but you can't. This would be a violation of the contest rule "It is strictly forbidden to communicate with people other than your team members about issues that concern solving the problems." But not everything is lost. Indeed, if you were bored during the second half of the contest, guess what, you can be the mechanical turk – with three people in a team, it is measly 1200 letters per person to recognize (out of 800 captchas, only 600 are unlabelled so there are total of 3600 letters and 3 people). And this can go even further – the winning non-machine-learning strategy would go something like this:

1. Realize that humans are much better at "comparison" than at "classification"
2. Sort the images by similarity. This could be something as simple as the "diff" from the easy subtask or something more complex
3. Write a simple program or a webpage which just displays a captcha letter and a few options (sorted by relevance) on which you can click
4. Click through the sequence

The golden middle way

Maybe you come up with something (perhaps a standard neural network, SVM, or other machine learning technique) but you realized that the accuracy is way below what was needed. But don't worry – most of the machine learning tools actually output also the "probability" of the classification being right. So, the solution is to let the machine classify automatically, then take the worst predictions (according to their probability) and classify them manually. This would save you a lot of human work while not requiring expensive fine-tuning of the machine learning algorithms.

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).

Mystic Monastery
Land
Mystic Monastery enters the battlefield tapped.
Add , , or to your mana pool.
When asked how many paths reach enlightenment, the monk kicked a heap of sand. "Count," he smiled, "and then find more grains."

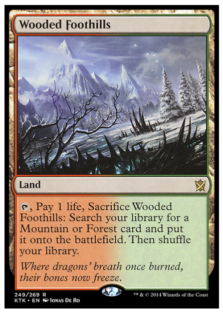
Bayou
Land
Counts as both swamp and forest and is affected by spells that affect either. Tap to add either or to your mana pool.

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


Wooded Foothills
Land
, Pay 1 life, Sacrifice Wooded Foothills: Search your library for a Mountain or Forest card and put it onto the battlefield. Then shuffle your library.
Where dragons' breath once burned, their bones now freeze.

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.

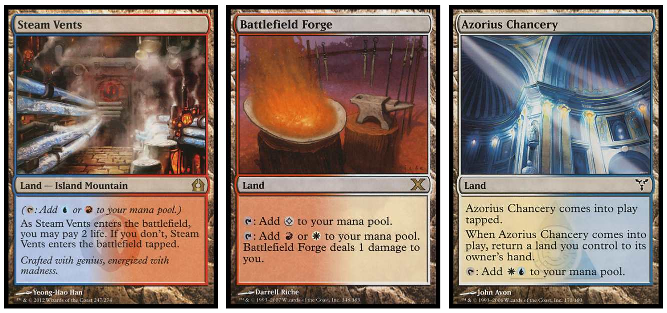
The fetch lands themselves do not have any basic land types. Thus, you cannot use a fetch land to fetch another fetch land.

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.


Steam Vents
Land — Island Mountain
(: Add or to your mana pool.)
As Steam Vents enters the battlefield, you may pay 2 life. If you don't, Steam Vents enters the battlefield tapped.
Crafted with genius, energized with madness.
Yeong-Hao Han
™ & © 2012 Wizards of the Coast 247/274
Battlefield Forge
Land
: Add to your mana pool.
: Add or to your mana pool. Battlefield Forge deals 1 damage to you.
Darrell Riche
™ & © 1993–2007 Wizards of the Coast, Inc. 348/383
Azorius Chancery
Land
Azorius Chancery comes into play tapped.
When Azorius Chancery comes into play, return a land you control to its owner's hand.
: Add to your mana pool.
John Avon
™ & © 1993–2006 Wizards of the Coast, Inc. 170/180

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.)

City of Brass

Land

Whenever City of Brass becomes tapped, it deals 1 damage to you.

: Add one mana of any color to your mana pool.

Illus. Tom Wänerstrand

©1997 Wizards of the Coast, Inc. All rights reserved.

Gemstone Mine

Land

When Gemstone Mine comes into play, put three mining counters on it.

, Remove a mining counter from Gemstone Mine: Add one mana of any color to your mana pool. If there are no mining counters on Gemstone Mine, bury it.

Illus. Brom

©1997 Wizards of the Coast, Inc. All rights reserved.

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.

Urza's Mine

Land

Tap to add 1 colorless mana to your mana pool. If you have Urza's Mine, Urza's Tower, and Urza's Power Plant in play at the same time, tap to add 2 colorless mana to your mana pool.

Mines became common as cities during the days of the artificers.

Illus. © Anson Maddocks

Urza's Power Plant

Land

Tap to add 1 colorless mana to your mana pool. If you have Urza's Mine, Urza's Tower, and Urza's Power Plant in play at the same time, tap to add 2 colorless mana to your mana pool.

Artifact construction required immense resources.

Illus. © Mark Tedin

Urza's Tower

Land

Tap to add 1 colorless mana to your mana pool. If you have Urza's Mine, Urza's Tower, and Urza's Power Plant in play at the same time, tap to add 3 colorless mana to your mana pool.

Urza always put Tocasia's lessons on resource-gathering to effective use.

Illus. © Mark Poole

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.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	true men need no stinkin' quality assurance

Solution

To start us off, here is a fun fact about this task: out of the 2111 tests, 1295 were hand-crafted and all implementations were checked against human-provided answers for those :)

Our solution will begin with a very simple first phase: tap everything where you don't have to make a decision. In other words, tap all basic lands (for 1 mana each), all bounce lands (for 2 mana each), and all Tron pieces (for some amount of colorless mana). Subtract the mana produced this way from the requirements. From this point on, we can ignore the lands mentioned above. This leaves us with roughly one half of the original card pool.

Easy subproblem without pain lands

If we don't have any fetch lands on the battlefield, we get to ignore the entire library. Only the lands on the battlefield matter.

For now, let's ignore the lands that can hurt us, and let's just focus on making the right decisions. How can we tell which land should produce which mana color?

There are various ways to approach this question, for instance, a lot of case analysis. Our preferred way is one that uses a more advanced tool but has no special cases: network flow. Indeed, network flow is a good way to model our situation.

We will have a node for each land on the battlefield. For each of these nodes, there is an edge from the source to the node with capacity 1. This means that this land can generate at most 1 mana. Next, we will have 7 nodes: one for each mana type (5 colors + colorless) and one for generic mana. Each land-node will be connected to those type-nodes that correspond to the colors that can be produced by that particular land. Additionally, each of the six mana type nodes has an outgoing edge leading into the generic mana node. This edge (with infinite capacity) represents that any mana can be used to pay the generic mana requirement. Finally, each of the seven mana-type-nodes has an edge to the sink, and the capacity of this edge is the required amount of mana of that type.

Clearly, flows in this network correspond to all possible ways to tap some of the lands we have for mana and to use that mana to pay the required amounts. We can cast the spell if and only if the maximum flow in our network saturates all the edges that lead into the sink.

Easy subproblem with pain lands

We are almost done with the easy subproblem. All that remains are the pain lands, and City of Brass. These can sometimes hurt us. How do we incorporate them into the previous solution?

First of all, we may tap some of the pain lands for colorless mana without loss of life. We will try all possibilities for the number of such lands and evaluate each of those separately.

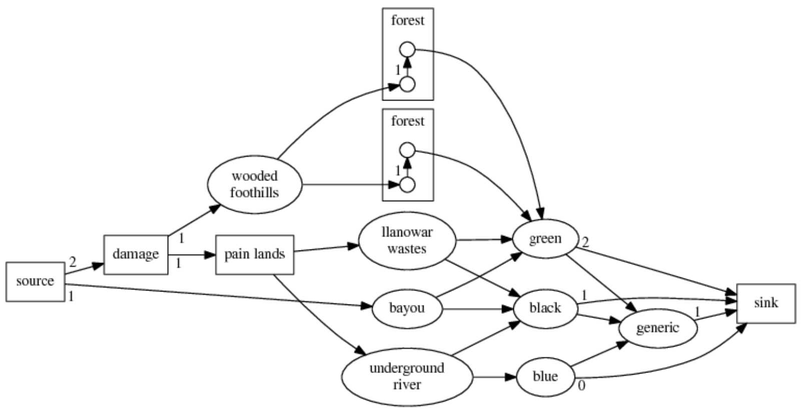
We will now add two more nodes to our flow network: a node D that represents all mana that gives us damage when produced, and a node P that only represents pain lands. The capacity of the edge from source to D will be one less than our current life total, as we cannot produce more painful mana than that amount. Each City of Brass will have an edge from D to its own node with capacity 1. Additionally, there will be an edge from D to P and the capacity of this edge will be the number of pain lands we *didn't* tap for colorless mana. Finally, we'll have edges from P to each pain land, and from each pain land to the colors it can produce.

Hard subproblem: fetching some shock lands

Now that we are done with the easy subproblem, it's fetching time. The main issue here is the extra damage dealt by fetched shock lands. How to deal with these?

The thing that will save us here is the amount of life we have. Fetching an untapped shock land costs 3 life. The highest allowed life total is 20. Hence we can fetch at most 6 shock lands. With 6 fetch lands on the battlefield, this leaves us with at most 28 shock lands in the library, and therefore at most $\binom{28}{6} < 400,000$ ways to choose which specific 6 lands to fetch. (Actually, it's much fewer, as there are only 10 distinct shock lands.) Again, we can afford to try all of these and see whether any of those possibilities gives us a working solution.

Once we choose a specific set of shock lands to fetch, we will subtract 2 times their count from our life total. We will now add a new part to our flow network. The node D that represents damage taken will now also have edges of capacity 1 into nodes that correspond to fetch lands. There will be new nodes for fetchable lands in the library. Each fetch land will have edges to lands it can fetch. Each fetchable land node will have capacity 1 (to model that it can only be fetched once, even if multiple fetch lands allow fetching it). All fetchable land nodes have edges into mana type nodes, just like any other lands.



Above: a sample subset of the flow graph.

An even easier alternate solution

For a slightly alternate solution with even less cases and an even more advanced algorithm, all loss of life can be modeled as costs, and then we can find the **minimum cost maximum flow** and check that both the size is optimal and the cost is lower than your starting life total. With this approach, Tron is actually the only thing you need to special-case.