Problem A King's Heir

Input file:	`king.in`
Output file:	`king.out`
Time limit:	2 seconds
Memory limit:	256 megabytes

The king is dead, long live the king! After the sudden death of the king Fert XIII the people of the Flatland Kingdom are going to welcome the new king. Unfortunately, there is a problem, Fert has too many sons.

Actually, he has n sons and he loved each new son more than all of his previous sons. Well, probably he just stopped loving his sons because of their bad behavior. Anyway, after the new son was born Fert made the new testament that declared that the newly born son would be the heir.

However, there is a problem. Only the king's son who is at least 18 years old at the moment of the king's death can become a new king. Now the ministers of the government are trying to find the correct new king, but they seem to fail. Help them!

Input

The first line of the input contains three integers: d, m and y — the day, the month and the year of the king's death, d is from 1 to 31, m is from 1 to 12, y is from 1 to 9999. It is guaranteed that there exists day d in month m, all months have the same number of days in Flatland as in our country, except that Flatland calendar doesn't have leap years, so February (month 2) always has 28 days.

The second line contains n ($1 \le n \le 100$) — the number of king's sons. The following n lines contain three integers each d_i, m_i and y_i and specify the birth dates of king's sons. All dates are correct and no son is born after or on the day of king's death. The king had no twins, so no two sons were born on the same date.

Output

Output one integer — the number of the son that would become the king, or -1 if none of them is at least 18 years old. The sons are numbered from 1 to n in order they are described in the input. The youngest son who is at least 18 years old at the moment of the king's death would become the king. If the son has his 18th birthday exactly on the day of the king's death, he can become a king.

Examples

`king.in`	`king.out`

22 10 2016
7
28 2 1999
22 7 1995
21 10 1998
23 10 1998
3 9 2000
1 4 2013
17 12 2004
	3

22 10 2016
1
28 2 1999
	-1

Problem B Bricks

You are given a sequence of white (W) and black (B) bricks. The goal is to partition it into some number of non-empty, contiguous blocks, each one having the same ratio of white and black bricks.

Of course one can always "partition" the sequence into one single block (which is not very interesting). We want, however, to have as many blocks as possible. Consider for example the following sequences and its partitions:

- BWWWBB = BW + WWBB (ratio 1:1),
- WWWBBBWWWWWWWWWB = WWWB + BBWWWWWW + WWWB (ratio 3:1).

Note that both of these partitions are optimal with respect to the number of blocks.

Input

The first line of input contains the number of test cases T. The descriptions of the test cases follow:

Each test case starts with one line containing an integer n ($1 \leqslant n \leqslant 10^5$) which is the length of the description of a sequence. Each of the following n lines consists of an integer k ($1 \leqslant k \leqslant 10^9$) and one of the characters `W` or `B`, meaning that k bricks of the given color follow next in the sequence. It is guaranteed that the total length of the brick sequence does not exceed 10^9.

Output

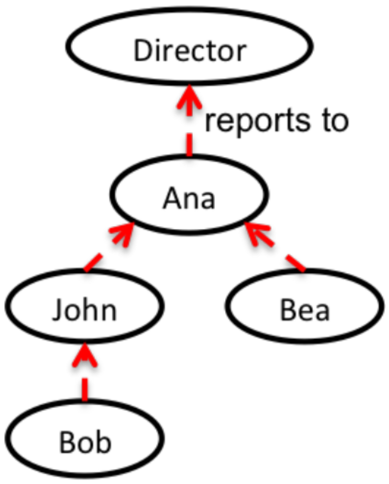
For each test case, output a single line containing the largest possible number of blocks.

Example

For an example input	the correct answer is:

```
3
3
1 B
3 W
2 B
4
3 W
3 B
9 W
1 B
2
2 W
3 W
```
	```
	2
	3
	5
	```

Problem C Performance Review



Employee performance reviews are a necessary evil in any company. In a performance review, employees give written feedback about each other on the work done recently. This feedback is passed up to their managers which then decide promotions based on the feedback received.

Maria is in charge of the performance review system in the engineering division of a famous company. The division follows a typical structure. Each employee (except the engineering director) reports to one manager and every employee reports directly or indirectly to the director.

Having the managers assessing the performance of their direct reports has not worked very well. After thorough research, Maria came up with a new performance review system. The main idea is to complement the existing corporate structure with a technical rank for each employee. An employee should give feedback only about subordinates with lower technical level.

Hence, the performance review will work as follows. Employees prepare a summary of their work, estimate how much time it takes to review it, and then request their superiors with higher technical rank to review their work.

Maria is very proud of this new system, but she is unsure if it will be feasible in practice. She wonders how much time each employee will waste writing reviews. Can you help her out?

Task

Given the corporate structure of the engineering division, determine how much time each employee will spend writing performance reviews.

Input

The first line of input has one integer E, the number of employees, who are conveniently numbered between 1 and E. The next E lines describe all the employees, starting at employee 1 until employee E. Each line contains three space-separated integers m_i r_i t_i, the manager, the technical rank and the expected time to perform the review of employee i. The engineering director has no manager, represented with $m_i = -1$. The other employees have m_i between 1 and E.

Constraints

$1 \leq E \leq 100\,000$	Number of employees
$1 \leq r_i \leq 100\,000$	Technical rank of each employee
$1 \leq t_i \leq 100\,000$	Expected time to perform each review

Output

The output contains E lines. Line i has the time employee i will spend writing reviews.

Sample Input

```
5
4 4 80
1 1 40
-1 10 60
3 5 50
4 8 70
```

Sample Output

```
40
0
240
120
0
```

Problem D Berodoskar Development

Input file:	`stdin`
Output file:	`stdout`
Time limit:	1 second
Memory limit:	64 megabytes

Not far away from Berland there is a small stony isle in the ocean. The name of the isle is Berodoskar and it is an ideal place for a military base. The only problem is lack of fresh water, but such a negligible problem will never stop Berl III, the King of Berland. He has stated that there would be exactly two water-storage reservoirs with desalinated water. The best way to organize reservoirs is to use existing craters located on the isle and connect them with the ocean by a water-pipe. And you are the person responsible for the water-storage system creation.

You have a map of Berodoskar on which the isle is represented as a rectangle divided into unit squares by horizontal and vertical lines. Each unit square is marked with either '`X`' character if it contains a part of crater or with '`.`' character if there is no crater on this square. If two squares marked with '`X`' have the common edge they are considered as parts of one crater. There are no squares marked with '`X`' connected with the ocean. The whole area around the square is the ocean.

The water-pipe you are to create should connect any two of craters with the ocean. It is allowed to connect more than two craters, but at least two craters should be connected. The water-pipe will consist of segments connected by edge and each segment will occupy the whole non-crater unit square. A pipe is connected with a crater if they have a common edge. You want to minimize number of squares involved into water-storage system creation. It is possible that two independent water-pipes are more efficient than one branched (see examples).

Input

The first line contains two integer numbers N and M — number of unit squares in vertical and horizontal directions ($3 \le N, M \le 15$). After that N lines follow describing the isle. Each line contains exactly M characters '`X`' or '`.`'. It is guaranteed that there are at least two craters reachable from the ocean.

Output

Output N lines with M characters in each — an isle map is in the same format as in the input, but with the water-pipe indicated. Squares with the water-pipe should be marked with '`*`' instead of '`.`'. If there are several solutions with the same minimal number of squares used, output any one of them.

Examples

stdin	stdout
`5 7`	`.......`
`.......`	`.......`
`.......`	`**X....`
`..X....`	`.....X*`
`.....X.`	`.......`
`.......`	
`10 13`	`........*....`
`.............`	`........*....`
`.............`	`..XXXXX**....`
`..XXXXX......`	`..X..X..*....`
`..X..X.......`	`..X.X.X.X....`
`..X.X.X.X....`	`..X...X......`
`..X...X......`	`..XXXXX......`
`..XXXXX......`	`.............`
`.............`	`.............`
`.............`	`.............`
`.............`	

Note: in second example you are not allowed to connect a crater with the ocean via another crater.

Problem E Sort It!

Input file:	*standard input*
Output file:	*standard output*
Time limit:	1.5 seconds
Memory limit:	512 mebibytes

You are given a permutation of length n: p_1, p_2, ..., p_n. Consider some array of length n consisting of integers between 1 and n (equal elements are allowed). Let us transform the array in the following manner: at first, let us take all elements equal to p_1 from it and write them on a piece of paper (if there are no such elements, just do not write anything). Then write all elements equal to p_2, then equal to p_3 and so on, finishing by all elements equal to p_n, thus obtaining a new array of length n. For example, if the permutation is 2 1 3 and the array is 2 3 2, the resulting array will be 2 2 3. If after this transformation we get a sorted array, let us call the original array *sortable by p*. Calculate the total number of arrays that are sortable by p.

As the answer can be very large, output it modulo $10^9 + 7$.

Input

The first line contains a single integer n ($1 \le n \le 2000$): the length of the permutation. The second line contains n distinct integers p_1, p_2, $\ldots$, p_n ($1 \le p_i \le n$): the permutation itself.

Output

Output a single integer: the answer to the problem modulo $10^9 + 7$.

Examples

standard input	standard output
2 2 1	2
3 2 1 3	15

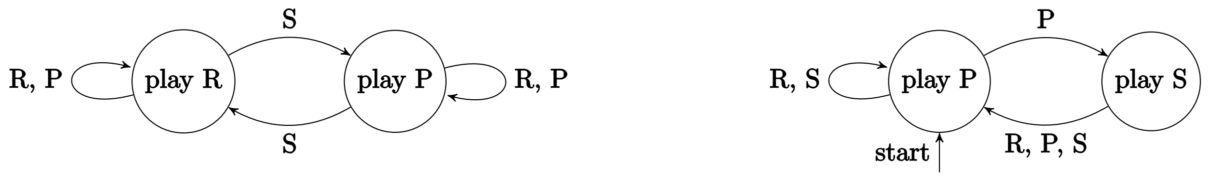
Problem F Epic Win!

Input file:	`epic.in`
Output file:	`epic.out`

A game of rock-paper-scissors is played by two players who simultaneously show out their moves: *Rock*, *Paper*, or *Scissors*. If their moves are the same, it's a draw. Otherwise, *Rock* beats *Scissors*, *Paper* beats *Rock*, and *Scissors* beat *Paper*.

The described procedure can be repeated many times. In this problem, two Finite State Machines (FSMs) will compete in a series of rounds. (Formally speaking, by FSMs we mean Moore machines in this problem.)

An FSM for playing rock-paper-scissors has finitely many states. Each state is described by the following: what move the FSM will make in the upcoming round, and what will be the new state in case of its opponent playing *Rock*, *Paper*, and *Scissors*.



Fortunately, you know your opponent FSM — the entire scheme except for one thing: you *do not know* the initial state of that FSM.

Your task is to design your own FSM to fight the given one. Your FSM must beat the opponent in at least 99% of the first 1 billion rounds. That's what we call an epic win!

Input

The input file contains a description of the opponent FSM in the following format.

The first line contains an integer n ($1 \le n \le 100$) — the number of states in the FSM. States are numbered from 1 to n. Each of the following n lines contains a description of the state: a character c_i denoting the move made by FSM and integers r_i, p_i, s_i denoting the next state in case of seeing *Rock*, *Paper*, or *Scissors* respectively (c_i can be "R", "P", or "S"; $1 \le r_i, p_i, s_i \le n$).

Output

Write to the output the description of your FSM in the same format.

The initial state of your FSM is the first state.

The number of states may not exceed 50 000.

Sample input and output

`epic.in`	`epic.out`

`2`
`R 1 1 2`
`P 2 2 1`
	`2`
	`P 1 2 1`
	`S 1 1 1`

The picture in the problem statement illustrates the opponent FSM given in the above sample input and a possible solution of yours given in the sample output.

Opponent FSM keeps playing *Rock* or *Paper* (depending on its initial state) until it sees *Scissors* — seeing *Scissors* triggers a change in its behaviour.

One way to beat such FSM is to play *Paper*. If your opponent keeps playing *Rock*, just continue playing *Paper* and thus win. If the opponent FSM is playing *Paper*, trigger it to playing *Rock* by playing *Scissors* once, and then it'll keep playing *Rock* and you'll keep beating it with your *Paper*.