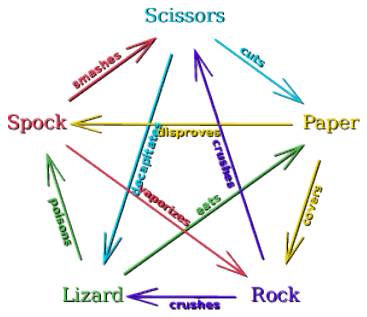
Against a rock play Spock

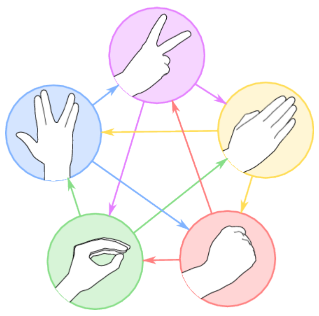
For many people, a Rock-Paper-Scissors game is too simple and thus boring. As IPSC should not be boring, we will be playing an extended version called Rock-Paper-Scissors-Lizard-Spock.

The game contains five gestures: *rock*, *paper*, *scissors*, *lizard*, and *Spock*. Lizard is usually formed by a hand closed to a sock-puppet-like mouth and Spock is formed by the Star Trek Vulcan salute. In each round of the game, both players simultaneously show one of the five gestures.

Each round is evaluated as follows: If the two gestures are equal, the round ends in a tie (nobody wins). Otherwise look at the table below and find the row that contains both gestures shown. The player showing the gesture in the left column wins. To the right of the table you can see figures from Wikipedia that contain the same information.

scissors	cut	paper
paper	covers	rock
rock	crushes	lizard
lizard	poisons	Spock
Spock	smashes	scissors
scissors	decapitate	lizard
lizard	eats	paper
paper	disproves	Spock
Spock	vaporizes	rock
rock	breaks	scissors





Problem specification

The input contains the sequence of gestures we are going to play in the following rounds. For each gesture, output a line containing the name of the gesture you are going to play against it.

To solve the **easy subproblem**, your task is to win every round of the game.

To solve the **hard subproblem**, your task is the same. But you have to do it without using the same gesture twice in a row.

For instance, suppose that in round 1 we played `Spock` and you played `paper`. If we play `rock` in round 2, you can only win by playing `Spock`, you may not play `paper` again. In round 3, you then cannot play `Spock`, but you may play `paper` again, if you wish.

Input specification

The first line contains a positive integer r: the number of rounds we will play.

Each of the next r lines contains one of the five strings `rock`, `paper`, `scissors`, `lizard`, and `Spock`.

Output specification

For each round, output a single line containing your gesture – again, one of the strings `rock`, `paper`, `scissors`, `lizard`, and `Spock`. (Note the uppercase 'S' in Spock.)

Example

input

```
3
Spock
rock
rock
```

output

```
paper
Spock
paper
```

BFS killer

Mark just solved a traditional programming task: finding the shortest path in a maze. The map of his maze is a rectangular grid with r rows and c columns ($1 \leq r, c \leq 600$). Some of the cells contain walls (denoted by #, ASCII code 35), others are empty (denoted by `.`, ASCII code 46). There are two special empty cells: one starting point (denoted by `S`) and one treasure (denoted by `T`).

In each step the player may only move in one of the four cardinal directions. In other words, each two subsequent cells on his path have to share a common side. The treasure is always reachable from the starting point. The player is not allowed to leave the maze or enter a cell with a wall.

Mark's task was to find the length of shortest path from the starting point to the treasure. He solved this task using breadth-first search (BFS). His C++ solution is given in the input file. Below you can find the solution in pseudocode. However, Mark made a common bug in his implementation of the FIFO (first in, first out) queue – he is using a circular array of size k. Show him that his queue is too short!

Problem specification

Find a map of a maze such that if Mark's program is executed on your maze, the queue size will exceed k at some moment. In the easy subproblem, $k = 5\,000$. In the hard subproblem, $k = 15\,000$. (The queue size is the number of cells it contains.) Your maze must satisfy all the constraints given above.

```
define process_cell(r,c):
    if (r,c) is not outside the maze:
        if cell at (r,c) is not a wall:
            if (r,c) is unvisited:
                if treasure is at (r,c), terminate the entire search
                put (r,c) into Q and mark (r,c) as visited

define BFS(startr,startc):
    initialize an empty queue Q
    put (startr,startc) into Q and mark (startr,startc) as visited
    while Q is not empty:
        get (r,c) from the queue
        for each (nr,nc) in (r,c+1), (r-1,c), (r,c-1), (r+1,c):
            process_cell(nr,nc)
```

Input specification

The input file for this problem contains a reference C++ implementation of Mark's solution. The variable `size` in this implementation must exceed k.

Output specification

On the first line output two integers separated by a single space: number of rows r and number of columns c. Then output r rows with c characters in each – the map of your maze.

Example

A syntactically valid submission with maximum queue size 4.

```
3 4
....
.S#.
..#T
```

Candy for each guess

Guess the number is a very simple game. Given is a positive integer n. The first player picks a number x from the set $\{0, 1, \ldots, n-1\}$, writes it on a card and places the card into an envelope. The second player then makes guesses. Each guess has the form of an integer g. The first player answers "too low" if $g < x$, "too high" if $g > x$, or "correct" if $g = x$. For each guess the first player gets a candy from the second player. The game ends when the second player gets the answer "correct".

Hannah discovered this game when she was eight and she started playing it with her younger brother Gunnar. Hannah always picked the number and Gunnar was always guessing.

When they were young, Hannah had it easy. Gunnar only knew a few deterministic guessing strategies, such as "binary search". In each game he would use one of the strategies he knew, picked uniformly at random. Hannah knew all Gunnar's strategies, and she always picked a number that maximized the expected number of candies she would get.

As Gunnar grew up, the games became much more involved. Eventually, both he and Hannah became perfect in playing the game.

Problem specification

In the easy input each test case contains n and a list of Gunnar's strategies. Find the expected number of candies Hannah will get per game if she plays optimally. Also, find one optimal strategy for Hannah.

In the hard input each test case contains n. Find the expected number of candies Hannah will get per game if both players play optimally (i.e., try to maximize/minimize the expected number of candies paid per game). Also, find one optimal strategy for Hannah and one optimal strategy for Gunnar.

Strategy format specification

We will only consider strategies where Gunnar never makes a guess that reveals no new information. (All strategies in the input for the easy data set will be such, and all strategies you will provide in the output for the hard data set must be such.)

Each such strategy can be encoded into a sequence of integers: First, you write down the first query he will make. Then you recursively write the strategy used if the guess was too large, and finally the strategy if the guess was too small. (I.e., the encoding is the pre-order traversal of the decision tree.)

For example, for $n = 6$ the strategy $(2, 0, 1, 4, 3, 5)$ means that in the first round Gunnar will guess 2. If the correct number is smaller, he will then guess 0 (and afterwards 1 if necessary), otherwise he will guess 4 (and then possibly 3 or 5).

Input specification – easy subproblem

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 several lines. The first line contains the integer n (up to 16) and an integer s (up to 100) giving the number of strategies Gunnar knows. Each of the next s lines describes one strategy in the format specified above.

Output specification – easy subproblem

For each test case, output two lines. The first line must contain a fraction p/q giving the exact expected number of candies per game Hannah will gain by playing optimally. This fraction must be reduced (i.e., p and q must be coprime).

The second line must contain one optimal strategy for Hannah – a sequence of integers $h_0, \dots, h_{n-1}$ giving the relative probabilities of Hannah's choices. That is, Hannah should choose the number i with probability $h_i/(\sum h_i)$. The numbers h_i must not exceed 10^9.

Example – easy subproblem

input

```
1

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

output

```
13/3
0 0 0 0 1 0 2
```

Gunnar picks one of three strategies: the first is a linear search, the second is a binary search, and the third is ad-hoc. One optimal strategy for Hannah is to pick the number 4 in 1/3 of all games and the number 6 in the remaining 2/3 of all games. This strategy guarantees her to get 13/3 of a candy per game, and there is no better strategy.

Input specification – hard subproblem

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 with the integer n.

Output specification – hard subproblem

For each test case, output several lines. The first line must contain a fraction p/q giving the exact expected number of candies per game Hannah will gain if both players play optimally. This fraction must be reduced (i.e., p and q must be coprime).

The second line must contain one optimal strategy for Hannah – a sequence of integers $h_0, \dots, h_{n-1}$ giving the relative probabilities of Hannah's choices. That is, Hannah should choose the number i with probability $h_i/(\sum h_i)$. The numbers h_i must not exceed 10^9.

The third line must contain a positive integer s not exceeding 1000 – the number of deterministic strategies Gunnar will be using with a non-zero probability. Each of the remaining s lines should contain $n+2$ space-separated tokens: the encoding of a strategy, a colon, and a positive integer g_i. The numbers g_i are relative probabilities of Gunnar's choices; they must not exceed 10^9.

Example – hard subproblem

input

```
1

2
```

output

```
3/2
1 1
2
0 1 : 1
1 0 : 1
```

The optimal strategies are "pick 0 or 1 with equal probability" and "start by guessing 0 or 1 with equal probability". In 50% of games Gunnar will succeed with his first guess, in the other 50% he will need a second guess.

Clearly, Hannah's strategy guarantees she can expect to gain at least 3/2 candies per game, and Gunnar's strategy guarantees she can expect at most 3/2. Hence both strategies are optimal.

Divide the rectangle

Given is a rectangle consisting of $r \times c$ unit squares. One of these squares, in row r_r and column c_r, is colored red. A different square, in row r_b and column c_b, is colored blue.

Easy subproblem: Color each of the remaining squares red or blue in such a way that the red part and the blue part have the same shape. (That is, the set of blue squares can be obtained from the set of red squares by using shifts, rotations and reflections.)

Hard subproblem: Color each of the remaining squares red or blue in such a way that the red part has and the blue part have the same shape. Additionally, the red part (and therefore also the blue part) must be connected.

A set of squares S is connected if it is possible to travel between any two of them without leaving S, using horizontal and vertical steps only.

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 three lines. The first line contains the dimensions r and c. The second line contains the coordinates r_r and c_r. The third line contains the coordinates r_b and c_b. You may assume that $1 \leq r_r, r_b \leq r \leq 100$, $1 \leq c_r, c_b \leq c \leq 100$, and $(r_r, c_r) \neq (r_b, c_b)$.

Output specification

For each test case, output r lines containing c characters each, each of the characters being `R` (red) or `B` (blue). If there are multiple solutions, pick any of them. If there is no solution, output one line with the string "`IMPOSSIBLE`". You may output empty lines between test cases.

Example

input

```
3

5 5
1 1
3 4

4 6
1 1
4 6

4 6
1 1
1 2
```

output

```
IMPOSSIBLE

RRRRRR
RRRRRB
RBBBBB
BBBBBB

RBBBBB
RBRRRB
RBBBRB
RRRRRB
```

This output would be correct in both subtasks.

Elementary math

Nowadays, people use calculators and computers for almost everything. However, we are now working on the Intercontinental Plan for Sabotage of Calculators and soon all calculators will be destroyed. You better dust off you old skills and get a pencil and some paper.

Problem specification

Your task will be to compute the square root of a given number by hand (or if you insist, with help of your computer). In case you have never learned how to do this essential mathematical operation, we provide detailed instructions:

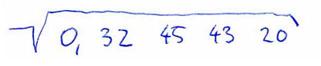
At the beginning, separate the number into pairs of digits, starting at decimal point. In case there is odd number of digits after the decimal point, add single zero digit at the end of the number. In case there is odd number of digits before the decimal point, the first "pair" will consist of only one digit. For example, if the number is 12345.678, we will split the number into pairs "1", "23", "45", "67" and "80". The square-root algorithm will compute the result in several iterations, each iteration taking into account one new pair of digits and appending one digit to the result.

In the first iteration you should estimate the square root of the first pair of digits. That is, you should find the largest integer x such that $x^2 \leq first\_pair$. The value of x will be the first digit of the result. Now you should compute value of x^2 and subtract it from the value of the first pair.

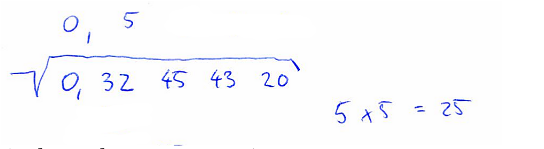
Each of the remaining iterations can be computed as follows: Drop the next pair of digits next to the result of the previous subtraction. Denote the resulting number as t. You need to estimate next digit of the result. For this write "$y\_ \times \_ =$" to some temporary place, where y is twice the value of the result computed so far. Now replace both "_"s with the same digit i such that the result will be less or equal than t and i will be as large as possible. The value of i will be the next digit of our result. After i is found, the result of "$yi \times i$" should be subtracted from the value of t. This step ends the current iteration.

You may see the whole computation in little steps on the following pictures:

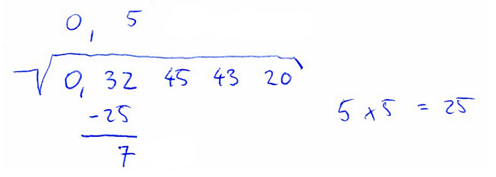
1. create pairs



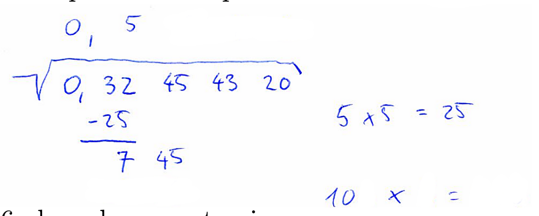
2. estimate first digit



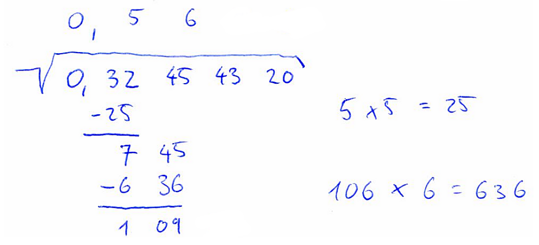
3. subtract



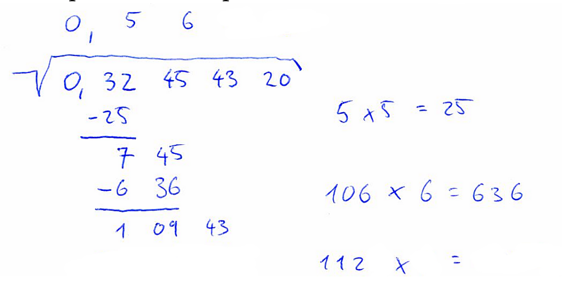
4. drop down next pair



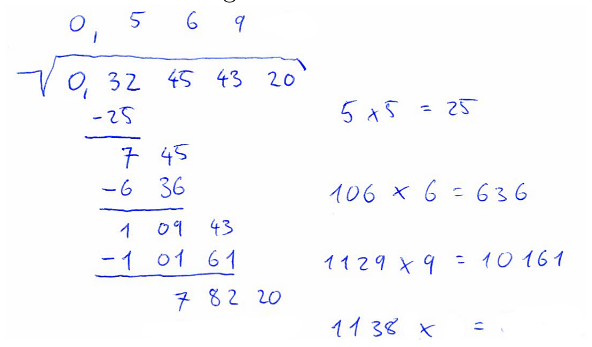
5. estimate next digit



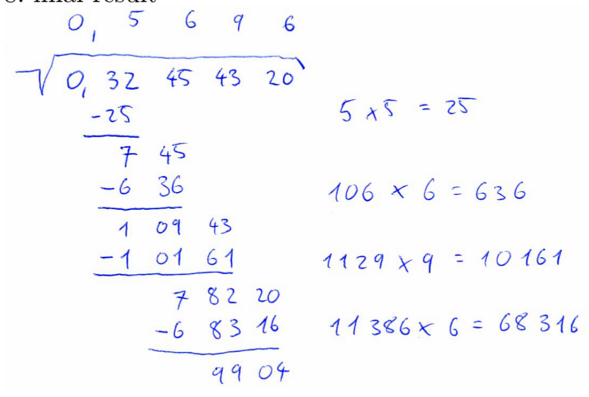
6. drop down next pair



7. estimate next digit



8. final result



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 with a single real number containing up to 16 digits. You may assume that the number does not contain unnecessary leading zeroes. (The "0" in "0.12" is considered neccessary and it will always be present if such a number occurs.)

Output specification

Output will consist of outputs of individual test cases. Put a blank line between consecutive test cases.

The output of one test case is a character matrix consisting of r rows and c columns. The numbers r and c should be as small as possible. Blank spaces in the matrix should contain the character "." (dot).

Basically, the output matrix should look like the result of the handwritten computation shown in the problem statement. It should consist of the following parts: the input value, a square root sign, the result of calculation, and the intermediate calculations needed to obtain the result.

The square root sign should consist of character "_" (underscore), then characters "\/" (backslash and forward slash) on the line below, then as many dashes as necessary "-" and the root sign should be finalized by a backtick "`" on the line below. The space under the dashes should be filled with the input value. The input value should be written as two-digit pairs separated with one blank character. (Remember that the first pair may consist of only one digit.) There should also be exactly one blank space on the left and one on the right of the formatted input value.

If the input value contains a decimal sign (a dot in the input file), it must also be present in the output. First, format the digit pairs and the square root sign as described above. Then, replace the appropriate blank space by a comma "," character.

The result should be on the line above the square root sign and each digit of the result should be right-aligned with the corresponding input pair. In case the input value contains a decimal sign, the result must contain a comma in the same column as the input value does.

The intermediate calculations may be divided into two groups: subtractions and multiplications. Each multiplication should be in format "`12.x.3.=.456`" and should be left-aligned on the first position after the end of the square root sign. Each subtraction consists of the result of previous subtraction, the new dropped pair, the subtracted value, an underline, and the result of subtraction. The dropped pair should be aligned with its occurrence in the input. The subtracted value should consist of the character "-" (minus/dash) immediately followed by the value. The value should be split into pairs of digits and it should be right-aligned with the previous row. The underline should consist of several dashes, right aligned with the current computation. The underline must be as short as possible, but long enough so that

that both above operands (including the minus sign, but excluding any leading zeroes the operands may have) will be completely above the underline. Finally, the result of subtraction should be right-aligned with the whole computation and should be split into pairs of digits separated by blanks.

Because computing square root of leading zeroes is useless (square root of zero is zero), the computation shown in your output should start with the first pair of digits that is not all zeroes. Note hovewer, that the input and the result must contain all the necessary zeroes. In particular this means that for the input "0" you should only output the square root sign with a zero below and a zero above it.

Example

input

```
3

17

3.14159

0.005
```

output

```
.....4............
_..----...........
.\/.17.‘..........
...-16..4.x.4.=.16
...---............
.....1............

....1,.7..7..2.................
_..------------................
.\/.3,14.15.90.‘...............
...-1...........1.x.1.=.1......
...--..........................
....2.14.......................
...-1.89........27.x.7.=.189...
...-----.......................
......25.15....................
.....-24.29.....347.x.7.=.2429.
.....------....................
.........86.90.................
........-70.84..3542.x.2.=.7084
........------.................
.........16.06.................

....0,.0..7............
_..---------...........
.\/.0,00.50.‘..........
........-49..7.x.7.=.49
........---............
..........1............
```

Flipping coins

A group of n friends (numbered 1 to n) plays the following game: Each of them takes a fair coin, flips it (getting a head or a tail with equal probability), and without looking at it glues it to her forehead. Then everyone is allowed to look at all other players, but all communication is forbidden during this phase. Finally, each of the players takes a piece of paper and writes down one of "head", "tail", and "pass".

A player is *correct* if she writes down the coin side that is visible on her forehead. A player is *wrong* if she writes down the opposite side. A player that writes "pass" is neither right nor wrong. The players win the game if *at least one of them* is right and at the same time *none of them* is wrong.

Problem specification

One possible strategy is that one of the players guesses "head" and all others pass. With this strategy the players win in exactly 50% of all possible cases. Is there a better strategy for $n = 3$? If yes, find one.

Then, as the harder part of this problem, find optimal strategies for all n from 1 to 8, inclusive.

Strategy format specification

It can be proved that even if we allowed randomized strategies for all players, there would be an optimal deterministic strategy. Hence we will only consider deterministic strategies.

From player i's point of view, there are 2^{n-1} different outcomes. Each outcome can be described by a string of $n - 1$ `H`s (heads) and `T`s (tails) – for each player from 1 to n except for i, write her coin side. To write down the strategy for player i, order all outcomes alphabetically, and for each of them write down `H`, `T`, or `P` (pass) – the action player i should take when she sees the corresponding outcome.

The strategy for the entire game consists of n lines, with line i containing the strategy for player i.

Input specification

There is no input.

Output specification

For the easy subproblem, find and submit any strategy for $n = 3$ for which the probability of winning is more than 50%. If there is no such strategy, submit any strategy for which the probability of winning is exactly 50%.

For the hard subproblem, find and submit optimal strategies for all n from 1 to 8, inclusive, in order. If there are multiple optimal strategies for a given n, pick any of them. (That is, the output for this data set should start with one line of length 1, two lines of length 2 each, three lines of length 4 each, etc.)

Example

Example output for the easy subproblem

```
HHHH
PTPP
PPPP
```

This is a syntactically correct strategy for $n = 3$. In this strategy, player 1 always guesses "head", player 3 always passes, and player 2 passes except for one case – when she sees a head on player 1 and a tail on player 3, she guesses "tail".

Grid surveillance

Big Brother needs a new machine that will keep track of all the people moving into town.

Problem specification

The town is a grid $G[0..4095][0..4095]$. Initially, all entries in G are zeroes. The machine will need to support two types of instructions: additions and queries. Each addition modifies a single cell of the grid. Each query asks you about the sum of a rectangle. However, queries can also ask about older states of the grid, not only about the present one.

In order to have a smaller input file (and also in order to force you to process the instructions in the given order) we used the input format given below. There is a helper variable c, initially set to zero. Let $\varphi(c, x) = (x \text{ xor } c) \bmod 4096$. The instructions are processed as follows:

- Each addition is described by three integers x, y, a. To process it, first compute the new coordinates $x' = \varphi(c, x)$ and $y' = \varphi(c, y)$. Then, add a to the cell $G[x'][y']$. Finally, set c to the current value of $G[x'][y']$.
- Each query is described by five integers x_1, x_2, y_1, y_2, t. To process it, first compute the new coordinates $x'_i = \varphi(c, x_i)$ and $y'_i = \varphi(c, y_i)$. If necessary, swap them so that $x'_1 \le x'_2$ and $y'_1 \le y'_2$.

  Next, take the grid G_t that is defined as follows: If $t = 0$, then $G_t = G$. If $t > 0$, G_t is the state of G after the very first t additions. (If t exceeds the total number of additions so far, $G_t = G$.) Finally, if $t < 0$, then G_t is the state of G before the very last $-t$ additions. (If $-t$ exceeds the total number of additions so far, G_t is the initial state.)

  The answer to the query is the sum of all $G_t[i][j]$ where $x'_1 \le i \le x'_2$ and $y'_1 \le j \le y'_2$. This sum is also stored in c.

Input specification

The first line of the input contains an integer r ($1 \le r \le 100$). The second line contains an integer q ($1 \le q \le 20,000$). Each of the next q lines contains one instruction. The sequence of instructions you should process is obtained by concatenating r copies of these q instructions.

Each instruction starts with its type z. If $z = 1$, we are doing an addition, so three integers x, y, a follow ($0 \le x, y < 4096$, $0 \le a \le 100$). If $z = 2$, we are doing a query, so five integers x_1, x_2, y_1, y_2, t follow ($0 \le x_1, x_2, y_1, y_2 < 4096$, $|t| \le 500,000$).

Output specification

For each query, output a single line with the appropriate answer.

In the hard data set, only submit the answers to the last **20,000 queries**.

Example

input

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

output

```
3
3
3
0
6
0
```

HQ0-9+−INCOMPUTABLE?!

HQ9+ is an esoteric programming language specialized for certain tasks. For example, printing "Hello, world!" or writing a quine (a program that prints itself) couldn't be any simpler. Unfortunately, HQ9+ doesn't do very well in most other situations. This is why we have created our own variant of the language, HQ0-9+−INCOMPUTABLE?!.

A HQ0-9+−INCOMPUTABLE?! program is a sequence of commands, written on one line without any whitespace (except for the trailing newline). The program can store data in two memory areas: the *buffer*, a string of characters, and the *accumulator*, an integer variable. Initially, the buffer is empty and the accumulator is set to 0. The value of the buffer after executing all the commands becomes the program's output.

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

command	description
`h`, `H`	appends `helloworld` to the buffer
`q`, `Q`	appends the program source code to the buffer (not including the trailing newline)
`0-9`	replaces the buffer with n copies of its old value – for example, '2' doubles the buffer
`+`	increments the accumulator
`-`	decrements the accumulator
`i`, `I`	increments the ASCII value of every character in the buffer
`n`, `N`	applies ROT13 to the letters and numbers in the buffer (for letters ROT13 preserves case; for digits we define $\text{ROT13}(d) = (d + 13) \bmod 10$)
`c`, `C`	swaps the case of every letter in the buffer; doesn't change other characters
`o`, `O`	removes all characters from the buffer such that their index, counted from the end, is a prime or a power of two (or both); the last character has index 1 (which is a power of 2)
`m`, `M`	sets the accumulator to the current buffer length
`p`, `P`	removes all characters from the buffer such that their index is a prime or a power of two (or both); the first character has index 1 (which is a power of 2)
`u`, `U`	converts the buffer to uppercase
`t`, `T`	sorts the characters in the buffer by their ASCII values
`a`, `A`	replaces every character in the buffer with its ASCII value in decimal (1–3 digits)
`b`, `B`	replaces every character in the buffer with its ASCII value in binary (exactly eight '0'/'1' characters)
`l`, `L`	converts the buffer to lowercase

`e`, `E`	translates every character in the buffer to l33t using the following table:
	`ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 0123456789`
	`48(03=6#|JXLM~09Q257UVW%Y2 a6<d3f9hijk1m^0p9r57uvw*y2 0!ZEA$G/B9`

`?`	removes 47 characters from the end of the buffer (or everything if it is too short)
`!`	removes 47 characters from the beginning of the buffer (or everything if it is too short)

Problem specification

As you can see, HQ0-9+−INCOMPUTABLE?! is much more powerful than HQ9+. Demonstrate this by writing and submitting a HQ0-9+−INCOMPUTABLE?! program that outputs your TIS.

- For the easy subproblem, any valid program will do.
- For the hard subproblem, you must use the command "+" at least once.

Limits

Your program must be at most 10 000 commands long. The accumulator is unbounded (it can store an arbitrarily large integer). After each command, the buffer must be at most 10 000 characters long. To prevent code injection vulnerabilities, during the execution of your program *the buffer must never contain non-alphanumeric characters*, i.e. characters other than `A-Z`, `a-z`, and `0-9`. Should this happen, the program fails with a runtime error, and your submission will be rejected.

Examples

program	output
`h5!`	`rld`
`QCq`	`qcQQCq`
`q23`	`q23q23q23q23q23q23`
`h?h`	`helloworld`
`H2O`	`hlwolheo`
`h4op`	`ollwldwlhe`
`hint`	`ccfkrsvzzz`
`q18N`	`d41Ad41Ad41Ad41Ad41Ad41Ad41Ad41A`
`3QAh`	`518165104helloworld`
`Qb`	`0101000101100010`
`opaque`	`094QU3`
`h1Qt`	`1Qdehhllloortw`
`H9999`	(error: buffer size exceeded 10 000)
`quine`	(error: buffer contains "`|`")
`LMAO`	(empty output)

Inverting bits

The wannabe scientist Kleofáš has recently developed a new processor. The processor has got 26 registers, labeled `A` to `Z`. Each register is an 8-bit unsigned integer variable.

This new processor is incredibly simple. It only supports the instructions specified in the table below. (In all instructions, `R` is a name of a register, `C` is a constant, and `X` is either the name of a register or a constant. All constants are 8-bit unsigned integers, i.e., numbers from 0 to 255.)

syntax	semantics
`and R X`	Compute the bitwise and of the values `R` and `X`, and store it in `R`.
`or R X`	Compute the bitwise or of the values `R` and `X`, and store it in `R`.
`not R`	Compute the bitwise not of the value `R`, and store it in `R`.
`shl R C`	Take the value in `R`, shift it to the left by `C` bits, and store the result in `R`.
`shr R C`	Take the value in `R`, shift it to the right by `C` bits, and store the result in `R`.
`mov R X`	Store the value `X` into `R`.
`get R`	Read an 8-bit unsigned integer and store it in `R`.
`put R`	Output the content of the register `R` as an 8-bit unsigned integer.

Notes:

- After any instruction other than `not`, if the second argument was a register name, its content remains unchanged.
- Whenever `shl` or `shr` is called, the shifted value of the first argument is truncated on one end and padded with zeroes on the other end. For example, if `X` equals to binary 10110110, then `X shr 2` equals to 00101101.

Example task

Assume that the input contains an arbitrary two 8-bit unsigned integers a and b. Write a program that will read these numbers and produce an integer z such that $z = 0$ if and only if $a = b$. (That is, if a and b differ, z must be positive.)

Solution of the example task

One program solving the example task looks as follows:

```
get A
get B

mov C A
not C
and C B

mov D B
not D
and D A

mov Z C
or Z D

put Z
```

Explanation: There are many possible solutions. In our solution, we read the input to registers `A` and `B`, compute their bitwise xor in register `Z` and output it.

However, we do not have an instruction for xor. We have to assemble it from ands, ors, and nots. First, we create (not-`A` and `B`) in `C` and (not-`B` and `A`) in `D`. Finally, we store (`C` or `D`) in `Z` and we are done.

Why does it work? If the two numbers are identical, then `C` and `D` will both be zero, and so will `Z`. If the two numbers are not identical, consider a digit d where the two numbers differ in their binary representations. If this digit is 0 in `A`, it has to be 1 in `B`. But then it is 1 in `C` and therefore it is 1 in `Z`. And if the digit is 1 in `A`, it has to be 0 in `B`, 1 in `D`, and 1 in `Z`.

Input specification

This problem has no input.

Problem specification

For each data set, you have to write a program for Kleofáš's processor that solves the following task: The input for your program is a sequence of integers, each of them a 0 or a 1. The output of your program must be a sequence of their negations, in order. (That is, change each 0 into a 1 and vice versa.)

Easy subproblem: The sequence contains **exactly 7** integers. In your program, you may only use the instruction `not` at most **once**. Your program must have at most 100 instructions.

(Note that you have to use each of the instructions `get` and `put` exactly 7 times.)

Hard subproblem: The sequence contains **exactly 19** integers. In your program, you may only use the instruction `not` at most **twice**. Your program can have at most 300 instructions.

Output specification

Send us your program as a plain ASCII text. Each line of the text may either contain one complete instruction, or just whitespace. If you submit anything that is not a proper program, it will be judged "Wrong answer" and you will get an appropriate error message.

Jedi academy

One of the excercises young Jedi must go through in Jedi acedemy training is flying and shooting on a special training machine. Now Luke Skywalker has to examine the results of this training. He wants to know whether the pilots hit something or not.

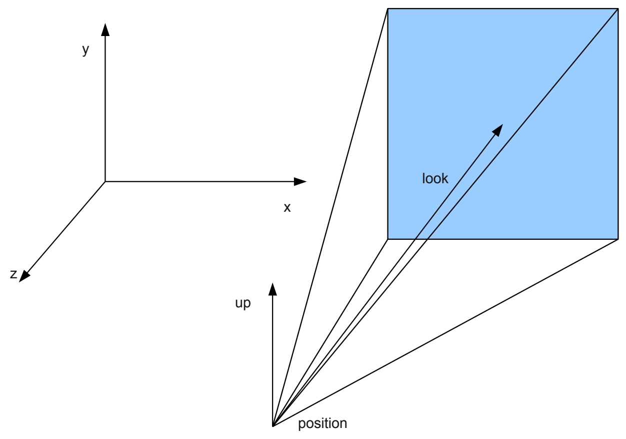
Problem specification

You are given a description of several models of combat ships. Each model is a set of triangles in 3D space. All models are sane (they have a reasonable structure, they are not just random triangles).

The 3D scene consists of several ships called objects. Each object is one of the models, rotated around some axis and moved into some position. The objects are disjoint.

We took snapshots of the scene from several different positions and angles. Each snapshot is described by the camera position (a point), the camera look direction (a vector) and the camera up vector (the direction that will be "up" on the snapshot). Each camera uses a standard perspective projection with a field of view 90 degrees. (Field of view is the angle between the left and right boundary of view, and also the angle between up and down boundary of view. The camera look direction passes through the center of the snapshot.)

The entire scene seen by the camera is projected onto a square screen with fixed dimensions 500×500 pixels. Its lower left corner has coordinates $(0, 0)$, its upper right corner has coordinates $(499, 499)$.



Obr. 1: A right-hand coordinate system (left) and camera description (right).

At the moment of the snapshot, a laser shot was taken from the position of the camera, passing through the point (x, y) on the screen. Your should find whether an object is hit by the shot. If yes, find the closest such object (that is the one actually hit). You may assume that the shot is never on the object boundary (all neighbouring pixels of the snapshot will always correspond to the same object).

Input specification

First line of the input contains an integer m $(1 \leq m \leq 10)$, the number of ship models. A description of m models follows. First line of the i-th description contains an integer p_i $(4 \leq p_i \leq 50,000)$, the number of points on the model boundary. Next p_i lines contain coordinates of points – three floating point numbers x_{ij}, y_{ij}, z_{ij}. The next line of a description contains the number t_i $(4 \leq t_i \leq 50,000)$, the number of triangles forming the boundary of the model. Each of the last t_i lines describes one triangle using three zero-based indices of points (each number is an integer between 0 and $p_i - 1$, inclusive).

After all model descriptions, the next line contains an integer o $(1 \leq o \leq 5,000)$, the number of objects. Next, o object descriptions follow. The object description starts by a number m_i $(0 \leq m_i < m)$, model number which is used for the object. The next line contains a description of a rotation of the model. It starts by three floating point numbers x_{ir}, y_{ir}, z_{ir} – the vector specifying the rotation axis. (Always at least one of the numbers is non-zero.). These are followed by a floating point number a_{ir} – the angle of the rotation in degrees. The rotation follows the right-hand rule, so if the vector (x_{ir}, y_{ir}, z_{ir}) points toward you, the rotation will be counterclockwise from your point of view. The rotation axis always passes through $(0, 0, 0)$. The last line of an object description is the object position, three floating point numbers x_{ip}, y_{ip}, z_{ip}. These numbers are added to the coordinates of all points in the model after the rotation (i.e., first we rotate the model, then we translate it).

The next line contains the number v $(1 \leq v \leq 10,000)$ of snapshots. Each snapshot is described on four lines. The first line contains the camera position, three floating point numbers. The second line contains the camera look direction, three floating point numbers. The third line contains the camera up vector, three floating point numbers. This vector is always perpendicular to the look direction. The fourth line contains the pixel hit by the shot, two integers c_x, c_y $(0 \leq c_x, c_y < 500)$.

Output specification

For each snapshot you should output one number. Either the number of an object hit by the laser shot (object numbers are zero-based), or the number -1, if there is no such object.

Example

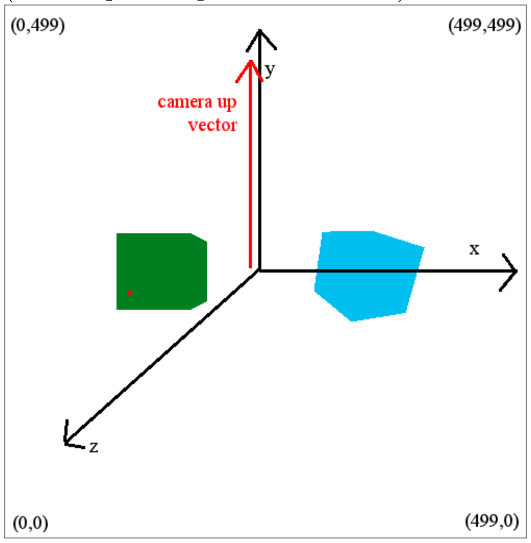
input

```
1
8
-1.000000 -1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 -1.000000
-1.000000 1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
12
0 1 2
0 2 3
4 5 6
4 6 7
0 1 4
1 4 5
3 2 7
2 7 6
1 5 2
5 2 6
0 4 3
4 3 7
2
0
0.000000 1.000000 0.000000 0.000000
-3.000000 0.000000 0.000000
0
2.000000 2.000000 0.000000 30.000000
3.000000 0.000000 0.000000
3
0.000000 0.000000 8.000000
0.000000 0.000000 -1.000000
0.000000 1.000000 0.000000
120 229
0.000000 0.000000 6.000000
0.000000 0.000000 -1.000000
1.000000 1.000000 0.000000
380 309
0.000000 0.000000 6.000000
0.000000 0.000000 -1.000000
1.000000 1.000000 0.000000
380 269
```

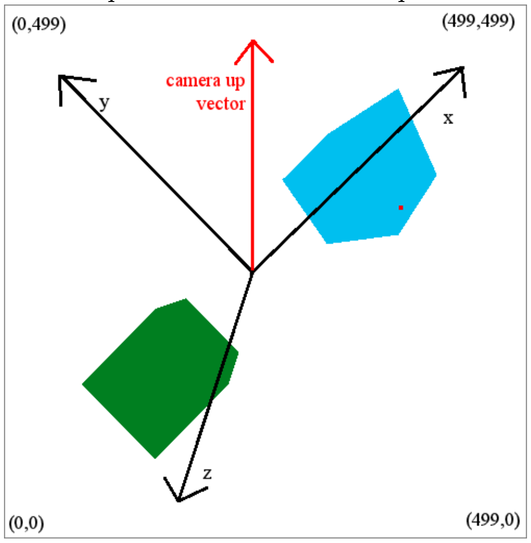
output

```
0
1
-1
```

Here is the picture of the first snapshot (the red point represents the click):



And the picture of the second snapshot:



Keep clicking, keep flipping

The "Flip it" puzzle is played on a board with black and white tiles. In each step you pick a tile, flip it, and also flip all the adjacent tiles (from white to black and vice versa). The goal of this game is to turn all the tiles to their white side. We found this puzzle too easy, so we picked a harder version for you:

- Instead of square tiles, we will consider a graph with n nodes, each black or white.
- *We only allow clicking on black nodes.*
- When we click on a node x, its colour and the colour of all the adjacent nodes flips.
- Additionally, all edges in the subgraph containing x and all its neighbours are flipped: If two of these vertices were adjacent before, now they are not, and vice versa.
- The goal of the game: At the end, all nodes must be *white* and no two of them may be *adjacent.*

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 two integers: n and m. Nodes are numbered from 0 to $n-1$. The second line contains a string of length n. The i-th character of this string is either `B` or `W` and gives the colour of the i-th node at the beginning of the game. Each of the following m lines contains a pair of nodes x_i, y_i that are connected by an edge at the beginning of the game.

In the easy data set, $n \leq 50$, in the hard data set, $n \leq 2000$.

Output specification

For each test case, output the number of clicks and one particular sequence of clicks that wins the game. If there is no such sequence, output a single line containing the number -1.

Example

input

```
2

3 2
BBW
0 1
0 2

7 12
WBBWWWW
0 2
0 3
1 2
1 4
1 6
2 3
2 4
2 6
3 4
3 6
4 5
5 6
```

output

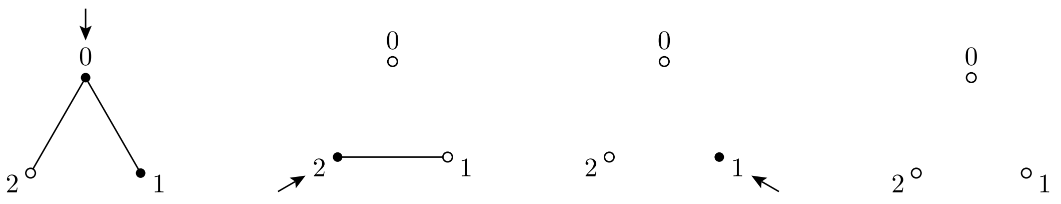
```
3
0 2 1

5
1 6 3 0 2
```

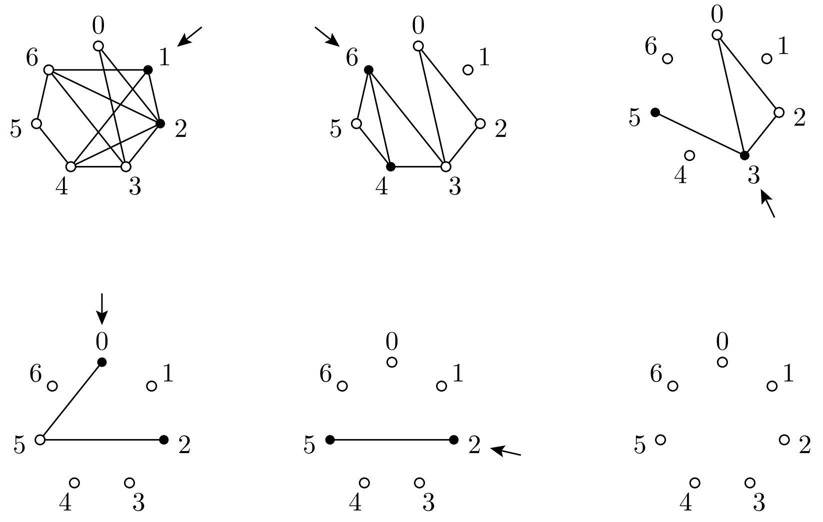
Note that in the first test case, if we started by clicking at 1, we would end up stuck with two adjacent white vertices.

The figures below illustrate the solutions for the two test cases.

Test case #1:



Test case #2:



Lame crypto

Bob wants to draw tons of .png images (`http://www.w3.org/TR/PNG/`) and send them to Alice. He does not want anyone else to see those pictures. Therefore, he designed the Super Secret Protocol (SSP) and he uses it to communicate with Alice.

Eve wants to destroy their relationship. She bribed their internet providers to slow down their internet so that she could get access to their communication. Now, every message between Bob and Alice travels so slow that Eve can hold it for up to several hours. She has also drawn an evil image, which she wants to deliver to Alice in Bob's name. Then Alice will break up with Bob for sure[1].

Unfortunately (for Eve), Eve knows exactly nothing about cryptography. Therefore, she hired IPSC organizers to take care of it. And as they were just preparing this contest, they decided to kill two birds with one stone – and let you do the dirty work for them. The evil images (one for each subproblem, see below) are provided in the respective input files.

SSP protocol specification

Alice has a secret key K_A, Bob has a secret key K_B. When Bob wants to send a message M of length n to Alice, the following happens:

1. Bob chooses some (contiguous) subsequence K'_B of size n of key K_B. He sends $M \oplus K'_B$ to Alice.
2. Alice receives message $C = M \oplus K'_B$. She chooses some (contiguous) subsequence K'_A of size n of her key K_A and sends $C \oplus K'_A$ to Bob.
3. Bob receives message D, sends her back $D \oplus K'_B$.
4. Alice receives $E = D \oplus K'_B$. Since $\oplus$ is commutative, $A \oplus A = 0$ and $A \oplus 0 = A$, we know that $E = M \oplus K'_A$. Hence Alice can now compute M as $E \oplus K'_A$.

Notes: $\oplus$ is bitwise xor. Bob needs exactly six minutes to draw an image. Therefore he will always start sending the next image six minutes after Alice receives the previous one.

Problem specification

Your task is to monitor the messages sent between Alice and Bob and to change some of the messages in such a way that Alice will get the evil image.

In the easy subproblem, you can change any message you want. However, in the hard subproblem, you can only change the messages that go from Alice to Bob.

Note that the hard subproblem uses different secret keys and other messages than the easy subproblem. The protocol remains the same.

Input/output specification

We will provide you with two web pages (one for the easy subproblem, one for the hard one) where you can download all messages that were already sent (possibly with your modifications). Messages are in binary format, each one has exactly 10000 bytes.

The most recently sent message in on the top of the list. If you want to change this message, submit (using the standard submission interface) a binary file containing the new message. If you just wish to deliver it in its original state, submit a text file containing one line with a single word: "`forward`".

Valid submissions during the protocol are **not** counted as incorrect – you will **not** receive penalty minutes for these submissions.

In the hard subproblem you can only change one message from the protocol, the other ones will be forwarded to their recipient automatically.

[1] Well, with 95% confidence.

In both subproblems, the time between any two submissions in different instances of the protocol has to be at least 6 minutes. In other words, you may not try to submit anything while Bob is drawing the next picture. Submissions that violate this rule will be judged as wrong answers; upon solving the task you **will** receive penalty minutes for these submissions.

The replaced message always has to have the same size as the original message (otherwise you will get an wrong answer).

At the end of the protocol, if Alice received anything other than the evil image, you will also get a wrong answer. If Alice receives the evil image, you solved the subproblem.

The evil image and all messages sent by Alice or Bob have exactly 10000 bytes. All images are valid PNG images. Alice and Bob have keys of size 50000 bytes.

My little puppy

You have a puppy. The puppy wants your attention. As every puppy, it wants to sleep, eat, play, and sometimes something more. Be careful, be patient and read its requests carefully. Your reward will be priceless. Well, not really priceless. Your reward will be paid in negative penalty time.

For each of the puppy's requests you'll have a time interval during which you have to choose among several options and send us your choice. If you miss time limit or choose incorrectly, you lose and your puppy runs away forever. If you choose correctly, you will be rewarded. Additionally, a few minutes after your submission the puppy's next request will be shown.

The requests will appear in the online version of this problem statement.

Input specification

For each request, you will get an ASCII art of the situation, and a list of available options you have. Each option is labeled by a lowercase letter.

Output specification

During the appropriate time interval, submit a text file containing a single line with a single letter – the option you chose.

(The data set for the submission can be either M1 or M2, both will work. You are only playing one game, not two of them.)

You may submit your choice for the first request at any moment during the contest. Your time starts running once you make this first submit. (It is recommended to start as early as possible, so that you have the chance to see many requests.)

Example

input

```
  |\_/|
 / @ @ \
( > ° < )
 ‘>>x<<’
 /  O  \

What do you see?

a) a kitten
b) an elephant
c) a dog
d) where?
```

output

```
a
```