Arithmetics for dummies

While browsing aimlessly, Peter stumbled upon an old riddle he used to solve on his calculator when he was still young. It was the kind of a riddle where you punch in a bunch of numbers and operators into a simple pocket calculator and then turn it upside down to get the answer:

These come in many different sizes but they are always exactly one foot long. Answer: $103 \times 103 \times 5$.

What are made of ice to keep people warm? Answer: $50 \times 40 \times 250 + 791$.

After a few minutes he found a large amount of such riddles and full of excitement he went to solve them. He turned his computer screen upside down…

… only to find out that he does not have a reasonable calculator program installed on his computer.

Problem specification

You are given multiple sequences of button presses of a simple pocket calculator that consist of digits and arithmetic operators. For each such sequence find the number it would produce on a pocket calculator's display.

Note that the pocket calculator evaluates the operators in the order in which they are given. (I.e., there is no operator precedence.) Assume that the display of the calculator is large enough to show the result, and that its memory is sufficient to store all intermediate results.

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 represents one sequence of button presses for a pocket calculator. The sequence consists of non-negative integers and arithmetic operators and ends with an equal sign. It may also contain spaces to improve readability.

The operator `/` represents integer division, rounded down. You may assume that no test case contains division by zero and that in all test cases all intermediate results are non-negative.

The easy data set only contains the operator `+`.

Output specification

For each sequence from the input file output the number that would be displayed on the calculator.

Example

input

```
4

1 + 1 * 2 =

29 / 5 =

103 * 103 * 5 =

50 * 40 * 250 + 791 =
```

output

```
4
5
53045
500791
```

The first test case shows that there is no operator precedence.
The second one shows that integer division always rounds down.
The last two outputs are the answers to the two riddles in the problem statement:
"shoes" (53045 upside down), and
"igloos" (500791 upside down).

Bouncing balls

“Behold, my queen”, said the jester, “the great Bouncing Ball Bowl!” The queen boredly waved her hand and sarcastically replied: “Let the fun begin!”. And the fun begun! The jester spoke a magic word and all the colorful balls in his bowl started to roll and bounce, creating interesting pictures.

The queen watched vividly for a few minutes, but then she started to be bored again. “Just wait a moment, Your Majesty, in a minute they’ll. . . ” started the jester, but the queen interrupted: “I’m a queen! I don’t want to wait! Can’t you just fast forward it or something?”

Problem specification

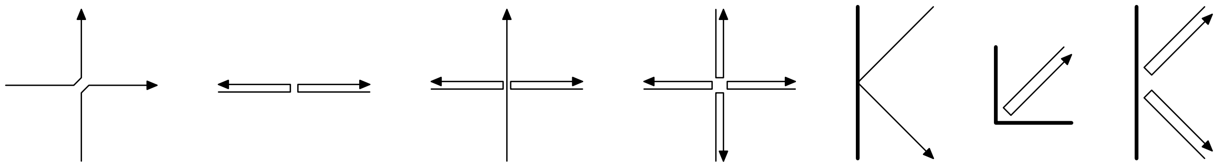
The jester’s box is an $X \times Y$ rectangle. The rectangle contains N small balls. At any moment, each ball is travelling at the same speed in one of the four diagonal directions.

The movement of the balls is continuous and for the purpose of this problem we may consider them to be points. When two or more balls meet, they bounce in a way described below.

Your task is to determine the state of the box at given moments in time.

Bouncing specification

Bouncing does not change the speed of the balls. Following images show how the balls bounce off each other, and also off walls. Each image can be rotated arbitrarily. For example, the first image shows that whenever two balls meet at a right angle, they bounce and depart at a right angle again. One particularly tricky case is shown in the third image.



Input specification

The input starts with a line containing the dimensions X and Y of the box. We will use a coordinate system with axes parallel to the sides of the box, $(0, 0)$ at one of the corners and (X, Y) at the opposite corner.

The second line contains the number of balls N.

Each of the next N lines contains four integers x, y, v_x, v_y, where (x, y) are the coordinates of one ball at time 0 and (v_x, v_y) is its current velocity vector. (Each ball will be strictly inside the box and for each ball both v_x and v_y will be equal to ± 1. No two balls will start at the same place.)

The following line contains the number of queen’s requests M.

On the last line there are M numbers $t_1, \dots, t_M$ – the points in time the queen wants to see.

Output specification

As a solution to this problem, we expect a file with M blocks, with the i-th block describing the situation at time t_i.

Each block must contain N lines, and each line must contain the x and y coordinates of one ball. The balls in each block must be sorted – primarily by to their first, secondarily by their second coordinate at that point in time.

You may output an empty line between the blocks.

Example

input

```
6 4
4
1 2 1 1
5 2 1 1
2 1 1 -1
3 1 -1 -1
1
4
```

output

```
1 3
3 2
5 2
6 3
```

Note that the balls that start at (2,1) and (3,1) bounce off each other at a non-integer point in time.

Cryptic punchcards

Punch cards are an important part of programming history. Quoting Wikipedia, *a punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions. Now almost an obsolete recording medium, punched cards were widely used throughout the 19th century for controlling textile looms and in the late 19th and early 20th century for operating fairground organs and related instruments. It was used through the 20th century in unit record machines for input, processing, and data storage. Early digital computers used punched cards as the primary medium for input of both computer programs and data, with offline data entry on key punch machines.*

One of the most famous types of punch cards were IBM's 80-column punch cards. Their width is the reason why even nowadays many terminals default to 80 columns of characters.

Various encodings were used to punch the data onto cards. One of the most famous ones is EBCDIC. Quoting the Jargon File, *It exists in at least six mutually incompatible versions, all featuring such delights as non-contiguous letter sequences and the absence of several ASCII punctuation characters fairly important for modern computer languages. [...] Hackers blanch at the very name of EBCDIC and consider it a manifestation of purest evil.*

You probably already know where this leads to.

Problem specification

In this problem you will be given image files showing some punch cards. Imagine that you are several decades in the past. You actually have the same data punched on real paper punch cards. You take them and feed them into the machine...

Your task is to find out what the machine (most probably) read, and use that to determine the output you send us.

Input specification

The input is a set of image files numbered in order in which they should be fed into the machine.

Output specification

Send us a text file containing a single line with a single positive integer.

Don't worry about wrong answers

Every year many teams competing in the IPSC have huge penalty times due to lots of incorrect submissions – and also due to the fact that they failed to send us a nice postcard.

We know that many of the tasks are hard, with tricky cases, and try as you might, you just keep getting those `WRONG ANSWER` messages.

To make up for those bad feelings, in this task we decided to be extra nice to you. We will accept almost anything you submit for this problem!

And it gets even better. In this problem, if we somehow manage to disappoint you by a `WRONG ANSWER` message, we will try to make it up to you.

Scoring specification

For each of the data set D1 and D2 you **do not** solve, you score 0 points and gain 0 penalty minutes, as in all other tasks. Solving D1 is worth 1 point, solving D2 is worth 2 points, as in all other tasks.

The formula to calculate the penalty time is different. For this problem, the penalty time is calculated using the formula $T + (-40) \cdot R \cdot D$. Here T is the time in minutes when you submitted a correct solution, $D \in \{1, 2\}$ is the difficulty of the data set, and R is the count of previously rejected submissions for that data set.

In words, for every wrong answer you make we **decrease** your penalty time by 40 minutes in D1 and by 80 minutes in D2.

Note that the submission limit still applies. (You are allowed to make at most 10 submissions for each particular data set.)

Problem specification

1. You submit a text file.
2. We look at its first line. If it contains anything other than a single integer, we will accept your submission.
3. If it does contain a single integer and you already submitted this integer before, we will accept your submission.
4. If we still did not accept your submission, we run the program we provide in the input file to decide whether to accept or reject your submission.

(A small technical detail: The program in `d1.in` may seem really slow, but don't worry, our machine is really fast. Or maybe we are cheating and use a faster version of the same program, who knows.)

Input specification

The input file contains the source code of the program we will use to judge your submissions. The program is written in Python, which should make it readable to almost anyone.

Output specification

You may submit any text file the contest system allows you to submit.

Example

input

```
#!/usr/bin/python
import sys

def correct():
  print("OK")
  sys.exit(0)

def wrong_answer():
  print("Wrong answer")
  sys.exit(1)

# we already know that the first line contains a number, read it
N = int(sys.stdin.readline())

if N==47:
  wrong_answer()
else:
  correct()
```

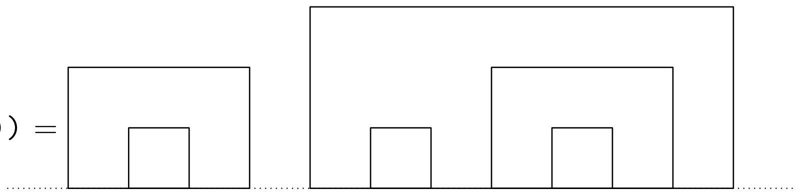
output

```
42
is the answer to life, universe and everything.

This is just one of very many inputs that would get accepted.
This task is that easy!
```

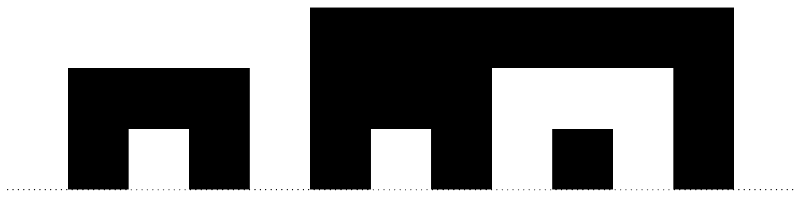
Easy representation

Peter is preparing slides for his lecture on parsing arithmetic expressions. In the first part of the lecture he wants to focus just on parsing brackets. He invented an interesting geometric representation of a correct bracket sequence for his students, because one image is better than a thousand words:

`(())(()(()))` =


Formally, the definition of the geometric representation looks as follows. The simplest correct bracket sequence `()` is represented by a 1×1 square. If `A` is a correct bracket sequence and $g($`A`$)$ its represenation, then the representation for `(A)` is $g($`A`$)$ surrounded by a rectangle two units wider than $g($`A`$)$ and one unit taller than the highest point of $g($`A`$)$. If `A` and `B` are two correct bracket sequences and $g($`A`$)$ and $g($`B`$)$ are their representations, then we get $g($`AB`$)$ by placing $g($`B`$)$ one unit to the right of $g($`A`$)$.

After he finished his slides, Peter started to play with the images he prepared. He painted the bounded areas of the images alternately black and white, in such a way that the outer-most areas are all painted black. For the example above this coloring looks as follows:



Problem specification

You are given a correct bracket sequence. Calculate the area that is colored black.

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 with a correct bracket sequence. Every line will only contain characters '`(`' and '`)`'.

Output specification

For each test case output one line with one integer – the area of the black part of the corresponding geometric representation.

Example

input

```
2

(())

(())(()(()))
```

output

```
10
20
```

The second test case is the one shown in the pictures above.

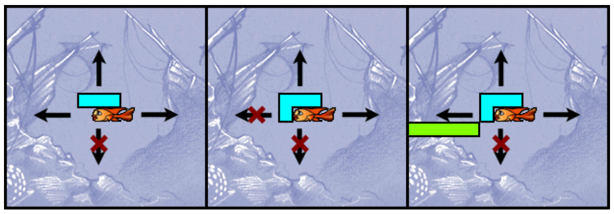
Fish Fillets

Fish Fillets is little known but extremely cool and challenging logic game made by the Czech company Altar, and later released as open source. In this task you will have to solve two new levels – by any means you can use.

You are controlling two fish trapped in a room. The big blue fish is Agent Max Flounder, and the small orange one is Agent Tina Guppy. The entire game is played on a discrete grid. In each step, you may move one of the fish by one square in one of the four basic directions. The fish can push and carry other objects according to specific rules described later. The player's goal is to get both fish out of the room. Sounds simple – but make one bad move and falling objects will kill your fish.

To get yourselves acquainted with the game, you can watch a video with instructions at `http://www.youtube.com/watch?v=qs1JedhK0fI` or download this video from our server. Alternately, the same rules as in the video are summarized below.

Pushing objects



The fish are only in danger when they move some object. The simplest situation is lifting objects – there is no danger in it. If the fish on the left picture moves up, it lifts the object. If it moves left or right, the object stays in place. If the fish goes on moving left or right long enough, it will eventually come out from under the object, and the object will fall harmlessly down. The only dangerous direction is down. If the fish moves down, the object it was supporting will fall on it and kill it.

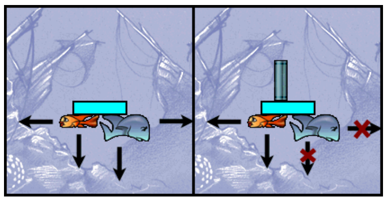
If the fish want to push an object left or right, the object must be supported by some structure or other object. The fish cannot support the object it is pushing. The fish on the middle picture cannot move left because it would get killed.

The fish can push an unsupported object only if it becomes supported in the new position. An example of such a situation is on the right picture.

Steel objects

In some levels you will encounter steel objects. They look like the cylinder on the picture above. Steel objects can only be lifted and pushed by the big fish. The small fish cannot move them. Moreover, if she gets to a situation where she has to support a group of objects that includes some steel objects, she will perish.

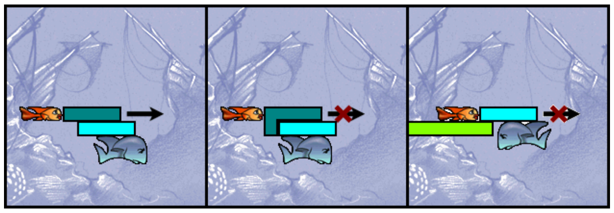
Transferring objects



The fish can transfer objects between themselves. If one fish supports the object and the other one gets into position where the object rests also upon her, the first fish can move away. Any fish can move away in the left picture and the object will stay upon the other.

Be careful with transfer of steel objects. In the situation on the right picture only the smaller fish can move away.

Pushing objects along other objects



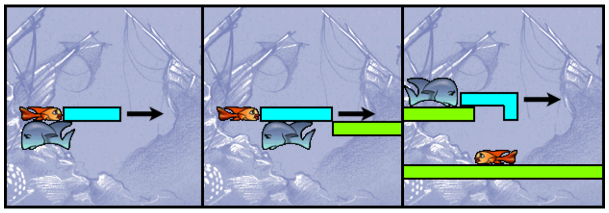
It is allowed to push objects that are not directly supported by the fish. One such case is presented on the left picture.

However, a fish is not allowed to push an object that is directly supported by her partner; in such case the partner would perish. More precisely, this is disallowed in the case where the object would rest directly on the partner after the push. For example, the small fish on the middle picture cannot push right because it would kill the big fish.

For the same reason it is impossible to place an object upon a fish. This is shown on the right picture.

Note that on the middle picture the small fish could swim around and push the L shaped object left.

Falling objects



Falling objects are always deadly. It doesn't matter how long the object falls, if it hits a fish or something that rests upon a fish, the fish is dead.

The object supported directly by a fish can only be pushed if it falls down or rests upon some structure immediately afterwards. This is shown on the left and middle pictures.

Be on the lookout for the situation where the object seemingly hits the fish but at the same time it hits some structure. In such a situation the fish is safe. An example of such a situation is on the right picture.

Problem specification

Your task is to solve two levels of this game. A level is solved if both fish leave the room alive. A fish will leave the room the moment it touches the boundary of the level After a fish leaves, the player can not return it back into the room.

We provide you both with a textual description of the levels and with a Flash applications which you can use to familiarize yourselves with the rules and even to solve the entire levels, if you prefer to do it by hand. If you solve a level, the application will congratulate you and show you the sequence of moves you made. (You can then copy this sequence and submit it.)

Instructions for the Flash application

- R: restarts the level
- S or F2: saves the current position
- L or F3: loads the most recently saved position
- space: switches which fish is active
- arrow keys: moves the active fish one step in the given direction
- some objects are animated – this is eye candy only, ignore it when solving

Input specification

All coordinates in the input are 0-based, with (0,0) in the upper left corner. For each object the initial coordinates of its upper left corner are given. The rest should be obvious.

Output specification

Send us a text file with a sequence of moves that wins the given level. Any such sequence of 5 000 or less moves will be accepted. Each move is a character from the set `UDLRudlr`. Lowercase letters are movements of the small fish, uppercase ones are movements of the big fish. The letters represent the directions: Up, Down, Left and Right. Whitespace does not matter.

Example

input

```
room:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

small fish:
row 4 col 1
1 1 1

large fish:
row 1 col 10
1 1 1 1
1 1 1 1

item 1:
row 2 col 9 steel
1 0
1 1
1 1
```

output

```
L R D L L L L
R R R R D R
u u u r r r r d r r r r r r r d r
```

The first row of the output is the big fish pushing the steel object to the left. In the second row it exits the room. Finally, the small fish can now swim around the steel object and exit the room as well.

The entire initial position in ASCII art looks as follows:

```
111111111111111
1.......1.BBBB1
1........oBBBB1
1........oo....
1SSS.....oo....
111111111111111
```

Here, `B` *is the big fish,* `S` *the small one, and* `o` *is the steel object.*

Going to the movies

N friends decided to go to the local cinema together. They all bought tickets to the same row. As there was still some time left, each of them took her ticket and went shopping until the movie starts.

They all arrived back late, the movie already started. The usher standing at the door agreed to let them in one by one. Each of the girls was supposed to find her place and sit down.

However, the machine that printed their tickets was broken. Instead of consecutive numbers, each girl received a random seat number between 1 and K, where K is the number of seats in their row. The seat numbers they received were not necessarily distinct.

When a girl tries to sit down, she enters the row at the end where seat number 1 is, and walks until she reaches the number on her ticket. If her desired seat is free, she just sits down. If it is already taken, she continues to walk in the same direction until she finds the first free seat, and sits there.

Of course, it is possible that some unfortunate girl will reach the end of the row without finding a place to sit. In that case, the usher comes and throws her out.

Problem specification

You are given the numbers N and K.

Assume that each girl's ticket had a number between 1 and K, inclusive. Each number was drawn uniformly at random, and draws were independent.

Also assume that the entire row was empty when the first girl started to look for her seat.

Compute the probability that at least one girl suffered the sad fate of being thrown out by the usher.

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 two integers N and K.

Output specification

For each test case output a single line with the probability as a simplest fraction.

(Do not output any spaces before or after the / sign.)

Example

input

```
3

1 10

2 3

3 3
```

output

```
0/1
1/9
11/27
```

In the third case there are $3^3 = 27$ possibilities. Out of these, in 11 some girl is thrown out. These 11 sequences are: 133, 222, 223, 232, 233, 313, 322, 323, 331, 332, and 333.

For example, if the sequence of numbers were 322, the first girl sits at seat #3, the second one at #2, and then the third one tries to sit at #2, but finds both seat #2 and seat #3 occupied, and she's thrown out.

Hunt!

This year new sheep-hunting rules were set in Wolfenstein:

- wolves are allowed to form packs of up to 10 animals
- hunting is only allowed on a torus divided into 30 × 30 cells
- there are 60 sheep randomly placed on the torus
- in each round:
  - wolves move before sheep do
  - a move means that an object moves from its cell into one of the four neighboring ones
  - all wolves together can do up to 10 moves
  - no single wolf can do more than 5 moves
  - each sheep can do up to 1 move
  - sheep moves are deterministic – they basically try to move away from the closest wolf
- if at any time a wolf moves to a field occupied by a sheep, the sheep is eaten
- if at any time a wolf moves to a field occupied by another wolf, one of the wolves is eaten

A torus is a surface shaped like a donut. You can imagine the hunting area as a normal 30 × 30 chessboard, except for the fact that in each column the top and bottom cells are neighbors as well, and the same holds for the leftmost and rightmost cell in each row.

Problem specification

You are controlling the wolves. To gain one point for solving the easy data set, your task is to eat 10 sheep. To gain two more points for solving the hard data set, you have to eat all 60 sheep.

Note that you only play a single game for both data sets, and this game **can not be restarted**.

You are allowed to send us the moves of your wolves multiple times. However, there must be **at least five minutes** between any two consecutive submissions you make. Each time you send us your moves, we evaluate them and then move the sheep in your game.

Scoring

As this task is a bit special, so is the scoring:

- You will get a time penalty (a '*' in the results) for each submission that does not obey the "five minutes between submissions" rule.
- Rejected submissions count towards the data set you are working on. I.e., until you eat 10 sheep all penalties are assigned to the easy data set, afterwards they are assigned to the hard data set.
- The limit is 10 incorrect submissions per data set, as usual.

Input specification

There is no input for this task. You will receive your game description after your first submit.

Output specification

Every round we expect a list of 0 to 10 moves of your wolves (all tokens are separated by any whitespace). Each move is described by a pair `id dir` where `id` is the number of wolf (1 to 10, in the order they appear in the latest state description), `dir` is one of '`U`', '`D`', '`L`', and '`R`' (for moving the wolf up, down, left or right along the grid). Invalid moves in your submission will be ignored.

After any non-rejected submit you will receive the description of the current state.

Which data set to use?

Don't worry about this. Regardless of how many sheep you already ate, you may always submit your moves as your "solution" to either of the data sets H1 and H2. The tester will handle it for you.

State description

The state description you receive after each submit is formatted as follows: It consists of 2 blocks – wolves block and sheep block. Each block starts with a number `N` giving the count of wolves/sheep. This number is followed by N pairs x y ($0 \leq x, y < 30$) describing positions of each wolf/sheep.

You may assume that in each state all living sheep have distinct positions.

The directions up, down, left and right are such that from a cell $(4, 7)$ a move up takes you to $(4, 6)$, and a move left takes you to $(3, 7)$.

Example

game description

```
3
4 0   5 5   29 7
5
4 8   4 28   15 3   18 12   6 27
```

The above game description contains 3 wolves and 5 sheep.

moves

```
1 U
1 U
1 L
3 R
3 D
```

This is a sequence of moves in which first wolf 1 makes three moves, and then wolf 3 makes two moves.

If we apply this sequence of moves to the game state described above, we'll get the following new state:

game description

```
3
3 28   5 5   0 8
4
4 8   15 3   18 12   6 27
```

Note that after two steps wolf 1 ate the sheep at $(4, 28)$.

Now the sheep would move. For example, the sheep at $(6, 27)$ is closest to the wolf at $(3, 28)$, hence it would move away from him.

Instructions

The wannabe scientist Kleofáš has recently developed a new processor. The processor has got 26 registers, labeled `A` to `Z`. Each register is a 64-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, and `X` is either the name of a register, or a 64-bit unsigned integer constant.)

syntax	semantics
mov R X	Store the value X into R.
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.
xor R X	Compute the bitwise xor 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 X	Take the value in R, shift it to the left by X bits, and store the result in R.
shr R X	Take the value in R, shift it to the right by X bits, and store the result in R.

Notes:

- After any instruction other than `not`, if the second argument was a register name, its content remains unchanged.
- If `shl` or `shr` is called with a non-zero second argument, the shifted value of the first argument is padded with zeroes. Note that this means that if the second argument is 64 or more, the result will always be zero, regardless of the first argument.

Example task:

Assume that the register `A` contains an arbitrary input number between 0 and 15, and that all the other registers contain zeroes. Write a program that will set `Z` to 1 if the number of set bits in `A` is odd, and to 0 otherwise.

Solution for the example task:

The answer can easily be computed as the bitwise `xor` of the four lowest bits in `A`.

We can isolate a single bit X by first shifting the number $63 - X$ bits to the left (the X-th bit will become the most significant one, and all more significant bits of `A` become lost), and then 63 bits to the right (the originally X-th bit is now the least significant one).

In this way we can take the four bits of `A` that may be non-zero, and store them into `B` to `E`, respectively. We then compute the bitwise `xor` of these four bits and store it in `Z`.

The entire program solving the example task:

```
mov B A
shl B 63
shr B 63

mov C A
shl C 62
shr C 63

mov D A
shl D 61
shr D 63

mov E A
shl E 60
shr E 63

mov Z B
xor Z C
xor Z D
xor Z E
```

Easy task:

Assume that the register `A` contains an arbitrary input number, and that all the other registers contain zeroes. Write a program that will increase the value in `A` by 8 (computing modulo 2^{64}, if necessary). After your program terminates, the other registers are allowed to contain anything. Your program must have less than 64 instructions.

Hard task:

Assume that the register `A` contains an arbitrary input number, and that all the other registers contain zeroes. Write a program that will change the value in register `A` into the next larger value with the same number of set bits. After your program terminates, the other registers are allowed to contain anything. Your program must have less than 300 instructions.

In other words, if we regard `A` as a sequence of 0s and 1s, your task is to produce the next permutation of the given sequence.

For example, if the input is 23, which is 10111 in binary, the output should be 27, which is 11011 in binary. If the input is 24, which is 011000 in binary, the output should be 33, which is 100001.

You may assume that the output for the given number in `A` is always less than 2^{64}. I.e., the input will be such that the next larger value with the same number of set bits still fits into the 64-bit register.

Input specification

This problem has no input.

Output specification

Send us your program as a text file. Each line of the file may either contain one complete instruction, or just whitespace.

Jumbo Airlines

The new catchphrase of Jumbo Airlines is "No annoying neighbors, each flight a unique experience!"

And as in most cases, the advertisement was produced by the marketing department, without ever consulting the engineers. They only learned about it after the boss asked them to "handle it ASAP".

There are M seats in each row, and there are N rows of seats in the airplane. Hence the seats form an $M \times N$ grid. (For the purpose of this problem we will ignore the presence of aisles.) The airline sells exactly K tickets for each flight.

To make sure that the "no annoying neighbors" part of the motto is satisfied, the seating must obey the following rule: Whenever a seat is occupied, the seats immediately in front of it and behind it, as well as the seats immediately to the left and to the right must remain free.

An *allowed arrangement* is a set of K occupied seats that obeys the rule above.

The "unique experience" part of the motto is then satisfied by using a different arrangement of occupied seats for each flight. (Two seating arrangements are different if there is at least one seat which is occupied in one arrangement and free in the other.)

Problem specification

You are given the numbers M, N and K. Find the number of different allowed seating arrangements.

As this number can be very large, we're only interested in its value modulo 420 047.

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 three integers M, N and K.

Output specification

For each test case output a single line with the number of allowed arrangements modulo 420 047.

Example

input

```
3

2 3 2

2 4 4

2 5 1
```

output

```
8
2
10
```

Karl's shopping

Karl is going to spend his holiday in Nothingland. Since there is nothing there, he has to buy all supplies now. At the moment, he is waiting at the checkout counter with a shopping cart full of stuff.

Of course, he has a sufficient amount of money in his wallet. However, he prefers to use alternate means of payment if possible: luncheon vouchers, gift certificates, different types of coupons, etc. What makes the matter complicated is that the use of these items is often limited: e.g., luncheon vouchers can only be used to buy food, and gift certificates are often limited to a certain type of gifts.

Problem specification

You are given the number N of items in Karl's shopping cart and their prices. You are also given the number M of vouchers in his wallet, together with the information on their allowed use.

When paying for his shopping, Karl may use vouchers for a larger sum than the cost of the things he is buying. It is also possible to split an item's cost between multiple vouchers and use a voucher to pay for more than one item.

Compute the minimum amount of additional cash money Karl needs to pay for his shopping.

Input specification

The first line of input file contains an integer T specifying the number of test cases. T blocks follows, each block describes one test case. Each block is preceded by a blank line.

Each block starts with line containing two positive integers N (the number of items) and M (the number of vouchers). The second line contains N numbers, the i-th of them being the price of the i-th item in Karl's shopping cart. The third line contains M numbers, the i-th of them being the cash value of the i-th voucher Karl has in his wallet. M lines follow. Each line consists of a number K_i (the count of items such that you can pay for them using the i-th voucher) followed by K_i numbers (the numbers of those items; items are numbered from 1 to N).

Output specification

For each test case output a single number specifying how much cash money Karl needs to pay for his shopping.

Example

input

```
1

3 2
15 20 10
20 30
3 1 2 3
1 3
```

output

```
15
```

Let there be rainbows!

Once upon a time there was the kingdom of Absurdistan. There were N cities in the kingdom.

As it is often the case, some pairs of cities were connected by roads. The Great Vizier of Absurdistan was a terrible scrooge, hence the road network was a tree. (I.e., for any pair of cities there was exactly one way how to get from one to the other, while travelling by roads.)

The roads were made of concrete blocks and they all had a dull grey color.

One day, the Great Vizier came to the conclusion that travelling along dull grey roads makes people unhappy. As he does not want to risk a revolution, he decided to paint the roads using happy colors – the 7 colors of the rainbow.

Each day, he picked two cities s_i and t_i, and a color c_i. Painters with an ample supply of the color c_i were sent to the city s_i, and then they travelled to t_i. On their way, whenever they encountered a road that did not have the color c_i, they painted it to this color.

Problem specification

You are given the number of cities N, the description of the road network, and the sequence of Great Vizier's orders.

For each order, compute the number of roads the painters had to repaint.

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 line containing a positive integer N ($N \leq 1\,000\,000$). For convenience, the cities in Absurdistan are numbered 0 to $N-1$.

$N-1$ lines follow. Each of these lines contains two numbers a_i b_i ($0 \leq a_i, b_i < N$) – the endpoints of one road.

The next line contains a single integer Q ($Q \leq 200\,000$) specifying the number of orders.

Q lines follow, each describing one order. Each description consists of two integers s_i t_i ($0 \leq s_i, t_i < N$) and a string c_i (c_i is one of `red`, `orange`, `yellow`, `green`, `blue`, `indigo`, and `violet`).

In the easy input, you may also assume that for each test case except for one we have $NQ \leq 10^8$. In the remaining test case the road network is a simple line.

Output specification

For each test case output seven lines. Each of these lines should contain a color name (in the order given above) and a number. The number shall be the total number of times some road was painted using the corresponding color.

You may output empty lines between the test cases. (The amount of whitespace in your output does not matter anyway; we do this for readability in the example below.)

Example

input

```
2

8
0 4
7 5
5 6
3 5
1 4
4 2
4 3
4
0 7 red
2 6 orange
4 3 green
1 7 green

12
0 1
1 2
2 3
2 4
2 5
0 6
6 7
7 8
7 9
7 10
11 6
3
3 8 blue
6 1 violet
5 11 blue
```

output

```
red 4
orange 4
yellow 0
green 4
blue 0
indigo 0
violet 0

red 0
orange 0
yellow 0
green 0
blue 10
indigo 0
violet 2
```

We will explain the second test case in detail:

When executing the query "3 8 blue", we paint six roads blue: 3-2, 2-1, 1-0, 0-6, 6-7, and 7-8.

When executing the query "6 1 violet", we paint two roads violet: 6-0 and 0-1. (Both of these roads were previously blue.)

Finally, when executing the query "5 11 blue", we paint four roads blue: 5-2, 1-0, 0-6, and 6-11. Note that the road 2-1 was still blue, hence we did not paint it now.

Thus we painted a road blue 6+4=10 times, and we painted a road violet 2 times.

Muzidabutur

Muzidabutur is a very powerful magic phrase. For example, if you write the phrase on a monitor using a non-erasable marker, your local computer administrator will throw you into a waste bin (and we do not mean the folder). The problem is that nobody really remembers this phrase anymore.

The famous archeologist Alabama Steve recently discovered two ancient tomes. In one of them he found a short description of the Muzidabutur. He suspects that one of the sentences in the other tome is Muzidabutur. But the tome is way too thick and he does not have the time to process it by hand.

Problem specification

You are given a description of Muzidabutur, and N queries. For each query decide whether it can be the Muzidabutur.

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 line containing a textual description of the Muzidabutur. You may assume that spaces and characters '`(`' and '`)`' will not occur inside strings and character sets used to define the Muzidabutur.

The next line contains N, which is the number of sentences in the second tome. Each of the next N lines contains one sentence.

Output specification

For each sentence output the string "`YES`" if it exactly matches the description of a Muzidabutur, and output "`NO`" otherwise.

Example

input

```
1

THE LETTER A FOLLOWED BY (ONE OF THE LETTERS abc OR THE STRING gg) AT LEAST 3 TIMES
4
Abggaggc
abbb
Abc
Hello, Abggaggc, how are you?
```

output

```
YES
NO
NO
NO
```

Note that the Muzidabutur is case sensitive, therefore the second sentence is not a Muzidabutur.
Also, the description must match the entire sentence, therefore the last sentence is not a Muzidabutur.