Problem A

Pizza delivery

source: pizza.*

Your Irish pizza and kebab restaurant is doing very well. Not only is the restaurant full almost every night, but there is also an ever increasing number of deliveries to be made, all over town. To meet this demand, you realize that it will be necessary to separate the delivery service from the restaurant. A new large kitchen, only for baking pizzas and being a base for deliveries, has to be established somewhere in town.

The main cost in the delivery service is not the making of the pizza itself, but the time it takes to deliver it. To minimize this, you need to carefully plan the location of the new kitchen. To your help you have a database of all last year's deliveries. For each block in the city, you know how many deliveries were made there last year. The kitchen location will be chosen based on the assumption that the pattern of demand will be the same in the future.

Your city has a typical suburban layot – an orthogonal grid of equal-size square blocks. All places of interest (delivery points and the kitchen) are considered to be located at street crossings. The distance between two street crossings is the Manhattan distance, i.e., the number of blocks you have to drive vertically, plus the number of blocks you have to drive horizontally. The total cost for a delivery point is its Manhattan distance from the kitchen, times the number of deliveries to the point. Note that we are only counting the distance *from* the kitchen *to* the delivery point. Even though we always drive directly back to the kitchen after a delivery is made, this (equal) distance is not included in the cost measure.

Input specifications

On the first line, there is a number, $1 \leq n \leq 20$, indicating the number of test cases. Each test case begins with a line with two integers, $1 \leq x \leq 100$, $1 \leq y \leq 100$, indicating the size of the two-dimenstional street grid. Then follow y lines, each with x integers, $0 \leq d \leq 1000$, indicating the number of deliveries made to each street crossing last year.

Output specifications

For each test case, output the least possible total delivery cost (the sum of all delivery costs last year), assuming that the kitchen was located optimally. There should be one line for each test case, with an integer indicating the cost, followed by a single space and the word 'blocks'.

Sample input

```
2
4 4
0 8 2 0
1 4 5 0
0 1 0 1
3 9 2 0
6 7
0 0 0 0 0 0
0 1 0 3 0 1
2 9 1 2 1 2
8 7 1 3 4 3
1 0 2 2 7 7
0 1 0 0 1 0
0 0 0 0 0 0
```

Output for sample input

```
55 blocks
162 blocks
```

Problem B

Ball bearings

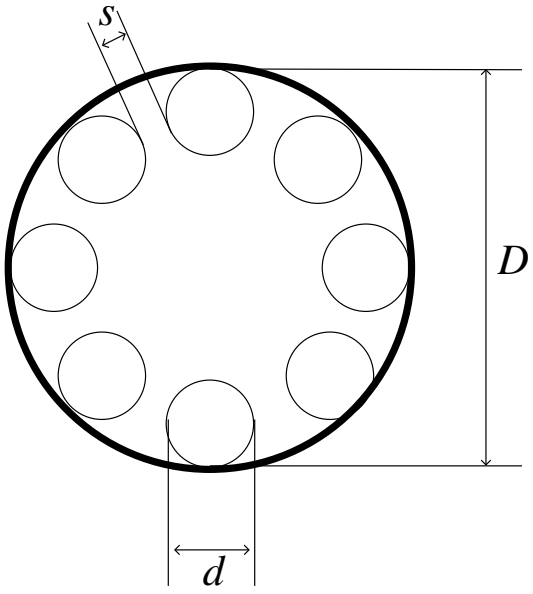
source: ballbearings.*

The Swedish company SKF makes ball bearings. As explained by Britannica Online, a ball bearing is

"one of the two types of rolling, or anti friction, bearings (the other is the roller bearing).

Its function is to connect two machine members that move relative to one another so that the frictional resistance to motion is minimal. In many applications, one of the members is a rotating shaft and the other a fixed housing. Each ball bearing has three main parts: two grooved, ring like races and a number of balls. The balls fill the space between the two races and roll with negligible friction in the grooves. The balls may be loosely restrained and separated by means of a retainer or cage."

Presumably, the more balls you have inside the outer ring, the smoother the ride will be, but how many can you fit within the outer ring? You will be given the inner diameter of the outer ring, the diameter of the balls, and the minimum distance between neighboring balls. Your task is to compute the maximum number of balls that will fit on the inside of the outer ring (all balls must touch the outer ring).



Input specifications

The first line of input contains a positive integer n that indicates the number of test cases. Then follow n lines, each describing a test case. Each test case consists of three positive floating point numbers, D, d, s, where D is the inner diameter of the outer ring, d is the diameter of a ball, and s is the minimum distance between balls. All parameters are in the range $[0.0001, 500.0]$.

Output specifications

For each test case output a single integer m on a line by itself, where m is the maximum number of balls that can fit in the ball bearing, given the above constraints. There will always be room for at least three balls.

Sample input

```
2
20 1 0.1
100.0 13.0 0.2
```

Output for sample input

```
54
20
```

Problem C

S-Nim

source: nim.*

Arthur and his sister Caroll have been playing a game called Nim for some time now. Nim is played as follows:

- The starting position has a number of heaps, all containing some, not necessarily equal, number of beads.
- The players take turns chosing a heap and removing a positive number of beads from it.
- The first player not able to make a move, loses.

Arthur and Caroll really enjoyed playing this simple game until they recently learned an easy way to always be able to find the best move:

- Xor the number of beads in the heaps in the current position (i.e. if we have 2, 4 and 7 the *xor-sum* will be 1 as 2 **xor** 4 **xor** 7 = 1).
- If the xor-sum is 0, too bad, you will lose.
- Otherwise, move such that the xor-sum becomes 0. This is always possible.

It is quite easy to convince oneself that this works. Consider these facts:

- The player that takes the last bead wins.
- After the winning player's last move the xor-sum will be 0.
- The xor-sum will change after every move.

Which means that if you make sure that the xor-sum always is 0 when you have made your move, your opponent will never be able to win, and, thus, you will win.

Understandibly it is no fun to play a game when both players know how to play perfectly (ignorance is bliss). Fourtunately, Arthur and Caroll soon came up with a similar game, S-Nim, that seemed to solve this problem. Each player is now only allowed to remove a number of beads in some predefined set S, e.g. if we have $S = \{2, 5\}$ each player is only

allowed to remove 2 or 5 beads. Now it is not always possible to make the xor-sum 0 and, thus, the strategy above is useless. Or is it?

Your job is to write a program that determines if a position of S-Nim is a losing or a winning position. A position is a winning position if there is at least one move to a losing position. A position is a losing position if there are no moves to a losing position. This means, as expected, that a position with no legal moves is a losing position.

Input specifications

Input consists of a number of test cases.

For each test case: The first line contains a number k ($0 < k \leq 100$) describing the size of S, followed by k numbers s_i ($0 < s_i \leq 10000$) describing S. The second line contains a number m ($0 < m \leq 100$) describing the number of positions to evaluate. The next m lines each contain a number l ($0 < l \leq 100$) describing the number of heaps and l numbers h_i ($0 \leq h_i \leq 10000$) describing the number of beads in the heaps.

The last test case is followed by a 0 on a line of its own.

Output specifications

For each position:

- If the described position is a winning position print a 'W'.
- If the described position is a losing position print an 'L'.

Print a newline after each test case.

Sample input

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

Output for sample input

```
LWW
WWL
```

Problem D

Sylvester construction

source: sylvester.*

A Hadamard matrix of order n is an $n \times n$ matrix containing only 1s and -1s, called H_n, such that $H_n H_n^T = nI_n$ where I_n is the $n \times n$ identity matrix. An interesting property of Hadamard matrices is that they have the maximum possible determinant of any $n \times n$ matrix with elements in the range $[-1, 1]$. Hadamard matrices have applications in error-correcting codes and weighing design problems.

The Sylvester construction is a way to create a Hadamard matrix of size $2n$ given H_n. H_{2n} can be constructed as:

$$H_{2n} = \begin{pmatrix} H_n & H_n \\ H_n & -H_n \end{pmatrix}$$

For example:

$$H_1 = (1)$$

$$H_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

and so on.

In this problem you are required to print a part of a Hadamard matrix constructed in the way described above.

Input specifications

The first number in the input is the number of test cases to follow. For each test case there are five integers: n, x, y, w and h. n will be between 1 and 2^{62} (inclusive) and will be a power of 2. x and y specify the upper left corner of the sub matrix to be printed, w and h specify the width and height respectively. Coordinates are zero based, so $0 \leq x,y < n$. You can assume that the sub matrix will fit entirely inside the whole matrix and that $0 < w,h \leq 20$. There will be no more than 1000 test cases.

Output specifications

For each test case print the sub matrix followed by an empty line.

Sample input

```
3
2 0 0 2 2
4 1 1 3 3
268435456 12345 67890 11 12
```

Output for sample input

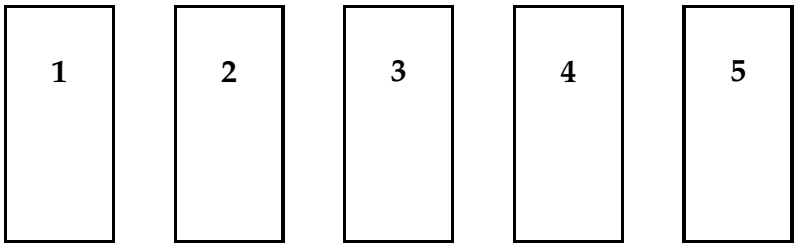
```
1 1
1 -1

-1 1 -1
1 -1 -1
-1 -1 1

1 -1 -1 1 1 -1 -1 1 1 -1 -1
-1 -1 1 1 -1 -1 1 1 -1 -1 1
1 1 1 -1 -1 -1 -1 1 1 1 1
-1 1 -1 -1 1 -1 1 1 -1 1 -1
1 -1 -1 -1 -1 1 1 1 1 -1 -1
-1 -1 1 -1 1 1 -1 1 -1 -1 1
-1 -1 -1 -1 -1 -1 -1 1 1 1 1
1 -1 1 -1 1 -1 1 1 -1 1 -1
-1 1 1 -1 -1 1 1 1 1 -1 -1
1 1 -1 -1 1 1 -1 1 -1 -1 1
-1 -1 -1 1 1 1 1 1 1 1 1
1 -1 1 1 -1 1 -1 1 -1 1 -1
```

Problem E

Who owns the Amiga?

source: amiga.*

In a corridor in a student dormitory, there are five rooms numbered **1**, **2**, **3**, **4** and **5**; room number **1** is the left-most room. The rooms have doors in different colours: **blue**, **green**, **red**, **white** and **yellow**, but not necessarily in that order.



In these rooms live five students **Anna**, **Bernhard**, **Chris**, **David** and **Ellen** of five different nationalities **Danish**, **Finnish**, **Icelandic**, **Norwegian** and **Swedish**. (Both the names and the nationalities are given in alphabetical order, so it does *not* follow automatically that Anna is Danish.)

These students have one computer each, and these computers are of different kinds: **Amiga**, **Atari**, **Linux**, **Mac** and **Windows** (given here in alphabetical order). They each have their own favourite programming language: **C**, **C++**, **Java**, **Pascal** and **Perl** (also listed in alphabetical order).

You want to find out who owns the Amiga computer based on some facts about the students.

Input specifications

The input consists of several scenarios. The first input line contains a number 1–1000 indicating how many scenarios there are.

Each scenario starts with a line with a number 1–2000 telling how many fact lines there are for that scenario. Then follow the fact lines which each contains three words separated by one or more spaces:

- The first and third word is one of these names:

  1 2 3 4 5
  blue green red white yellow
  anna bernhard chris david ellen

danish finnish icelandic norwegian swedish
amiga atari linux mac windows
c c++ java pascal perl

(Note that no uppercase letters are used.)

- The second word specifies a relationship; it is one of

  same-as left-of right-of next-to

  same-as tells that the first and third fact words apply to the same room; for instance

  blue same-as bernhard

  tells that Bernhard lives in the room with a blue door.

  left-of tells that the first fact word applies to the room immediately to the left of the one to which the third fact word applies. For example,

  chris left-of perl

  means that Chris lives in the room immediately to the left of the Perl programmer.

  right-of tells that the first fact word applies to the room immediately to the right of the one to which the third fact word applies.

  next-to tells that the two fact words apply to rooms next to each other. For example,

  swedish next-to linux

  means that the Swedish student lives in the next room (either to the left or the right) of the owner of the Linux computer.

You may assume that there are no inconsistencies in the input data. In other words, there will in every scenario be at least one person who may own the Amiga without violating the constraints.

Output specifications

For each scenario, you should print a line starting with

`scenario #`*`n`*`:`

where *n* is the scenario number. If you can determine who (i.e., Anna, Bernhard, Chris, David or Ellen) owns the Amiga, you continue the line with

`xxxx` `owns the amiga.`

or, if you cannot name the Amiga owner, you print

`cannot identify the amiga owner.`

Sample input

```
2
8
red same-as 1
danish same-as 1
perl same-as 5
atari same-as 2
linux same-as 3
mac same-as 4
windows same-as 5
anna same-as 1
8
chris left-of amiga
chris left-of 4
c same-as 1
danish same-as 1
red same-as 1
linux same-as red
windows next-to linux
mac left-of swedish
```

Output for sample input

```
scenario #1: anna owns the amiga.
scenario #2: cannot identify the amiga owner.
```

This page is intentionally left blank.

Problem F

Lazy Evaluation

source: lazy.*

Most of the programming languages used in practice perform *strict-evaluation*. When a function is called, the expressions passed in the function arguments (for instance $a + b$ in $f(a + b, 3)$) are evaluated first, and the resulting values are passed to the function.

However, this is not the only way how expressions can be evaluated, and in fact, even such a mainstream language as C++ is sometimes performing *lazy evaluation*: the operators && and || evaluate only those arguments that are needed to determine the value of the expression.

Pål Christian is now working on a comparative study of the performance of the lazy and strict evaluation. Pål wants to evaluate in both ways a set of expressions that follow this simplified syntax:

- an expression is either a *constant*, a *name*, or a *function call*
- a *constant* is a signed 32-bit integer; for example, 3, 0, -2, 123 are constants;
- a *name* is a name of a built-in or user-defined function, for example, f, or *add* are names; names are words containing up to 32 lowercase letters from the English alphabet;
- *function call* has the form: $(\ function\ arg_1\ \ldots\ arg_N\ )$, where $function$ is an expression that evaluates to some function of N arguments, and $arg_1 \ldots arg_N$ are expressions that evaluate to arguments passed to the function. For example, $(\ f\ 3\ 5\ )$, or $(\ add\ 2\ (\ add\ 1\ 2\ ))$ are valid function calls.

Expressions are evaluated according to the following simple rules:

- *constants* evaluate to themselves
- *names* evaluate to the functions they denote
- *function calls*:

  in lazy evaluation: the first expression is evaluated first to obtain a function, whose function body, with formal parameters substituted for the expressions provided as the arguments, is evaluated; however, whenever some argument gets evaluated while evaluating the function body, the resulting value will replace all occurences of the same parameter in that function body. In other words, the expression passed in the argument is never evaluated more than once.

in strict evaluation: all expressions are evaluated first: the first expression should evaluate to a function, the remaining to values that are used as function arguments; the result is the result of evaluating the corresponding function body, where all occurences of formal parameters are replaced by the values obtained by evaluating the arguments.

The following built-in functions are available: *add x y* - sum of the constants x and y, *sub x y* - returns the value $x - y$, *mult x y* - product of x and y, *div x y* - integer division, and *rem x y* - remainder (same semantics as '/' and '%' in C, C++ or Java), *true x y* - always returns x, $false$ *x y* - always returns y , *eq x y* - returns *true* if x and y is the same constant, otherwise returns $false$, *gt x y* - returns *true* if x is greater than y, otherwise returns $false$.

User-defined functions are defined using the following syntax: $function\_name\ arg_1\ \ldots\ arg_N\ =\ body$, where $arg_1 \ldots arg_N$ are distinct words (up to 32 English lowercase letters), denoting the formal parameters of the function, and the body is an expression. The formal parameters can occur in the body of the function in place of constants or names. The function name and the formal parameters are separated by a single space. There is one space on both sides of the "=". Functions with zero (no) arguments are legal. Note that the formal parameters can overshadow the function names (i.e. *op* in definition of *not* in sample input overshadows the function name *op*), but each function must have a unique name.

Input specifications

The first part of the input contains (less than 1000) lines with one function definition each, followed by a single empty line. Forward references (that is, referring to functions defined later in the input) and recursion are legal.

The second part of the input contains less than 1000 *test expressions*. Each test expression is an expresion occupying a single line. Function names and the arguments are always separated by a single space, but there are no extra spaces around parentheses (see sample input). There is an empty line after the last expression. Expressions are to be evaluated by both the lazy and the strict evaluation.

You can assume that all function definitions and expressions are syntactically correct, and that the arithmetic built-in functions (*add*, *sub*, *mult*, *div*, *rem*, *eq*, *gt*) will always be called with integers only, and no division by 0 occurs. Overflows outside the 32-bit integer range are legal and do not require any special treatment (just use the value produced by C, C++, or Java operators +, -, *, /, or %). In strict evaluation, built-in functions evaluate all their arguments too. In lazy evaluation, arithmetic built-in functions always evaluate all their arguments. All lines on the input contain no more than 255 characters including spaces.

Output specifications

The program should produce a table in exactly the following format:

operator	lazy_evaluation	strict_evaluation
add	add_{lazy}	add_{strict}
sub	sub_{lazy}	sub_{strict}
mult	$mult_{lazy}$	$mult_{strict}$
div	div_{lazy}	div_{strict}
rem	rem_{lazy}	rem_{strict}

where each op_{lazy} is an integer - how many times *op* has been executed in lazy evaluation of all expressions, and op_{strict} is the number of evaluations of *op* in strict evaluation. Spaces can occur arbitrarily. If the evaluation of a *test expression* does not terminate after a total of 2345 function evaluations, you can assume that it is in an infinite loop, the program should skip that expression, and do not count it into the totals (omit counting operations both in lazy and strict evaluation of this expression).

Sample input

```
if cond truepart elsepart = (cond truepart elsepart)
fact x = (facta x 1)
facta x a = (if (eq x 0) a (facta (sub x 1) (mult a x)))
and x y = (x y false)
ident x = x
two = 2
op op x = ((if (eq op 1) add sub) op x)
not op = (op false true)
sum n = (suma n 0)
suma n a = (((gt n 1) suma false) (sub n 1) (add a n))

(true (add 1 2) (mult 1 2))
5
true
(and (gt (op (sub 2 1) 1) 5) (eq (two) (op 1 1)))
(false (sub 1 2) (sum 4))
((eq (true 1 2) (false 2 1)) (add 1 2) (sub 1 2))
(fact 3)
```

Output for sample input

```
operator lazy_evaluation strict_evaluation
add             7               8
sub             4               7
mult            0               1
div             0               0
rem             0               0
```

This page is intentionally left blank.

Problem G

Easter holidays

source: holidays.*

Scandinavians often make vacation during the Easter holidays in the largest ski resort Åre. Åre provides fantastic ski conditions, many ski lifts and slopes of various difficulty profiles. However, some lifts go faster than others, and some are so popular that a queue forms at the bottom.

Per is a beginner skier and he is afraid of lifts, even though he wants to ski as much as possible. Now he sees that he can take several different lifts and then many different slopes or some other lifts, and this freedom of choice is starting to be too puzzling...

He would like to make a ski journey that:

- starts at the bottom of some lift and ends at that same spot
- has only two phases: in the first phase, he takes one or more lifts up, in the second phase, he will ski all the way down back to where he started
- is least scary, that is the ratio of the time spent on the slopes to the time spent on the lifts or waiting for the lifts is the largest possible.

Can you help Per find the least scary ski journey?

A ski resort contains n places, m slopes, and k lifts ($2 \leq n \leq 1000$, $1 \leq m \leq 1000$, $1 \leq k \leq 1000$). The slopes and lifts always lead from some place to another place: the slopes lead from places with higher altitude to places with lower altitude and lifts vice versa (lifts cannot be taken downwards).

Input specifications

The first line of the input contains the number of cases – the number of ski resorts to process. Each ski resort is described as follows: the first line contains three integers n, m, and k. The following m lines describe the slopes: each line contains three integers – top and bottom place of the slope (the places are numbered 1 to n), and the time it takes to go down the slope (max. 10000). The final k lines describe the lifts by three integers – the bottom and top place of the lift, and the time it takes to wait for the lift in the queue and be brought to its top station (max. 10000). You can assume that no two places are connected by more than one lift or by more than one slope.

Output specifications

For each input case, the program should print two lines. The first line should contain a space-separated list of places in the order they will be visited – the first place should be the same as the last place. The second line should contain the ratio of the time spent in the slopes to the time spent on the lifts or wating for the lifts. The ratio should be rounded to the closest 1/1000th. If there are two possibilities, then the rounding is away from zero (e.g., 1.9812 and 1.9806 become 1.981, 3.1335 becomes 3.134, and 3.1345 becomes 3.135). If there are multiple journeys that prior to rounding are equally scary, print an arbitrary one.

Sample input

```
1
5 4 3
1 3 12
2 3 6
3 4 9
5 4 9
4 5 12
5 1 12
4 2 18
```

Output for sample input

```
4 5 1 3 4
0.875
```

Problem H

Tourist

source: tourist.*

A lazy tourist wants to visit as many interesting locations in a city as possible without going one step further than necessary. Starting from his hotel, located in the north-west corner of city, he intends to take a walk to the south-east corner of the city and then walk back. When walking to the south-east corner, he will only walk east or south, and when walking back to the north-west corner, he will only walk north or west. After studying the city map he realizes that the task is not so simple because some areas are blocked. Therefore he has kindly asked you to write a program to solve his problem.

Given the city map (a 2D grid) where the interesting locations and blocked areas are marked, determine the maximum number of interesting locations he can visit. Locations visited twice are only counted once.

Input specifications

The first line in the input contains the number of test cases (at most 20). Then follow the cases. Each case starts with a line containing two integers, W and H ($2 \leq W, H \leq 100$), the width and the height of the city map. Then follow H lines, each containing a string with W characters with the following meaning:

'.'	Walkable area
'*'	Interesting location (also walkable area)
'#'	Blocked area

You may assume that the upper-left corner (start and end point) and lower-right corner (turning point) are walkable, and that a walkable path of length $H + W - 2$ exists between them.

Output specifications

For each test case, output a line containing a single integer: the maximum number of interesting locations the lazy tourist can visit.

Sample input

```
2
9 7
*........
.....**#.
..**...#*
..#####*#.
.*.#*.*#.
...#**...
*........
5 5
.*.*.
*###.
*.*.*
.###*
.*.*.
```

Output for sample input

```
7
8
```