Problem A

The Mailbox Manufacturers Problem

source: mailbox.*

In the good old days when Swedish children were still allowed to blow up their fingers with fire-crackers, gangs of excited kids would plague certain smaller cities during Easter time, with only one thing in mind: To blow things up. Small boxes were easy to blow up, and thus mailboxes became a popular target. Now, a small mailbox manufacturer is interested in how many fire-crackers his new mailbox prototype can withstand without exploding and has hired you to help him. He will provide you with k $(1 \leq k \leq 10)$ identical mailbox prototypes each fitting up to m $(1 \leq m \leq 100)$ crackers. However, he is not sure of how many fire-crackers he needs to provide you with in order for you to be able to solve his problem, so he asks you. You think for a while and then say: "Well, if I blow up a mailbox I can't use it again, so if you would provide me with only $k = 1$ mailboxes, I would have to start testing with 1 cracker, then 2 crackers, and so on until it finally exploded. In the worst case, that is if it does not blow up even when filled with m crackers, I would need $1 + 2 + 3 + ... + m = m * (m + 1)/2$ crackers. If $m = 100$ that would mean more than 5000 fire-crackers!". "That's too many", he replies. "What if I give you more than $k = 1$ mailboxes? Can you find a strategy that requires less crackers?"

Can you? And what is the minimum number of crackers that you should ask him to provide you with?

You may assume the following:

1. If a mailbox can withstand x fire-crackers, it can also withstand $x-1$ fire-crackers.

2. Upon an explosion, a mailbox is either totally destroyed (blown up) or unharmed, which means that it can be reused in another test explosion.

Note: If the mailbox can withstand a full load of m fire-crackers, then the manufacturer will of course be satisfied with that answer. But otherwise he is looking for the maximum number of crackers that his mailboxes can withstand.

Input specifications

The input starts with a single integer N $(1 \leq N \leq 10)$ indicating the number of test cases to follow. Each test case is described by a line containing two integers: k and m, separated by a single space.

Output specifications

For each test case print one line with a single integer indicating the minimum number of fire-crackers that is needed, in the worst case, in order to figure out how many crackers the mailbox prototype can withstand.

Sample input

```
4
1 10
1 100
3 73
5 100
```

Output for sample input

```
55
5050
382
495
```

Problem B

Parallel computer simulator

source: parallel.*

Programs executed concurrently on a uniprocessor system appear to be executed at the same time, but in reality the single CPU alternates between the programs, executing some number of instructions from each program before switching to the next. You are to simulate the concurrent execution of up to ten programs on such a system and determine the output that they will produce.

The program that is currently being executed is said to be **running**, while all programs awaiting execution are said to be **ready**. A program consists of a sequence of no more than 200 statements, one per line, followed by an end statement. The statements available are listed below.

Statement type	Syntax
Assignment	$\langle \textit{variable} \rangle = \langle \textit{constant} \rangle$
Output	print $\langle \textit{variable} \rangle$
Begin mutual exclusion	lock
End mutual exclusion	unlock
Stop execution	end

A $\langle \textit{variable} \rangle$ is any single lowercase alphabetic character and a $\langle \textit{constant} \rangle$ is an unsigned decimal number less than 1000. There are only 26 variables in the computer system, and they are shared among the programs. Thus assignments to a variable in one program affect the value that might be printed by a different program. All variables are initially set to zero.

Each statement requires an integral number of time units to execute. The running program is permitted to continue executing instructions for a period of time called its **quantum**. When a program's time quantum expires, another ready program will be selected to run. Any instruction currently being executed when the time quantum expires will be allowed to complete.

Programs are queued first-in-first-out for execution in a **ready queue**. The initial order of the ready queue corresponds to the original order of the programs in the input file. This order can change, however, as a result of the execution of lock and unlock statements.

The lock and unlock statements are used whenever a program wishes to claim mutually exclusive access to the variables it is manipulating. These statements always occur in pairs, bracketing one or more other statements. A `lock` will always precede an `unlock`, and these statements will never be nested. Once a program successfully executes a `lock` statement, no other program may successfully execute a `lock` statement until the locking program runs and executes the corresponding `unlock` statement. Should a running program attempt to execute a `lock` while one is already in effect, this program will be placed at the end of the **blocked queue**. Programs blocked in this fashion lose any of their current time quantum remaining. When an `unlock` is executed, any program at the head of the blocked queue is moved to the head of the ready queue. The first statement this program will execute when it runs will be the `lock` statement that previously failed. Note that it is up to the programs involved to enforce the mutual exclusion protocol through correct usage of `lock` and `unlock` statements. (A renegade program with no `lock`/`unlock` pair could alter any variables it wished, despite the proper use of `lock`/`unlock` by the other programs.)

Input specifications

The first line of the input file consists of seven integers separated by spaces. These integers specify (in order): the number of programs which follow, the unit execution times for each of the five statements (in the order given above), and the number of time units comprising the time quantum. The remainder of the input consists of the programs, which are correctly formed from statements according to the rules described above.

All program statements begin in the first column of a line. Blanks appearing in a statement should be ignored. Associated with each program is an identification number based upon its location in the input data (the first program has ID=1, the second has ID=2, etc.).

Output specifications

Your output will contain the output generated by the print statements as they occur during the simulation. When a print statement is executed, your program should display the program ID, a colon, a space, and the value of the selected variable. Output from separate print statements should appear on separate lines. A sample input and correct output is shown below.

Sample input

```
3 1 1 1 1 1 1
a = 4
print a
lock
b = 9
print b
```

```
unlock
print b
end
a = 3
print a
lock
b = 8
print b
unlock
print b
end
b = 5
a=17
print a
print b
lock
b = 21
print b
unlock
print b
end
```

Output for sample input

```
1: 3
2: 3
3: 17
3: 9
1: 9
1: 9
2: 8
2: 8
3: 21
3: 21
```

Problem C

Moving Pianos

source: piano.*

A piano-tuner indeed has a hard time earning his living. (Un)fortunately, most people want their pianos tuned only when they move to a new place, and thus also someone to help them move the piano. Therefore, quite a few piano-tuners combine their profession with lifting and transporting pianos. However, as you might guess, they are not all that keen on doing a lot of heavy-weight carrying in a day, and they certainly don't want to work weekends! In the city of Hampstead, the small group of piano-tuners have joined forces, and they help each other by taking turns in moving the pianos. No piano-tuner transports more than one piano a day under any circumstance, and they always work in pairs. As with most craftsmen, you seldom get to know precisely when the job you request will be carried out. Instead, when ordering a piano move, you are given an arbitrary long time interval containing your desired moving day, during which your piano is guaranteed to be moved by the piano-tuners. This strategy often makes the scheduling task easy, but this time they just might have put themselves in some trouble. During the last days, the piano-tuners have received a lot of piano move orders for the next 100 days, and they are afraid that even if they work weekends they cannot keep their promises. To help them see if this is the case, and to avoid it in the future, they would like a computer program testing different scenarios.

Input specifications

On the first line is a single positive integer n, specifying the number of test scenarios to follow. Each scenario begin with two positive integers m and p on a line of their own, $m \leq 1000$, telling the number of pianos to be moved, and $p \leq 2000$ the number of piano-tuners. Then follow m lines, each containing a piano move order on the format of two integers b_i and e_i with $1 \leq b_i \leq e_i \leq 100$ meaning that between the beginning of day b_i and the end of day e_i, the piano i must be moved. Day 1 is always a Monday, 2 a Tuesday and so on.

Output specifications

For each test scenario, output the text "`serious trouble`" if the list of piano move orders cannot be executed as promised even if all the p piano-tuners worked everyday, the text "`weekend work`" if at least two piano-tuners have to work some Saturday or Sunday, or the text "`fine`" if it is possible to carry out all the moves during the five working days of the weeks.

Sample input

```
3
3 2
4 5
5 6
5 7
2 5
49 50
50 50
4 2
8 10
8 10
8 10
8 10
```

Output for sample input

```
weekend work
fine
serious trouble
```

Problem D

Collecting Beepers

source: `beepers.*`

Karel is a robot who lives in a rectangular coordinate system where each place is designated by a set of integer coordinates (x and y). Your job is to design a program that will help Karel pick up a number of beepers that are placed in her world. To do so you must direct Karel to the position where each beeper is located. Your job is to write a computer program that finds the length of the shortest path that will get Karel from her starting position, to each of the beepers, and return back again to the starting position.

Karel can only move along the x and y axis, never diagonally. Moving from one position (i, j) to an adjacent position $(i, j+1)$, $(i, j-1)$, $(i-1, j)$, or $(i+1, j)$ has a cost of one.

You can assume that Karel's world is never larger than 20×20 squares and that there will never be more than 10 beepers to pick up. Each coordinate will be given as a pair (x, y) where each value will be in the range 1 through the size of that particular direction of the coordinate system.

Input specifications

First there will be a line containing the number of scenarios you are asked to help Karel in. For each scenario there will first be a line containing the size of the world. This will be given as two integers (x-size and y-size). Next there will be one line containing two numbers giving the starting position of Karel. On the next line there will be one number giving the number of beepers. For each beeper there will be a line containing two numbers giving the coordinates of each beeper.

Output specifications

The output will be one line per scenario, giving the minimum distance that Karel has to move to get from her starting position to each of the beepers and back again to the starting position.

Sample input

```
1
10 10
1 1
4
2 3
5 5
9 4
6 5
```

Output for sample input

```
The shortest path has length 24
```

Problem E

Quantum

source: `quantum.*`

At the Institution for Bits and Bytes at University of Ramville, Prof. Jeremy Longword and his eight graduate students are investigating a brand new way of storing and manipulating data on magnetic disks for use in hard drives. The method is based on letting quasimagnetic quantum operations operate on the sectors on the disk, and is, of course, safer and more reliable than any earlier invented storage method. The use of each quantum operation costs a certain amount of energy, and the more energy the storage unit consumes, the warmer it will get. Therefore, you and your research team, are assigned the task of writing a program that, given sets of possible quantum operations and their costs, can calculate the lowest possible total cost for transforming a set of data to the wanted result.

On the disk, binary words of length $1 \leq L \leq 20$ are treated. The quantum operations are defined by strings of the same length as the binary words, and are built from the four letters N (does nothing), F (inverts one bit), S (sets a bit to 1), and C (resets a bit to 0). Each letter in the string corresponds to an operation on the bit in the binary word at the same position. The binary words are transformed one by one and the total energy cost for the transformation is calculated as the sum of the costs for the performed quantum operations.

Input specifications

The input starts with a single positive integer $N \leq 20$ on a row, deciding the number of test cases that will follow. Then, for each of the test cases:

- One line containing three integers: L, n_{op} and n_w separated by one space.
- L indicates the length of the binary words and the quantum operations.
- $n_{op}(\leq 32)$ is the number of quantum operations that are available for use when transforming the binary words.
- $n_w(\leq 20)$ is the number of binary words that are to be transformed in the current test case.

After this, n_{op} rows follows, each of them containing the definition of a quantum operation followed by the energy cost $0 \leq c_i \leq 1000$ of carrying out the quantum operation. The definition and the cost are separated by a single space.

Finally, there are n_w rows, each containing two binary words separated by a single space. The first of these words should, when possible, be transformed to the second using the quantum operations. The binary words are expressed as sequences of 1's and 0's. After these rows, the next test case follows, if there is any.

Output specifications

Each test case should produce a row containing a list of the energy costs of transforming each of the binary words. The costs should be separated by a single space and presented in the same order as the corresponding input. When there is no successful way of transforming a binary word, "`NP`", meaning *not possible* should be printed instead.

Sample input

```
2
4 3 3
NFFN 1
NFNF 2
NNFN 4
0010 0100
0001 0010
0100 1000
4 4 5
CFSF 4
NNSS 3
FFFF 5
FNFN 6
1111 0000
1001 0110
0101 1000
1000 0011
0000 1001
```

Output for sample input

```
1 3 NP
5 4 8 9 9
```

Problem F

Goldbach's conjecture

source: goldbach.*

For any even number n greater than or equal to 4, there exists at least one pair of prime numbers p_1 and p_2 such that

$$n = p_1 + p_2$$

This conjecture has not been proved nor refused yet. No one is sure whether this conjecture actually holds. However, one can find such a pair of prime numbers, if any, for a given even number. The problem here is to write a program that reports the number of all the pairs of prime numbers satisfying the condition in the conjecture for a given even number.

A sequence of even numbers is given as input. There can be many such numbers. Corresponding to each number, the program should output the number of pairs mentioned above. Notice that we are interested in the number of essentially different pairs and therefore you should not count (p_1, p_2) and (p_2, p_1) separately as two different pairs.

Input specifications

An integer is given in each input line. You may assume that each integer is even, and is greater than or equal to 4 and less than 2^{15}. The end of the input is indicated by a number 0.

Output specifications

Each output line should contain an integer number. No other characters should appear in the output.

Sample input

```
6
10
12
0
```

Output for sample input

```
1
2
1
```

Problem G

Downpayment

source: `downpay.*`

Large houses cost a lot of money. Typically, a loan is required in order to finance the investment. The credit institutions provide a plentitude of different mortgage alternatives, e.g., you may choose to tie the current interest rate for a period. During this period, you may not change institution or terms. After this period, you may change terms, at a penalty cost, or start a new period. The tricky thing is that the interest changes over time.

In this problem, we make the absurd assumption that a crystal ball is at your disposal. That is, you know, in advance, what the different interest rates for all institutions are going to be. Given this information, your task is to come up with a plan for the payments which minimizes the absolute amount of money paid over the entire period.

For each month, the following items are *added* to the debt:

1. Any possible penalties for changing the terms. For the first month, you may choose any alternative without penalty.

2. The interest for the dept, including the penalty.

After these items has been added, the amount is rounded to two decimals towards zero. Then, a fixed amount is paid, i.e., the debt is *reduced* by this amount. However, if the debt is less than the fixed amount, only the remaining debt is paid and the loan is fully paid. You do not have to pay any penalty if the binding time for your current terms is not at end when the loan is paid.

Input specifications

On the first line of input there is one integer, $N \leq 50$, giving the number of test cases in the input. Each test case starts with a line containing one integer m and two floating point numbers x and y with at most two decimals, where $1 \leq m \leq 20$ stands for the number of different loan alternatives, $1 \leq x \leq 1000000$ stands for the investment amount, and $1 \leq y \leq 10000$ is the fixed amount paid every month. Following this line are m lines, each with an integer l, which stands for the binding time in months of this alternative, $1 \leq l \leq 60$. Following these lines are yet another m lines, each with m floating point numbers with at most two decimals, which gives $C(a,b) \geq 0$, the cost of switching from loan alternative a to loan alternative b. Now, $C(x,y) = C(y,x)$ and $C(x,x) = 0$. After these lines, a line with a single integer t follows. t stands for the number of months that interest information is known. You may assume that t is sufficiently large for the optimal solution. Ending the test case are t lines, each with m floating point numbers, giving the actual monthly interest rates, in percent, for all of the alternatives for each month. You may assume that the loan can be fully paid after 100 years. You may also assume that all lines of input are maximum 255 characters.

Output specifications

Start each test case with a line stating the test case, as this: "`Test case` u". For every month required to pay the loan, output one line stating which alternative would be the best one, as this: "`Month` v`: Alternative` w" (there should be only one space between ":" and "A"). u, v, w all starts at 1. Ending the test case, output a line stating the total amount paid, with two decimal digits: "`Total:` s" (there should be only one space between ":" and s).

Sample input

```
2
1 200 100
1
0
5
3
3
3
3
3
2 300 100
1
2
0 4
4 0
4
7 15
20 5
3 10
4 10
```

Output for sample input

```
Test case 1
Month 1: Alternative 1
Month 2: Alternative 1
Month 3: Alternative 1
Total: 209.45
Test case 2
Month 1: Alternative 1
Month 2: Alternative 2
Month 3: Alternative 2
Month 4: Alternative 2
Total: 354.85
```

Problem H

Simplified λ-evaluations

source: lambda.*

Lambda calculus is the main theoretical core of functional languages. It is based on evaluating λ-expressions.

A λ-expression is a string of characters consisting either of *1)* a single constant (for example t), or *2)* a function definition written as $L\langle var\rangle.\langle body\rangle$, where L denotes λ, $\langle var\rangle$ is always some variable (for example x) that can occur in the $\langle body\rangle$, which is again some λ-expression, or *3)* an application of some $\langle function\rangle$ (for example f) to an $\langle argument\rangle$ (for example x), in our case written as $(f)x$, both $\langle function\rangle$ and $\langle argument\rangle$ are λ-expressions.

The occurences of variable $\langle var1\rangle$ in an expression E are called bound, if they are inside of the $\langle body\rangle$ of some sub-expression $L\langle var1\rangle.\langle body\rangle$ of E. Variable occurences that are not bound are called unbound.

Evaluation of λ-expression results in another λ-expression and is performed using the following rules:

1. Constant is evaluated to itself,

2. Function definition evaluates to itself,

3. Function application $(\langle function\rangle)\langle argument\rangle$ is evaluated as follows: first, the $\langle function\rangle$ is evaluated to $L\langle var\rangle.\langle body\rangle$ or something else. In the latter case, the $\langle argument\rangle$ is evaluated, and the whole function application evaluates to $(\langle evaluated\ function\rangle)\langle evaluated\ argument\rangle$. If the $\langle function\rangle$ evaluates to $L\langle var\rangle.\langle body\rangle$, the $\langle argument\rangle$ is not evaluated, but all unbound occurences of variable $\langle var\rangle$ inside of the expression $\langle body\rangle$ are directly replaced by (substituted with) the argument $\langle argument\rangle$. The result of the whole evaluation of function application is then the evaluated $\langle body\rangle$, after the substitution was performed.

We limit the expressions to contain a single-character lowercase ($a \ldots z$) variables and constants only. Sometimes the evaluation never stops. Your program should perform at most 1000 function applications when evaluating any λ-expression. During all evaluation steps, the evaluated λ-expression will not exceed 10000 characters.
Here are some examples of evaluation of simple λ-expressions (shown in several steps, the text behind ; is a comment):

```
1.
  (Lx.x)y                     ; x <- y
  y

2.
  ((Lx.Ly.(x)y)Lz.z)Lq.q      ; x <- Lz.z
  (Ly.(Lz.z)y)Lq.q            ; y <- Lq.q
  (Lz.z)Lq.q                  ; z <- Lq.q
  Lq.q
```

Note that the scope of the variable next to λ covers only the body of the lambda expression, and the same variable can occur at other places of the expression, for example:

```
3.
  ((Lx.x)(Ly.y)Lx.x)x
  ((Ly.y)Lx.x)x
  (Lx.x)x
  x
```

The body of the λ-expression can contain unbound variables - constants:

```
4.
  (((Lx.Ly.q)Lz.t)r)u     ; x <- Lz.t
  ((Ly.q)r)u              ; y <- r
  (q)u
```

Finally note that our evaluation is simplified as compared to the real λ-calculus: when the argument is substituted for all unbound occurences of some variable during function application, some of the variable occurences in the argument that were previously unbound can become bound... This is normally solved by renaming all the variables in the argument before the function application, and keeping track of correct substitutions. In this problem, you don't need to take care of this.

```
5.
  ((Ly.Lx.y)x)w            ; y <- x
  (Lx.x)w
  w
```

The Department of Programming Languages decided to implement a new functional language. However, they first need a core engine that will evaluate λ-expressions. Write a program that will read a set of expressions from the input file and output their evaluations.

Input specifications

The input contains set of λ-expressions, one per line. Only the last line contains the expression consisting of a single constant z. You can assume that the input is correct.

Output specifications

The output should contain the result of evaluation of the expressions on the input in the same order as they appear in the input including the last line. If the expression leads to more than 1000 function applications, the line should contain single word "`unterminated`".

Sample input

```
Lq.q
((Lx.Ly.(x)y)Lz.z)Lq.q
(Lx.x)x
((((Lm.Ln.Lf.Lx.((m)f)((n)f)x)Lo.Lt.(o)t)Lu.Lv.(u)(u)v)a)b
(Lx.(x)x)Lx.(x)x
(q)(Lx.Lx.x)z
z
```

Output for sample input

```
Lq.q
Lq.q
x
(a)(a)(a)b
unterminated
(q)Lx.x
z
```