A. Pirate versus Ninja

Author: Nathan Scott, September 30, 2004

There is a single question which has plagued humanity for ages: "Who would win in a fight between a pirate and a ninja?" With today's astounding computer technology, we can finally put this question to rest.

Of course, you cannot perfectly model the subtle nuances of such a combat, so we've provided you with a simple model to approximate some of the more complicated variables. Each combatant has two parameters at the outset of a fight: *Health* and *Power*. The fight should be simulated as a sequences of *exchanges* where each combatant's power value is subtracted from the opponent's health value. The blows within the exchange take place simultaneously. The exchanges continue until one combatant's health value is less than or equal to zero.

An example combat follows between a Pirate with 25 health and 5 power fighting against a Ninja with 18 health and 6 power:

Exchange #	Pirate Health	Ninja Health
1	19	13
2	13	8
3	7	3
4	1	-2

The pirate wins, since the Ninja has less than or equal to zero health.

There can be only one victor, so no test case will ever involve the pirate and ninja both killing each other simultaneously. For simplicity, we are ignoring external variables such as Cyborgs from the future or Vikings.

Write a program to determine the result of a combat between a given pirate and ninja.

Input

The input begins with a number N, specifying how many test cases there will be. Each of the N cases will consist of 2 lines. The two lines specify two combatants in the following format:

$Name\ H\ P$

Where $Name$ is the name of the combatant (a single word, no whitespace), H (> 0) is the initial Health value for the combatant, and P (> 0) the Power value for the combatant.

Output

For each test case, print the name of the winner and the winner's remaining health at

the end of the fight, separated by a single space and all on a single line.

Sample Input

2
Blackbeard 42 8
Ryu 38 12
Hook 26 18
Haruko 36 12

Sample Output

Ryu 6
Hook 2

B. Textbooks

Author: Nathan Scott, September 30, 2004

Write a program to simulate the UNB bookstore's inventory and predict how much money they will earn.

You will be given a list of textbooks for sale along with their prices. You may have the same textbook listed under more than one price (for example, a "new" price and a "used" price.) You will also know the quantity available of each textbook.

A series of customers will then arrive to purchase their books. Each customer has a list of textbooks they will be buying. They will always buy the cheapest edition of the textbook that is available.

Your program should process each customer in the order they arrive, and output the amount of money they will have to spend to buy their textbooks.

Input

The input will be divided into two sections.

The first section represents the starting inventory for the bookstore. It will begin with a single positive integer, n. This is the number of textbooks on sale. The following n lines contain three items: the title of a textbook (enclosed in double quotes, i.e., "Textbook Title"), the price of the textbook (a positive integer, in dollars), and the quantity in stock (a positive integer). Textbook titles can be repeated: Textbooks can be available at more than one price. However, title-price pairs will be unique.

The next section also begins with a single positive integer, m. This is the number of incoming customers. Customers' orders should be processed in the order listed in the input. Each customer begins with yet another single positive integer, k. This is the number of books this customer requires. The next k lines contain textbook titles (again, enclosed in double quotes) that the customer requires.

Output

For each customer order, output the minimum cost for all the textbooks, or "Cannot fill order" if any of the requested textbooks are not available. After a customer purchases books, those books are removed from inventory. NOTE: If a customer cannot fill his entire order, he storms out in a fit of rage and does not purchase ANY of the other books even if they are available, and so those books should not be removed from inventory.

Sample Input

```
8
“Software Engineering” 100 5
“Software Engineering” 70 1
“Nonlinear Dynamics” 95 2
“Calculus” 130 12
“Calculus” 100 3
“Nonlinear Dynamics” 60 2
“Calculus” 80 1
“How To Cheat And Get An A” 250 1
5
3
“Software Engineering”
“Calculus”
“How To Cheat And Get An A”
2
“Software Engineering”
“Nonlinear Dynamics”
4
“Nonlinear Dynamics”
“How To Cheat And Get An A”
“Calculus”
“Software Engineering”
1
“Nonlinear Dynamics”
3
“Software Engineering”
“Calculus”
“Nonlinear Dynamics”
```

Sample Output

```
400
160
Cannot fill order
60
295
```

C. The Most Interesting Problem Ever

Author: Sean Falconer, September 30, 2004

Hi, welcome to the most interesting problem ever! Ok, ok, I lied, it's not really that interesting, but I was afraid no one would want to read this problem if I told you what the problem was really about. So, since you are here, you might as well finish reading the description and code this puppy.

You must implement a simplified version of a spreadsheet program. You will read in the description of a spreadsheet, and the descriptions of several commands. For each command, you must perform the appropriate operation if legal, and after processing all operations, print the new spreadsheet description.

The following is the command list that your program must be able to perform:

1. del cell r:c - empties the cell contents at row r and column c.
2. del row r - deletes row r, shifting all rows after r up one row.
3. del col c - deletes column c, shifting all columns after c to the left by one.
4. modify cell r:c v - change the contents of the cell at row r and column c for the integer v
5. add cell $r1$:$c1$ $r2$: $c2$ - add the cell contents at row $r2$ and column $c2$ to the cell at row $r1$ and column $c1$.
6. add row $r1$ $r2$ - for each column in row $r2$, add the value to the same column in row $r1$.
7. add col $c1$ $c2$ - for each row in column $c2$, add the value to the same row in column $c1$.
8. sub cell $r1$: $c1$ $r2$: $c2$ - subtract the cell contents at row $r2$ and column $c2$ to the cell at row $r1$ and column $c1$.
9. sub row $r1$ $r2$ - for each column in row $r2$, subtract the value to the same column in row $r1$.
10. sub col $c1$ $c2$ - for each row in column $c2$, subtract the value to the same row in column $c1$.

During an add or subtract, if either cell in the add or subtract is empty, treat the empty cell as equal to zero. If both cells are empty, then the resulting cell is empty. If a command is invalid, i.e. the row or column indicated is too large, ignore that command.

Input

There will be multiple test cases. Each test case begins with two integers, $0 < R \leq 100$ and $0 < C \leq 100$, where R is the number of rows and C is the number of columns for the spreadsheet. Next there will be R lines, each with C values. Each value of the spreadsheet will be an integer, or 'e' (without quotes), the 'e' represents an empty cell. After the description of the spreadsheet, there will be an integer N indicating how many commands you must process. After this line is N lines, each with a command as specified above. The end of input is indicated by an R and C value of zero, this test case is not to be processed.

Output

After processing each set of commands, you must output the final version of the spreadsheet. There should be $R2$ lines, each line with $C2$ values, where $R2$ and $C2$ are the number of rows and columns for the final version of the spreadsheet. Empty cells must be printed as 'e'. Each value in a row must be separated by a tab character, i.e. '\t'. If $R2 = 0$ or $C2 = 0$, print "Empty" instead of the final version of the spreadsheet. Each spreadsheet for each test case must be separated by a blank line.

Sample Input

```
3 3
1 2 12
3 3 3
e e 10
3
add row 1 5
sub cell 1:2 1:3
sub row 1 2
3 3
1 2 12
3 3 3
e e 10
4
add row 1 2
del row 1
sub col 1 2
del col 3
0 0
```

Sample Output

```
-2      -13     9
3       3       3
e       e       10

0       3
e       e
```

D. Caesar Cipher Madness

Author: Sean Falconer, September 30, 2004

One of the simplest encryption techniques is the substitution cipher. This is where one letter is substituted for another letter. The first known version of this is credited to Julius Caesar and is aptly named, the Caesar Cipher. With the Caesar Cipher, characters are shifted or offset by a given key, this is known as a shift transformation. For example, every appearance of a letter may be shifted to the right by 3. As "sophisticated" as this is, it can be broken, simply by trying every possible shift transformation.

Being the "intelligent" computer scientists that we are, we realized that this method is far too simple, so we added a modification to the traditional Caesar Cipher. Our algorithm works with multiple shifts, so that once we apply the Caesar Cipher for a given key, we switch the key value by adding an offset value, and apply the algorithm again. We do this over and over, until we feel the text is sufficiently disguised.

In this problem, you must read in the plaintext value along with its corresponding ciphertext, and then try to discover the original key, the offset, and the number of times the Caesar Cipher was applied. The original key and offset will never be larger than 25. Also, the Caesar Cipher is run at least once and a maximum of 100 times.

Since we are only concerned with seeing if our encryption algorithm can be cracked, we made things easier for you. All plaintext strings will not contain uppercase letters. Also, we want you to ignore all non-lowercase letters. That means, when we encrypt a plaintext value of "hi programmers, i hope things are going well.", we actually used "hiprogrammersihopethingsaregoingwell" as our plaintext value.

Input

The input consists of multiple test cases. Each test case consists of two lines, the plaintext value followed by the ciphertext. Each line consists of a maximum of 80 characters. The plaintext value will consist of only lowercase letters, whitespace, and punctuation.

Output

For each test case, one line of output is to be displayed. The line must consist of three integer values, k, o, and ℓ, where k is the original key used by our encryption algorithm, o is the offset used to modify the key for each application of the cipher, and ℓ is the number of times the Caesar Cipher was applied. Each value must be separated by a space.

NOTE: Multiple solutions may exist, if so, you must minimize o and ℓ and maximize k.

Sample Input

hi programmers, i hope things are going well.
delnkcnwiianoedklapdejcownackejcsahh
unb programming club!!!!
vocqsphsbnnjohdmvc

Sample Output

22 0 1
1 0 1

E. Magic Squares

Author: Nathan Scott, September 30, 2004

A magic square is an arrangement of the numbers from 1 to n^2 in an $n \times n$ matrix, with each number occurring exactly once, and such that the sum of the entries of any row, any column, or any main diagonal is the same. (Mathforum.org)

For example, take the 3×3 magic square:

8	1	6
3	5	7
4	9	2

Each row, column and diagonal sum to 15. Now a magic square must contain the numbers 1 to n^2, but we can have squares of the same size with the same "magic properties" of the row, column and diagonal sums, only with larger (although still consecutive) numbers within the square.

For example:

19	12	17
14	16	18
15	20	13

with the sums equal to 48.

Given an incomplete 3×3 square with magic properties, fill in the missing values.

Input

Each case will contain 3 lines, each with 3 entries, making up a 3×3 square. Each entry will be either a positive integer, or X. A blank line will separate each case. No test cases will be unsolvable.

Output

For each square in the input, output the same square with the X entries replaced with positive integers such that the result is a square with magical properties (each row, column and diagonal sum to the same value) and uses consecutive integers in the solution. A blank line should separate each case.

Sample Input

11 4 9
X X 10
X 12 5

24 X X
X X 23
20 25 X

Sample Output

11 4 9
6 8 10
7 12 5

24 17 22
19 21 23
20 25 18

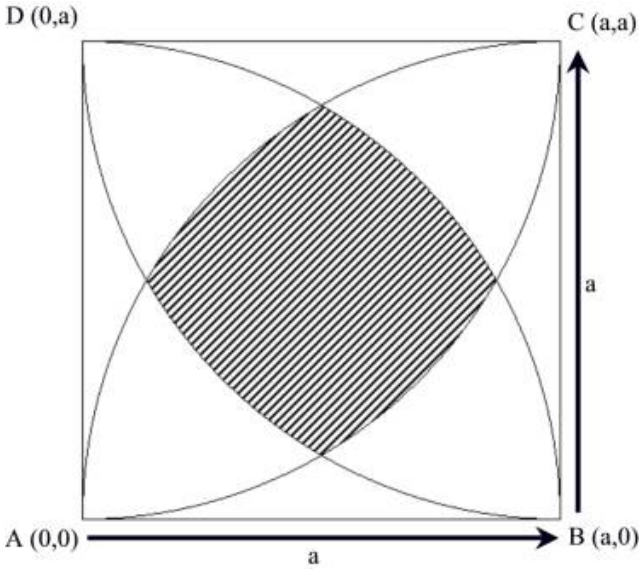
F. The Wireless Network Problem

Author: Sean Falconer, September 30, 2004

My hometown of St. Stephen heard that Fredericton has a wireless network and out of jealousy, they decided to implement one as well. However, St. Stephen's pockets do not run quite as deeply as Fredericton's, therefore, they had to buy much of their wireless equipment secondhand. Apparently, some of it is slightly faulty and only works some of the time. The St. Stephen town council decided this isn't an issue because they have 4 wireless routers in the downtown area, so if one goes down, most of downtown is still covered by the other 3 routers.

The problem is, sometimes 3 of the 4 routers will go down all at the same time, the town council still refuses to acknowledge this as an issue, claiming that even with one station, the critical areas of downtown are covered. I don't agree with the town council, however they won't listen to me unless I have evidence to back up my claim that downtown needs more consistent wireless coverage. I want you to conduct a private study to determine how many downtown shops will always be covered by at least one station and if this number is less than half the total number of downtown shops, we'll force the town council to fix the problem.

The figure below shows the downtown coverage by all four routers, where the four stations are located at A, B, C, and D. For each station, the wireless coverage for that given station is represented by a half circle with a certain radius. The crossed area in the middle is the "critical" downtown area. Given a, the radiuses of each half circle, and N points representing stores in the downtown area, you must make a recommendation to me based on how many stores are within the critical area. A store is covered by a given router only if it is within the bounds of the area it covers, not on the bounds.



Input

The input consists of multiple test cases. Each test case begins with an integer $10 \leq a \leq 100$ representing the length of each side of the downtown area as shown above. This is followed by the 4 integer values representing the radiuses for wireless routers: A, B, C, and D. Next is an integer $N \geq 0$, representing the total number of stores. This is followed by N lines, each consisting of two integers representing a given stores's x and y coordinates.

Output

Output for each test case consists of a single line stating either "St. Stephen needs new equipment.", if less than half the stores are covered, or "The wireless coverage still stinks, but good enough for now.", if more than or equal to half the stores are covered.

Sample Input

```
10
10 10 10 10
1
5 5
20
20 18 19 15
4
5 6
1 1
6 5
19 19
```

Sample Output

```
The wireless coverage still stinks, but good enough for now.
St. Stephen needs new equipment.
```

G. Double Checking

Author: Nathan Scott, September 30, 2004

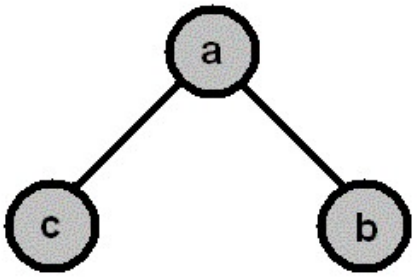
A simple and common data structure that facilitates searching is a tree. A tree usually consists of a single node, the "root" node, that has a number of children, each of which is also a tree (i.e. each child may have children of its own.) A binary tree is one in which each node can have at most 2 children. Such a tree (which stores letters from a-z) can be specified easily with the following format:

Either

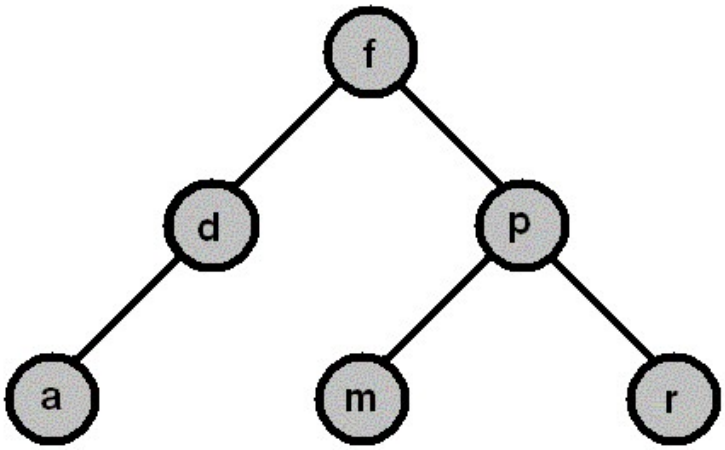
1. A single letter, or
2. (Left Subtree) A single letter (Right Subtree) *Note subtrees may be empty

The height of a node is the longest distance from that node to the "bottom" of the tree (also called the "leaves"). Some examples:

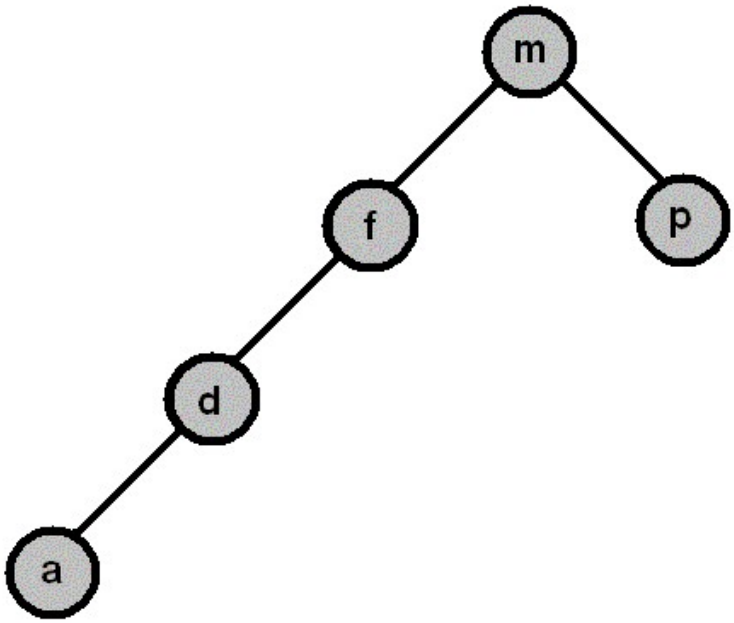
1) a (height of a = 1)



2) (c)a(b) (height of a = 2, c and b = 1)



3) ((a)d())f((m)p(r)) (height of f = 3, d = 2, p = 2, rest are 1)



4) (((a)d())f())m(p) (height of m = 4, f = 3, d = 2, p = 2, a = 1)

Easy enough so far? A binary search tree isn't much more complicated: It's a binary tree, only for any given node n in the tree, all nodes in the LEFT subtree of a node have lower values than n, and all nodes in the RIGHT subtree have greater values than n (we're going to pretend we have no equal values for now.) If we are looking at alphabetical order, trees 1), 3), and 4) are binary search trees, but 2) is not because the node 'c' is greater than 'a'.

One step further is the AVL tree. An AVL Tree is a type of "balanced" binary search tree, or more specifically a binary search tree with the additional property that for any node

n, the heights of n's subtrees differ at most by one. Again, from the above examples: 1) and 3) are AVL Trees. 2) isn't one because it isn't even a binary search tree. 4) isn't one because f's left subtree has height 2, and its right subtree has height 0 (alternatively, m's left subtree has height 3, and its right subtree has height 1).

At some point, for some class or another, many CS students end up having to implement an AVL tree. Coincidentally, we here at the UNB Programming Club are acquainted with one of the markers for one of these classes, and danged if he isn't sick and tired of double checking that everyone's AVL trees are proper. More importantly, we're all getting sick and tired of hearing him complain about it. So we want you to do him a favour and write a program that verifies if a given tree is either an AVL Tree, an ordinary binary search tree, or neither.

Input

The first line of the input will have a number, N, indicating the number of test cases to follow.

A properly formatted binary tree will be specified on each of the following N lines, according to the format described above. There will be no whitespace in the tree description. Nodes will have values consisting of lower case letters from 'a' to 'z'. You can assume that no two nodes in the tree will have the same values.

Output

For the n'th test case, print out "Tree #n " followed by: "is an AVL Tree." if the tree is a proper AVL Tree; "is a binary search tree." if the tree is a binary search tree, but NOT an AVL Tree; "is just an ordinary tree." if the neither of the above cases apply. Each case's output should be followed by an end of line.

Sample Input

```
4
a
(c)a(b)
((a)d())f((m)p(r))
(((a)d())f())m(p)
```

Sample Output

```
Tree #1 is an AVL Tree.
Tree #2 is just an ordinary tree.
Tree #3 is an AVL Tree.
Tree #4 is a binary search tree.
```

H. Candy Machine

Author: Sean Falconer, September 30, 2004

I occasionally do contract work for a local company, usually software design, but sometimes they have me help out with programming when they are close to a deadline. There's a candy machine at work, and myself and the other guys sometimes indulge ourselves with some candy from the machine. The other day, we started discussing the problem of trying to find out what the minimum number of coins we'd need in order to buy C number of candies assuming the machine always gives us back change using the minimum number of coins. Due to a deadline, we did not have time to come up with an adequate solution to this problem, so we'd like you to figure it out for us.

To simplify the problem slightly, we'll assume all candies cost 8 cents and the only coins you need to be concerned with are pennies, nickels, and dimes. You may assume that the machine will never run out of coins and that you will always have enough coins to buy all the candies that you want.

Input

The first line of input contains an integer, T, representing the number of test cases. Following this is T lines consisting of 4 integers $C \leq 150$, $0 \leq P \leq 500$, $0 \leq N \leq 100$, and $0 \leq D \leq 50$ where C is the number of candies you want to buy, P is the number of pennies you start with, N is the number of nickels you start with, and D is the number of dimes you start with.

Output

For each test case, output the minimum number of coins you need to insert into the machine to buy the C candies.

Sample Input

```
2
2 2 1 1
20 200 3 0
```

Sample Output

```
5
148
```