acm High School Programming Contest @ sponsored by:

2015 University of Virginia High School Programming Contest

Welcome to the 2015 University of Virginia High School Programming Contest. Before you start the contest, please be aware of the following notes:

Rules

1. There are ten (10) problems in this packet, using letters A-J. These problems are *loosely* sorted by difficulty. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	**Problem Name**	**Baloon Color**
A	Storing Super Suits	dark purple
B	Fury's Graffiti	dark green
C	Encrypted Messages	orange
D	Mapping the Universe	dark blue
E	You Wouldn't Like Me When I'm Angry	red
F	Arrow Addition	light green
G	Understanding the Internet	yellow
H	Infiltrating the Enemy	gold
I	How Much Longer?	pink
J	Important Puzzles	silver

2. Solutions for problems submitted for judging are called runs. Each run will be judged.

   The judges will respond to your submission with one of the following responses. In the event that more than one response is applicable, the judges may respond with any of the applicable responses.

Response	**Explanation**
Yes	Your submission has been judged correct.
No - Wrong Answer	Your submission generated output that is not correct or is incomplete.
No - Output Format Error	Your submission's output is not in the correct format or is misspelled.
No - Excessive Output	Your submission generated output in addition to or instead of what is required.
No - Compilation Error	Your submission failed to compile.
No - Run-Time Error	Your submission experienced a run-time error.
No - Time Limit Exceeded	Your submission did not terminate within one minute.

3. A team's score is based on the number of problems they solve and penalty minutes, which reflect the amount of time and number of incorrect submissions made before the problem is solved. For each problem solved correctly, penalty minutes are issued equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty minutes are added for problems that are never solved. Teams are ranked first by the number of problems solved and then by the fewest penalty minutes.

4. This problem set contains sample input and output for each problem. However, the judges will test your submission against several other more complex datasets, which will not be revealed until after the contest. One challenge is designing other input sets for yourself so that you may fully test your program before submitting your run. Should you receive a "wrong answer" judgment, you should consider what other datasets you could design to further evaluate your program.

5. In the event that you think a problem statement is ambiguous or incorrect, you may request a clarification. Read the problem carefully before requesting a clarification. If the judges believe that the problem statement is sufficiently clear, you will receive the response, "The problem statement is sufficient; no clarification is necessary." If you receive this response, you should read the problem description more carefully. If you still think there is an ambiguity, you will have to be more specific or descriptive of the ambiguity you have found. If the problem statement is ambiguous in specifying the correct output for a particular input, please include that input data in the clarification request.

   You may not submit clarification requests asking for the correct output for inputs that you provide. Sample inputs may be useful in explaining the nature of a perceived ambiguity, e.g., "There is no statement about the desired order of outputs. Given the input: . . . , would not both this: . . . and this: . . . be valid outputs?".

   If a clarification that is issued during the contest applies to all the teams, it will be broadcast to everybody.

6. Runs for each particular problem will be judged in the order they are received. However, it is possible that runs for different problems may be judged out of order. For example, you may submit a run for B followed by a run for C, but receive the response for C first.

   Do not request clarifications on when a response will be returned. If you have not received a response for a run within 30 minutes of submitting it, **you may have a runner ask the site judge to determine the cause of the delay. Under no circumstances should you ever submit a clarification request about a submission for which you have not received a judgment.**

   If, due to unforeseen circumstances, judging for one or more problems begins to lag more than 30 minutes behind submissions, a clarification announcement will be issued to all teams. This announcement will include a change to the 30 minute time period that teams are expected to wait before consulting the site judge.

7. The submission of abusive programs or clarification requests to the judges will be considered grounds for immediate disqualification. This includes submitting dozens of runs within a short time period (say, within a minute or two).

Your Programs

8. All solutions must read from standard input and write to standard output. In C this is `scanf()` / `printf()`, in C++ this is `cin` / `cout`, and in Java this is `System.in` / `System.out`. The judges will ignore all output sent to standard error (`cerr` in C++ or `System.err` in Java). You may wish to use standard error to output debugging information. From your workstation you may test your program with an input file by redirecting input from a file:

```
program < file.in
```

9. All lines of program input and output should end with a newline character (`\n`, `endl`, or `println()`).

10. All input sets used by the judges will follow the input format specification found in the problem description. You do not need to test for input that violates the input format specified in the problem.

11. Unless otherwise specified, all lines of program output should be left justified, with no leading blank spaces prior to the first non-blank character on that line.

12. Unless otherwise specified, all numbers in your output should begin with a '-' if negative, followed immediately by 1 or more decimal digits. If it is a real number, then the decimal point should be followed by as many decimal digits as can be printed. This means that for floating point values, use standard printing techniques (cout and System.out.println). Unless otherwise noted, the judging will check your programs with 10^{-3} accuracy, so only consider the sample output up until that point.

    In simpler terms, neither scientific notation nor commas will be used for numbers, and you should ensure you do not round or use a set precision unless otherwise specified in the problem statement.

13. If a problem specifies that an input is a floating point number, the input will be presented according to the rules stipulated above for output of real numbers, except that decimal points and the following digits may be omitted for numbers with no fractional component. Scientific notation will not be used in input sets unless a problem statement explicitly specifies it.

Good luck, and HAVE FUN!!!

A. Storing Super Suits

Description

Iron Man stores his high-tech gear in temperature-controlled facilities all across the world so that he can fight crime anywhere. However, he has a problem configuring each facility, as some track temperature in Celsius and some in Fahrenheit. He needs to write a program that converts from Fahrenheit to Celsius, and vice versa.

The formula to convert a temperature in Fahrenheit (f) to a temperature in Celsius (c) is:

$$c = (5/9)(f - 32)$$

The formula to convert a temperature in Celsius (c) to a temperature in Fahrenheit (f) is:

$$f = (9/5)c + 32$$

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case consists of two lines. The first line will be either "f" or "c" – designating whether the given temperature is in Fahrenheit or Celsius, respectively. The second line will be a floating point number representing the value to convert. This number will have at most two digits after the decimal place. You need to convert Fahrenheit values into Celsius, and Celsius values into Fahrenheit.

Each given temperature will be between 0 and 1000.

Output Format

For each case, output the number converted to the appropriate scale on its own line. You should not specify units, just output the value itself.

Each converted number should have *exactly* two digits of accuracy after the decimal place, rounding as necessary (any value with a 5 in the thousandths place rounds up).

Sample Input

```
4
f
32
c
0
f
-40
c
-12.34
```

Sample Output

```
0.00
32.00
-40.00
9.79
```

B. Fury's Graffiti

Description

After the events of the first Avengers movie, Nick Fury got in trouble for taking matters into his own hands. As punishment, he has to repaint and label all the helicarriers in his fleet. However, his labelling equipment was damaged in the fight with the Chitauri, and now it can only print round letters and numbers. He needs help figuring out what labels he will be able to print on his helicarriers.

Given a word or number, determine if the word or number is made up completely of rounded letters and digits. A rounded letter is one that has a curve. For example, 'S' is rounded, as is '5', but 'I' and 'T' are not. All letters that we are considering are upper-case letters.

For the purposes of this problem, the rounded letters are: B, C, D, G, J, O, P, Q, R, S, and U. The rounded digits are: 2, 3, 5, 6, 8, 9, and 0. We realize that other letters may appear rounded when rendered with specific fonts; however, the letters and digits listed here are the only rounded ones as far as this problem is concerned.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each of the following n lines will consist of one string. Each string will only consist of alphanumeric characters (A-Z, 0-9). There will be no spaces on each line. A test case could contain both letters and numbers, but only capital letters will be used.

Each string will have at most 254 characters.

Output Format

If the word or number is made up completely of rounded letters and numbers, print `ROUNDED` followed by a new line. If there is a non-round letter of number, print `NOT ROUNDED` followed by a new line. Please pay attention to case in the output!

Sample Input

```
5
GOOD
LUCK
AT
HSPC
2015
```

Sample Output

```
ROUNDED
NOT ROUNDED
NOT ROUNDED
NOT ROUNDED
NOT ROUNDED
```

C. Encrypted Messages

Description

Like all comic book bad guys, members of HYDRA are not all that smart. Using simple encryption techniques, S.H.I.E.L.D. can obfuscate their messages by reversing parts of them within double quotations so that HYDRA is not able to comprehend them at all. S.H.I.E.L.D. needs your help to encode these messages, using the algorithm described here.

We are interested in reversing a string; however, there are some constraints to the reversal process. For one, substrings which are within double quotation marks (i.e. surrounded by the `"` character) must not be reversed – in other words, these quoted substrings must remain in place. Additionally, substrings within double quotation marks which are already inside quoted substrings must be reversed the same way we would reverse a string originally.

Let us consider some sample cases to illustrate the details of the process:

- `123456` → `654321` – since there are no quotes, this string is reversed the standard way
- `12"34"56` → `65"34"21` – the "34" substring remains in place due to the quotes, but the remaining string (i.e. what remains after taking out that substring) is reversed
- `12"34"56"78"9` → `92"34"65"78"1` – the `"34"56"78"` substring remains mostly in place, with the exception of the "56" substring of that substring, which is reversed as if it was an instance of the original problem. The remaining part of the string (composed of the 1st, 2nd, and 9th characters) is also reversed.

Note that for the last example, it was assumed that the outermost and innermost pairs of double quotation marks were the ones that matched (i.e. the quote opened to the left of 3 was closed by the quote to the right of 8, and the quote opened to the left of 5 was closed to by the quote to the right of 6). You may assume in general that this pattern holds true – in other words, the first and last double quotation marks in an input string will always match each other, and within a properly quoted substring the first and last double quotation marks will also always match each other.

Also note that a substring within quotes may be empty, and that the entire input string may be within quotes.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

The next n lines will each contain a string to be reversed, as described above. Any string will be on a single line of length no greater than 255 characters. Each character of that string is either an alphanumeric character or the double quotation mark.

Each string will have at most 250 total characters, and at most 50 quotation marks. All quotation marks will be balanced (i.e. there will never be an odd number of quotation marks).

Output Format

Each string is printed reversed, according to the description above, on it's own line.

Sample Input

```
3
uvahspc
co"din"g""is"f"un
she"sells"seashells"by"the"seashore"
```

Sample Output

```
cpshavu
nu"din"s""ig"f"oc
ehs"sells"ehtsllehs"by"aes"seashore"
```

D. Mapping the Universe

Description

Thor is trying to plot all the realms on a map because he's having trouble keeping track of them. For maps like this, it is useful to give each region a color so that they are easier to recognize. It is also best not to give adjacent regions the same color, otherwise we couldn't tell them apart. Thor is having trouble designing his map because he doesn't know how to color each realm. He has given you a bunch of different maps he's made, and he needs you to tell him if each one is valid or not.

A valid coloring of a map is then one where no two adjacent regions have the same color. Given a list of colored regions, and a list of borders between regions, determine whether the coloring is valid or invalid.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case will start with a line containing a single integer l. The next l lines will each contain an integer r and string c, where r is an integer representing a region and c is that region's color. The region numbers will start at zero and increase by one for each consecutive region. The region numbers will always appear in properly sorted order (0 to $l-1$). The map colors will be alphanumeric strings, without any punctuation or spaces.

The following line will contain a single integer b. The next b lines will each contain two integers a and b indicating that region a borders region b (e.g. "1 2" indicates that region 1 borders region 2). These b lines may appear in any order.

Region names and color names will only contain alphanumeric characters. However, they are case sensitive, so 'ASGARD' is not the same region as 'Asgard'.

There will be at most 200 regions, thus at most 40000 possible borders. The same border may be listed multiple times for any one problem; additional listings have no extra significance.

Output Format

One line per test case containing `valid` if the coloring is valid, or `invalid` if the coloring is invalid. Please pay attention to case in the output!

Sample Input

```
2
3
0 red
1 orange
2 red
2
0 1
1 2
3
0 red
```

```
1 orange
2 red
3
0 1
1 2
0 2
```

Sample Output

```
valid
invalid
```

E. You Wouldn't Like Me When I'm Angry

Description

Bruce Banner gets mad when he sees numbers that are sad during his calculations, and he turns into the Hulk and smashes everything in his path. We want to make sure he won't get mad, so we need to write a program to determine if numbers are happy or not before we let him see them.

"Happy Numbers" have been studied in the branch of mathematics known as Number Theory (you don't need to know what a happy number is for this problem). This problem explores a variation on these numbers, which we will call "quasi-happy" and "quasi-sad" numbers.

To determine if a number is quasi-happy, first break it into groups of two adjacent digits; if there are an odd number of digits, the most significant digit should be left by itself. For example, 123456 would split into three groups: 12, 34, and 56. Likewise, 99941 would split into three groups: 9, 99, and 41.

Next, square each of those groups and add them together: $12^2 + 34^2 + 56^2 = 4436$ and $9^2 + 99^2 + 41^2 = 11563$, respectively.

You now have a new number. If you can repeat this process any number of times and reach a value greater or equal to 50 and less than 100 (i.e., $50 \leq x < 100$), then the number is "quasi-happy." Similarly, if you can repeat this process any number of times and reach a value less than 50, we call the number "quasi-sad".

Continuing with our example from before: $99941 \rightarrow 11563 \rightarrow 4195 \rightarrow 10706 \rightarrow 86$.

Thus, we can see that 99941 is a quasi-happy number, and it took four iterations of our procedure to determine that fact.

There are some quasi-happy or quasi-sad numbers that require an enormous number of iterations to validate, and it is possible for some numbers to be neither quasi-happy nor quasi-sad. Thus, if it takes more than 1000 iterations of our procedure to determine if a number is quasi-happy or quasi-sad, print "LONG".

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each of the n lines that follow consist of a single integer x. That number is guaranteed to fit in a `int` variable. To be more specific, x will have a value between 0 and 2147483647.

Output Format

For each x in the file, your program is to determine if it is quasi-happy. quasi-sad, or too long to decide.

If the number is quasi-happy, print "QUASI-HAPPY: k", where k is the number of iterations of our procedure it took to find that result.

Similarly, if the number is quasi-sad, print "QUASI-SAD: k", where k is the same as before.

If it took more than 1000 iterations of our procedure to determine if a number is in either category, print "LONG".

Note that it is possible for input to take 0 iterations to determine if a number is quasi-happy or quasi-sad.

Sample Input

```
6
1214829
995
1214831
986
1214837
```

Sample Output

```
QUASI-SAD: 41
QUASI-HAPPY: 26
QUASI-SAD: 46
LONG
QUASI-HAPPY: 10
QUASI-HAPPY: 0
```

F. Arrow Addition

Description

Hawkeye is trying to bring down the Chitauri, but he doesn't have enough arrows to hit all of them. He knows how much damage each arrow will do to an enemy, and he knows how much health each one has. He doesn't want to waste arrows, so he does not want to cause *more* damage than an enemy has health.

For any given Chitauri, he can choose from a large number of arrow combinations to do enough damage to bring it down. Obviously, the number of ways to reach this target damage is dependent on the available arrows. For example, there are four ways to reach exactly 10 points of damage using arrows that cause 1, 5, and 10 points of damage respectively. There are more ways to do *more* than 10 points of damage, but we are not considering cases of causing more than 10 points of damage for this problem. However, if we also allow an arrow that does 2 points of damage, then there are now eleven ways to reach 10 points of damage. The order in which he shoots the arrows does not matter.

Hawkeye wants to write a program that determines the number of possible ways to bring down a Chitauri given its health and the types of arrows available to him.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case comprises three lines. The first line contains two integers, s and t, which represent the number of distinct arrow types and the number of target health amounts, respectively. The second line contains s integers $v_1, v_2, \ldots, v_s$ which provide the values of each arrow type in ascending order. These values are space separated. You are always guaranteed $v_1 = 1$, which ensures that you can reach any health amount. The third line contains t integers $x_1, x_2, \ldots, x_t$ which provide the target health amounts. These values are also space separated.

In the first example below, the input file has 2 test cases: the first has arrows of value 1, 5, 10, 25, and asks for the number of ways to reach 10 health points 50 health points. The second test case uses a base-2 system with arrows of value 1, 2, 4, 8, and 16; it asks for the number of ways to reach 10, 50, and 63 health points.

All given arrow types will have a value between 1 and 50. All health ranges will be between 1 and 150.

Output Format

The output for each test case is a single line, consisting of "Case k:", where `k` is the current case number, followed by a space-separated listing of the number of ways to make each target health points. It does not matter if you have a extra space at the end of each line. Note that the output entries must exactly correspond to target amounts; do not sort or rearrange the output amounts.

Sample Input

```
2
4 2
1 5 10 25
```

```
10 50
5 3
1 2 4 8 16
10 50 63
```

Sample Output

```
Case 1: 4 49
Case 2: 14 740 1460
```

G. Understanding the Internet

Description

Captain America is trying to work through his list of things to do that he missed while he was frozen for decades. However, he's completely stumped by the popular internet game 2048. He thinks that, if he could see some simulated games, he might understand it better.

You will simulate a game in which a 4 by 4 grid containing numbered tiles is manipulated to yield the highest number possible.

The grid will be represented by a 4 by 4 matrix of integers. A value of zero indicates a blank space (i.e. no tile) at that position. A positive integer indicates a tile with that integer's value at that position. No negative integers are used in this representation.

On one turn of the game, you can tilt the grid in one of four directions: up (U), down (D), left (L), or right (R). When you tilt the grid, the grid's tiles will all slide in the given direction until they either collide with the edge of the grid or they hit another tile. If a tile collides with another tile of the *same* value, then the two tiles will merge, yielding a new tile numbered with the sum of the two integers. The new tile will be in the position closer to the edge to which the grid was tilted (if you swipe down, the new tile will be in the position of the tile closest to the bottom of the screen). Only tiles present on the board before a move can be combined during a move. If a tile is created by combination during a move, it cannot be further combined during the same move.

For example, if we start with the grid:

```
8 2 2 2
4 2 0 0
4 2 0 2
2 2 0 0
```

and tilt it down (D), we will get the grid:

```
0 0 0 0
8 0 0 0
8 4 0 0
2 4 2 4
```

Note that a tile created through a merge cannot merge again on the same turn. The second column of the previous example demonstrates this rule.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case will contain five lines.

The first four lines of each test case will each contain four space separated integers. Collectively, they specify the initial grid in the format described above.The integers are guaranteed to be within the bounds of a 32 bit `int`. Likewise, the sums that are generated during each simulation will also be within the bounds of a 32 bit `int`. There will always be at least one tile in the grid.

The fifth line will contain a string (of maximum length 64) of uppercase characters representing moves (‘D’, ‘U’, ‘L’, or ‘R’). Each character specifies the direction of a move, as described above. The moves must be executed in order (e.g. for the string ‘LU’, the initial grid is tilted left, and the resulting grid is tilted up).

Each test will consist of at most 64 moves.

Output Format

The output will have one line per input test case.

Each output line will contain the highest integer value present on a tile in the grid after the initial grid has undergone all the moves specified in the input.

The highest value is guaranteed to be within the bounds of a 32 bit `int`.

Sample Input

```
2
8 2 2 2
4 2 0 0
4 2 0 2
2 2 0 0
DUR
0 3 5 6
6 3 12 0
0 3 0 3
3 0 0 0
LDLUR
```

Sample Output

```
16
24
```

H. Infiltrating the Enemy

Description

S.H.I.E.L.D. is trying to learn more about HYDRA's management structure so they can figure out who is in charge at each level, and insert spies in the best places. HYDRA's management structure forms a tree-based hierarchy, and this is stored as trees in infix and prefix form. Unfortunately, S.H.I.E.L.D. can only understand trees in postfix form. We'll discuss trees along with prefix, infix, and postfix form below.

Trees are some of the most important things in the world. They convert the carbon dioxide you breath out into oxygen. They prevent the erosion of coastlines, and provide a habitat for a variety of different animals. Most importantly, trees allow us to represent a hierarchical structure.

Here is an example of a tree:

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

Here is another example of a tree:

```
    1
   / \
  2   3
   \   \
   5    6
  / \
 4   7
```

However, as computer scientists, it is hard for us to represent trees, since keyboards are notoriously bad at free form drawing. There are three ways to represent trees with text by printing out each of the individual values. But if we are going to print out the values in the tree, we need a system for doing so. In all cases, we will print out the left child *before* the right child. However, that does not tell us when to print out the root node.

Consider this simple tree:

```
  2
 / \
1   3
```

- If we print it out in *prefix* notation, that means we print out the node value *before* the left child (and we always print out the left child before the right child). Thus, that tree would be printed as, "2 1 3".
- If we print it out in *infix* notation, that means we print out the node value *between* (or *in*-between) the left and right child. Thus, that tree would be printed as, "1 2 3".

- If we print it out in *postfix* notation, that means we print out the node value *after* the children (and always the left child before the right child). Thus, that tree would be printed as “1 3 2”.

In all cases, “printing a child” means recursively printing the entire sub-tree contained underneath that child. For example, when we say to print the left child before printing the right child, all the data contained at and under the left child will be printed before anything contained at or under the right child is printed.

Here is how to represent the first tree shown above:

```
Prefix: 1 2 4 5 3 6 7
Infix: 4 2 5 1 6 3 7
Postfix: 4 5 2 6 7 3 1
```

Here is how to represent the second tree shown above:

```
Prefix: 1 2 5 4 7 3 6
Infix: 2 4 5 7 1 3 6
Postfix: 4 7 5 2 6 3 1
```

Given a tree printed out in prefix and infix form, print it out in postfix form

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case will consist of three lines. The first line of a test case is a number v, the number of vertices (nodes) in the tree. The second line is a list of v integers that is the tree in prefix form. The third is a list of v integers that is the tree in infix form.

Each test will have at most 1000 vertices. Each vertice will have a value between 0 and 1000000

Output Format

Each test case will have a single line in the output, with a space-separated list of the tree vertex values in postfix form. It does not matter if there is a trailing space on a line.

Sample Input

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

Sample Output

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

I. How Much Longer?

Description

When the Avengers put Loki in prison, they gave him an incredibly menial task and told him that if he did it successfully, they would let him out. They figured this task was so mind-numbingly boring that he would never be able to do it, so there was little risk.

Specifically, Loki has to count the number of times the hour hand and minute hand cross each other over a set period of time for a number of different clocks the Avengers put in the room. However, they all work differently than normal clocks. Loki wants to write a program that will calculate the answers for him, so he has more time to plot how to take over Asgard. The format of the clocks is described below.

Earth's predominant analog clock design has a circular face, as well as an hour hand and a minute hand which move smoothly around the clock face. Every revolution of the minute hand around the clock face indicates that the hour hand has moved forward one hour. Usually, one revolution around the clock face represents 12 hours for the hour hand and 60 minutes for the minute hand. For this problem we are not considering the distinction between AM/PM.

What if clocks had a different number of hours and minutes? Given an analog clock that represents h hours for the hour hand and m minutes for the minute hand, as well as a start time and an end time, determine how many times the minute hand overlaps with the hour hand.

If the minute hand and hour hand overlap at the start time or end time, that does count as an overlap. For our clocks, the hour hand and minute hand both move smoothly, even between minutes. This means that at 0:01, the hour hand is no longer exactly on the 0 o'clock position.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each of the next n lines will be a test case containing four integers in the form h m s e, where h is the number of hours the clock represents, m is the number of minutes the clock represents, s is the start time in the form (hours):(minutes), and e is the end time in the form (hours):(minutes). The integers h and m are guaranteed to be greater than 1. The hours and minutes that make up s and e can vary in number of digits, depending on the values of h and m. Hours begin at 0 and end at $h - 1$.

Each clock will have at most 12 hours and at most 60 minutes.

Output Format

For each case, output the number of overlaps there are in the given time range for the given time system, each on a separate line.

Sample Input

```
2
12 60 0:00 11:59
4 13 2:07 3:05
```

Sample Output

```
11
1
```

J. Important Puzzles

Description

The Avengers are trying to catch Loki, but he's trapped himself behind a series of locked doors. Thor tried to smash through them, but they're made of the same material as the prison cell on the helicarrier. The doors can each be unlocked by solving a puzzle consisting of a row of coins, some showing "heads" and some showing "tails". The Avengers need to figure out if each door can possibly be unlocked given its initial configuration and a set of possible moves they can make, or if they need to call Nick Fury and come up with a backup plan.

The puzzle consists of a row of coins that all start face-side up. There is a "goal" configuration of the board, in which coins may be oriented as "heads" or "tails." There is also a set of allowed "moves" – a move consists of a list of coins that *all* must be flipped when someone uses that move. The object of the puzzle is to determine a sequence of moves that will take the board from the starting configuration to the goal configuration.

For example, one instance of the puzzle could be described as follows:

Goal configuration: `HTTHT`. Possible moves:

1. Flip coin 5
2. Flip coins 1 and 4
3. Flip coins 1, 2, 3, 4, and 5
4. Flip coins 1, 4, and 5

To solve this puzzle, we could apply move 2 then move 3:

$$\text{HHHHH} \rightarrow \text{move 2} \rightarrow \text{THHTH} \rightarrow \text{move 3} \rightarrow \text{HTTHT}$$

There may be multiple ways to solve the puzzle; for instance, we could also apply move 1, followed by 3, and then 4:

$$\text{HHHHH} \rightarrow \text{move 1} \rightarrow \text{HHHHT} \rightarrow \text{move 3} \rightarrow \text{TTTTH} \rightarrow \text{move 4} \rightarrow \text{HTTHT}$$

However, there are also puzzles that cannot be solved. Your program, given a goal configuration and list of moves, must determine whether or not it is possible to solve the puzzle.

Input Format

The first line of the file will have an integer, n, which is the number of test cases in this file.

Each test case starts with a line containing two positive integers, l and m, denoting the board length and number of possible moves, respectively.

The second line of each test case is a string of l characters (H's and T's) describing the goal configuration.

After this, m lines follow, each describing a possible move. A move is given as a string of S's and F's, representing coins that stay (for 'S') or flip (for 'F') on that move.

Note that the first test case shown below is the puzzle given above.

Each board will have a length between 5 and 64, and between 1 and 64 possible moves.

Output Format

If the puzzle can be solved, print "CAN BE SOLVED". If not, print "CANNOT BE SOLVED".

Sample Input

```
5
5 4
HTTHT
SSSSF
FSSFS
FFFFF
FSSFF
5 1
HTTHT
SSSSF
5 2
HTTHT
SSSSF
SFFSS
3 1
HTH
SFS
3 1
HTH
FSF
```

Sample Output

```
CAN BE SOLVED
CANNOT BE SOLVED
CAN BE SOLVED
CAN BE SOLVED
CANNOT BE SOLVED
```