BWC++ Programming C# Contest 20

DO NOT OPEN UNTIL INSTRUCTED!!!!!

Bank Robbin

Your friend, Rob N. Hood, is an expert, but lazy, bank robber. For many years, he has been robbing banks and being the people's hero. Lately banks have begun to use electronic locks that require a sequence of hexadecimal numbers to unlock. However Rob is very lazy and doesn't understand numbers very well (particularly "funny" numbers that involve A, B, C, etc.) so he has asked you to write an application that he can use on his trusty smart phone to help him unlock the safe when he is robbing the bank.

Both lock and key consist of exactly 6 hexadecimal digits. A key will unlock the lock if each corresponding digit in the lock and key add up to F. That is, if all of the digits in the key and lock individually add up to F, then the key will unlock the lock.

Details of the input

The first line of input will be a single integer, *n*, that indicates how many locks you will be testing. For each case, the first line of input will be a single integer, *k*, that represents the number of keys you are to test for that lock. The next line of input will be a string representing the hexadecimal number for the current lock. Each of the next *k* lines will contain a string representing a key combination that you are to test against the lock. The locks and keys will be in hexadecimal format with the letters in uppercase.

Details of the output

For each case, the program should display "UNLOCKED BY KEY XXXXXX" replacing XXXXXX with the correct 6 digit key value if a one of the keys is successful. If a correct key is not found then the program should output "NOT UNLOCKED".

Sample Input

```
2
5
000000
123456
ABCDEF
023456
A1B2C3
FFFFFF
3
ABCDEF
123456
222222
000000
```

Sample Output

```
UNLOCKED BY KEY FFFFFF
NOT UNLOCKED
```

Tower of Defense

Tower Defense is a popular subgenre game. The goal in a Tower Defense game is to stop the enemies that are walking across a path by 'shooting' at them from a tower. If the enemies make it to their destination, then the player loses the game. The paths and types of enemies can vary from game to game.

In our B-W Tower Defense game, paths and towers are placed on a map which we represent as a two-dimensional grid. Each space in the grid can be part of a path, have a tower, or contain nothing. The enemy will walk the grid along the path, which will always start at the upper left corner of the grid (which will have coordinates 0,0) and end at the bottom right corner.

Towers can be placed anywhere on the grid but will never be right on the path**.** Shots can only reach the enemy from a tower that is directly adjacent, including diagonally, to the path. Shots from a single tower can hit the enemy once for each path block adjacent to it.

The enemy walks along the path one grid position at a time. When the enemy is at a grid position, each tower within range shoots at the enemy once. When the enemy moves to the next position, each tower within range shoots at the enemy again. This enemy starts with a number of hit points that enables him to survive that number of shots from the towers. Every time a shot from a tower hits the enemy, his hit points are decremented by one. If the enemy's hit points fall below 0, the enemy dies, and you win the round. Two towers can hit the enemy on the same path position if they are both adjacent to the path, and the enemy loses two hit points at that position. If the enemy passes through the last grid position on the path and still has more than 0 hit points left, you lose the round.

The game board grid is limited to a maximum size of 10 by 10 and a minimum size of 2 by 2, and does not necessarily need to be a square board.

Details of the input

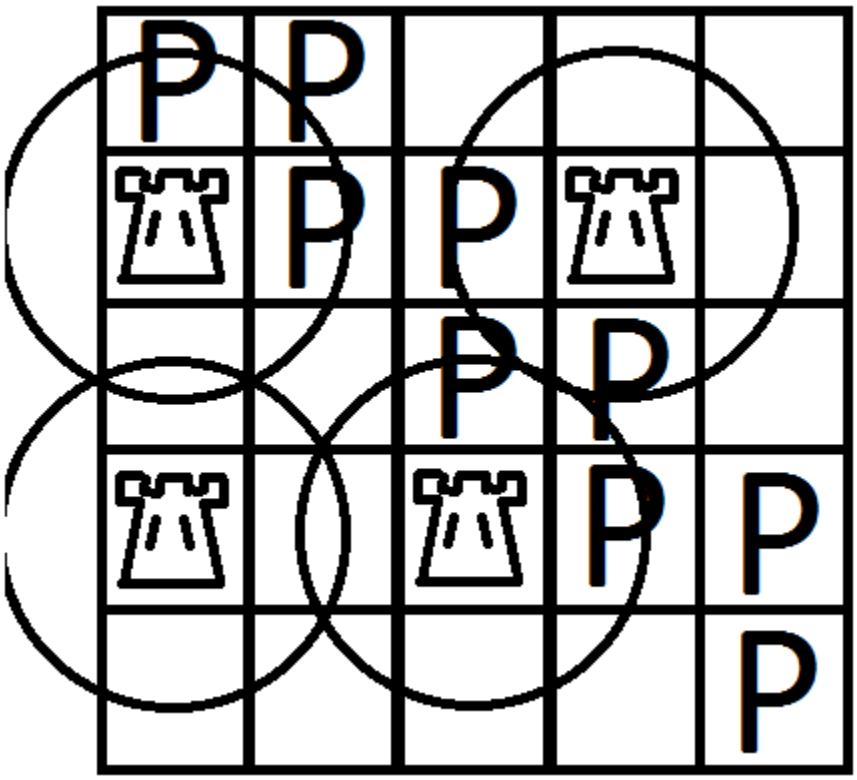
The first line of input is an integer, *n,* that indicates the number of rounds to be played. For each round, the first line of input contains two positive integers, *r* and *c*, each of which will be in the range 2..10, indicating the number of rows and columns in the grid, respectively. The next line of input contains a single positive integer, *k*, that indicates the number of moves in the path. The next *k* lines each contain two non-negative integers that give the coordinates (row then column) of the move in the grid. On the line following the last move is a single integer, *t*, which specifies the number of towers. The next *t* lines each contain two integers that specify the positioning (row then column) of the towers. The last line of input is a single integer, *h*, that indicates the number of hit points the enemy has for the round.

Details of the output

The output for each round is a single line that either states "PASSED" if the enemy makes it through the grid or "FAILED" otherwise.

Sample Input

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



Sample Output

```
FAILED
```

Collatz Conjecture

Consider the following function:

$$C(n) = \begin{cases} 3n+1, & \text{if } n \text{ is odd} \\ n/2, & \text{if } n \text{ is even} \end{cases}$$

In 1937, Lothar Collatz conjectured that the sequence formed by repeatedly applying this function (taking the output of one step and using it as input in the next step) would eventually reach 1 for all positive starting values of *n*. Here's an example of the sequence using 7 as *n*:

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

The Collatz Conjecture has never been proven to reach 1 for all positive input, but it has been shown to reach 1 for numbers as large as 2,362,292,408,970,609,843,065 (much larger than the largest number that can be stored in a 32-bit integer – if you're interested in helping the distributed computing project dedicated to testing large numbers against the Collatz Conjecture, visit boinc.thesonntags.com/collatz/).

We can stop computation early if the sequence reaches an integer which is a power of 2. If it reaches a number $n = 2^k$ (for any non-negative k, because a value of $k = 0$ corresponds to $2^0 = 1$, which is a power of 2), then the sequence will then repeatedly fall into the "even" category and go $2^k \rightarrow 2^{k-1} \rightarrow \ldots \rightarrow 4 \rightarrow 2 \rightarrow 1$. Looking at the example that began with $n = 7$, the first power of 2 reached is 16 (you can see from that point forward each step divides by 2 until it reaches 1). The goal of this problem is to determine the first power of two reached in the sequence created starting by with *n* and repeatedly applying the C(*n*) function. We will call the first power of 2 appearing in the sequence CC(*n*).

Details of the Input

The first line of the input file is a single integer, *i*, which indicates the number of cases to follow. The following *i* lines will each contain single positive integer representing *n*. You are guaranteed that the values produced while repeatedly calculating C(*n*) will never exceed the maximum value of a signed 32-bit integer.

Details of the Output

For each case, output

CC(<*n*>)=(<result>)

where *n* is the value given in the input, and result is the calculated value of CC(*n*).

Sample Input	**Sample Output**
2	
7	CC(7)=16
4	CC(4)=4

N Degrees of Ashton Kutcher

Movie buffs have long (well, since the mid-1990's) played a game called *Six Degrees of Kevin Bacon*. In the game, players try to link a given actor to Kevin Bacon through their in roles a series of movies. For example, Charlie Sheen was in *Ferris Bueller's Day Off* with Edie McClurg, who was in *She's Having a Baby* with Kevin Bacon. Thus, Charlie Sheen is two degrees from Kevin Bacon.

The game is based on the high likelihood that most notable actors with a finite distance from Kevin Bacon would be at most six degrees from him. Of course, not all actors are within six degrees of Kevin Bacon, but enough are to make the game fun.

(Aside for future mathematics majors: Paul Erdős, prolific 20th century mathematician, has a similar distance rating scheme bearing his name. The Erdős numbers serve as an amusing measurement of mathematical prominence in that era based on co-authorship of articles.)

As a budding social media expert, you're looking to measure your success on Twitter, the popular microblogging service. Twitter has over 190 million active users, but you're really only interested in your distance from one of them: Ashton Kutcher (known by his username, "@aplusk").

The number of degrees from Ashton Kutcher to a particular Twitter user is less straight-forward to calculate than the Bacon (or Erdős) numbers above because Twitter has no guaranteed bidirectional relation to use (like co-starring in a movie or co-authoring a paper). Instead, you'll make use of two particular Twitter features: the ability of a user to *follow* another user and/or *mention* another user.

- *Follow*: A Twitter user can subscribe to see messages from other Twitter users on their home page by "following" them. If user A follows user B, then the distance from A to B is 1. However, there is no direct connection from B to A.
- *Mention*: Twitter users can mention other users as part of their posts. If user A mentions user B, then it is clear that user A knows who user B is, but they may or may not respect B enough to actually follow them. Thus, if A mentions B, then the distance from A to B is defined as 2. Again, this does not imply any direct from B to A.

A consequence of this rating system is that the distance between two users could be different depending on which direction you are interested in. For our purposes, we're only interested in the distance *from* Ashton *to* the specified user.

Using this system, you will be asked to calculate the minimum degree of separation from Ashton Kutcher to several Twitter users.

Details of the Input

The first line of input will be a single integer, *n*, that represents the number of cases to follow. Each input case will begin with a single integer *p* ($1 < p < 11$) on its own line, representing the number of Twitter users to be considered.

Each of the *p* twitter users will be described using exactly three lines:

- Line 1 contains the current user's Twitter username. All usernames will begin with an ampersand ('@') and the remainder of the username will contain only alphabetic characters, numeric characters, or underscores.
- Line 2 contains a list of the Twitter users that the current user is following. The line will begin with a single integer *f* indicating the number of users followed by the current user. A space delimited list of *f* distinct Twitter usernames follows. Users cannot follow themselves.
- Line 3 contains a list of the Twitter users mentioned by the current user. The line will begin with a single integer *m* indicating the number of users mentioned by the current user. A space delimited list of *m* distinct Twitter usernames follows. Users can mention themselves.

The very first Twitter user described will always represent the person whose distance from Ashton we are trying to determine. Ashton will always appear in the user list, but could appear in any location in the list (other than the very first position). He will always be identifiable by his username, "@aplusk".

Details of the Output

For each ease print one of the following lines depending on whether there exists any path from Ashton to the user:

```
Case k: d
Case k: No connection
```

The value of *d* represents the smallest distance from Ashton to the user.

Sample Input

```
2
3
@me
1 @aplusk
1 @dave
@dave
2 @me @aplusk
1 @me
@aplusk
0
1 @dave
2
@my_name
1 @aplusk
1 @aplusk
@aplusk
0
1 @aplusk
```

Sample Output

```
Case 1: 3
Case 2: No connection
```

Broken Texting

Your Friend John's phone has lost the ability to text using alphabetic lettering. When he writes a text, he can no longer generate letters of the alphabet by repeatedly pressing the number keys. In order that he can continue to text his friends, John has decided to create a substitute message scheme involving patterns of numbers to represent letters. Since typing the letter 'B' would normally require pressing the '2' key twice, the letter 'B' is represented by '22' in the new scheme. Similarly, 'C' is represented by '222' and 'T' is represented by '8'.

When texting, a longer pause between key presses is used to indicate that the user wants to stop cycling through letters related to a number. So in order to spell a word like "bar", the '2' key is pressed 3 times with a pause in between the 2nd and 3rd press, in order to indicate that the user wants 'ba' instead of 'c'. John has decided to use a space to represent a pause in the textual version of the message. Thus, 'ba' would be '22 2'.

The use of a space to represent pauses means that there must be another character used to represent actual spaces. Normally when texting, the '#' key is used to create a space between words, so this is how spaces will appear in the textual version of the message.

John believes that you can decipher his 'sentence', consisting of numbers 2 through 9, the '#' symbol, and spaces, into the desired sentence he wants. John is limited in the length of a text, so his input 'sentences' are no longer than 255 characters.

Details of the input

The first line of input will be an integer, *n*, representing the number of transmitted sentences that you must decode. For each of the *n* sentences there will be one line of input no longer than 255 characters.

Details of the output

The program should output one line for each transmitted sentence in its translated form. All letters should be output in lowercase.

Sample Input

```
1
844 4447777#4447777#2#83377778#66 666844 444664#22338 833777#2233#9777666 664
```

Sample Output

```
this is a test nothing better be wrong
```

A Rough Slide

Many a high school physics student has spent time calculating the time it takes for an object, starting at rest, to slide down an incline. In this problem, we "ramp up" this basic problem slightly by placing materials of different consistencies and varying lengths on the surface of the ramp that will cause the object to experience variation in its acceleration as it makes its way to the bottom. The goal is to calculate the time it takes for the object to reach the bottom to the closest one second.

Solving this problem requires a few basic physics equations. The first is the classic equation F=MA, where F is the force on the object, M is the mass of the object, and A is the acceleration on the object. This equation can be rewritten in the form A = F/M, which is a useful form if you need to determine acceleration (which we clearly do in this problem). The next equation needed is that to compute the total force, F, which acts upon the object. In this situation, total force is the sum of two forces, $F = F_f + F_g$ where F_f is the force due to the friction created by the various surfaces on the incline and F_g is the force due to gravity. Another equation needed is that to compute the force due to friction, $F_f = \mu F_n$, where μ is the coefficient of friction and F_n is the normal force. Once the acceleration of an object is known, speed can computed by recognizing that the increase/decrease in speed is a product of the acceleration and the length of the time interval (1 second in our case). Likewise, the distance traveled over a time interval is the product of the speed and the length of the time interval.

The complexity of solving this problem is due to the various surfaces on the incline that the object experiences as it moves. These varying surfaces mean that the coefficient of friction continues to change each time a new surface is encountered. As a result, the acceleration of the object must be recalculated frequently. In reality, the acceleration changes continuously, but we will simplify the problem somewhat by defining that the acceleration will be recalculated after each one second time interval. There is a slight twist to utilizing the acceleration, however, which needs to be explained. The speed of the object does not immediately increase by the acceleration at the beginning of a time interval, but gradually increases to its top speed by the end of the interval. To simulate this gradual increase in speed, we will assume that the average acceleration over the time interval was only ½ of what it is at the end of the interval. For example, if your calculation of acceleration at the end of an interval is 10 ft/sec^2, you will use 5 ft/sec^2 as the acceleration that impacts the speed during the interval. At the end of the interval, however, you need to account for all of the acceleration so the speed should be increased to account for the other 5 ft/sec^2 of acceleration before beginning the next interval.

Another consideration that must be made is that it is possible that the speed of an object as it reaches the beginning of a surface is so fast that the object passes up that surface before any impact is realized. For example, let's assume one of our surfaces is 2 feet long and that as the block slid through the previous surface, its built-up speed at the last time interval caused it to move beyond the end of that previous surface by 3 additional feet. Then the impact of the 2 foot section would never be realized. This situation is not difficult to recognize since you will be keeping track of the total distance traveled by the object, so if your total distance is further than the end of the previous section by more than 2 feet, you can ignore the two foot section in your calculations.

You may safely assume that all calculations will produce only integer results. You should report the total time at the first point of recalculation where the distance traveled is greater than the length of the ramp.

Details of the input

The first line of input will contain a single integer, *n*, which indicates the number of test cases to follow. For each case, the first line of input will have 3 integers representing F_n, F_g, and M, where F_n is the normal force on the box, F_g is the force due to gravity on the box in the direction of the ramp, and M is the mass of the object. The next line will have a single integer,*k* , which specifies the number of sections on the ramp. This is followed by *k* lines, each containing two integer values, *l* and μ, where *l* is the length of the section in meters, and μ is the coeffecient of friction of that section. The first of these *k* lines corresponds to the top section of the ramp and the last to the bottom.

Details of the output

For each case, one line of output is generated. The output should read "Case *i*: *X*" where *i* is the case number, starting at 1 and going to *n*, and *X* is the total time it took the box to reach the bottom of the ramp in seconds for that case.

Sample Input

```
1
5 60 5
4
4 2
5 4
6 6
7 8
```

Sample Output

```
Case 1: 3
```