The 2nd Annual University of Akron Programming Competition

Presented by
The University of Akron Computer Science Department
Association for Computing Machinery Student Chapter

April 30, 2011

Rules:

1. There are **seven** questions to be completed in **four hours**.

2. C, C++ and Java are the only languages available.

3. Data is read from Standard Input and output is sent to Standard Output. Do not prompt for input values in the code you submit to be judged. Do not attempt to read from or write to any files.

4. All programs are submitted as source code only. Submitting compiled information (binary, .class) will result in a Compilation Error penalty

5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed. The standard Java API is available, except for those packages that are deemed dangerous by contest officials (e.g., that might generate a security violation).

6. The input to all problems will consist of multiple test cases unless otherwise noted. The input listed on the problem description is not intended to be a comprehensive input set. The input is guaranteed to adhere to the descriptions in each problem; you do not need to check for invalid input.

7. Use of personal electronics during the competition is cause for disqualification. Cell phones must be turned all the way off (not silent mode, not airplane mode, etc.). You may use any written notes, books, or reference materials you bring with you.

8. Programming style is not considered in this contest. You can code in any style or with any level of documentation your team prefers.

9. All communication with the judges will be handled through PC^2. All judges' decisions are final.

[intentionally left nearly blank]

Problem A *Overlapping 2D Barcodes*

Two dimensional barcodes have recently become prominent as the need to encode more data has increased and the technology required to scan and interpret them has become cheap. These 2D barcodes consist of a matrix of pixels, each of which is either white or black. An example QR code (specific type of 2D barcode) appears in Figure 1.

Figure 1

A local retail store has started a practice of printing stickers with QR codes on them to place on their merchandise. They've used a transparent sticker material onto which they only print the black pixels. They then place the printed sticker onto a white area of each product's packaging so that the scanners can correctly read and interpret the barcode.

Occasionally, an employee accidentally places the wrong sticker on a product. Most employees peel the incorrect sticker off and then affix the correct sticker, but not Bob. Nope. Bob just sticks the correct sticker down right on top of the incorrect sticker. Because of the transparent sticker material being used, the resulting "barcode" is really a combination of both the incorrect sticker's black pixels and the correct sticker's black pixels. Sometimes he uses the wrong sticker on his second attempt as well, so he just keeps trying until he gets it right.

In this problem, you will show the resulting "barcode" on a series of products after one of Bob's night shifts.

Details of the Input

Each input case will begin with a single positive integer $n < 10$ indicating the number of incorrect stickers placed on this product before the correct sticker is finally added. A value of $n = 0$ indicates the end of input and should not be processed.

There will then be the description of $n + 1$ QR codes (n incorrect stickers and one correct sticker). Each QR code is 21 x 21 pixels, so they will each use 21 lines in the input (each of which will have 21 characters). Black pixels are represented by an 'X' while transparent pixels are represented by an '_' (underscore).

Details of the Output

For each case, print the case number and display the resulting QR code as formatted in the Sample Output.

Sample Input

```
2
XXXXXXX_XXX___XXXXXXX
X_____X_______X_____X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_____X_______X_____X
XXXXXXX_______XXXXXXX

_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
XXXXXXX_____________
X_____X_____________
X_XXX_X_____________
X_XXX_X_____________
X_XXX_X_____________
X_____X_____________
XXXXXXX_____________
XXXXXXX___XX__XXXXXXX
X_____X_______X_____X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_____X_______X_____X
XXXXXXX_______XXXXXXX

_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
XXXXXXX_____________
X_____X_____________
X_XXX_X_____________
X_XXX_X_____________
X_XXX_X_____________
X_____X_____________
XXXXXXX_____________
XXXXXXX_______XXXXXXX
X_____X_______X_____X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_____X_______X_____X
XXXXXXX_______XXXXXXX

_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
XXXXXXX_____________
X_____X_____________
X_XXX_X_____________
X_XXX_X_____________
X_XXX_X_________XXXXX
X_____X_____________
XXXXXXX_____________
0
```

Sample Output

```
Case 1:
XXXXXXX_XXXX__XXXXXXX
X_____X_______X_____X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_XXX_X_______X_XXX_X
X_____X_______X_____X
XXXXXXX_______XXXXXXX

_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
XXXXXXX_____________
X_____X_____________
X_XXX_X_____________
X_XXX_X_____________
X_XXX_X_________XXXXX
X_____X_____________
XXXXXXX_____________
```

Problem B *Randomly Evaluated Boolean Expressions*

You've grown tired of all the new programming languages released year after year, and decided that they're all just a bit too deterministic, so you decide to make a new programming language that inserts some fun (randomness) into an otherwise boring world of 0s and 1s.

The first draft of your language is rather limited, and methods that return Booleans are particularly constrained. They may contain 0 or more local variable definitions and exactly one return statement, but there are a couple twists (remember, this is supposed to be a fun language!).

Local variable definitions and return statements both consist of Boolean expressions involving Boolean constants ('T' for true, 'F' for false) and previously defined local variables, along with a set of predefined operators: {"or", "and", "xor"} (each described in Figure 2). These expressions do not involve any sort of negation operator, nor do they involve parentheses. Furthermore, your language specifies that all operators have the same precedence level and that Boolean expressions will be evaluated in random order. That is, when it comes time to evaluate a Boolean expression, the runtime randomly inserts sufficiently many sets of parentheses to guarantee a specific evaluation ordering. When an expression is evaluated for a 2nd time, the runtime starts over and randomly inserts the parentheses from scratch.

x	y	x and y	x or y	x xor y
T	T	T	T	F
T	F	F	T	T
F	T	F	T	T
F	F	F	F	F

Figure 2

To make things even more exciting, you've added an "optimization." There may be local variables that get defined but never used, so you want to avoid having to go through the intensive process of inserting parentheses and evaluating if the resulting value is just going to be thrown away. So, when evaluating a method, you start with the return statement. Then, for each local variable encountered in the return statement, that local variable is evaluated (randomly) at that time. If the same local variable appears twice in the return statement, then that local variable is evaluated (randomly) two times, which could result in different values. In the process of evaluating local variables, you might have to go back and evaluate earlier local variables.

Consider the method `foo` written in this new language:

```
Boolean foo()
{
     Boolean x = T or F xor T;
     return x xor T or T;
}
```

The evaluation process begins with the return statement, and based on the return statement we know that we're going to need one random evaluation of x before we can successfully evaluate the return statement as a whole, so we jump to the definition of x.

The definition of x could be evaluated in the following ways:

```
Boolean x = ((T or F) xor T);     Results in F
Boolean x = (T or (F xor T));     Results in T
```

There are 2 evaluations of x, one of which is T, and one of which is F.

Now we are ready to consider the return statement as a whole. Here are the possible evaluation orderings of the return statement:

```
return ((x xor T) or T);      If x is T, then this results in T.
                              If x is F, then this results in T.
return (x xor (T or T));      If x is T, then this results in F.
                              If x is F, then this results in T.
```

Thus, there are 4 possible evaluations of the overall method, 3 of which result in T.

You are tasked with figuring out the total number of evaluations of a method and how many of those evaluations result in a return value of T (in other words, find the probability of the return value being T in unreduced form). Note that the expression "`T or T or T`" has 2 evaluations, regardless of the fact that they're both always T.

Details of the Input

Each case will begin with a line containing a single positive integer $m < 6$ indicating the number of lines in the method. A value of $m = 0$ indicates the end of input and should not be processed. The first m-1 lines contain local variable declarations formatted as follows:

```
Boolean <b> = <expression>;
```

The value of <*b*> will be a single lowercase character. Each variable declaration will use a unique value for <*b*>.

The final line contains the return statement, formatted as follows:

```
return <expression>;
```

In both cases, <*expression*> is a Boolean expression as described above and as formatted in the sample input. The total number of evaluations for the return statement and any individual local variable will be no greater than 2^{63}-1.

Details of the Output

For each case, print

```
Case k: t of c
```

where k is the case number, c is the total number of possible evaluations of the method, and t is the number of evaluations where a value of true is returned.

Sample Input

```
1
return T or T or T or T or T;
2
Boolean x = T or F xor T;
return x xor T or T;
2
Boolean a = T or T or T;
return a and a and a;
3
Boolean x = F xor F or T;
Boolean y = x xor T and T or x;
return x xor y or x and F;
0
```

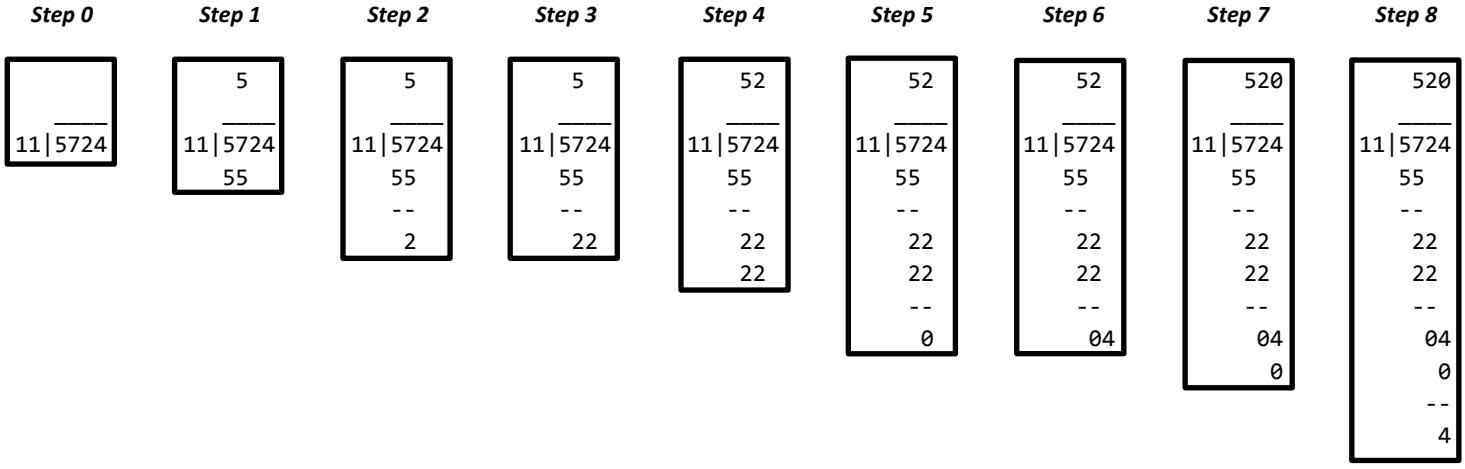
Sample Output

```
Case 1: 14 of 14
Case 2: 3 of 4
Case 3: 16 of 16
Case 4: 176 of 400
```

Problem C

Long Division

The process of long division is best described by an example. Consider the following simple nine step process required to calculate 5724 ÷ 11 using long division:



In this example, 5724 is called the dividend and 11 is called the divisor.

Step 0: Set up the basic structure of the division. The divisor appears on the left, followed by a vertical bar and then the dividend. The dividend is also covered from above by underscores on the previous line. Our eventual answer (called the *quotient*) will appear on the line above the line with the underscores.

Step 1: Find the smallest contiguous leftmost subset of the dividend that is greater than or equal to the divisor. In this case, the contiguous leftmost subsets of the dividend form the set {5, 57, 572, 5724}. The smallest of these which is greater than 11 is 57. We then calculate the largest integer we can multiply the divisor by while keeping the product less than or equal to 57. In this case, that number is 5 (since 5 * 11 = 55 ≤ 57 < 66 = 6 * 11), which we write in the quotient area, right aligned above the 57 (so directly above the 7). The product of 5 and 11 (which is 55) is then written on the next blank line, right aligned under the 57.

Step 2: We now want to subtract 55 from 57. We draw a horizontal line (using hyphens) on the next line long enough to cover all of the columns in which any part of this subtraction appears. In this case, the 57 and 55 both appear as part of the same two columns, so the horizontal line spans those two columns. The result of the subtraction is then written in its simplest form (with no leading zero unless the result of the subtraction is 0, in which case a single 0 character is the result), right aligned under the horizontal line on the following line. In this case, the result of our subtraction is 57 – 55 = 2.

Step 3: We are now ready to move to the next column, so we copy the digit from the immediate right of the 57 in the dividend down to the immediate right of the result of our subtraction in the previous step. In this case, we copy down the 2 from 57**2**4 and place it to the right of the 2 (from 57 – 55) to result in 22.

Step 4: Calculate the largest integer we can multiply the divisor by while keeping the product less than or equal to our new number, 22. In this case, that number is 2 (since 2 * 11 = 22 ≤ 22 < 33 = 3 * 11), which we write in the quotient area, right aligned above the 22 (so immediately to the right of the 5 that's already part of the quotient). The product of 2 and 11 (which is 22) is then written on the next blank line, right aligned under the previous number (which was also 22 in this case).

Step 5: Same as Step 2. We draw our horizontal line two characters wide on the next line, then write the result of our subtraction, 0, on the following line right aligned under the new horizontal line.

Step 6: Same as Step 3. Copy down the 4 and place it to the right of the 0 (from 22 – 22) to result in 04.

Step 7: Same as Step 4. The largest number we can multiply by the divisor while keeping the product lest than or equal to 4 is 0 (since 0 * 11 = 0 ≤ 04 < 11 = 1 * 11), which we write in the quotient area, right aligned above the 04. This makes our quotient 520. The product of 0 and 11 (which is 0) is then written on the next blank line, right aligned under the 04.

Step 8: Same as step 2. We draw our horizontal line two characters wide on the next line (because it's "04 – 0", not "4 – 0"), then write the result of our subtraction, 4, on the following line right aligned under the new horizontal line.

And that's it. The 4 on the final line is called the *remainder*. The actual result of the division is 520 4/11. As you might have noticed, Step 2, Step 3, and Step 4 cycle for each column once you figure out the correct starting column in Step 1. You will be taking integers and showing how to divide one by the other using long division.

Details of the Input

Each case will consist of a single line with two integers, a and b, with $0 < b \leq a < 10^{18}$. Input values of $a = b = 0$ indicate the end of input and should not be processed. You are to compute $a \div b$ (the long way).

Details of the Output

Output the case number on its own line, then display the long division output as seen in the Sample Output and as described in the problem statement. Lines should never end with whitespace.

Sample Input

```
45 3
5724 11
0 0
```

Sample Output

```
Case 1:
  15

3|45
  3
  -
  15
  15
  --
   0
Case 2:
    520

11|5724
   55
   --
    22
    22
    --
     04
      0
     --
      4
```

Problem D *Heart Monitor*

Patients with many types of heart problems are asked to wear heart monitors to help doctors capture information about their heart activity during their day-to-day lives. One particular brand of heart monitor, DiscountHeartMon, records the patient's heart rate in beats per minute (BPM) every 10 seconds. DiscountHeartMon is known for two main things (neither of them very good):

1. The connection between DiscountHeartMon and the patient is sometimes disrupted, resulting in "impossible" heart rate values (less than 29 BPM or greater than 250 BPM).
2. DiscountHeartMon is also not very good at reporting its findings; it simply outputs one long list of rates (in beats per minute, or BPM). Thus, the work of analyzing a recording is left to a doctor.

In addition to "impossible" heart rates that must be weeded out due to the incompetence of the DiscountHeartMon designers, doctors also want to make sure there are no "concerning" recordings. The doctors who work with DiscountHeartMon are just about as lazy as the folks behind DiscountHeartMon themselves, so they identify a heart rate as "concerning" if it is more than 30 BPM different from the immediately previous rate (but only if both the current rate and the previous rate are not "impossible").

The ineptitude of these doctors truly knows no bounds, so they've decided to combine their collective knowledge and skills to automate the process of identifying "impossible" and "concerning" heart rates. But, after politely requesting "teh codez" they might use to automate this process on various web forums, they've given up and hired you. Please take this seriously as lives hang in the balance.

Details of the Input

Each case will begin with a positive integer $n \leq 100$ representing the number of heart rates (in order, as recorded by DiscountHeartMon) to follow. A value of $n = 0$ indicates the end of input and should not be processed.

The following n lines each contain a single non-negative integer less than 1000 representing a heart rate in BPM.

Details of the Output

For each case, print the case number and the number of "impossible" and "concerning" heart rates as formatted in the Sample Output.

Sample Input

```
4
60
90
50
94
4
100
260
240
200
0
```

Sample Output

```
Case 1: 0 Impossible, 2 Concerning
Case 2: 1 Impossible, 1 Concerning
```

[intentionally left nearly blank]

Problem E — *Inverse Function Domains*

A function f(x) can be defined in many ways. One way is with a polynomial expression (e.g. $f(x) = 3x^4 - 10x^2 + 2x - 3$). Another is with trigonometric functions (e.g. $f(x) = \sin^{-1}(x / \pi)$). Another way is through a piecewise function, where the body of the function is an array of functions and associated subdomains. Consider the function shown in Figure 3. Its *domain* (set of x values used) is [0, 25], and its *image* (set of y values used) is [0, 30]. It could be defined as follows:

$$f(x) = \begin{cases} x, & \text{if } 0 \le x \le 10 \\ -x/2+15, & \text{if } 10 < x \le 20 \\ 5x - 95, & \text{if } 20 < x \le 25 \end{cases}$$

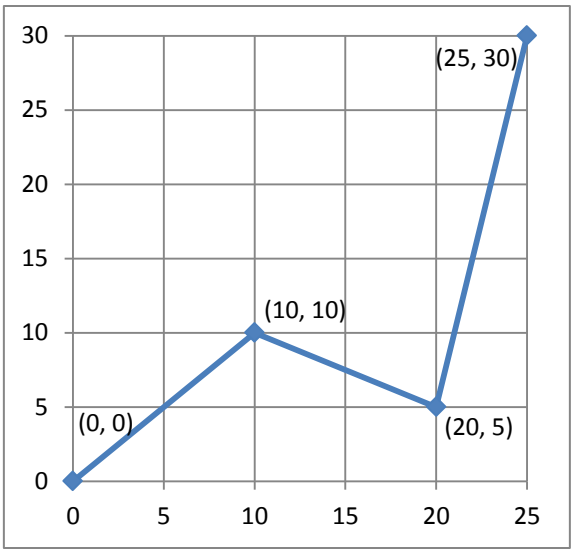


Figure 3

This type of description can be reduced down to a simple list of coordinates representing the endpoints of each piece if two conditions are met:

1. The endpoints of adjacent pieces coincide (that is, the resulting function is continuous).
2. The function representing each piece is linear (no exponents on x, trig functions, exponentials, etc.).

Thus, we will describe this function as the set of points {(0, 0), (10, 10), (20, 5), (25, 30)}.

Our function is a function from x to y where y = f(x). Consider the inverse function of f which maps y values back to x values, $f^{-1}(y) = x$. The domain of $f^{-1}(y)$ appears to be equal to the image of f(x), making the domain [0, 30]. However, there are some y values that do not map back to precisely one x value. In our example, $f^{-1}(0) = 0$, $f^{-1}(17.5) = 22.5$, but what is $f^{-1}(5)$? It could be 5 or 20. Since there are multiple options for $f^{-1}(5)$, 5 is not part of the domain of $f^{-1}(y)$.

In our example, the domain of $f^{-1}(y)$ would be all the points between 0 and 5 (including 0 but excluding 5) and all the points between 10 and 30 (excluding 10 but including 30). In set notation (explained below), this would look like [0,5)U(10,30].

You are to determine the domain of $f^{-1}(y)$ for various functions y = f(x).

Details of the Input

Each case will begin with a single integer *n* ($2 \le n \le 100$) representing the number of points explicitly defined in the graph. The following *n* lines each contain the integer coordinates of one of the *n* points (not necessarily in any sort of order) in (*x*, *y*) format as shown in the Sample Input (*x* and *y* are integers such that $0 \le x \le 300$, $0 \le y \le 300$). A value of *n* = 0 indicates the end of input and should not be processed.

Details of the Output

For each case, print the case number and the domain of the inverse function using set notation:

1. If there are no points in the domain of f^{-1}, then print "{}" to represent the empty set.
2. For each continuous set of points in the domain of f^{-1}, print the set in one of the following ways:
    a. {*y*} if the point is not immediately surrounded by other points in the domain of f^{-1}.
    b. $[y_1,y_2]$ if the entire set of real numbers between y1 and y2 are in the domain of f^{-1}, *including* both points y_1 and y_2.
    c. (y_1,y_2) if the entire set of real numbers between y1 and y2 are in the domain of f^{-1}, *excluding* both points y_1 and y_2.
    d. $[y_1,y_2)$ if the entire set of real numbers between y1 and y2 are in the domain of f^{-1}, including y_1 but excluding y_2.
    e. $(y_1,y_2]$ if the entire set of real numbers between y1 and y2 are in the domain of f^{-1}, including y_2 but excluding y_1.
3. Between each set of continuous points, add a capital letter U to indicate the union of those sets. The sets should be combined in such a way that all *y* values appearing in the output case are in non-decreasing order.

Sample Input

```
4
(0, 0)
(25, 30)
(10, 10)
(20, 5)
7
(1, 1)
(2, 0)
(3, 4)
(4, 3)
(5, 8)
(8, 8)
(10, 10)
0
```

Sample Output

```
Case 1: [0,5)U(10,30]
Case 2: {0}U(1,3)U(4,8)U(8,10]
```

Problem F — *Late For the Final Exam*

You're late for your final exam of the year, Early 21st Century Philosophy. Well, you might be late if you don't hurry up. Normally being a few minutes (or seconds (or nanoseconds)) late would not be a huge deal, but this professor is notoriously strict about timeliness, and if you miss the final you'll fail the class. A harsh punishment indeed, but the quicker you get in your car and start driving to campus the better so let's get right to the details.

You live pretty far from campus, so there are a number of different paths you could take to get there. Your car is capable of decelerating at 6 m/s^2 and accelerating at 4 m/s^2 and all the roads you would consider taking have the same speed limit (30 m/s) which you refuse to exceed because the police in your town have even harsher punishments than your Philosophy professor.

Since all the speed limits are the same, your first thought is to drive the route with the shortest distance. You quickly realize that might not be the best strategy because one particularly crazy turn on the shortest path requires you to slow down drastically, and you suspect a slightly longer path may result in a shorter commute time.

Consider the simplified map in Figure 4. Your apartment is at A and the Philosophy building is at C, and there is a one-way road from A to B and a one-way road from B to C. You leave your apartment at 0 m/s and immediately begin accelerating at 4 m/s^2. At some point, you'll need to consider slowing down to handle the turn at B. We measure the angle of the turn as shown in the figure (θ). At the point of the turn, you may be travelling no faster than:

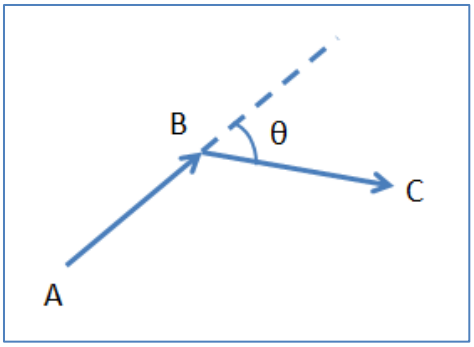


Figure 4

if $\theta \le 10°$,	30 m/s (full speed)
if $10° < \theta < 90°$,	$30 - (3/8 * (\theta - 10))$ m/s
if $\theta \ge 90°$,	0 m/s (complete stop)

So if θ in Figure 2 is 50°, then in order to make the turn towards C your speed at B may be no faster than 15 m/s, and so you may need to begin decelerating an appropriate distance before getting to the turn. If there was a road from B to some other point D, then the max speed at B would still be 15 m/s if you were turning toward C, but some (potentially) different speed if you were turning toward D.

Given just a map, you must calculate how long it would take to get to your exam using the path that takes the least amount of time to drive.

Details of the Input

Each input case will begin with a single positive integer $n \leq 10$ indicating the number of vertices on the map. A value of $n = 0$ indicates the end of input and should not be processed.

The following n lines describe each of the n vertices (numbered 1 through n) and the roads connecting them to other vertices in the following format:

$$x \ y \ c \ r_1 \ r_2 \ \ldots \ r_c$$

The vertex is located at (x, y) in the standard Cartesian coordinate system (x and y are floating point values with $-10000 \leq x \leq 10000$, $-10000 \leq y \leq 10000$, specified to no greater than $1/1000^{th}$ of a meter precision) and is connected to c other vertices, namely vertices $r_1, r_2 \ldots r_c$. This imples that roads may be one-way or bidirectional.

Each vertex will be at a distinct point, but roads may have some overlap. No pair of vertices connected by a road will be closer than 75m from each other.

Each case will then contain a list of times measured in seconds. The list begins with a single integer $k > 0$ indicating the number of times to follow. Each of the following k lines contains a time represented by a single positive floating point number no greater than 1000000. For each of these times, you are to determine whether you can get to school within that amount of time. Times will be specified with no greater than $1/1000^{th}$ of a second precision.

Details of the Output

For each case, print the case number and then iterate through the times listed and indicate whether you can get to your exam in that amount of time. The line with the case number should be left aligned and the lines describing each time should begin with 2 spaces and then be formatted as

```
<input time>: <result>
```

where `<input time>` is the time as it appeared in the input and `<result>` is either `FAIL` or `SUCCESS`.

Sample Input

```
4
0 0 2 2 3
200 0 3 1 3 4
0 220 3 1 2 4
200 200 3 1 2 3
3
25.75
26.448
25
2
0 0 1 2
400 275 0
2
24
20
0
```

Sample Output

```
Case 1:
  25.75: FAIL
  26.448: SUCCESS
  25: FAIL
Case 2:
  24: SUCCESS
  20: FAIL
```

Problem G *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 roles in 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.

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 numbers above because Twitter has no guaranteed bidirectional relation (like co-starring in a movie) to utilize. 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 connection 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

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. A value of $p = 0$ indicates the end of input and should not be processed

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

- Line 1 contains the user's Twitter username. All usernames will begin with an "at sign" ('@') 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. All followed users will be described in the input.
- 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. All mentioned users will be described in the input.

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

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

depending on whether there exists any path from Ashton to the user. The value of d represents the smallest distance from Ashton to the user.

Sample Input

```
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
0
```

Sample Output

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