A. Bowling Score Assistant

Bowling is a very popular sport in the U.S.. But projecting frames needed for winning a game is a bit painful for non-ACMers, so you are tasked with writing a program to compute this for them.

For anyone who has not bowled or has forgotten how the score is computed, there is a description of how to total a bowling score on the next couple pages, following the sample input and output. The main idea is that in order to win, you must get a higher score than your competitor.

Given your opponent's final score and the number of pins you knocked down on each ball through the eighth frame, you are to compute what you need to win (if it is even possible). If you can't possibly win, print "impossible", otherwise print the sequence of rolls that will allow you to win that is first in lexicographical order. That is, the one with the lowest first roll, lowest second roll given the first roll, lowest third roll given the first two, etc..

Input:

Input for this problem will be a series of test cases. Each case will consist of your opponent's final score followed by the number of pins knocked down with each roll of the ball for your first eight frames. End-of-input will be indicated by a negative number. All other values on the file will be valid game pin counts. Your opponent's score will be an integer from 0 to 300, and the numbers of pins knocked down on your rolls will all be nonnegative integers less than or equal to 10 (the 10 being the number of pins standing at the start of each frame). The sample input below is neatly formatted **only** to make it clear how it matches the examples that follow. You will need to process the input carefully in order to determine where one test case ends and the next begins, because you can not assume any format in terms of spacing and line breaks in the actual input file!

Output:

For each input case, print the rolls needed to win the game. Follow this format exactly: "`Case`", one space, the case number, a colon and one space, and the answer for that case given as either "`impossible`" or the required numbers of pins, with exactly one space between each pair of numbers and no trailing spaces.

Sample Input	Sample Output

```
80
4 3    6 2   4 2
8 1    4 5   6 2
7 2    9 0

215
4 6    6 2   4 2
8 1    10    6 2
7 2    9 0

299
10 10 10 10
10 10 10 10

-1
```
	```
	Case 1: 0 0 0 10 6
	Case 2: impossible
	Case 3: 10 10 10 10
	```

How to keep score in bowling:

There are ten pins that you attempt to knock down by rolling a ball at them.

For each set of ten pins, if you do not knock them all down with your first ball, you get a second try.

The one or two balls you roll at the set of ten pins is called a "frame," and there are ten frames in a game.

If you knock all the pins down with your first roll, that is called a "strike." If you knock all the pins down, but it takes both rolls, that is called a "spare."

The score is computed as the sum of the pins you knock down, but there are bonuses for spares and strikes which are computed as described below. If you never get a spare or a strike, your score is simply the total of your twenty rolls.

If you get a strike, you double count the score of the next two rolls. If you get a spare, you double count the next one roll.

The tenth frame is special in that the "next" roll(s) would not normally exist for a strike or a spare in the tenth frame. If you get a spare in the tenth frame, you get one more roll at a fresh set of ten pins which is simply added to the 10 for your spare. If you get a strike in the tenth frame, you get two more rolls. If the first of these is also a strike, you get your second roll at another fresh set of ten pins. Note, however, that you don't go forever if you keep getting strikes. You only get the two rolls after the first strike in the tenth frame, and they are simply added to the 10 for the first strike, making a maximum score of 30 for the tenth frame (and for any other frame).

Thus, you can have three rolls in the tenth frame, so the maximum number of rolls in a game is 2 * 9 + 3 = 21. The minimum number of rolls occurs if you get a strike for each of the first nine frames, but then do not get a strike or a spare in the tenth. This results in 9 + 2 = 11 rolls.

Examples:

We give the number of pins knocked down for each roll for ten frames and then compute the score.

Example 1:

1	2	3	4	5	6	7	8	9	10
4 3	6 2	4 2	8 1	4 5	6 2	7 2	9 0	0 5	6 3

Since there are no spares or strikes, the score is simply the sum of all these numbers which is 79.

Example 2:

1	2	3	4	5	6	7	8	9	10
4 6	6 2	4 2	8 1	10	6 2	7 2	9 0	0 5	6 3

This is similar to example 1, but there is a spare in frame 1 and a strike in frame 5. Note that we have a total of 10 in frame 1 before double counting the 6 (first roll) in frame 3. Also, because of the strike in frame 5, we double count the two rolls in frame 6. The final score is 97.

Example 3:

1	2	3	4	5	6	7	8	9	10
4 6	6 2	4 2	8 1	10	10	7 2	9 0	0 5	6 3

This is similar to example 2, but there is a strike in frame 6. Note that we have added two more pins being knocked down, so the base score is 85, instead of 83 as in example 2. As before, we double count the first roll in frame 2, which gives us 91. Next, we double count the two rolls after the strike in frame 5 which are 10 and 7. This gives us 108. Finally we double count the two rolls in frame 7, which gives us 117.

Example 4:

1	2	3	4	5	6	7	8	9	10
4 6	6 2	4 2	8 1	10	10	7 2	9 0	0 5	6 4 7

This is similar to example 3, but there is a spare in frame 10. The extra pin on the second roll in the tenth frame would boost the score to 118 (same computations as in example 3), but we get to add the 7 on the last roll to the spare, so the total is 125.

Example 5 (maximum score, a "perfect game"):

1	2	3	4	5	6	7	8	9	10
10	10	10	10	10	10	10	10	10	10 10 10

The total score for each frame is 30 since we get the 10 for the strike and the next two rolls which are both 10. This gives us the maximum score of 300.

B. Spectrum

Swamp County Consulting has just been awarded a contract from a mysterious government agency to build a database to investigate connections between what they call "targets." Your team has been sub-contracted to implement a system that stores target information and processes the commands listed below.

A *target* is represented by a string of up to 32 printing characters with no embedded spaces. A *connection* is a bi-directional relationship between two targets.

The *hop count* between a given target (called "target1") and other targets is determined by the following rules:
1. Targets directly connected to *target1* are 0 hops away.
2. Targets directly connected to the 0 hop targets, and not already counted as a 0 hop target or the original target, are 1 hop targets.
3. Similarly, targets directly connected to *n* hop targets, and not already counted in 0..*n* hop targets are *n*+1 hop targets.

There will be no more than 100,000 targets and no more than 500,000 connections.

Commands

The data base system has only three commands: *add, associated,* and *connections.* Targets and connections are never deleted because the Agency never forgets or makes mistakes. Commands start in the first column of a line. Commands and their parameters are separated by whitespace. No input line will exceed 80 columns. No leading or trailing whitespace is to appear on an output line.

add takes one or two parameters:

add target1 //single parameter
Function: Adds the target to the database, with no connections.
Note: If target is already in the database, do nothing (not an error)

add target1 target2 //two parameters
Function: Creates a bidirectional connection between the targets.

- If either target is not yet in the database, add it/them, and create the connection.
- If there is already a connection between the targets, do nothing (not an error)
- If target1 and target2 are the same, treat this as if the command were "add target1" (not an error) There can be at most one direct connection between targets.

connections target1
Function: Returns the number of hops to direct and indirect connections from a target.

- Print the hop count, a colon, a single space, and the number of targets with that hop count with no leading zeroes on a separate line for each hop count. Start with hop count 0 and end with the hop count for the last non-zero number of targets. Do not print trailing spaces.
- If the target has no connections, print a line containing only the string "no connections".
- If the target is not in the database, print a line containing only the string "target does not exist" (no period).

associated target1 target2
Function: Returns information about the existence of a connection between the two targets
- If there is a path between the targets, print "yes: n" on a separate line, where *n* is the hop count of *target2* with respect to *target1*. There is one space after the colon and no leading zeroes and no trailing spaces.
- If there is no connection between the targets, print "no" on a separate line.
- If either *target1* or *target2* is not in the database, print a line containing only the string "target does not exist".

Input:

To allow for multiple test cases on the input file, we add a command "reset" that is not a database command and occurs between the database commands for the different cases on the input file. Be sure to reset all of your data structures when you read this command. Process until end of file; there is no end-of-data flag and no "reset" command at the end of the file.

Output:

Start each case with a line consisting of "`Case`", one space, the case number, and a colon. End each case with a line of ten minus signs.

Sample Input

```
add a b
add a c
add b d
add e b
add c f
add c g
add f h
add h i
add j k
associated a i
associated i a
associated f k
associated a h
connections a
connections i
add k g
associated a j
connections a
add h a
connections a
associated a h
add m
add n n
connections n
add a n
connections n
```

Sample Output

```
Case 1:
yes: 3
yes: 3
no
yes: 2
0: 2
1: 4
2: 1
3: 1
0: 1
1: 1
2: 1
3: 2
4: 1
5: 2
yes: 3
0: 2
1: 4
2: 2
3: 2
0: 3
1: 5
2: 1
3: 1
yes: 0
no connections
0: 1
1: 3
2: 5
3: 1
4: 1
----------
```

C. One Move from Towers of Hanoi

For this problem, we are concerned with the classic problem of Towers of Hanoi. In this problem there are three posts and a collection of circular disks. Let's call the number of disks *n*. The disks are of different sizes, with no two having the same radius, and the one main rule is to never put a bigger disk on top of a smaller one. We will number the disks from 1 (smallest) to *n* (biggest) and name the posts A, B, and C. If all the disks start on post A, and the goal is to move the disks to post C by moving one at a time, again, never putting a bigger one on top of a smaller one, there is a well-known solution that recursively calls for moving *n*–1 disks from A to B, then directly moves the bottom disk from A to C, then recursively calls for moving the *n*–1 disks from B to C.

Pseudocode for a recursive solution to classic Towers of Hanoi problem:

```
move(num_disks, from_post, spare_post, to_post)
  if (num_disks == 0)
    return
  move(num_disks - 1, from_post, to_post, spare_post)
  print ("Move  disk ", num_disks, " from ",
    from_post, " to ", to_post)
  move(num_disks - 1, spare_post, from_post, to_post)
```

The problem at hand is determining the k^{th} move made by the above algorithm for a given *k* and *n*.

Input:

Input will be two integers per line, *k* and *n*. End of file will be signified by a line with two zeros. All input will be valid, *k* and *n* will be positive integers with *k* less than 2^n so that there is a k^{th} move, and *n* will be at most 60 so that the answer will fit in a 64-bit integer type.

Output:

Output the requested k^{th} move made by the above algorithm. Follow this format exactly: "`Case`", one space, the case number, a colon and one space, and the answer for that case given as the number of the disk, the name of the from_post, and the name of the to_post with one space separating the parts of the answer. Do not print any trailing spaces.

Sample Input	Sample Output

`1 3`
`5 3`
`8 4`
`0 0`
	`Case 1: 1 A C`
	`Case 2: 1 B A`
	`Case 3: 4 A C`

D. Four-Tower Towers of Hanoi

Refer to problem three for a description of the classic three-tower version of the Towers of Hanoi problem.

For this problem, we extend the Towers of Hanoi to have four towers and ask the question "What are the fewest number of moves to solve the Towers of Hanoi problem for a given *n* if we allow four towers instead of the usual three?" We keep the rules of trying to move *n* disks from one specified post to another and do not allow a bigger disk to be put on top of a smaller one. What is new for this problem is to have two spare posts instead of just one.

For example, to move 3 disks from post A to post D, we can move disk 1 from A to B, disk 2 from A to C, disk 3 from A to D, disk 2 from C to D, and disk 1 from B to D, making a total of 5 moves.

Input:

Input will be positive integers (*n*), one per line, none being larger than 1,000. For each value of *n*, compute the fewest number of moves for the four-tower problem. Stop processing at the end of the file. (There is no end-of-data flag.)

Output:

Output the fewest number of moves. Follow this format exactly: "`Case`", one space, the case number, a colon and one space, and the answer for that case. You may assume the answer will fit in a 64-bit integer type. Do not print any trailing spaces.

Sample Input	Sample Output

`1`
`3`
`5`
	`Case 1: 1`
	`Case 2: 5`
	`Case 3: 13`

E. Light Switches

You are given a string of synchronized blinking lights with *N* bulbs. This string of lights is interesting in that instead of blinking on and off in unison, they follow a very specific pattern. Assume that at time $t = 0$ all bulbs are off. At each subsequent (integral) time *t*, bulbs toggle from on to off or off to on depending on their current configuration.   When a bulb will toggle on or off depends on its position from the beginning of the string. If its position is a multiple of time *t*, it will toggle. So at time $t = 1$ all bulbs will toggle on (1, 2, 3, 4, etc.). At time $t = 2$ only even numbered bulbs (2, 4, 6, 8, etc.) will toggle again. At time $t = 3$ every third bulb (3, 6, 9, 12, etc.) toggles. This continues up to time $t = N$, at which point all bulbs are reset to off and the blinking pattern restarts at time $t = N+1$. (Thus time $t = N+1$ is viewed as equivalent to time $t = 1$: all bulbs are toggled on.)

Quality Control is having a hard time verifying that the bulbs are turning on and off at the appropriate times. Your team has been asked to write a verification program that can be given the number of bulbs *N* on the strand, a particular time *t*, and bulb position *b*, then determines if that bulb is on or off at time *t* + *epsilon*. In other words, if the bulb is on at time *t* + *epsilon*, then the bulb either toggled on at time *t* or was already on at time *t*.

The following limits hold for *n*, *t*, and *b*:

$$3 \le N < 2^{54}$$
$$1 \le t, b < 2^{54}$$
$$b \le N$$

[The judge's largest test case involves 17-digit numbers that start 123, so they are indeed < 2^{54}.]

Input:

Input to your program will be multiple lines each containing the number of bulbs, *N*, the time since they were turned on, *t*, and the bulb number we are interested in, *b*, separated by spaces. Read until at end of file, there is no end of data indicator.

Output:

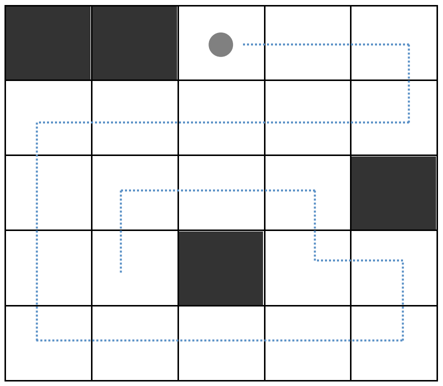
Indicate if the specified bulb is on or off at the end of the requested time. Follow this format exactly: "`Case`", a space, the case number, a colon and one space, and the answer which is either "`On`" or "`Off`". Do not print any trailing spaces.

Sample Input	Sample Output

`55 10 24`
`55 68 24`
`20 70 5`
	`Case 1: Off`
	`Case 2: On`
	`Case 3: Off`

F. Full Board

A game starts with an M × N board with some squares marked as "obstacles" (drawn as dark squares in the figure below). The player chooses a starting position for a ball (drawn as solid gray dot) and chooses a direction (up, down, left, or right) to advance. Once the direction is chosen, the ball will advance in that direction until it hits an obstacle, the boundary of the board, or its own trajectory. When it hits one of these, the ball stops. Then the player is allowed to choose another direction, and the ball will advance in the same manner. The game ends when no legal move can be made. The player wins if and only if the trajectory includes all the empty squares on the board. The figure below traces a trajectory that shows one way to win the game in 10 steps.



Given an initial setting of a board, write a program to calculate the minimum number of steps to win the game.

Input:

Each case on the input file begins with two integers *m* and *n* ($1 \le m, n \le 30$), indicating the size of board. The next *m* lines describe the initial setting of the board. Each line contains *n* characters, either '*' or '.', indicating if the corresponding square is an obstacle or empty, respectively. [For example, see below for the input file corresponding to the case in the figure above.] It is guaranteed that the initial board is not fully covered by obstacles. Process until end-of-file; there is no end of data flag.

Output:

Output the minimum number of steps to cover the board. Follow the format exactly: "`Case`", one space, the case number, a colon and one space, and one integer indicating the minimum number of steps to win the game, or if there is no way to win, the number -1. Do not print any trailing spaces.

Sample Input	Sample Output

`5 5`
`**...`
`.....`
`....*`
`..*..`
`.....`
	`Case 1: 10`

G. Centroid of Point Masses

The "centroid" of a region in two dimensions can be thought of as the point at which the region would balance on the end of a pencil. Computing this would require a bit more effort than we have in mind for this problem, so we restrict our attention to finding the centroid of a collection of point masses.

In this case, we could view the plane as being a thin, massless sheet with a few heavy points on it and ask where the plane would balance. To be more rigorous, we present a mathematical definition of the centroid for this case.

Given the coordinates (x_i, y_i) of a set of n points and the mass m_i at each point, we define the x-moment of that set of points relative to a given point (a, b) as follows (note the x-moment is defined in terms of y differences, but we will need both moments, so it doesn't really matter which way this is done for this particular problem)

$$M_x = \sum_{i=1}^{n} m_i \left(b - y_i\right)$$

Similarly, the y-moment is defined as $M_y = \sum_{i=1}^{n} m_i \left(a - x_i\right)$

The centroid of that set of points is defined to be the point (a, b) for which both moments are zero.

Input:

Input will be sets of points. Each set will be specified by the number of points n in the set followed by n lines of three numbers representing x_i, y_i, and m_i values for i = 1 to n. All these numbers will be integers from 1 to 5000. That is, n will be from 1 to 5000 and all the coordinates and masses will also be from 1 to 5000, just to make input easier. End of input will be marked by a negative value of n. There will be extra white space in input so that judges can read the input cases easily. Do not assume any particular number of spaces before, between, or after the input values, and do not assume a particular number of blank lines between cases.

Output:

Print the coordinates of the centroid. Follow the format exactly: "`Case`", a space, the case number, a colon and one space, and the values of a and b rounded to two decimal places separated by one space. Input will be constructed so that rounding will not cause problems for values that are sufficiently close to correct. Do not print any trailing spaces.

Sample Input	Sample Output

`3`
`1 1 10`
`22 1 10`
`1 31 10`
 	
`3`
`10 10 100`
`20 20 50`
`10 40 30`
 	
`-4`
	`Case 1: 8.00 11.00`
	`Case 2: 12.78 17.78`

H. Strings with Same Letters

A professor assigned a program to his class for which the output is a string of lower-case letters. Unfortunately, he did not specify an ordering of the characters in the string, so he is having difficulty grading student submissions. Because of that, he has requested that ICPC teams help by writing a program that inputs pairs of strings of lower-case letters and determines whether or not the strings have the same letters, though possibly in different orders. Note that repeated letters are important; the string "abc" and "aabbbcccc" are not viewed as having the same letters since the second one has more copies of each letter.

Input:

Input to be processed will be pairs of lines containing nonempty strings of lower-case letters. All input will be valid until the end of file indicator. End of file will be signaled by two lines that contain just the word "END" in upper case. No input line will be longer than 1,000 characters.

Output:

Report whether pairs of strings have the same letters or not. Follow the format exactly: "`Case`", a space, the case number, a colon and one space, and the result given as "`same`" or "`different`" (lower-case, no punctuation). Do not print any trailing spaces.

Sample Input	Sample Output
`testing`	`Case 1: same`
`intestg`	`Case 2: different`
`abc`	`Case 3: same`
`aabbbcccc`	`Case 4: different`
`abcabcbcc`	
`aabbbcccc`	
`abc`	
`xyz`	
`END`	
`END`	