Eighteenth Annual Ohio Wesleyan University Programming Contest

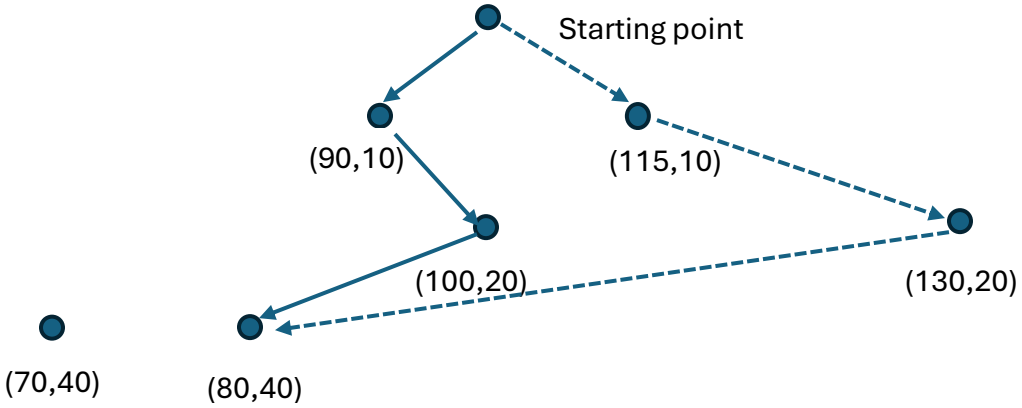
March 28, 2026

Rules:

1. There are six questions to be completed in four hours.
2. All questions require you to read the test data from standard input and write results to standard output. **Do not** use files for input or output. **Do not** include any prompts, other debugging information, or any other output except for exactly what is specified by the problem.
3. The allowed programming languages are C++, Python 3, and Java.
4. All programs will be re-compiled prior to testing with the judges' data.
5. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed.
6. Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required.
7. If multiple input values exist on the same line, they will be separated by a single space.
8. Each line of output case will end with a newline character.
9. All communication with the judges will be handled in the PC^2 environment.
10. Judges' decisions are final. No cheating will be tolerated.

Problem A: Alpine Line

In downhill skiing races, competitors need to navigate through "gates" represented by flags or poles on the course. The racers naturally want to find the path (called the *line*) through these gates that minimizes the total distance traveled.



Above, we see three horizontal gates a racer must pass through, and two possible lines. The solid line is the best line, and has a total distance of 56.57. The dashed line is a longer path, with a total distance of 89.89.

To help our calculations, we will make some simplifying assumptions:

- The skier always starts at position (100,0), and y values will increase as the skier goes down the hill.
- Each gate is horizontal (both endpoints will share the same y value).
- While in a real race, the racer can go anyplace in between the two gate endpoints, for our purposes, the skier will always go to one of the two endpoints of each gate.
- While in a real race, there is some distance between the last gate and the finish line, we only want to calculate the distance to one of the endpoints of the final gate.

Input:

Each input instance will begin with an integer n (n <= 100), the number of gates. A value of 0 will denote the end of the input cases. Otherwise, there will follow n lines of input, one for each gate, each line of the form:

$y\ x_l\ x_r$

y is the y coordinate of the gate (0 <= y <= 1000), and x_l and x_r are the x coordinates of the left and right endpoints of the gate (-1000 <= $x_l < x_r <= 1000$), respectively. The

gates will be given in increasing y value (the order in which the racer will encounter the gates on the way down the hill).

Output:

For each input instance *i*, output:

`Case i: d`

..where *d* is the minimum distance from the starting point through all gates. Your answer will be considered correct if it is within .01 of the judge's answer.

Sample Input:

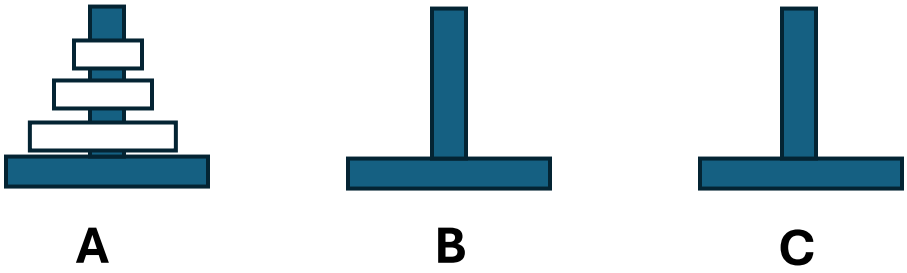
```
3
10 90 115
20 100 130
40 70 80
3
10 100 110
20 120 130
30 200 220
0
```

Sample Output:

```
Case 1: 56.5685
Case 2: 107.213
```

Problem B: Hanoi Snapshot

In the classic "Towers of Hanoi" problem, we are given a set of pegs. One peg has a series of discs on it of decreasing sizes:



The goal is to get all discs from peg A to peg B while observing the following rules:

- Only one disc (the top disc of a stack) can be moved at a time.
- You can never place a larger disc on top of a smaller. You can move any disc from the top of its stack to a position above a larger disc on a different stack, or to an empty peg.

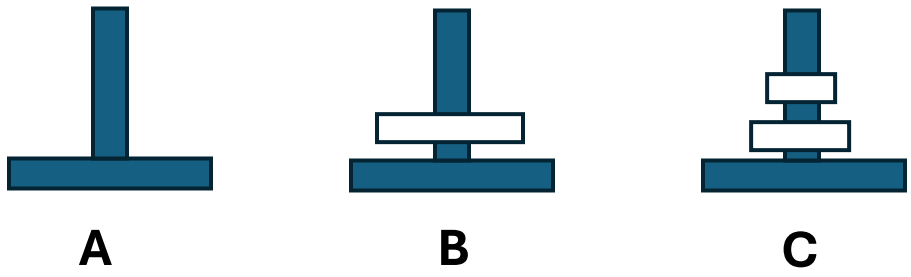
This is a common problem taught when learning recursion, since the problem has a simple recursive algorithm:

To move n discs from peg "source" to peg "dest", with a "spare" peg:

- (Recursively) move n-1 discs from "source" to "spare"
- Move the bottom disc directly from "source" to "dest"
- (Recursively) move the n-1 discs on "spare" over to "dest"

In the starting configuration, the 'source" peg is A, the "dest" peg is B, and the "spare" peg is C, but that will change as recursive calls create subproblems.

It turns out that to move n discs according to this algorithm it takes 2^n-1 individual disc movements. Simulating that many moves is a lot to handle as n gets even into the double digits (with 10 discs, that's 1023 moves, for example). But what we could do is create a *snapshot* of what the algorithm has done up through a certain point. For example, after 4 moves, the 3 disc solution has progressed to this point:



For this problem, all we'll care about is the number of discs on each peg (not their relative sizes) for a given number of starting discs and a number of moves into the solution.

Input:

Each input instance will contain 2 integers *n* (0 <= *n* <= 60) and *s* (0 <= s <= 2^n-1). A value of n=0 denotes the end of the input cases. Notice that *s* may be larger than what a 32-bit integer can hold.

Output:

For each input case `i`, output:

`Case i: a b c`

...where *a*, *b*, and *c* are the number of discs on pegs A, B, and C (as pictured above) after *s* disc moves of the solution.

Sample Input:

```
3 4
30 0
50 1000000000000
0 0
```

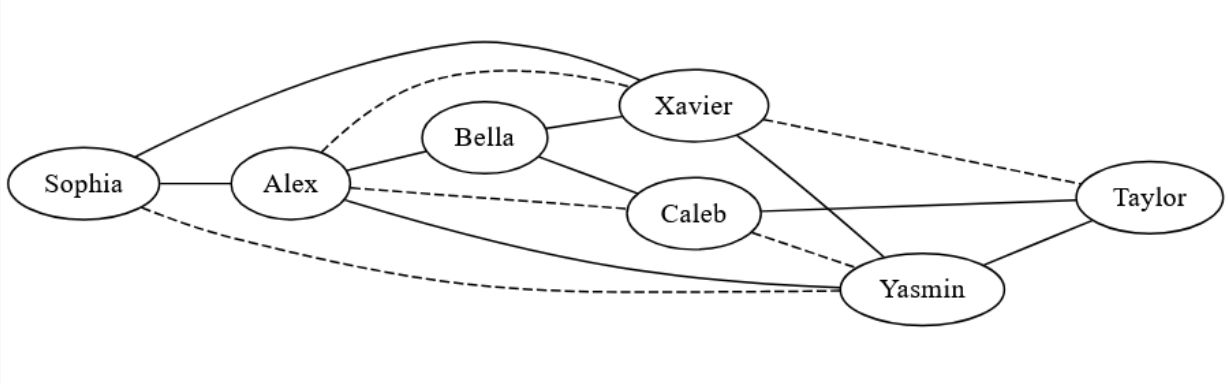
Sample Output:

```
Case 1: 0 1 2
Case 2: 30 0 0
Case 3: 19 24 7
```

Problem C: Message Passing

In the ever-changing world of middle-school friendships, at any given time it's possible for a pair of students to like each other, dislike each other, or be indifferent to each other. This makes it hard for a message to be passed from one person to another. Students will only pass messages to people they like, but every person who comes in contact with the message will read the message, and if anyone who touches the message does not like either the original sender or the ultimate recipient (or both!) they will throw away the message. (This is true even if the ultimate recipient does not like the ultimate sender- we'll say that the recipient refuses the message, so the message isn't considered to be delivered). The student who is sending the message would like to find the shortest chain of passed messages to have it reach the destination, if one exists.

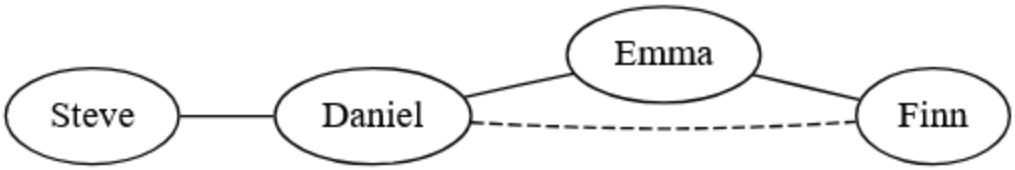
For example, consider the following group of students:



Solid lines indicate pairs of students who like each other, dashed lines indicate pairs of students who dislike each other.

If Sophia wanted to send a message to Taylor, the shortest chain would be Sophia-Alex-Yasmin-Taylor. But since Yasmin doesn't like Sophia, that chain is not feasible. Instead, the chain Sophia-Alex-Bella-Caleb-Taylor will have to be used. (Alex and Caleb don't like each other, but that is ok because neither will know the other is in the message chain. Only the first sender and final receiver is known)

But here is another situation:



If Finn wanted to pass a message to Steve, it won't be possible because the only chain between them of people who like each other goes through Daniel, and Daniel doesn't like Finn.

Input:

There will be several input instances, each will have several messages to pass. Each input instance begins with an integer n, the number of students in the instance (n <= 100). A value of n=0 denotes the end of the input instances.

Otherwise, the next line will contain an integer l, the number of pairs of students who like each other. There will then follow l lines each containing two strings- the names of students who like each other.

There will then follow an integer d, representing the number of pairs of students who dislike each other. The next d lines will contain pairs of names of students who dislike each other.

All names will be single strings without spaces. The lists of pairs of students who like or dislike each other will use at most n different names. Additionally, the relationships are symmetric- if student A likes (or dislikes) student B, then student B also likes (or dislikes) student A.

After the pairs, there will be an integer m, representing the number of messages passed. There will then follow m lines of 2 names each, representing the sender and recipient of each message

Output:

For each group of students i, output:

`Case i: p1 … pm`

..where each p_i is either the length (in terms of number of messages passed) of the shortest feasible path to send message i, or the word "`impossible`" if the message cannot be sent.

Sample Input:

```
7
9
Sophia Alex
Sophia Xavier
Alex Bella
Alex Yasmin
Bella Caleb
Bella Xavier
Xavier Yasmin
Yasmin Taylor
Caleb Taylor
5
Sophia Yasmin
Xavier Taylor
Alex Xavier
Xavier Taylor
Alex Caleb
3
Sophia Taylor
Bella Xavier
Alex Caleb
4
3
Steve Daniel
Daniel Emma
Emma Finn
1
Daniel Finn
1
Finn Steve
0
```

Sample Output:

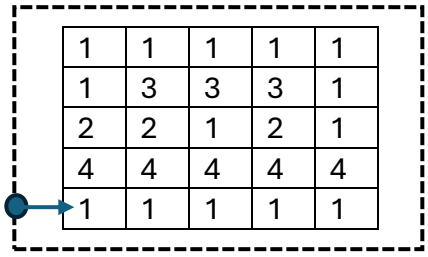
```
Case 1: 4 1 impossible
Case 2: impossible
```

Problem D: Perimeter Shooter

In the puzzle game Perimeter Shooter, the player is given a two-dimensional grid of colored pixels. Then the player can choose a "pixel gun" that comes loaded with a certain amount of "color shots". The gun is placed on a conveyor belt that wraps around the outside of the grid. The gun travels on this conveyor belt, moving clockwise around the grid, starting at the bottom left corner (facing right, pointing at the grid), and then moving around the grid, up the left side, then right across the top (facing down), then down the right side (facing left), then left across the bottom (facing up). The pixel gun stays on the belt outside the grid and always points towards the interior of the grid.

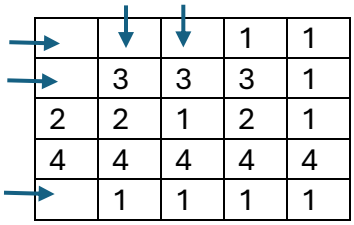
At each position along the conveyer belt, the pixel gun looks straight ahead from its point of view towards the grid in a line until it sees a pixel or the line of sight exits the grid (traveling through only empty spaces). If the line of sight encounters a pixel of the gun's color, the gun shoots the pixel, eliminating the pixel, and using a color shot, otherwise the gun will not shoot. At most one pixel will be removed at each position, after which the pixel gun will move on to the next position. After one lap around the outside of the grid, or when the pixel gun runs out of shots, the pixel gun is removed, and the player can send a new pixel gun around the grid. The new pixel gun may be the same, or a different color, and moves clockwise around the grid again, shooting corresponding-colored pixels that remain in the grid.

For example, suppose this was our grid, using numbers as colors. The dashed line is the conveyer belt around the outside. The solid dot is the pixel gun, and it is initially looking right, towards the cell in the bottom left corner:



1	1	1	1	1
1	3	3	3	1
2	2	1	2	1
4	4	4	4	4
1	1	1	1	1

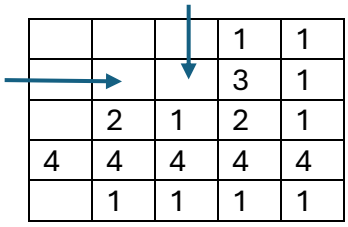
If we sent out a pixel gun with color 1 and 5 shots, it would start at the bottom left, looking towards the bottom left corner, move up while facing towards the left edge, and eliminate the three 1's it saw along the left edge while moving up. It would then turn to move along the top edge, looking down, using its next two shots on the color 1 pixels it sees while moving right. The resulting grid would eliminate five pixels and look like this:



			1	1
	3	3	3	1
2	2	1	2	1
4	4	4	4	4
	1	1	1	1

(the arrows show the directions of shots that are made, they are not part of the grid)

With several pixels removed from the grid, if our next pixel gun was color 3 with 4 shots, the pixel gun now has a clear view at a color 3 pixel on the way up, and another one while moving across the top. It can use two of its 4 shots around its one circuit of the grid:



			1	1
			3	1
	2	1	2	1
4	4	4	4	4
	1	1	1	1

Notice that if we would have sent the pixel gun with color 3 before the color 1 gun went around, it would not be able to see any of the color 3 pixels, so none would be shot.

We are interested in finding out what the state of the grid looks like after several pixel guns have been used in sequence, in terms of the number of pixels left of each color.

Input:

Each input instance will begin with an integer n (<= 100), the number of colors in the grid. A value of n=0 denotes the end of the input cases.

Otherwise, the next line will contain two integers, r and c (1 <= r, c <= 20), the number of rows and columns in the grid. There will then follow r lines of c numbers, separated by spaces. Each number will be between 1 and n, inclusive, and will denote the color of that position of the grid.

There will then follow a positive integer p, the number of pixel guns that will be used (p <= 20). There will then follow p lines of the form:

pc pa

Which represent the color of the pixel gun (1 <= pc <= n) and the number of shots it has (1 <= pa <= 500)

Output:

For each input case i, output:

Case i: nc_1 nc_2 ... nc_n

..where each nc_i represents the number of color *i* remaining in the grid after all pixel guns have been shot, in order.

Sample Input:

```
4
5 5
1 1 1 1 1
1 3 3 3 1
2 2 1 2 1
4 4 4 4 4
1 1 1 1 1
4
1 5
2 3
4 3
1 20
5
3 6
5 4 4 4 4 4
5 1 2 3 2 1
5 4 4 4 4 4
3
3 5
1 5
2 5
0
```

Sample Output:

```
Case 1: 1 2 3 3
Case 2: 1 1 1 10 3
```

Problem E: Phonetic Turbulence

Welcome to the future… a future in which vowels and consonants have trouble coexisting, but are forced into words together anyway. In this scary time, words that have many of both vowels and consonants are tense and unstable. We can measure the phonetic turbulence of a word as the number of consonants multiplied by the number of vowels. For our purposes, vowels are 'A', 'E', 'I', 'O', and 'U'. Every other letter is a consonant. Except 'Y'. "Sometimes Y" isn't trusted by either side, and is neither counted as a vowel nor a consonant.

As a member of the Word Patrol, your job is to measure the phonetic turbulence of words you come across, and report ones that fall within given turbulent ranges. The future of the dictionary is up to you!

Input:

There are several input instances. Each input instance begins with a line containing 3 numbers: n, the number of words (<= 100), and l and u (0 <= l <= u <=5000), the lower and upper bounds of the turbulence values we are interested in looking at. A value of n=0 denotes the end of the input cases.

Otherwise, the next line will consist of n strings, separated by spaces. Each word will consist of at most 100 capital letters.

Output:

For each input case i, output:

`Case i:`
$Word_1$ t_i
…
$Word_n$ t_n

Where each "$Word_i$" is a word from the input that has a phonetic turbulence >= l and <= u, and each "t_i" is its turbulence, one pair per line. Output the words in increasing order of turbulence. If two words have the exact same turbulence, output them in alphabetical (lexicographic) order. Note that if no words fall within this range, you should have the "`Case i`" line, but no words should be output.

It would be nice if you separate the output cases with a blank line, but PC² ignores all whitespace, so it will accept your solution if you don't.

Sample Input:

```
3 1 1000
ROUND ONE FIGHT
6 1 1000
THIS IS A BUNCH OF WORDS
5 5 50
AAAAYBBBB XYZXYZABCAB YYYYY AAAA BBBBO
0 0 0
```

Sample Output:

```
Case 1:
ONE 2
FIGHT 4
ROUND 6

Case 2:
IS 1
OF 1
THIS 3
BUNCH 4
WORDS 4

Case 3:
XYZXYZABCAB 14
AAAAYBBBB 16
```

Problem F: Rolling with Super (dis-)advantage

In some games, especially role-playing games like *Dungeons and Dragons*, the outcome of certain decisions is made by a die roll. In these games, the higher you roll, the better. Sometimes, due to factors in the game, you could be able to roll with advantage: where you roll two dice and take the higher. Other times, if bad things happen, you might be forced to roll with disadvantage: where you roll two dice and take the lower.

In the official *Dungeons and Dragons* rules, you can only ever roll two dice at most- for example, getting advantage 10 times is the same as getting advantage once. But I think it would be fun to roll with super advantage (or disadvantage): If you qualify for advantage or disadvantage several times then you roll several dice, and take either the highest of all of them (with super advantage) or the lowest of all of them (with super disadvantage).

Input:

There will be several input instances. Each will begin with a line containing a number n (<= 100) representing the number of dice to roll, and a character t (which will be either 'a' or 'd', signifying if we are rolling with advantage or disadvantage). A value of n=0 will denote the end of the input cases. Otherwise, the next line will have n positive integers (each roll >= 1 and <=100,000), representing the rolls made.

Output:

For each input case *i*, output:

`Case i: v`

..where *v* is the highest of all rolls (if we were rolling with advantage) or the lowest of all rolls (if we were rolling with disadvantage)

Sample Input:

```
6 a
19 7 16 2 20 1
6 d
19 7 16 2 20 1
5 d
100 200 300 400 500
4 a
99 50 2 77
0 a
```

Sample Output:

```
Case 1: 20
Case 2: 1
Case 3: 100
Case 4: 99
```