Thirteenth Annual Ohio Wesleyan University Programming Contest

November 5, 2022

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, C++, Python2 and 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. All output cases will end in a newline character. So after your program finishes output, the cursor should be on the first column of the line immediately below your last case's output.
8. All communication with the judges will be handled in the PC^2 environment.
9. Judges' decisions are final. No cheating will be tolerated.

Problem A: Crafting Resources

In many role-playing video games, you're given a crafting system by which you can make new items out of components you find in the wild. Sometimes the things you craft are themselves components for more powerful items.

Suppose you find yourself wanting to make the "Killbeast Awesomesword". The recipe says that it's made out of:

- 1 Mediocre Sword
- 2 Awesome Charms

The Mediocre Sword is made from:

- 1 Sad Sword
- 3 Iron Ore

(all of which are collectable in the wild)

But each Awesome Charm is made from:

- 2 Boring Charms
- 3 Glowing Fireflies
- 4 Special Sauce.
  (all of which are collectible in the wild)

So, if you're sitting on 8 glowing fireflies, 2 iron ores and one special sauce, what you'd need to collect to make the Killbeast Awesomesword is:

- 1 Sad Sword
- 1 Iron Ore
- 4 Boring Charms
- 7 Special Sauce

Your task is to create this list of required basic resources, given a set of recipes and a desired object to make.

Input:

Each input case begins with an integer *n* (<= 50), the number of objects that exist. A value of *n*=0 denotes end of input.

There will then be *n* lines, one for each item. Each line contains a string s_i (the name of object i) and an integer q_i (<=50), the amount of that object you currently possess. The names of the objects will be given in alphabetical order, and the strings will not have spaces in them.

Then there will follow *n* lines, consisting of the recipes for how to make each object. Each line will begin with an integer *c* (<=10): the number of components needed to make the item. A value of *c*=0 means that the item is found in the wild, a value of *c* > 0 means it must be crafted.

If *c* > 0, the line will continue with c pairs of integers and strings. The integer is the number of the component needed (<= 50), the string is the name of the component (which will be one of the objects listed previously).

The last line of the input case will be a string from the *n* object strings, denoting the desired object to create.

Every item will be created by at most one recipe. Items may appear as components in multiple recipes.

Output:

For each input case it. Output:

```
Case i:
Obj₁ Quant₁
…
Objₖ Quantₖ
```

..where each "Obj$_i$" is the name of a resource found in the wild. List out the resource names in alphabetical order, and only list out resources for which the quantity needed is greater than 0.

Sample Input:

```
8
Awesome_Charm 0
Boring_Charm 0
Glowing_Firefly 8
Iron_Ore 2
Killbeast_Awesomesword 0
Mediocre_Sword 0
Sad_Sword 0
Special_Sauce 1
3 2 Boring_Charm 3 Glowing_Firefly 4 Special_Sauce
0
0
0
2 1 Mediocre_Sword 2 Awesome_Charm
2 1 Sad_Sword 2 Awesome_Charm
0
0
Killbeast_Awesomesword
0
```

Sample Output:

```
Case 1:
Boring_Charm 8
Glowing_Firefly 4
Sad_Sword 1
Special_Sauce 15
```

Problem B: Fabric Stripes

Fran has lots of small, long strips of leftover fabric from her last quilting project. She would like to sew them together into a piece that has stripes from each fabric. She would like to see how many ways she can combine the fabrics of different lengths to make different sized stripes. For example, if she wants to make a 3 inch piece out of 3 different fabrics, she could have all 3 stripes be 1 inch wide, or one stripe be 2 inches wide, and the other stripes be .5 inches wide each and so on. The fabrics will all be placed in the same order in the square- Fran just wants to consider width variations. The variations of widths are limited

There is a lower limit to how small a stripe can be before it's too small to be able to be worked on. There is also a limit on the amount by with the fabric widths can vary- Fran can't measure things to all possible fractions of an inch. Given the constraints of minimum stripe size, how much the stripe size can vary, fabrics to use, and desired width of the final piece, Fran would like to know how many different permutations of stripes there are. (So different orderings of the same fabric widths should count separately).

Input:

Each input instance begins with a positive integer *N* (<=100), the number of pieces of fabric to be used. A value of *N*=0 denotes the end of input.

Otherwise, there will follow two numbers *W*, *M*, and F. *W* (1 <= *W* <=1000) is the desired width of the fabric. *M* (.25 <= *M* <= 10) is the minimum width of each stripe, and *F* (.25 <= *F* <= 10) is the factor by which the sizes can change. (So, if M=1, and F = .5, we can only consider widths of 1, 1.5, 2, 2.5, and so on).

W, *M*, and *F* can all be decimals, but will always be multiples of .25.

The values of the input parameters will be set so that it is always possible to make a legal set of fabric (So, for example, it won't be the case that *N** *M* > *W*), and the number of permutations will always fit into a 32-bit integer.

For example, the first input case describes a project where Fran wants to use 5 stripes to make a 6-inch quilt. The minimum width is 1 inch, and the stripe widths can vary by multiples of 1 inch. In this case, there are 5 possible outcomes- each of the five stripes could be 2 inches long, while the rest are 1 inch long. Since we have to use all 5 stripes, no single stripe could be 3 inches long.

The second input case changes the allowable stripe sizes to be multiples of ½ an inch. Now, in addition to the 5 patterns with 4 one-inch stripes and a two-inch stripes, there are 10 additional patterns where 2 stripes are 1.5 inches long, and the rest are 1 inch long. No stripe can be .5 inches, because the minimum width is 1.

Output:

For each input case I, output:

```
Case i: P
```

..where “P” is the number of permutations of the fabric.

Sample Input:

```
5
6 1 1
5
6 1 .5
10
20 1.5 .5
0
```

Sample Output:

```
Case 1: 5
Case 2: 15
Case 3: 92378
```

Problem C: GCD Alternative

A programming contest a few years ago asked students, among other things, to reduce a fraction to lowest terms. The classic way to do this is to find the greatest common divisor of both numbers, using, for example, Euclid's algorithm:

```
GCD(a,b)
   If b is 0
      Return a
   Else
     Let c = remainder of a/b
     Return Gcd(b,c);
```

..and then dividing both the numerator and denominator by the greatest common divisor.

What we saw from several teams instead was an iterative approach, trying all possible factors starting from 2 going upwards, and dividing both numbers by any factors that go into both the numerator ("num") and denominator ("den") evenly:

```
Factor = 2;
While(factor <= num)
   If num/factor and den/factor both have remainders of 0
      Divide both num and den by factor.
   Increase factor by 1
```

This actually works surprisingly often (surprising, at least, to those of us who saw it). For example, if we are reducing the fraction 90/150 we would first consider the factor of 2 (creating 45/75), then a factor of 3 (creating 15/25), then 5 (creating 3/5). The GCD method, on the other hand, finds that the greatest common divisor of 90 and 150 is 30, and divides both 90 and 150 by that, also creating 3/5.

The alternate method doesn't always work, though. For example, if we are reducing 4/8, we would first consider a factor of 2 (creating 2/4), but then we would need another factor of 2, but the factor counter has moved on to 3, and no larger factors will be considered.

Your task is to determine, for a given numerator and denominator, whether this alternative method gives the correct answer or not.

Input:

There will be several input instances. Each input instance will contain one line with two non-negative integers n and d (both <= 100,000), the numerator and denominator of the fraction to reduce. A value of n=d=0 denotes end of input.

Output:

For each case i, output either:

```
Case i: YES
```

(if the alternate method does reduce the fraction correctly)

Or

```
Case i: NO
```

(if it doesn’t)

Sample Input:

```
90 150
4 8
0 0
```

Sample Output:

```
Case 1: YES
Case 2: NO
```

Problem D: Quasi-Palindromic Dates

I've been holding this question since before the pandemic. Think back to those carefree days of early 2020..

On February 6, my wife tells me "Hey, today's date is a palindrome!"

I respond: "No it isn't: 02/06/2020. Even without the slashes, 02062020 backwards is 02026020- the 6's don't match up."

She says: "No: 02/6/20"

I respond: "Who puts the zero in front of the month but not the date? That's not a way anyone does anything!"

Yes, these are the types of conversations we have at my house.

Anyway, it got me thinking. If we start with a date of the form MM/DD/YYYY, there are really 8 ways to check if it's a palindrome (after removing the slashes):

1. The MM/DD/YYYY format itself, with no changes
2. Removing just the first digit of the month (and only if it is a 0) to create a M/DD/YYYY format
3. Removing just the first digit of the day (and only if it is a 0) to create a MM/D/YYYY format.
4. Removing the first 2 digits of the year to create a MM/DD/YYYY format (we can do this no matter what the digits we remove are).
5. Combining methods 2 and 3 to create a M/D/YYYY format (only if the digits we remove are both 0)
6. Combining methods 2 and 4 to create a M/DD/YY format (only if the first digit of the month is a 0)
7. Combining methods 3 and 4 to create a MM/D/YY format (only if the first digit of the day is a 0)
8. Combining all of methods 2, 3, and 4 to create a M/D/YY format (only if the first digit of both the day and the month are 0).

Your task is to take a date written in MM/DD/YYYY format and output which of the rules above create a palindrome.

Input:

There will be several input instances. Each input case will be a string of the form MM/DD/YYYY where each M, D, and Y character is between 0 and 9 (The input may not necessarily form a legal date, see sample input), or the string "end", denoting the end of the input cases.

Output:

For each input case I, output:

```
Case i: m1 m2 …
```

..where each mi is a number between 1 and 8 above. Output a number if and only if applying the method corresponding to the number creates a palindromic date (after removing the slashes). Output the numbers in increasing order, separated by spaces. If no methods create palindromes, output "None"

Sample Input:

```
02/06/2020
02/02/2020
02/30/1932
77/77/7777
01/23/4567
end
```

Sample Output:

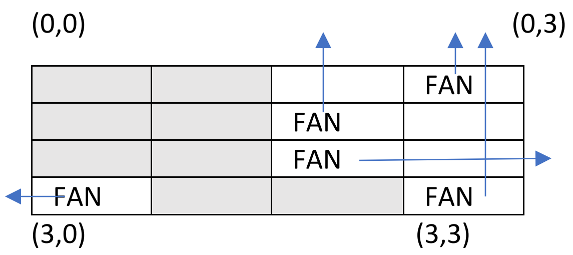
```
Case 1: 7
Case 2: 1 7
Case 3: 6
Case 4: 1 4
Case 5: None
```

Problem E: That Breath of the Wild Windmill Puzzle

<u>NOTE: THIS PROBLEM HAS A 30 SECOND TIME LIMIT INSTEAD OF THE USUAL 10 SECONDS</u>

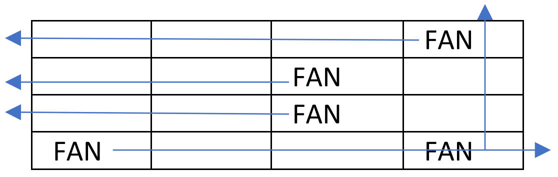
One of the puzzles in the video game "Legend of Zelda: Breath of the Wild" places you near a rectangular grid of blocks. Some of the blocks have windmills mounted on them, which spin when air is blown through them. The remaining blocks have fans, which blow wind in a single direction (up, down, left, or right). A fan will blow wind in a straight line, spinning each windmill in that line, no matter how far away, and no matter what other objects are in between. A windmill will spin if one or more fans are pointed at it.

Here is a diagram of a possible board configuration (any square that is not a fan is a windmill). The arrows point the way the fans are blowing at the start of the sample input. The grey squares show fans that are not spinning.



The goal of the puzzle is to make all of the windmills spin simultaneously by turning the fans. We'd like to do it in the least amount of moves possible. A "move" turns a block's fan ninety degrees in either direction.

A solution to the above puzzle in 6 moves would give us:



Input:

Each input case will begin with two integers *r* and *c* (<10), the number of rows and columns of the grid. A value of *r*=*c*=0 denotes end of input.

The next line will contain an integer *n* (1 <= *n* < 10), the number of fans.

On the next *n* lines, there will be three integers r_i c_i d_i. r_i and c_i are integers denoting the row and column of the fan on the grid (positions start from 0). d_i will be one of the 4 characters {U,D,L,R}, signifying whether the fan is initially blowing air Up, Down, Left, or Right.

Output:

For each input case I, output:

```
Case i: k
```

..where k is the minimum number of turns of fans needed to make wind blow on all windmills.

If it is not possible to make the wind blow on all windmills, output:

```
Case i: Impossible
```

Sample Input:

```
4 4
5
0 3 U
1 2 U
2 2 R
3 0 L
3 3 U
4 4
4
0 3 U
1 2 U
2 2 R
3 0 L
0 0
```

Sample Output:

```
Case 1: 6
Case 2: Impossible
```

Problem F: White Elephant

In a “white elephant” gift exchange, each person in the exchange brings a wrapped present. The people in the exchange get assigned an order.

On each person’s turn, they have two choices:

1. Open a new present
2. Steal a gift from someone else. It then becomes that person’s turn.

Once a gift is stolen, it cannot be stolen again until a present is opened (this prevents infinite loops). Once the last gift is opened, the exchange ends.

Suppose we’re interested in predicting the outcome of an upcoming exchange. Though various social engineering techniques, we’ve learned what all of the gifts will be, and assigned a numerical “desirability” value for how much each person likes each gift (higher numbers are more desirable). We can then simulate the exchange by assuming that each person will trade for their most desirable gift they can get, but also that there is a certain “threshold” value that they will not go below, and will take a chance on opening the next gift. (Just because we know what the gifts are doesn’t mean that they do).

If a person has a choice to take two presents of the same desirability, they choose the lowest numbered gift.

For example, suppose we have the following table of desirabilities for persons A, B, C, and D and gifts 1,2,3, and 4:

	Gift1	Gift2	Gift3	Gift4
A	10	4	3	4
B	3	8	3	2
C	1	2	3	1
D	9	5	3	2

Suppose that at some point in the middle of the exchange, person A has Gift 1, person B has gift 2, and person C has gift 3. The “minimum threshold” value is 4. Then the following sequence occurs:

- Person D steals Gift 1 from person A
- Person A can’t steal Gift 1 back, so takes the next most desirable gift, Gift 2, from person B (the value of 4 is equal to, but not less than, the threshold value, so they choose to steal instead of opening a new gift)
- Person B can only steal Gift 3, but that is below the threshold value, and so will open the next gift (Gift 4).

Your task is to simulate the entire exchange and determine who ends up with what gift.

Input:

Each input case begins with an integer *n* (1 <= *n* <= 20) the number of people (and presents). A value of *n*=0 denotes end of input.

There will then follow n rows of n integers. The first row is Person A's desirability for each of the n presents, the second row is Person B's desirability for each present, and so on. Each score will be an integer between 1 and 100, higher scores are more desirable.

The input concludes with the threshold value t, also between 1 and 100.

Output:

For each input case *i*, output:

`Case i:` p_a p_b … p_n

..where p_a is the number of the present Person A ends up with after the entire exchange (starting from 1), p_b is the number of the present Person B ends up with after the entire exchange, and so on.

Sample Input:

```
4
10 4 3 4
3 8 3 2
1 2 3 1
9 5 3 2
4
0
```

Sample Output:

```
Case 1: 2 4 3 1
```