Instructions

Write one program per task. Before writing a program, make a flowchart for the control flow (this can be abstract - you don't have to mention every little detail). You may complete the tasks in any order you wish, although it will probably be easiest to follow the numerical ordering. **Make sure to test out your programs on a variety of inputs to ensure they are working correctly!** Once you are satisfied that your code works correctly, make sure you've followed all the style guidelines, and show your instructor. Feel free to work together, but each student should write their own code!

Flowcharts

Before writing a program for each task, it will be helpful to think about the steps that will be needed to complete the task (the algorithm), and draw a flowchart representing these steps.

Task 1: Factorials

The *factorial* of a number is the product of all the positive integers less than or equal to the number. For example, the factorial of 5 is 5*4*3*2*1 == 120, and the factorial of 9 is 9*8*7*6*5*4*3*2*1 == 362880. For n>=1, the factorial of n is n*(n-1)*(n-2)*...*1. The factorial of 0 is special: it's 1.

Write a program to get an integer as input from the user, then print out the factorial of that integer. If the user enters a negative number, an appropriate error message should be printed instead.

Task 2: Primes

A *prime number* is a positive integer with exactly two factors: 1 and itself. If a positive integer has more than two factors, it is known as *composite*. 1 is neither prime nor composite.

Write a program that takes an integer as input from the user, then (correctly) prints one of the following messages (replacing <the number> with the actual number the user entered):

- "<the number> is prime."
- "<the number> is composite."
- "1 is special - it is neither prime nor composite."
- "ERROR! Negative number entered. Please try running the program again."

To find if a number is prime, you can simply write a loop to check divisibility for every integer less than the input number. (Extra credit possible for more efficient approaches, with explanatory comments).

Task 3: n Bottles of <beverage> on the Wall

The classic song starts like this:

99 bottles of <beverage> on the wall, 99 bottles of <beverage>!
Knock one down, pass it around, 98 bottles of <beverage> on the wall.

It goes on for 99 verses, until all the bottles have been knocked off the wall, with the last verse being:

1 bottle of <beverage> on the wall, 1 bottle of <beverage>!
Knock it down, pass it around, no more bottles of <beverage> on the wall.

You will write a program to "sing" a generalization of this song. The program should prompt the user to enter a number, and then a beverage. Then, the program should print out the entered number of verses of the song, with the appropriate beverage name slotted in place of <beverage>. A blank line should be printed between each verse. The last two verses will need to be handled specially: your program should say "1 bottle" instead of "1 bottles", and the last verse should match the one above (with "no more bottles" in place of "0 bottles", and "Knock it down" in place of "Knock one down").

Task 4: 12 Days of Christmas

Write a program to "sing" the 12 Days of Christmas song, with each gift appearing on a new line, and a blank line in between each verse. Example of output through verse 3:

On the 1 day of Christmas, my true love gave to me:
A partridge in a pear tree.

On the 2 day of Christmas, my true love gave to me:
Two turtle doves, and
A partridge in a pear tree.

On the 3 day of Christmas, my true love gave to me:
Three French hens,
Two turtle doves, and
A partridge in a pear tree.

If you don't know the rest of the words, look them up online!

Use a loop to accomplish this. The loop will keep track of the verse number; inside should be a sequence of nested if statements printing out the gifts.

Task 5: Binary

Write a program to take in an integer n from the user, then print out that number in 8 bits of unsigned binary (so it can print anything from 0b00000000 to 0b11111111, which in base 10 is 0 to 255). We will do this via the following procedure:

1. Prompt the user to enter a number, store that number as a variable called n.
2. Make a new integer variable called temp. Start it off at the value of n.
3. If temp is bigger than 255, first set it to temp modulo 256 (use the % operator) to get it into the appropriate range.
4. If temp is negative, first find temp modulo 256, then add 256 (this is because % gives a negative result if the number is negative).
5. Once we have temp between 0 and 255, make a new String variable and set it to have the value "0b". For each power of 2 from 128 down to 1: {If the power of 2 is less than temp, add "1" to the String, and subtract the power of 2 from temp. Else, add "0" to the String.}
6. Print out a message "The number <n> is equal to <binary version of n> in binary.", replacing the <> parts with the appropriate values (this is why we made a new variable temp instead of just using n, so we could remember the original value of n).

Task 6: Times Tables

Write a program to take in a positive integer n from the user, then print out an n by n grid of multiplication facts with at least 2 spaces between each number in a given row. For example, if the user enters the number 5, then the following should be printed out:

```
 1  2  3  4  5
 2  4  6  8  10
 3  6  9  12  15
 4  8  12  16  20
 5  10  15  20  25
```

You will likely want to use two nested loops.

Extra credit: Get it to print out in the following nicely aligned and labeled format:

```
    1   2   3   4   5
   --------------------
1 | 1   2   3   4   5
2 | 2   4   6   8   10
3 | 3   6   9   12  15
4 | 4   8   12  16  20
5 | 5   10  15  20  25
```

Hint: To figure out how much space to add, first find out how many digits d are in the largest product, n*n (there are several ways to do this, such as a loop checking if it's less than successive powers of 10, or just use of .length() after conversion to a String). Then, the amount of space to put after a number with x digits is simply $(d - x + 1)$.