Function Name: `badApples`

Inputs:

1. *(double)* The number of red apples
2. *(double)* The number of green apples
3. *(double)* The number of good red apples
4. *(double)* The number of good green apples

Outputs:

1. *(double)* The probability of randomly picking a bad red apple
2. *(double)* The probability of randomly picking a bad green apple

Function Description:

This function should calculate the probability of randomly picking a bad red or green apple from a bag. The number of red and green apples will be given in the first two inputs and the number of good red and green apples will be given as the 3rd and 4th inputs. The output value should be expressed as a percentage, not a decimal. So if there was a 25% chance of drawing a bad red apple, this would be expressed as the integer 25 rather than the decimal 0.25.
Round all percentages to the nearest hundredth of a percent.

Notes:

- The number of good apples will always be less than or equal to the number of apples of that color.

Function Name: `fib`

Inputs:

1. *(double)* Which Fibonacci number to calculate

Outputs:

1. *(double)* The specified Fibonacci number

Function Description:

You are probably familiar with the Fibonacci sequence: the sequence of numbers (starting with 0, 1) where every number is the sum of the previous 2 numbers. From this definition, you might think in order to calculate the n^{th} Fibonacci number, you would have to first calculate all the numbers that came before it. However, this is not the case! The formula for any Fibonacci number is:

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

φ is the golden ratio:

$$\varphi = \frac{1+\sqrt{5}}{2}$$

To calculate any Fibonacci number, all you have to do is plug ‘n’ into this equation. This formula is mathematically exact (no rounding is necessary in theory) but because of MATLAB’s internal rounding of large numbers, you should round your answer to the nearest integer.

Function Name: `spherePacking`

Inputs:

1. *(double)* The sphere radius
2. *(double)* The length of the side of a cube

Outputs:

1. *(double)* The number of spheres that will fit into the cube

Function Description:

The problem of sphere packing (how many spheres will fit into a given volume) is actually a non-trivial problem in mathematics and was only proven in the 1990's. If you would like to read more about the problem, you can follow these links:

http://mathworld.wolfram.com/SpherePacking.html
https://en.wikipedia.org/wiki/Sphere_packing

Otherwise, you are welcome to take for granted that the maximum packing density of spheres in a cube is 74.048%. That means that under the best packing scenario, the spheres will occupy 74.048% of the cube's volume. Given this property and the radius of a sphere and the edge length of a cube, calculate the maximum number of spheres that will fit in the cube.

Notes:

- You cannot have fractional spheres.

Function Name: `clockHands`

Inputs:

1. *(double)* The current position of the hour hand
2. *(double)* The current position of the minute hand
3. *(double)* A positive or negative number of minutes

Outputs:

1. *(double)* The position of the hour hand after the specified time
2. *(double)* The position of the minute hand after the specified time

Function Description:

It is not always immediately obvious where the hands of a clock will be after a certain amount of time. It is even harder to visualize where the hands *were* some amount of time in the past. Luckily, this is not a very difficult problem for a computer to solve, so you will use that to your advantage. This function will take in the current position of the hour hand, as an integer between 0 and 11, inclusive (0 for noon/midnight) and the current position of the minute hand, as an integer between 0 and 59, inclusive (0 for “on-the-hour”) and a positive or negative number of minutes elapsed. Given this information, calculate the new position of the clock hands. You should assume that the hour hand does not move until the next hour has begun. For example, the hour hand stays on “2” from 2:00 until 2:59 and only at 3:00 does the hour hand move to “3”.

Notes:

- The `mod()` and `floor()` functions will be useful.
- As you do this problem notice the behavior of `mod()` for negative inputs. This is a very important function in programming and will come up again in the class!

Hints:

- One way of solving this problem involves calculating the total number of minutes after noon/midnight before and after the given minutes have elapsed.
- Another way of solving it involves splitting the given number of minutes into a number of hours and a number of minutes.
- Pick whichever method makes more sense to you (or come up with your own method).