Exercise 1)

What is the Big-Oh of the following computation?

```
int sum = 0;
for (int counter = n; counter > 0; counter = counter - 2)
    sum = sum + counter;
```

It seems like this is a for loop that goes through every other number in a set of numbers ‘n’ amount of times and adds it to the ‘sum’ integer. The notation of this would be **O(n)** because the time it takes to complete this operation is directly correlated to how large ‘n’ is.
I think about it like this: # of times to run = size of n/2 which you could basically turn that into a formula “y = x/2” and charted that, it would be a linear graph. Hence the answer is **O(n).**

Exercise 2)

Suppose your implementation of a particular algorithm appears in Java as follows:

```
for (int pass = 1; pass <= n; pass++)
{
        for(int index  = 0; index < n; index++)
        {
                for(int count = 1; count < 10; count++)
                {
                        . . .

                } //end for
        } // end for
} //end for
```

The algorithm involves an array of "n" items. The previous code shows only the repetition in the algorithm, but it does not show the computations that occur within the loops.Those computations, however, are independent of "n." What is the order of the algorithm?

The first, outermost for loop runs from 1 to an ‘n’ number of times. The second, middle loop runs the same number of times despite starting at 0 because it uses a ‘< n’ instead of ‘<=n’. The third innermost loop runs 9 times. For this entire program to complete, the outermost loop has to run ‘n’ times. However, that outer loop going through one iteration means that the middle loop has also run ‘n’ times making an exponential effect. The third loop can basically be ignored for the purposes of measuring the time complexity because the amount of times it runs is set and not dependent on ‘n’. This program can basically be: n * n * 9.
The portion “n*n” means it’s exponential and the answer would be **O(n^2).**

Exercise 3)

Consider two programs, A and B. Program A requires 1000 x n^2 operations and Program B requires 2^n operations. For which values of n will Program A execute faster than Program B?

The best way to solve this, in my opinion, is to graph this out and see where they intercept and pick that point. I did this on Wolfram alpha and it provided the answer as **n < 18.36.**
The 'n' would have to be 18 or less in order for Program A to execute faster.

I did attempt to come at this problem algebraically, but because of the way 'n' is dispersed as exponents in both operation equations, I'm not actually sure if it is possible to isolate the n.

And yes… I am aware of the "brute force" way. Honestly, I would rather not when the graphing option is available.

Exercise 4)

Consider an array of length "n" containing unique integers in random order and in the range 1 to n + 1. For example an array of length 5 would contain 5 unique integers selected randomly from the integers 1 through 6. Thus the array might contain 3 6 5 1 4. Of the integers 1 through 6, notice that 2 was not selected and is not in the array. Write Java code that finds the integer that does not appear in such an array. Explain the Big-Oh in your code.

As for its Big Oh notation, this would simply be another **O(n).** This essentially for the exact same reasoning as in exercise 1. The calculations of finding the array length, expected sum and actual sum are all in one-line calculations not adding to the time complexity. However, the for loop is what scales the time complexity up to be equal to the 'n' value.

A screenshot of the full code and its output is on the next page. The java file will also be submitted with this assignment.

```java
public class FindMissingNumber {
    //This function will return an int that is the number missing from the sequential numbers.
    public static int missingNumber(int[] numberPool){

        //Finding the length of the array passed into this function.
        int arrayLength = numberPool.length;

        //Go through each number in the array and add it up.
        int arraySum = 0;
        for(int i = 0; i < numberPool.length; i++){
            arraySum += numberPool[i];
        }

        //Assuming all the numbers in the array were sequential, this is what the sum WOULD be.
        int sequentialSum = (arrayLength+1)*(arrayLength+2)/2;

        //The missing number would be the expected sum if they were all sequential subtracted by the ACTUAL arrays sum.
        int missingNumber = sequentialSum - arraySum;

        //Returning that missing number we found.
        return missingNumber;
    }

    //This is the main entry point, the main function.
    public static void main (String[] args){
        //Making an example array based on the example provided from the assignment.
        int[] numbers = {3, 6, 5, 1, 4};

        //Printing that missing number to the console log. (Should print 2.)
        System.out.println("Missing Number: " + missingNumber(numbers));

        //For testing purposes, I made my own array. (Missing 5).
        int[] secondArray = {1, 2, 4, 3, 6};

        //Printing that missing number to the console log. (Should print 5.)
        System.out.println("Missing Number: " + missingNumber(secondArray));
    }
}
```

```
Missing Number: 2
Missing Number: 5
[Finished in 666ms]
```