Math Review and Algorithm Analysis

Use empirical analysis methods and code analysis methods to determine running time complexity in Big O notation.

Review of Common Math Functions.

1) Use Excel or some other graphing tool to graph the following equations.

$$y = x$$
$$y = 2x$$
$$y = x^2$$
$$y = 2^x$$
$$y = x^3$$
$$y = \log_2 x$$

2) Rank the graphs of the above equations by rate of growth, fastest (non-initial) growth first.

3) Match the shape of each graph with the closest common Big(O) curve and label them so.

Empirical Analysis.

4) Complete the table for each of the following functions. For each foo, write a small program with a loop where n is a counter from 0 to at least 64. Call the foo within the loop, passing it each value of n, and getting the return value from foo. Fill out a table with each n and its corresponding return value. You can skip some values of n when n starts to get biggish. Capture your output and generate the tables.

```
int foo1(int n)
{
   int counter = 0;

   for(int i = 0; i < n; i++)
      counter++;

   return counter;
}
```

n	return value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
32	
64	

```
int foo2(int n)
{
   int counter = 0;

   for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
         counter++;

   return counter;
}

int foo3(int n)
{
   int counter = 0;
```

```
   for(int i = n; i > 0; i = i/2)
      counter++;


   return counter;
}

// note:  you might not make it much past n = 32
int foo4(int n)
{
    static int counter = 0;
    counter++;

    if(n > 0)
    {
        foo4(n-1);
        foo4(n-1);
    }

    return counter;
}
```

5) Use Excel to GRAPH the data tables from the previous functions. Use the return value as a function of n. That means, put n on the horizontal axis (x) and put the return value on the vertical axis (y). Use Excel or some other graphics tewl.

6) Rank the graphs above by rate of growth, fastest first.

Code Analysis

7) ***Implement*** and test each of the following series for several different values of n and A. Present your output in a nice table of values. Use ***iterative solutions***, do not use the equivalent (condensed) algebraic formula.

I. Arithmetic Series

Test for values of n from 1 to 10.

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + \ldots + N$$

Iterative code solution:

```
int sum = 0;
for(int i = 1; i <= n; i++)
       sum = sum + i;
```

II. Geometric Series

Test for values of n from 1 to 5 and A from 1 to 5 (25 rows total).

$$\sum_{i=1}^{N} A^i = A^1 + A^2 + A^3 + \ldots + A^N$$

Iterative code solution:

```
int term, sum = 0;
for(int i = 1; i <= n; i++)
{
       term = A;
```

```
        for(int j = 1; j < i; j++)
              term = term * A;
        sum = sum + term;
}

return sum;
```

III. A More Efficient Geometric Series

Redo the previous solution using only a single loop instead of the nested loop.

IV. Another Series

Implement the following series and test for n = 1 to 5 and A = 1 to 5 (25 rows total). Produce a nice table of values.

$$\sum_{i=1}^{N} iA^{i} = 1A^{1} + 2A^{2} + 3A^{3} + \ldots + NA^{N}$$

8) For the previous series implementations, determine the BigO of each series by analysis of the source code. A code analysis using our Big O "rules of thumb" is sufficient; you do not need to perform an exact mathematical analysis. List the Big O answer and explain why it is so in English.