Discussion 2

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$\log n^n,\ n^2,\ n^{\log n},\ n \log\log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

Solution: First separate functions into logarithmic, polynomial, and exponential
Logarithmic: $\log^2 n$
Exponential: $n^{\log n}$
Polynomial: everything else

So we just need to order the ones that are polynomial now:
Since $2^{\log n} = n$ (assuming log base 2), and $\log n^n = n \log n$, we will have the following order:

$$n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2$$

Now we put the whole list in this order: Logarithmic, polynomial, exponential which gives us:

$$\log^2 n\ ,\ n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2\ ,\ n^{\log n}$$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n)).* Is it true that $2^{f(n)} = O(2^{g(n)})$?

Solution: Not true. Will not work for f(n) = 2n and g(n) = n

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

Solution: The call to find_max takes linear time with respect to n, and we know that the while loop terminates after log n (base 4) iterations. So, it is easy to say that this code runs in O(n log n). But if you look closer and add up the cost of the calling find_max at every iteration, we will get:

1st call:	cn
2nd call:	cn/4
3rd call:	cn/16
….	

The sum of this series adds up to less than 2cn -- without doing any math. (Since we know that cn + cn/2 + cn/4 + cn/8 + … + c adds up to 2cn and the above sum is less than that)

So, O(n log n) is not a tight upper bound. The tight upper bound is $\theta (n)$

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

Solution: We quickly see that the recursive call bigOh2 calls itself with a problem size one less than the current size. So, the function will call itself exactly n times. So, we can say that the best case run time is $\Omega (n)$. But if we look more carefully, we will see the string concatenation operation will cost a lot more:

n	output string
----	----------
0	a
1	aa
2	aaaa
3	aaaaaaaa
….	
n	2^n

So, $\Omega (n)$ is not a tight lower bound. In fact, the tight lower bound here is exponential $\theta (2^n)$ $(2+4+8+\ldots+2^n = 2^{n+1} = \theta (2^n))$

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote

a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

Solution:

a) Create a graph where nodes represent mathematicians and edges represent co-authorship.
b) Run BFS in that graph starting from Erdős and see at which level that particular mathematician appears in the BFS tree. That will be his/her Erdős number.
c) All mathematicians within the first 2 levels of the BFS tree (staring from Erdős) have Erdős numbers of at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

Solution: Find a topological order (See textbook for details on finding topological ordering in DAGs). See if there is a path that goes through the nodes of the graph in that topological order. If there is, then this will be the longest path we are looking for. In fact, if this path exists, it gives us a strict precedence order from the starting node to the end node on the path, and therefore the graph can only have one topological ordering. [Any other ordering of the nodes in the graph will violate this precedence order because if we move the position of any node in that order, we will end up with at least one edge that goes in the opposite direction of the topological order.] So, if our topological order does not provide such a path, then we can conclude that such a path does not exist in the graph.