CSCI 570 - HW 1

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

$2^{\log n}$, $(\sqrt{2})^{\log n}$, $n(\log n)^3$, $2^{\sqrt{2}\log n}$, 2^{2^n}, $n\log n$, 2^{n^2}

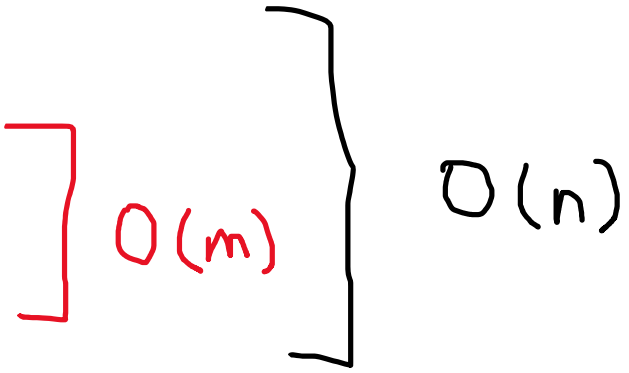
$2^{\sqrt{2}\log n} < (\sqrt{2})^{\log n} < 2^{\log n} < n\log n < n(\log n)^3 < 2^{2^n} < 2^{n^2}$

2. Give a linear time algorithm based on BFS to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. It should not output all cycles in the graph, just one of them. You are NOT allowed to modify BFS, but rather use it as a black box. Explain why your algorithm has a linear time runtime complexity.

BFS Algorithm pseudo code to find and print cycles:
Step 1: change the flag of every node to -1 (-1 represents not visited)
Step 2: take current node(as root node or initial node) and enqueue, change the flag value as 0(0 represents vertex in queue)
Step 3: push all the adjacent nodes or vertices of current node into the queue and change the flag value of current node as 1(1 represents vertex visited) and dequeue or pop current node from the queue
Step 4: assign the value of current node as next vertex in the queue and repeat Step 3 till all the vertices flag values are marked as 1 or any adjacent vertex flag value is found to be 0
Step 5: if the current vertex or node finds its adjacent vertex with flag value 0, then the graph contains cycle
Step 6: Then using back tracking, find the least common parent node(V) of current node and its adjacent node, print the path between V-current node, V-adjacent node and current node-adjacent node, these three paths form a cycle

BFS algorithm for traversal:

```
q=queue()
q.enqueue(start_vertex)
while(!q.empty()){
        v=q.dequeue()
        for (all u adj to v):{
                if(u is not visited){
                        q.enqueue(u)
                        }
                }
        }
```



Inner loop time complexity is O(m)
Outer loop time complexity is O(n)
Therefore the total time complexity is:
T = O(m+n)

There the BFS algorithm has a linear time complexity of O(m+n)

3. A binary tree is a rooted tree in which each node has at most two children. Show by *induction* that in any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.

Proof:

- n number of nodes in a binary tree
- Assume n=1, one leaf node or root node and zero nodes with two children
- By induction hypothesis, for any binary tree with n nodes, the number of nodes with two children is one less than number of leaves
- Now need to prove that the above hypothesis will also hold for n+1 nodes-
  Case 1- Inserting a child node to a node with no other child nodes. The tree with n+1 number of nodes will have same number of two child nodes and number of leaves
  Case 2- Inserting a new child node to a node with one child node. The new tree with n+1 nodes has one more node with two children and also one more leaf
- With the above assumption, using induction we can prove that for an arbitrary binary tree, the number of nodes with two children is always one less than the number of leaves

4. Prove by contradiction that a complete graph $K5$ is not a planar graph. You can use facts regarding planar graphs proven in the textbook.

Proof:

- Assume K5 is planar
- Then it should follow Euler's theorem
  R + N = A+2 (R-regions, N-nodes, A-arcs)
- Example-
  R=8 N=4 A=10
  Each region is bounded by at least 3 arcs
  So there are at least 3R/2 arcs, since each arc is 2 region boundaries
  (3*8)/2=12, so there must be at least 12 arcs
  But here are only 10 arcs, there is contradiction to Euler's Theorem
- So K5 is not planar

5. Suppose we perform a sequence of n operations on a data structure in which the i^{th} operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

In a sequence of n operations there are logn + 1 powers of two (1,2,4,…,2^logn)
The total cost of this sequence of n operations is

E(i=0 t0 logn) 2^i
=2^(logn +1)-1 <= 2^(logn +1)
= 2n

The other all operations cost will be n (since each operation costs 1 and there are n operations)
The total cost will be

T(n)<=2n +n

=3n
=O(n)

To find the amortized cost of each operation, divide the total cost by number of operations(n)

Amortized cost = O(n)/n = O(1)

Therefore the Amortized cost per operation is O(1)

6. We have argued in the lecture that if the table size is doubled when it's full, then the amortized cost per insert is acceptable. Fred Hacker claims that this consumes too much space. He wants to try to increase the size with every insert by just two over the previous size. What is the amortized cost per insertion in Fred's table?

Insert	Old size	New size	Copy
1	1	-	-
2	1	3	1
3	3	-	-
4	3	5	3
5	5	-	-
6	5	7	5
7	7	-	-
8	7	9	7
9	9	-	-
10	9	11	9
2n	2n-1	2n+1	2n-1

Total number of insertions = 2n (initially table size is n, but it is doubled to 2n)
Each insertion cost is 1
Total insertions cost = 2n
Total cost of all copies = 1+3+5+…+(2n-1) = n^2
Total cost = Total insertions cost + Total cost of all copies
= 2n + n^2
Amortized cost = total cost/number of insertions
= (2n+n^2)/2n
= 1 + n/2 = n (approximately)
Therefore amortized cost is O(n)

7. You are given a weighted graph G, two designated vertices *s* and *t*. Your goal is to find a path from *s* to *t* in which the minimum edge weight is maximized i.e. if there are two paths with weights 10→1→5 and 2→7→3 then the second path is considered better since the minimum weight (2) is greater than the minimum weight of the first (1). Describe an efficient algorithm to solve this problem and show its complexity.

Algorithm pseudo code:

Step 1: select s as the source vertex and t as the target vertex

Step 2: construct a binary search tree with s as the root node of the tree

Step 3: apply breadth-first-search using the root node to the target node, if there exists a path between s and t then store all the intermediate vertices or nodes to an array and assign the least weight among the set of all the edges in that path to a variable min_edge_weight

Step 4: apply BFS repeatedly to find different paths from s to t and find the edge weight which is minimum (current_min_edge_weight) in every iteration and compare it with the min_edge_weight, if the current_min_edge_weight> min_edge_weight then replace the previous path by new or the current path (minimum edge wight is maximized)

Step 5: repeat step 4 until we find all the possible paths and find best path among the all paths and print the path

Time Complexity:

Time complexity to construct binary search tree = O(E)

Time complexity of BFS = O(V+E)

Total time complexity = O(E) + O(V+E)