CSCI 570 – Homework 1
Due: Feb 07th 2021

Name: Rohan Mahendra Chaudhari
USC ID: 6675-5653-85
Email ID: rmchaudh@usc.edu

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}

   Solution:

   Functions in increasing order of growth rate results -

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

   Here,

   $2^{\sqrt{2}\log n} = \log(2^{\sqrt{2}\log n}) = \sqrt{2}\log n$

   $(\sqrt{2}\log n)^2 = 2\log n$ = Logarithmic Function

   $(\sqrt{2})^{\log n} = \sqrt{n}$

   $2^{\log n} = n$ = Linear Function

   $n\log n$ = Linearithmic Function

   $n(\log n)^3$ = Cubic Function

   2^{n^2} = Exponential Function = n^2 (On simplification)

   2^{2^n} = Exponential Function = 2^n (On simplification)

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.

   Solution:

   Assumptions:
   Let us consider a undirected graph G(V, E)
   V: set of vertices
   E: set of edges
   Let the spanning tree we get as a result of BFS on G(V, E) be T(V', E').

   Step 1
   To detect a cycle:
   Check for an edge e that is present in graph G but not in the spanning tree T. A Spanning Tree is a tree which is bipartite and thus has no cycles.

   Return NULL if we don't find such an edge. If all the edges e in set E of Graph G are present in set E' of the spanning tree T that means the graph does not have a cycle.

   Time complexity analysis for step 1:
   Given: In a Tree(V, E): Let v be the number of vertices and e be the number of edges in Tree T, then v=e+1. – **(1)**

   Linear time complexity proof for finding an edge that belongs to graph G but not tree T:
   Let us assume there are 'n' number of edges in the spanning tree T. In the edge set E of graph G, we look for any edge that does not belong to the spanning tree T. Using the given statement **(1)** mentioned above, the maximum number of edges that will have to be checked for getting to an edge that belongs to the graph G but not tree T would be = number of edges of tree=n+1=E'+1 **(Linear time): O(E)**

   Step 2
   Find Least Common Ancestor(LCA) between the two vertices which will give us the cycle inclusive of those two vertices and LCA node.

a. Find paths to both nodes (x,y)
   Paths - root to x and root to y
   and store the sequence in two arrays. **(Takes linear time)**
b. Compare array values at same index. Last common value is the LCA. **(Takes linear time)**
c. We obtain a cycle from the LCA, together with the root to x and root to y paths in T.

The time complexity would be linear since it only depends on creating the BFS tree and traversing it one time, both of which have linear time complexity.
Here the time complexity is O(|V|+|E|) which would be linear. Linear means linear in the size of input. If the input contains a list of all the edges as in this case, then O(|V|+|E|) is linear.

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.

   Solution:

   Theorem:
   If there are N full nodes in a non-empty binary tree then there are N+1 leaves. A full node is a node with 2 children.

   Base Step:
   If there are 0 full nodes in a non-empty binary tree then there is at most one leaf. This is true because it has at most one branch in the tree due to 0 full node.

   Inductive Step:
   If there are k+1 full nodes in a non-empty binary tree then there are k+2 leaves.

   Pick a leaf node and keep removing its parent recursively until a full node is reached. This full node becomes a non-full node because one of its child node is removed. At this point the tree will have one less leaf and one less full node.

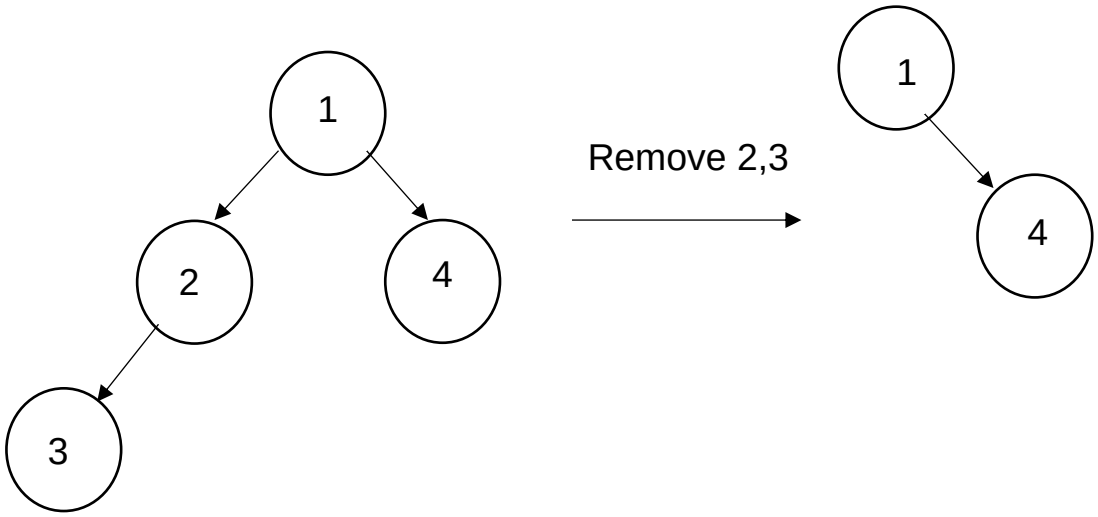

Figure 1

Therefore, the tree has k full nodes after the nodes are removed. By the hypothesis there are k+1 leaves. Add all the nodes that were removed back into the tree the same way to create the original tree. We are adding one full node and one leaf node. Therefore, we have k+1 full nodes with k+2 leaves. ($\forall k \geq 0$)

Thus, we can prove than for a nonempty binary tree the number of nodes with 2 children is exactly one less than the number of leaves.

Proved.

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.

   Solution:

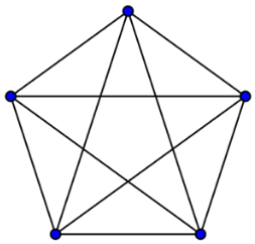

Figure 2

Proof by Contradiction –

a. Assume K5 is planar
b. Since we assume K5 is planar it must obey the Euler's relationship,

   R + V = E + 2

   Here, R = No of regions
   V = No of vertices
   E = No of edges

c. Considering a K5 planar graph as shown in the above figure we can infer that a K5 planar graph has,

   V = 5 & E = 10

   Thus by the Euler relationship,

   R = 7

d. Each region is bounded by at least 3 edges as shown in figure 2 above.
e. Thus, there are at least 3R region boundaries
f. As there are at least 3R region boundaries, we can further state that there would be at least 3R/2 edges, since each edge boundaries 2 regions.

   Thus, (3*7)/2 = 10.5, so at least 11 edges.

g. But as shown in the K5 planar figure above there are only 10 edges.

   Thus, this proves a contradiction due to which K5 does not obey the Euler's relationship and is thus not planar.

   Proved.

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.

   Solution:

   C_i = i (if i is an exact power of 2)

   C_i = 1 (Otherwise)

Operation (i)	Cost (C_i)
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1

Table 1

Cost of n operations:

$$\sum_{i=0}^{n} C_i <= n + \sum_{j=0}^{\lg n} 2^j$$

$= n + (2^{\lg n+1} – 1)/(2-1)$
$= n +(2n – 1)$
$< 3n$

Thus, the average cost of operations = Total Cost / # of operation < 3

By, aggregate analysis, the amortized cost per operation = 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?

Solution:

According to Fred's hypothesis,

Cost of operation 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

Table 2

The above table shows that 10 inserts require 1+3+5+7+9 = 25 copy operations.

Therefore, the amortized cost of a single insert is the total cost (Inserts plus copies)/(inserts) = (10+25)/10 = 3.5

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.

Solution:

Step 1:
Run DFS on the graph G to detect and remove all the cycle and obtain a binary tree.

Step 2:
Convert the above binary tree to a binary search tree using the edges weights as a comparison mechanism.

Step 3:
Make a BFS tree using the root node of the binary search tree created above while ignoring all the edges having weight less than the root and check if a path is found from s to t including the root of the BFS - **O(V+E)**.

Step 4:
If we are able to find a path we store it and then move towards the right child of the root in the binary search tree to try bigger weights and run the step 3 with the right child rather than the root node.

Step 5:
Alternatively, if we are not able to find the path, then we try with shorter weights which will be the left child in the Binary search tree. In the worst case scenario we are going to repeat step 4 or step 5 'h' number of times where h=Log E=height of the BFS tree.

Complexity:
This can be done in O((V + E) log E) time using binary search on the edge weights. Since there are |E| edges, there are at most |E| unique weights. The desired weight must be some edge weight so we binary search for the largest weight of w_{min} that maintains a path from s to t. Sorting the edges take O(E log E) time, and binary search takes O(log E) time, with each iteration requiring O(V + E) time to give O((V+E)log E).