Homework 1

Shreenidhi Suresh Hegde

February 7, 2021

Problem 1

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

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

Answer:

- $2^{\log n} = n$ (linear)
- $\left(\sqrt{2}\right)^{\log n} = 2^{\frac{1}{2}\log n} = 2^{\log\sqrt{n}} = \sqrt{n}$
- $n\left(\log n\right)^3 \geq n\log n$ as $\log n$ is an increasing function
- $2^{n^2} \leq 2^{2^n}$ as 2^n is exponential and n^2 is polynomial
- Taking log of $2^{\sqrt{2\log n}}$ and $\left(\sqrt{2}\right)^{\log n}$, we get $\left(\sqrt{2}\right)^{\log n} = \sqrt{n}$ and $2^{\sqrt{2\log n}} = \sqrt{2\log n}$.

The increasing order of the functions in terms of growth rate is as follows:

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

Problem 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.

Answer:
Let G be the given graph. Consider G is connected. If we do the BFS traversal,for every visited vertex V of the graph, if there is an adjacent vertex U which is already visited and U is not the parent of V, then the graph G contains a cycle.

Step 1 : Run BFS on the graph G from any vertex z.

Step 2 : We get a BFS Search tree, say T.

Step 3 : Compare the edges in G and T.

If T has all the edges in G, then there is no cycle in the graph G

If we find any edge say $u \longrightarrow v$ which is in G, but not present in T.

Then there is a cycle in graph G.

Step 4 : Find least common ancestor of vertices u and v by comparing the paths from the starting vertex z.

Step 5 : We now have a least common ancestor say, l. We can output the cycle $l \longrightarrow ...u \longrightarrow v \longrightarrow ...l$

Time complexity Of the algorithm:
Let E be the edges and V be the vertices of the graph G. Once we construct BFS search tree, we are storing the result in an adjacency matrix. To check if all the edges of G are present in T, we compare the adjacency matrix of T and G, which takes $O(E)$ times.If we find an edge e' consisting of vertices (u, v) which is in G but not in T, we say graph G has a cycle.
Now we have to find the least common ancestor of u and v. This we can find by constructing path from u to z and v to z. This takes $O(V)$ times as we traverse the vertices. Then we compare them to find least common vertex a.Comparing also will $O(V)$ times. Hence finding the least common ancestor takes $O(V)$ times.
Therefore, total time complexity of the algorithm is $O(E + V)$

Problem 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.

Answer:
Let P be the number of nodes with two children. Let L be the number of leaves

1. Base Case : Let's consider $L = 2$. Since we have a single node with two children, $P = 1$. Therefore, the base case satisfies $P = L - 1$.
2. Induction Hypothesis: Let us assume that $P = L - 1$ holds true for all the binary trees with $L = n$.
3. Induction Step: Let T be a binary tree with $L = n + 1$. Then, without loss of generality, T is obtained by adding a node with no children to a tree with $L = n$, say T', at a node with a single child. Using the induction hypothesis from step (2) we know that, the number of nodes with two children is exactly equal to the number of leaves minus one for the binary tree, T'. By construction, the number of nodes with two children of T' is one less than the number of nodes with two children of T. Therefore, the number of nodes with two children is equal to the number of leaves minus one for the binary tree, T. Since T is arbitrary, the equation $P = L - 1$ is true for any binary tree with $n + 1$ leaves.

Hence, by using the principle of mathematical induction, the number of nodes with two children is equal to the number of leaves minus one for any non-empty binary tree.

Problem 4

Prove by contradiction that a complete graph K_5 is not a planar graph. You can use facts regarding planar graphs proven in the textbook.

Answer:
K_5 is a simple connected graph with 5 vertices and 10 edges. We know that in any simple connected planar graph with at least 3 vertices, $E \leq 3V - 6$ as proved in the textbook. If K_5 is a planar graph, then $E \leq 3V - 6$ should holds true.
If we substitute values $E = 10$ and $V = 5$, we get $3V - 6 = 3 \times 5 - 6 = 9$ which is less than $E = 10$. This contradicts the proven theorem stated above. Hence K_5 is not planar.

Problem 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.

Answer:
Without loss of generality, let n is power of 2. Then, nth term which is a power of 2 would be $2^{\log n}$. The $1^{st}, 2^{nd}, 4^{th}$ and n^{th} operation will cost $1, 2, 4, .., 2^n$ and the remaining operations will cost $n - \log n + 1$ operations will cost 1.
Then the total cost of operation would be

$$\begin{aligned}
1 + 2 + 1 + +4 + 1 + 1 + 1 + 8 + ... + 2^{\log n} &= \left(1 + 2 + 4 + 8 + ... + 2^{\log n}\right) + (n - (\log n + 1)) \\
&= \sum_{k=0}^{\log n} 2^k + (n - \log n - 1) \\
&= 2^{\log n+1} - 1 + (n) - \log n - 1) \\
&= 2n + n - \log n - 2 \\
&= 3n - (\log n + 2)
\end{aligned}$$

Amortized cost per operation = Total cost of operations / Number of operations
Hence, average cost of operation when n tends to infinity is

$$\lim_{n \to \infty} \frac{3n - (\log n + 2)}{n} = 3$$

Therefore, amortized cost of the operation is a constant value.

Problem 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?

Answer:
According to the question, in Fred's table, size of the array is been increased by 2 every time the array is full. Let us consider n insertions starting with array of size 1.

- Insertion of first element is a direct insertion − 1 time step.
- To insert second element, array size is to be increased by 2. Size of our new array is 3. This will be 1 copy from the previous array and 1 new insertion.
- Insertion of 3rd element is again a direct insertion − 1 time step.
- As of now the array is again full, we are increasing the size by 2 and insertion of 4th element takes 3 copy and 1 insertion.
- Insertion of 5th element is a direct insertion − 1 time step.

Insertions	Old Size	New Size	Number of Copies
1	1	0	0
2	1	3	1
3	3	0	0
4	3	5	3
5	5	0	0
6	5	7	5

Now we can see that, there are n insertions and $2k - 1$ copy operations for $n = 2k$ elements. So, the total cost would be

$$\begin{aligned} n + 1 + 3 + 5 + ... + 2k - 1 &= n + \frac{k}{2}(1 + 2k - 1) \\ &= n + k^2 \\ &= n + \left(\frac{n}{2}\right)^2 \end{aligned}$$

The amortized cost per operation is

$$\begin{aligned} \frac{n + \left(\frac{n}{2}\right)^2}{n} &= \frac{4n + n^2}{4n} \\ &= 1 + \frac{n}{4} \\ &= O(n) \end{aligned}$$

Problem 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.

Answer:
Let G be the weighted graph and all the edges have a unique weight. First, we will sort all the weighted edges of G in non decreasing order and store it in a data structure say, SortedList. Let the total number of edges be E.In the function Traversal, we are picking up each edge from the

SortedList and checking if there are any edges whose weight is less than the picked up edge. If there is, then we are removing those edges.After removing the edges, we run DFS from s to v. If we find any path, that means we haven't found the path with maximum of the minimum edges. We keep doing this step and store the previous path in a list say *previousPpath* until we get no path from the vertex s to v. Once we don't get any path, the previous path which we found through DFS would be our solution.

```
Function Traversal ( G, SortedList )
 previousPpath = NULL
   for $i$ in range $(0, E)$
      Remove all the edges from $G$ whose weight is less than the weight of $i^{\text{th}}$ edge of
      SortedList
      Run DFS to find the path from the vertex $S$ to vertex $t$.
      path = DFS output
      If path not empty :
         previousPath = path
      Else
         print previousPpath
         break
```

Time Complexity:

Sorting all the edges of G in non decreasing order will take $E \log E$ times. Running DFS in each iteration will take $O(E(E + V))$. Then the time complexity of our whole algorithm is $O(E(V + E))$ times.