CSCI 570

Analysis of Algorithms Assignment - I

Harshitha Kurra | USC ID: 3161505044

---

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: let us consider the following where A(n), B(n), C(n), D(n), E(n), F(n), G(n) are functions of n:

$$A(n) = 2^{\log n}$$
$$B(n) = \sqrt{2}^{\log n}$$
$$C(n) = n(\log n)^3$$
$$D(n) = 2^{\sqrt{2\log n}}$$
$$E(n) = 2^{2^n}$$
$$F(n) = n\log n$$
$$G(n) = 2^{n^2}$$

Let us simplify the functions that can be simplified now.

$A(n) = 2^{\log n} = 2^{\log_2 n} = n^{\log_2 2} = n$

$B(n) = \sqrt{2}^{\log n} = 2^{\log_2 \sqrt{n}} = \sqrt{n}$

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

Apply log on both sides:

$\log(D(n)) = \sqrt{2\log n} = \sqrt{\log n}$ compared to B(n), log grows slower. It is slower than any other functions mentioned. $\sqrt{\log n}$ grows much slower so this will be the least.

Comparing C(n) and F(n): C(n) is $(log n)^2$ times greater than F(n) so, F(n) will be smaller than C(n)

D(n), B(n), A(n) are polynomial so they grow very slow compared to all the other functions

Comparing E(n) and G(n): both are exponential functions. They grow faster than all the other functions. Out of 2^n and n^2 , the n^2 grows slower than 2^n. So, G(n) will be slower than E(n)
The comparison happens like:

$O(1) <= O(logn) <= O(\sqrt{n}) <= O(n) <= O(nlogn) <= O(n^2) <= \ldots . <= O(2^n) <= O(n^n) <= O(n!)$

so, the sequence will be:

$$D(n) < B(n) < A(n) < F(n) < C(n) < G(n) < E(n)$$

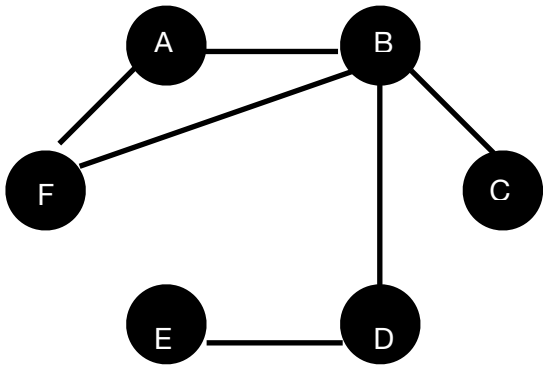
2. **Give a linear time algorithm based on BFS to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. It should not output all cycles in the graph, just one of the……,. You are NOT allowed to modify BFS, but rather use it as black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

Solution: To detect if there is a cycle in BFS, we can mark the node as we reach it. If the sequence reaches the node which is already marked then we are quite certain that there exists a cycle. Consider a BFS tree, we start from the root and whenever there is edge to another vertex, we add that node to the tree with the edge. We maintain an array to keep track of the parent of the vertices. For every vertex ‘v’, if there exists an edge in the graph but the edge is missing in the BFS tree, then we check the parent vertex and check if the edge exists with the parent node or any other node. If it is not a parent node then there exists a cycle. Cycle here is back tracing from this edge to the point where they meet. BFS runs with linear time as each vertex/edge is visited just once. **For V nodes and E edges, the runtime complexity is O(V+E).**

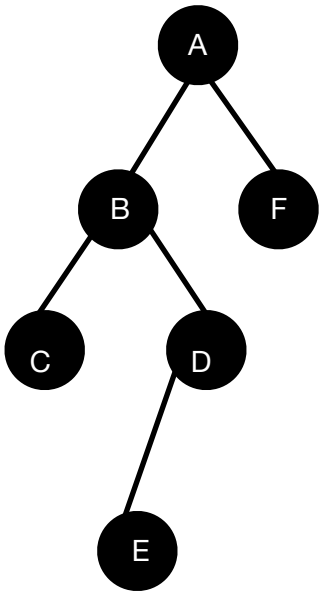
Pseudo Code:

```
Define BFS_Cycle(Graph):
        BFS_edges = output of BFS tree
        parent = [-1] * V                   //Initializing the parent array
        for (u, v) in BFS_edges:            //for the edge in the edges
                parent[v] = u               // store the parent of each vertex
        for (u,v) edge in graph and not in BFS_edges:  //if you find a edge in graph which is not
                                                        //in the tree and if the node is not the
                                                       //parent of the vertex then that means a
                                                       //cycle exists
                if u != parent[v] and v!=parent[u]:
                        return cycle path
        return 0
```

Example:



The edge pairs here are: (A,F), (A,B), (B,F),(B,C), (B,D), (D,E)



The edge pairs here are: (A,B), (A,F), (B,C), (B,D), (D,E)

Parent array:

A	B	B	D	A
B	C	D	E	F

The edge that is missing between the graph and tree is (B,F).

In the parent array, the parent of F is A

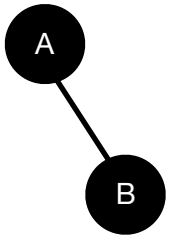
=> B != parent(F)

=> there exists a cycle.

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

Solution: By Proof of Induction, Let P(k) denote that, in a binary tree for k nodes with two children, there exists k+1 leaves.

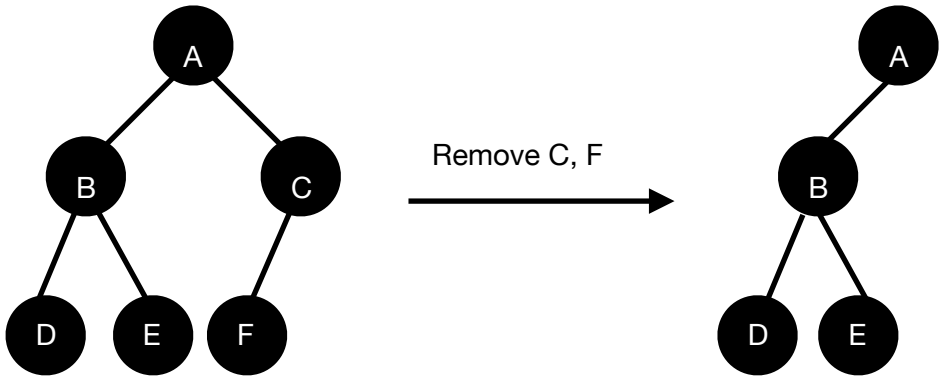
Base Case: P(0): If there is a tree with zero full nodes, that means there will be one leaf node and therefore the **nodes with two children= 0, leaves = 1**. Thus, it is implied that the number of nodes with two children is one less than leaf nodes. So, P(0) holds true.



Inductive Hypothesis: Assume P(k) is true $\forall k < n$. Then P(k+1) i.e; It contains k+1 nodes with two children and k+2 leaves.

Inductive Step: As we are assuming P(k) is true, that means the binary tree has k full nodes have k+2 leaves.

Let us consider Trees T and T'



T **T'**

In the tree T, keep removing nodes from leaf node and its parents till you reach the full node and let's call this modified tree as T'. By this time, we will be having 1 less leaf node and 1 less full node. So, the number of full nodes after the nodes are removed will be k. According to the Inductive hypothesis, the number of leaf nodes are k+1. Now, let's add all the removed nodes. I.e; we get one leaf node and one full node. So the number of full nodes now will be k+1 and the leaves will be k+2.

The difference is: (k+2) - (k+1) = 1

Which implies, if we add one node then either both number of leaves and full nodes(nodes with 2 children) increases by 1 or none of them change.

therefore, P(k+1) is always true.

So, we can conclude that P(k) => P(k+1)

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.

Solution: A planar graph is a graph where any two nodes do not have crossing edges. K5 is a complete graph with 5 nodes.

Proof by contradiction: Assume K5 is planar, prove that there is a contradiction in this assumption and hence, K5 is not planar.

According to Euler's formulae: F + N = E + 2 (number of faces + number of nodes = number of edges + 2)

Here N = 5 and Edges = 10

So F + 5 = 10 + 2 => F = 7

Which implies the number of faces to be 7.

The number of edges E <= 3V - 6 for a maximal planar graph according to Eulers formulae. So

E < = 3V - 6

E <= 3(5) - 6

E <= 9 but out Edges here are 10 so there is a contradiction.

Hence, K5 is non-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.

Solution: Assuming the base is 2 for all logs mentioned below.

For example, In a sequence of 1,2,3,4,5,6,7,8,9,10 (10 numbers) we have 3 exact powers of 2(= 2^1, 2^2, 2^3) + 1 (= 2^0). So for 10 numbers we have 3 exact powers of 2.

3 ~= log(10)

So in a sequence of n numbers, we have (1 + log(n)) exact powers of 2.

The terms are 1,2,4,8,16,…….. $2^{\log n}$. The sum of the geometric sequence is

Sum of n terms formulae in G.P: $a(r^n - 1)/r-1$

Sum of logn terms is = $1(2^{\log n+1} - 1)/2-1$

= $2^{\log n+1} - 1 <= 2^{\log n+1} = 2n = O(n)$

The remaining terms (n - 1 - logn) the cost is 1. And the number of remaining terms is < n.

So the total cost = cost for exact power of 2 + cost for remaining operations = $O(n) <= 2n + n$

= O(3n) for n operations.

= O(3) per operation

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: If we increase the array just by size 2, worst case let us assume there are n pushes, then the cost for n operations = 1+2+3+4+….. n

= n(n+1)/2

So, the cost for 1 operation = (n+1)/2 = O(n)

If we double the table size then the cost for n operations = 1+2+4+8+…….2i where 2i <= n

= 2n - 1 for copying + n for inserting

< 3n

Therefore, the cost for 1 operation = 3 which implies it is done in constant time.

Since, doubling the table size is taking a constant time, we prefer this over increasing the size by any integer which takes a linear time.

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: We can solve the problem in two ways with the same complexity:

I. Using BFS and Binary search: We initially get the paths from source to target using BFS. We store all the weights in an array and sort them. After sorting them, we apply binary search to get the max(min weight). In the binary search, we take the middle weight, if the weight exists then we move to the right to see if there is any max value available. If the middle element(weight) of the array does not exist, then we assume the weight might be too max and move left.

calculate the weights from source to destination and sort the results. Sorting takes O(nlogn). Since we have E edges, it takes O(ElogE). Later, we need to search for the maximum value of min weight from source to destination. For this, we use binary search. It takes O(logE) time. Since we have V vertices, E edges; for each iteration **Overall, it takes O((E+V)logE)**

(Pseudo code on next page…)

Pseudo Code:

```
Define BFS_BS(Graph):
    Get the weights of all edges and store them in an array
    for (u,v) in graph:
        Insert w(u,v) into weights_array[]
    weight_array.sort()   //sort the array in ascending order
    //perform the binary search
    min = 0, max= len(weights_array)
    while (min<= max):
        average = (min+max)/2
        // run BFS algorithm, ignore paths with minimum weight < weight_array[average]
        edges[] = BFS_BS(Graph)
        if no edges[]:              //no edges whose min weight < weight_array[average]
            max = average -1; //this implies the weight[average] is too high and we
                              //need to go to the left side to find the smaller value
        else:
            min = average + 1; // this implies go to the right of weight[average]
                               //because we found a path and we are trying to see if
                               //there is a better path. Since the goal is to find the
                               //max(min weight)
    return
```

II. Using Dijkstra's algorithm with priority queue: We begin by setting all node distances to ∞ and starting nodes distance to 0. This is O(V) complexity. From the start vertex and go to vertices that are connected to it and store the path weights. We add these elements into priority queue. Since, the priority queue is initially empty, adding the first element costs O(1). When there are no more edges the start vertex, we pop the next connected vertex which costs O(logV). We are counting each vertex only once in the priority queue and the entire

complexity is the complexity of adding the (vertex, edge) into the priority queue and popping the same. Then, compare the weight of the path weight from this vertex to next with the previously stored path weight value. The minimum value will be stored and …so on till we reach the target vertex. At the target vertex, we choose the maximum of all the minimum values we stored. **Overall, It takes O((V+E)logV)**.

Pseudo Code:

```
Define Dijkstra_PQ(Graph):
    S = {}; d[s] = negative infinity; d[v] = infinity for v != s // initialising
    While S != V
        Choose v in V-S with minimum d[v]  //choose min weight
        Add v to S
        For each w in the neighborhood of v
            d[w] = max(min(d[v], weight(v, w)), d[w])  // get maximum value out of
                                                       //  all the min values
```