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

Solution:

Below are the functions ordered in increasing order of growth rate with g(n):

$$2^{\sqrt{2logn}} < (\sqrt{2})^{logn} < 2^{logn} < nlogn < n(logn)^3 < 2^{n^2} < 2^{2^n}$$ (Left to right arranged in increasing order)

The reasoning for the above order is that we know that 2^{logn} gets reduced to n when we apply log_2 on both sides of the equation. Hence, $nlogn$ and $n(logn)^3$ would be greater than 2^{logn} as it has a multiplying factor of $logn$ in the respective order.

We also know that exponential functions have greater order of growth and hence 2^{n^2} and 2^{2^n} being exponential are arranged at the end in the respective order as 2^{2^n} has another exponent of a factor of n. Function $(\sqrt{2})^{logn}$ reduces $0.5n$ and is hence placed before 2^{logn}.

$2^{\sqrt{2logn}}$ reduces to $\sqrt{2logn}$ after taking log on both sides of the equation hence it is the function that has smaller growth than the rest.

2. Given 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: We have the below given to us,

- Use BFS as blackbox for cycle detection and printing the path.
- $G(V, E)$ which is undirected

To Do: Give a linear time algorithm based on BFS to detect whether a given undirected graph contains a cycle and if it does output one of the cycles.

DETECT-AND-PRINT-CYCLE-PATH(G):

In order to detect cycles in an undirected graph using BFS, if we come across any adjacent vertex *v* to every visited vertex *u* which has already been visited and *v* is not the parent of *u i.e* if there is any cross edge/back edge from a vertex *a* to its ancestor *b,* we can tell that there is a cycle in the graph *G*.

1. Initially, we store all the adjacencies for each vertex in an adjacency list. Give the graph as input to the blackbox(*that uses BFS*) start with a vertex starting from v, where v ε V and run it through all vertices, it returns a spanning tree, *S* as output.
2. Find all edges in the graph G that is not present in the spanning tree, *S* i.e these would essentially constitute as the back/cross edges of the graph *G.* Let one of such an edge be between v_1 *and* v_2 . This can be found using the adjacency list.
3. Also, find the least common ancestor from the parent array (*that stores the parent of every vertex*), let it be named as *a*.
4. Find a path from $a -> v_1$ and $a -> v_2$ and the edge (v_1, v_2). This path can be traced from the parent array.
5. Return from the algorithm if a cycle is found and a path is printed based on a boolean flag set which allows to break out of the loop after we find the first cycle.

We need to make sure that we process the undirected edge exactly once and store in the adjacency list as otherwise it may give rise to false two vertex cycles.

The above algorithm has linear time complexity because we just input the graph *G* to the blackbox that is based on BFS which has a complexity of $O(V + E)$ that produces the spanning tree, wherein further we just iterate across all the vertices in a graph in order to find those edges in the spanning tree and not in graph and this would take $O(V)$ In order to print the path we make use of the preexisting parent array which hence is in fact precalculated in the BFS traversal section of the blackbox. Thereby, the total complexity of the algorithm is $O(V + E)$ ie. its linear.

3. Let $G = (V, E)$ be a connected undirected graph and let *v* be a vertex in *G*. Let *T* be the depth-first search tree of *G* starting from *v*, and let *U* be the breadth-first search tree of G starting from *v*. Prove that the height of *T* is at least as great as the height of *U*.

Solution: We have the below given to us:

$G(V, E)$ – *connected undirected graph*, *v is a starting vertex in G*

T – *Depth First Search* (*DFS*) *Tree*, U – *Breadth First Search* (*BFS*) *tree*

To Prove: Height of T $\geq$ Height of U

Proof: Using *Proof by Contradiction*,

Assume *Graph G* has a BFS tree (U) of height greater than the height of the DFS tree (T) starting from *v*. But we know that by definition of a BFS tree, it always produces a spanning tree i.e *U* (*Breadth First Search Tree*) that has a height *h* starting from vertex v wherein the path to any vertex reachable from it consists of smallest number of edges as compared to the height of *T (Depth First Search Tree)*. Hence, the vertex *v* cannot be at a height lesser than or greater than *h* from *v* thereby contradicting with our assumption that *G* has a BFS tree whose height is greater than the DFS tree whose height is not bound by the condition that it produces a tree consisting of smallest number of edges from a source. Hence, proving that height of a DFS tree is always greater than or equal to the height of a BFS tree *(Height of T* $\geq$ *Height of U)*

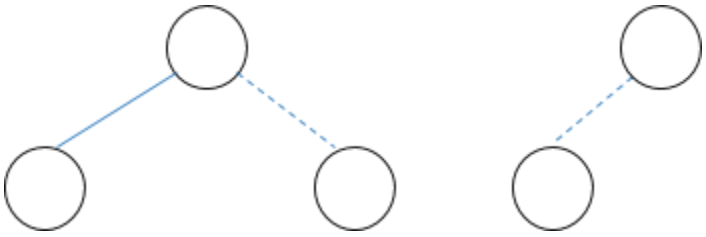
4. A binary tree is a rooted tree in which each node has at most two children. Show by *induction* that in any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.

Solution:

To Prove: In any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.

Proof: Using *Proof by Induction* on the number of leaves in a tree.

- **Base Case:** Consider a tree with one leaf *(L)* ie only a root exists with no children. In this case, the number of nodes with two children is 0 i.e *(L-1)*. Similarly for tree where the root has one leaf, the number of nodes with two children is 0. For trees where the root has two leaves, the number of nodes with two children is 1. Hence, reiterating the fact that the number of nodes with two children is one less than the number of leaves.
- **Inductive Hypothesis:** Assume $T(n)$: For a binary tree consisting of n nodes , the number of nodes with two children is exactly one less than the number of leaves is true for some n.
- **Inductive Synthesis:** Assuming *T(n)* is true, we can now build a new tree with $n+1$ nodes by adding the new node to either a parent who has one child or a single parent with no children. In the first case we see that the number of leaves increases by 1 and also the number of nodes with two children increases by 1. In the second case, we see that the number of leaves and the number of nodes, both remain the same.



With these two observations, we claim that $T(n)$ implies $T(n+1)$. Hence, by the principle of mathematical induction we can claim that $T(n)$ is true for all integers n, where the number of nodes with two children is exactly one less than the number of leaves.

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

Solution:

To Prove: K_5 is not a planar graph.

Proof: Using *Proof by Contradiction,*

We first assume K_5 is a connected planar graph,

In a K_5 we have the below,

- No. of vertices $(V) = 5$
- No. of edges $(E) = 10$

We know that in any graph, $2E = \sum degree(V)$

$2E = 4+4+4+4$

$E = 10$

Using the below theorem that is proved in the textbook for simple connected planar graphs,

Theorem: *In any simple connected planar graph with at least three vertices,* $E \leq 3V-6$.

Substituting the values of V in the above equation we get,

$E \leq 3(5) - 6$

$E \leq 9$

Although, from our initial calculation we got $E = 10$ and from the theorem we are arriving at $E \leq 9$, This is a contradiction. Thereby proving that, K_5 is not a planar graph.

6. Suppose we perform a sequence of *n* operations on a data structure in which the *ith* 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: We have the below given,

- Total number of operations performed: n
- Let cost of the operation for i^{th} *operation* be $cost(i)$

Let us define the cost function as below,

$cost(i)$ = i if i is an exact power of 2

1 otherwise

Below we have constructed a sample table for some operations in order to generalize the amortized cost per operation.

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8

We can hence see that in a sequence of n operations, there are $log_2 + 1$ exact powers of 2.

Total cost of n operations: $\sum_{i=1}^{\log n} 2^i + \sum_{i \le n \text{ and not a power of } 2} 1$

Using the geometric sum for the first term we get,

$$\leq \frac{2^{\log n+1}-1}{2-1} + n$$

$$\leq 2n - 1 + n$$

$$\leq 3n$$

Amortized Cost = $\frac{Total\ cost\ of\ operations}{No.\ of\ operations}$

$$= \frac{3n}{n} = 3$$

Thus, amortized cost per operation is: $O(1)$

7. 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: Based on Fred's description of the new table wherein we increase the size with insert by 2 over previous size, let's construct a table in order to generalize the amortized cost per insert,

Input size	Old size	New size (old size + 2)	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
11	11	-	-

We can generalize the the copies as:

 # Copy = $1 + 3 + 5 + 7 + 9.....2n - 1$

(Last term for sum of copies is 2n -1, Using the formula $a_n = a_1 + (n-1)d$ of arithmetic series)

wherein a_1 *is* 1 , $d = 2$ of

$= \sum_{1}^{2n-1}$ Sum of all odd numbers = $\frac{n}{2}$ [$2a + (n-1)d$]

$= n^2$

 # Inserts = $2n + 1$

Total Work = #Inserts + #Copy

$= n^2 + 2n + 1$

$= (n+1)^2$

Amortized Cost = $\frac{Total\ Work}{No.\ of\ insertions}$

$= \frac{(n+1)^2}{2n-1}$

Applying limits to $n \rightarrow \infty$, we get amortized cost per insert as $O(n)$

8. 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:

Given: We have a weighted graph $W(V, E)$, source vertex *s* and destination vertex *t*

To Do: Find a path from *s* to *t* in which the minimum edge weight is maximized.

Before iterating over the algorithm we will be needing the below data structures:

- $Priority\ Queue < Weight, < Path >>$ wherein the priority of this queue is the weight associated. We implement the priority queue using a heap which would essentially be a max-heap.
- *Array/List* for storing the path from $s -> t$

Algorithm to compute such a path wherein the minimum edge weight is maximized:

FIND-PATH-WITH-MAX-MIN-EDGE-WEIGHT(W, *s, t*):

1. We call a recursive function that starts with a DFS traversal from the source and in the process keep storing the vertices traversed in a *Path[].* While in that process, we also mark the vertices that are in the path as seen or else the traversal would get into a cycle.
2. In the process of the traversal and adding the path we also check for the minimum weight in the path while traversing from $s -> t$ and store the localPath minimum weight in a variable.
3. When we reach the destination vertex and we have calculated the entire path from $s -> t$, we store the localPath minimum weight as the priority and the path stored in *Path[]* as the data in the priority queue.
4. For every other path we check for priority i.e the localPath minimum weight and check if that is greater than the one stored in the priority queue and if greater we swap and place the highest minimum weight to the root.
5. At the end of calculating all paths we call *getHighestPriority* in order to retrieve the highest minimum weight and retrieve the path for the same and print it. If there are two values with the same value, then we return any element among them and print that path.

The major complexity of the above algorithm comes from the following :

- Complexity of DFS traversal: $O(V + E)$
- Complexity of inserting edge weights in the priority queue: $O(\log E)$
- Complexity of *getHighestPriorityCall:* $O(1)$

Combing all the terms, we get the complexity as $O((V + E) \log E)$

9. When we have two sorted lists of numbers in non-descending order, and we need to merge them into one sorted list, we can simply compare the first two elements of the lists, extract the smaller one and attach it to the end of the new list, and repeat until one of the two original lists become empty, then we attach the remaining numbers to the end of the new list and it's done. This takes linear time. Now, try to give an algorithm using O(n log k) time to merge k sorted lists (you can also assume that they contain numbers in non-descending order) into one sorted list, where n is the total number of elements in all the input lists. Use a binary heap for k- way merging.

Solution:

To Do: Algorithm to merge k sorted lists in $O(n \log k)$ time using binary heap for $k-way\ merging$

Given:

n - Total number of elements in all input lists.

MERGE-K-SORTED-LISTS(k, n, l_i.....l_k):

1. Build a *min-heap* of all the k sorted lists (l_i.....l_k) where in each sublist is a node in the min-heap.

   In the process of comparing two sublists, we must go ahead and compare the respective first element(minimum element in each sublist) of each sublist.
2. Use *remove-min* algorithm and remove the minimum element from the root(*sublist*).
3. Revise the root(*sublist*) and call *heapify* on the min-heap based on the new minimum element in the root sublist.

   In any case if the root*(sublist)* becomes empty, then we can pick any leaf(*sublist*) and further call *heapify*.

Creation of *min-heap* in Step 1 will take $O(k)$ as node in the tree is a sublist of length n and the number of swaps needed are based on the comparisons made with k sublists.

Each extraction of *min-element* will take $O(\log k)$, but since we would end extracting n elements, the total complexity for Step 2 will be $O(n \log k)$.

The total complexity of both the steps would be: $O(n \log k) + O(k) = O(n \log k)$ as the assumption is $k << n$