CPSC 413 – Design and Analysis of Algorithms I
ASSIGNMENT 1

Name: Jian Liao

Student ID: 30081230

This LaTeX solution template is referenced from CPSC 418 / MATH 318

Question 1 — For each of the following pairs of functions, $f(n)$ and $g(n)$, indicate whether $f = O(g), f = o(g), f = \Omega(g), f = \omega(g), or f = \Theta(g)$. Give an exhaustive list of all true relationships. No justification required.

(a) $f = 2^n, g(n) = n^2$;
Answer: $f = \omega(g)$ *and* $f = \Omega(g)$

(b) $f = 2^n, g(n) = 2^{2n-4}$;
Answer: $f = o(g)$ *and* $f = O(g)$

(c) $f(n) = log(n!), g(n) = n^{1.02}$
Answer: $f = \omega(g)$ *and* $f = \Omega(g)$

(d) $f(n) = n^{cosn}, g(n) = \sqrt{n}$
Answer: The relationship between $f(n)$ and $g(n)$ does not exist.

Question 2

(a) Look up the definition of a biconnected undirected graph on Wikipedia. Give a one sentence definition based on induced sub-graphs. Start your definition with "An undirected graph $G = (V, E)$ is biconnected, if ..."

Answer: An undirected graph $G = (V, E)$ is biconnected, if G is a connected and "nonseparable" graph, meaning that if any one vertex were to be removed, the graph will remain connected. In other words, a biconnected graph has no articulation vertices.

(b) For a directed graph $G = (V, E)$, its underlying undirected graph is obtained by replacing every directed edge (u, v) with an undirected one $\{u, v\}$. (If (u, v) and (v, u) are both in E, then the underlying undirected graph still contains only one edge $\{u, v\}$.)

Give a strongly connected directed graph such that its underlying undirected graph is not biconnected.

Answer:

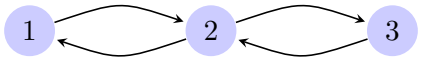

Figure 1: A strongly connected directed graph $G(V, E)$

Figure 2: The underlying undirected graph of $G(V, E)$

Proof of correctness: Figure 1 is a strongly connected directed graph because every node $x \in V$, and let $s \in V$, s and x are mutually reachable. Figure 2 is the underlying undirected graph of $G(V, E)$ obtained by replacing every directed edge (u, v) with an undirected one $\{u, v\}$. And Figure 2 is not biconnected because if vertex 2 is removed, the graph will not remain connected, in other words, vertex 2 is an articulation vertex.

Question 3 — Design an algorithm that takes as input a DAG $G = (V, E)$ in adjacency list representation, and outputs for each vertex $v \in V$ the length $d(v)$ of the longest directed path that ends at v. For full marks, your algorithm should have worst-case running time $O(|V|+|E|)$. State your algorithm's worst-case running time, and provide a brief justification for correctness and for your running time statement.

Answer:

Algorithm 1: The longest directed path in a DAG

Input: A DAG $G = (V, E)$ in adjacency list representation
Output: For each vertex $v \in V$ the length $d(v)$ of the longest directed path that ends at v

```
1  Run topological sort on the input DAG G = (V, E), return a topological order of G and
     store it in a Stack;
2  int dist[V];
3  for i = 0 → V do
4  |  dist[i] ← 0; // Initialize distances to all vertices as 0
5  end
   // Process vertices in topological order
6  while Stack is not empty do
   |  // Get the next vertex from topological order
7  |  int u ← Stack.pop();
   |  // Update distances of all adjacent vertices
8  |  foreach v ∈ Adj[u] do
9  |  |  dist[v] ← max(d[v], d[u] + 1);
10 |  end
11 end
12 for i = 0 → V do
   |  // Return the d(v) of the longest directed path that ends at each vertex
13 |  return dist[i];
14 end
```

Proof of analysis: Time complexity of topological sorting is $O(|V| + |E|)$. After sorting this graph G, the algorithm processes all adjacent vertices for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(|E|)$. So the inner loop runs $O(|V| + |E|)$ times. And also the algorithm will print the results which is $O(|V|)$. Therefore, overall time complexity of this algorithm is $O(|V|+|E|+|V|+|E|+|V|)$, which is $O(|V|+|E|)$.

Proof of correctness: For every DAG $G = (V, E)$, there must exist a topological order representing a linear order $(V_S, ..., V_N)$ of this graph G. In this algorithm, it initializes the distances to all vertices as 0 and then finds out the topological order of this graph G. Then the algorithm processes all vertices one by one in topological order. For every vertex being processed, the algorithm updates distances of its adjacent vertices using the distance of current vertex which is $d[u] + 1$, or the distances remain the same if $d[v] \geq d[u] + 1$. After the algorithm processing all the vertices one by one in topological order, the distance $dist[v]$ of each vertex $v \in V$ is the longest directed path that ends at v in G.

This can be proved by induction. The base case is that $G(V_1, ..., V_N, E)$ only has the edges $\{(V_1, V_2), (V_2, V_3), ..., (V_{N-1}, V_N)\}$, the topological order of G is exactly $(V_1, ..., V_N)$ and the distance of the longest directed path of each vertex $v \in V$ is exactly the distance of $|V_1 \to V_v|$.

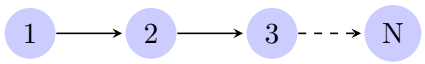

Figure 3: The base case of $G(V_1, ..., V_N, E)$

Assume adding an edge (V_1, V_N) to G, the distance of the longest direted path that ends at vertex N still remain the same as $|V_1 \rightarrow V_N|$. Because when the algorithm is processing V_1, V_N is one of its adjacent vertices, then $dist[V_N]$ will be set to $dist[V_1] + 1 = 1$. After N-1 steps, the algorithm will process V_{N-1} and update $dist[V_N]$ to $dist[V_{N-1}] + 1$, which is still $|V_1 \rightarrow V_N|$ eventually. By induction, adding any edges $(V_x, V_y), (V_x, V_y \in V)$ to E in topological order will not be changing the distance of the longest directed path that ends at each vertex.

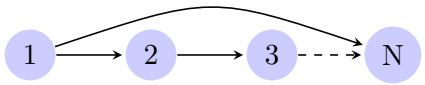

Figure 4: Adding an edge (V_1, V_N) to $G(V_1, ..., V_N, E)$

Assume adding a vertex V_M that also satisfies $InDegrees = 0$ and an edge (V_M, V_N) to the graph in topological order, the distance of the longest direted path that ends at vertex N still remain the same as $|V_1 \rightarrow V_N|$, and also it satisfies $dist[V_M] = 0$. Because when the algorithm is processing V_M, V_N is its adjacent vertex, then $dist[V_N]$ will be set to $dist[V_M] + 1 = 1$ and $dist[V_M]$ remains 0. After N-1 steps, the $dist[V_N]$ will be updated to $dist[V_{N-1}] + 1$, which is still $|V_1 \rightarrow V_N|$ eventually. By induction, adding any vertices V_v that satisfies $InDegress = 0$ and edges $(V_v, V_n), (V_v, V_n \in V)$ to E in topological order will not be changing the distance of the longest directed path that ends at each vertex.

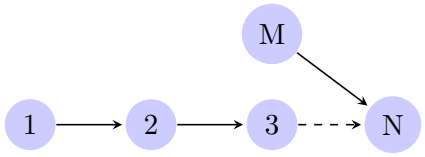

Figure 5: Adding a vertex V_M and an edge (V_M, V_N) to $G(V_1, ..., V_N, E)$

Proved by induction as required. After the algorithm processing all the vertices one by one in topological order, the distance $dist[v]$ of each vertex $v \in V$ is the longest directed path that ends at v in G.

Question 4 — Design an algorithm that outputs for a given list of n integers, one of the integers that occurs most often in the list. E.g., if the list is (1, 5, 3, 5, 1, 2, 7, 5, 3, 4, 9, 3), then the algorithm might output either 3 or 5, because both integers occur three times in the list, while every other integer occurs only once or twice. For full marks your algorithm should have worst-case running time $O(nlogn)$. State your algorithm's worst-case running time, and provide a brief justification for correctness and for your running time statement.

Answer:

Algorithm 2: Find integer(s) that occurs most often in the list

Input: A list of integers
Output: The integer(s) that occurs most often in the list

```
1  Run merge sort on the input list of integers in descending order, return a sorted list of
     integers List;
2  int max; // Temporary num
3  int count ← 1; // Temporary count
4  int maxcount ← count; // Store the max count
5  int maxnum; // Store the max num
6  for i = 0 → List.size − 1 do
7  |   max ← List[i];
8  |   if List[i+1]==max then
9  |   |   count + +;
10 |   else
11 |   |   count ← 1;
12 |   end
   |   // FindMax
13 |   if count > maxcount then
14 |   |   maxcount ← count; // Store the maxcount
15 |   |   maxnum ← max; // Store the maxnum
16 |   end
17 end
18 return maxnum;
```

Proof of analysis: Time complexity of merge sort is $O(nlogn)$. After sorting the input list of integers. The algorithm processes all the elements in the list, which is $O(n)$. Therefore, overall time complexity of this algorithm is $O(nlogn + n)$, which is $O(nlogn)$.

Proof of correctness: By sorting the list, the integers that are the same will be placed together and sorted in descending order. Therefore, the algorithm can easily count the occurence of each interger in the list by doing a traversal of the list. And essentially, the algorithm will find the max count and also the integer that occurs most often in the list, which is similar to the process of FindMax.

Cover Page for CPSC 413
Homework Assignment #1

Name: Jian Liao

Course Section: CPSC 413 Tutorial 01

Collaborators:

Question1: N/A

Question2: N/A

Question3: N/A

Question4: N/A

Other Sources:

Question1: N/A

Question2: https://en.wikipedia.org/wiki/Biconnected_graph

Question3: https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/

Question4: https://blog.csdn.net/Allenalex/article/details/10830797

Declaration:

I have written this assignment myself. I have not copied or used the notes of any other student.

Date/Signature: Jian Liao, May 23, 2019