Name: ______________________

Entry number: ______________________

There are 6 questions for a total of 75 points.

---

1. Consider functions $f(n) = 10n2^n + 3^n$ and $g(n) = n3^n$. Answer the following:
   (a) (½ point) State true or false: $f(n)$ is $O(g(n))$.

   (a) **True**

   (b) (½ point) State true or false: $f(n)$ is $\Omega(g(n))$.

   (b) **False**

   (c) (2 points) Give reason for your answer to part (b).

   Solution: $f(n)$ is $\Omega(g(n))$ iff there exists constants $c > 0, n_0 \geq 0$ such that for all $n \geq n_0$ $f(n) \geq c \cdot g(n)$. So, $f(n)$ is not $\Omega(g(n))$ iff for all constants $c > 0, n_0 \geq 0$, there exists $n \geq n_0$ such that $f(n) < c \cdot g(n)$. Note that $(c/2)n3^n > 3^n$ when $n > 2/c$ for any constant c. Moreover, note that $(c/2)n3^n > 10n2^n \Leftrightarrow (3/2)^n > 20/c \Leftrightarrow n > \log_{3/2}(20/c)$.
   Combining the previous two statements, we get that for any constant $c > 0$, $cn3^n > (10n2^n + 3^n)$ when $n > \max(2/c, \log_{3/2}(20/c))$. This further implies that for any constants $c > 0, n_0 \geq 0$, $cn'3^{n'} > (10n'2^{n'} + 3^{n'})$ for $n' = \max(2c, \log_{3/2}(20c), n_0)$. Note that $n' \geq n_0$. So, for any constants $c > 0, n_0 \geq 0$, there is a number $n \geq n_0$ (n' above is such a number) such that $cn3^n > (10n2^n + 3^n)$. This implies that $f(n)$ is not $\Omega(g(n))$.

2. (2 points) A rooted tree has a special node r called the *root* and zero or more rooted subtrees $T_1, ..., T_k$ each of whose roots are connected to r. The root of each subtree is called the *child* of r and r is called the *parent* of each child. Nodes with no children are called *leaf* nodes. A *binary tree* is a rooted tree in which each node has at most two children.

   Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

   Solution: We prove by induction on the number of vertices of the binary tree. Consider the proposition:
   $P(i)$: *In any binary tree with i nodes, the number of nodes with two children is exactly one less than the number of leaves.*

   We will show by induction that $P(i)$ is true for all i.

   Base case: $P(1)$ is true since a binary tree with one node has 1 leaf and 0 nodes with two children.

   Inductive step: Assume that $P(1), P(2), ..., P(k)$ are true. We will argue that $P(k+1)$ is true. Indeed, consider any binary tree T with $k+1$ nodes and consider any leaf v in this tree. Let u be the parent of v in the tree. Consider the tree T' obtained by removing v and its edge from T. Note that T' is a binary tree with k nodes. Let $x_{T'}$ and x_T denote the number of nodes with two children in T' and T respectively. Let $y_{T'}$ and y_T denote the number of leaves in T' and T respectively. By induction hypothesis we know that:
   $$x_{T'} = y_{T'} - 1 \tag{1}$$

There are two cases to consider:

Case 1 <u>u has two children in T</u>: In this case, $x_{T'} = x_T - 1$ and $y_{T'} = y_T - 1$. This is because in T', u is a node with only one child and T' has one less leaf node. So, from (1), we get that $x_T = y_T - 1$.

Case 2 <u>u has only one child in T</u>: In this case, $x_{T'} = x_T$ and $y_{T'} = y_T$. This is because after removing v, u becomes a leaf node in T' and the number of nodes with two children does not change between T and T'. So, from (1), we get that $x_T = y_T - 1$.

This completes the proof.

3. (20 points) Given an undirected graph, we call a vertex critical if its removal disconnects the graph. Design an algorithm that finds all the critical vertices in a given graph. Give pseudocode, discuss running time, and give proof of correctness.

Solution: Critical vertices are more popularly known as *articulation points*. The edges in any undirected graph may be categorized into the following two classes with respect to the execution of DFS with starting vertex s:

1. *Tree edges*: These are the edges in the DFS tree obtained.
2. *Back edges*: These are the remaining edges. These are the edges the lead to a vertex that has already been explored. Note that for undirected graphs such a vertex is a *predecessor* (w.r.t. DFS tree) of the vertex being explored.

Let T be the DFS tree rooted at s that is obtained by running DFS starting from s. Before going further, let us make the following observations:

Claim 1: The leaves of T are not critical vertices.

Proof. This is trivial since removing a leaf does not disconnect the tree and hence does not disconnect the graph. □

Claim 2: The root s is a critical vertex iff it has at least two children in T.

Proof. If s has only one child, then removing s does not disconnect the graph. If s has more than 1 child, then removing s will disconnect the children since there cannot be an edge between a vertices that belong to different sub-trees of s. □

For determining whether an internal node is a critical node, we will have to do a bit more. While doing DFS, let us number the vertices in order of their discovery. That is, every time an unexplored vertex is visited, we number it with the next integer. Let us call this the *start time* for the vertex (since in some sense, it denotes the time at which this node was discovered). The following algorithm does this numbering:

```
- $time \leftarrow 0$
DFS-with-start($u$)
    - Mark $u$ as "explored" and set $start(v) \leftarrow$ ++$time$
    - While there is an "unexplored" neighbor $v$ of $u$:
        - DFS-with-start($v$)
```

Let us define a *back-path* for a vertex u to be a path that takes tree edges starting from u followed by a back edge. For each vertex v, we also calculate a number $low(v)$, which is the start time of a vertex with lowest value of *start* that is reachable using a back-path from v. The following algorithm calculates such a number.

```
- $time \leftarrow 0$
DFS-with-low($u$)
    - Mark $u$ as "explored", set $start(u) \leftarrow ++time$, and set $low(u) \leftarrow start(u)$
    - For every neighbor $v$ of $u$:
        - If $v$ is "unexplored"
            - $parent(v) \leftarrow u$
            - DFS-with-low($v$)
            - $low(u) \leftarrow \min(low(u), low(v))$
        - Else if $(parent(u) \neq v)$
            - $low(u) \leftarrow \min(low(u), start(v))$
```

Now, we can make the following claims with respect to the numbering defined above.

Claim 3: If an internal node u has a child v such that $low(v) \geq start(u)$, then v is a critical node.

Proof. This is true since there is no path from any vertex in the tree T_v rooted at v to any other vertex that is not v or a vertex in T_v. □

Claim 4: For any internal node u, if all its children have low value smaller than the start time of u, then u is not a critical node.

Proof. Consider the sub-tree T_v rooted at any child v of u. Since $low(v) < start(u)$, there is a path from any vertex in T_v to a predecessor of u in T. So removal of u does not disconnect the graph. □

Finally, we note that the above checks may be performed while computing these numbers. The algorithm is given below:

```
- $time \leftarrow 0$
OutputCritical(G)
  - Call DFS-with-critical($s$) for any starting vertex $s$
  - If there are at least two nodes $u, v$ such that $parent(u) = parent(v) = s$,
    then output "$s$ is a critical vertex"
DFS-with-critical($u$)
  - Mark $u$ as "explored", set $start(u) \leftarrow ++time$, and set $low(u) \leftarrow start(u)$
  - For every neighbor $v$ of $u$:
    - If $v$ is "unexplored"
      - $parent(v) \leftarrow u$
      - DFS-with-low($v$)
      - If ($low(v) \geq start(u)$), then output "$u$ is a critical vertex"
      - $low(u) \leftarrow \min(low(u), low(v))$
    - Else if ($parent(u) \neq v$)
      - $low(u) \leftarrow \min(low(u), start(v))$
```

The correctness follows from claims 3 and 4.

Running time: The running time is proportional to the running time of DFS. So, it is $O(n+m)$.

4. Consider two vertices s and t in a given graph. A pair of vertices (u, v) (different from s and t) are called bi-critical with respect to s and t if the removal of u and v from the graph disconnects s and t. Suppose we consider graphs where the shortest distance between s and t is strictly greater than $\lceil n/3 \rceil$. With respect to such graphs, answer the questions below:

   (a) (5 points) Prove or disprove the following statement:

   There exists a pair of vertices that are bi-critical with respect to s and t.

   Solution: Consider the layers obtained when doing BFS with starting vertex s. Let these layers be $L(0), L(1), ...$ We know that t is present in a layer with number $r > \lceil \frac{n}{3} \rceil$.

   Claim 1: There is at least one layer $1 \leq i < r$ such that $|L(i)| \leq 2$.

   Proof. For the sake of contradiction assume that for all $1 \leq j < r, |L(j)| > 2$. Then $L(0) \cup L(1) \cup ... \cup L(r) \geq 1 + \underbrace{3 + 3 + ... + 3}_{\geq \lceil n/3 \rceil \text{ terms}} + 1 \geq n + 2$ which is a contradiction. □

   Claim 2: Removing the vertices in any layer $L(i)$ for $i < r$, disconnects s and t.

   Proof. This follows from the property that all graph edges are between vertices in the same or adjacent layers. □

   (b) (5 points) Design an algorithm for finding a bi-critical pair of vertices with respect to given vertices s and t. Discuss running time. Proof of correctness is not required.

Solution: Given the above claims, the algorithm and its proof of correctness is simple:

```
BiCrititcal($G, s, t$)
  - Do $BFS(G, s)$ storing the layers.
  - Let $L(r)$ be the layer in which $t$ is present and let $L(i)$ be the layer s.t. $i < r$ and $|L(i)| \leq 2$.
  - If $|L(i)| = 2$, then output the vertices of $L(i)$. Otherwise, output the vertex in $L(i)$ and any
      other vertex other than $s$ and $t$.
```

Running time: The running time of the above algorithm is the same as that of BFS plus the time spent in checking the layers. This would be $O(n + m)$.

5. (20 points) A directed graph $G = (V, E)$ is called *one-way-connected* if for all pair of vertices u and v there is a path from vertex u to v **or** there is a path from vertex v to u. Note that the **or** in the previous statement is a logical OR and not XOR. Design an algorithm to check if a given graph is one-way-connected. Give pseudocode, discuss running time and give proof of correctness.

Solution: Let $V_1, V_2, ..., V_k$ be the vertex sets of the strongly connected components of the graph G. As discussed in class, there is an $O(n+m)$ algorithm to find the strongly connected components in any directed graph. We will construct another graph $G^{scc} = (V^{scc}, E^{scc})$ in which there is a node corresponding to each strongly connected component of G. Let the nodes in G^{scc} be denoted by $1, 2, ..., k$. As for the edges, $(i, j) \in E^{scc}$ iff there is a vertex $u \in V_i$ and a vertex $v \in V_j$ such that $(u, v) \in E$.

Claim 1: G^{scc} is a DAG.

Proof. Assume for the sake of contradiction that there is a cycle in G^{scc}. Let $i_1, i_2, ..., i_l, i_1$ denote this cycle in G^{scc}. Then note that this implies that for any pair of vertices $u, v \in V_{i_1} \cup V_{i_2} \cup ... \cup V_{i_l}$ there is a path from u to v and there is a path from v to u. We only prove existence of path in one direction, and the other direction follows by symmetry. Suppose $u \in V_{i_p}$ and $v \in V_{i_q}$ such that $p < q$. Note that since there is an edge from i_p to i_{p+1} in G^{scc}, this means that there is a directed edge from a vertex $x \in V_{i_p}$ to a vertex $y \in V_{i_{p+1}}$. Since V_{i_p} is a strongly connected component, this means that there is a path from u to x in G and this further means that there is a path from u to y in G. This further means that there is a path from u to all vertices in $V_{i_{p+1}}$ since $V_{i_{p+1}}$ is a strongly connected component. We can extend this argument to $V_{i_{p+2}}, ..., V_q$ proving that there is a path from u to any vertex in V_q and in particular path from u to v.

This gives us a contradiction since the above implies that $V_{i_1}, ..., V_{i_l}$ should form a single strongly connected component. $\square$

Next, we give the crucial claim that will give us our algorithm.

Claim 2: G is one-way connected iff G^{scc} has a unique topological ordering.

Proof. The proof follows from the proof of the next two claims. $\square$

Claim 2.1 If G^{scc} has a unique topological ordering, then G is one-way connected.

Proof. Let $i_1, i_2, ..., i_k$ be the unique topological ordering of G^{scc}. It follows that $(i_1, i_2), (i_2, i_3), ..., (i_{k-1}, i_k) \in E^{scc}$. Consider any two vertices $u, v \in V$. Let $u \in V_{i_p}$ and $v \in V_{i_q}$. WLOG assume that $p < q$. Then by the same arguments as that in the proof of Claim 1, we get that there is a path from u to v. □

Claim 2.2 If G^{scc} does not have a unique topological ordering, then G is not one-way connected.

Proof. Since G^{scc} does not have a unique topological ordering, it means that there are two orderings and two nodes i_p, i_q in V^{scc} such that i_p appears before i_q as per the first ordering and i_p appears later than i_q as per the second ordering. The former means that there is no path from i_q to i_p in G^{scc} and the latter means that there is no path from i_p to i_q in G^{scc}. This in turn implies that there is no path from any vertex in V_{i_p} to any vertex in V_{i_q} in G and vice versa. □

Now what remains to be done is to figure out a way to determine if there is a unique topological ordering of G^{scc}. This is easily done using the following algorithm.

```
OneWayConnected(G)
  - Use algorithm discussed in class to find the strongly connected components and
    construct the graph $G^{scc} = (V^{scc}, E^{scc})$
  - $G' \leftarrow G^{scc}$
  - For $i = 1$ to $|V^{scc}|$
      - If there are more than two vertices with in-degree 0 in $G'$
          - return("$G$ is not one-way connected")
      - Else let $v$ be the vertex in $G'$ with no incoming edges
      - Let $G''$ be the graph obtained from $G'$ by removing $v$ and its edges
      - $G' \leftarrow G''$
  - return("$G$ is one-way connected")
```

Proof of correctness: If the algorithm outputs "G is one-way connected", that means that at each step it found a unique vertex with 0 in-degree. This means that there is a directed edge from this unique vertex in the i^{th} step to the unique vertex in the $(i+1)^{th}$ step which further implies that there is a unique topological ordering.

If in any iteration, there are at least two vertices with 0 in-coming edges, then that means that these vertices can be in any relative order with respect to each other. Which means that unique ordering is not possible. Now the correctness follows from Claim 2.

Running time: Tarjan's algorithm takes $O(n + m)$ and so does constructing G^{scc}. The time taken within the for loop is proportional to the size of G^{scc} which is at most $O(n + m)$. So, the total running time of the algorithm is $O(n + m)$.

6. (20 points) Given a directed acyclic graph $G = (V, E)$ and a vertex u, design an algorithm that outputs all vertices $S \subseteq V$ such that for all $v \in S$, there is an even length simple path from u to v in G.
(A simple path is a path will all distinct vertices.)

   Give pseudocode, discuss running time, and give proof of correctness.

Solution:

Algorithm: Given a graph $G = (V, E)$, we construct a new graph $G' = (V', E')$ by "splitting" every vertex in V into two vertices. For a vertex $i \in V$ we create two copies of the vertex and call it i^{odd} and i^{even}. That is, for every vertex $i \in V$, V' has i^{odd} and i^{even}. As for edges, for every edge $(i, j) \in E$, we add two edges (i^{odd}, j^{even}) and (i^{even}, j^{odd}) to E'. Now, we just run $DFS(u^{even})$ on the graph G' and output all i such that i^{even} is visited while doing $DFS(u^{even})$.

The following two claims prove the correctness of our algorithm.

Claim 1: For every vertex v such that there is an even length path from u to v, the above algorithm will output v.

Proof. Let $u, i_1, i_2, ..., i_k, v$ be an even length path from u to v. Then k is an odd number. Given this, note that there is a path $u^{even}, i_1^{odd}, i_2^{even}, ..., i_k^{odd}, v^{even}$ in the graph G'. So, v^{even} is reachable from u_{even}. So our algorithm will output v. □

Claim 2: If our algorithm outputs a vertex v, then there is an even length path from u to v in G.

Proof. Our algorithms outputs v iff v^{even} is reachable from u^{even} in G'. Consider the path from u^{even} to v^{even} in the DFS tree obtained during execution of $DFS(u^{even})$ on G'. Let this path be denoted by $u^{even}, i_1^{\ell_1}, i_2^{\ell_2}, ..., i_k^{\ell_k}, v^{even}$. Note that $\ell_1 = odd$ since all directed edges are from vertices labeled even to vertices labeled odd or vertices labeled odd to vertices labeled even. This further implies that $\ell_2 = even, \ell_3 = odd, ..., \ell_k = odd$. This means that k is odd. This implies that there is an even length path in G from u to v since $u, i_1, i_2, ..., i_k, v$ is a path in G. □

Running time: The running time is proportional to the running time of DFS on a graph that is at most double the size of G. So, the running time is $O(n + m)$.