HW2 - CSCI 570

Q1. What is the tight bound on worst-case runtime performance of the procedure below? Give an explanation for your answer. (10 points)

```
int c = 0;
for(int k = 0; k <= log2n; k++)
        for(int j = 1; j <= 2^k; j++)
                c=c+1
return c
```

Lets assume $n = 8$

Outer Loop		Inner Loop	
Step 1:	$k=0$	1	$\log_2 n + 1$
Step 2:	$k=1$	2	
Step 3:	$k=2$	2^2	
Step 4:	$k=3$	2^3	

Total

$$= 2^0 + 2^1 + 2^2 + 2^3 + \dots 2^{\log_2 n}$$

$$= \sum_{k=0}^{\log_2 n} 2^k = 2^{(\log_2 n + 1)} - 1$$

$$= 2^{\log_2 n} \cdot 2 - 1$$

$$= n \times 2 - 1$$

∴ Tight Bound on worst case time should be $\Theta(n)$

Q2. Given an undirected graph G with n nodes and m edges, design an O(m+n) algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one. (10 points)

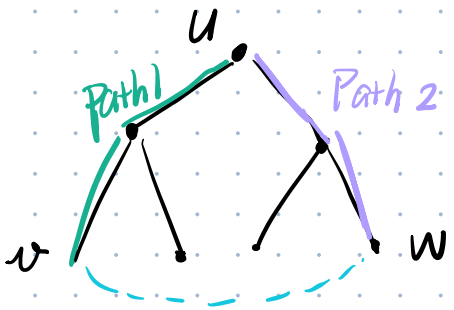
Assume G is connected.

1. Use BFS to find all nodes reachable from s (an arbitrary node) in G. $\Rightarrow O(m+n)$ Time.

When performing BFS, mark all the edges in (u,v) we traverse and create a hash set for this. Besides, record the father node for futher use. $\Rightarrow O(m)$

2. Check each edge in G is in T. If you find an edge in G that is not in T, this means G is not equal to T hence, G contains a cycle. The checking of each edge is O(1) due to hash set, so the total time for checking all edges in G is O(m)

3. For edge (u,v) that exists in G but not in T. We could find the unique path from u,v to their least common ancestor, let's say w. $\Rightarrow O(m)$



(v, w) is an edge in G but not in T

4. Output the cycle by concatenating the Paths.

All in all, it's an O(m+n) algorithm.

Q3. For each of the following indicate if f = O(g) or f = Θ(g) or f = Ω(g) (10 points)

	f(n)	g(n)
1	nlog(n)	$n^2\log(n^2)$
2	log(n)	log(log(5^n))
3	$n^{1/3}$	$(\log(n))^3$
4	2^n	2^{3n}
5	$n^4/\log(n)$	$n(\log(n))^4$

1. $f(n) = n\log(n)$. $g(n) = n^2\log(n^2) = 2n^2\log n$

$0 \le n\log n \le c(2n^2\log n)$ ✓ $\Rightarrow f = O(g)$

$0 \le c(2n^2\log n) \le n\log n$ ✗ $\Rightarrow f \neq \Omega(g)$

$f \neq \Theta(g)$

2. $f(n) = \log(n)$

$g(n) = \log(\log(5^n)) = \log(n\log 5)$ (constant)

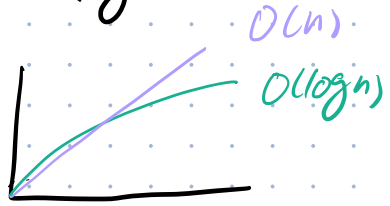
$0 \le c_1\log(n\log 5) \le \log(n) \le c_2\log(n\log 5)$ ✓

$f = \Theta(g)$, $f = O(g)$, $f = \Omega(g)$

3. $f(n) = n^{\frac{1}{3}}$

$g(n) = (\log(n))^3$

$0 \le c(\log(n))^3 \le n^{\frac{1}{3}}$ $f = \Omega(g)$



4. $f(n) = 2^n$

$g(n) = 2^{3n}$

$0 \le 2^n \le c\,2^{3n}$ $f = O(g)$

5. $f(n) = \frac{n^4}{\log n}$

$f(n) = n(\log(n))^4$

$0 \le c(n(\log n)^4) \le \frac{n^4}{\log n}$

n^4 grows faster than $\log n$

$\therefore$ $f = \Omega(g)$

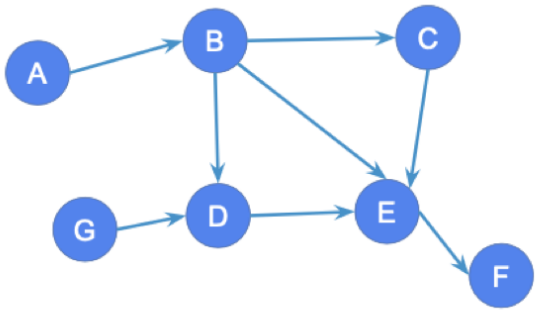
Q4. Indicate for each pair of expressions (A,B) in the table below, whether A is O, Ω, or Θ of B (in other words, whether A=O(B), A= Ω(B), or A= Θ (B)). Assume that k and C are positive constants. You can mark each box with Yes or No. No justification needed. (9 points)
(Note: log is base 2)

A	B	O	Ω	Θ
$n^3+\log(n)+n^2$	$C*n^3$	Yes	Yes	Yes
n^2	$C*n*2^{\log(n)}$	Yes	Yes	Yes
$(2^n)*(2^k)$	n^{2k}	NO	Yes	NO

$2^{\log_2 n} = n$

2^n grows faster than any polynomial function of n, including n^{2k}.

Q5. Find the total number of possible topological orderings in the following graph and list all of them (15 points)



A B C G D E F.
A B G C D E F.
A B G D C E F
A G B C D E F.
A G B D C E F.
G A B C D E F.
G A B D C E F

There are 7 possible topological orderings.

Q6. Given a directed graph with m edges and n nodes where every edge has weight as either 1 or 2, find the shortest path from a given source vertex 's' to a given destination vertex 't'. Expected time complexity is O(m+n). (8 points)

Assume G is connected.

Step 1: Transform the graph. For each edge with weight 2, we split it into two edges of 1, by introducing a new intermediate vertex.

$\Rightarrow O(m)$ for iterating over all edges.

Step 2: The new graph, in worst case, would have $n+m$ vertices and $m+m$ edges. (Every edge's weight is 2)

Apply BFS now to find the shortest path

$\Rightarrow O(n+m+2m) = O(n+3m)$

Overall, $O(m) + O(n+3m) \approx O(m+n)$

So we could find the shortest path in $O(m+n)$ time.

Q7. Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample. (12 points)

(a) $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

(b) $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

(c) $f_1(n)^2 = O\, g_1(n)^2$

(d) $\log 2\, f_1(n) = O(\log 2\, g_1(n))$

By definition

$f_1(n) = O(g_1(n)) \Rightarrow 0 \le f_1(n) \le C_1 g_1(n)$

$f_2(n) = O(g_2(n)) \Rightarrow 0 \le f_2(n) \le C_2 g_1(n)$

where $C_1, C_2 > 0$ exsits for n sufficiently large.

(a) True.

$f_1(n) \cdot f_2(n) \le C_1 g_1(n) \cdot C_2 g_2(n) = C_1 \cdot C_2\, g_1(n) g_2(n)$

$\therefore f_1(n) f_2(n) = O(g_1(n) g_2(n))$

(b) True

$f_1(n) + f_2(n) \le C_1 g_1(n) + C_2 g_2(n)$

Besides, $g_1(n) + g_2(n) \le 2\, Max(g_1(n), g_2(n))$

$\therefore f_1(n) + f_2(n) \le (C_1 + C_2) g_1(n) + (C_1 + C_2) g_2(n)$

$= (C_1 + C_2)(g_1(n) + g_2(n))$

$\le 2(C_1 + C_2)\, Max(g_1(n), g_2(n))$

(c) $\therefore f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

True

$f_1(n)^2 \le C_1^2 g_1(n)^2 \quad \therefore f_1(n^2) = O(g_1(n^2))$

(d) False

$\log_2(f_1(n)) \le \log_2(C_1 g_1(n)) = \log_2 C_1 + \log_2(g_1(n))$

$\le C \log_2(g_1(n))$

However, if $g_1(n) = 1$, $f_1(n) = 2$

$1 = \log_2 f_1(n) \not\le C \log_2(g_1(n)) = C \cdot 0 \quad \therefore 1 \ne O(0)$

Q8. Design an algorithm which, given a directed graph G = (V, E) and a particular edge e ∈ E, going from node u to node v determines whether G has a cycle containing e. The running time should be bounded by O(|V | + |E|). Explain why your algorithm runs in O(|V | + |E|) time. (8 points)

Step1: Remove Edge e from the graph (the edge from node u to node v) => O(1)

Step2: Perform DFS on the Graph. Starting with v. Check whether there is a path from v to u after removing edge e. If such a path exists, then adding the edge e forms a cycle containing e.

So, if a path from v to u is found in DFS, then there is a cycle in the graph containing edge e. Otherwise, no such cycle exists. => O(|V|+|E|)

Therefore, the time complexity of entire algorithm is O(|V|+|E|).

Q9. Solve Kleinberg and Tardos, **Chapter 3, Exercise 6.** (8 points)

6. We have a connected graph $G = (V,E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Proof by contradiction:

Assume An edge (v,w) in G but not in T.

① Because T is a BFS Tree, we know that the distance of v, w should no more than 1 layer.

② Because T is a DFS Tree at the same time. We know that there is (v,w) in G. and in DFS, when you start at a node and explore its neighbours. you follow a path deeper until you can go no further and then backtrack. Thus, if you want node v, and node w within 1 layer in DFS, you must have one node an ancestor of another node.

Therefore, edge should be in T, which contradicts the assumption, then G cannot contain any edges that do not belong to T.

Q10. Solve Kleinberg and Tardos, **Chapter 3, Exercise 9.** (10 points)

9. There's a natural intuition that two nodes that are far apart in a communication network-separated by many hops-have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.
Suppose that an n-node undirected graph $G = (V,E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m + n)$ to find such a node v.

By running a BFS from the node s. The BFS establishs a series of levels $L_1, L_2, \ldots L_{d-1}$, where node t appears in level d. Level L_1 is the set of nodes directly reachable from s. L_2 is the set of nodes in the next level reachable from s in addition to L_1.

Among these levels L_1 to L_{d-1}, at least one level contains only a single node. This is because if there are at least two nodes at each level, then there are at least n nodes by level $d-1$

$\therefore$ distance (# of levels) $> \frac{n}{2}$

$\therefore$ # of nodes $> 2 \times \frac{n}{2} = n$

Then, this single node is the node v we are looking for. Removing v will destroy all paths from s to t.

To prove this, consider the set of nodes:

$$X = \{s\} \cup L_1 \cup L_2 \cup \ldots \cup L_{i-1}, \text{ when } L_i = \{v\}$$

Since t is not in this set and all paths from s to t must cross each layer sequentially, they all must go through L_i at some point.

Since L_v contains only the single node v, all s-t paths must pass through this layer. removing v will disrupt every possible path from s to t.

This algorithm has a running time of $O(m+n)$ due to the linear nature of the BFS algorithm and the straightforward check for the special single-node layer.