Homework 2

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
```

Ans:
The given procedure involves nested loops, where outer loop runs for k=0 to $\log_2 n$ and the inner loop runs for j=1 to 2^k. The inner loop has a basic operation of c=c+1.
Hence, the total number of iterations would be $\sum_{0}^{\log 2n} \sum_{1}^{2^k} .1$

Inner loop:
c=c+1 operation is a simple sum of 1 repeated 2^k. Hence, the inner loop runs for:

$$\sum_{1}^{2^k} 1 = 2^k$$

Outer loop:
The outer loop runs from k=0 to $\log_2 n$. Hence, the number of iterations is:

$$\sum_{0}^{\log 2n} 2^k$$

This gives us a geometric series resulting in $2^{\log 2n+1} - 1$, which can be simplified to 2n-1.
Hence, the tight bound is O(n) as the dominant term is 2n after the constants are dropped.

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)

Ans:
To detect whether an undirected graph G contains a cycle, you can use DFS.

The algorithm used for this would be:

```
function has_cycle(graph):
   visited = set()
   parent = [-1] * len(graph)
   cycle_start, cycle_end = -1, -1
```

This algorithm uses DFS to traverse the undirected graph. The “has_cycle” function initializes data structures, including the visited set to keep track of visited nodes, the parent array to track parent nodes during traversal, and variables to store information about the detected cycle.

```
   function dfs(node, par):
      nonlocal cycle_start, cycle_end
```

```
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor == par:
                continue  # Skip the edge to the parent
            if neighbor in visited:
                # We found a cycle
                cycle_start = neighbor
                cycle_end = node
                return
            parent[neighbor] = node
            dfs(neighbor, node)
```

The dfs function is a recursive function that performs the DFS traversal. It marks nodes as visited, checks for cycles, and updates the parent array.

```
    for node in range(len(graph)):
        if node not in visited:
            dfs(node, -1)
    if cycle_start == -1:
        return "No cycle found"
    else:
        cycle = []
        while cycle_end != cycle_start:
            cycle.append(cycle_end)
            cycle_end = parent[cycle_end]
        cycle.append(cycle_start)
        return "Cycle found:", cycle
```

The main loop iterates through all nodes in the graph and calls the DFS function on unvisited nodes. The result is either "No cycle found" or "Cycle found" along with the nodes in the cycle if one exists.

Here, the runtime complexity of the algorithm is O(m+n) as the dominant function dfs traversal has a time complexity of O(m+n).

Q3. For each of the following indicate if f = O(g) or f = Θ(g) or f = Ώ(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	$n^4/\log(n)$	$n(\log(n))^4$

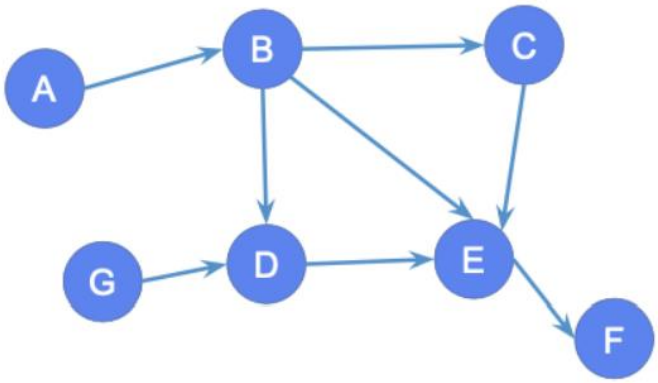
Ans:

1. f(n) = O(g(n))
2. f(n) = Θ(g(n))
3. f(n) = Ω (g(n))
4. f(n) = O(g(n))
5. f(n) = Ω(g(n))

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

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



Ans:

Total number of possible topological orderings in the graph are 7.

1. A G B C D E F
2. A G B D C E F
3. A B C G D E F
4. A B G C D E F
5. A B G D C E F
6. G A B D C E F
7. G A B C D E F

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)

Ans:
To have a time complexity of O(m+n), we have to use BFS.

Pseudo code:

```
function shortestPath(graph, s, t):
    n = number of nodes in the graph
    m = number of edges in the graph
    dist = array of size n
    dist[s] = 0
    queue = empty queue
    enqueue(queue, s)
```

This algorithm uses BFS to traverse the directed graph. Here, “n” represents the number of nodes, “m” is the number of edges, and “dist” is an array used to store the shortest distances from the source vertex 's' to all other vertices. Initially, all distances are set to infinity, except for the source vertex, which is set to 0. The algorithm uses a queue named queue to perform BFS traversal, and it starts by enqueueing the source vertex 's'.

```
    while queue is not empty:
        u = dequeue(queue)
        for each neighbor v of u:
            if weight(u, v) == 1:
                newDist = dist[u] + 1
            else:
                newDist = dist[u] + 2

            if newDist < dist[v]:
                dist[v] = newDist
                enqueue(queue, v)
    return dist[t]
```

Here, BFS takes place. In each iteration, a vertex 'u' is dequeued, and the algorithm explores its neighbors. For each neighbor 'v', it calculates the new distance newDist based on the weight of the edge (u, v). If newDist is smaller than the current distance dist[v], the algorithm updates dist[v] and enqueues 'v'. This process continues until all reachable vertices are visited. This continues till queue is empty. Finally, it returns the shortest distance from the source vertex 's' to the destination vertex 't'.

The algorithm has a time complexity of O(m+n) as we traverse through all the edges and nodes while performing BFS (The while loop runs for each node(O(n)) and the for loop inside the while loop iterates over the neighbors of each node, contributing O(m) to the time complexity (as each edge is considered).

Q7. Given functions f1 , f2 , g1 , g2 such that f1 (n) = O(g1 (n)) and f2 (n) = O(g2 (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) f1 (n) · f2 (n) = O (g1 (n) · g2 (n))
(b) f1 (n) + f2 (n) = O (max (g1 (n), g2 (n)))
(c) f1 (n) 2 = O g1 (n) 2
(d) log2 f1 (n) = O (log2 g1 (n))

Ans:
f1 (n) = O(g1 (n)) and f2 (n) = O(g2 (n)). Hence, there exists a c1,c2>0, where
1. f1(n) <= c1. g(n)
2. f2(n) <= c2. g(n).

(a) True
If we multiply statement 1 and 2, we get
f1(n). f2(n) <= c1. g1(n) * c2. g2(n)
f1(n). f2(n) <= (c1.c2). g1(n) * g2(n)
f1(n). f2(n) <= (C). (g1(n) * g2(n)) where C=c1.c2
This can be written as f1(n).f2(n) = O(g1(n). g2(n))
Hence proved that f1 (n) · f2 (n) = O (g1 (n) · g2 (n))

(b) True
If we add statement 1 and 2, we get
f1(n) + f2(n) <= c1. g1(n) + c2. g2(n)
f1(n) + f2(n) <= (c1+c2). (g1(n) + g2(n))
We take the max of g1(n) and g2(n) to ensure that the upper bound captures the growth of both the functions.
f1(n) + f2(n) <= (C). O(max(g1(n), g2(n))) where C = c1 + c2
This can be written as f1(n) + f2(n) = O(max(g1(n), g2(n)))
Hence proved that f1 (n) + f2 (n) = O (max(g1 (n), g2 (n)))

(c) True
If we take the square of statement 1,
$f1(n)^2 <= c1^2. g1(n)^2$
$f1(n)^2 <= C. g1(n)^2$ where $C = c1^2$
This can be written as $f1(n)^2 = O(g1(n)^2)$
Hence proved that $f1(n)^2 = O(g1(n)^2)$

(d) False
Here, if we take f1(n) = 2 and g1(n) = 1, then after taking log on both sides,
log(f1(n) = 2) = 1 and log(g1(n)) = 0
Hence proved that log2 f1 (n) != O (log2 g1 (n))

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)

Ans:
To detect whether a directed graph G = (V, E) contains a cycle that includes a particular edge e going from node u to node v, we can use DFS.

The algorithm used for this would be:

```
function hasCycleContainingEdge(graph, u, v, e):
    n = number of nodes in the graph
    m = number of edges in the graph
    visited = array of size n initialized to False
    hasCycle = False
```

This algorithm uses DFS to traverse the directed graph. It initializes necessary variables, including a boolean array “visited” to keep track of visited nodes and a boolean “hasCycle” variable to indicate whether a cycle is found.

```
    for each node x in the graph:
        if not visited[x]:
            if dfs(x, graph, visited, e):
                hasCycle = True
                break

    return hasCycle
```

For each unvisited node, it calls the DFS function to explore the graph, starting from that node. The DFS function “dfs” performs the traversal. It marks the current node as visited, then iterates through its neighbors.

```
function dfs(x, graph, visited, e):
    visited[x] = True

    for each neighbor y of x in graph[x]:
        if (x, y) == e:
            continue  # Skip the edge e to avoid revisiting it

        if not visited[y]:
            if dfs(y, graph, visited, e):
                return True
        else:
            return True  # Cycle detected

    return False
```

If the edge being checked (e) is encountered during the traversal, it is skipped to avoid revisiting it. If any cycle containing the edge e is found, the algorithm returns True; otherwise, it returns False.

During the DFS traversal, the algorithm skips the edge being checked (e). This optimization prevents unnecessary revisits to the specified edge.
Hence, the runtime is O(|V|+|E|). Each node and each edge is visited at most once during the DFS traversal, contributing O(|V| + |E|) to the time complexity.

Q9. We have a connected graph G = (V, E), and a specific vertex u ∈ 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.)

Ans:
We can prove this by contradiction.
Suppose there exists an edge (x, y) in G, such that it does not belong to the tree T formed by BFS and DFS rooted at u. Since T is the tree obtained by both DFS and BFS, there must be a path from u to x and a path from u to y in T.
Let z be the first common ancestor of x and y along their paths in T. Since T is a tree, there is only one path from z to x and one path from z to y in T.
Let's assume that the path from z to x is longer than the path from z to y. Now, consider the edge (a, b) in the path from z to x such that a is an ancestor of y in T. Since the path from z to x is longer, there must be a node n on the path from z to x such that n is not in the path from z to y. In both DFS and BFS, the node n would have been visited before y. However, since n is not in the path from z to y, the edge (n, y) cannot be in T.
This contradicts the assumption that T is a DFS and BFS tree rooted at u. Therefore, the assumption that there exists an edge (x, y) in G not belonging to T is false.
Hence, we can conclude that G = T.

Q10. 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.

Ans:

If an 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, then there must exist some node v (not equal to either s or t) such that deleting v from G destroys all s-t paths.

Let d be the distance between nodes s and t in G, where d is greater than n/2 and let P be the shortest path from s to t in G, consisting of at most d edges.

Split P into 2 paths: Path P1 from s to some intermediate node x, and Path P2 from x to t. Since the distance between s and t is strictly greater than n/2, there exists a node x such that both P1 and P2 have at least d/2 edges each.

Consider the node v that is the last node on Path P1, i.e., the node right before x on the path. By deleting v from G, all edges incident to v are removed. Since v is the last node on Path P1, deleting it disconnects P1 from the rest of the graph. Consequently, there is no longer a path from s to t in the modified graph (G - {v}) as Path P1 is broken.

Hence, deleting v from G destroys all s-t paths, and it is important to note that v is not equal to either s or t, serving as an intermediate node on the shortest path P1. This completes the proof, illustrating the existence of a node v that, when deleted, disrupts all paths from s to t in the graph G.