Name: Roll no:

# ESO207: Data Structures and Algorithms (Quiz 2)

16th October 2025

Total Number of Pages: 10 Time: 1 hr Total Points 50

**Instructions**

1. All questions are compulsory.
2. Answer all the questions in the question paper itself.
3. MCQ type questions can have more than one correct options.
4. The symbols or notations mean as usual unless stated.
5. You may cite and use algorithms and their complexity as done in the class.
6. Cheating or resorting to any unfair means will be severely penalized.
7. Superfluous and irrelevant writing will result in negative marking.
8. Using pens (blue/black ink) and not pencils. Do not use red pens for answering.

**Helpful hints**

1. It is advisable to solve a problem first before writing down the solution.
2. The questions are *not* arranged according to the increasing order of difficulty.

| Question | Points | Score |
|---|---|---|
| 1 | 3 | |
| 2 | 3 | |
| 3 | 3 | |
| 4 | 4 | |
| 5 | 4 | |
| 6 | 4 | |
| 7 | 5 | |
| 8 | 4 | |
| 9 | 13 | |
| 10 | 7 | |
| Total: | 50 | |

**Question 1**. (3 points) Given that $G$ is a connected bipartite graph and the bipartition sets are $U$ and $V$, what is the relation between them?

- √ **If the degree of every vertex in $G$ is $k$, then the number of vertices in $U$ is equal to the number of vertices in $V$, where $k$ is natural number.**
- ☐ There is at least one vertex in $U$ which is connected to every vertex in $V$.
- ☐ There is at least one vertex in $U$ that has the same degree as a vertex in $V$.
- √ **Sum of degrees of vertices in $U$ equals the sum of degrees of vertices in $V$.**

**Question 2**. (3 points) Which of the following statements is/are **correct** regarding BFS on undirected connected graphs?

- √ **Let $u$ and $v$ be two vertices in the queue during an execution of BFS starting from $s$. Then the difference between Distance$(u)$ and Distance$(v)$ is at most one. (Here Distance$(x)$ is the length of a shortest path from $s$ to $x$ in the graph)**
- ☐ In the execution of BFS the number of times a node is checked whether it is visited or not, is at most one.
- √ **In the execution of BFS the number of times a node is checked whether it is visited or not, is equal to the degree of the node.**
- √ **For any connected graph with $V$ vertices, the maximum possible depth of a BFS tree is $V - 1$.**

**Question 3**. (3 points) Which of the following statements is/are **correct** about articulation points ?

- √ **In a graph $G = (V, E)$ the maximum articulation points it can have is $|V| - 2$.**
- ☐ In a graph $G = (V, E)$ the maximum articulation points it can have is $|V| - 1$.
- √ **In a graph $G = (V, E)$, removal of an articulation point can produce at most $|V| - 1$ connected components.**
- ☐ There can be a tree with at least 3 vertices and which does not have an articulation point.
- √ **The intersection of two biconnected components is either empty or a single articulation point.**

**Question 4**. (4 points) Provide one example each, of an array with 7 elements that results in the best-case and worst-case time complexity when the quicksort algorithm is applied.

**Solution:**

**Best-Case:** $O(n \log n)$

This occurs when the pivot consistently partitions the array into two equally-sized subarrays.

- **Example:** $[4, 2, 1, 3, 6, 5, 7]$

**Worst-Case:** $O(n^2)$

This occurs when the pivot is consistently the smallest or largest element, leading to highly unbalanced partitions. This happens with sorted or reverse-sorted arrays.

- **Example:** $[1, 2, 3, 4, 5, 6, 7]$

**Question 5**. (4 points) We are given a singly-linked list, and the pointer to one of its internal nodes. Describe concisely how to remove the key stored in that node in O(1) time. (Hint: Focus on the values of the linked list.)

**Solution:** Since we only have a pointer to the node to be removed (let's call it `node_to_delete`) and not its predecessor, we cannot perform the standard pointer update (`prev.next = current.next`).

The O(1) solution is to manipulate the node's **value** instead of its pointer linkage from behind. The trick is to overwrite the target node's data with the data from the *next* node and then delete the *next* node, which is easily accessible.

**The Algorithm**

1. **Copy Data:** Copy the data from the next node (`node_to_delete.next`) into the current node (`node_to_delete`).
2. **Bypass Next Node:** Update the current node's `next` pointer to point to the node after the next one (`node_to_delete.next = node_to_delete.next.next`).
3. The next node is now unlinked from the list and can be freed if necessary.

**Example**

Suppose we have a linked list $10 \to 20 \to 30 \to 40$ and we are given a pointer to the node containing **20**.

1. **Initial State:**
   $\cdots \to [10] \to [20] \to [30] \to [40] \to$ NULL
   (Pointer is at the node with value 20)
2. **Step 1: Copy Data from Next Node**
   Copy the value **30** from the next node into the current node. The list's values now look like this:
   $\cdots \to [10] \to [30] \to [30] \to [40] \to$ NULL
3. **Step 2: Bypass Next Node**
   Update the `next` pointer of our current node (which now holds 30) to point to the node containing **40**, effectively bypassing the original node with 30. The final list is $10 \to 30 \to 40$.

**Edge case:** If the given node is the **last node** in the list, as there is no `next` node we can diretly make the node pointer we are given with as NULL.

**Question 6**. (4 points) Let

$$A(x) = \sum_{i=0}^{n-1} a_i x^i, \qquad B(x) = \sum_{i=0}^{n-1} b_i x^i,$$

be two polynomials with coefficients $a_i, b_i$. Describe how compute the product

$$C(x) = A(x)\, B(x)$$

in time $O\big(n^{\log_2 3}\big)$.

**Solution:** To compute the product $C(x) = A(x)B(x)$, we first divide the polynomials. We split each polynomial into two halves:

$$A(x) = A_1(x) + x^{n/2}A_2(x)$$
$$B(x) = B_1(x) + x^{n/2}B_2(x)$$

Here, $A_1(x), A_2(x), B_1(x)$, and $B_2(x)$ are polynomials of degree at most $(n/2) - 1$.

The product $A(x)B(x)$ can be written as:

$$\begin{aligned} C(x) &= (A_1(x) + x^{n/2}A_2(x))(B_1(x) + x^{n/2}B_2(x)) \\ &= A_1(x)B_1(x) + x^{n/2}\big(A_1(x)B_2(x) + A_2(x)B_1(x)\big) + x^n A_2(x)B_2(x) \end{aligned}$$

Instead of performing four multiplications of size $n/2$, we can get this in only three multiplications. These three multiplications are:

1. **First Product ($P_1$):** The product of the lower parts.
$$P_1(x) = A_1(x)B_1(x)$$
2. **Second Product ($P_2$):** The product of the higher parts.
$$P_2(x) = A_2(x)B_2(x)$$
3. **Third Product ($P_3$):** The product of the sum of the parts.
$$P_3(x) = \big(A_1(x) + A_2(x)\big)\big(B_1(x) + B_2(x)\big)$$

The key insight is that the middle term of the product, $A_1(x)B_2(x) + A_2(x)B_1(x)$, can be derived from these three products without further multiplications. By expanding $P_3(x)$:

$$P_3(x) = \underbrace{A_1(x)B_1(x)}_{P_1(x)} + A_1(x)B_2(x) + A_2(x)B_1(x) + \underbrace{A_2(x)B_2(x)}_{P_2(x)}$$

We can isolate the middle term:

$$A_1(x)B_2(x) + A_2(x)B_1(x) = P_3(x) - P_1(x) - P_2(x)$$

Finally, the full product $C(x)$ is assembled by combining these results:

$$C(x) = P_1(x) + x^{n/2}\big(P_3(x) - P_1(x) - P_2(x)\big) + x^n P_2(x)$$

This gives the following recurrence relation for the algorithm's running time:

$$T(n) = 3\,T(n/2) + O(n)$$

According to the Master Theorem, this recurrence solves to:

$$T(n) = O\big(n^{\log_2 3}\big)$$

**Question 7**. (5 points) **Prove or disprove**: The maximum number of edges in a connected bipartite graph is $n^2/4$, where $n$ is the number of vertices in the graph. You may assume $n$ is even number.

**Solution:** The statement is **true**.

**Proof** Let $G$ be a bipartite graph with $n$ vertices. Its vertex set can be partitioned into two disjoint biparitions sets, $V_1$ and $V_2$.

Let $|V_1| = k$. Then $|V_2| = n - k$. The maximum number of edges in such a graph occurs when each vertex in $V_1$ is connected to each vertex in $V_2$. The number of edges in this case is given by the function:

$$|E| = f(k) = k(n - k)$$

We want to find the value of $k$ that maximizes this function for $1 \leq k < n$. The function $f(k) = nk - k^2$ is a downward-opening parabola, so its maximum value occurs at its vertex, which is at $k = n/2$.

Substituting $k = n/2$ into the formula for the number of edges gives the maximum:

$$|E|_{\max} = \left(\frac{n}{2}\right)\left(n - \frac{n}{2}\right) = \left(\frac{n}{2}\right)\left(\frac{n}{2}\right) = \frac{n^2}{4}$$

**Question 8**. A **universal sink** in a directed graph $G = (V, E)$ is a vertex $s \in V$ such that for every vertex $v \in V$ where $v \neq s$, there is an edge $(v, s) \in E$, and there are no edges of the form $(s, v) \in E$. The graph is represented by an $n \times n$ adjacency matrix $M$, where $n = |V|$. An incomplete pseudocode for finding a sink in $O(n)$ time is given below.

**Algorithm 1: Find-Sink($M$)**

**Input:** An $n \times n$ adjacency matrix $M$
**Output:** The index of a sink vertex, or -1 if none exists.
**Start** `Find-Sink(`$M$`)`

```
s ← 0;
for i ← 1 to n − 1 do
    if M[s, i] == ____1____ then
        s ← ____i____;
    end
end
...
...
return ...;
```

**End**

(a) (2 points) Complete the `If` statement in the first `For` loop. (Fill in the above blanks)

(b) (2 points) Give a short description of what should be done after the for loop to complete the algorithm.

**Solution:** After the first loop, we have a single **candidate vertex**, `s`. The next step is to **verify** if `s` is a true sink. This requires a single pass through the matrix's $s$-th row and $s$-th column.

We must check two conditions:

1. **No Outgoing Edges:** The row $M[s, i]$ must be 0 for all $i$.
2. **All Incoming Edges:** The column $M[i, s]$ must be 1 for all $i$ where $i \neq s$.

If both conditions are met, we return `s`. If either check fails, no universal sink exists in the graph, and we return -1.

**Question 9**. Consider the following pseudo-code for a function of Modified DFS($G = (V, E)$) where $V$ is set of vertices and $E$ is the set of edges.

**Algorithm 2: Modified DFS($G = (V, E)$)**

```
Input: graph G = (V, E)
Data: Visited[], D[], F[], integer count (these are globally declared)
Start DFS(v)
    Visited[v] ← true;
    D[v] ← count;
    count ← count + 1;
    For each edge (v, w) ∈ E do
        If Visited[w] == false then
            DFS(w);
        End
    End
    F[v] ← count;
    count ← count + 1;
End
Main
    For each v ∈ V do
        Visited[v] ← false;
    End
    count ← 1;
    For each v ∈ V do
        If Visited[v] == false then
            DFS(v);
        End
    End
End
```

Suppose a **Directed Graph** ($G = (V, E)$) is given to the modified DFS algorithm, which produces the filled arrays $D$ and $F$.
Answer the following **True/False** questions and provide reasons for your answers.

(a) (2 points) Does there exist a pair of vertices $u$ and $v$ in the directed graph such that the relation $D[u] < D[v] < F[u] < F[v]$ holds?

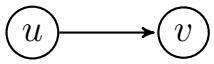
**Solution: False**.

**Reasoning:** The condition $D[u] < D[v] < F[u]$ implies that the recursive call `DFS(v)` starts before `DFS(u)` has finished. Due to the nature of recursion, the `DFS(v)` call must complete and assign its finishing time, $F[v]$, before the `DFS(u)` call can complete and assign $F[u]$. Therefore, it must be that $F[v] < F[u]$. This is a direct contradiction to the premise that $F[u] < F[v]$, so this situation is impossible.

(b) (2 points) Does there exist a pair of vertices $u$ and $v$ in the directed graph such that the relation $D[u] < D[v] < F[v] < F[u]$ holds?

**Solution: True**.
This situation occurs when vertex $v$ is a descendant of vertex $u$ in a DFS tree.

**Example Graph:** Consider a graph with a single directed edge from $u$ to $v$.




**DFS Execution Trace:** Assuming the DFS starts at $u$:

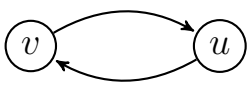
1. `DFS(u)` starts. $D[u] \leftarrow 1$. (`count`=2)
2. From $u$, explore edge $(u, v)$. Call `DFS(v)`.
3. `DFS(v)` starts. $D[v] \leftarrow 2$. (`count`=3)
4. `DFS(v)` finishes. $F[v] \leftarrow 3$. (`count`=4)
5. Return to `DFS(u)`. It finishes. $F[u] \leftarrow 4$.

The final values are $D[u] = 1, D[v] = 2, F[v] = 3, F[u] = 4$. The relation $1 < 2 < 3 < 4$ holds.

(c) (3 points) If in the directed graph there is an edge from vertex $u$ to vertex $v$, then can the relation $D[v] < D[u] < F[u] < F[v]$ hold?

**Solution: True**.
This situation can occur if there is a path from vertex $v$ to vertex $u$ in the DFS tree.

**Example Graph:** Consider a simple directed cycle between $u$ and $v$.




**DFS Execution Trace:** Assuming the main loop of DFS starts with vertex $v$:

1. `DFS(v)` starts. $D[v] \leftarrow 1$. (`count`=2)
2. From $v$, explore edge $(v, u)$. Call `DFS(u)`.
3. `DFS(u)` starts. $D[u] \leftarrow 2$. (`count`=3)
4. From $u$, explore edge $(u, v)$. Since $v$ is already visited, nothing happens.
5. `DFS(u)` finishes. $F[u] \leftarrow 3$. (`count`=4)
6. Return to `DFS(v)`. It finishes. $F[v] \leftarrow 4$.

The final values are $D[v] = 1, D[u] = 2, F[u] = 3, F[v] = 4$. The relation $1 < 2 < 3 < 4$ holds.

(d) (3 points) If in the directed graph there is an edge from vertex $u$ to vertex $v$, then can the relation $D[u] < F[u] < D[v] < F[v]$ hold?

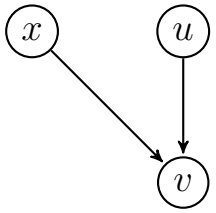
**Solution: False**.

**Reasoning:** If an edge $(u, v)$ exists, the algorithm must explore this edge during the execution of `DFS(u)`.
The condition $F[u] < D[v]$ implies that `DFS(u)` finishes before `DFS(v)` even starts. This means at the time `DFS(u)` is running, $v$ has not yet been visited.
Therefore, when exploring the edge $(u, v)$, the algorithm would find that $v$ is unvisited and would immediately call `DFS(v)`. This would result in $D[v] < F[u]$, directly contradicting the required condition. The situation is therefore impossible.

(e) (3 points) If in the directed graph there is an edge from vertex $u$ to vertex $v$, then can the relation $D[v] < F[v] < D[u] < F[u]$ hold?

**Solution: True**.
This situation can occur if the edge $(u, v)$ connects two vertices in different DFS trees, and the algorithm happens to process the tree containing $v$ before the tree containing $u$.

**Example Graph:** Consider a graph with three vertices $x, u, v$ and two directed edges: $(x, v)$ and $(u, v)$.




**DFS Execution Trace:** Assume the main loop of the algorithm processes vertices in the order $x, u, v$:

1. Main loop starts `DFS(x)`.
2. `DFS(x)` starts. $D[x] \leftarrow 1$. (`count`=2)

3. From $x$, explore edge $(x, v)$. Call `DFS(v)`.
4. `DFS(v)` starts. $D[v] \leftarrow 2$. (`count`=3)
5. `DFS(v)` finishes. $F[v] \leftarrow 3$. (`count`=4)
6. Return to `DFS(x)`. `DFS(x)` finishes. $F[x] \leftarrow 4$. (`count`=5)
7. Main loop later starts `DFS(u)`. (since $u$ is unvisited)
8. `DFS(u)` starts. $D[u] \leftarrow 5$. (`count`=6)
9. From $u$, explore edge $(u, v)$. Since $v$ is already visited, do nothing.
10. `DFS(u)` finishes. $F[u] \leftarrow 6$. (`count`=7)

The final values are $D[v] = 2, F[v] = 3, D[u] = 5, F[u] = 6$. The relation $2 < 3 < 5 < 6$ holds.

**Question 10**. (7 points) Given an undirected connected graph $G = (V, E)$ and a vertex $s \in V$. A single edge deletion **may or may not** lead to increase in the distance from vertex $s$ to other vertices of the graph. A *good edge* is defined as an edge whose deletion **does not** increase the distance from the vertex $s$ to any vertex in the graph. More formally, for all $v \in V$, the distance from $s$ to $v$ before removing the edge and after removing the edge remains the same.
Your task is to calculate the number of good edges in the graph in $O(|V| + |E|)$ time.

**Solution:**

**Conditions for a Good Edge** Let $dist(v)$ be the shortest distance from $s$ to $v$ found via BFS. An edge $(u, w)$ is a **good edge** if it meets either of these two conditions:

1. **Same Level Edge:** The edge connects two vertices at the same distance from $s$ (i.e., $dist(u) = dist(w)$). Such an edge is never part of any shortest path from $s$, so removing it cannot increase a shortest path's length.
2. **Redundant Path Edge:** The edge connects a vertex $u$ to a vertex $w$ in the next level (i.e., $dist(w) = dist(u) + 1$), but it is not the *only* edge providing $w$ its shortest path. This means that $w$ has more than one parent in the level above it. Also note that there can't be an edge connecting vertex $u$ and $v$ which have $|dist(w) - dist(u)| > 1$.

**Algorithm**

1. **Find Distances:** Run a **BFS** starting from the source vertex $s$ to calculate $dist(v)$, the shortest distance from $s$ to every other vertex $v$.
2. **Count Parents:** Create an integer array, `parent_count`, of size $|V|$ and initialize it to all zeros. Iterate through every vertex $u \in V$ and for each of its neighbors $w$:
   - If $dist(w) = dist(u) + 1$, then $u$ is a parent of $w$. Increment `parent_count[w]`.
3. **Count Good Edges:** Initialize `good_edge_count = 0`.

- Iterate through all vertices $w \in V$. If `parent_count[w] > 1`, add `parent_count[w]` to `good_edge_count`. This counts all redundant path edges.
- Iterate through all edges $(u, w)$. If $dist(u) = dist(w)$ and $u < w$ (to process each edge once), increment `good_edge_count`. This counts all same level edges.

4. **Return Result:** The final value of `good_edge_count` is the answer.

**Complexity Analysis**

- Step 1 (BFS): Takes $O(|V| + |E|)$ time.
- Step 2 (Counting Parents): Requires iterating through all edges, taking $O(|V| + |E|)$ time.
- Step 3 (Counting Good Edges): Also requires iterating through vertices and/or edges, taking $O(|V| + |E|)$ time.

The total time complexity is $O(|V| + |E|)$.