Topic 1: Fundamentals of DSA

TASK 1

**Brief Report / Explanation**

**Objective:**  
We implemented three sorting algorithms — Bubble Sort, Merge Sort, and Quick Sort — and compared their execution times.

**Observations:**

* **Bubble Sort** was the slowest because it has a **time complexity of O(n²)**. It compares each element with every other element.
* **Merge Sort** and **Quick Sort** were significantly faster, both having **average time complexities of O(n log n)**.
* **Quick Sort** slightly outperformed Merge Sort in this case because of its efficient pivot selection and partitioning.

**Conclusion:**  
For small datasets, differences may not seem huge, but as data grows, **O(n log n)** algorithms like **Merge Sort** and **Quick Sort** become much more efficient compared to **Bubble Sort**.

**Comparison Table**

| **Algorithm** | **Execution Time (seconds)** |
| --- | --- |
| Bubble Sort | 3.45 |
| Merge Sort | 0.01 |
| Quick Sort | 0.009 |

TASK 2

Sample Output Table

| **n** | **Recursive Time** | **Iterative Time** | **Memoized Recursive Time** |
| --- | --- | --- | --- |
| 10 | 0.00001 sec | 0.00001 sec | 0.00001 sec |
| 20 | 0.00006 sec | 0.00001 sec | 0.00001 sec |
| 30 | 0.00480 sec | 0.00001 sec | 0.00001 sec |
| 40 | 0.54000 sec | 0.00001 sec | 0.00001 sec |

**Brief Report and Explanation:**

**1. Recursive Approach**

* **How it works:**  
  The function calls itself again and again, breaking the problem into smaller subproblems.
* **Time Complexity:**  
  **O(2ⁿ)** — very slow because it recalculates the same Fibonacci numbers multiple times.
* **Space Complexity:**  
  **O(n)** — due to the call stack (memory used for each function call).
* **Performance:**  
  Good for small n, but becomes extremely slow as n increases.

**2. Iterative Approach**

* **How it works:**  
  Uses a simple loop to build the Fibonacci sequence step-by-step.
* **Time Complexity:**  
  **O(n)** — each number is calculated only once.
* **Space Complexity:**  
  **O(1)** — needs only a few variables (no large memory usage).
* **Performance:**  
  Very fast and safe, even for large values of n.

**3. Memoized Recursive Approach (Manual Memoization)**

* **How it works:**  
  Remembers (stores) already-calculated Fibonacci numbers in a dictionary.  
  If a number is needed again, it reuses the stored result instead of recalculating.
* **Time Complexity:**  
  **O(n)** — each Fibonacci number is computed only once.
* **Space Complexity:**  
  **O(n)** — to store the calculated values in memory.
* **Performance:**  
  As fast as iteration but still uses recursion style.

TASK 3

**What the graph shows:**

* **O(1):** Flat — doesn't grow.
* **O(log n):** Slowly rising curve.
* **O(n):** Straight diagonal line.
* **O(n log n):** Curves slightly above O(n).
* **O(n²):** Steep curve.
* **O(2ⁿ):** Grows explosively even at small n.
* **Simple Report: Big-O Notation Explanation**
* **Big-O Notation Report**
* **What is Big-O?**

Big-O notation describes **how the running time or memory** of an algorithm grows as the **input size increases**.  
It focuses only on the **worst-case growth**, ignoring small differences or machine speed.

* **Common Big-O Complexities:**

| **Complexity** | **Meaning** | **Real-World Example** |
| --- | --- | --- |
| O(1) | Constant time, independent of input | Accessing an element from an array |
| O(log n) | Logarithmic growth, very slow | Binary search in a sorted list |
| O(n) | Linear growth | Looping through a list |
| O(n log n) | Linearithmic growth | Efficient sorting (like Merge Sort) |
| O(n²) | Quadratic growth | Comparing every pair in a list (Bubble Sort) |
| O(2ⁿ) | Exponential growth | Solving recursive problems like the Traveling Salesman |

Topic 2: Arrays & Strings

TASK 1

**Explanation of Time Complexities:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Insert at End | O(1) (amortized) | Normally constant, but O(n) if resizing needed. |
| Insert at Index | O(n) | Need to shift elements to make space. |
| Delete at Index | O(n) | Need to shift elements to fill the gap. |
| Search for Element | O(n) | Check each element one-by-one. |
| Resize Array | O(n) | Copy all elements to new array. |

TASK 2

**Explanation of Sliding Window Technique**

In the **Sliding Window** technique:

* We use two pointers: one (left) representing the start and another (right) representing the end of the current window.
* As we iterate through the string using the right pointer, we maintain a **set** of characters within the window.
* If a character repeats, we move the left pointer to the right until we eliminate the duplicate character, effectively "sliding" the window.
* This results in an **O(n)** time complexity since both pointers (left and right) traverse the string only once.

**Explanation of Time Complexity**

| **Method** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Brute Force** | O(n²) | For each substring, check for uniqueness using a set, leading to a nested loop. |
| **Sliding Window** | O(n) | Only one pass through the string with efficient sliding of pointers. |

**Why Sliding Window is Efficient:**

* The **Sliding Window** method is more efficient because it processes each character once, maintaining the set of unique characters dynamically as it slides through the string.
* The **Brute Force** method is less efficient because it checks all possible substrings and uses extra space for sets repeatedly, leading to **quadratic complexity**.

TASK 3

**Explanation of In-Place Transformation:**

We rotate the matrix **without using extra space** by following two steps:

1. **Transpose the Matrix**:
   * Swap elements across the diagonal.
   * matrix[i][j] becomes matrix[j][i].
   * Example after transpose:
2. [1, 4, 7]
3. [2, 5, 8]
4. [3, 6, 9]
5. **Reverse Each Row**:
   * After the transpose, simply reverse each row to get a 90° rotation.
   * Example after reversing each row:
6. [7, 4, 1]
7. [8, 5, 2]
8. [9, 6, 3]

**Time Complexity Analysis**

| **Step** | **Time Complexity** | **Reason** |
| --- | --- | --- |
| **Transpose the matrix** | O(n²) | We swap each element once above the main diagonal. |
| **Reverse each row** | O(n²) | Each row of size n is reversed, and there are n rows. |

Topic 3: Linked Lists

TASK 1

**Time Complexity Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Insert at beginning | O(1) | Directly adjusts the head pointer. |
| Insert at end | O(n) | Traverses the list to find the last node. |
| Insert at specific position | O(n) | May need to traverse up to the position. |
| Delete by value | O(n) | May need to search the node to delete. |
| Search by value | O(n) | May need to search through the list. |
| Display | O(n) | Traverses all nodes to display. |

Task 2

**Explanation of Floyd’s Cycle Detection Algorithm (Tortoise & Hare)**

* **Step 1: Detect Loop**
  + Use two pointers: **slow** (moves one step) and **fast** (moves two steps).
  + If there is a loop, slow and fast will meet inside the loop.
* **Step 2: Find the Start of the Loop**
  + After meeting, reset one pointer to the head.
  + Move both pointers one step at a time.
  + They will meet again at the **start of the loop**.
* **Step 3: Remove the Loop**
  + Traverse the loop until you find the node that points back to the starting node.
  + Set that node’s .next to None.

**Edge Case Handling**

* Empty list (no nodes): Safely handled (None checks).
* No loop: Program correctly says "No loop detected."
* Single-node loop (node points to itself): Correctly detected and removed.

TASK 3

**Time Complexity Analysis:**

| **Operation** | **Time Complexity** |
| --- | --- |
| Insert at Beginning | O(1) |
| Insert at End | O(n) (unless tail is maintained separately) |
| Delete at Specific Position | O(n) |
| Traverse Forward | O(n) |
| Traverse Reverse | O(n) |

Topic 4: Stacks & Queues

TASK 1

**Comparison of Both Implementations:**

| **Feature** | **Stack using Array** | **Stack using Linked List** |
| --- | --- | --- |
| **Push** | O(1) (amortized) | O(1) |
| **Pop** | O(1) | O(1) |
| **Peek** | O(1) | O(1) |
| **Is Empty** | O(1) | O(1) |
| **Size** | O(1) | O(1) |
| **Memory Usage** | Slightly higher (extra unused space in list capacity) | Efficient memory (only as many nodes as elements) |
| **When to prefer?** | Best for known, fixed-size stacks and when fast random access needed | Best for frequent, unpredictable inserts/deletes |

TASK 2

**Explanation: How Stack is Used**

* **Postfix Expression** means **operator comes after operands** (e.g., 5 6 + instead of 5 + 6).
* **Step-by-Step working**:
  1. Read the expression **left to right**.
  2. If the token is a **number**, **push** it onto the stack.
  3. If the token is an **operator**:
     + **Pop two operands** from the stack (right first, then left).
     + **Apply the operator**.
     + **Push the result** back onto the stack.
  4. After finishing, the stack will have **only one element** — the **final result**.

This way, **stack handles intermediate results** without needing complex parsing or precedence rules!

**Time Complexity**

* Each token is processed exactly once.
* Stack operations (push, pop) are **O(1)**.
* Therefore, overall time complexity is **O(N)**, where **N = number of tokens**.

TASK 3

**Comparison: Circular Queue vs Linear Queue**

| **Feature** | **Linear Queue (List)** | **Circular Queue (Fixed Array)** |
| --- | --- | --- |
| **Space Usage** | May waste space after dequeues | Utilizes space efficiently |
| **Insertion Complexity** | O(1) (if no resizing needed) | O(1) |
| **Deletion Complexity** | O(N) (due to shifting elements) | O(1) |
| **Memory Management** | Dynamic (grows/shrinks) | Fixed memory (constant space) |
| **Performance** | Slower after multiple dequeues | Consistent O(1) operations |
| **When to Use** | Small, simple tasks | Performance-critical, fixed size queues |

**In short:**

* **Linear queues** become inefficient when many elements are dequeued because shifting is costly.
* **Circular queues** avoid shifting and maximize memory use, perfect for systems like printers, CPU scheduling, or memory buffers.

**Time Complexity**

* **enqueue, dequeue, front, rear, is\_empty, is\_full**: **O(1)**

Topic 5: Hashing & Hash Tables

TASK 1

**Comparison**

| **Feature** | **Chaining** | **Linear Probing** |
| --- | --- | --- |
| **Space Usage** | Flexible (dynamic linked list) | Fixed table size |
| **Performance (Average)** | O(1 + λ) where λ = load factor | O(1 / (1 - λ)) |
| **Deletion** | Easy | Tricky (may need "tombstones") |
| **When Table is Full** | Can still grow a chain | Must resize the table |

TASK 2

**Why Hashing Is Efficient?**

* Counting characters takes O(n).
* Comparing counts takes O(n).
* Overall **Time Complexity = O(n)** where n = length of the strings.

TASK 3

**Why Hashing Improves Cache Performance?**

* Hash map provides **O(1)** access to nodes.
* Doubly linked list allows **O(1)** updates for recent usage order.
* Together, **all cache operations are O(1)**.

Topic 6: Trees & Binary Search Trees (BST)

TASK 1

**Explanation:**

* **Insert:** Recursively find correct position.
* **Search:** Recursively find if value exists.
* **Delete:**
  + **No child:** Remove it.
  + **One child:** Replace with child.
  + **Two children:** Replace with inorder successor (smallest in right subtree).
* **Inorder Traversal:** Left → Root → Right (ascending order).

**➔ Time Complexity:**

| **Operation** | **Average Case** | **Worst Case (Unbalanced Tree)** |
| --- | --- | --- |
| Insert/Search/Delete | O(log n) | O(n) |

TASK 2

**Explanation:**

* If both nodes are smaller, move left.
* If both are larger, move right.
* Otherwise, current node is LCA.

**Time Complexity:**

* **O(log n)** for balanced BST
* **O(n)** for skewed BST

TASK 3

**Explanation:**

* **Recursive approach** checks height of left and right subtrees.
* Tree is **balanced** if difference is ≤ 1 for all nodes.

**Time Complexity:**

* **O(n)** where *n* = number of nodes

Topic 7: Heaps & Priority Queues

TASK 1

**Heap Properties:**

* A Heap is a **complete binary tree**.
* Min-Heap: Parent node ≤ child nodes.
* Max-Heap: Parent node ≥ child nodes.

**Time Complexity:**

* **Insertion:** O(log n)
* **Extraction:** O(log n)
* **Peek:** O(1)
* **Heapify an array:** O(n)

TASK 2

**Explanation:**

* Priority Queue gives the highest priority element first.
* Lower number = higher priority.
* Heap makes enqueue and dequeue efficient: O(log n) instead of O(n).

TASK 3

**Comparison of Heap vs Sorting:**

| **Method** | **Time Complexity** |
| --- | --- |
| Heap (for K elems) | O(n + k log n) |
| Full Sorting | O(n log n) |

* **Heap is more efficient** when k is much smaller than n (e.g., 5 smallest from 1000 numbers).
* Heap only gives what we need without sorting the full array.

Topic 8: Graphs & Graph Algorithms

TASK 1

**Explanation:**

* **Adjacency List**: A dictionary of lists where the keys are the vertices, and the values are the lists of adjacent vertices.
* **Adjacency Matrix**: A 2D matrix where the rows and columns represent the vertices, and the values (0 or 1) indicate the presence of an edge.

**When to use:**

* **Adjacency List**: More memory-efficient for sparse graphs.
* **Adjacency Matrix**: Better for dense graphs and when you need quick lookups for edge existence.

TASK 2

**Explanation:**

* **BFS (Breadth-First Search)**: Uses a queue to explore the graph level by level. It’s suitable for finding the shortest path in an unweighted graph.
* **DFS (Depth-First Search)**: Uses recursion (or a stack) to explore the graph as deep as possible before backtracking. It’s good for searching through a graph or tree structure.

**Real-World Use Cases:**

* **BFS**: Social networks (finding shortest connections), web crawlers.
* **DFS**: Solving puzzles (like mazes), topological sorting.

TASK 3

**Explanation:**

* **Dijkstra’s Algorithm**: A greedy algorithm to find the shortest path in a weighted graph. It works by iteratively selecting the vertex with the smallest known distance, updating the distances of its neighbors, and using a priority queue for efficiency.
* **Time Complexity**: O(V log V + E log V), where V is the number of vertices and E is the number of edges.

**Use Cases:**

* **GPS Navigation**: Finding the shortest path.
* **Network Routing**: Efficient data transmission paths.

**Comparison of Algorithms:**

| **Algorithm** | **Time Complexity** | **Space Complexity** | **Use Cases** |
| --- | --- | --- | --- |
| **BFS** | O(V + E) | O(V) | Shortest path (unweighted), social networks |
| **DFS** | O(V + E) | O(V) | Tree traversal, path finding |
| **Dijkstra’s** | O(V log V + E) | O(V + E) | Shortest path in weighted graphs |

Topic 9: Sorting Algorithms

TASK 1,2,3

**Time Complexity Analysis**

* **Bubble Sort**: O(n^2)
* **Selection Sort**: O(n^2)
* **Insertion Sort**: O(n^2) in the worst case, but O(n) if the list is nearly sorted.

**Explanation of Time Complexity and Best Use Cases:**

* **Heap Sort**: O(n log n) for all cases (ideal for large datasets that don't fit into memory).
* **Counting Sort**: O(n+k) where k is the range of input values (ideal for small ranges of integers).
* **Quick Sort**: O(n log n) on average, but O(n^2) in the worst case (ideal for general-purpose sorting).
* **Merge Sort**: O(n log n) for all cases (ideal for linked lists and large datasets).

Topic 10: Searching Algorithms

TASK 1

**Explanation of When to Use Each Algorithm:**

* **Linear Search**: Suitable for unsorted lists or when the list is small. It checks every element sequentially, making it slower for larger datasets.
* **Binary Search**: Suitable for sorted lists. It is faster than Linear Search with a time complexity of O(log n) and is highly efficient for large datasets.

TASK 2

**Explanation of Best Use Cases:**

* **Jump Search**: Ideal for sorted lists when the dataset is large and the elements are uniformly distributed. It reduces the number of comparisons compared to linear search.
* **Interpolation Search**: Best for sorted, uniformly distributed data. It performs better than binary search when the data is evenly spread out.
* **Binary Search**: Suitable for sorted lists, providing efficient search with O(log n) complexity.

TASK 3

**Explanation of When to Use These Algorithms:**

* **Exponential Search**: Best for sorted lists, especially when the target is likely to be near the beginning of the list.
* **Fibonacci Search**: Suitable for sorted lists, and may be preferred when the dataset is large and the partitioning technique suits the Fibonacci sequence.

Topic 11: Hashing & Hash Tables

TASK 1

**Time Complexity:**

* **Insert**: O(1) on average for both methods (Chaining and Linear Probing), but can degrade to O(n) if there are many collisions.
* **Get**: O(1) on average.
* **Delete**: O(1) on average.

TASK 2

**Time Complexity:**

* **Insert**: O(1) for both custom and built-in hash functions, depending on the load factor and collision resolution strategy.
* **Collision Analysis**: We analyze the number of collisions by looking at the distribution of keys across hash values.

TASK 3

**Time Complexity:**

* **Get**: O(1) using the hash table.
* **Put**: O(1) for both insert and eviction due to the hash table and doubly linked list.

Topic 12: Graph Data Structure

TASK 1,2,3

**Explanation of Time Complexity:**

1. **Adjacency List and Matrix**:
   * Adding vertices and edges: O(1)
   * Displaying the graph: O(V + E) for adjacency list, O(V^2) for adjacency matrix.
2. **BFS and DFS**:
   * Both BFS and DFS have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges.
3. **Dijkstra’s Algorithm**:
   * The time complexity of Dijkstra’s algorithm with a priority queue is O((V + E) log V), where V is the number of vertices and E is the number of edges.

Topic 13: Greedy Algorithms

TASK 1

**Explanation:**

* **Time Complexity**: Sorting the activities takes O(n log n), and the subsequent loop runs in O(n), so the overall time complexity is **O(n log n)**.
* **Greedy Approach**: The algorithm selects activities that leave the most room for future activities, ensuring the maximum number of non-overlapping activities.

**Real-world Applications:**

* Scheduling tasks or meetings.
* Resource allocation (e.g., assigning rooms or equipment to tasks).

TASK 2

**Explanation:**

* **Time Complexity**:
  + Constructing the priority queue takes **O(n log n)**.
  + Building the tree and generating codes takes **O(n)**.
  + Overall complexity: **O(n log n)**.
* **Greedy Approach**: At each step, Huffman Coding chooses the two least frequent characters to combine, minimizing the average length of the encoded string.

**Comparison of Original vs. Compressed Data Size:**

* The original string hello greedy might be represented in a binary format with ASCII codes, each taking 8 bits.
* Using Huffman encoding, the characters are assigned shorter codes based on their frequency, leading to compression.

TASK 3

**Explanation:**

* **Time Complexity**:
  + Sorting edges: **O(E log E)** where E is the number of edges.
  + Union-Find operations: **O(α(V))**, where α is the inverse Ackermann function, which is nearly constant.
* **Greedy Approach**: Kruskal’s algorithm chooses the smallest edge that doesn’t form a cycle, ensuring the MST with minimum weight.

**Comparison with Prim’s Algorithm:**

* **Prim's Algorithm** is a more efficient choice for dense graphs (O(E + V log V) with a priority queue).
* **Kruskal's Algorithm** is better for sparse graphs (O(E log E)).

Topic 14: Dynamic Programming

TASK 1

**Time Complexity:**

* **Memoization (Top-Down)**: O(n) because each Fibonacci number is computed only once and stored in the memo dictionary.
* **Tabulation (Bottom-Up)**: O(n) because we iteratively fill a table of size n to solve the problem.

**Efficiency:**

* **Memoization** has an overhead due to recursion and dictionary lookups, but it avoids redundant calculations, making it more efficient than the naive recursive approach (which has a time complexity of **O(2^n)**).
* **Tabulation** is typically more space-efficient because it avoids recursion and can be done in a single loop with an array.

TASK 2

**Time Complexity:**

* **Memoization (Top-Down)**: O(m \* n) because each pair of (m, n) is computed only once and stored in the memo dictionary.
* **Tabulation (Bottom-Up)**: O(m \* n) because we iteratively fill a 2D table with dimensions m x n, where m is the length of X and n is the length of Y.

**Efficiency:**

* **Memoization** has a smaller memory footprint due to the recursion and the dictionary storing only the necessary subproblems.
* **Tabulation** is often faster and uses an explicit table to avoid recursion, making it more space-efficient than the memoization approach for large inputs.

TASK 3

**Time Complexity:**

* **O(n \* W)** where n is the number of items and W is the weight capacity. We fill a 2D DP table of size n x W.

**Efficiency:**

* This DP solution efficiently solves the knapsack problem, and the time complexity is polynomial in terms of both the number of items and the weight capacity.

**Real-World Applications:**

* **Budget optimization**: Allocating resources to maximize value without exceeding a budget.
* **Resource allocation**: Choosing the best set of resources (e.g., materials, people) under a weight or capacity constraint.

**Deliverables:**

1. **Python implementations** for Fibonacci using Memoization & Tabulation, LCS using Memoization & Tabulation, and 0/1 Knapsack Problem.
2. **Time Complexity analysis** for each problem (O(n), O(m\*n), O(n \* W)).
3. **Comparison of Recursive vs Tabulation approaches**: Both approaches solve the problems in polynomial time, with tabulation often being more efficient in practice for large inputs.

Topic 15: Backtracking

TASK 1

**Explanation of Time Complexity:**

* The **time complexity** for the N-Queens problem is **O(N!)**, since in the worst case, we try to place each queen in every row and column (N choices for each of the N queens). This can be reduced using constraint propagation and pruning.

TASK 2

**Explanation of Time Complexity:**

* The **time complexity** for generating permutations of a string is **O(N!)**, where N is the length of the string, because there are N! permutations in total.
* **Space complexity** is **O(N)**, where N is the length of the string.

TASK 3

**Explanation of Time Complexity:**

* The **time complexity** is difficult to express directly, but in general, it's **O(9^(n))**, where n is the number of empty cells. This is because, in the worst case, we explore all possible configurations, trying up to 9 possibilities for each empty cell.
* **Forward checking** helps reduce the number of possibilities early, making the algorithm more efficient.

**Deliverables:**

1. **Python Implementations:**
   * **N-Queens Problem using backtracking with constraint propagation.**
   * **String Permutations using backtracking with recursion.**
   * **Sudoku Solver using backtracking with forward checking.**
2. **Time Complexity:**
   * **N-Queens: O(N!) due to the recursive exploration of all queen placements.**
   * **String Permutations: O(N!) because there are N! possible permutations of a string of length N.**
   * **Sudoku Solver: O(9^(n)) where n is the number of empty cells. Forward checking helps prune the search space.**
3. **Use Cases:**
   * **N-Queens: Solving chessboard placement problems.**
   * **String Permutations: Useful for password generation, generating anagrams, etc.**
   * **Sudoku Solver: Solving Sudoku puzzles with efficient backtracking.**

**Top of Form**

**Bottom of Form**

Topic 16: Graph Algorithms

TASK 1

**Time Complexity:**

* **DFS & BFS Time Complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**When to use DFS vs BFS:**

* **DFS** is better when you want to explore as far down a branch as possible before backtracking (useful in pathfinding and puzzle solving).
* **BFS** is better when you want to explore all neighbors level by level (useful in shortest path algorithms like Dijkstra’s).

TASK 2

**Time Complexity:**

* **Dijkstra’s Time Complexity:** O((V + E) log V), where V is the number of vertices and E is the number of edges. This is due to the priority queue operations.

**Real-World Applications:**

* **Navigation Systems (e.g., Google Maps):** For finding the shortest route.
* **Network Routing Algorithms:** To minimize the time or cost of data transmission.

TASK 3

**Explanation of Approaches:**

* **Union-Find (Disjoint Set)** is used in undirected graphs to check if two nodes belong to the same set, which indicates a cycle.
* **DFS with Recursion Stack** is used in directed graphs to track nodes in the current recursion stack, helping detect cycles.

**Real-World Applications:**

* **Deadlock Detection in Operating Systems:** Detecting cycles in resource allocation graphs.
* **Network Analysis:** Detecting cycles in communication networks can help avoid routing loops.