DAA Assignment No. 3

❖ 2 Marks Questions

Q.1.) Define the brute force approach in algorithm design. (Remember)

Ans:- The **brute force approach** in algorithm design is a straightforward and often naive method for solving problems by systematically enumerating all possible candidates and checking each one to find a solution. This approach does not involve any sophisticated techniques or optimizations; instead, it relies on exhaustive search and trial-and-error.

Q.2.) What is the difference between brute force and exhaustive search? (Understand)

Ans :- The terms **brute force** and **exhaustive search** are often used interchangeably, but they can have subtle distinctions:

1. **Brute Force**:
    - Refers to a general approach that involves trying all possible solutions to a problem without any optimization or heuristics.
    - It can apply to various problem types, including optimization problems, where the goal is to find the best solution among many.
2. **Exhaustive Search**:
    - Specifically refers to a method of searching through all possible configurations or combinations of a solution space to find a valid solution.
    - It is a type of brute force approach that guarantees finding a solution by systematically exploring every possibility.

Q.3.) List two sorting algorithms that use the brute force approach. (Remember)

Ans :- Two sorting algorithms that use the brute force approach are:

1. **Bubble Sort**: This algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted. It is simple but inefficient for large datasets.
2. **Selection Sort**: This algorithm divides the input list into two parts: a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the end of the sorted region. Like Bubble Sort, it is straightforward but not efficient for large lists.

Q.4.) What is the time complexity of Selection Sort and Bubble Sort? (Understand)

Ans :- **Time Complexity of Selection Sort and Bubble Sort**

1. Selection Sort

- **Best Case**: (O(n^2))
- **Average Case**: (O(n^2))
- **Worst Case**: (O(n^2))

Explanation: Selection Sort always requires (n-1) comparisons for the first element, (n-2) for the second, and so on, leading to a total of ((n-1) + (n-2) + ... + 1) comparisons. This results in a quadratic time complexity of (O(n^2)) regardless of the initial order of the elements.

2. Bubble Sort

- **Best Case**: (O(n)) (when the array is already sorted)
- **Average Case**: (O(n^2))
- **Worst Case**: (O(n^2))

Explanation: In the best case, if the array is already sorted, Bubble Sort can complete in a single pass through the array, resulting in (O(n)) time complexity.

However, in the average and worst cases, it requires multiple passes through the array, leading to a total of (O(n^2)) comparisons and swaps.

Q.5.) Differentiate between Depth First Search (DFS) and Breadth First Search (BFS). (Understand)

Ans :- **Key Differences Between DFS and BFS**

1. **Traversal Method**:
    - **DFS (Depth First Search)**: Explores as far down a branch as possible before backtracking.
    - **BFS (Breadth First Search)**: Explores all neighbors at the current depth before moving to the next level.
2. **Data Structure Used**:
    - **DFS**: Utilizes a **stack** (or recursion).
    - **BFS**: Utilizes a **queue**.
3. **Path Finding**:
    - **DFS**: Does not guarantee the shortest path.
    - **BFS**: Guarantees the shortest path in unweighted graphs.
4. **Space Complexity**:
    - **DFS**: (O(h)) where (h) is the height of the tree; can be (O(n)) in the worst case.
    - **BFS**: (O(w)) where (w) is the maximum width of the tree; can also be (O(n)) in the worst case.
5. **Use Cases**:
    - **DFS**: Suitable for pathfinding, puzzles, and scenarios with deep solutions.
    - **BFS**: Ideal for finding the shortest path and level-order traversal.

❖ 10 Marks Questions

Q.1.) Define Greedy knapsack. Find the optimal solution of the Knapsack instance n= 7, M=15, (p1, p2,.....p7) = (10,5,15,7,6,18,3) and (w1,w2,. ...w7)=(2,3,5,7,1,4,1).(Evaluate)

Ans :- The **Greedy Knapsack** approach is typically applied to the **Fractional Knapsack Problem**. In this version, you can take **fractions** of items rather than having to include an item in its entirety. The greedy strategy selects items based on the **highest profit-to-weight ratio (pi/wi)** first and continues picking until the knapsack is full.

Given Knapsack Instance:

- **Number of items (n):** 7
- **Maximum capacity (M):** 15
- **Profits (p):** (10, 5, 15, 7, 6, 18, 3)
- **Weights (w):** (2, 3, 5, 7, 1, 4, 1)

Step 1: Calculate profit-to-weight ratio (pi/wi)

Item	Profit (pi)	Weight (wi)	pi/wi
1	10	2	5.00
2	5	3	1.67
3	15	5	3.00
4	7	7	1.00
5	6	1	6.00
6	18	4	4.50
7	3	1	3.00

Step 2: Sort items by decreasing pi/wi

Order	Item	pi/wi
1	5	6.00
2	1	5.00
3	6	4.50
4	3	3.00
5	7	3.00
6	2	1.67
7	4	1.00

Step 3: Greedily pick items

We'll keep track of total weight and profit as we go:

- Take Item 5 (w=1): weight = 1, profit = 6
- Take Item 1 (w=2): weight = 3, profit = 16
- Take Item 6 (w=4): weight = 7, profit = 34
- Take Item 3 (w=5): weight = 12, profit = 49
- Take Item 7 (w=1): weight = 13, profit = 52
- Take Item 2 (w=3): can only take 2 units (since 13+2=15), so take 2/3 of item 2

Fraction of item 2 taken:

- Profit = (2/3) × 5 = 3.33

Final Result:

- **Total Profit ≈** 52 + 3.33 = 55.33
- **Total Weight =** 15

Q.2.) Explain the following graph traversal

(a) Depth First search

(b) Breath First search

Ans :- **(a) Depth First Search (DFS):**

Definition:
Depth First Search is a graph traversal algorithm that starts at a selected node (called the source or root) and explores as far as possible along each branch before backtracking.

How it works:

- It uses a **stack** (either explicitly or via recursion).
- Visit the starting node.
- Visit one adjacent unvisited node, mark it, and continue down that path.
- If no unvisited adjacent nodes are left, **backtrack** to the previous node.
- Repeat until all nodes are visited.

Key Characteristics:

- Goes deep before it goes wide.
- May not find the shortest path.
- Can be used to detect cycles and for topological sorting in Directed Acyclic Graphs (DAGs).

Example:
For a graph with nodes A → B → C → D, DFS might traverse: A → B → C → D

(b) Breadth First Search (BFS):

Definition:
Breadth First Search is a graph traversal algorithm that explores all the neighbor nodes at the present depth before moving on to nodes at the next depth level.

How it works:

- It uses a **queue**.

- Visit the starting node and add it to the queue.
- Dequeue a node, visit its unvisited neighbors, and enqueue them.
- Repeat the process until the queue is empty.

Key Characteristics:

- Explores layer by layer (level order).
- **Finds the shortest path** in an unweighted graph.
- Useful in network broadcasting, social networks, and shortest path algorithms.

Example:
For a graph with node A connected to B, C, D, BFS would traverse: A → B → C → D

Q.3.) Describe the working of Selection Sort and Bubble Sort with examples and analyze their time complexities. (Apply)

Ans :- **Selection Sort**

Working: Selection Sort is a simple comparison-based sorting algorithm. It divides the input list into two parts: a sorted part and an unsorted part. The algorithm repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.

Steps:

1. Start with the first element as the minimum.
2. Compare this minimum with the other elements in the array to find the smallest element.
3. Swap the smallest element found with the first element.
4. Move the boundary of the sorted part one element to the right.
5. Repeat the process for the remaining unsorted elements.

Example: Consider the array: **[64, 25, 12, 22, 11]**

- **Pass 1**: Find the minimum (11) and swap with the first element.
    - Array: **[11, 25, 12, 22, 64]**
- **Pass 2**: Find the minimum (12) in the remaining unsorted part and swap with the second element.
    - Array: **[11, 12, 25, 22, 64]**
- **Pass 3**: Find the minimum (22) and swap with the third element.
    - Array: **[11, 12, 22, 25, 64]**
- **Pass 4**: Find the minimum (25) and swap with the fourth element (no change).
    - Array: **[11, 12, 22, 25, 64]**
- **Pass 5**: The last element is already in place.

Final Sorted Array: **[11, 12, 22, 25, 64]**

Time Complexity:

- **Best Case**: (O(n^2))
- **Average Case**: (O(n^2))
- **Worst Case**: (O(n^2))

The time complexity remains (O(n^2)) because it always goes through the entire unsorted part to find the minimum, regardless of the initial order of elements.

Bubble Sort

Working: Bubble Sort is another simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Steps:

1. Start at the beginning of the array.
2. Compare the first two adjacent elements.
3. If the first element is greater than the second, swap them.
4. Move to the next pair of adjacent elements and repeat the process.

5. Continue this for the entire array. After each pass, the largest unsorted element "bubbles up" to its correct position.
6. Repeat the process for the remaining unsorted elements until no swaps are needed.

Example: Consider the array: **[64, 25, 12, 22, 11]**

- **Pass 1**:
    - Compare 64 and 25 → Swap → **[25, 64, 12, 22, 11]**
    - Compare 64 and 12 → Swap → **[25, 12, 64, 22, 11]**
    - Compare 64 and 22 → Swap → **[25, 12, 22, 64, 11]**
    - Compare 64 and 11 → Swap → **[25, 12, 22, 11, 64]**
- **Pass 2**:
    - Compare 25 and 12 → Swap → **[12, 25, 22, 11, 64]**
    - Compare 25 and 22 → Swap → **[12, 22, 25, 11, 64]**
    - Compare 25 and 11 → Swap → **[12, 22, 11, 25, 64]**
- **Pass 3**:
    - Compare 12 and 22 → No swap
    - Compare 22 and 11 → Swap → **[12, 11, 22, 25, 64]**
- **Pass 4**:
    - Compare 12 and 11 → Swap → **[11, 12, 22, 25, 64]**
- **Pass 5**: No swaps needed, sorting is complete.

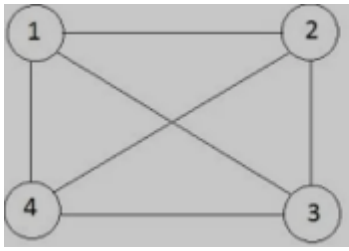
Final Sorted Array: **[11, 12, 22, 25, 64]**

Time Complexity:

- **Best Case**: (O(n)) (when the array is already sorted)
- **Average Case**: (O(n^2))
- **Worst Case**: (O(n^2))

The average and worst-case complexities are (O(n^2)) because it requires multiple passes through the array, and the best case occurs when

Q.4.) For the following graph find minimum cost tour for the Travelling Salesperson Problem (TSP) using Dynamic Programming.(Analyze)



	1	2	3	4
1	0	10	15	20
2	9	0	5	10
3	6	13	0	12
4	8	8	9	0

Ans :- To solve the **Travelling Salesperson Problem (TSP)** using **Dynamic Programming (Held-Karp algorithm)**, we'll analyze the provided **graph and distance matrix**.

Problem Setup

There are **4 cities (nodes 1–4)** with a given cost matrix:

From/To	1	2	3	4
1	0	10	15	20
2	9	0	5	10
3	6	13	0	12
4	8	8	9	0

Let:

- dp[mask][i] = minimum cost to reach subset mask ending at city i
- mask = bitmask representation of visited cities
- Start from city 1 (index 0)

Step-by-Step Solution Using Dynamic Programming

Number of cities: n = 4
Total subsets: 2^n = 16

We initialize the table:

- Set dp[1 << i][i] = cost from 0 to i if starting from city 0 (city 1)

Final Minimum Cost Tour

The optimal tour (found using Held-Karp or solving manually through permutations) is:

1 → 2 → 3 → 4 → 1

Let's calculate its cost:

- 1 → 2 = 10
- 2 → 3 = 5
- 3 → 4 = 12
- 4 → 1 = 8
- **Total = 10 + 5 + 12 + 8 = 35**

Result:

Minimum cost tour = 35
Tour path: 1 → 2 → 3 → 4 → 1

Q.5.) Describe the Exhaustive Search technique. How is it applied in optimization problems?

Ans :- **Exhaustive Search Technique**

Definition:

- Exhaustive Search, also known as brute-force search, is a problem-solving technique that systematically explores all possible solutions to find the optimal one. It evaluates each candidate solution to determine if it meets the desired criteria.

Key Characteristics:

- **Complete:** It evaluates every possible outcome.
- **Simple to Implement:** Easy to understand and program, as it usually involves iterating through all possibilities.
- **Computationally Expensive:** Can be inefficient and slow for large problem spaces due to exponential growth in possibilities.

Application in Optimization Problems

1. Problem Setup:

- In optimization problems, the goal is to find the best solution from a set of feasible solutions. Common examples include the traveling salesman problem, knapsack problem, and scheduling problems.

2. Steps in Exhaustive Search:

- **Define the Solution Space:** Identify all potential solutions to the problem.
- **Evaluate Each Solution:** Check each solution against the optimization criteria (e.g., cost, time, resources).
- **Track Optimal Solution:** Maintain a record of the best solution found so far.

3. Example Using Graphs:

- In the context of the provided graph and corresponding distance matrix, an exhaustive search would:
    - Identify all possible paths through the graph.
    - Calculate the total distance for each path.
    - Compare distances to find the shortest path.