Q.1) a)An algorithm is a step-by-step procedure or a well-defined set of instructions to solve a specific problem or perform a computation. It takes input, processes it, and produces the desired output. Algorithms are the foundation of all programming and are essential for solving problems efficiently and logically.

Importance in Computer Science:

Algorithms form the core of computer programming. They help in designing efficient software, optimizing resource usage, and ensuring that problems are solved in the best possible way. Whether it's searching data, sorting lists, or managing memory — algorithms make everything work faster and smarter.

Recursion:

Recursion is a concept where a function calls itself to solve a smaller subproblem of the original problem. Every recursive function has a base case (termination point) and a recursive case (repeated call).

Example – Factorial (n!):

```
def factorial(n):
   if n == 0:
      return 1
else:
      return n * factorial(n-1)
```

Calling factorial(4) results in:

4 * factorial(3) → 4 * 3 * factorial(2) → ... → 4 * 3 * 2 * 1 = 24

Q1(b): In Linear Search, each element in the array is checked sequentially until the desired element is found or the end is reached.

Time Complexity:

- **Best Case: O(1) → When the element is at the first position.**
- **Worst Case: O(n) → When the element is at the last position or not found.**
- **Average Case: O(n)**

Example:

Suppose we have the list: [10, 20, 30, 40, 50] and we want to find 30.

- **Linear search will compare:**
- **10 ≠ 30 → 20 ≠ 30 → 30 = 30 (found at 3rd position)**
- **Number of comparisons = 3 → Time complexity O(n)**

Linear Search is simple but inefficient for large datasets.

Q1(c): (i) Backtracking:

Backtracking is a problem-solving algorithm that builds a solution incrementally and abandons (backtracks) a path if it leads to a dead end. It is used in problems like maze solving, N-Queens, and Sudoku. It tries all possibilities and removes wrong paths through recursion.

Example: Solving N-Queens problem by placing queens row by row, and backtracking when there's a conflict.

(ii) Fractional Knapsack Problem:

In this greedy algorithm, items can be broken into smaller parts. We pick the item with the highest value-to-weight ratio first, and continue until the knapsack is full. This approach maximizes value and allows selecting fractions of items.

Example:

If an item weighs 10kg and you can carry only 5kg, you take half of it and get half the value.

(iii) Connected Graph:

A connected graph is a type of graph in which every vertex is reachable from every other vertex by some path. In undirected graphs, there is a path between every pair of nodes. A disconnected graph has at least two isolated parts.

Example:

In a graph with nodes A, B, C, D — if all are connected, it is a connected graph.

Q1(d): Greedy Approach – Minimum Number of Notes for ₹437

Given denominations: ₹2, ₹5, ₹10, ₹15, ₹20, ₹100, ₹500 (available in unlimited quantity)

Goal: Use the Greedy Algorithm to select the least number of notes to form ₹437

🔹 **Greedy Strategy:**

Choose the highest possible denomination ≤ remaining amount, subtract it, and repeat the process until the amount becomes 0.

🔹 **Step-by-Step Selection:**

1. **Amount = ₹437**

   Use ₹500 → ✗ (Too big)

2. Use ₹100 → ✅

437 - 100 = ₹337

Notes used: 1 × ₹100

3. Use ₹100 → ✅

337 - 100 = ₹237

Notes used: 2 × ₹100

4. Use ₹100 → ✅

237 - 100 = ₹137

Notes used: 3 × ₹100

5. Use ₹100 → ✅

137 - 100 = ₹37

Notes used: 4 × ₹100

6. Use ₹20 → ✅

37 - 20 = ₹17

Notes used: 1 × ₹20

7. Use ₹15 → ✅

17 - 15 = ₹2

Notes used: 1 × ₹15

8. Use ₹2 → ✅

2 - 2 = ₹0

Notes used: 1 × ₹2

🔹 Final Answer:

- Total notes used = 4 (₹100) + 1 (₹20) + 1 (₹15) + 1 (₹2) = 7 notes

🔹 Sequence of Notes Used:

₹100, ₹100, ₹100, ₹100, ₹20, ₹15, ₹2

Q3(a): The Bubble Sort algorithm is a simple comparison-based sorting technique. It works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. This process continues until the array is sorted.

- Best Case (Already Sorted Array): When the input array is already sorted, Bubble Sort can be optimized by introducing a flag that checks whether any swaps were made during the pass. If no swaps were made, the algorithm stops. In this case, the time complexity is:
  - $O(n)$
- Worst Case (Reverse Order): When the input array is in descending order, the algorithm has to make the maximum number of comparisons and swaps. Time complexity:
  - $O(n^2)$
- Average Case: In most cases, the time complexity remains:
  - $O(n^2)$

Thus, although easy to implement, Bubble Sort is inefficient for large datasets.

Q3(b): We are given two algorithms with recurrence relations:

- T(n) = 7T(n/2) + n^2 (Algorithm A)
- T'(n) = aT(n/4) + n^2 (Algorithm A')

Using the Master Theorem:

For A:

- a = 7, b = 2, f(n) = n^2
- Compare f(n) to n^log_b(a) = n^log_2(7) ≈ n^2.81
- Since f(n) = o(n^log_b(a)), Case 1 of Master Theorem applies:
  - T(n) = Θ(n^log_2(7)) ≈ Θ(n^2.81)

For A':

- a is unknown, b = 4, f(n) = n^2
- We want A' to be asymptotically faster, i.e., T'(n) = o(n^2.81)

So:

- Require: log_4(a) < log_2(7) ≈ 2.81

- log_4(a) = x ⇒ a = 4^x
- Set x < 2.81 ⇒ a < 4^2.81 ≈ 55.2

So, the largest integer a = 55

Q4(a): Depth First Search (DFS) and Breadth First Search (BFS) are graph traversal techniques used in various graph-related problems.

DFS:

- Uses a stack (or recursion)
- Goes deep in a branch before backtracking
- Can be used for topological sort, cycle detection, maze solving
- Not guaranteed to find the shortest path

BFS:

- Uses a queue
- Explores all neighbors level by level
- Guaranteed to find shortest path in unweighted graphs
- Useful in shortest path, social network analysis

Key Differences:

- Memory Usage: DFS uses less memory in sparse graphs, BFS uses more due to queue
- Traversal Nature: DFS is depth-wise, BFS is breadth-wise
- Time Complexity: Both are O(V + E) for a graph with V vertices and E edges

Use Case Examples:

- DFS: Maze solving
- BFS: Finding shortest path in an unweighted graph

Q4(b): A Minimum Spanning Tree (MST) is not always unique. The MST is unique only if all edge weights are distinct. If some edges have equal weight, more than one MST can be formed.

Prim’s Algorithm Steps:

1. Start from any node (e.g., node 'a')
2. Select the edge with the minimum weight from the current node to an unvisited node
3. Add the edge to the MST
4. Repeat until all nodes are included

Why Uniqueness Fails:

- Consider two edges with equal minimum weight connecting different nodes. Either can be selected without affecting the total weight, resulting in different MSTs.

Therefore, Prim’s algorithm does not guarantee a unique MST unless all weights are unique.

Q5(a): The partition procedure is the heart of the Quick Sort algorithm. It takes a pivot element (usually the last element of the array) and rearranges the array so that:

- **All elements less than or equal to the pivot are on the left,**
- **All elements greater than the pivot are on the right.**

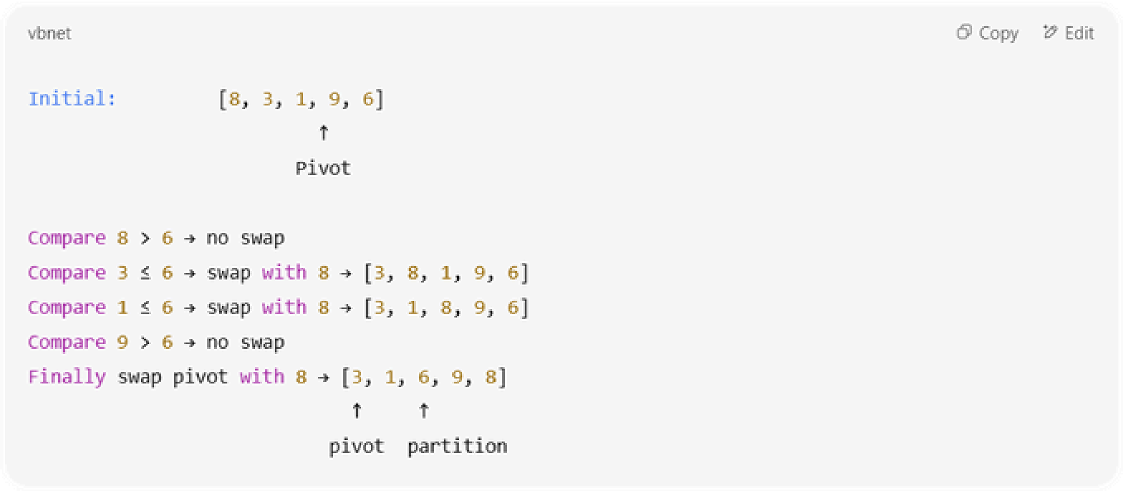
This is done using a single scan of the array with two pointers.

Example:

Consider the array:

A = [8, 3, 1, 9, 6] and pivot = 6

Step-by-step partition:



```
Initial:        [8, 3, 1, 9, 6]
                            ↑
                          Pivot

Compare 8 > 6 → no swap
Compare 3 ≤ 6 → swap with 8 → [3, 8, 1, 9, 6]
Compare 1 ≤ 6 → swap with 8 → [3, 1, 8, 9, 6]
Compare 9 > 6 → no swap
Finally swap pivot with 8 → [3, 1, 6, 9, 8]
                                  ↑     ↑
                                pivot  partition
```

Now, 6 is at the correct position, and all smaller elements are on its left.

Time Complexity:

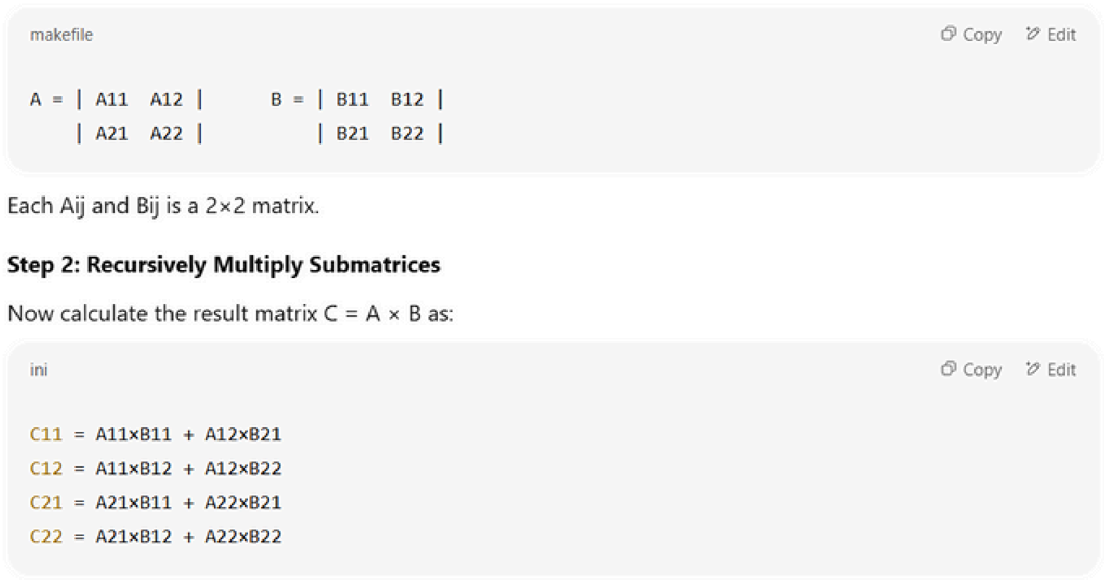
Only one pass is done over the entire array of size n, hence:

Time Complexity=Θ(n)\text{Time Complexity} = \Theta(n)Time Complexity=Θ(n)

Q.5) b) To multiply two 4×4 matrices using the divide and conquer approach, we divide each matrix into four 2×2 submatrices.

Step 1: Divide each matrix

Let matrix A and B be:



```
A = | A11  A12 |      B = | B11  B12 |
    | A21  A22 |          | B21  B22 |
```

Each Aij and Bij is a 2×2 matrix.

Step 2: Recursively Multiply Submatrices

Now calculate the result matrix C = A × B as:

```
C11 = A11×B11 + A12×B21
C12 = A11×B12 + A12×B22
C21 = A21×B11 + A22×B21
C22 = A21×B12 + A22×B22
```

Diagram:

```rust
Matrix A        Matrix B        Result C
| A11 A12 |     | B11 B12 |     | C11 C12 |
| A21 A22 |  x  | B21 B22 |  =  | C21 C22 |
```

Each multiplication (e.g., A11×B11) is a 2×2 matrix multiplication and can again be solved recursively.

Advantages:

- This approach breaks large problems into smaller subproblems.
- It is the basis for efficient algorithms like **Strassen's algorithm**.
- Reduces number of scalar multiplications compared to naive methods.