BFS:

Algorithm:

Step	Description
Step-1	Initialize an empty queue and a `visited` array. Start from the given node (e.g., 1). Mark it as visited and push it into the queue.
Step-2	While the queue is not empty, dequeue the front element and print it.
Step-3	For the current node, explore all its neighbors. If a neighbor is not visited, mark it as visited and push it into the queue.
Step-4	Repeat the process until the queue becomes empty.

Complexity Analysis

- **Time Complexity:**
  $O(V+E)$, where V is the number of vertices and E is the number of edges.
  - Visiting each vertex takes $O(V)$ time.
  - Exploring the neighbors of all vertices requires $O(E)$ time (because each edge is visited once).
- **Space Complexity:**
  $O(V)$ for the queue and the visited array.

1. Recursive Relation for Breadth-First Search (BFS)

BFS uses a **queue** to explore all nodes level by level. Although not inherently recursive, we can represent its exploration behavior mathematically.

$$Q(i) = \begin{cases} 0 & \text{if } i = 0 \text{ (No nodes left to explore)} \\ Q(i-1) - 1 + N(v_i) & \text{if } i > 0 \text{ (exploring neighbors of } v_i\text{)} \end{cases}$$

- $Q(i)$ is the size of the queue after processing the i-th node.
- $N(v_i)$ represents the number of unvisited neighbors of node v_i.
- The recursion terminates when all nodes have been visited and the queue becomes empty.

Discussion

- BFS is used for **finding the shortest path** in an unweighted graph.
- It explores neighbors level by level, ensuring that the first time it reaches a node, it uses the shortest path.
- Applications include shortest path algorithms, maze solving, and web crawlers.

DFS

Algorithm

Step	Description
Step-1	Start from the given node (e.g., 1). Mark it as visited and print it.
Step-2	For the current node, explore all its neighbors one by one. If a neighbor is not visited, recursively call DFS on that neighbor.
Step-3	Backtrack once all neighbors of the current node are visited.
Step-4	Repeat the recursive process until all reachable nodes are visited.

Complexity Analysis

- **Time Complexity:**
  $O(V+E)$, where V is the number of vertices and E is the number of edges.
  - Each vertex is visited once, taking $O(V)$ time.
  - Each edge is traversed once, contributing $O(E)$ time.
- **Space Complexity:**
  - $O(V)$ for the recursion stack (in the worst case for a graph with all vertices connected in a path).
  - $O(V)$ for the visited array.

2. Recursive Relation for Depth-First Search (DFS)

DFS explores a path as deep as possible before backtracking. The recursive relation captures the depth-first exploration:

$$T(i) = \begin{cases} 1 & \text{if all neighbors are visited} \\ 1 + \sum_{j \in N(v_i)} T(j) & \text{if } j \text{ is an unvisited neighbor of } v_i \end{cases}$$

- $T(i)$ is the time taken to explore node i and all its neighbors.
- $N(v_i)$ is the set of neighbors of v_i.
- The base case occurs when no unvisited neighbors remain.

Discussion

- DFS is used in **topological sorting**, **connected components detection**, and **detecting cycles** in graphs.
- Unlike BFS, DFS goes as deep as possible along a branch before backtracking.
- Recursive DFS is elegant but may hit stack overflow for large graphs (iterative DFS avoids this).

Travelling Salesman Problem using Branch and Bound

Algorithm

Step	Description
Step-1	Compute the **row-wise minimum edge** for each city. Sum these minimums to get a bound. This is the initial lower bound.
Step-2	Start from the first city and explore all unvisited cities. Track the path and the cost incurred so far.
Step-3	For each new node (city), subtract the minimum edge for the previous node from the bound. Update the cost.
Step-4	If the current cost + updated bound is less than the minimum cost so far, continue exploring further.
Step-5	If a complete path is formed, compare the total cost with the minimum cost. If it is smaller, update the minimum cost and store the path.
Step-6	Backtrack if the current bound exceeds the minimum cost. Explore other branches.
Step-7	Repeat the process until all possible paths are explored. Finally, print the path with the minimum cost.

Complexity Analysis

- **Time Complexity:**
  $O(n!)$ in the worst case, where n is the number of cities.
  However, using **branch and bound**, the search space is pruned, resulting in an **average case** complexity of around $O(2^n \cdot n^2)$.
- **Space Complexity:**
  $O(n^2)$ for storing the cost matrix and additional space for recursion stack.

3. Recursive Relation for Travelling Salesman Problem (TSP)

In the **Branch and Bound** approach to TSP, we recursively try all possible paths, pruning those that exceed the current minimum cost.

$$T(i, S) = \begin{cases} \text{cost}(i, 0) & \text{if } S = \emptyset \text{ (all cities visited)} \\ \min_{j \in S} \left(\text{cost}(i, j) + T(j, S \setminus \{j\}) \right) & \text{otherwise} \end{cases}$$

- $T(i, S)$ is the minimum cost to visit all cities in set S starting from city i.
- The base case occurs when all cities are visited (i.e., $S = \emptyset$).
- For each remaining city j, we compute the minimum cost by moving to j and solving the subproblem for the remaining cities.

Discussion

- TSP is a classic **NP-hard problem**.
- **Branch and Bound** optimizes the naive solution by **pruning paths** where the lower bound already exceeds the current minimum cost.
- It is useful in **routing problems**, **logistics**, and **scheduling**.
- While the **exact solution** is feasible for small inputs, **heuristic solutions** (e.g., genetic algorithms or simulated annealing) are often used for large-scale problems.