1. Implement Linear Search in Python

🔹 What is Linear Search?

Linear Search is a simple algorithm that checks every element in the list one by one until it finds the target value or reaches the end.

🔹 Python Code:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Found
    return -1  # Not found
```

🔹 How it Works:

We loop through each item in the list.

If any item matches the target, we return its index.

If we finish the loop and don't find it, we return -1.

✅ Result:

Searching for 5 in [1, 3, 5, 7, 9] returns index 2

✅ 2. Implement Binary Search in Python

🔹 What is Binary Search?

Binary Search is a faster method that only works on sorted arrays. It divides the array into halves and eliminates one half each time.

🔹 Python Code:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

🔹 How it Works:

Start with the middle item.

If it's your target, return it.

If the target is smaller, search the left half.

If it's larger, search the right half.

✅ Result:

Searching for 5 in [1, 3, 5, 7, 9] returns index 2

✅ 3. Implement Depth-First Search (DFS) using adjacency list

🔹 What is DFS?

DFS is used to explore a graph by going deep into one branch before backtracking.

🔹 Graph (Adjacency List):

```
graph = {
   'A': ['B', 'C'],
```

```
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```

🔹 Python Code:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited
```

🔹 How it Works:

Use recursion to visit one node and its children deeply.

Keep track of visited nodes to avoid cycles.

✅ Result:

DFS from A visits: {'A', 'B', 'C', 'D', 'E', 'F'}

✅ 4. Implement Breadth-First Search (BFS) using adjacency list

🔹 What is BFS?

BFS explores all neighbors of a node before moving deeper. It's like a level-by-level exploration.

🔹 Python Code:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
```

```
            queue.append(neighbor)
    return visited
```

🔹 How it Works:

Use a queue to explore nodes level by level.

Each node is visited once and its neighbors added to the queue.

✅ Result:

BFS from A visits: {'A', 'B', 'C', 'D', 'E', 'F'}

✅ 5. Solve the 8-puzzle problem (Concept only)

🔹 What is the 8-Puzzle Problem?

A 3x3 board with 8 tiles and 1 empty space. You move tiles by sliding them into the empty spot. Goal: Reach a target configuration from an initial one.

🔹 Concept using DFS or BFS:

Each board configuration is a state.

Moves: Up, Down, Left, Right (if valid).

BFS is preferred to find the shortest path.

DFS can be used, but may go too deep.

🔹 Key Terms:

Initial state → Current arrangement

Goal state → Desired arrangement

Successor → New state after one move

We won't write full code here since only concept is asked.

✅ 6. Research Report: A, Greedy Best-First Search*

🔹 A* Search Algorithm:

Combines cost so far (g(n)) and heuristic (h(n))

Formula: f(n) = g(n) + h(n)

Complete and optimal if h(n) is admissible.

🔹 Greedy Best-First Search:

Uses only heuristic: f(n) = h(n)

Faster but may not give shortest path.

🔹 Real-Life Applications:

Algorithm	Applications
A*	GPS navigation, Robotics, AI games
Greedy BFS	Puzzle solving, simple path finding in maps
Both	Used in AI planning, pathfinding, game development
