📌 1. What is a Graph?

A **graph** is a data structure made up of nodes (vertices) and edges (connections between nodes).
Graphs can be:

- **Directed** (edges have direction) or **Undirected**
- **Weighted** (edges have values) or **Unweighted**

Graphs are used to model networks, relationships, and connections.

🧱 2. Graph Representations

a. Adjacency List

- Each node stores a list of its neighbors.
- **Space Complexity:** O(V + E)
  - **Why?**
    - O(V): You need space for each vertex (the keys in the dictionary).
    - O(E): Each edge is stored once (undirected) or twice (directed), so total space is proportional to the number of edges.
- **Efficient for:** Sparse graphs (few edges).

```
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0, 3],
    3: [1, 2]
}
```

b. Adjacency Matrix

- 2D array where `matrix[i][j] = 1` if there's an edge from i to j, else 0.
- **Space Complexity:** O(V^2)
  ◦ **Why?**
    - You must store a value (0 or 1) for every possible pair of vertices, even if there is no edge between them.
    - For V vertices, that's $V \times V = V^2$ entries.
- **Efficient for:** Dense graphs (many edges).

```
adj_matrix = [
    [0, 1, 1, 0],  # Node 0 connects to 1 and 2
    [1, 0, 0, 1],  # Node 1 connects to 0 and 3
    [1, 0, 0, 1],  # Node 2 connects to 0 and 3
    [0, 1, 1, 0]   # Node 3 connects to 1 and 2
]
```

🧭 3. Depth-First Search (DFS)

- Explores as far as possible along each branch before backtracking.
- **Time Complexity:** O(V + E)
  ◦ **Why?**
    - Every vertex is visited once (O(V)).
    - Every edge is explored once (O(E)).
    - Total work is proportional to the number of vertices plus the number of edges.
- **Space Complexity:** O(V)
  ◦ **Why?**
    - The recursion stack or explicit stack can grow up to the number of vertices in the worst case.

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

🌊 4. Breadth-First Search (BFS)

- Explores all neighbors at the current depth before moving to the next level.
- **Time Complexity:** O(V + E)
  ◦ **Why?**
    - Each vertex is enqueued and dequeued once (O(V)).
    - Each edge is checked once (O(E)).
- **Space Complexity:** O(V)
  ◦ **Why?**
    - The queue and visited set can hold up to all vertices in the worst case.

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
```

🏁 5. Dijkstra’s Algorithm (Shortest Path in Weighted Graph)

- Finds the shortest path from a source to all other nodes in a weighted graph with non-negative weights.
- Uses a priority queue (min-heap).
- **Time Complexity:** O((V + E) log V)
  ◦ **Why?**
    - Each vertex can be inserted into the heap once (O(V) insertions).
    - Each edge can cause a decrease-key operation (O(E) operations).
    - Each heap operation (insert or decrease-key) is O(log V).
    - Total: O((V + E) log V)
- **Space Complexity:** O(V)
  ◦ **Why?**
    - The distance table and heap store up to V entries.

```python
import heapq

def dijkstra(graph, start):
    heap = [(0, start)]
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    while heap:
        curr_dist, node = heapq.heappop(heap)
        if curr_dist > distances[node]:
            continue
        for neighbor, weight in graph[node]:
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))
    return distances

# Example weighted graph (adjacency list)
weighted_graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: []
}

print(dijkstra(weighted_graph, 0))  # Output: {0: 0, 1: 3, 2: 1, 3: 4}
```

🧪 6. Practice Problems

a. Traverse a Graph Using DFS and BFS

```
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0, 3],
    3: [1, 2]
}

print("DFS:")
dfs(graph, 0)  # Output: 0 1 3 2

print("\nBFS:")
bfs(graph, 0)  # Output: 0 1 2 3
```

---

b. Find Shortest Path Using Dijkstra’s Algorithm

Already shown above with `weighted_graph` and `dijkstra` function.

---

c. Detect a Cycle in an Undirected Graph (DFS)

```
def has_cycle(graph):
    visited = set()
    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, -1):
                return True
    return False

# Example 1: Graph with a cycle
graph_with_cycle = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1, 3],
    3: [2]
}

print(has_cycle(graph_with_cycle))  # Output: True
```

```
# Example 2: Graph without a cycle
graph_without_cycle = {
    0: [1],
    1: [0, 2],
    2: [1, 3],
    3: [2]
}

print(has_cycle(graph_without_cycle))  # Output: False
```

Explanation:

- `graph_with_cycle` contains a cycle: 0-1-2-0.
- `graph_without_cycle` is a simple path: 0-1-2-3, with no cycles.

📊 Summary Table

Representation

Representation	Space Complexity	Why?
Adjacency List	O(V + E)	Store each vertex and each edge
Adjacency Matrix	$O(V^2)$	Store every possible vertex pair

Algorithms

Algorithm	Time Complexity	Why?
DFS/BFS	O(V + E)	Visit every node and edge
Dijkstra	O((V + E) log V)	Heap for shortest path, process all edges