# Graph: DFS (Recursive) — Coding Interview Notes (Light Theme)

## General Pattern Template

```
def fn(graph):
    def dfs(node):
        ans = 0
        # do some logic
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                ans += dfs(neighbor)

        return ans

    seen = {START_NODE}
    return dfs(START_NODE)
```

**Concept:**
The **Depth-First Search (DFS)** recursive pattern for graphs explores nodes deeply before backtracking. Nodes are typically numbered 0 to n–1, and the graph is represented as an **adjacency list** (a list of neighbor lists).

This approach is useful for exploring connected components, detecting cycles, finding paths, or performing topological sorts.
**Time Complexity:** O(V + E) **Space Complexity:** O(V) for recursion and visited set.

## Key Ideas

1 Use a set (or boolean array) `seen` to avoid revisiting nodes.

2 Recurse into all unvisited neighbors from the current node.

3 DFS works for both directed and undirected graphs.

4 For disconnected graphs, run DFS from every unvisited node to cover all components.

## Example 1: Count Connected Components (Undirected Graph)

**Goal:** Count the number of connected components in an undirected graph.
**Approach:** Run DFS from every unseen node, incrementing count for each DFS call.

```
def count_components(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    seen = set()

    def dfs(node):
        for nei in graph[node]:
            if nei not in seen:
                seen.add(nei)
                dfs(nei)

    count = 0
    for i in range(n):
        if i not in seen:
            seen.add(i)
            dfs(i)
            count += 1

    return count

# Example
print(count_components(5, [[0,1],[1,2],[3,4]]))  # Output: 2
```

## Example 2: Detect Cycle in Directed Graph

**Goal:** Determine if a directed graph has a cycle.
**Approach:** Use two sets — `visiting` (current recursion stack) and `visited` (fully processed nodes). If we revisit a node in `visiting`, a cycle exists.

```
def has_cycle(graph):
    visited, visiting = set(), set()

    def dfs(node):
        if node in visiting:
            return True
        if node in visited:
            return False

        visiting.add(node)
        for nei in graph[node]:
            if dfs(nei):
                return True
        visiting.remove(node)
        visited.add(node)
        return False

    for node in range(len(graph)):
        if dfs(node):
            return True
    return False

# Example
graph = [[1], [2], [0]]  # 0→1→2→0 forms a cycle
print(has_cycle(graph))  # Output: True
```

## Example 3: Topological Sort (Directed Acyclic Graph)

**Goal:** Return a valid topological ordering of a directed acyclic graph (DAG).
**Approach:** Perform DFS and append nodes to result after processing all neighbors (postorder).

```
def topo_sort(graph):
    n = len(graph)
    seen = set()
    order = []

    def dfs(node):
        seen.add(node)
        for nei in graph[node]:
            if nei not in seen:
                dfs(nei)
        order.append(node)

    for i in range(n):
        if i not in seen:
            dfs(i)

    return order[::-1]

# Example
graph = [[1, 2], [3], [3], []]
print(topo_sort(graph))  # Output: [0, 2, 1, 3] (valid topological order)
```

## Summary Table

ProblemGraph TypeLogicComplexity Count connected componentsUndirectedDFS from unseen nodesO(V+E) Detect cycleDirectedTrack recursion stackO(V+E) Topological sortDAGPostorder traversalO(V+E)