Graph: DFS (Iterative) — Coding Interview Notes (Light Theme)

General Pattern Template

```
def fn(graph):
    stack = [START_NODE]
    seen = {START_NODE}
    ans = 0

    while stack:
        node = stack.pop()
        # do some logic
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                stack.append(neighbor)

    return ans
```

Concept:
The **Iterative Depth-First Search (DFS)** pattern for graphs uses an explicit stack to simulate recursion. It explores nodes deeply before backtracking, like recursive DFS, but gives you more control over traversal order and avoids recursion limits.

Typical Uses: Graph traversal, connected components, cycle detection, and topological ordering.
Time Complexity: O(V + E) **Space Complexity:** O(V) for the stack and visited set.

Key Ideas

1. Use an explicit stack instead of recursion for DFS traversal.
2. Push unvisited neighbors to the stack; pop nodes for processing.
3. The stack order controls traversal (push neighbors in desired order).
4. Useful for large graphs or when recursion depth may exceed system limits.

Example 1: Count Connected Components (Undirected Graph)

Goal: Count the number of connected components using an iterative DFS.
Approach: For each unseen node, run DFS using a stack and mark visited nodes.

```
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()
    count = 0

    for i in range(n):
        if i not in seen:
            stack = [i]
            seen.add(i)
            while stack:
                node = stack.pop()
                for nei in graph[node]:
                    if nei not in seen:
                        seen.add(nei)
                        stack.append(nei)
            count += 1

    return count

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

Example 2: Detect Cycle in Undirected Graph

Goal: Check if an undirected graph contains any cycle.
Approach: Track parent nodes during DFS; if a visited neighbor ≠ parent, a cycle exists.

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

    seen = set()

    for start in range(n):
        if start not in seen:
            stack = [(start, -1)]
            seen.add(start)
            while stack:
                node, parent = stack.pop()
                for nei in graph[node]:
                    if nei == parent:
                        continue
                    if nei in seen:
                        return True
                    seen.add(nei)
                    stack.append((nei, node))
    return False

# Example
print(has_cycle(3, [[0,1],[1,2],[2,0]]))  # Output: True
```

Example 3: Topological Sort (Directed Graph)

Goal: Perform an iterative topological sort on a DAG.
Approach: Use a stack to simulate postorder traversal.

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

    for start in range(n):
        if start not in seen:
            temp_stack = [(start, False)]
            while temp_stack:
                node, visited = temp_stack.pop()
                if visited:
                    order.append(node)
                else:
                    if node not in seen:
                        seen.add(node)
                        temp_stack.append((node, True))
                        for nei in reversed(graph[node]):
                            if nei not in seen:
                                temp_stack.append((nei, False))
    return order[::-1]

# Example
graph = [[1, 2], [3], [3], []]
print(topo_sort_iterative(graph))  # Output: [0, 2, 1, 3]
```

Summary Table

ProblemGraph TypeLogicComplexity Count connected componentsUndirectedDFS from unseen nodesO(V+E) Detect cycleUndirectedTrack parent during traversalO(V+E) Topological sortDirected (DAG)Iterative postorderO(V+E)