Graph: BFS (Breadth-First Search) — Coding Interview Notes (Light Theme)

General Pattern Template

```
from collections import deque

def fn(graph):
    queue = deque([START_NODE])
    seen = {START_NODE}
    ans = 0

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

    return ans
```

Concept:
The **Breadth-First Search (BFS)** pattern explores a graph layer by layer using a queue. Starting from a source (START_NODE), BFS visits all neighbors before moving to the next layer.

Typical Uses: Shortest path in unweighted graphs, connected components (undirected), level/distances labeling, bipartite checks.
Complexity: O(V + E) time, O(V) space (queue + visited).

Key Ideas

1. Use a queue to process nodes in FIFO (level) order.
2. Maintain a visited set to avoid revisiting nodes.
3. For shortest paths in unweighted graphs, maintain a distance map: dist[neighbor] = dist[node] + 1.
4. To reconstruct paths, also keep a parent map and backtrack from the target.
5. Run BFS from each unseen node to enumerate connected components in an undirected graph.

Example 1: Shortest Path Length (Unweighted Graph)

Goal: Return the shortest number of edges from src to dst in an unweighted graph.
Approach: Standard BFS with a distance map. Stop once dst is dequeued.

```
from collections import deque
```

```
def shortest_path_length(graph, src, dst):
    if src == dst:
        return 0
    dist = {src: 0}
    q = deque([src])

    while q:
        node = q.popleft()
        for nei in graph[node]:
            if nei not in dist:
                dist[nei] = dist[node] + 1
                if nei == dst:
                    return dist[nei]
                q.append(nei)
    return -1  # unreachable

# Example
graph = [[1,2], [2], [3], []]  # 0->1->2->3 path length 3
print(shortest_path_length(graph, 0, 3))  # Output: 3
```

Example 2: Count Connected Components (Undirected)

Goal: Count how many connected components an undirected graph has.
Approach: BFS from each unseen node, marking all nodes in that component.

```
from collections import deque

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:
            seen.add(i)
            q = deque([i])
            while q:
                node = q.popleft()
                for nei in graph[node]:
                    if nei not in seen:
                        seen.add(nei)
                        q.append(nei)
            count += 1
    return count

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

Example 3: Compute Distances from START_NODE

Goal: Label each node with its shortest distance (in edges) from a start node in an unweighted graph.
Approach: BFS using a distance map; unreachable nodes get None.

```
from collections import deque

def distances_from_start(graph, start):
    n = len(graph)
    dist = {start: 0}
    q = deque([start])

    while q:
        node = q.popleft()
        for nei in graph[node]:
            if nei not in dist:
                dist[nei] = dist[node] + 1
                q.append(nei)

    # Map nodes not reached to None
    result = [None] * n
    for i in range(n):
        result[i] = dist.get(i, None)
    return result

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

Summary Table

Problem	Graph Type	Logic	Complexity
Shortest path length	Unweighted	Distance map, early exit at dst	O(V+E)
Connected components	Undirected	BFS from each unseen node	O(V+E)
Distances from start	Unweighted	Level-by-level labeling	O(V+E)