.

🌐 Graph Interview Quick-Check Pattern

A tactical reference guide to help you confidently solve graph-based interview problems — whether it's DFS, BFS, cycles, or components.

---

1. 🧠 When to Use a Graph Approach

Ask yourself:

- ❓ Is the input or problem about **nodes connected to other nodes**?
- ❓ Are you working with a **network**, **map**, or **grid with directions**?
- ❓ Is the question asking about **reachability**, **paths**, **cycles**, or **connected groups**?
- ❓ Do you need to **build the graph** yourself from raw data?

> 💡 Graph problems almost always involve **relationships**, **recursion or iteration**, and **visited tracking**.

---

2. 🧰 Common Graph Traversal Techniques

Technique	When to Use
BFS (Queue)	Shortest path, levels, spreading/infection
DFS (Stack or Recursion)	Component discovery, path exploration
Union-Find / DSU	Connected components, cycle detection (undirected)
Topological Sort	Scheduling tasks, course prerequisites
Dijkstra / A*	Weighted shortest paths

---

3. 🔧 Graph Construction Patterns

Input Type	Adjacency Representation Example
Edge List	`[ [u, v], ... ]` → use `adjList[u].push(v)`
Grid	2D matrix → use directions: `[[0,1], [1,0], ...]`
Weighted Graph	`[ [u, v, w], ... ]` → `adjList[u].push([v, w])`
Undirected	Add both `u → v` and `v → u`
Directed	Add `u → v` only

---

4. 🧪 Must-Know Graph Templates

✅ DFS – Count Connected Components

```
function countComponents(n, edges) {
  const graph = Array.from({ length: n }, () => []);
  for (let [u, v] of edges) {
    graph[u].push(v);
    graph[v].push(u);
  }

  const visited = new Set();
  let count = 0;

  function dfs(node) {
    if (visited.has(node)) return;
    visited.add(node);
    for (let neighbor of graph[node]) dfs(neighbor);
  }

  for (let i = 0; i < n; i++) {
    if (!visited.has(i)) {
      dfs(i);
      count++;
```

```
    }
  }

  return count;
}
```

✅ BFS – Shortest Path in Unweighted Graph

```
function shortestPath(n, edges, start, end) {
  const graph = Array.from({ length: n }, () => []);
  for (let [u, v] of edges) {
    graph[u].push(v);
    graph[v].push(u);
  }

  const queue = [[start, 0]];
  const visited = new Set([start]);

  while (queue.length) {
    const [node, dist] = queue.shift();
    if (node === end) return dist;
    for (let neighbor of graph[node]) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.push([neighbor, dist + 1]);
      }
    }
  }

  return -1;
}
```

✅ Topological Sort (Kahn's Algorithm)

```
function topoSort(numCourses, prerequisites) {
  const graph = Array.from({ length: numCourses }, () => []);
```

```
  const inDegree = Array(numCourses).fill(0);

  for (let [course, pre] of prerequisites) {
    graph[pre].push(course);
    inDegree[course]++;
  }

  const queue = [];
  for (let i = 0; i < numCourses; i++) {
    if (inDegree[i] === 0) queue.push(i);
  }

  const order = [];
  while (queue.length) {
    const node = queue.shift();
    order.push(node);
    for (let neighbor of graph[node]) {
      inDegree[neighbor]--;
      if (inDegree[neighbor] === 0) queue.push(neighbor);
    }
  }

  return order.length === numCourses ? order : [];
}
```

✅ Detect Cycle in Directed Graph (DFS + Rec Stack)

```
function hasCycle(graph) {
  const visited = new Set();
  const stack = new Set();

  function dfs(node) {
    if (stack.has(node)) return true;
    if (visited.has(node)) return false;

    visited.add(node);
    stack.add(node);

    for (let neighbor of graph[node]) {
```

```
      if (dfs(neighbor)) return true;
    }

    stack.delete(node);
    return false;
  }

  for (let node in graph) {
    if (dfs(node)) return true;
  }

  return false;
}
```

---

5. 🧱 Edge Cases to Watch For

- Disconnected graphs → multiple components
- Cycles (directed vs undirected)
- Nodes with no outgoing edges (sink nodes)
- One-way connections (e.g., only `u → v`)
- Empty input or one node
- Graph contains duplicate or self edges

> 🧠 Always track visited nodes — infinite loops come from visiting again.

---

6. 🧠 Mental Model for Graph Problems

Question Type	Pattern / Strategy
“How many groups?”	DFS/BFS (connected components)
“Shortest path”	BFS (unweighted), Dijkstra (weighted)
“Is there a cycle?”	DFS + visited/recStack / Union-Find

“Can I finish all tasks?”	Topo sort + inDegree
“Return a path or all paths”	Backtracking (DFS with path array)
“Spread from multiple sources”	Multi-source BFS

🔁 Problem Solving Loop

1. 🔍 Should I use DFS, BFS, or Topo Sort?
2. 🧱 Is it a **directed** or **undirected** graph?
3. 📌 Do I need to **build an adjacency list** first?
4. ✅ Do I need to track **visited nodes** or **recursion stack**?
5. 🔗 Is this about **reachability**, **components**, or **ordering**?

✅ Final Interview Checklist

- Did I build the graph correctly (directed vs undirected)?
- Am I tracking visited nodes to avoid revisiting?
- Did I handle disconnected graphs?
- Is this a cycle, topo sort, or shortest path question?
- Are edge cases (empty, one node, self-loop) covered?