

BFS of Graph

● EASY

■ Introduction

Imagine you drop a stone in a pond. Ripples spread outward layer by layer — first the adjacent ring, then the next ring, then the next. BFS (Breadth-First Search) explores a graph exactly like this: level by level, using a Queue.

Why it matters: BFS is the foundation of shortest path algorithms, level-order traversal, finding connected components, and multi-source spreading problems.

■ Core Intuition

BFS Properties

- Uses a QUEUE (FIFO) — process nodes in the order they were discovered.
- Visits all neighbors at distance 1, then distance 2, then distance 3...
- Guarantees SHORTEST PATH (in terms of edge count) in unweighted graphs.
- Needs a visited[] array to avoid re-processing nodes (prevent infinite loops).

■■ BFS Algorithm Steps

Step 1: Add starting node to queue, mark it visited.

Step 2: While queue is not empty: dequeue front node, add to result.

Step 3: For each unvisited neighbor of current node: mark visited, enqueue.

Step 4: Repeat until queue is empty. All reachable nodes are processed.

■ Visual — BFS Traversal

BFS Trace

Graph (Adjacency List): BFS from node 0:

0 -- 1 Queue: [0] Visited: {0}

0 -- 2 Deq 0 → enq 1,2 Queue:[1,2]

1 -- 3 Deq 1 → enq 3 Queue:[2,3]

1 -- 4 Deq 2 → enq 5 Queue:[3,5] (4 not neighbor of 2)

2 -- 5 Deq 3 → no new Queue:[5]

Deq 4 → no new Queue:[]

BFS Order: 0 → 1 → 2 → 3 → 4 → 5 (Level by Level)

Level 0: [0]

Level 1: [1, 2]

Level 2: [3, 4, 5]

■ Java Implementation

Java

```
import java.util.*;

class Solution {
    // BFS traversal – returns list of nodes in BFS order

    public ArrayList<Integer> bfsOfGraph(int V, ArrayList<ArrayList<Integer>> adj) {
        ArrayList<Integer> result = new ArrayList<>();

        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();

        // Start from node 0
        queue.add(0);
        visited[0] = true;

        while (!queue.isEmpty()) {
            int node = queue.poll(); // Dequeue
            result.add(node); // Process

            // Enqueue all unvisited neighbors
            for (int neighbor : adj.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }

        return result;
    }

    // For disconnected graphs – BFS from every unvisited node
    for (int i = 0; i < V; i++) {
        if (!visited[i]) bfs(i, adj, visited, result);
    }
}
```

■■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	Visit each vertex once (V) + traverse each edge once (E)
Space	$O(V)$	$O(V)$	Queue + visited array, both at most V nodes

Common Mistakes

Avoid These

- Not marking visited BEFORE enqueueing — causes same node to be added multiple times.
- Forgetting disconnected components — always loop over all nodes for full traversal.
- Using Stack instead of Queue — that gives DFS, not BFS.
- Not handling edge case: $V=0$ or empty graph.

Practice Problems

Case	Time	Space	Why
Easy	GFG	BFS of Graph	Direct implementation
Easy	LC 102	Binary Tree Level Order	BFS on tree
Medium	LC 994	Rotten Oranges	Multi-source BFS
Medium	LC 127	Word Ladder	BFS on implicit graph

Mental Model

Remember Forever

- "BFS is a pandemic spreading level by level. Patient 0 infects neighbors on Day 1, they infect their neighbors on Day 2... Queue tracks who to process next, visited[] tracks who is already infected."