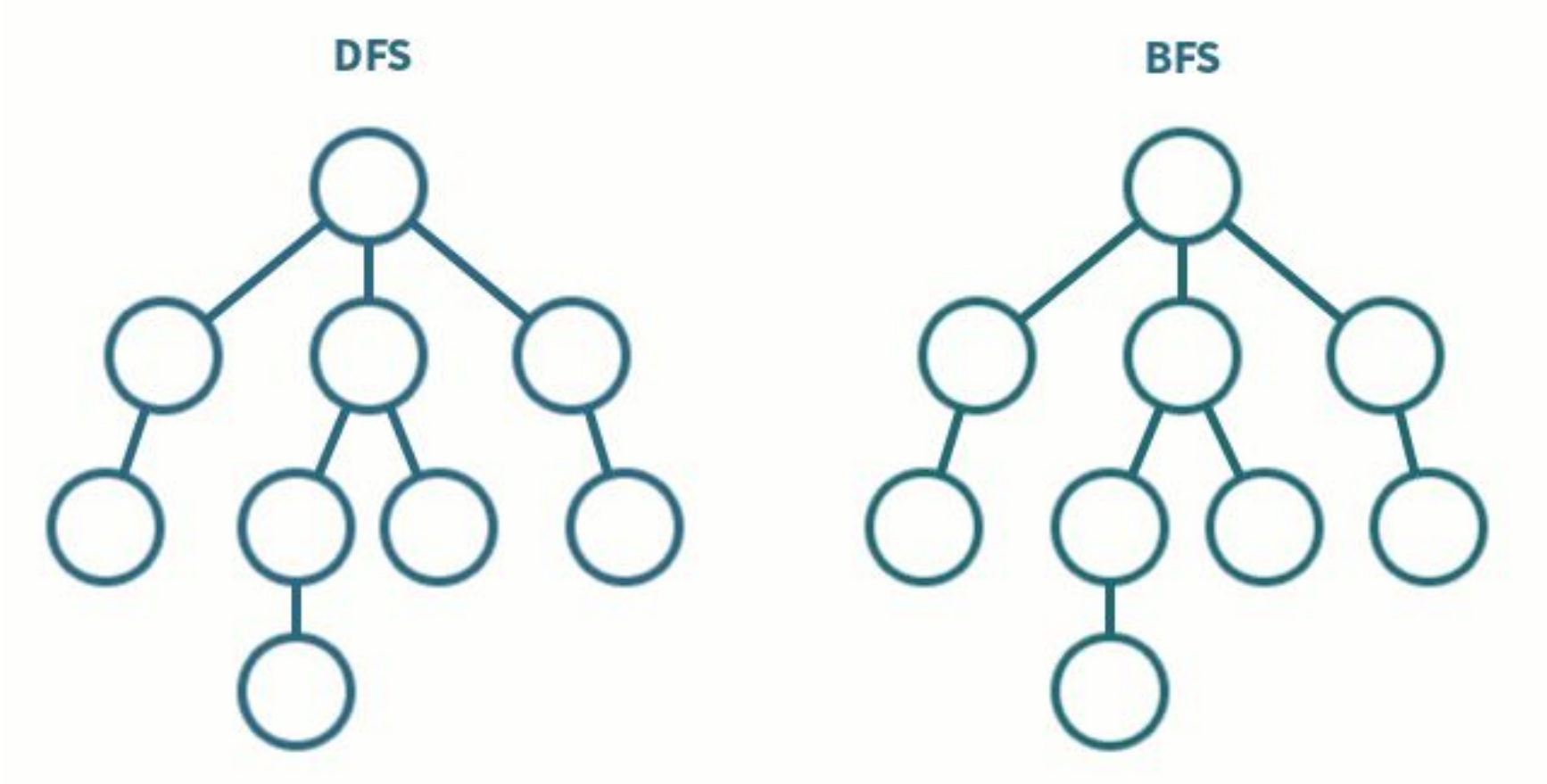
Breadth First Search

BFS vs DFS



BFS:

DFS:

BFS:

Lecture Flow

1) Pre-requisites
2) Definition
3) BFS Variations
4) Applications of BFS
5) Practice questions
6) Quote of the day

Prerequisite

- Introduction to Graph
- Basic understanding of Queue Data Structure
- DFS graph traversal algorithm

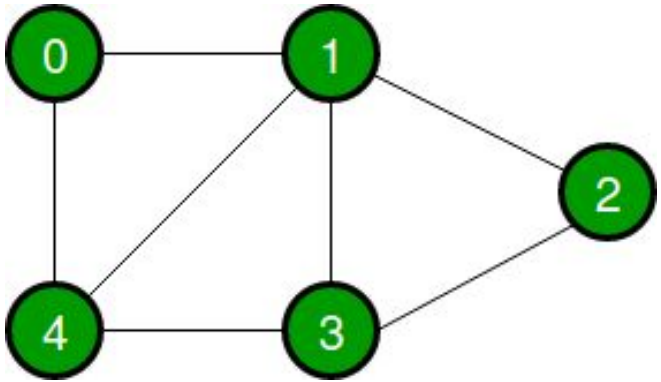
Introduction

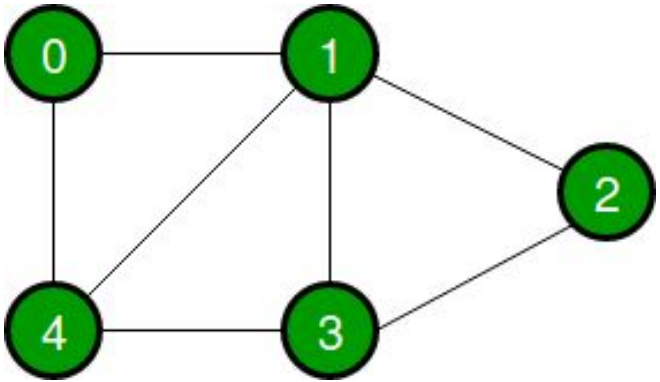
- **BFS (Breadth-First Search)** is a graph traversal algorithm that visits all the nodes of a graph in **breadth-first order.**

- It visits all the nodes at **a given level** before moving on to **the next level**.

- It starts at the root node and explores all the **neighboring nodes** before moving on to the next level.

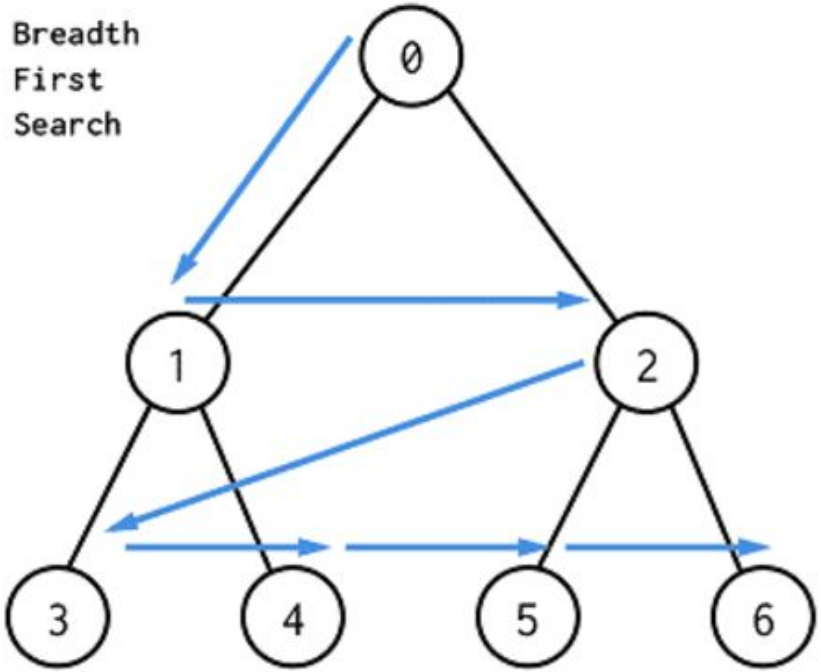
What is the bfs traversal of the graph if we start at 0?

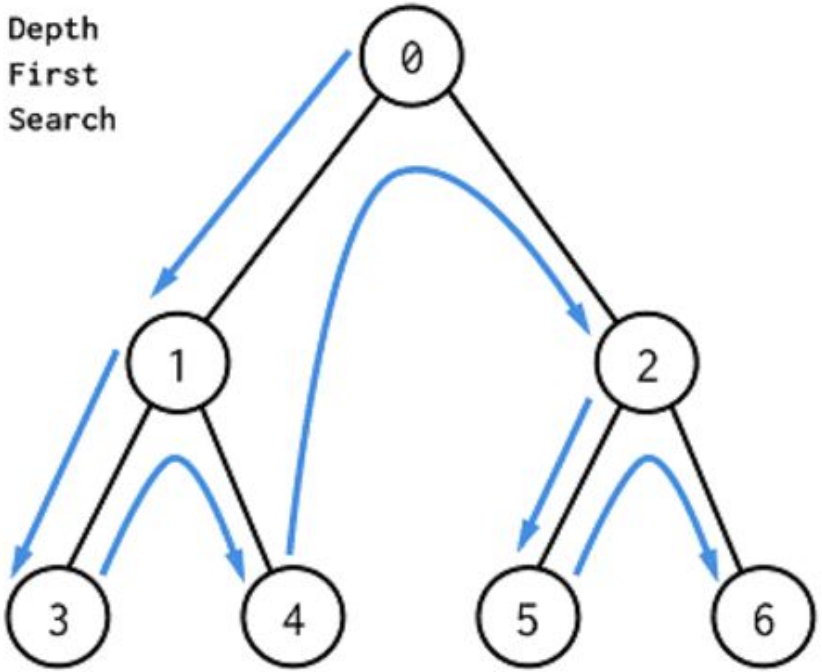




Answer: 0, 1, 4, 2, 3 or 0, 4, 1, 3, 2

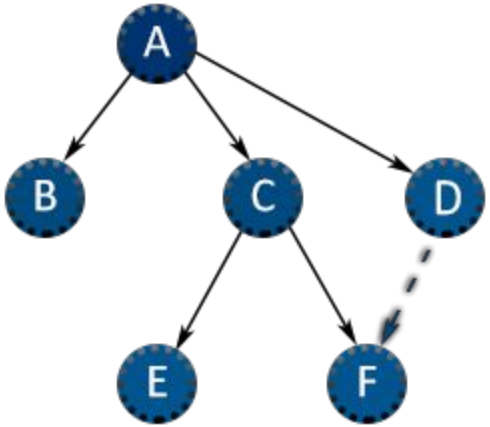
Another example





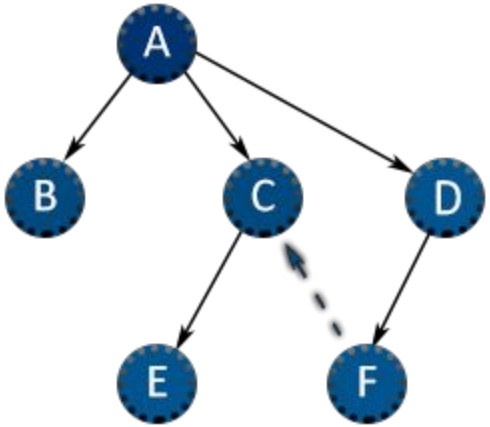
Yet another example

BFS



A B C D E F

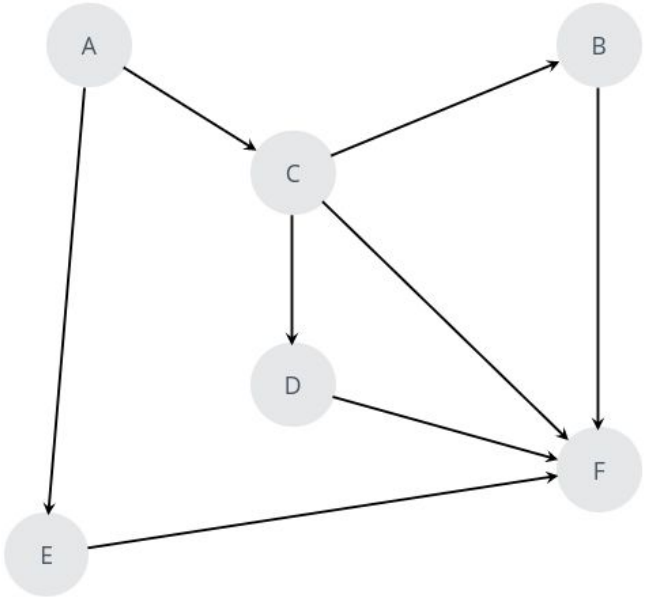
DFS



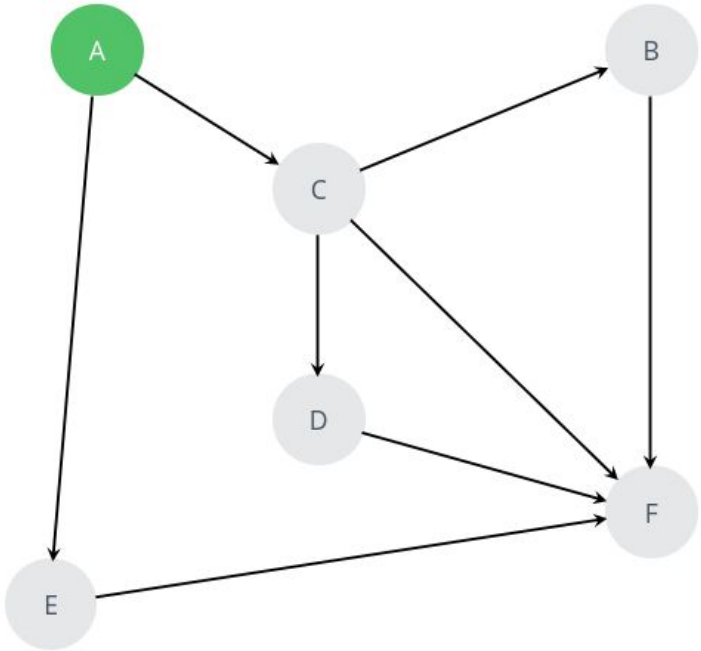
A D F C E B

Visualization

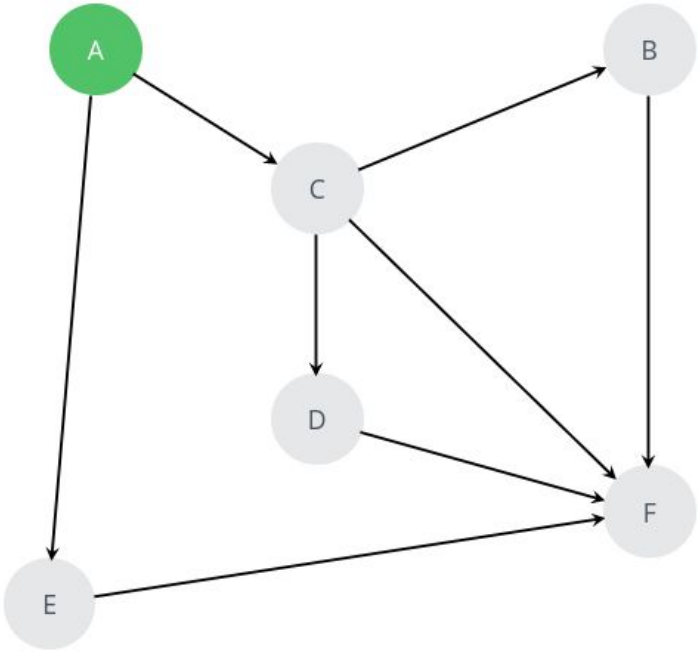
Let’s see how to implement BFS using queue data structure



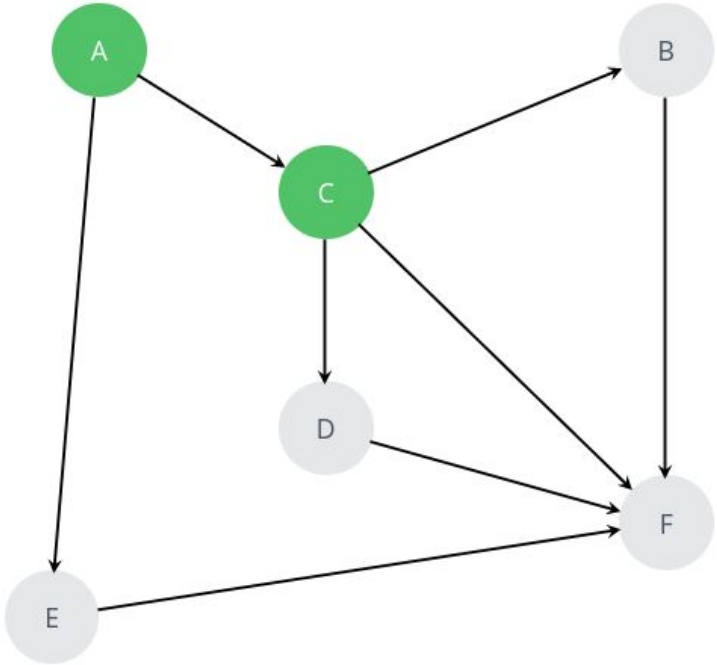
Steps:
Let us look at the details of how a breadth-first search works.



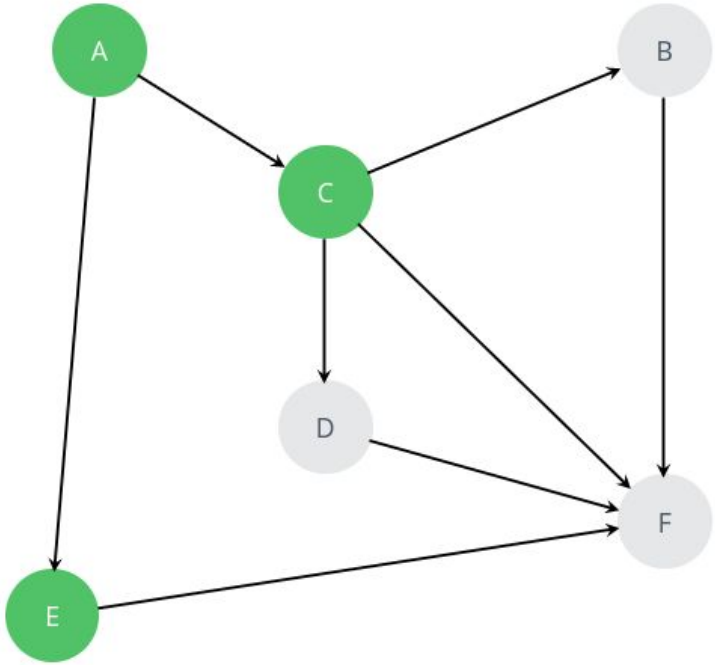
Steps:
Mark and enqueue A



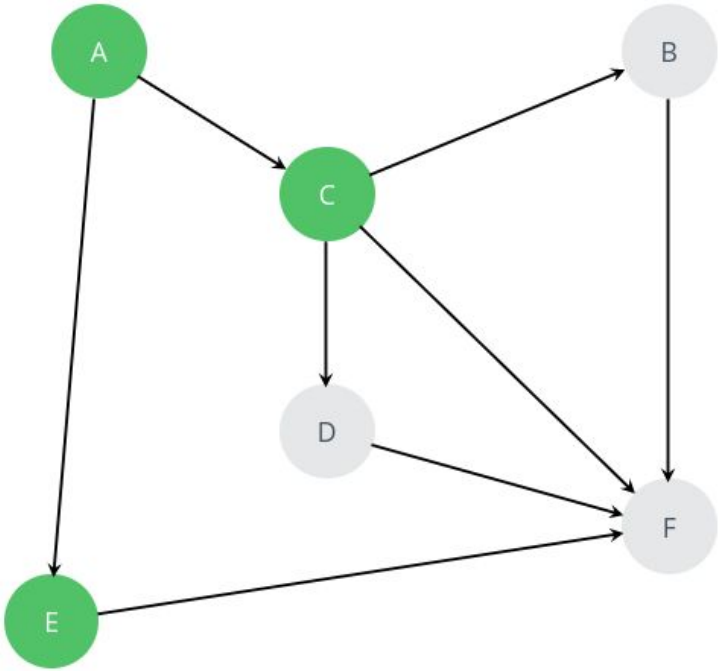
Steps:
Dequeue A



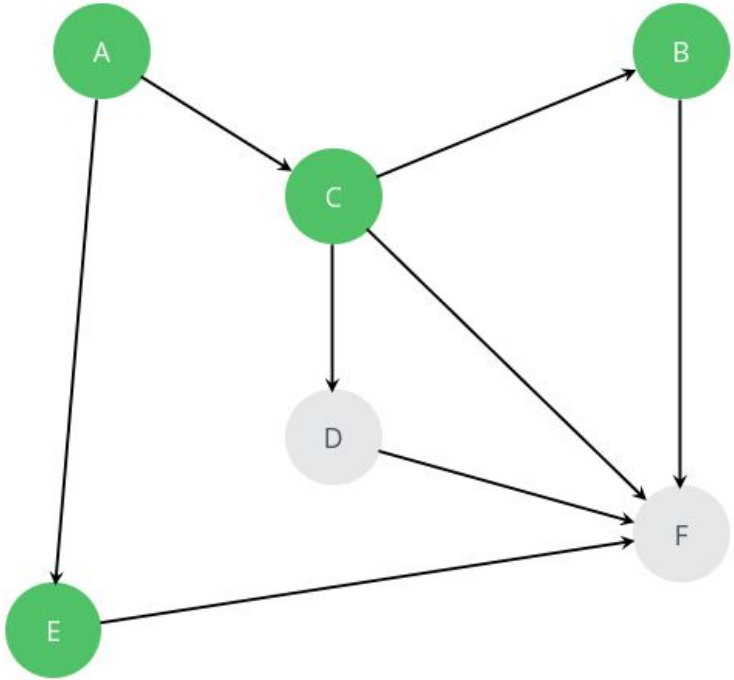
Steps:
Mark and enqueue C



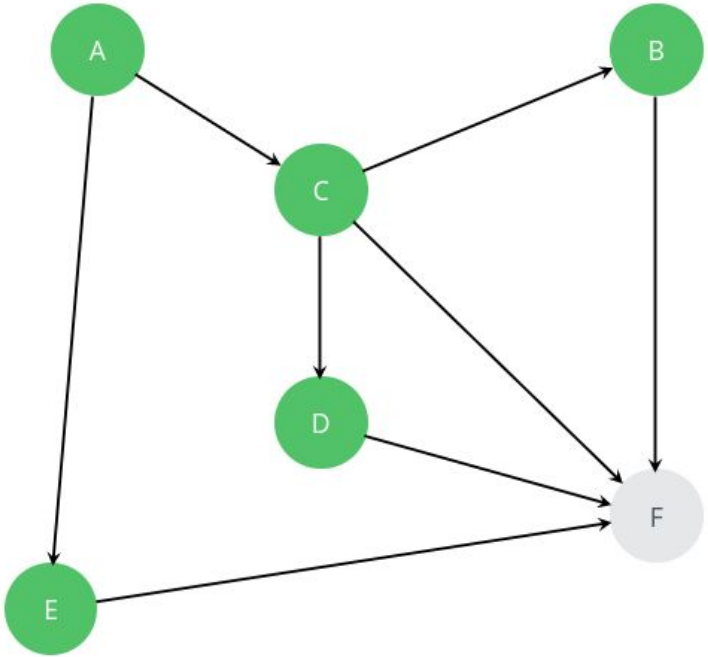
Steps:
Mark and enqueue E



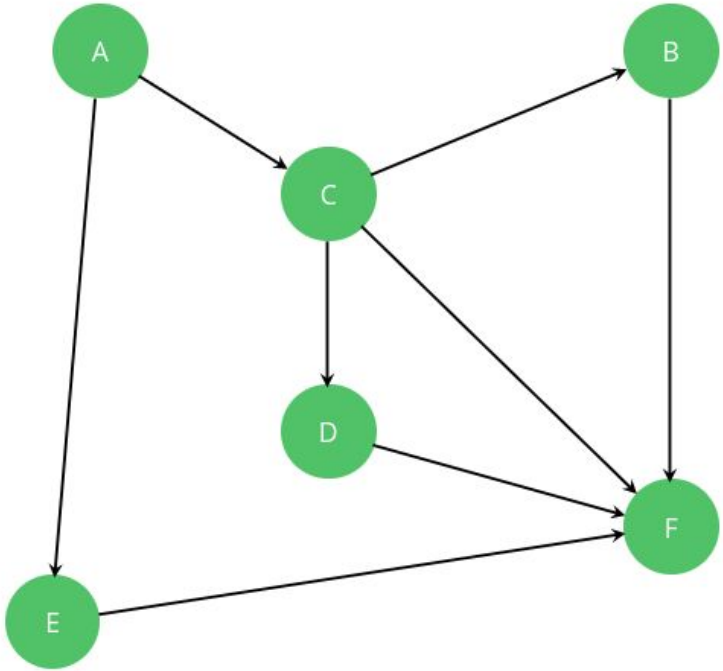
Steps:
Dequeue C



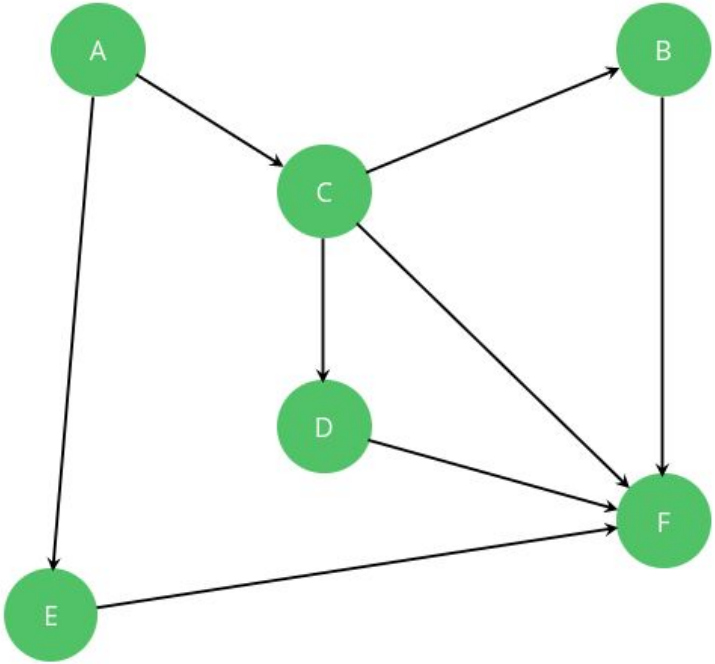
Steps:
Mark and enqueue B



Steps:
Mark and enqueue D

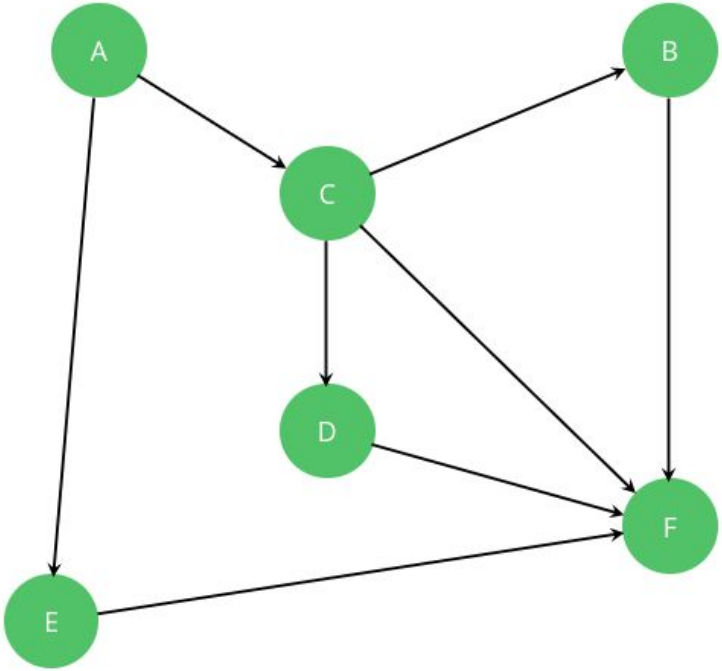


Steps:
Mark and enqueue F



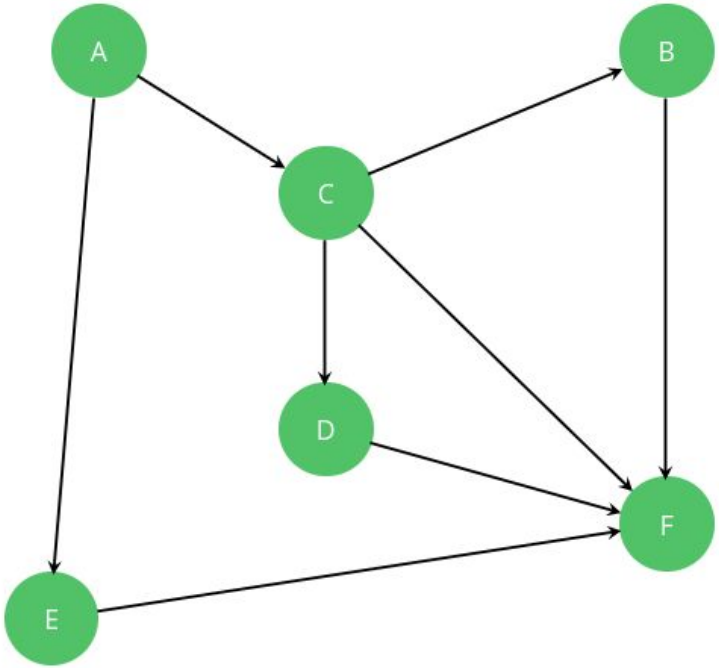
Steps:

Dequeue E

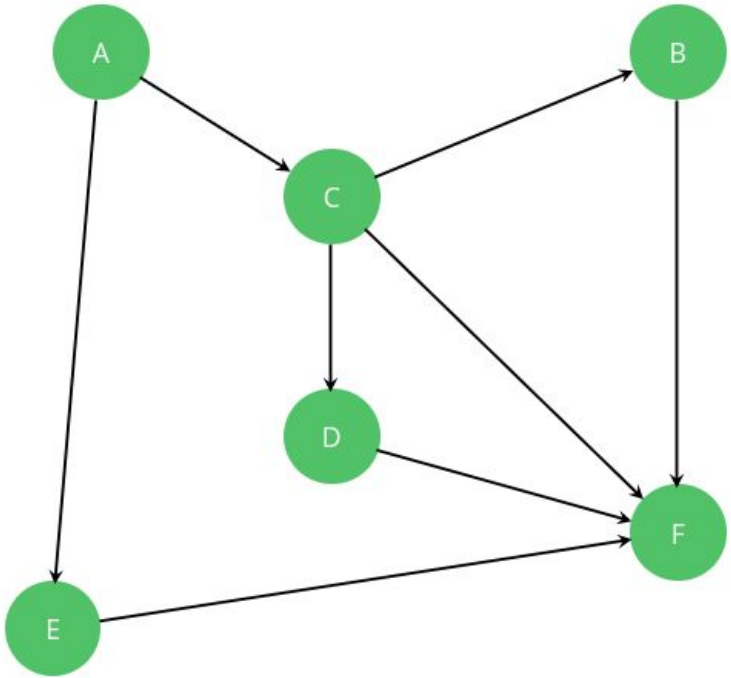


Steps:

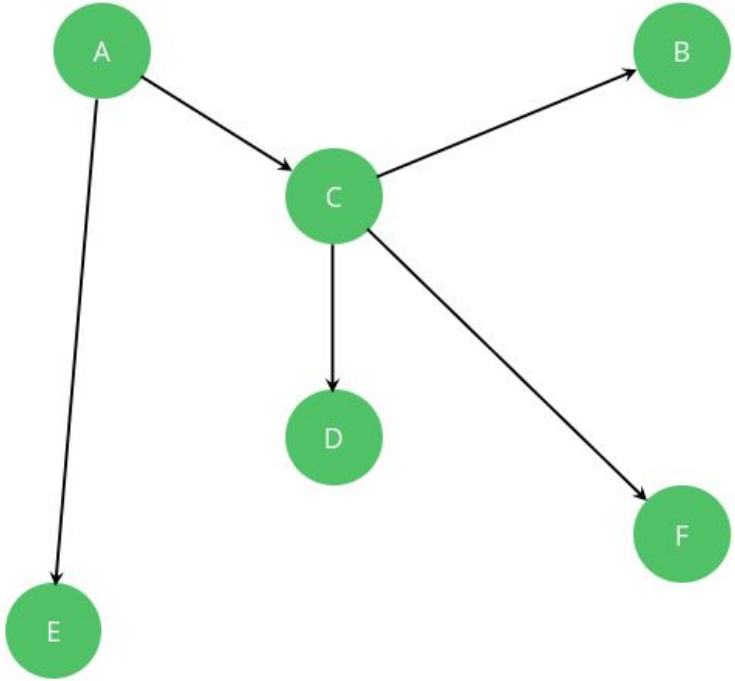
Dequeue B



Steps:
Dequeue D



Steps:
Dequeue F



Steps:

Completed breadth first search graph

Implementation

```
def bfs(graph, node):
    visited = set([node])
    queue = deque([node])

    while queue:
        node = queue.popleft()

        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
```

Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once**, and if it's **a complete graph** we will also **use every edge**.

  Time Complexity = **O (V + E)**

Space Complexity

- We have **V nodes** and we only visit each node at most once.
- **the space complexity is just the space required to input them in the visited set.**

  Space Complexity = **O (V)**

Let’s practice

Easy 2.8K 144

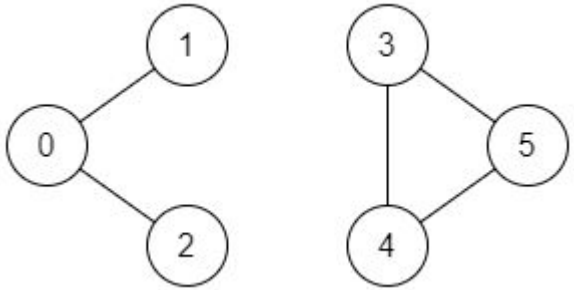
Companies

Find if Path Exists in Graph

There is a **bi-directional** graph with `n` vertices, where each vertex is labeled from `0` to `n - 1` (**inclusive**). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [u_i, v_i]` denotes a bi-directional edge between vertex `u_i` and vertex `v_i`. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` *if there is a* ***valid path*** *from* `source` *to* `destination`, *or* `false` *otherwise.*



Approach

1. Create an adjacency list to represent the graph.
2. Create a visited set.
3. Create a queue and enqueue the source vertex.
4. While the queue is not empty, dequeue a vertex v from the queue.
5. If v is the destination vertex, return true.
6. Mark v as visited and enqueue all **unvisited** neighbors of v.
7. Repeat steps 4-6 until the queue is empty.
8. If the **queue becomes empty** before reaching the destination vertex, return false.

Implementation

```
def validPath(n, edges, source, destination):

        graph = collections.defaultdict(list)
        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)

        visited = set()
        seen[source] = True
        queue = collections.deque([source])

        while queue:
            curr_node = queue.popleft()
            if curr_node == destination:
                return True

            for next_node in graph[curr_node]:
                if next_node not in visited:
                    seen[next_node] = True
                    queue.append(next_node)

        return False
```

Time Complexity

- We have **V nodes and we visit each node exactly once.**
- Since the number of edges is constant for every node, we can say that **E = 2V**. Which we ignore.

  Time Complexity = **O (V)**

Space Complexity

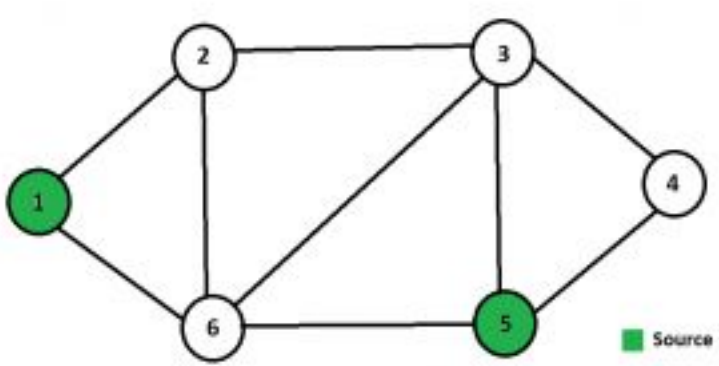
- We store at most M nodes in the queue, where **M is the maximum number of nodes in a level** of the binary tree

  Space Complexity = **O (M)**

Multi-Source BFS

Multi-source BFS

- If we have **multiple starting locations** for our BFS, there is **nothing stopping** us from **appending** all those locations into our starting queue.



Let’s practice

Rotting Oranges

994. Rotting Oranges

Medium 10237 341 Add to List Share

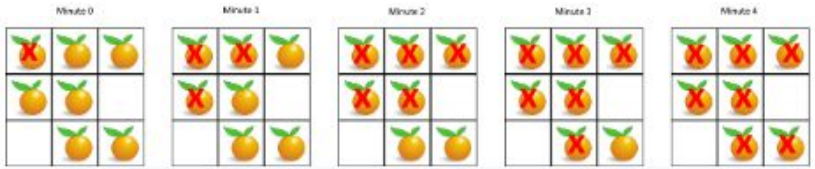
You are given an `m x n` `grid` where each cell can have one of three values:

- `0` representing an empty cell,
- `1` representing a fresh orange, or
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange.* If *this is impossible, return* `-1`.

Example 1:



```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

Example 2:

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left
corner (row 2, column 0) is never rotten,
because rotting only happens 4-directionally.
```

Approach

1. **Initialize** your queue.
2. Traverse through the **grid and insert all rotten orange** indices in to your queue.
3. After the insertions, while we still have elements in our queue, and as long as we have remaining fresh oranges in our grid we will continue our iteration.
    a. In here we will go through **every element in our queue,** and we will pop the leftmost element.
    b. As long as that cell is not in bound and the grid is containing a fresh orange.
        i. We change that orange to rotten, and we **append the new grid** in to our queue.
    c. Once the above steps are completed then we can **decrease the number of fresh** oranges that we have.

4. Now once we are done with going through the queue we can return the minimum time taken to make all oranges in that grid rotten.

```
def orangesRotting(self, grid):
        q = deque()
        time, fresh = 0, 0
        n, m = len(grid), len(grid[0])
        for r in range(n):
            for c in range(m):
                if grid[r][c] == 1:
                    fresh += 1
                if grid[r][c] == 2:
                    q.append([r,c])
        directions = [[0,1], [1,0], [0,-1], [-1,0]]
        while q and fresh > 0:
            for i in range(len(q)):
                r, c = q.popleft()
                for dr, dc in directions:
                    row, col = dr + r, dc + c
                    if (row < 0 or row == n or col < 0 or col == m or grid[row][col] != 1):
                        continue
                    grid[row][col] = 2
                    q.append([row, col])
                    fresh -= 1
            time += 1
        return time if fresh == 0 else -1
```

Time Complexity

- Let **N** be **number of rows** and **M** be **number of columns**.
- We are traversing through the entire grid. Which would take a time complexity of **O(N * M).**

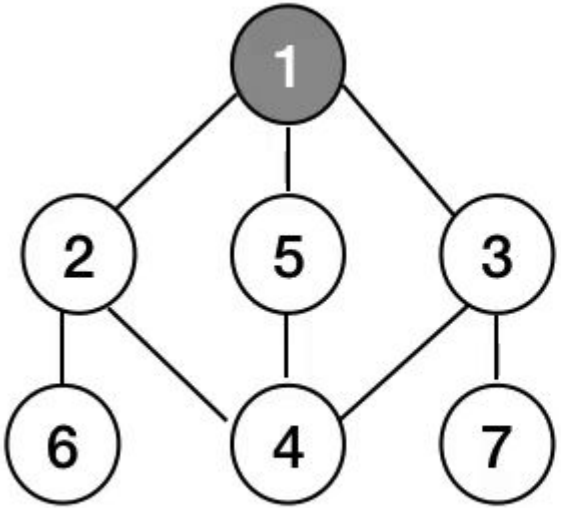
Space Complexity

- Let **N** be **number of rows** and **M** be **number of columns**.
- At worst we might have the entire grid in our queue if all grid positions are occupied by either rotten or fresh oranges, so the space complexity will also be **O(N * M).**

Level Order Traversal

Level order traversal

- BFS can be used for level order traversal by **visiting nodes based on their distance from the root node.**



Let’s practice

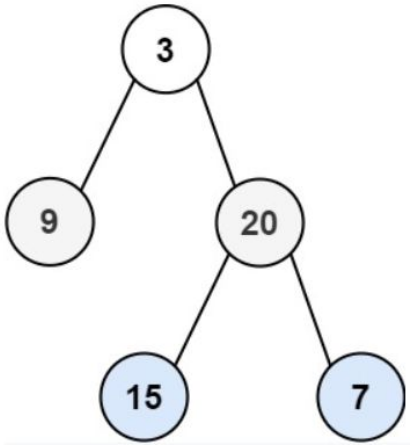
Binary Tree Level Order Traversal

102. Binary Tree Level Order Traversal

Medium 12944 257 Add to List Share

Given the `root` of a binary tree, return *the level order traversal of its nodes' values.* (i.e., from left to right, level by level).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Approach

1. **Initialize** your **queue**,**visited** set, **current level array**, **current level** you are currently at.
2. Check the **current level** with **global level**:
    a. If They are **not equal**, **add** current level **array** to answer and set the current level **array** to **empty**.
3. **Add current** node to current level **array**.
4. **Traverse** through the **neighbours** and insert **neighbour's** with their **level** in to your queue.
5. Repeat step **2, 3 , 4** until you left with **empty queue.**

```python
def levelOrder(self, root):
    levels = []
    level = 0
    queue = deque([(root, level)])
    currLevel = []

    while queue:
        node, nodeLevel = queue.popleft()

        if not node:
            continue

        if nodeLevel != level:
            levels.append(currLevel.copy())
            level += 1
            currLevel = []

        currLevel.append(node.val)
        queue.append((node.left, level + 1))
        queue.append((node.right, level + 1))

    if currLevel:
        levels.append(currLevel.copy())

    return levels
```

Time Complexity

- We have **V nodes and we visit each node exactly once.**
- Since the number of edges is constant for every node, we can say that **E = 2V**. Which we ignore.

  Time Complexity = **O (V)**

Space Complexity

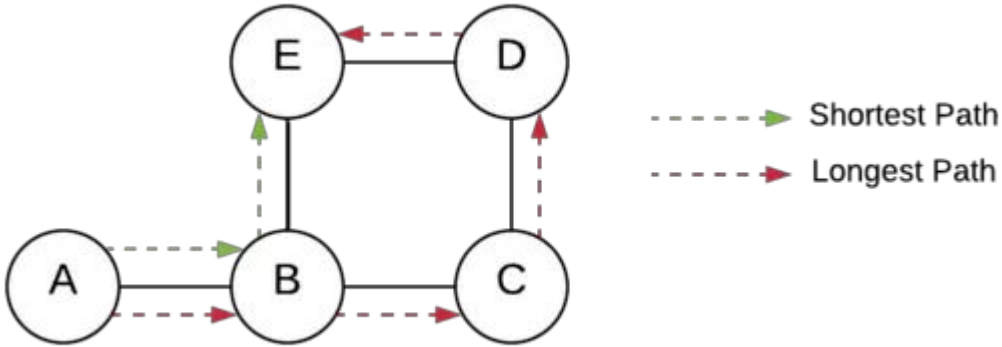
- We store at most M nodes in the queue, where **M is the maximum number of nodes in a level** of the binary tree

  Space Complexity = **O (M)**

Finding shortest path in unweighted graph

Problem

Given an unweighted undirected graph, we have to find the shortest path from the given source to the given destination using the Breadth-First Search algorithm.



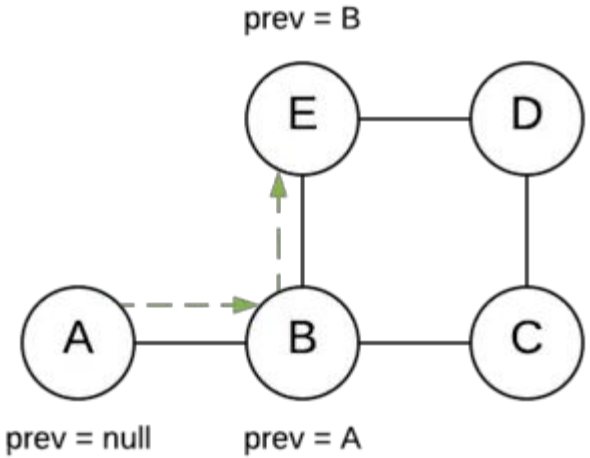
Approach

- Use BFS to traverse a graph or tree.
- Compare visited nodes with the end node. Stop BFS when the end node is found.
- Use either a cleared queue or a boolean flag to mark the end of BFS.

How to trace path from start to end node?

Trace path

- We use **"prev"** array or dictionary to **store the reference** of the **preceding node.**
- Using the "prev" value, we can trace the route back from the end node to the starting node.



The shortest path

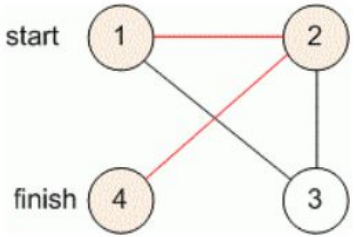
The undirected graph is given. Find the shortest path from vertex **a** to vertex **b**.

Input data

The first line contains two integers **n** and **m** (**1 ≤ n ≤ 5 * 10^4**, **1 ≤ m ≤ 10^5**) - the number of vertices and edges. The second line contains two integers **a** and **b** - the starting and ending point correspondingly. Next **m** lines describe the edges.

Output data

If the path between **a** and **b** does not exist, print **-1**. Otherwise print in the first line the length **l** of the shortest path between these two vertices in number of edges, and in the second line print **l + 1** numbers - the vertices of this path.



Approach

1. In order to track the shortest path we can use **path tracing**. We can either maintain a **list** or a **dictionary**. Let's assume we're using a dictionary.
2. Once we start our traversal, while we are going through the **neighbors** of that certain node, we can make the **neighbors** a **key**, and their **parent** i.e. the current node the **value**.
3. Now if our target is **not** in our dictionary then we can return **-1**
4. Otherwise we can return the shortest path by **going all the way back** to the initial node.

```
def shortestPath():
    graph = defaultdict(list)
    nodes, edges = map(int, input().split())
    source, target = map(int, input().split())
    prev = {source: -1}
    queue = deque([source])
    answer = []

    for _ in range(edges):
        node1, node2 = map(int, input().split())
        graph[node1].append(node2)
        graph[node2].append(node1)

    while  queue:
        node = queue.popleft()
        if node == target:
            break

        for neighbour in graph[node]:
            if neighbour not in prev:
                prev[neighbour] = node
                queue.append(neighbour)
```

```
# this section is for printing the path, and comes after the
code on the left
if target not in prev:
    print(-1)
    return

current = target
answer = []

while  current != -1:
    answer.append(current)
    current = prev[current]

print(len(answer) - 1)
print(*answer[::-1])
```

Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once**, and if it's **a complete graph** we will also **use every edge**.

  Time Complexity = **O (V + E)**

Space Complexity

- We have **V nodes** and **E edges**
- We will store **the nodes**, and also **the edges**.

  Space Complexity = **O (V + E)**

Common Pitfalls

Not maintaining visited nodes

It is important to keep **track** of **visited nodes** to avoid **revisiting them** and getting stuck in **an infinite loop**.

Using the wrong data structure

BFS requires a data structure that allows for **FIFO** (first in, first out) access, such as a queue. If you use a data structure that **does not allow** for FIFO access, such as a **stack**, you may **not get the correct** traversal order.

Not considering edge cases

BFS can **fail** if the input graph is **not connected or has cycles**. It's important to consider these edge cases and handle them appropriately.

Practice Problems

- Shortest Path in Binary Matrix
- Keys and Rooms
- Open the lock
- 01 Matrix
- Map of highest peak
- As Far from Land as Possible
- All Nodes Distance K in Binary Tree
- Shortest Path with Alternating Colors
- Nearest Exit from Entrance in Maze
- Snakes and Ladders
- Rotting Oranges
- Race Car
- Bus Routes
- Word Ladder

Quote of the day

“Success is not final, failure is not fatal: It is the courage to continue that counts.”

Winston Churchill.