**Documentation: Breadth-First Search Without Queue (Recursive BFS)**

**Overview**

This Python script implements **Breadth-First Search (BFS)** using **recursion** instead of a traditional queue. It traverses a graph level by level, starting from a given node.

**What is BFS?**

**Breadth-First Search (BFS)** is a graph traversal algorithm that explores all nodes at the present depth level before moving on to the nodes at the next depth level.

## ****How This Code Works****

### 🔧 Function Definition:

def bfs\_without\_queue(graph, current\_nodes):

* graph: A dictionary representing the graph (adjacency list).
* current\_nodes: List of nodes at the current level to be processed.

### 🔁 Step-by-Step Logic:

1. **Base Case:**
2. if not current\_nodes:
3. return
   * If there are no nodes to process, end the recursion.
4. **Process Current Level:**
5. next\_nodes = []
6. for node in current\_nodes:
7. print(node)
   * Loop through all nodes at the current level.
   * Print (visit) each node.
8. **Collect Next Level Nodes:**
9. for neighbor in graph[node]:
10. next\_nodes.append(neighbor)
    * For each node, collect its neighbors to process in the next level.
11. **Recursive Call for Next Level:**
12. bfs\_without\_queue(graph, next\_nodes)
    * Recursively call the function for the next level of nodes.

## ****Graph Used in Example****

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

* Starting node: 'A'
* Structure:
  + A → B, C
  + B → D, E
  + C → F
  + E → F

## ****Function Call****

bfs\_without\_queue(graph, ['A'])

## ****Output****

A

B

C

D

E

F

F

## ****Conclusion****

This is a simple and elegant recursive approach to BFS that avoids using a queue. It's great for understanding how BFS works level-by-level, but in real-world applications, a queue + visited is recommended for accuracy and efficiency

**Documentation: Breadth-First Search (BFS) Using Queue in Python**

## ****Function Explanation****

def bfs(graph, start\_node):

* graph: A dictionary representing the graph using an **adjacency list** format.
* start\_node: The node from which BFS traversal begins.

## ****Internal Variables****

* visited\_nodes: A set to keep track of all visited nodes. Prevents visiting the same node more than once.
* queue: A list used as a **queue** (FIFO - First In, First Out) to hold nodes to be explored.

## ****Step-by-Step Logic****

1. **Initialize**:
   * Add the start\_node to the queue.
   * Create an empty visited\_nodes set.
2. **While queue is not empty**:
   * Remove the first node from the queue using pop(0).
   * If it's not already visited:
     + Print the node (this is the "visit" step).
     + Mark it as visited by adding it to the visited\_nodes set.
     + Add all **unvisited neighbors** of the node to the queue.

## ****Graph Example****

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

### 🔹 Structure:

A

/ \

B C

/ \ \

D E F

\

F

## ****Function Call****

bfs(graph, 'A')

## ****Output****

A

B

C

D

E

F

## ****Conclusion****

This BFS implementation using a queue is:

* Efficient
* Easy to understand
* Works well for most real-world graph problems

It avoids cycles and re-visits using the visited\_nodes set, and is structured level by level using a queue.