BFS

```
from collections import defaultdict


# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print(s, end=" ")

            # Get all adjacent vertices of the
            # dequeued vertex s.
            # If an adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


# Driver code
```

```
if __name__ == '__main__':

    # Create a graph given in
    # the above diagram
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Breadth First Traversal"
        " (starting from vertex 2)")
    g.BFS(2)
```

#DFS

```
# A class to represent a graph object
class Graph:
    # Constructor
    def __init__(self, edges, n):
        # A list of lists to represent an adjacency list
        self.adjList = [[] for _ in range(n)]

        # add edges to the undirected graph
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)


# Function to perform DFS traversal on the graph on a graph
def DFS(graph, v, discovered):

    discovered[v] = True            # mark the current node as discovered
    print(v, end=' ')               # print the current node

    # do for every edge (v, u)
    for u in graph.adjList[v]:
        if not discovered[u]:       # if `u` is not yet discovered
            DFS(graph, u, discovered)


if __name__ == '__main__':
```

```
# List of graph edges as per the above diagram
edges = [
    # Notice that node 0 is unconnected
    (1, 2), (1, 7), (1, 8), (2, 3), (2, 6), (3, 4),
    (3, 5), (8, 9), (8, 12), (9, 10), (9, 11)
]

# total number of nodes in the graph (labelled from 0 to 12)
n = 13

# build a graph from the given edges
graph = Graph(edges, n)

# to keep track of whether a vertex is discovered or not
discovered = [False] * n

# Perform DFS traversal from all undiscovered nodes to
# cover all connected components of a graph
for i in range(n):
    if not discovered[i]:
        DFS(graph, i, discovered)
```