UNIT - 2

Graphs & Divide and Conquer

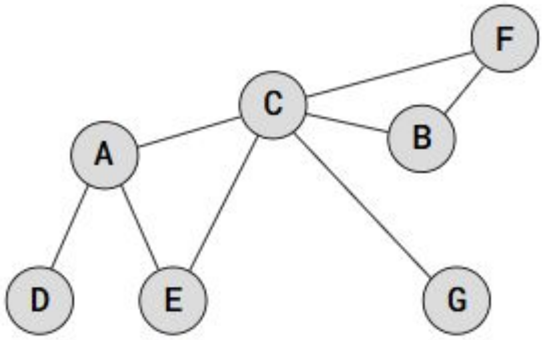
UNIT - 2

Graphs & Divide and Conquer: Graph Connectivity and Graph Traversal, Breadth-First Search: Exploring a Connected Component, Depth-First Search, Implementing Graph Traversal Using Queues and Stacks: Implementing Breadth-First Search, Implementing Depth-First Search, An Application of Breadth-First Search and Depth-First Search, Directed Acyclic Graphs and Topological Ordering. **Divide and Conquer Technique**: Masters Theorem for recurrence relations, The Merge sort Algorithm, Quick Sort Algorithm (Textbook 2).

Graph Connectivity and Graph Traversal

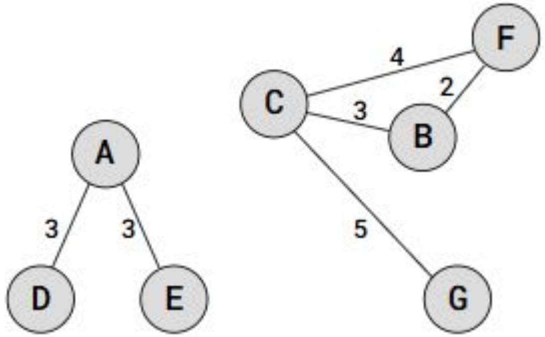
Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines that connect any two nodes in the graph.

More formally a Graph is composed of a set of vertices (**V**) and a set of edges (**E**). The graph is denoted by **G= (V, E).**

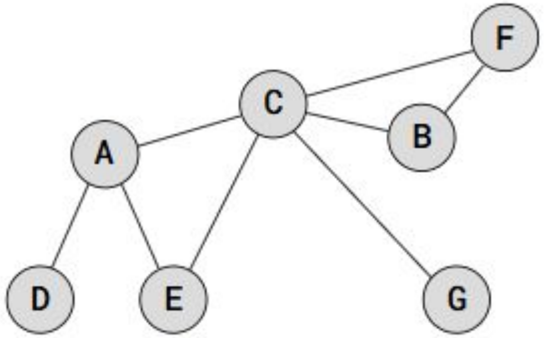


Different Types of Graph

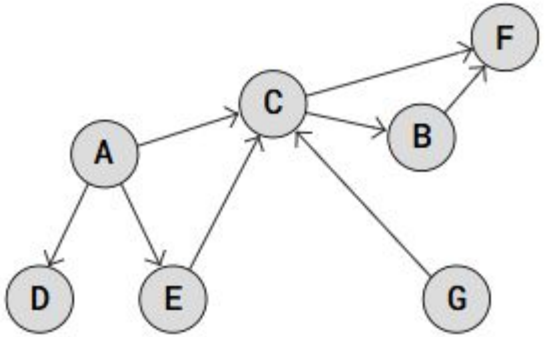
Weighted Graph: A weighted Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability.



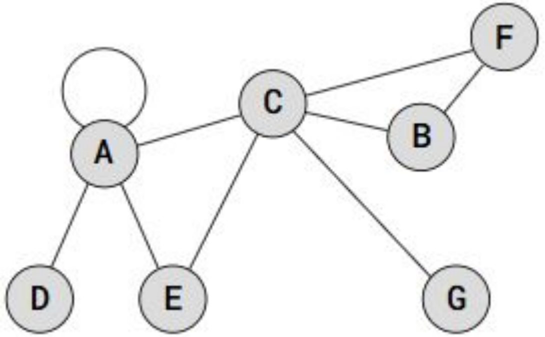
Connected Graph: A Connected Graph is when all the vertices are connected through edges somehow.



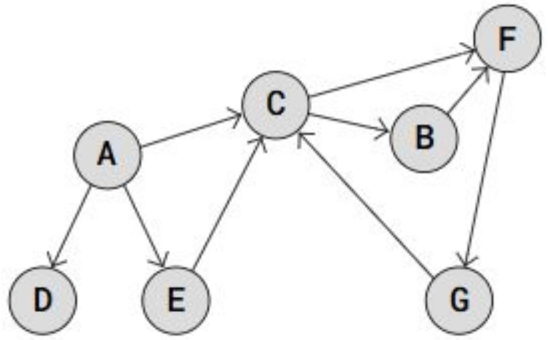
Directed Graph: A directed Graph, also known as a **digraph**, is when the edges between the vertex pairs have a **direction**. The direction of an edge can represent things like hierarchy or flow.



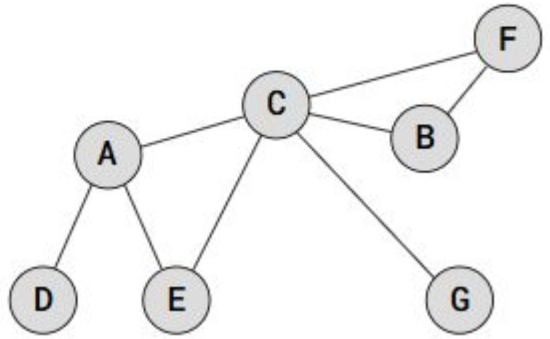
Loop: A loop, also called a **self-loop**, is an edge that begins and ends on the same vertex. A loop is a cycle that only consists of one edge. By adding the loop on vertex A, the Graph becomes cyclic.



Directed cyclic: A directed cyclic Graph is when you can follow a path along the directed edges that goes in circles.



Undirected cyclic: An undirected cyclic Graph is when you can come back to the same vertex you started at without using the same edge more than once.



Graph Representations

- Graphs for computer algorithms are usually represented in one of two ways: the **adjacency matrix** and **adjacency lists.**
- The a**djacency matrix** of a graph with **n** vertices is an **n × n boolean matrix** with one row and one column for each of the graph’s vertices, in which the element in the **ith row** and the **j th column** is equal to **1** if there is an edge from the ith vertex to the j th vertex, and equal to 0 if there is no such edge.

Adjacency Matrix Representation

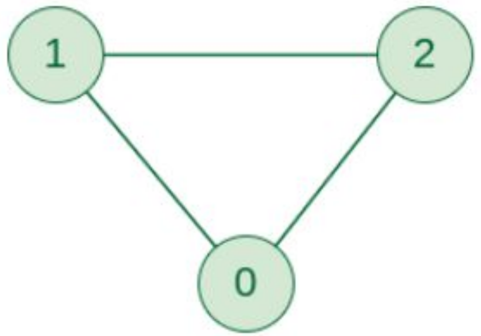
An adjacency matrix is a way of representing a graph as a matrix of boolean (0’s and 1’s)

Let’s assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension **n x n.**

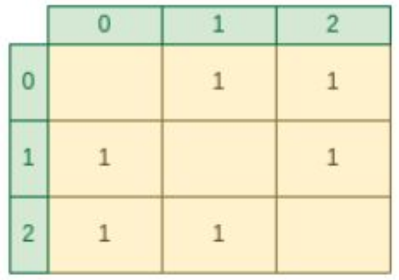
- If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.
- If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.

a) Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases (**adjMat[destination]** and **adjMat[destination])** because we can go either way.



Undirected Graph



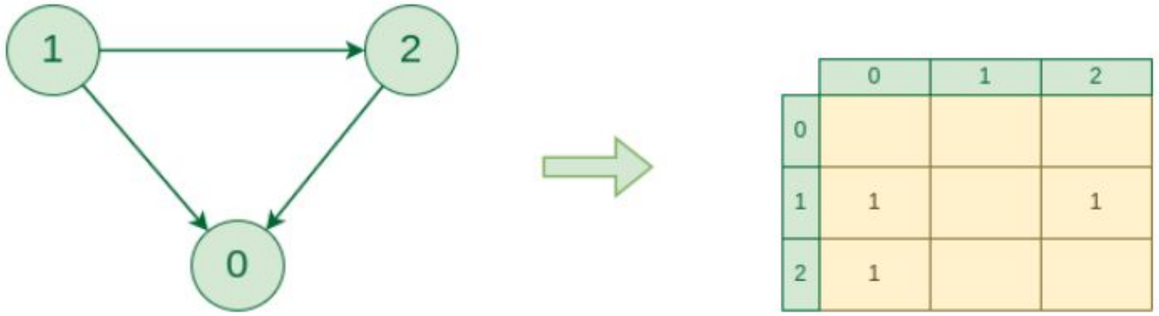
	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

b) Representation of Directed Graph as Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



	0	1	2
0			
1	1		1
2	1		

Directed Graph

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Adjacency List Representation

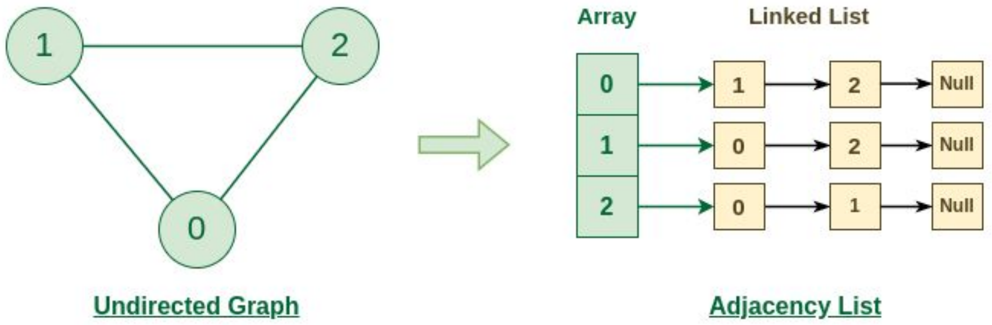
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Assume there are **n** vertices in graph, create an **array of list** of size **n** as **adjList[n].**

- adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.

a) Representation of Undirected Graph as Adjacency list:

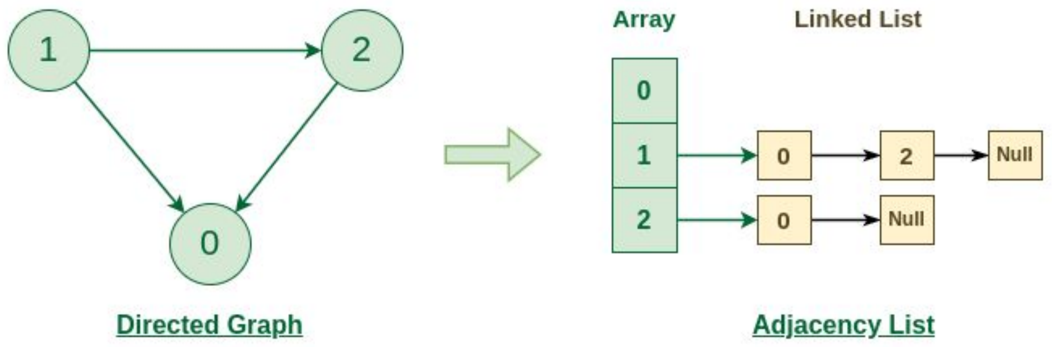
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list



Graph Representation of Undirected graph to Adjacency List

b) Representation of Directed Graph as Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

- The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list’s vertex
- To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1’s.

$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 0 & 0 & 1 & 1 & 0 & 1 \\ f & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

(a)

$$\begin{array}{l} \boxed{a} \to c \to d \\ \boxed{b} \to c \to f \\ \boxed{c} \to a \to b \to e \\ \boxed{d} \to a \to e \\ \boxed{e} \to c \to d \to f \\ \boxed{f} \to b \to e \end{array}$$

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.

Real-Time Applications of Graph:

- **Network monitoring:** Graphs can be used to monitor network traffic in real-time, allowing network administrators to identify potential bottlenecks, security threats, and other issues.
- **Social media analysis:** Social media platforms generate vast amounts of data in real-time, which can be analyzed using graphs to identify trends, sentiment, and key influencers.
- **Financial trading:** Graphs can be used to analyze real-time financial data, such as stock prices and market trends, to identify patterns and make trading decisions.

- **Internet of Things (IoT) management:** IoT devices generate vast amounts of data in real-time, which can be analyzed using graphs to identify patterns, optimize performance, and detect anomalies.
- **Autonomous vehicles:** Graphs can be used to model the real-time environment around autonomous vehicles, allowing them to navigate safely and efficiently.
- **Disease surveillance**: Graphs can be used to model the spread of infectious diseases in real-time, allowing health officials to identify outbreaks and implement effective containment strategies. This is particularly important during pandemics or other public health emergencies.

Implementing Graph Traversal Using Queues and Stacks:

The two most common ways a Graph can be traversed are:

- Breadth First Search (BFS)
- Depth First Search (DFS)

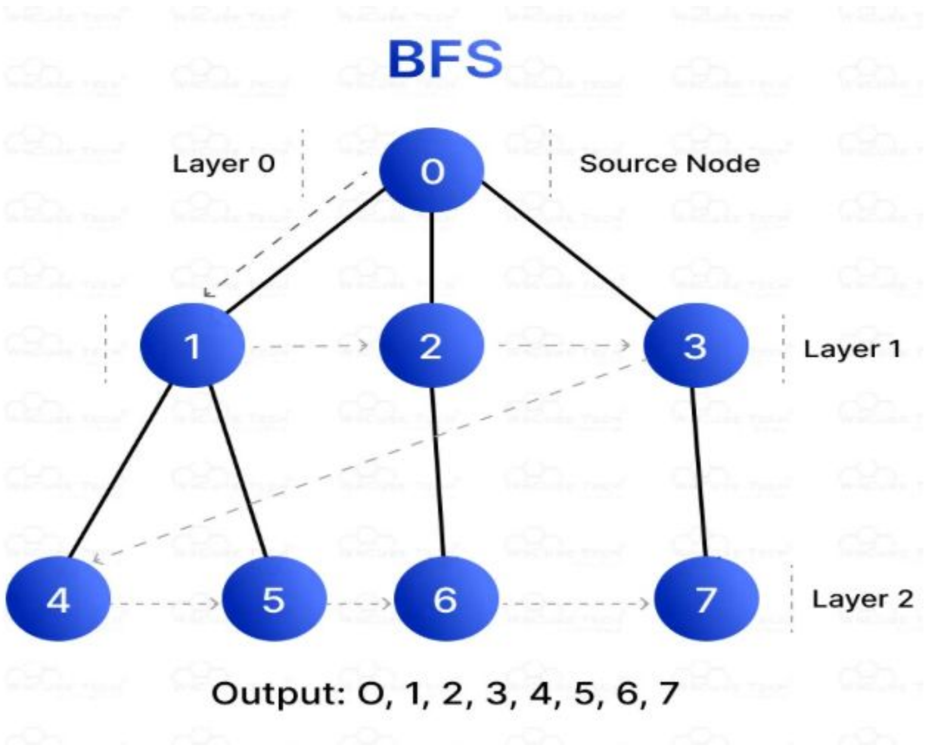
DFS is usually implemented using a Stack or by the use of recursion (which utilizes the call stack), while BFS is usually implemented using a Queue.

Breadth First Search (BFS):

- **Breadth First Search (BFS)** is a fundamental **graph traversal algorithm.**
- BFS explores all nodes at the present depth before moving deeper. It uses a **queue (FIFO)** structure.

BFS Algorithm:

- Start from a source node.
- Visit all its neighbors before moving to their neighbors.
- Use a queue to keep track of visited nodes.



Popular graph algorithms like Dijkstra’s shortest path and Prim’s algorithm are based on BFS.

Traversal Order:

- Starting from node 0:
- Layer 0: Visit node 0 (source node).
- Layer 1: Visit nodes 1, 2, 3 (neighbors of node 0).
- Layer 2: Visit nodes 4, 5, 6, 7 (neighbors of nodes 1, 2, 3).
- Output: 0, 1, 2, 3, 4, 5, 6, 7.

BFS is a level-order traversal where we visit all **neighboring** nodes before going deeper. It uses a **queue (FIFO: First In, First Out)**.

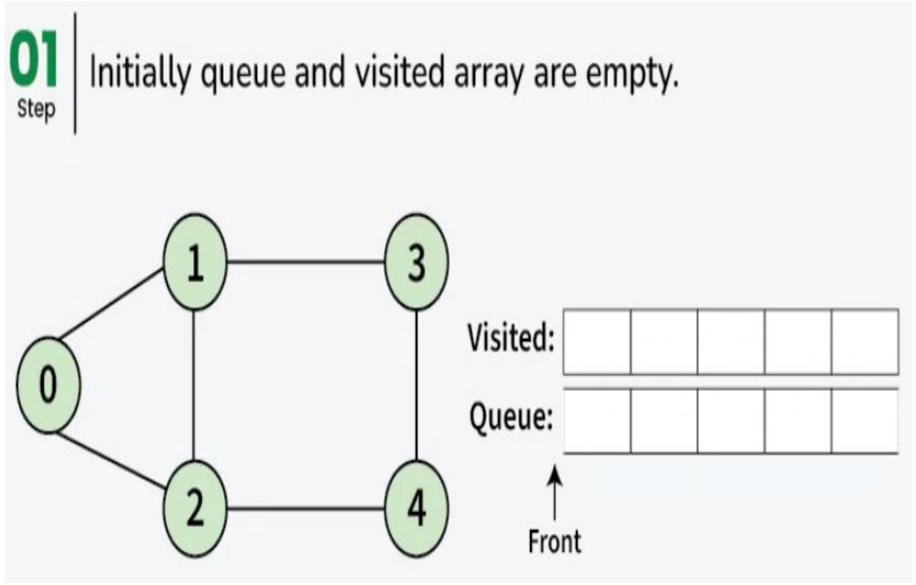
```
      A
     / \
    B   C
   / \   \
  D   E   F
```

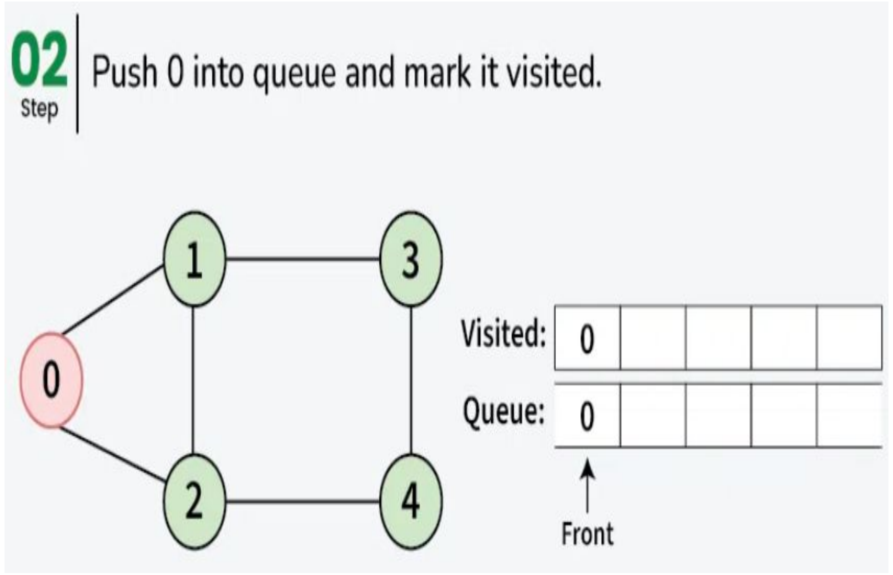
- **BFS Traversal (Starting from A) : Final BFS Order:A→B→C→D→E→F**
- Start at **A** → Visit **A**, add its neighbors (**B, C**) to the queue.
- Dequeue **B** → Visit **B**, add its neighbors (**D, E**) to the queue.
- Dequeue **C** → Visit **C**, add its neighbor (**F**) to the queue.
- Dequeue **D** → Visit **D** (no more neighbors).
- Dequeue **E** → Visit **E** (no more neighbors).
- Dequeue **F** → Visit **F** (no more neighbors).

BFS from a Given Approach

1. **Initialization:** Enqueue the given source vertex into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - Enqueue the neighbor into the queue and Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

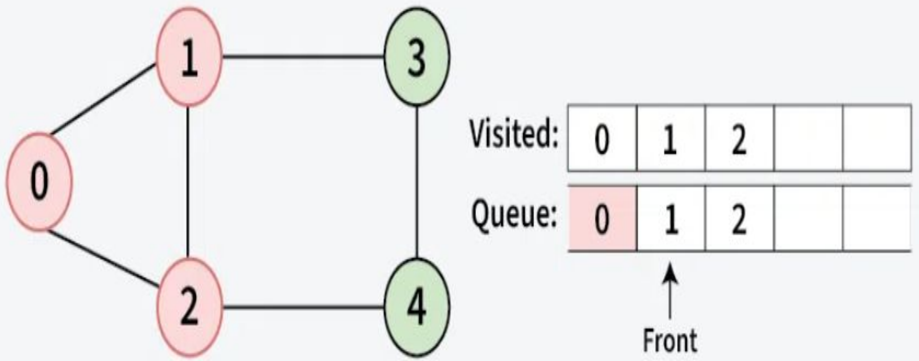
Example:


01
Step
Initially queue and visited array are empty.
1
3
0
2
4
Visited:
Queue:
Front


02
Step
Push 0 into queue and mark it visited.
1
3
0
2
4
Visited:
0
Queue:
0
Front

03 Step

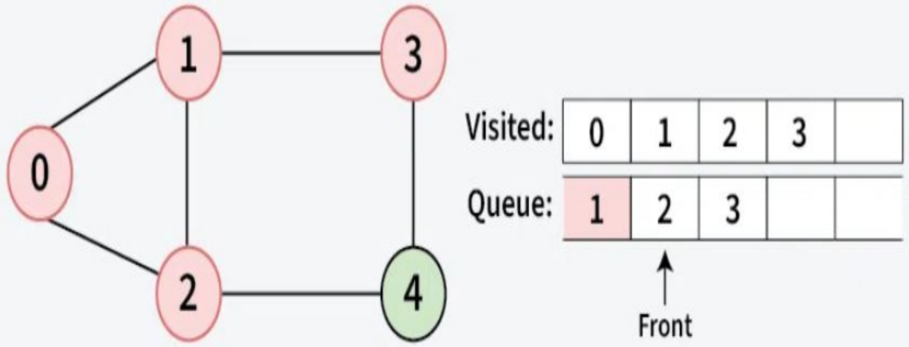
Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2		
Queue:	0	1	2		

04 Step

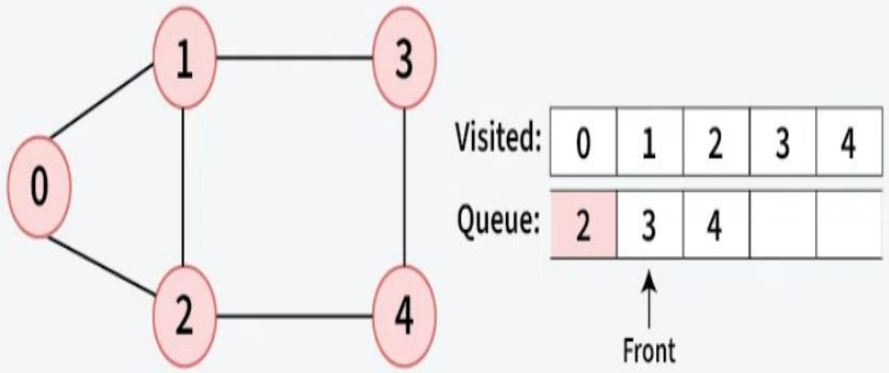
Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	
Queue:	1	2	3		

05 Step

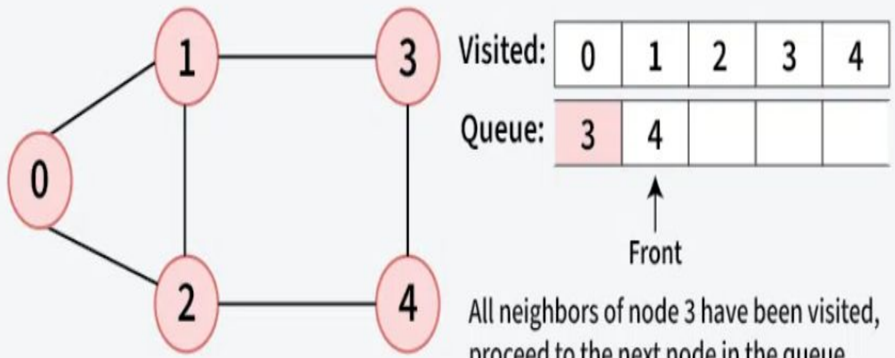
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	2	3	4		

06 Step

Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:	0	1	2	3	4
Queue:	3	4			

All neighbors of node 3 have been visited, proceed to the next node in the queue.

- It proceeds in a concentric manner by **visiting first all the vertices** that are adjacent to a **starting vertex**, then all **unvisited vertices** two edges apart from it, and so on, **until all the vertices** in the same connected component as the starting vertex **are visited.**
- If there still **remain unvisited vertices,** the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
- It is convenient to use a **queue to trace the operation** of breadth-first search.

- The **queue** is initialized with the traversal’s starting vertex, which is marked as **visited.**
- On each **iteration**, the algorithm identifies all **unvisited vertices** that are adjacent to the **front vertex**, mark them as **visited**, and adds them to the **queue**; after that, the front vertex is **removed from the queue.**
- The traversal’s starting vertex serves as the **root of the first tree**. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a **child to the vertex** it is being reached from with an edge called a **tree edge**.

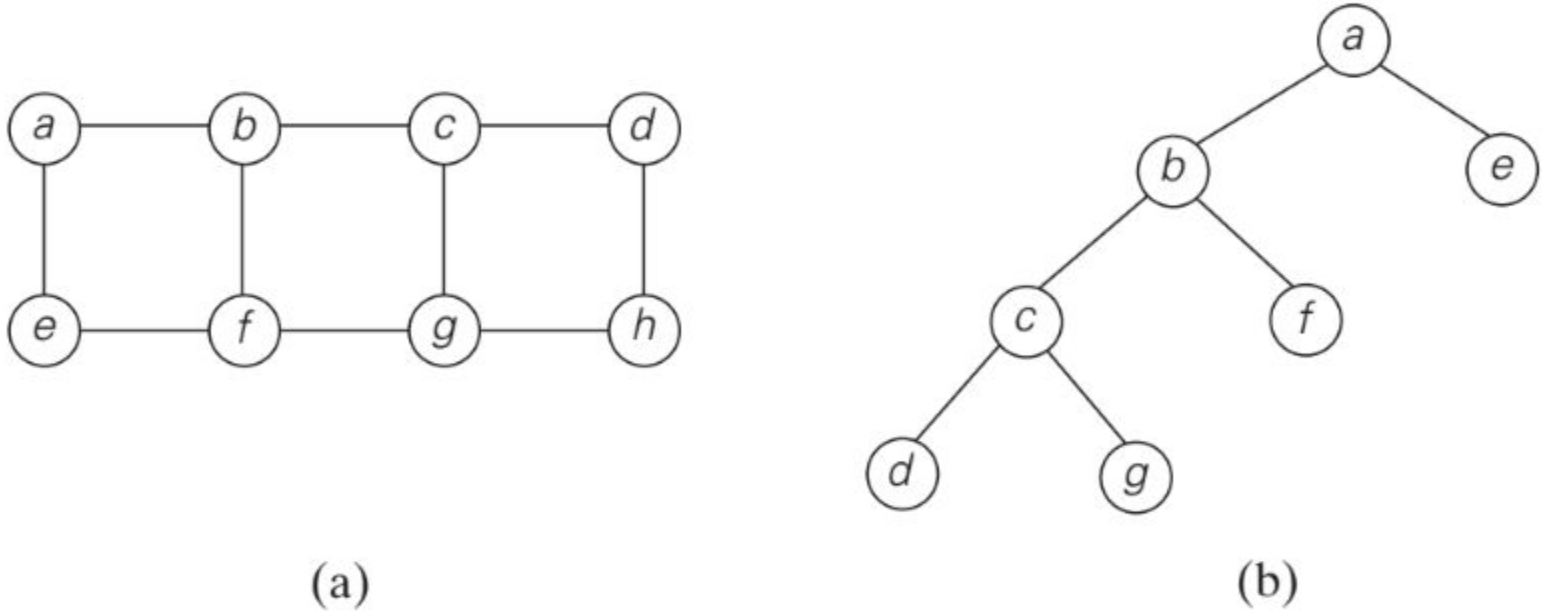


FIGURE 3.12 Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

Key Differences Between Graph and Tree

- **Cycles:** Graphs can contain **cycles,** while trees cannot.
- **Connectivity:** Graphs can be disconnected (i.e., have multiple components), while trees are always **connected.**
- **Hierarchy:** Trees have a **hierarchical structure**, with one vertex designated as the root. Graphs do not have this hierarchical structure.
- **Applications:** Graphs are used in a wide variety of applications, such as social networks, transportation networks, and computer science.

Understanding the Algorithm

1. **Initialization:**

- The algorithm takes a graph **G = (V, E)** as input.
- It marks all vertices as **unvisited** (represented by marking them with 0).
- A **global counter** (count) is initialized to 0.

2. **Processing Each Vertex:**

- It iterates through all vertices in **V**.
- If a vertex **v** is unvisited, it calls bfs(v) to explore all reachable vertices.

```
ALGORITHM  BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //            in the order they are visited by the BFS traversal
    mark each vertex in $V$ with 0 as a mark of being “unvisited”
    $count \leftarrow 0$
    for each vertex $v$ in $V$ do
        if $v$ is marked with 0
            $bfs(v)$
```

3. BFS Function (bfs(v)):

- The function starts at vertex **v**, increments count, and marks **v** with this value.
- It places **v** in a queue to process it in a **FIFO (First-In-First-Out)** manner.
- While the queue is not empty:
  a) The algorithm examines each neighbor **w** of the front vertex.
  b) If **w** is unvisited, it increments count, marks **w**, and adds it to the queue.
  c) The front vertex is removed from the queue.

```
$bfs(v)$
//visits all the unvisited vertices connected to vertex $v$
//by a path and numbers them in the order they are visited
//via global variable $count$
$count \leftarrow count + 1$;   mark $v$ with $count$ and initialize a queue with $v$
**while** the queue is not empty **do**
    **for** each vertex $w$ in $V$ adjacent to the front vertex **do**
        **if** $w$ is marked with 0
            $count \leftarrow count + 1$;   mark $w$ with $count$
            add $w$ to the queue
    remove the front vertex from the queue
```

Complexity Analysis of Breadth-First Search (BFS) Algorithm

Time Complexity: O(V + E), BFS explores all the vertices and edges in the graph. In the worst case, it visits every vertex and edge once. Therefore, the time complexity of BFS is O(V + E), where V and E are the number of vertices and edges in the given graph

Space Complexity: O(V), BFS uses a queue to keep track of vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is O(V).

Applications of BFS in Graphs

- **Shortest Path Finding:** BFS can be used to find the shortest path between two nodes in an unweighted graph.
- **Cycle Detection:** BFS can be used to detect cycles in a graph.
- **Topological Sorting:** Topological sorting arranges the nodes in a linear order such that for any edge (u, v), u appears before v in the order.

- **Connected Components:** BFS can be used to identify connected components
- **Network Routing:** BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols.
- **Level Order Traversal of Binary Trees:** BFS can be used to perform a level order traversal of a binary tree.

Depth First Search (DFS)

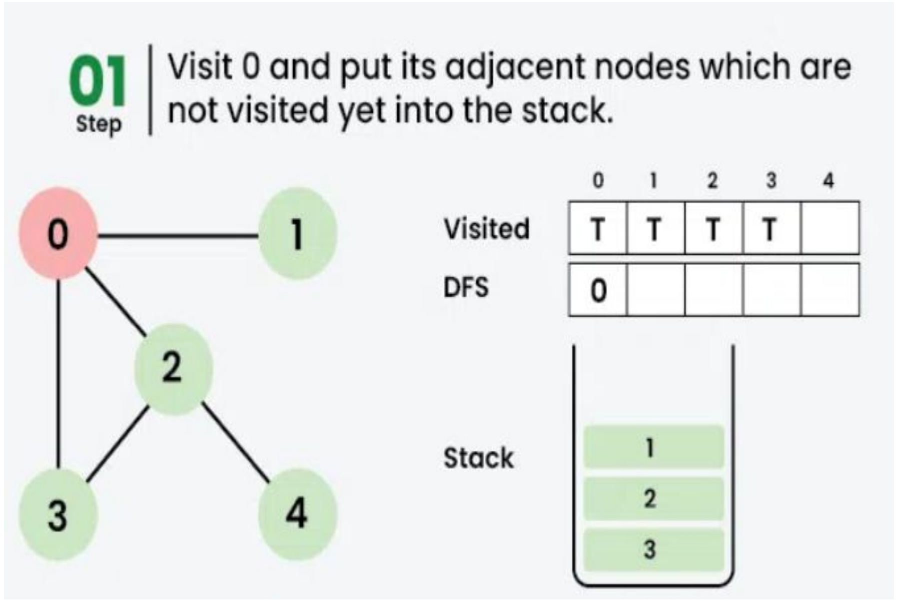
- In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices **one by one**.
- **DFS** explores as far as possible along each branch before backtracking. It uses a **stack (LIFO)** or recursion.
- When we traverse an **adjacent vertex**, we completely **finish the traversal of all vertices reachable through that adjacent vertex.**

This is similar to a **tree**, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles

DFS Algorithm:

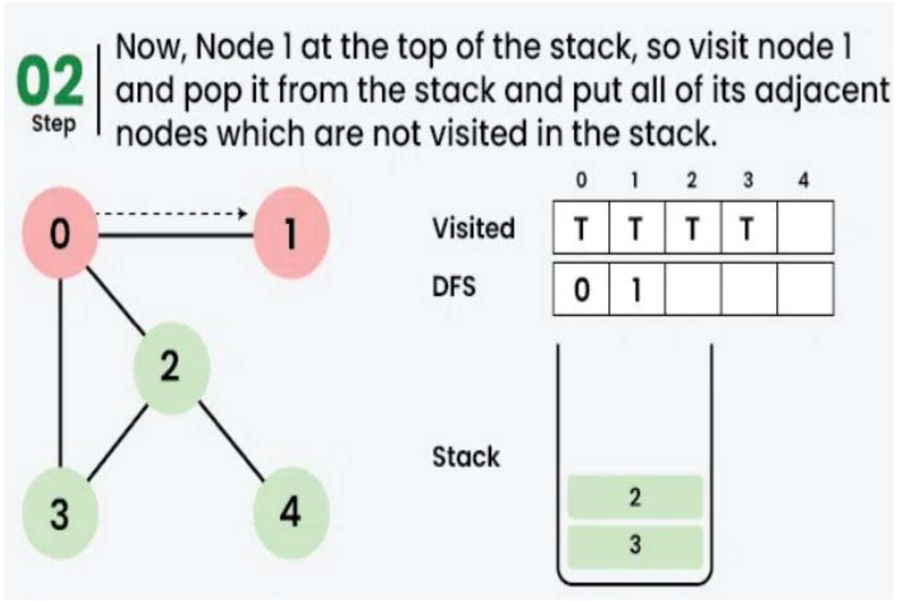
- Start from a source node.
- Recursively explore each unvisited neighbor.
- Backtrack when no unvisited neighbors are left.

Example:



01 Step | Visit 0 and put its adjacent nodes which are not visited yet into the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0				

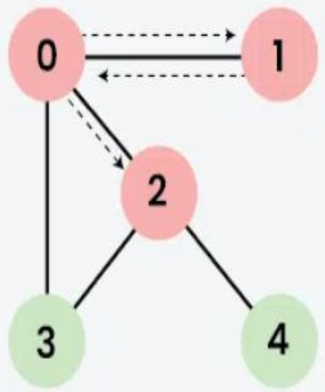


02 Step | Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

	0	1	2	3	4
Visited	T	T	T	T	
DFS	0	1			

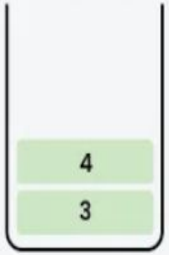
03 Step

Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



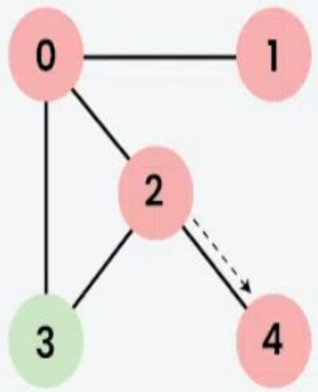
	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2		

Stack



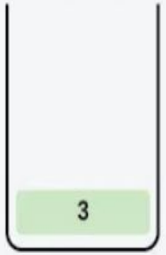
04 Step

Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



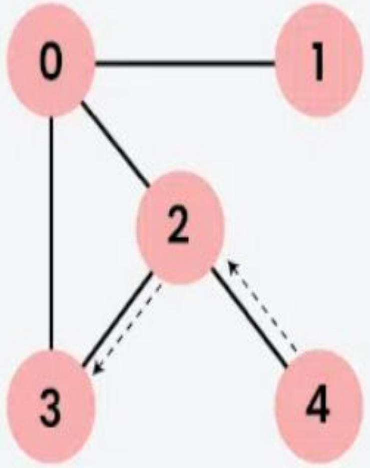
	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	

Stack



05 Step

Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	3

Stack

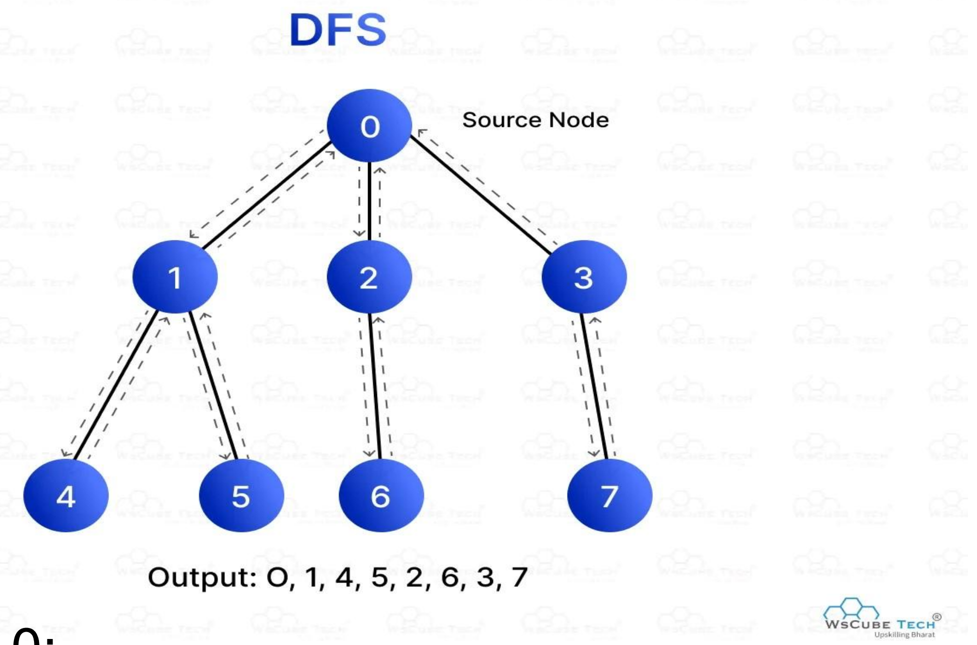
Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

DFS from a Given Source

The basic idea is to start from the **root node** and **mark the node** and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then **backtrack** and **check** for other unmarked nodes and traverse them. Finally **print the nodes** in the path.

Algorithm:

1. Insert the **root** in the stack.
2. Run a loop till the stack is not empty.
3. Pop the element from the stack and print the element.
4. For every adjacent and unvisited node of current node, mark the node and insert it in the stack.



Traversal Order:

- Starting from node 0:
- Visit node 0 → Visit node 1 → Visit node 4 (deepest) → Backtrack to node 1 → Visit node 5 → Backtrack to node 0 → Visit node 2 → Visit node 6 → Backtrack to node 2 → Visit node 3 → Visit node 7.
- **Output:** 0, 1, 4, 5, 2, 6, 3, 7

DFS explores as **deep as possible** along a branch before backtracking. It uses a **stack (LIFO)**

```
      A
     / \
    B   C
   / \   \
  D   E   F
```

- **DFS Traversal (Starting from A)**
- Start at **A** → Visit **A**, push neighbors (**B, C**) to the stack.
- Pop **C** → Visit **C**, push neighbor (**F**).
- Pop **F** → Visit **F** (no more neighbors).
- Pop **B** → Visit **B**, push neighbors (**D, E**).
- Pop **E** → Visit **E** (no more neighbors).
- Pop **D** → Visit **D** (no more neighbors).
- **Final DFS Order:** A→C→F→B→E→D

Algorithm

1) Initially mark all the vertices as not visited and create a stack for DFS.
2) Push the source node in it.
3) Loop until the stack is not empty

→ Pop the top node of the stack s and check whether s is visited or not, if it is not visited we print the node and mark as visited.

→ Traverse through the entire adjacency list of s, if we get a not visited node, we push it into the stack.

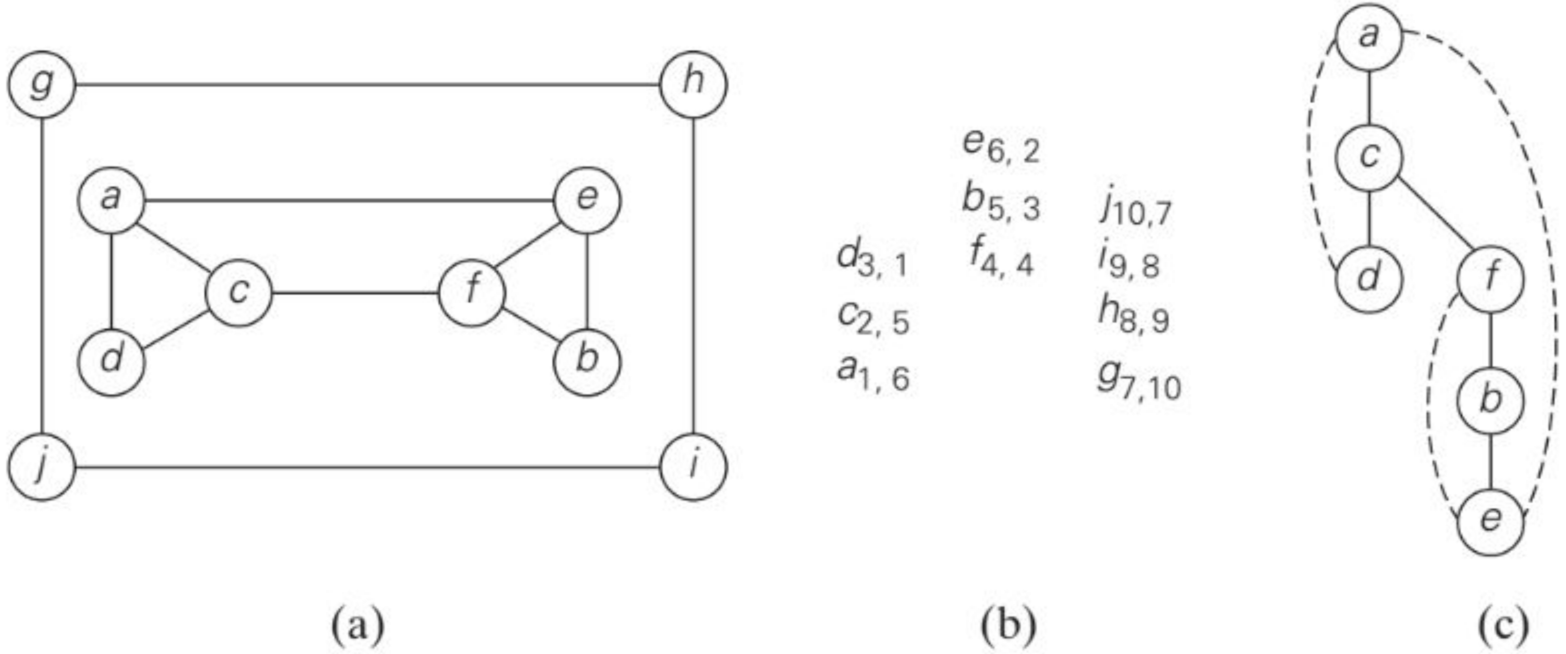


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Understanding the Algorithm

1. **Initialization:**

- Each vertex (node) in the graph is marked as **unvisited (0)**.
- A global counter count is set to 0. This helps keep track of the order in which nodes are visited.

2. **Start DFS for Each Node:**

- The algorithm loops through all nodes in the graph.
- If a node is **unvisited (marked 0)**, start a **DFS traversal** from that node.

```
ALGORITHM  $DFS(G)$
    //Implements a depth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //           in the order they are first encountered by the DFS traversal
    mark each vertex in $V$ with 0 as a mark of being "unvisited"
    $count \leftarrow 0$
    for each vertex $v$ in $V$ do
        if $v$ is marked with 0
            $dfs(v)$
```

3. DFS Function (Recursive Process):

- **Increase the count** and mark the current node with this value.
- **Explore all its unvisited neighbors** (connected nodes).
- If a neighbor is unvisited, **recursively call DFS** on it.

```
$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex $v$
//by a path and numbers them in the order they are encountered
//via global variable $count$
$count \leftarrow count + 1$;   mark $v$ with $count$
for each vertex $w$ in $V$ adjacent to $v$ do
    if $w$ is marked with 0
        $dfs(w)$
```

Time and Space Complexity of Depth First Search (DFS)

The **time complexity** of **Depth First Search (DFS**) is **O(V + E)**, where **V** is the number of vertices and **E** is the number of edges. The **space complexity** is **O(V)** for a recursive implementation.

Best Case of Depth First Search: O(V + E)

- The best-case time complexity of DFS occurs when the target node is found quickly after exploring only a few vertices and edges.
- In this scenario, the algorithm may terminate early without having to visit all vertices and edges.

Average Case of Depth First Search: O(V + E)

- The average-case time complexity of DFS is also **O(V + E).**
- DFS explores vertices and edges in a depth-first manner

Worst Case of Depth First Search: O(V + E)

- The worst-case time complexity of DFS also remains **O(V + E).**
- In the worst case, DFS explores all vertices and edges reachable from the source node.
- The algorithm systematically traverses each edge and explores all vertices in a depth-first manner.

Applications of DFS in Graphs

Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges

Solving puzzles with only one solution: such as mazes.

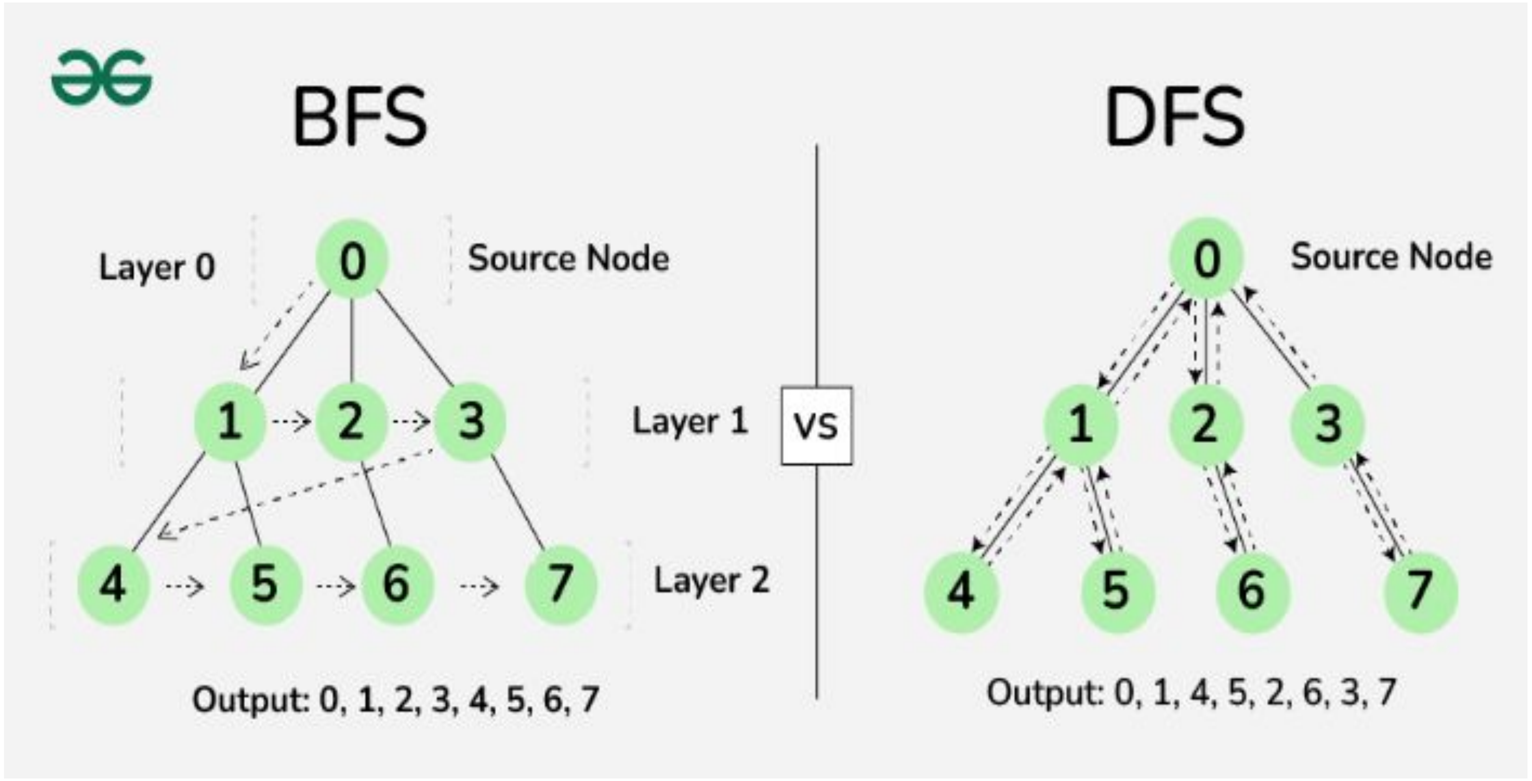
Topological Sorting

Backtracking: Depth-first search can be used in backtracking algorithms.

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	**BFS**
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\lvert V^2 \rvert)$	$\Theta(\lvert V^2 \rvert)$
Efficiency for adjacency lists	$\Theta(\lvert V \rvert + \lvert E \rvert)$	$\Theta(\lvert V \rvert + \lvert E \rvert)$

Difference between BFS and DFS


BFS
DFS
Layer 0
Source Node
Layer 1
Layer 2
VS
0
1
2
3
4
5
6
7
Output: 0, 1, 2, 3, 4, 5, 6, 7
Source Node
Output: 0, 1, 4, 5, 2, 6, 3, 7

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.

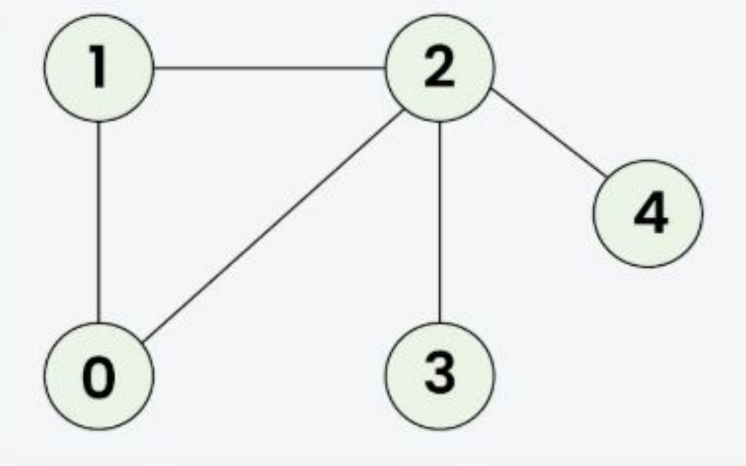
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc. If weight of every edge is same, then BFS gives shortest pat from source to every other vertex.	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc. There are many applications where both BFS and DFS can be used like Topological Sorting, Cycle Detection, etc.

Problem Solving work for BFS and DFS

Resolve problem and Write an Pseudocode

Given a undirected graph represented by an adjacency list adj, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Traversal (BFS) starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]



Output: [0, 1, 2, 3, 4]

Explanation: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 → Output: 0

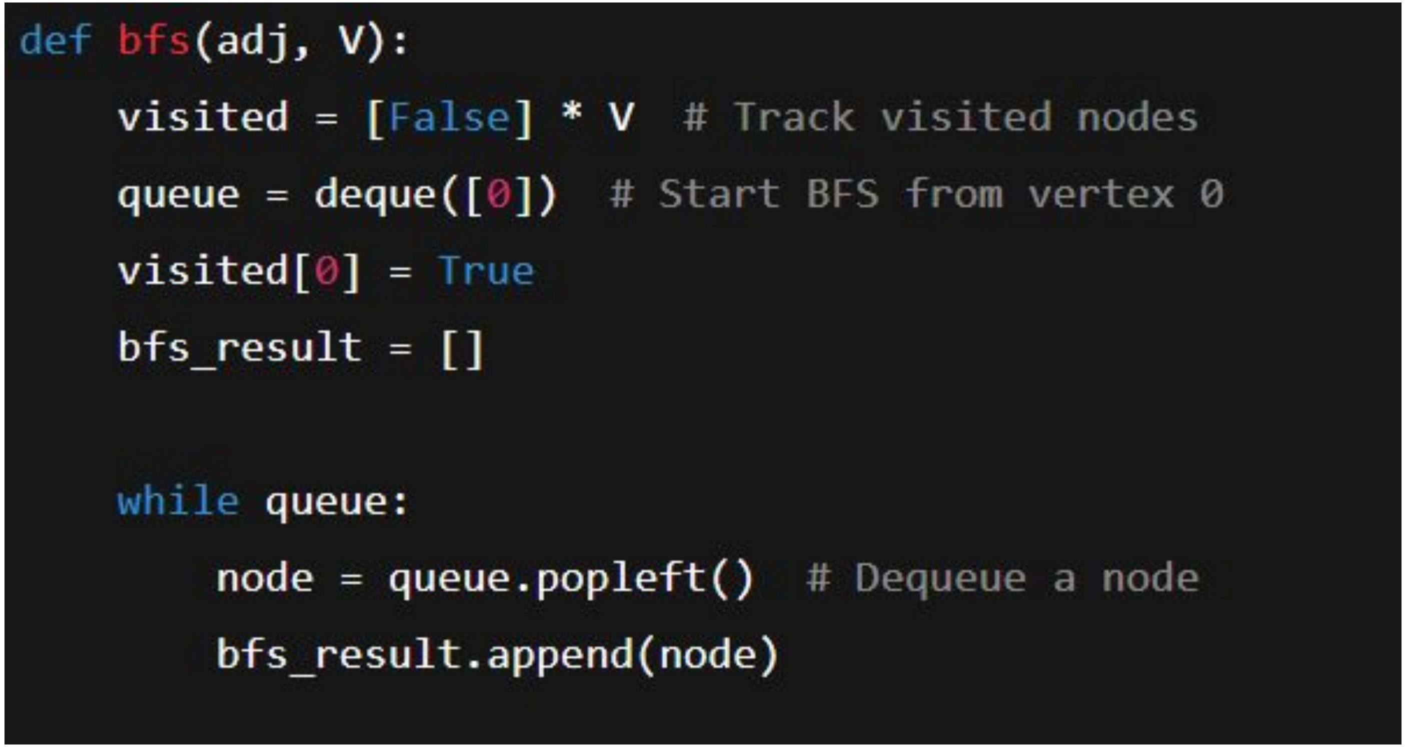
Visit 1 (the first neighbor of 0) → Output: 0, 1

Visit 2 (the next neighbor of 0) → Output: 0, 1, 2

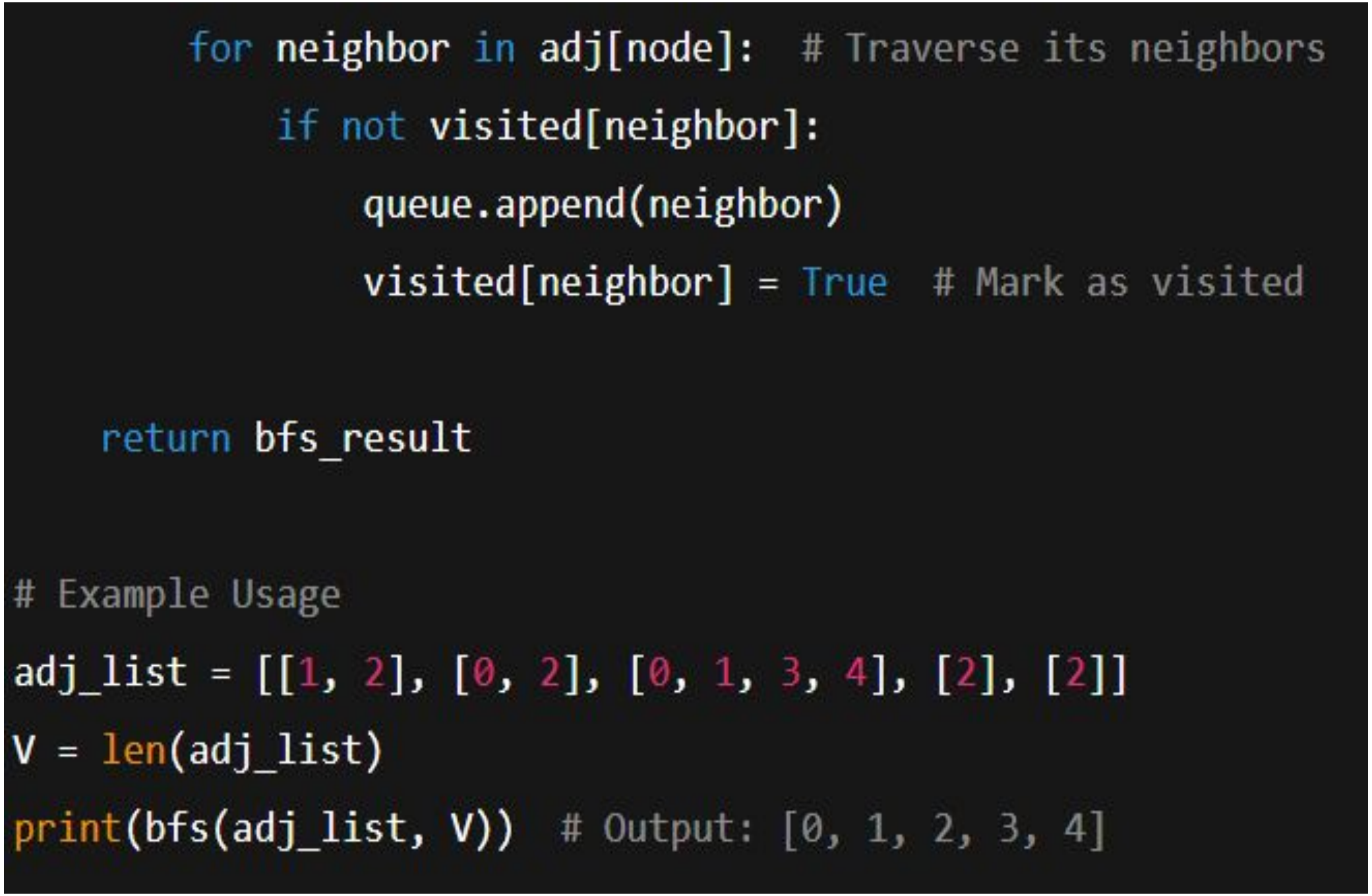
Visit 3 (the first neighbor of 2 that hasn't been visited yet) → Output: 0, 1, 2, 3

Visit 4 (the next neighbor of 2) → Final Output: 0, 1, 2, 3, 4

BFS CODE IMPLEMENTATION

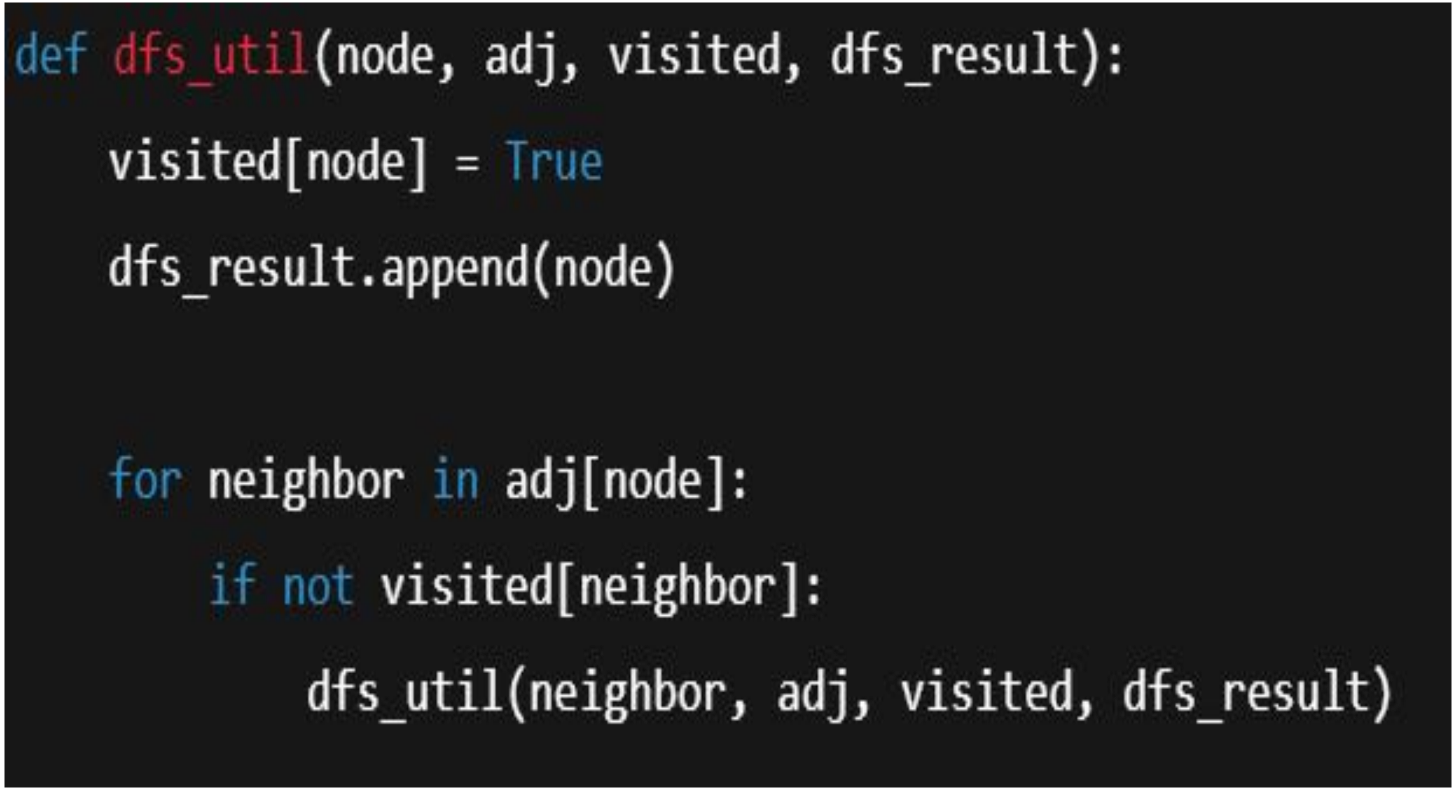


```python
def bfs(adj, V):
    visited = [False] * V  # Track visited nodes
    queue = deque([0])  # Start BFS from vertex 0
    visited[0] = True
    bfs_result = []

    while queue:
        node = queue.popleft()  # Dequeue a node
        bfs_result.append(node)
```

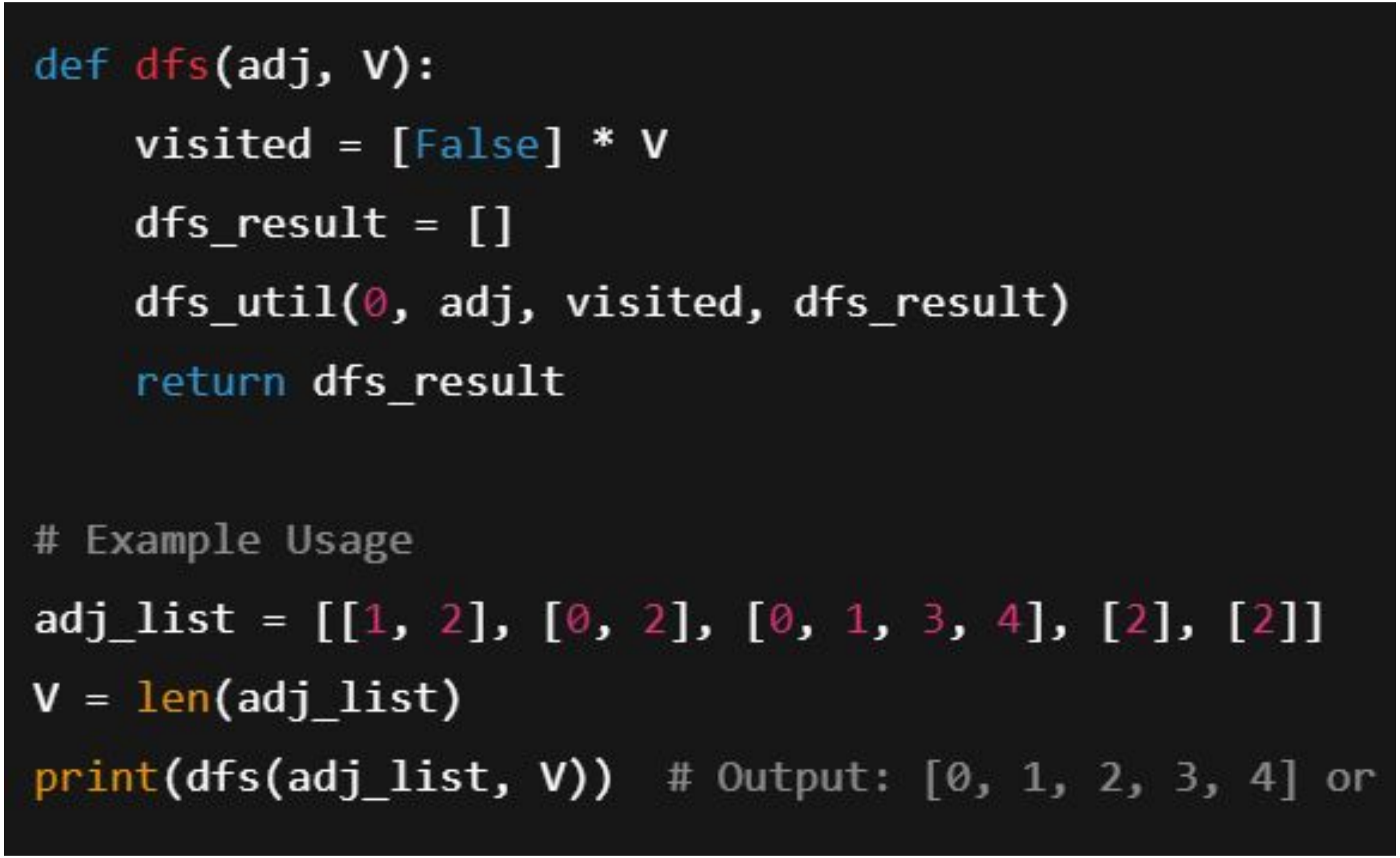


```python
        for neighbor in adj[node]:  # Traverse its neighbors
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True  # Mark as visited

    return bfs_result

# Example Usage
adj_list = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]
V = len(adj_list)
print(bfs(adj_list, V))  # Output: [0, 1, 2, 3, 4]
```

DFS CODE IMPLEMENTATION



```
def dfs_util(node, adj, visited, dfs_result):
    visited[node] = True
    dfs_result.append(node)

    for neighbor in adj[node]:
        if not visited[neighbor]:
            dfs_util(neighbor, adj, visited, dfs_result)
```



```python
def dfs(adj, V):
    visited = [False] * V
    dfs_result = []
    dfs_util(0, adj, visited, dfs_result)
    return dfs_result


# Example Usage
adj_list = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]
V = len(adj_list)
print(dfs(adj_list, V))  # Output: [0, 1, 2, 3, 4] or
```

Directed Acyclic Graphs and Topological Ordering

- If an undirected graph has no cycles, then it has an extremely simple structure: each of its connected components is a tree. But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure.
- If a directed graph has no cycles, we call it naturally enough a directed acyclic graph, or a **DAG** for short.

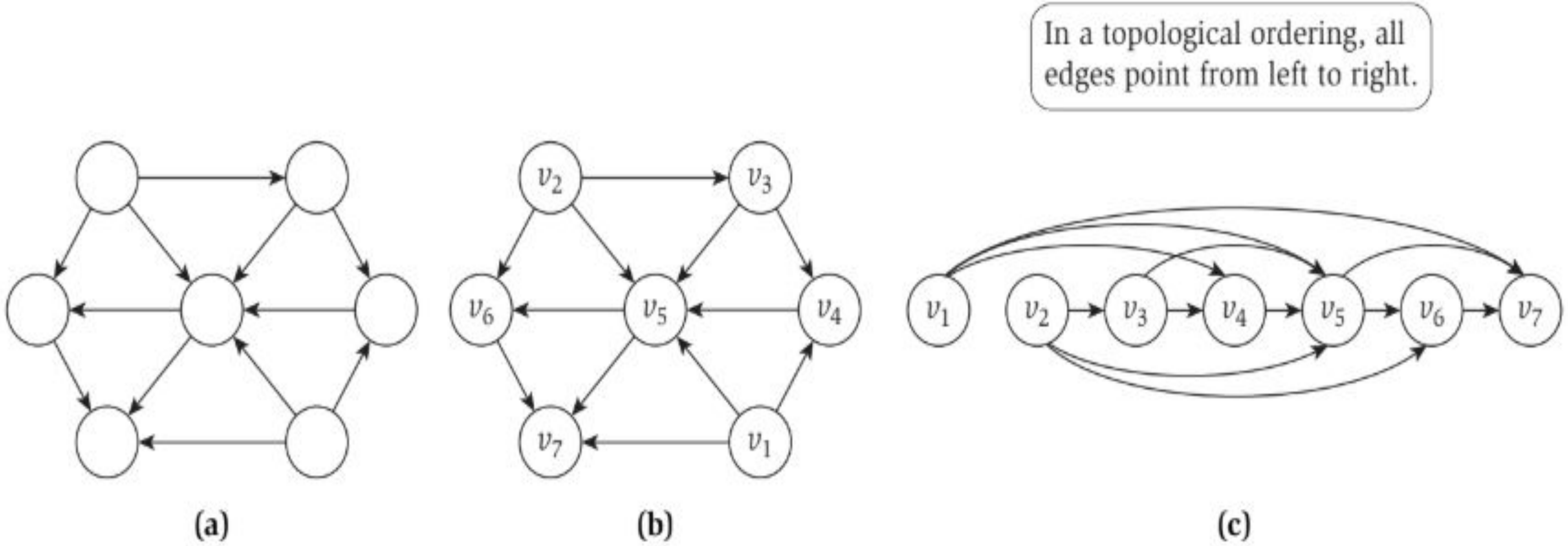


Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

Directed Acyclic Graph (DAG)

- **A Directed Acyclic Graph (DAG) is a directed graph with no cycles.**
- **Topological ordering** is a way to arrange the vertices of a DAG in a sequence or list.
- **Topological Sorting**

a) A linear ordering of vertices such that for every directed edge (u → v), u appears before v.

b) Used in task scheduling, dependency resolution, and course prerequisites.

Topological Sort

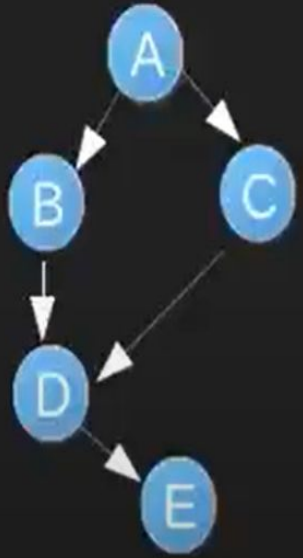
Idea : for a Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.

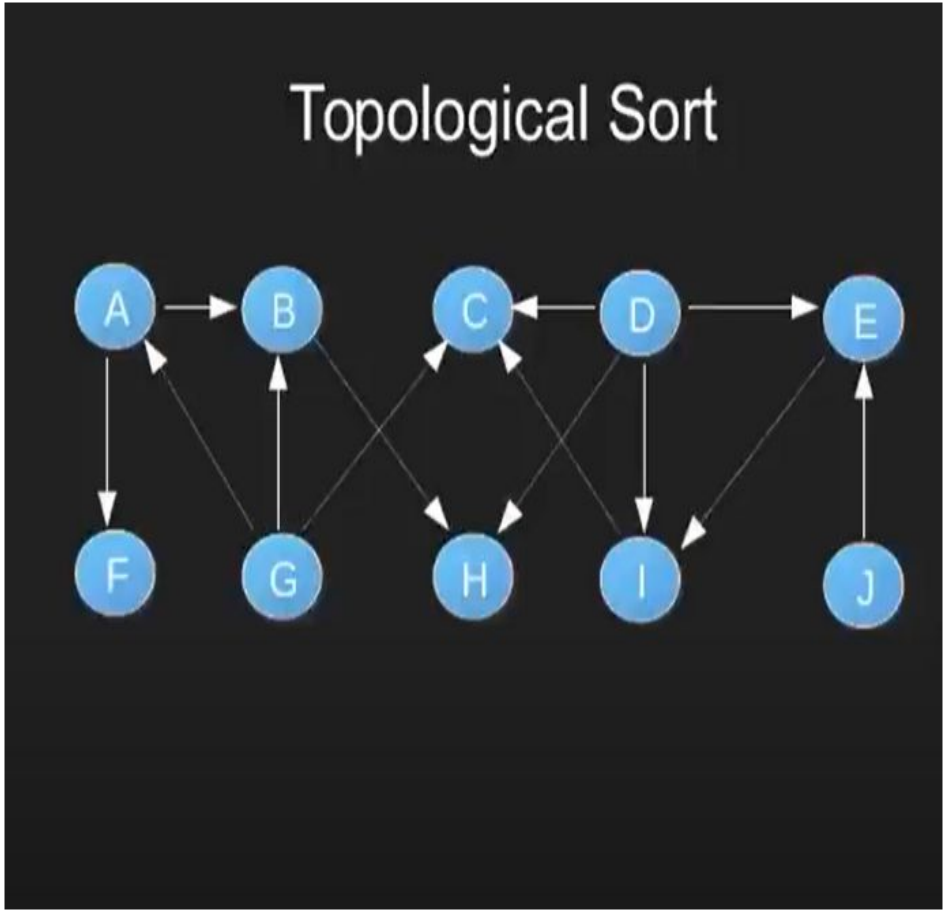
-For a DAG

-Topological sort is not unique.

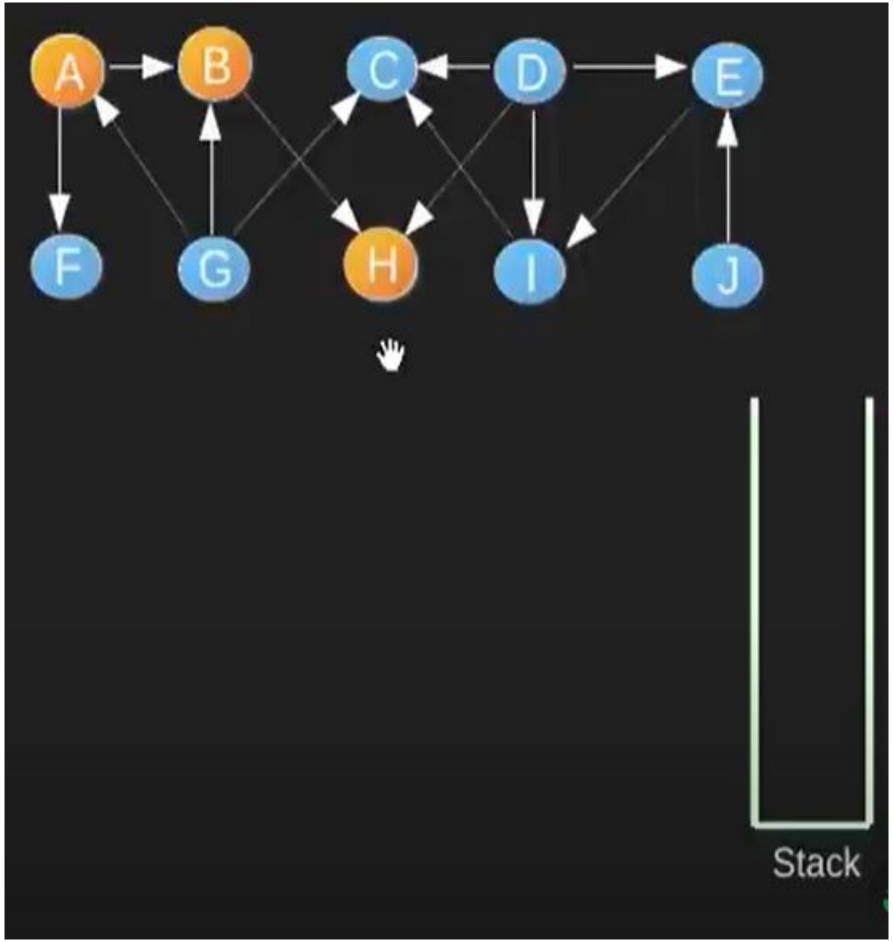
A B C D E

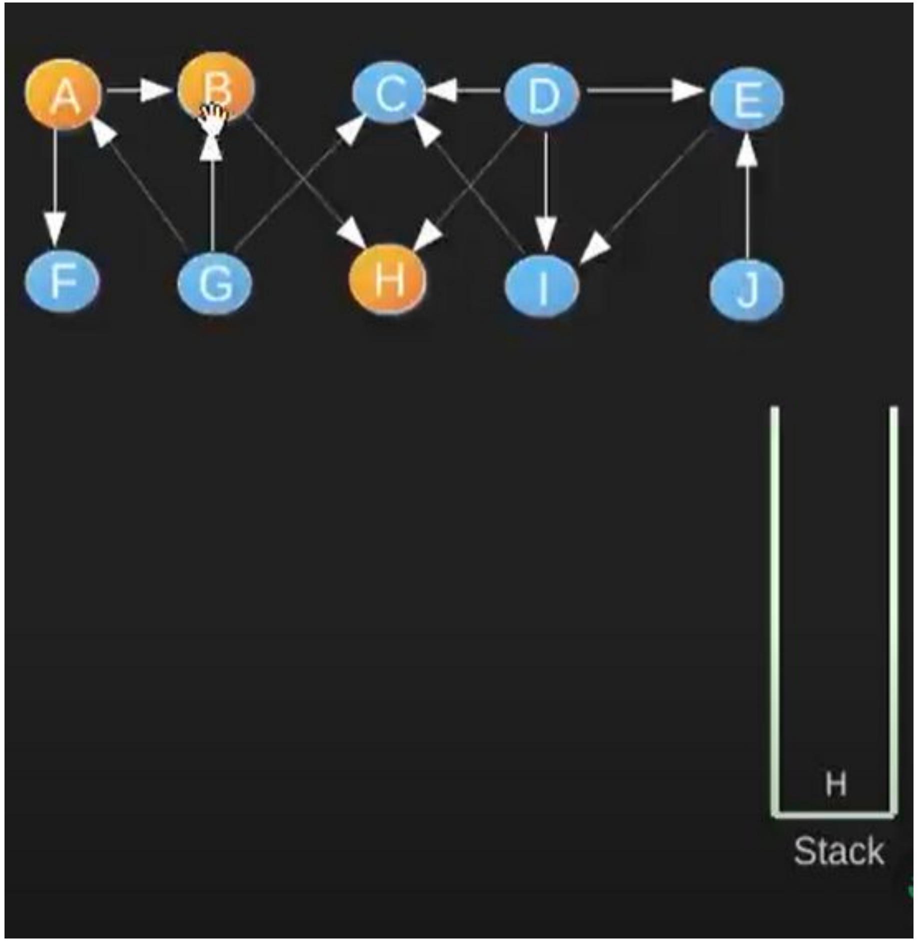
A C B D E

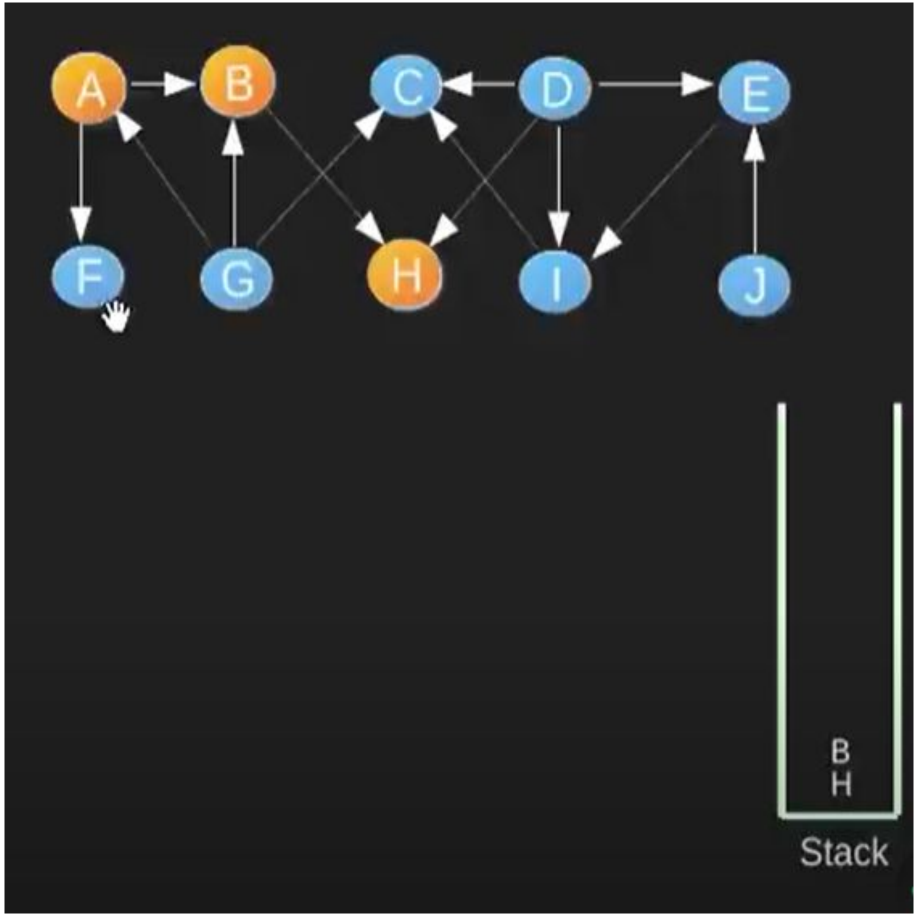


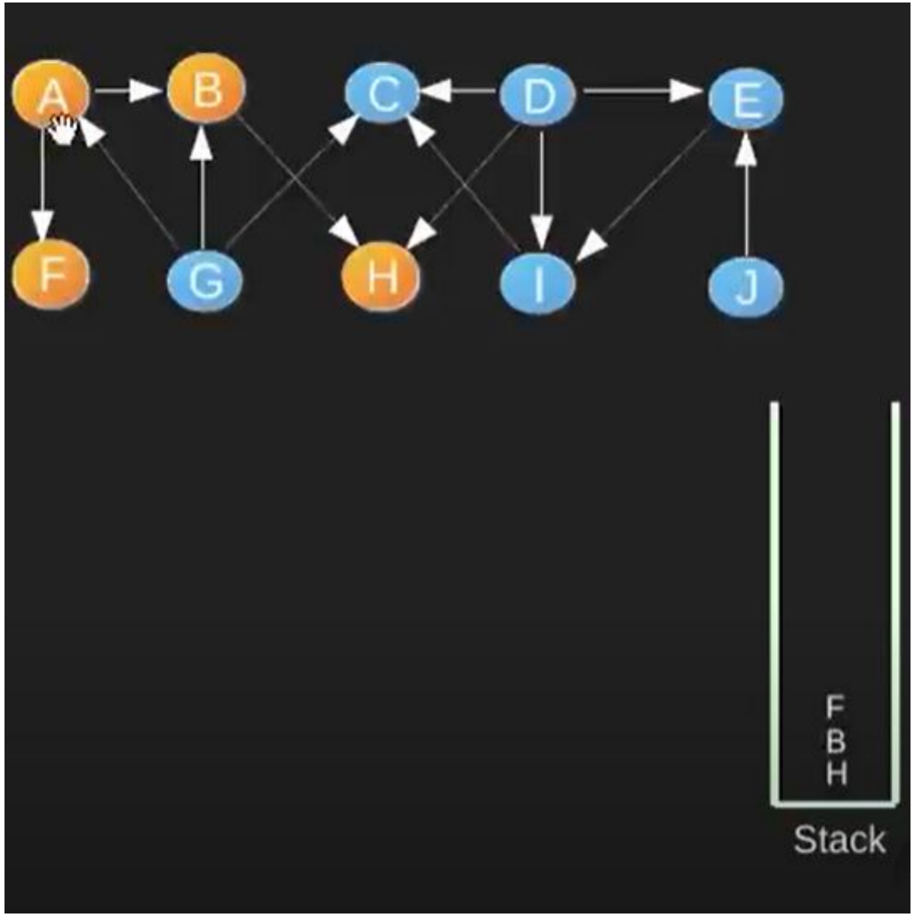


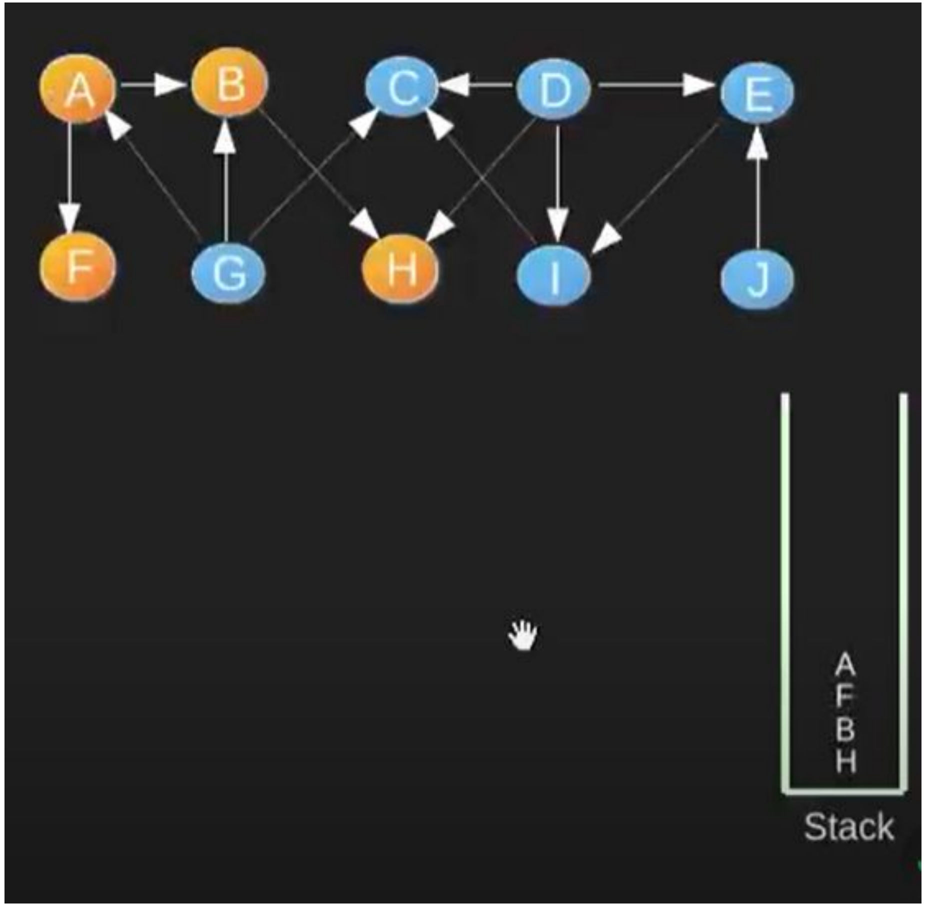
Topological Sort

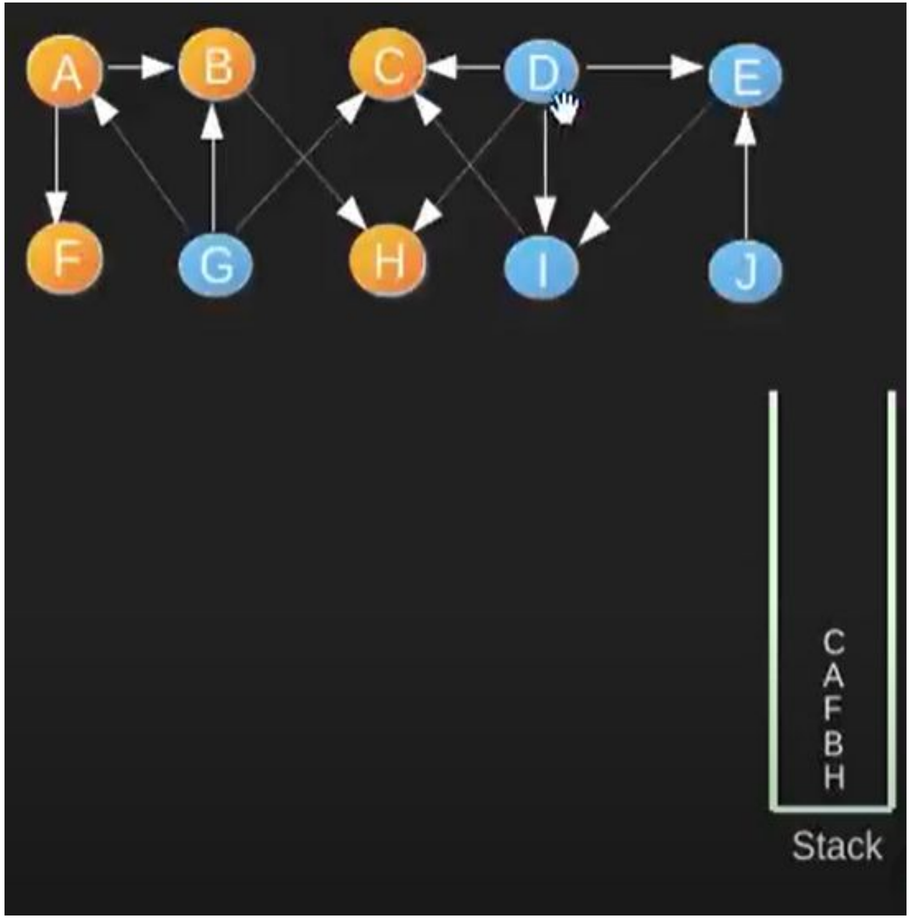


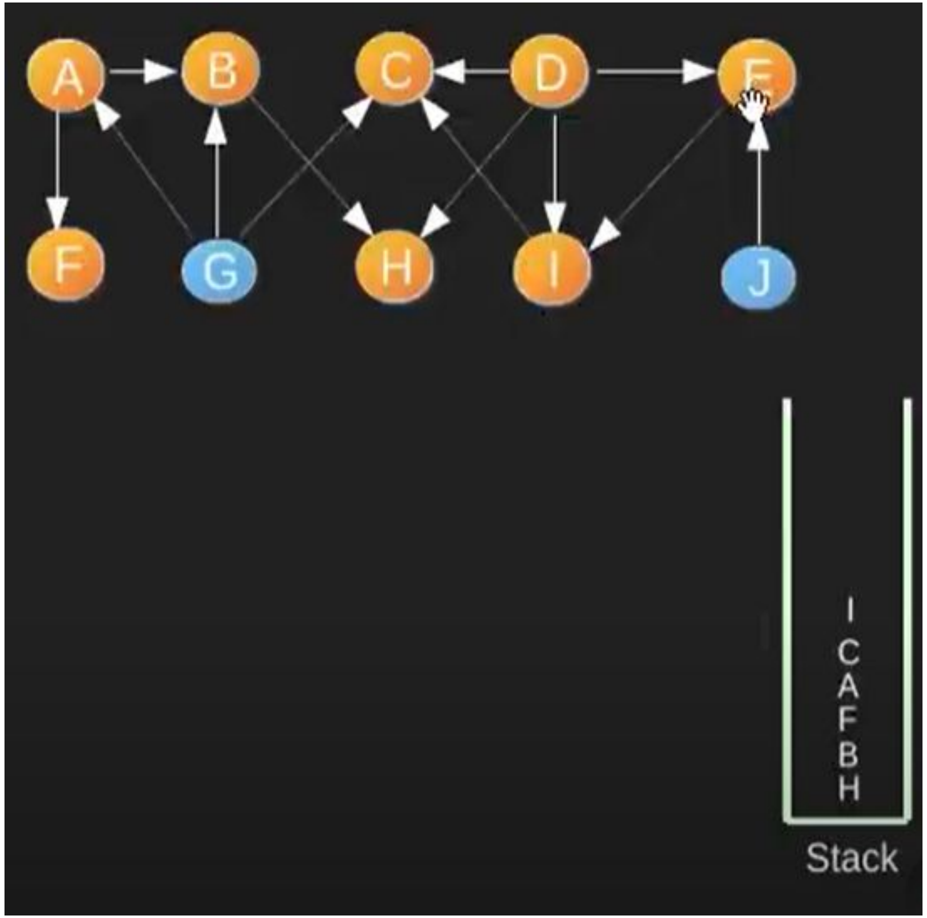

A
B
C
D
E
F
G
H
I
J
H
Stack

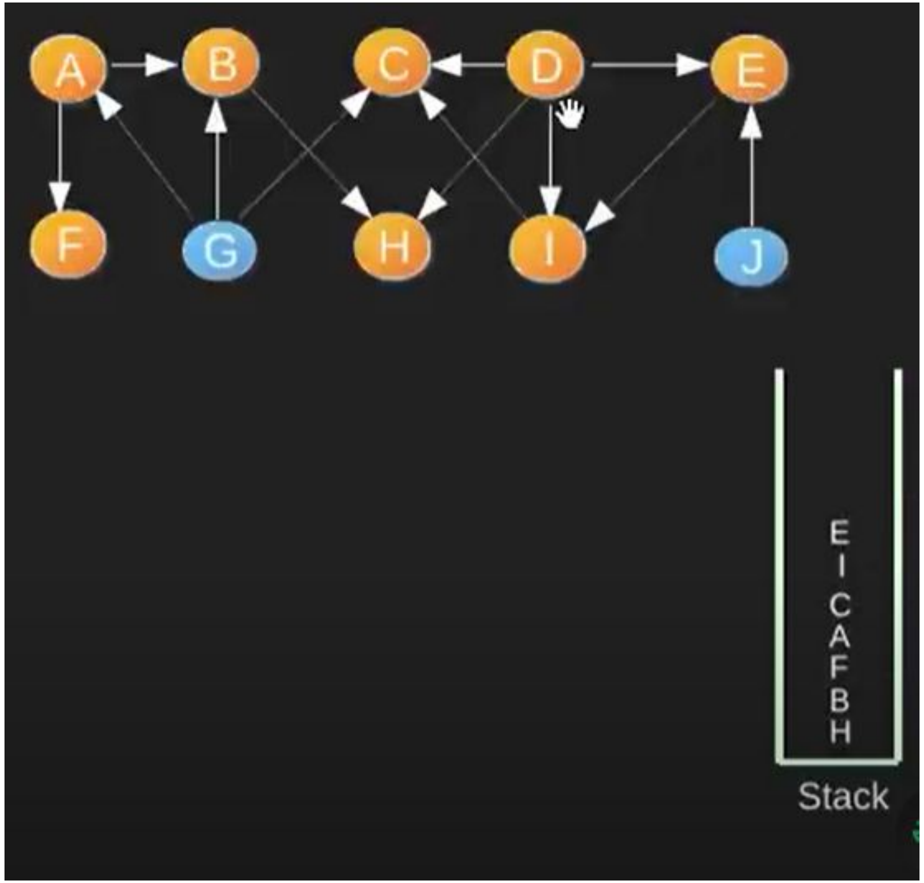

A
B
C
D
E
F
G
H
I
J
B
H
Stack

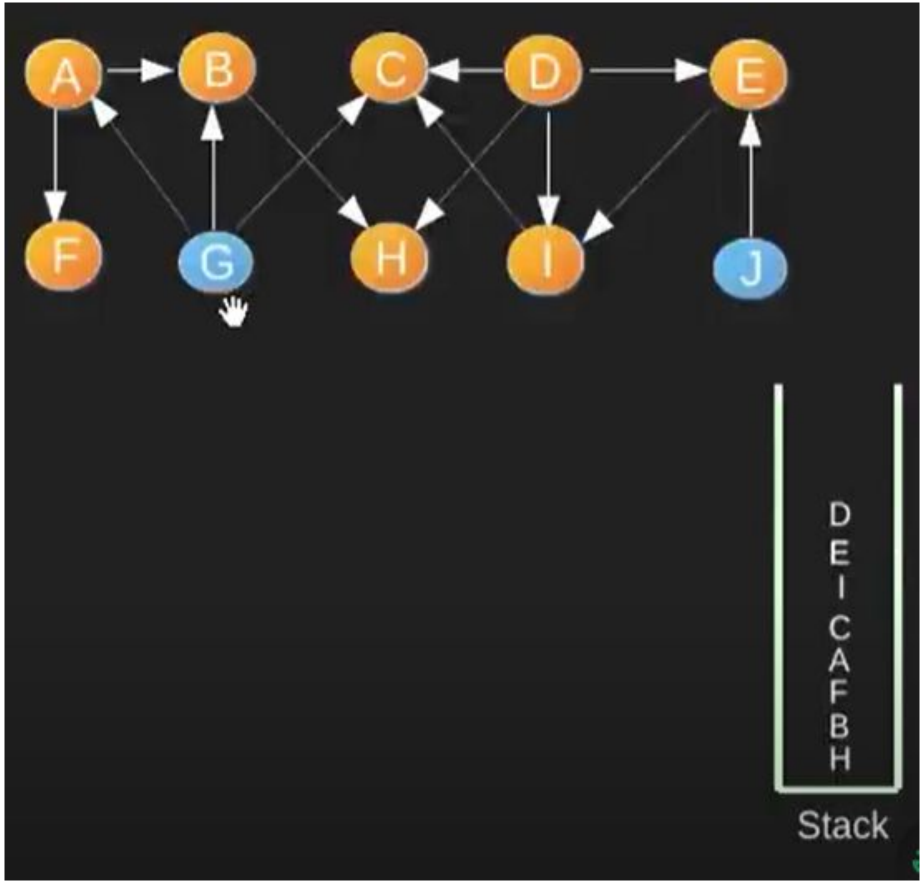

A
B
C
D
E
F
G
H
I
J
F
B
H
Stack

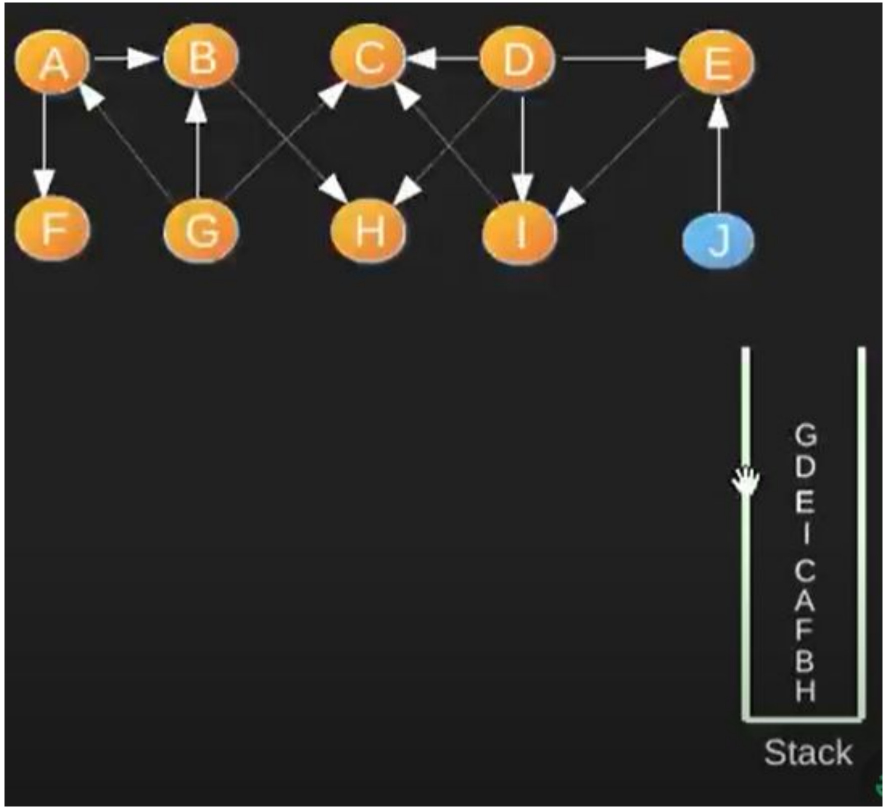

A
B
C
D
E
F
G
H
I
J
A
F
B
H
Stack

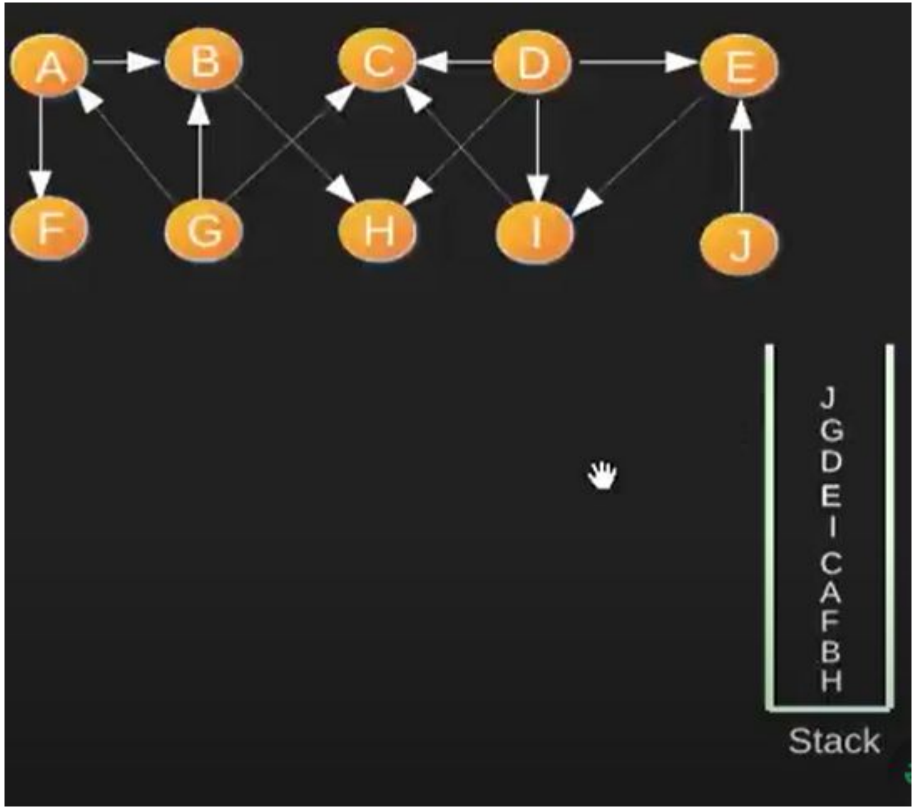

A
B
C
D
E
F
G
H
I
J
C
A
F
B
H
Stack


A
B
C
D
E
F
G
H
I
J
I
C
A
F
B
H
Stack


A
B
C
D
E
F
G
H
I
J
E
I
C
A
F
B
H
Stack

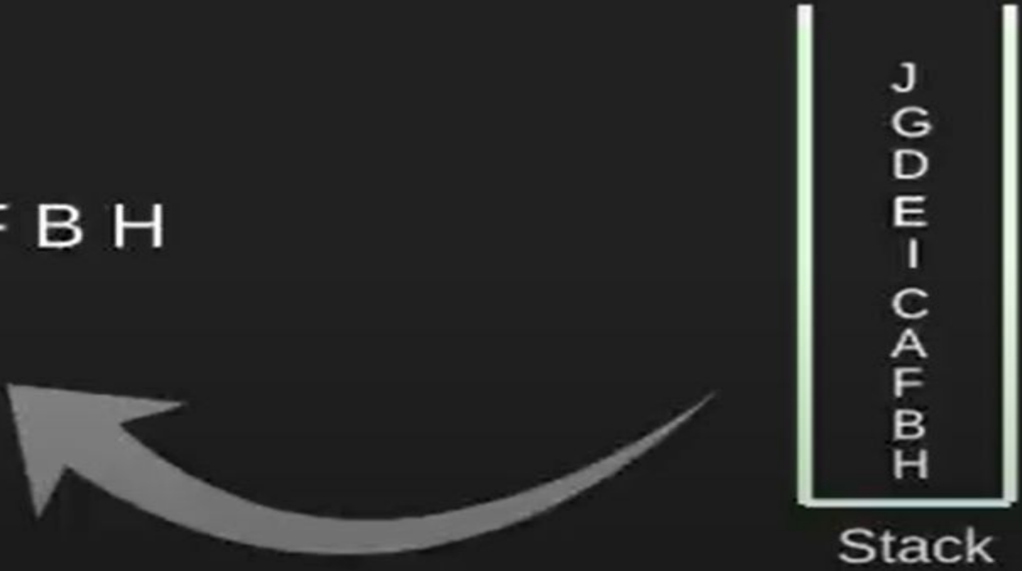

A
B
C
D
E
F
G
H
I
J
D
E
I
C
A
F
B
H
Stack

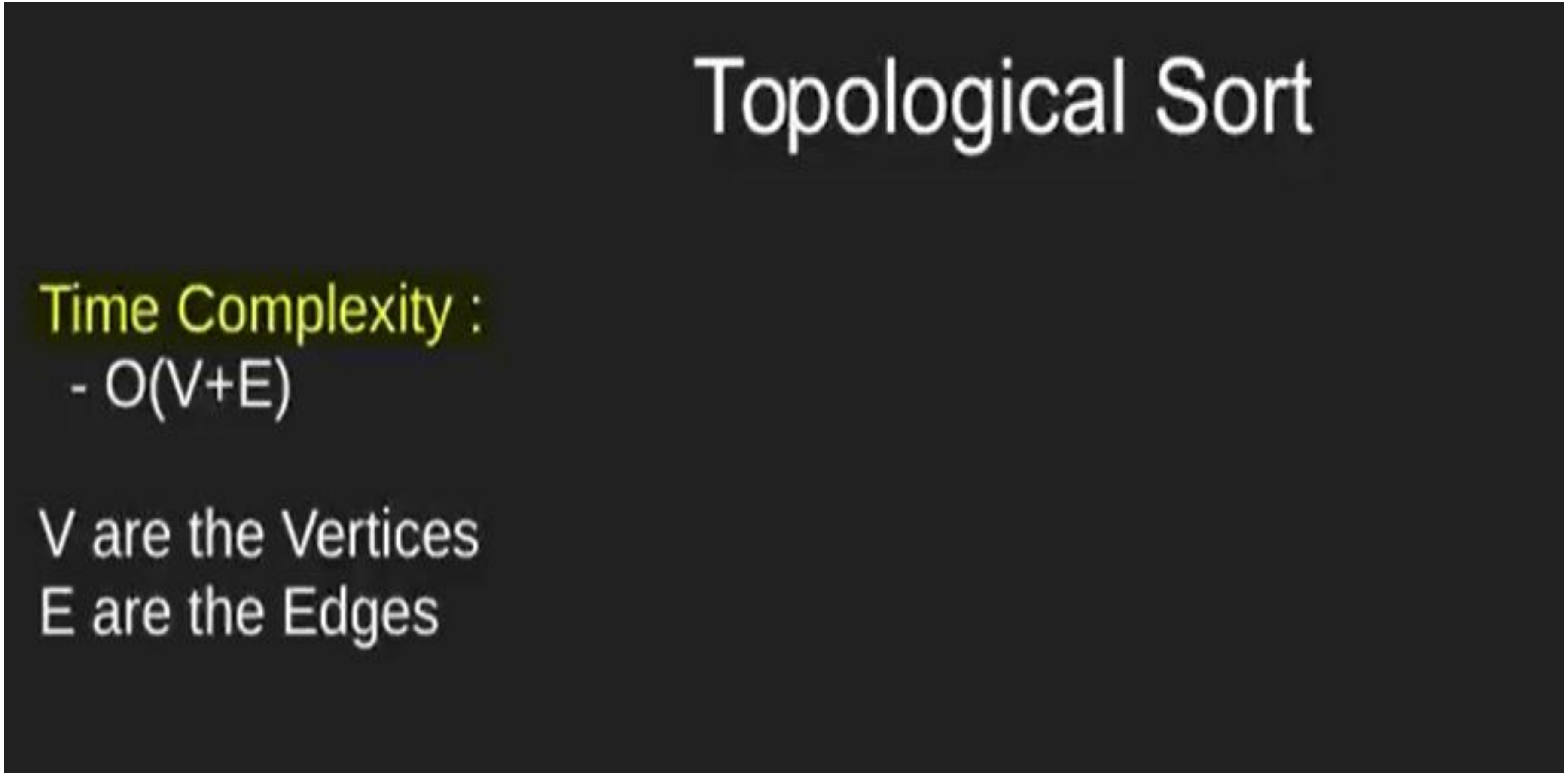

A
B
C
D
E
F
G
H
I
J
G
D
E
I
C
A
F
B
H
Stack


A
B
C
D
E
F
G
H
I
J
J
G
D
E
I
C
A
F
B
H
Stack

$$\begin{tikzcd}
A \arrow[r] \arrow[d] & B \arrow[dr] & C & D \arrow[l] \arrow[r] \arrow[dl] \arrow[d] & E \arrow[dl] \\
F & G \arrow[ul] \arrow[u] \arrow[ur] & H & I \arrow[ul] & J \arrow[u]
\end{tikzcd}$$

J G D E I C A F B H





Topological Sort

Time Complexity :
- O(V+E)

V are the Vertices
E are the Edges

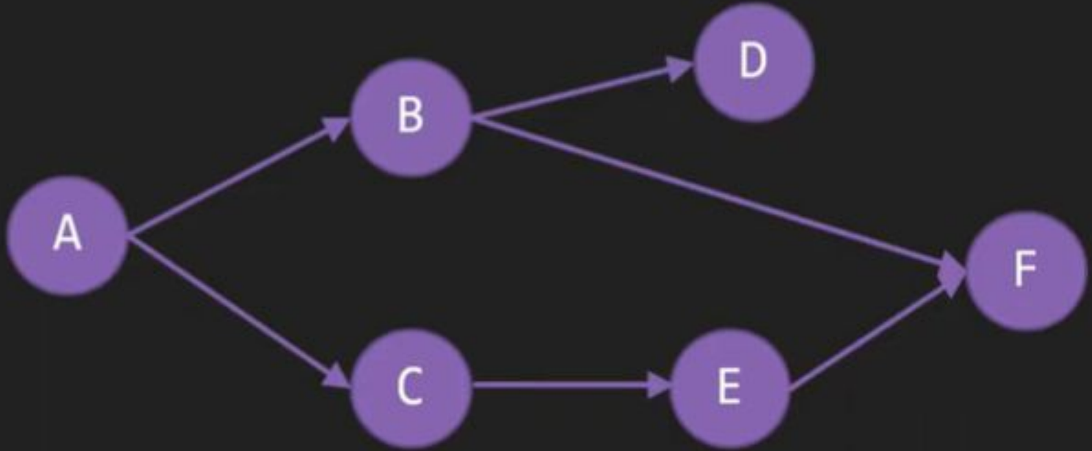
Directed Acyclic Graph (DAG)

- A directed graph
- With no directed cycles

$$\begin{tikzcd}
 & & D & \\
 & B \arrow[ur] \arrow[drr] & & \\
A \arrow[ur] \arrow[dr] & & & F \\
 & C \arrow[r] & E \arrow[ur] &
\end{tikzcd}$$

DAG Meaning

- Edges represent dependencies
- Can't do F until we've done A, B, C, and E


$$\begin{tikzcd}
 & B \arrow[r] \arrow[rrd] & D & \\
A \arrow[ru] \arrow[rd] & & & F \\
 & C \arrow[r] & E \arrow[ru] &
\end{tikzcd}$$

Topological Sorting

Topological sorting is a linear ordering of vertices in a Directed Acyclic Graph **(DAG)** such that for every directed edge (U → V), the vertex U appears before V in the ordering. It is only possible in DAGs.

<u>Steps to be followed for Topological sorting:</u>

- **DFS Exploration Order**: Explain that DFS visits deeper nodes first.
- **Stack Push Order:** Show that nodes are pushed to the stack only after all their dependencies are resolved.
- **Reversing the Stack:** Emphasize that the correct topological order is obtained by popping elements from the stack.

Why Do We Reverse the Stack?

- The **first element** pushed into the stack is the **last to be completed** in DFS.
- The **last element** pushed is the **first one that should appear** in topological order.
- **Reversing the stack** ensures that dependencies appear **before** dependents.
- Thus, when we print the stack in reverse order, we get the correct **topological sort order**.

Algorithm (Using DFS):

```
def topological_sort(graph, node, visited, stack):
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            topological_sort(graph, neighbor, visited, stack)
    stack.append(node)
```

Applications of Topological Sorting:

- Task scheduling (e.g., job scheduling in an operating system)
- Course prerequisite management
- Dependency resolution (e.g., package installation order)

- Topological ordering is useful for **scheduling tasks** where some tasks depend on the completion of others.
- Topological order can be used to efficiently solve **dynamic programming problems** on DAG.
- It helps resolve dependencies in various systems, such as **software build processes** or project management.
- Topological ordering can be used to quickly compute **shortest paths** through a weighted directed acyclic graph.

Divide and Conquer Technique

Divide-and-conquer algorithms work according to general plan:

1. A problem is divided into **several subproblems** of the same type, ideally of about **equal size**.

2.The subproblems are **solved** (typically recursively, though sometimes a different algorithm is employed)

3. If necessary, the solutions to the subproblems are **combined** to get a solution to the original problem.

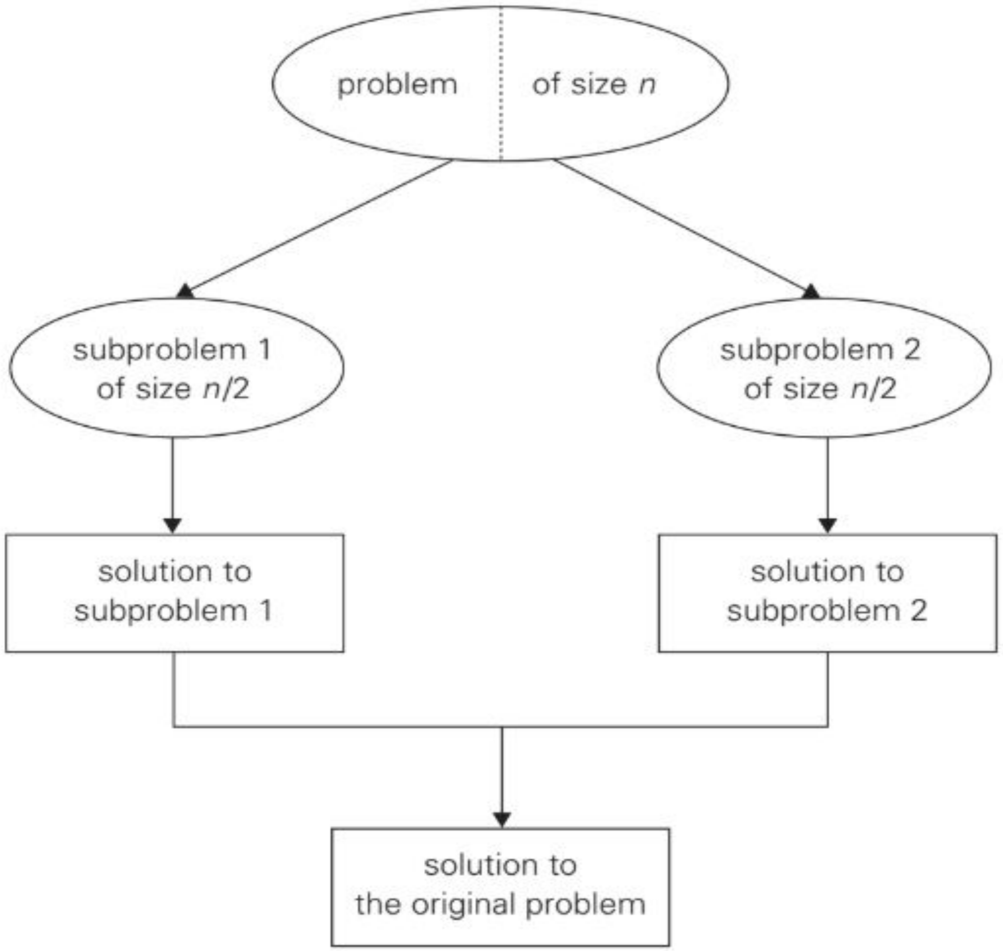


FIGURE 5.1 Divide-and-conquer technique (typical case).

Masters Theorem for recurrence relations

- We use the **Master Theorem** to quickly find the **time complexity** of divide-and-conquer problems. It applies to recurrence relations of this type:

$$T(n)=aT(n/b)+f(n)$$

- It provides a systematic way to determine the time complexity of **recursive algorithms**

Time complexity : $T(n)=aT(n/b) + f(n)$

where:

- **n** = Input size
- **a** = Number of subproblems
- **b** = How much each subproblem reduces (size of n)
- **f(n)** = time spent on dividing an instance of size n (splitting/merging)

Mergesort Algorithm

- DIVIDE the Array into halves
- RECURSIVELY sort the two halves
- COMBINE the two sorted subsequences by merging them

Merge sort is a sorting algorithm follows **divide-and-conquer** approach. It works by recursively dividing input array into smaller subarrays and sorting those subarrays then merging them back together to obtain sorted array.

How does Merge Sort work ?

1. **Divide:** Divide the list or array recursively into two halves.
2. **Conquer:** Each subarray is sorted individually using merge sort algo.
3. **Merge:** Sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Example for Merge Sort

Step 1: Divide

[38, 27, 43, 3] | [9, 82, 10]

Step 2: Further Divide

[38, 27] | [43, 3] | [9, 82] | [10]

Step 3: Base Case Reached (Single Elements)

[38] | [27] | [43] | [3] | [9] | [82] | [10]

Step 4: Merge Pairs

[27, 38] | [3, 43] | [9, 82] | [10]

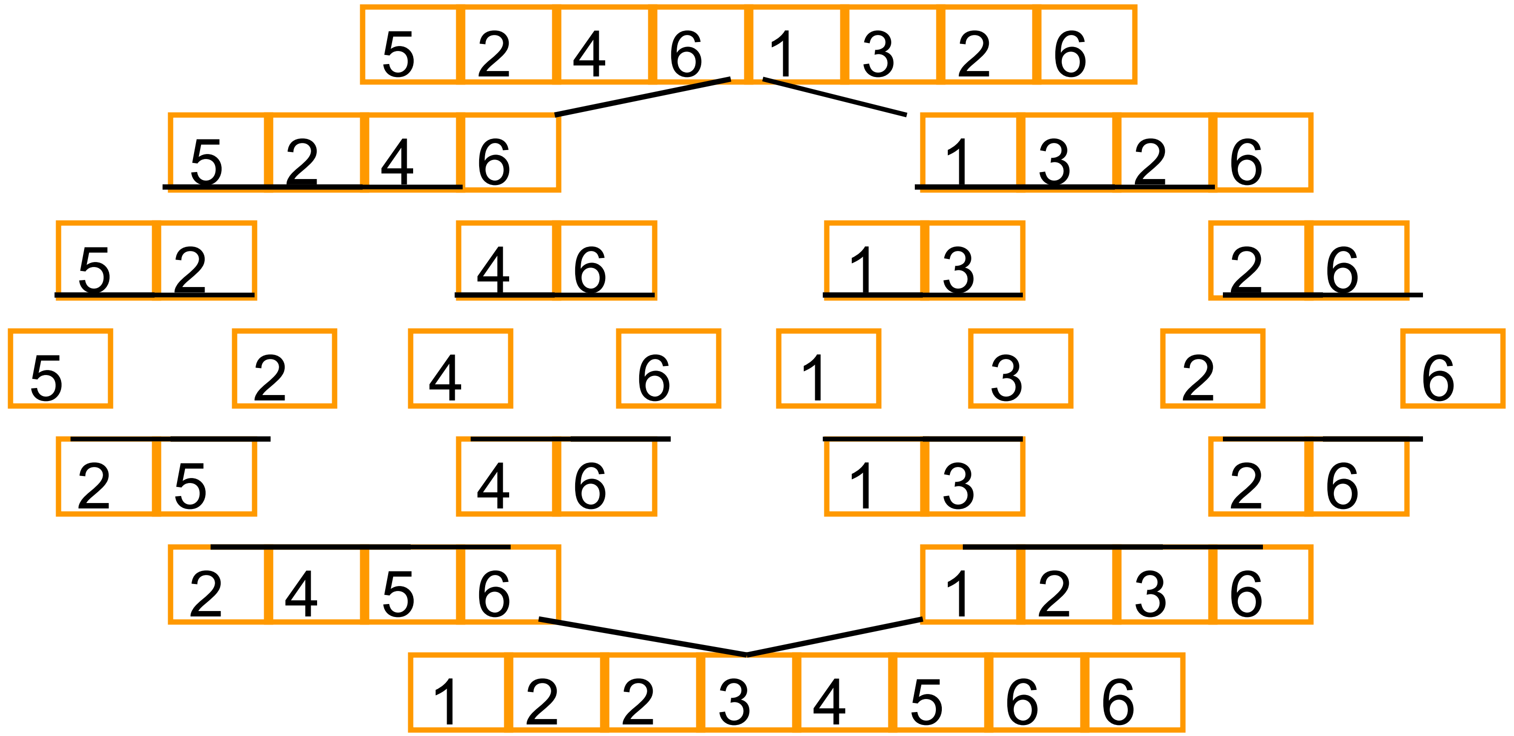
Step 5: Merge Again

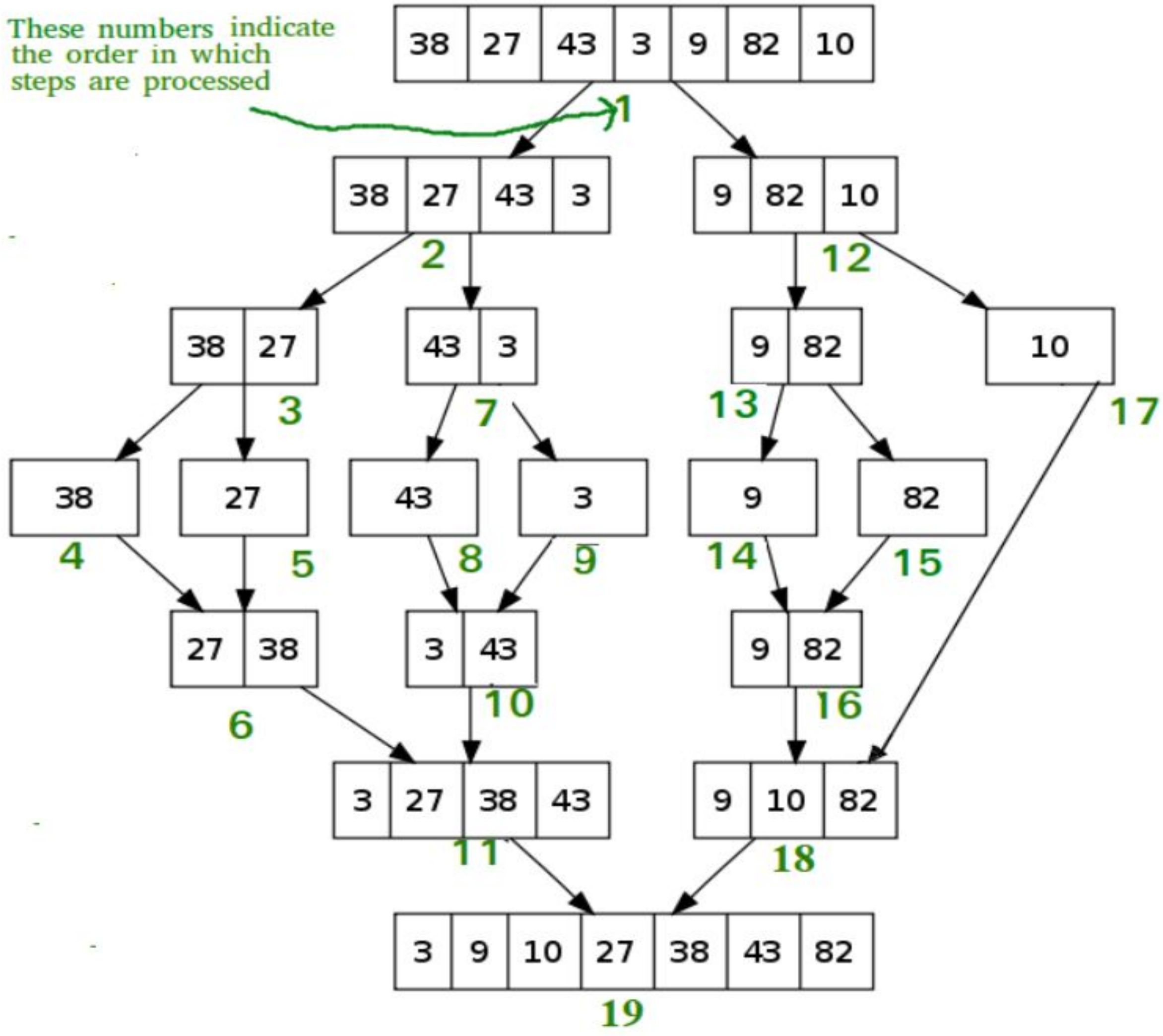
[3, 27, 38, 43] | [9, 10, 82]

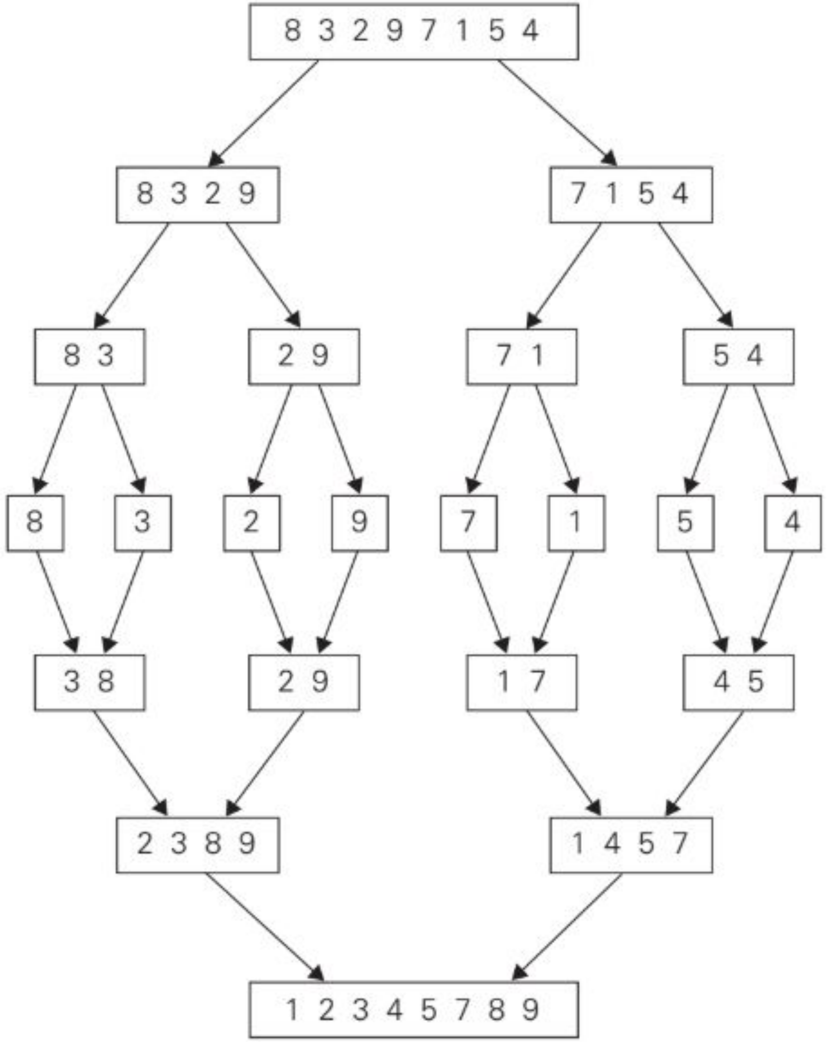
Step 6: Final Merge

[3, 9, 10, 27, 38, 43, 82] (Sorted Array)

Mergesort Example




These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
4
5
8
9
14
15
27 38
3 43
9 82
6
10
16
3 27 38 43
9 10 82
11
18
3 9 10 27 38 43 82
19


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Algorithm:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left < right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
step 4: Stop
```

```
ALGORITHM  Mergesort($A[0..n-1]$)
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
        Mergesort($C[0..\lceil n/2 \rceil - 1]$)
        Merge($B, C, A$)   //see below
```

Pseudocode

```
Merge(A, left, mid, right)

1. Create two temporary arrays:

    LeftArray = A[left...mid]

    RightArray = A[mid+1...right]

2. Set i = 0 (index for LeftArray)

3. Set j = 0 (index for RightArray)

4. Set k = left (index for merged array)
```

```
5. While i < size of LeftArray AND j < size of RightArray:
    If LeftArray[i] ≤ RightArray[j]:
        A[k] = LeftArray[i]
        i = i + 1
    Else:
        A[k] = RightArray[j]
        j = j + 1
    k = k + 1
6. Copy remaining elements of LeftArray (if any) into A
7. Copy remaining elements of RightArray (if any) into A
```

Merge Sort Time Complexity:

- Time Complexity: O(n log n)
- Space Complexity: O(n)

Applications of Merge Sort:

- Sorting linked lists.
- External sorting (e.g., sorting huge datasets that don’t fit into RAM).
- Stable sorting (preserving the order of equal elements).

Quick Sort:

<u>Steps:</u>

- Pick a **pivot** (typically the last or first element).
- Partition the array:
  - Move smaller elements to the left of the pivot.
  - Move larger elements to the right of the pivot.
- Recursively apply Quick Sort on the left and right partitions.

QuickSort

It is a sorting algorithm based on Divide and Conquer that picks an element as a pivot and partitions given array around picked pivot by placing pivot in its correct position in sorted array.

Time Complexity:

- **Average Case**: O(N logN), where N is the length of the array.
- **Best Case:** O(N logN)
- **Worst Case:** O(N^2)

Example: Sorting [10, 5, 2, 8, 7, 3]

Step 1: Choose Pivot We pick the last element as the pivot: 3

Step 2: Partition the Array

- Left Partition (≤ 3): [2]
- Pivot: [3]
- Right Partition (> 3): [10, 5, 8, 7]

Now, recursively sort the left and right partitions.

Step 3: Sort Left Partition [2] Since it has only one element, it remains [2].

Step 4: Sort Right Partition [10, 5, 8, 7]

- Pivot = 7
- Left Partition (≤ 7): [5]
- Pivot: [7]
- Right Partition (> 7): [10, 8]

Now, sort [5] (already sorted) and [10, 8]

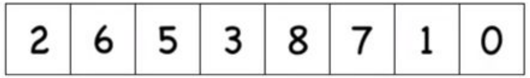
Step 5: Sort Right Partition [10, 8]

- Pivot = 8
- Left Partition (≤ 8): [] (empty)
- Pivot: [8]
- Right Partition (> 8): [10] Sorted result: [8, 10]

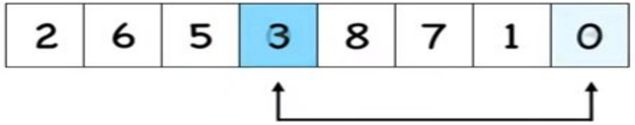
Step 6: Merge Results [2] + [3] + [5] + [7] + [8, 10]

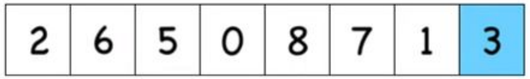
Quick Sort

1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

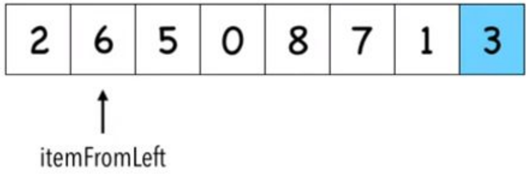


1. itemFromLeft that is larger than pivot

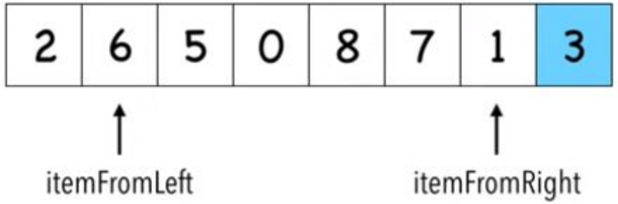




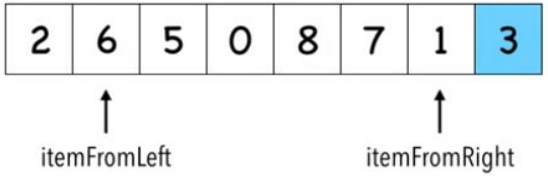
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



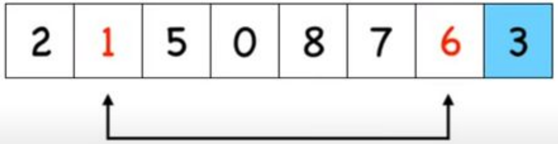
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



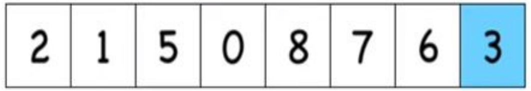
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

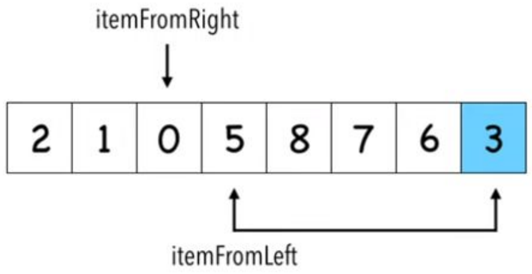


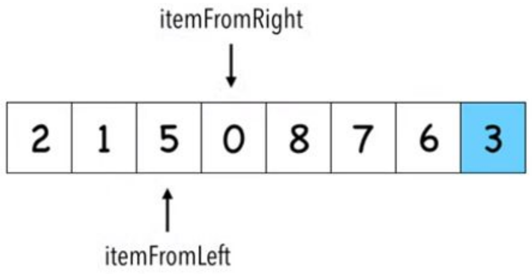
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



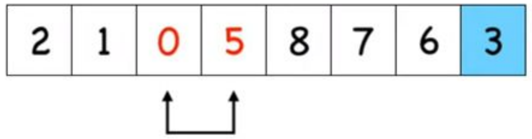
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

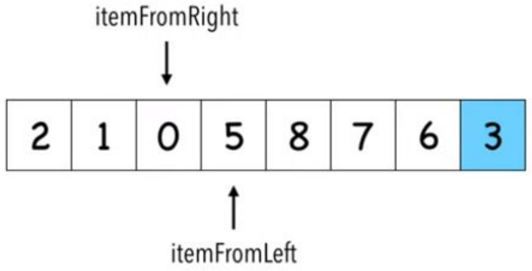






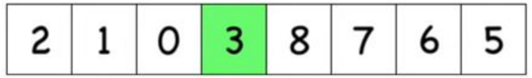
Swap itemFromLeft and pivot



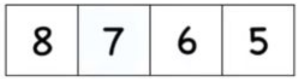


Stop when index of itemFromLeft > index of itemFromRight

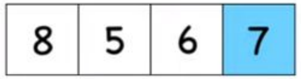
1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

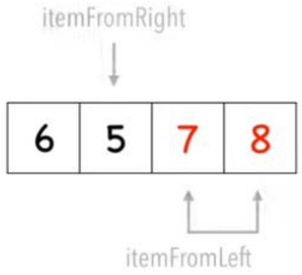


How to choose pivot:
Median position element:



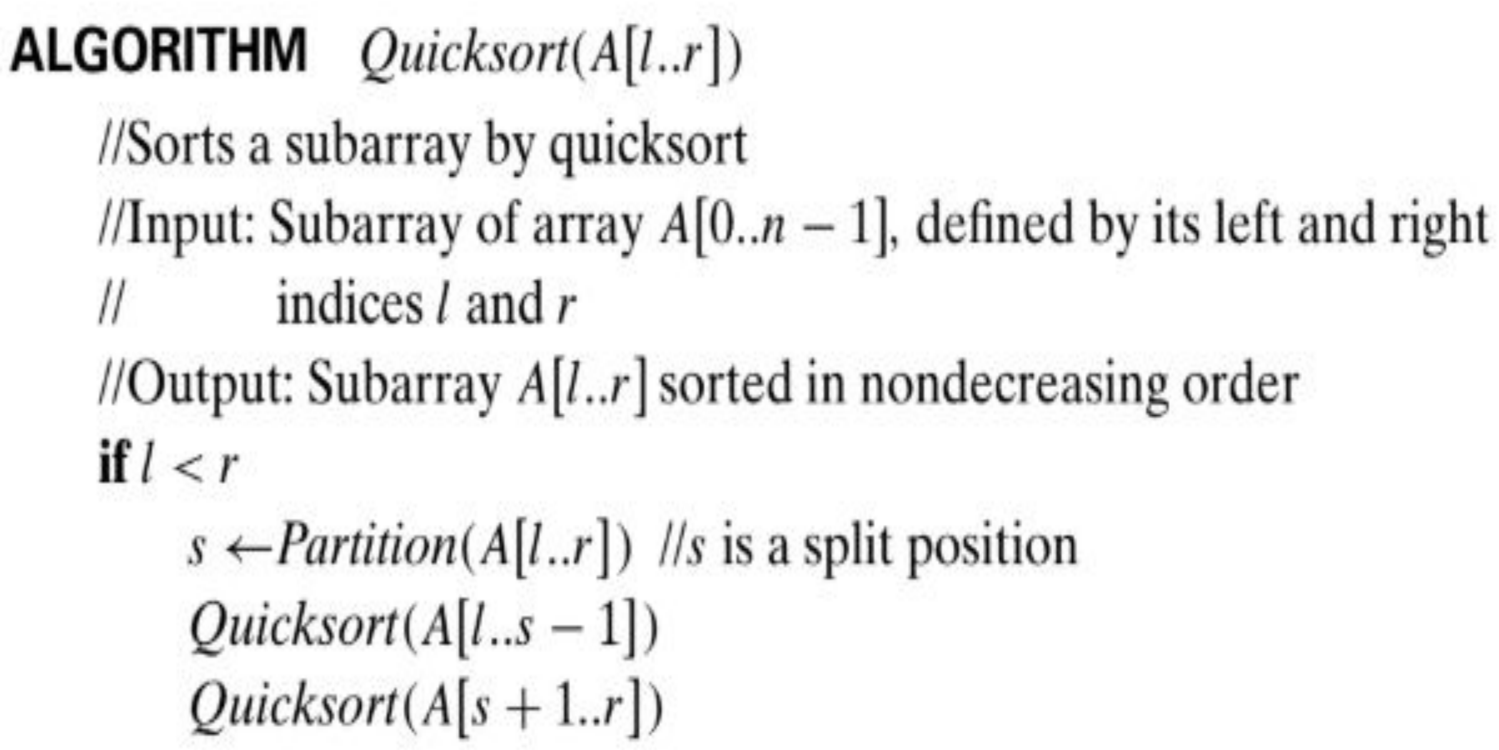
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot





Worst Case Complexity: $O(n^2)$
Average Case Complexity: $\Theta(n \log n)$

Algorithm:



```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if $l < r$
        $s \leftarrow$ Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

Pseudocode

```
if low < high:
    # Step 1: Partition the array around the pivot
    pivotIndex = Partition(A, low, high)

    # Step 2: Recursively apply to left subarray
    QuickSort(A, low, pivotIndex - 1)

    # Step 3: Recursively apply to right subarray
    QuickSort(A, pivotIndex + 1, high)
```

```
pivot = A[high]
i = low - 1  # index of smaller element

for j = low to high - 1:
    if A[j] <= pivot:
        i = i + 1
        swap A[i] with A[j]

# After loop, place pivot in correct position
swap A[i + 1] with A[high]
return i + 1  # Return pivot index
```

Example for Quick Sort

Let's say A = [8, 4, 7, 3, 10]

Step 1: QuickSort(A, 0, 4)

- Pivot = 10
- All elements ≤ 10 → no swaps needed (though the algorithm still checks)
- Final step: swap 10 with itself → pivot at index 4
- Recurse on [8, 4, 7, 3] and []

Step 2: QuickSort(A, 0, 3)

- Pivot = 3
- Compare:
  - 8 > 3 → no swap
  - 4 > 3 → no swap
  - 7 > 3 → no swap
- Swap pivot (3) with A[0] → [3, 4, 7, 8, 10]
- Recurse on [] and [4, 7, 8]

Applications of Quick Sort

- Applied in cryptography for generating random permutations and unpredictable encryption keys.
- Partitioning step can be parallelized for improved performance in multi-core or distributed systems.
- Important in theoretical computer science for analyzing average-case complexity and developing new techniques.
- Efficient for sorting large datasets with **O(n log n)** average-case time complexity.

Problem solving Techniques

Merge Sort

1. Input: A = [8, 3, 1, 7]
2. Input: A [5, 2, 9, 1, 6]
3. Input: A [4, 6, 2, 5, 1, 3]

Quick Sort

1. Input: A = [5, 3, 8, 4, 2]
2. Input: A = [9, 7, 5, 11, 12, 2, 14, 3, 10]
3. Input: A = [1, 2, 3, 4, 5]