Day 17 Graph

ITSRUNTYM

What is a Graph?

A **Graph** is a **non-linear data structure** that consists of:

- **Vertices (or Nodes)** → Entities
- **Edges** → Connections between those entities

A graph G is defined as G = (V, E) where:

- V = set of vertices/nodes
- E = set of edges connecting the vertices

Types of Graphs

Category	Type	Description
Based on Direction	**Undirected**	Edges have no direction → (A—B)
	Directed (Digraph)	Edges have direction → (A → B)
Based on Weight	**Unweighted**	All edges are equal
	Weighted	Each edge has a cost/ weight
Based on Connectivity	**Connected**	Every node reachable from another
	Disconnected	Some nodes cannot reach others
Based on Cycles	**Cyclic**	Contains at least one cycle
	Acyclic	No cycles

Graph Representation

1. **Adjacency Matrix**:
   A 2D array graph[i][j] = 1 if there is an edge between i and j
   Space: $O(V^2)$
2. **Adjacency List**:
   A list of lists where each node stores its neighbors
   Space: O(V + E) → Most efficient

Applications of Graphs

Domain	Use Case	Graph Type

Maps & GPS	Shortest path between cities	Directed Weighted
Internet	Web page links (Google PageRank)	Directed
Social Networks	Friends/followers	Undirected or Directed
E-Commerce	Product recommendation	Weighted, Undirected
Task Scheduling	Course/job scheduling	Directed Acyclic Graph (DAG)
Puzzle Solving	State transitions	Graph traversal
Network Security	Packet routing/ firewalls	Graphs

When to Use Graphs in DSA Problems

- When relationships/connections matter (like roads, dependencies, networks)
- If you are modeling:
  - Routes (shortest, cheapest, all paths)
  - Dependencies (topological sorting)
  - Networks (reachability, cycles)
  - Games or puzzles with state changes

Example DSA Problems

Problem	**Type**	**Algorithm**
Find shortest path (e.g., city to city)	Directed Weighted	Dijkstra's / Bellman-Ford
Detect cycle in dependencies	Directed	DFS with visited[]
Can finish all courses	DAG	Topological Sort
Count islands in a grid	Undirected	BFS / DFS
Clone a graph	Undirected	BFS / DFS

Representations of Graph
Here are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.
Adjacency Matrix
Adjacency List

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.

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. Let's assume there are n vertices in the graph So, 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.

#	Problem	Concept	LeetCode Link
1	**Number of Provinces**	Connected Components (Undirected Graph)	🔗 LeetCode 547
2	**Number of Islands**	Graph on Grid (DFS/BFS)	🔗 LeetCode 200
3	**Flood Fill**	DFS/BFS on 2D Matrix	🔗 LeetCode 733
4	**Max Area of Island**	DFS on Grid	🔗 LeetCode 695
5	**Find if Path Exists in Graph**	Path Existence using BFS/DFS	🔗 LeetCode 1971
6	**Clone Graph**	Graph Traversal and Copy	🔗 LeetCode 133
7	**Is Graph Bipartite?**	BFS/DFS Coloring	🔗 LeetCode 785
8	**Course Schedule**	Cycle Detection in DAG (DFS)	🔗 LeetCode 207

9	**Course Schedule II**	Topological Sort	🔗 LeetCode 210
10	**Rotting Oranges**	Multi-source BFS	🔗 LeetCode 994