GRAPH

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Number of Islands**			
	Given grid where '1' is land & '0' is water, return # of islands	**- Main function** : Iterates through the grid & calls DFS --> `if(grid[i][j]=='1')` **- DFS** : Marks all the cells in current connected component as visited	O(M*N)	O(M+N)
2	**Max Area of Island**			
	Given an m x n binary matrix grid. The area of an island is the number of cells with a value 1 in the island. Return the maximum area of an island in grid.	- **Main function** : Iterates through the grid & calls DFS --> `if(grid[i][j]=='1')` - **DFS** : Marks all the cells in current connected component as visited, and calculates the area of current connected component	O(M*N)	O(M+N)
3	**Clone Graph**			
	Given ref of a node in connected undirected graph, return a deep copy	- Traverse the original graph using either **BFS** or **DFS** traversal - Use **unordered_map** to map `{original node --> clone node}`	O(V+E)	O(V)
4	**Walls and Gates**			
	Given m x n grid where -1 = wall, 0 = gate, INF = empty. Fill each empty with distance to nearest gate	- Main Idea: **Multisource BFS** - sources would be all the cells with grid[i][j] == 0	O(M*N)	O(M*N)
5	**Rotting Oranges**			
	Given grid: 0 empty cell, 1 fresh orange, 2 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.	- Use `queue<pair<int, int>>` to store the coordinates of rotten oranges. `n_oranges` to count the total number of fresh oranges. - Use **Level Order BFS** to simulate the rotting process. - Update the 'cnt' at **each level** - if at the end `n_oranges == 0` then `return cnt` else `retrun -1`	O(M*N)	O(M*N)
6	**Pacific Atlantic Waterflow**			
	Top & left side : Pacific ocean, Bottom & right side: Atlantic ocean, Island is partitioned into a grid, return a list of grid coordinates from which the rain can flow to both the oceans	- Boolean arrays **pacific** : To track {i,j} that can reach pacific ocean | **atlantic** : To track {i,j} that can reach atlantic ocean - First row and column can reach pacific directly | Last row and Last column can reach atlantic directly - for(int i=0; i<**n**; i++) - **First row** can directly reach **pacific** Use BFS (**0, i**, m, n, **pacific** , heights) to mark {i,j} that can reach pacific via **that** cell (value of adj cells >= current cell) - **Last row** can directly reach **atlantic** Use BFS (**m-1, i** , m, n, **atlantic** , heights) to mark {i,j} that can reach atlantic via **that** cell (value of adj cell >= current cell) - for(int i=0; i<**m**; i++) - **First column** can directly reach **pacific** Use BFS (**i, 0** , m, n, pacific, heights) to mark {i,j} that can reach pacific via **that** cell (value of adj cells >= current cell) - **Last column** can directly reach **atlantic** Use BFS (**i, n-1** , m, n, **atlantic** , heights) to mark {i,j} that can reach atlantic via **that** cell (value of adj cell >= current cell) - Pus to 'ans' if {i,j} can reach **both** pacific and atlantic ocean	O(M*N)	O(M*N)
7	**Surrounded Regions**			
	Given an m x n matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'. *(A region is captured by flipping all 'O's into 'X's in that surrounded region.)*	- Main idea: 1) Use **DFS** to find all the cells {i,j} which does not need to be flipped, Mark those board[i][j] with '#' 2) i.e All the connected componets starting from the **border** (first row | las row | first col | last col)	O(M*N)	O(M*N)
8	**Course Schedule I**			

GRAPH

No.	Problem Statement	Solution	Time complexity	Space complexity
	Given 'numCourses', labeled from 0 to numCourses - 1, prerequisites[i] = [ai, bi] : Must take course bi first to take ai. Return true if you can finish all courses in 'numCourses'	- Create a **directed graph** where edge "*u-->v*" represents "*prerequisuite course --> main course*" - Detect **Cycle** (in Directed graph using **DFS**): Possible to finish all courses if there's no cycle	O(V+E)	O(V+E)
9	**Course Schedule II**			
	Given 'numCourses', labeled from 0 to numCourses - 1, prerequisites[i] = [ai, bi] : Must take course bi first to take ai. Return the ordering of courses you should take to finish all courses. If it is impossible to finish all courses, return an empty array.	- Create a directed graph where edge "u-->v" represents "prerequisuite course --> main course" - Perform **Topological sort** to get the course ordering	O(V+E)	O(V+E)
10	**Graph Valid Tree**			
	Given graph of n nodes & **a list of edges** in the graph, determine if edges make valid tree	- Graph is a tree if, 1) It's **connected** (**n** nodes --> exactly **n-1 edges**) 2) has **no cycle** 1) To check if we have n-1 edges, `edges.size() == n-1` 2) Use **Union Find** algorithm to Detect Cycle	O(E*logV)	O(V)
11	**Number of Connected Components I**			
	Given graph of n nodes & a **list of edges** in the graph, return # of connected components	- Initialialy, **No. of connected components = V** - **Union Find** : For each edge(u,v) check if u and v belong to the same set - If Yes: no. of connected components remain the same - If No: decrease the count of connected components by 1	O(ElogV)	O(V)
12	**Number of Connected Components II**			

	Given graph of n nodes & an **adjacency list representation** of the graph,
	return # of connected components
		- Use either **DFS** or **BFS**
		- Initially, **No. of connected components = 0**
		- Use a boolean array 'visited' to keep track of visited nodes
		- `for(int i=0; i<n; i++)`
		`if(!visited[i])`
		`DFS(i, visited, adj)` // Mark all the nodes in the same connected componets as visited
		`cnt++`
			O(E+V)	O(V)

13	**Redundant Connection**			
	Given an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph. Return an edge that can be removed so that the resulting graph is a tree.	- Main Idea: If we have **n nodes** & **n edges** then thre is **guaranteed a cycle** - Use: **Union Find Algorithm** to detect the edge that forms a cycle	O(E*logV)	O(V)
14	**Min Cost to Connect All Points**			
	You are given an array points, where points[i] = [xi, yi], coordinates on a 2D plane. The cost of connecting two points [xi, yi] and [xj, yj] is the manhattan distance between them: |xi - xj| + |yi - yj|. Return the minimum cost to make all points connected.	- **Idea: Prim's Algorithm** - priority_queue --> {min_cost, index} - Consider, points[0] as source node - Initialize the PQ with Manhattan distance from all the other points to the source point - Follow the normal logic for prim's algorithm	O(V^2*logV)	O(V)
15	**Network Delay Time**			
	Signal sent from node k to network of 'n' nodes, all numbered through 1 to n, return time for all nodes to receive it	- **Dijkstra's algorithm** to find the shortest path from source node to all the other nodes in the graph	O(E*logV)	O(E+V)

<table>
<tr><th colspan="5">GRAPH</th></tr>
<tr><th>No.</th><th>Problem Statement</th><th>Solution</th><th>Time complexity</th><th>Space complexity</th></tr>
<tr><td><b>16</b></td><td><b>Cheapest Flight Within K Stops</b></td><td></td><td></td><td></td></tr>
<tr><td></td><td>There are n cities connected by some number of flights. Given an array flights where flights[i] = [from, to, price] indicates that there is a flight from city 'from' to city 'to' with cost 'price', three integers src, dst, and k, return the cheapest price from src to dst with at most k stops. If there is no such route, return -1.</td><td>- <b>Idea: Modified Dijkstra's Algorithm</b>
- Priority_queue of {cost, node, stops}
- To keep track of minimum distance from source node to every other node
vector<int> min_dist(n, INT_MAX);
min_dist[src] = 0;
- To keep track of minimum number of stops to reach a node from source
For example: All the adjacent nodes from source can be reached with 0 stops
vector<int> stops(n, INT_MAX);
stops[src]=-1;
- Modified Dijkstra
while(!pq.empty()){
auto x = pq.top();
pq.pop();
// Suggests we can reach 'u' from source with cost='d' and stops='s'
d=x[0]; u=x[1]; s=x[2];
if(u==dst) return d;
if(s<k){
for(auto v: adj[u]){
// Check for: better cost || Better stops
<b>if(min_dist[v.first] > d + v.second || stops[v.first] > s+1)</b>{
min_dist[v.first] = d + v.second;
stops[v.first] = s+1;
pq.push({min_dist[v.first], v.first, stops[v.first]});</td><td>O(E*logV)</td><td>O(E+V)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td><b>17</b></td><td><b>Word Ladder</b></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Given 2 words: beginWord and endWord, a dictionary: wordList, return min number of words to transform beginword to endword. Perform transformation: beginWord -> s1 -> s2 -> ... -> endWord such that,
1) Every adjacent pair of words differs by a single letter.
2) Every si is in the dictionary except the beginword.</td><td>- <b>Level Order BFS Traversal</b>
- At each level, change one letter and check if the new word is in the 'dict'.
- If yes: Add it to the queue (<i>Create an unordered set 'dict' for efficient lookup whether a word exists in the wordList.</i>)
At level 1: we have changed 1 letter from the original word
At level 2: we have changed 2 letters from the original word. . . .</td><td>(N*M^2)
N = wordList.size()
M = size of the longest word</td><td>O(N*M)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td><b>18</b></td><td><b>Reconstruct Itinerary</b></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Given airline tickets, find valid itinerary (use all tickets once). Itinerary must begin with "JFK". If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.
tickets = [["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]
output = ["JFK","MUC","LHR","SFO","SJC"]</td><td>- <b>Topological sorting</b>
- We use unordered_map<string, multiset<string>> to store adjacency list representation of graph as we can have same ticket more than once.
- tickets = [["MUC","LHR"], ["JFK","MUC"], ["SFO","SJC"], ["MUC","LHR"], ["LHR","SFO"]]</td><td>O(E*logV)</td><td>O(E+V)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td><b>19</b></td><td><b>Swim In Rising Water</b></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Given an n x n integer matrix grid where each value grid[i][j] represents the elevation at that point (i, j). The rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distances in zero time. Return the least time until you can reach the bottom right square (n - 1, n - 1) if you start at the top left square (0, 0).</td><td>- Main Idea: <b>At every step find the lowest water level to move forward to</b>
- Use priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq
- To store cells with their elevations and positions, ordered by elevation in ascending order. --> {grid[i][j], i, j}</td><td>O(N^2 * logN)</td><td>O(N^2)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td><b>20</b></td><td><b>Alien Dictionary</b></td><td></td><td></td><td></td></tr>
</table>