My DSA Questions

(With Approach and solution and pattern to identify such questions)

* **Graph**

1. **BFS Traversal of a graph**

**I/P:** Adjacency matrix and no. of vertices

**o/p:** BFS traversed vector

**Approach:**

Take a queue, initially push initial element to queue. After that loop till queue not empty and for each element of queue check adjacent element of it and iterate over them and fallow same procedure, and during this entire procedure keep track of visited node.

**Solution:** <https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1>

**TC:** O (V X 2E), where V=no. of vertices and E=no. of edges and it is 2E because we are traversing adjacent node of each vertex i.e. degree of each vertex (vertex sum) and we know sum of vertex= 2E

**SC:** O(V)

1. **DFS Traversal of a graph**

**I/P:** Adjacency matrix and no. of vertices

**o/p:** DFS traversed vector

**Approach:**

Take initial node and make it visited and from that node Iterate over their adjacent nodes and for each node if it is not visited make recursive call and repeat above process.

**Solution:** <https://practice.geeksforgeeks.org/problems/depth-first-traversal-for-a-graph/1>

**TC:** O (V X 2E), where V=no. of vertices and E=no. of edges and it is 2E because we are traversing adjacent node of each vertex i.e. degree of each vertex (vertex sum)and we know sum of vertex= 2E

**SC:** O(V)

1. **Number of Provinces**

**I/P:** Adjacency matrix and no. of vertices

**o/p:** number of provinces

**Approach:**

Create a 1d visiting array of adjacency matrix size and initialize it with 0. And now iterate over this array for each index with value 0 make a recursive call to another function where we will traverse adjacent nodes for this ith node and make them visited simultaneously and maintaining count of each province.

**Pattern:** As we have to go through direct as well as indirectly connected nodes so DFS is suitable to solve this type of questions.

**Solution:** <https://leetcode.com/problems/number-of-provinces/description/>

**TC:** O(V x V) where V is number of nodes

**SC:** O(V)

1. **Rotten Oranges**

**I/P:** m\*n grid where each cell represents some value,0-empty cell 1-fresh orange and 2-rotten orange.

**o/p:** min. number of minutes required to rotten all oranges in grid if not return -1

**Approach:**

Create a queue of pair<pair<int, int>,int> to store indices and minutes, now iterate over grid and push all rotten oranges into queue. Iterate over queue and for each element in the queue rot adjacent oranges if they aren’t rotten then and push them into queue. And simultaneously check for minimum amount of time to rot all oranges. If any of oranges remains fresh return -1 else return minutes count.

**Pattern:** here, BFS is suitable as in one minute only adjacent oranges will rot.

**Solution:** [https://leetcode.com/problems/rotting-oranges/description/](https://leetcode.com/problems/rotting-oranges/description/%20)

**TC:** O(N\*M) where N is number of rows, m no. of columns.

**SC:** O(N\*M) In worst case queue may contains all cells.

1. **Flood fill**

**I/P:** m\*n grid of image, sr, sc, color

**o/p:** return modified vector after flood fill (starting from image[sr][sc] we have to color their connected pixels with value color recursively)

**Approach:**

Recursively color all connected vertices with value color.

**Pattern:** here, DFS is suitable as we have to color all connected pixels with value color.

**Solution:** <https://leetcode.com/problems/flood-fill/description/>

**TC:** O(N\*M) where N is number of rows, m no. of columns.

**SC:** O(N\*M) In worst case it uses recursion to traverse all cells.