- Question 1. All or None

Divyansh Mathur, 210357

a.) Depth-first traversal aims to ensure that each vertex is visited exactly once by recursively exploring unvisited neighboring vertices.

However, there is essentially no guarantee that each edge is traversed exactly once.

Our particular problem may involve situations where a vertex needs to be revisited and requires traversal of multiple edges connected to the same vertex.

b.) In commonly applied breadth-first traversal, vertices are visited in order of increasing distance from the source vertex, resulting in a BFS tree in which each vertex (except the root) contains exactly one input edge.

will be formed.

Connected.

This inherent property of BFS does not provide a guarantee that he will find a path that covers each edge exactly once.

This is because the problem may require revisiting at least one vertex multiple times to achieve this.

c.) To cross each road in a given diagram exactly once and return to the starting city, two basic conditions must be met: A single connected component: The diagram is One connected component must be formed.
This ensures that a path exists between each pair of cities in the graph, eliminating sections and ensuring access to the entire network.
Even vertex degree: For each city (vertex) in the graph, the number of streets (edges) connected to it must be an even number.
This condition ensures that all roads leading into a city pass through it only once, and that the road returns to the city each time it leaves the city.
In mathematical terms, this means vertices of even degree indicating an even number of visits to the circuit.
These conditions are very useful for setting up a racetrack that can reliably return to the starting city without untraveled roads or isolated components, while still being able to systematically traverse all roads in the graphic.
It is important.

d.) **Algorithm Steps:**

Initialization:

Begin with an empty stack to track vertices during traversal.

Create an empty path to record the order of visited vertices.

Select a Starting Vertex:

Choose any vertex from the graph as the initial current vertex for exploration.

Exploration Loop:

While the current vertex has at least one adjacent neighbor, continue with the following steps:

Step A: Push the Current Vertex:

Push the current vertex onto the stack to maintain traversal order.

Step B: Remove the Edge:

Delete the edge between the current vertex and the encountered adjacent vertex. This ensures that each edge is traversed exactly once.

Step C: Update Current Vertex:

Set the adjacent vertex as the new current vertex for further exploration.

Backtracking:

If the current vertex has no remaining neighbors (all edges have been traversed), perform the following actions:

Step A: Update the Path:

Add the current vertex to the path to document the order of visited vertices.

Step B: Pop from the Stack:

Pop a vertex from the stack, signifying backtracking to explore other untraveled paths if available.

Step C: Set the Popped Vertex as Current:

Assign the popped vertex as the new current vertex to continue exploration.

Repeat the Exploration:

Continue with the exploration, revisiting steps 4 and 5 until the stack becomes empty, and the current vertex no longer has unexplored neighbors.

This algorithm systematically explores the graph, removing edges as they are traversed, ensuring that every edge is visited exactly once while constructing an Eulerian path or circuit. It offers an efficient and structured approach to solving graph traversal problems of this nature.

Question 2. Chaotic Dino

Divyansh Mathur, 210357

a) **Main idea:** The main idea is to use breadth-first search (BFS) to discover vertices starting from vertex S up to a certain distance x.

During this exploration, all towers within that distance will be added to the queue.

Then, BFS is performed by exiting the queue while checking whether the destination vertex D is reached.

Algorithm:
The algorithm includes the following steps:
Initialize a queue named q_tower and list the source vertex S.

Create a visited table to keep track of visited and marked towers The source vertex is visited (visited[S] = true).

Initializing a Boolean variable with an initial value is false.

Execute while loop when queue q_tower is not empty.

Delete the preceding element, denoted t.

Initialize distance changed to 0.

Perform BFS from vertex t to distance x.

For each peak that has an unvisited tower, queue it up and mark it as visited.

If at any time during the BFS journey we reach destination vertex D, set ans to true.

Returns the value of the year.

This algorithm uses BFS to discover towers located within a certain distance from the source vertex and check whether the destination vertex is reachable.

If the destination vertex D is reached during exploration, ans is set to true, indicating a successful path within the given distance.

Pseudocode:

```
bool canSignalReach(int S, int D, int x, vector<vector<int>> graph, vector<bool> towers) {
    queue<int> q_tower; // Initialize a queue to track towers within the distance
    vector<bool> visited(number_of_cities + 1, false); // Array to track visited cities, assuming city indices start from 1
    visited[S] = true; // Mark the source city as visited
    bool ans = false; // Initialize the answer as false
```

```
    q_tower.push(S); // Start the exploration from the source city

    while (!q_tower.empty()) {
        int t = q_tower.front(); // Get the front of the queue
        q_tower.pop(); // Remove the front element
        int distance = 0; // Initialize the distance from the source city

        // Create a queue for BFS from the current city
        queue<int> q_t;
        q_t.push(t);
        while (distance <= x) {
            int size = q_t.size(); // Get the number of vertices at this distance
            while (size > 0) {
                int node = q_t.front(); // Get the front of the BFS queue
                q_t.pop(); // Remove the front element
                for (int neighbor : graph[node]) {
                    if (towers[neighbor] && !visited[neighbor]) {
                        q_tower.push(neighbor); // Enqueue towers within the distance
                        visited[neighbor] = true; // Mark them as visited
                        if (neighbor == D) {
                            ans = true; // Destination city is reached
                        }
                    }
                }
                size--;
            }
            distance++;
        }
    }
    return ans; // Return the answer
}
```

b.)

Algorithm: Finding Valid Power using Binary Search

Input: n (the maximum power), canSignalReach(x) function

Output: ans (the valid power)

1. Initialize right = n-1 and low = 0, as the maximum power can be n-1 in the graph.

2. Define ans = n-1.

3. While low <= high: a. Calculate mid = (low + high) / 2. b. Check if the power at mid is useful using the canSignalReach function. c. If canSignalReach(mid) returns true: - Update high to mid - 1. - Set ans = mid. d. Otherwise: - Update low to mid + 1.

4. Return ans as the valid power.

This algorithm finds the valid power using binary search while ensuring that the meaning remains unchanged.

PseudoCode

```
function findMinimumPower(S, D, graph, towers):
    low = 0
    high = n - 1  // Assuming max power can be n-1 in the graph
    ans = n - 1

    while low <= high:
        mid = (low + high) / 2

        // Check if the signal can reach D with mid power
        if canSignalReach(S, D, mid, graph, towers):  // this function is implemented in 'a' part
            ans = mid
            high = mid - 1
        else:
            low = mid + 1

    return ans
```

Question 3 Room Colors

Divyansh Mathur,210357

Algorithm: Maintain Binary Indexed Trees (BITs) for Room Colors and Update Times

Input:

- n: the number of rooms
- queries: a list of queries of the form (l, r, c), where l and r are room indices, and c is the color to update

Output:

- Answer: an array of size n, containing the final colors of all rooms

Procedure:

1. Calculate N = 2^ceil(log2(n)) to determine the size of the binary tree.
2. Initialize two arrays of size 2*N-1:
    - color (initialize with the original color and dummy nodes)
    - time (initialize to -1)
3. Initialize a variable t to 0 to keep track of the time of updates.
4. For each query (l, r, c) in queries, do the following:
    - Update the color and time of nodes from l to r as follows:
        - Set color[N-1+l] = color[N-1+r] = c
        - Set time[N-1+l] = time[N-1+r] = t
5. For each query (l, r, c), increment t by 1 (i.e., t++).
6. Initialize an Answer array of size n.
7. For each index i from 0 to n-1, do the following:
    - Initialize max_time as time[N-1+i]
    - Set Answer[i] = color[N-1+i]
    - Recursively traverse towards the root:
        - If time[parent] > max_time, update max_time to time[parent and set Answer[i] = color[parent].
8. Return the Answer array as the final colors of all rooms.

Pseudocode:

```
Function initializeBITs(n):
  N = 2^ceil(log2(n))
  Create empty arrays color and time of size 2*N-1
```

```
    Initialize each element of color from N-1 to N-1+n-1 with the original color
    Initialize each element of time to -1

Function update(n, l, r, c, t):
    N = 2^ceil(log2(n))
    u = N - 1 + l
    v = N - 1 + r
    color[u] = c
    color[v] = c
    time[u] = t
    time[v] = t

    while parent(u) != parent(v):
        if u < v:
            if isLeftChild(u):
                color[u + 1] = c
                time[u + 1] = t
            u = parent(u
            if not isLeftChild(v):
                color[v - 1] = c
                time[v - 1] = t
            v = parent(v
    Increment t by 1 (t++)

Function query(n):
    N = 2^ceil(log2(n))
    Create an empty array Answer of size n

    For i from 0 to n-1:
        max_time = time[N - 1 + i]
        Answer[i] = color[N - 1 + i]

        node = N - 1 + i
        while node > 0:
            parent = parent(node)
            if time[parent] > max_time:
```

```
            max_time = time[parent]
            Answer[i] = color[parent]
        node = parent

return Answer
```

Question 4. Fest Fever

Divyansh Mathur, 210357

Given the money M an n queries (updates + requests) in chronological order, To give an O(n log n) time algorithm to that outputs ”YES” if a request can be fulfilled and ”NO” if it cannot.

We can use the data structure Binary indexed tree as discussed in lectures.

Algorithm:

Set N = 2^ceil(log2(n)) to create a complete binary tree.

Declare an array Seg_Tree of size 2N - 1.

The new array stores the object sequence given in Seg_Tree[N-1] in Seg_Tree[N-1+n-1] as follows:

Seg_Tree[N-1+i]= price [i] from i=0 to n-1. The remaining nodes from Seg_Tree[N-1+n] to Seg_Tree[2N-2] are marked as 0 (dummy nodes).

Preprocessing :

- Starting from index N-1, traverse the tree recursively to its parents ((i-1)/2) and update the value of the parent by adding its children values.

Request Query:

- For query function initialize sum variable S = 0 .
- Define a = Seg_Tree[N-1+l] and b = Seg_Tree[N-1+r] and add their values to S.
- Return to the parent nodes recursively:
  - If a is the left child of that parent, add the value of the right sibling of u to S.
  - Similarly, if b is the right child of its parents, add the value of its left sibling to S.
- Continue this process until their Lowest Common Ancestor is found.
- See S <= M. If true, purchase is allowed; Otherwise, it is not.

Update Query

- For update action on element i, save u = Seg_Tree[N-1+i].
- Update Seg_Tree[N-1+i] to new value x.
- Recursively update its parents by returning ((N-2+i)/2) and adding (x - u) to it.

Pseudocode:

```
// Initialize Seg_tree array of size= 2^ciel(log(n))
N = pow(2, ceil(log2(n)));
for (int i = 0; i < N; i++) {
   Seg_tree[N - 1 + i] = sequence[i];  // Store the items in Seg_tree
}
```

```
// Preprocess the other elements of Seg_tree
for (int i = N - 2; i >= 0; i--) {
    Seg_tree[i] = Seg_tree[2 * i + 1] + Seg_tree[2 * i + 2];  // Update each node with the sum of its children
}
// Function to process a request query
string process_request(int l, int r, int M) {
    S = 0;
    a = N - 1 + l;  // Set 'u' to the index of item 'l' in Seg_tree
    b = N - 1 + r;  // Set 'v' to the index of item 'r' in Seg_tree
    S += Seg_tree[u];   // Add the price of 'a' to the sum
    S += Seg_tree[v];   // Add the price of 'b' to the sum

    while (a != b) {  // Traverse to their Lowest Common Ancestor (LCA)
        if (a < b) {
            if (a % 2 != 0) {  // If a is a left child, add its right sibling to the sum
                S += Seg_tree[a + 1];
            }
            a = (a - 1) / 2;  // Move 'a' to its parent
        } else {
            if (b % 2 != 0) {  // If 'b' is a right child, add its left sibling to the sum
                S += Seg_tree[b - 1];
            }
            b = (b - 1) / 2;  // Move 'b' to its parent
        }
    }

    if (S <= M) {
        return "YES";  // If the sum is less than or equal to 'M', return "YES"
    } else {
        return "NO";   // Otherwise, return "NO"
    }
}
```

```
// Function to update an item
void update_item(int i, int x) {
   old = Seg_tree[N - 1 + i];  // Store the old price of item 'i'
   Seg_tree[N - 1 + i] = x;       // Update the price of item 'i' to 'x'

   i = (N - 1 + i - 1) / 2;  // Move to the parent of item 'i'
   while (i >= 0) {         // Update all ancestors of the modified node
     parent = (i - 1) / 2;
      Seg_tree[parent] += (x - old);  // Update the parent node
      i = parent;                  // Move to the next parent
   }
}
```

Time complexity analysis:

Typical binary indexed tree takes logn time in updating/ requesting a query and since the total queries are also n.

Overall time complexity = O(nlogn)

Question 5 Edible Sequence

Divyansh Mathur,210357

To check whether a sequence of given tree structure is a valid BFS or not.

We know when we start BFS, we maintain a queue data structure and for a particular iteration we add all the child into the queue once done we pop the element.

Using the above criterion we can devise the following algorithm.

Algorithm:

Maintain an array no_of_child[n] such that no_of_child[i] has the no. of children of index i.

Maintain another array parent[n] such that parent[i] has the parent of index i.

And Let the given sequence be arr[n].

Maintain a queue Q and apply the following operations:

for a element in the sequence, if its parent is at the front of Q then ok, decrease no_of_child[front] and push the element into the Q, now if the no_of_child[front] is zero then pop the front.

If the parent is not at front then simply print not edible and return

At the end of all iterations if we reach end of loop print edible.

Pseudocode:

```
Let the tree is given in form of adjacency list.
//no_of_child array
for i from 0 to n-1
no_of_child[i]=adj[i].size()-1; //-1 becuase one neighbour is parent


//parent array { run a dry BFS}
intitalise empty queue P and panrent[n] array as -1 (={-1}).
parent [0]=0;
while(q is not empty){
   s=q.size();
   node= front(q);
   pop(q);
   for i from 0 to s{
      for(all elements 'ent' of adj[node]){
         if(parent[ent]==-1) {
            parent[ent]=node;
         }
      }
   }
```

```
}

//final algo
Q.push(0)
if(arr[0]!=0) {
   print "not edible"
   return
}
for i from 1 to n-1{
   if(parent[arr[i]]!=Q.front){
      print "not edible"
      return
   }else{
      Q.push(arr[i])
      no_of_child[Q.front]--;
      if(no_of_child[Q.front]==0) {
         Q.pop
      }
   }
}
print "edible"
```

Time Complexity Analysis:

Filling the array no_of_child takes O(n) time

Filling the parent array needs BFS hence O(V+E), given the no. of vertices is n and given the graph is tree E=n-1 hence O(2n-1)

Again we run a BFS-type-algorithm where we visit each node viewing each edge hence again O(2n-1)

Hence overall time complexity =O(n)