Theoretical Assignment 3

Dhruv Gupta (220361)

October 2023

Dhruv Gupta: 220361

Question 1

We want to travel through all the edges of a graph exactly once, visiting all the vertices atleast once, and return to the starting vertex. Such a route on a graph is called an **Eulerian cycle**.

(a) No, because two graphs can have same DFS traversal, even when one of them has an Eulerian cycle, while other does not. For example, in figure 1, graph G_1 does not have an Eulerian cycle, while graph G_2, has an Eulerian cycle, (0,1,2,0). But both have same DFS traversal (0,1,2), starting from 0. So if we are given a DFS traversal (0,1,2), we can not tell if the graph is G_2 or G_1, or if it has an Eulerian cycle or not.

Figure 1: Left: Graph G_1, Right: Graph G_2

(b) No, because two graphs can have same BFS traversal, even when one of them has an Eulerian cycle, while other does not. For example, in figure 1, graph G_1 does not have an Eulerian cycle, while graph G_2, has an Eulerian cycle, (0,1,2,0). But both have same BFS traversal (0,1,2), starting from 0. So if we are given a BFS traversal (0,1,2), we can not tell if the graph is G_2 or G_1, or if it has an Eulerian cycle or not.

Figure 2: Left: Graph G_1, Right: Graph G_2

(c) The graph must be connected and all the vertices must have even degree, if we want to traverse each road (edge) exactly once and return to the starting city (vertex).

(d) Assuming that a graph $G = (V, E)$ has an Eulerian cycle, and all the vertices are labeled from 0 to $n-1$, ($|V| = n$). We use the following algorithm to print the required path (cycle)–

- Since our ans is a cycle, we can take any vertex as the starting point. Without loss of generality, let us take $v = 0$ as the starting vertex. We will maintain a list L to store our ans.
- We will create an empty stack S, and push the starting vertex v in S.
- We look at the top element of the stack. If its degree is 0, we pop it from the stack, and add it in the answer (i.e., insert it in L at the end). If its degree is not 0, (i.e., it has atleast one edge) we delete any one edge from G, and push the vertex at the other end of the edge in S.
- We repeat the previous step until the stack S is not empty and stop when it becomes empty.
- At the end of this, our list L will have the sequence of vertices which represent the path.

To access the degree of any vertex in constant time, we can create a $degree[0, ..., n-1]$ array. We can iterate once through all the edges in the graph, and calculate the degrees in $O(E)$ time.
To delete any edge from the graph in $O(1)$ time, we can use a hash map.

Pseudo code for creating $degree[0, ..., n-1]$ array:

```
for(i = 0 to n − 1){ degree[i] ← 0; }
for each edge (u, v) in G{ degree[u] + +; degree[v] + +; }
```

Pseudo code for finding Eulerian cycle in graph G:

```
FindECycle(G, degree[0, ..., n − 1]){
    CreateEmptyStack S;
    CreateEmptyList L;
    push(0,S);
    while(not isEmpty(S)){
        x ←Top(S);
        if(degree[x] = 0){ Insert(x, L); pop(S); }
        else{
        // Let u be any neighbour of x.
// We can find u in O(1) time from the adjacency list representation of G.
            u ← neighbour of x;
            delete((x, u), G); // Delete the edge (x, u) from G.
            degree[u] − −; degree[v] − −;
```

```
            push(u, S);
        }
    }
    return L;
}
```

Time Complexity Analysis:

To create the *degree* array, We need one function call of **FindECycle**$(G,\ degree[0, ..., n-1])$. It consists of a while loop and rest of the operations take $O(1)$ time. Each iteration of the while loop also consists of $O(1)$ time operations, hence takes constant time. So we only need to find the maximum possible iterations of the while loop.

Now, note that, in each iteration, either (i) an element is removed from the stack or (ii) an edge is removed with addition of an element in the stack.

Suppose we have $|E| = m$ edges. Then, the loop will go in condition (i) m times. Since the stack initially has only 1 element, and elements are only added via condition (ii), there can be atmost $m+1$ elements in the stack. Also our ans must consist of $m+1$ vertices, since except the starting element, all elements have exactly 1 associated incoming edge, and there are total m edges. Hence, condition (i) will occur $m+1$ times. So, the while loop will iterate (at most) $2m+1$ times. Hence, total time complexity $= O(m) = O(|E|)$.

Dhruv Gupta: 220361

Question 2

(a) We assume that we have n cities labeled $1, ..., n$, and we are given an array $tower[1, ..., n]$, where for any city x, $tower[x] = 1$ iff x has a tower, and 0 otherwise. Without loss of generality, let us assume that city S is labelled 1, and city D is labelled n. We will maintain an array $destroyed[1, ..., n]$, to keep track of whether a city is destroyed or not. All the entries are initialised to 0 (signifying that no city is destroyed initially). If city i is destroyed at any point of time, we set $destroyed[i] = 1$. We will also use a queue *Active*, to keep track of the activated towers. We will use BFS to find distances between cities.
We use the following algorithm to check whether D can be destroyed or not-

- First apply BFS from city S. We will modify the BFS such that, whenever we visit a city, we will check if its distance is less than or equal to x. If yes, we will destroy that city (if not already destroyed). Also, if this city has a tower (and is not already destroyed), we will enqueue it to *Active*.
- After this we will dequeue the first element in Q, and apply BFS from this city, same as we applied from S in the previous step (Note that for doing so we will need to re-initialise our *visited* array to 0). We will update the destroyed array same as before, and also enqueue the cities with tower which are destroyed for the first time.
- Basically, we are activating a tower whenever we reach one, and destroy all the cities which can be destroyed.
- We will keep doing this until the queue *Active* becomes empty.
- At this point we will check if city D is destroyed or not.

In Pseudo code, we will assume the graph (adjacency list), *distance* array, *visited* array, and *tower* array used in BFS to be defined globally.

Pseudo code for modified BFS used in above algorithm:

```
modifiedBFS(p, destroyed[1, ..., n], x, Active){
    for (i = 1 to n){ visited[i] ← 0; }
    CreateEmptyQueue Q;
    distance[p] ← 0;
    Enqueue(p, Q); visited[p] ← 1;
    while(not isEmpty(Q)){
        v ← Dequeue(Q);
        for each neighbour w of v{
            if(visited[w] = 0){
                distance[w] ← distance[v] + 1;
                visited[w] ← 1;
                Enqueue(w, Q);
```

```
                if(distance[w] ≤ x and destroyed[w] = 0){
                    if(tower[w] = 1){ Enqueue(w, Active); }
                    destroyed[w] = 1;
                }
            }
        }
    }
}
```

Pseudo code for above algorithm:

```
// Returns 1 if D is destroyed, returns 0 otherwise.
Destruction(x){
    for (i = 1 to n){ destroyed[i] ← 0; }
// Creating and initialising destroy[ ] to 0.
    CreateEmptyQueue Active;
// Create (declare) an empty queue Active.
    Enqueue(1, Active);
// Enqueue S first, so that BFS is applied from S first.
    while(not isEmpty(Active)){
        p ← Dequeue(Active);
        modifiedBFS(p, tower[1, ..., n], destroyed[1, ..., n], x, Active);
    }
    flag ← 0;
    if(destroyed[n] = 1){ flag ← 1; }
    return flag;
}
```

Time Complexity Analysis:

In **modifiedBFS()**, we have only added a few $O(1)$ time operations in the (innermost loop) for loop of the normal BFS, which anyways takes $O(1)$ time in each iteration. Therefore, time complexity of **modifiedBFS()** is same as the normal BFS, which is $O(n + m)$, where $n = number\ of\ vertices$ ($cities$) and $m = number\ of\ edges$ in the graph.
We need exactly one function call of **Destruction(x)**. It consists of some $O(1)$ time operations and a **while** loop. Since we enqueue and dequeue each city having a tower exactly once, this **while** loop will have total t iterations, where $t = number\ of\ towers$. Each while loop has an $O(1)$ time operation (dequeue), and one function call of **modifiedBFS()**.
Therefore total time complexity is $O(t*(n+m)) \leq O(n*(n+m))$ (since $t \leq n$).

(b) We know for sure, if $x = 0$, D cannot be destroyed (assuming S and D are distinct). Also, any path from S to D, in any graph can have atmost n length. Therefore, if $x \geq n - 1$, D can be surely destroyed. Hence, minimum value of x, $min\_x \in (0, n - 1]$. So we can apply Binary Search type algorithm on the value of x in the range $(0, n - 1]$. If at any point we have L and R as the lower and upper bounds respectively, and we have, **Destruction(*mid*)**$= 1$,

then, $min\_x \in (L, mid]$, other wise $min\_x \in (mid, R]$.

Pseudo code for above algorithm:

```
// Returns the minimum power of tower required to destroy D.
FindMinX(n){
    L ← 0; R ← n − 1;
    ans ← n − 1;
// We will store the latest value of mid, for which D can be destroyed.
    while(L ≤ R){
        mid ← (L + R)/2;
        if(Destruction(mid) = 1){ R ← mid − 1; ans ← mid; }
        else{ L ← mid + 1; }
    }
//After this loop ends, ans will store the minimum value of power required.
    return ans;
}
```

Time Complexity Analysis:

We need just one function call of **FindMinX(**n**)**. The while loop in the function will run for $logn$ iterations, since we are halving the search area in each iteration, in the same way as we did in standard Binary Search. Each iteration has a few $O(1)$ time operations and one call of **Destruction(**x**)**. Hence each iteration takes $O(n * (n + m))$ time. And thus the while loop will take $O(n * (n + m) * logn)$ time. Rest all opertaions in **FindMinX(**n**)** take $O(1)$ time. Therefore, overall time complexity is $O(n * (n + m) * logn)$.

Dhruv Gupta: 220361

Question 3

We are assuming that we are given the initial colour of each room as an array $colour[1, ..., n]$. We create a data structure, a complete binary tree to solve the given problem. The implementation of the complete binary tree will be done using an array $A[\,]$(same as done in lectures). Let us assume that each colour is represented by a non-negative integer.

- First, we need n to be a power of 2. If it is not we find the smallest number p, which is a power of 2 and is greater than n, i.e., $n \leq p = 2^k$, for some k. For this we can take $k = \lceil logn \rceil$. Clearly, we have $n \leq p \leq 2n$. Our complete binary tree will have p leaf nodes.

- Create an array $A[0, ..., N-1]$ and an array $C[0, ..., N-1]$ of size $N = 2p-1$.

- We will also maintain an array $C[0, ..., N-1]$, such that $C[i]$ will store the bombing number at which the value of index i was changed for the last time. For example, if $A[5]$ was updated at 1^{st}, 2^{nd}, 7^{th}, and 11^{th} bombings, then $C[5]$ will store value 11. We will need both arrays $A[\,]$ and $C[\,]$ to determine the colour of each room after all the bombings are completed.

- We populate our complete binary tree as follows. All the leaf nodes will store the colour of each room. The first n leaf nodes will thus take values from $colour[1, ...n]$, and the rest $p-n$ leaf nodes will have a dummy value -1. Every internal node will also have a dummy value -1.

- For implementing this in $A[\,]$, we first initialise all the elements of with value -1. Then we update the values of indices $p-1$ to $p+n-2$ (both included), (since these are the first n leaf nodes), with corresponding values from the $colour[\,]$ array.
  Similarly, for initialising $C[\,]$, we first initialise all the elements with value -1. Then we update the values of indices $p-1$ to $p+n-2$ (both included), (since these are the first n leaf nodes), with value 0. (-1 means that the node was never given a colour, and 0 means that the node was given a colour initially.)

For each bombing we update the tree (i.e., the array $A[\,]$) as follows-

- We need to update the colours of l^{th} room to r^{th} room to c. These values are in our complete binary tree from l^{th} leaf node to r^{th} leaf node which are stored in $A[\,]$ at indices $p+l-2$ to $p+r-2$.

- We start at the two leaf nodes corresponding to the l^{th} and r^{th} room. We change the colour of these nodes to c.

- Now, if the left node, is the left child of its parent, we change the colour of its sibling to c. Similarly, if the right node is the right child of its parent, we change the colour of its sibling to c. Also whenever we change the colour of a node, we update the corresponding index in $C[\,]$ with the current bombing number.
- Then we move to their parents. Now the parent of the left node, becomes the left node, and parent of the right node becomes the right node. keep repeating the previous step until both left and right nodes have the same parent.
- For implementing this in $A[\,]$, we use the following property, that if node (index) x is odd, it must be the left child of its parent, and if it is even, it must be the right child of its parent. Also the parent of x will be the node (index) $\lfloor \frac{x-1}{2} \rfloor$.

After all the bombings are complete, we determine the colour of each room as follows-

- For determining the final colour of a given room, we start at its corresponding leaf node in the complete binary tree.
- We compare the latest bombing numbers of the node and its parent from $C[\,]$. We maintain a variable *temp* to store the colour of the node with greater bombing number in $C[\,]$.
- We then move to the parent and repeat the previous step, until we reach a child of the root (since root has no parent).
- The value of *temp* after this will store the final colour of the node after all the bombings are completed.
- We update the colour of the room in the $colour[\,]$ array to *temp*.
- $colour[\,]$ array will then contain the final colours of each room.

Pseudo code for populating the array $A[0, ..., N-1]$ and array $C[0, ...N-1]$:

Populate($A[0, ..., N-1]$, $colour[1, ..., n]$, $C[0, ..., N-1]$**)**{
// $N = 2p - 1$, means that $p = (N+1)/2$.
$p \leftarrow (N+1)/2$;
for $(i = 0$ **to** $N-1)$\{ $A[i] \leftarrow -1$; $C[i] \leftarrow -1$; \}
for $(i = 1$ **to** $n)$\{ $A[p+i-2] \leftarrow colour[i]$; $C[p+i-2] \leftarrow 0$; \}
}
}

Pseudo code for updating the array $A[0, ..., N-1]$ and array $C[0, ...N-1]$:
(for a given bombing k given by (l, r, c))

Bomb($A[0, ..., N-1]$, $C[0, ..., N-1]$, k, l, r, c**)**{
// $N = 2p - 1$, means that $p = (N+1)/2$.
$p \leftarrow (N+1)/2$;

```
        i ← l;
        j ← r;
        par_i ← (i − 1)/2; //Assuming integer division.
        par_j ← (j − 1)/2; //Assuming integer division.
        A[i] ← c; C[i] ← k;
        A[j] ← c; C[j] ← k;
        while(par_i ≠ par_j){
            if (i is odd){ A[i + 1] ← c; C[i + 1] ← k; }
        //If i is the left child, its sibling is at i + 1.
            if (j is even){ A[j − 1] ← c; C[j − 1] ← k; }
        //If j is the right child, its sibling is at j − 1.
            i ← par_i;
            j ← par_j;
        }
    }
```

Pseudo code for updating the colour of given room x:
(after all the bombings in the *colour*[ ] array)

```
    Update(A[0, ..., N − 1], C[0, ..., N − 1], colour[1, ..., n], x){
    // N = 2p − 1, means that p = (N + 1)/2.
    // x − th room corresponds to index p + x − 2 in A[ ], where p = (N + 1)/2.
        p ← (N + 1)/2;
        i ← p + x − 2;
        par ← (i − 1)/2; //Assuming integer division.
        while(i > 0){
            if(C[i] > C[par]){ temp ← A[i]; }
            else{ temp ← A[par]; }
        }
        colour[x] ← temp;
    }
```

Pseudo code for updating the colour of all rooms after all bombings:

```
    UpdateAll(colour[1, ..., n], m){
        p ← 1;
        while(p < n){ p ← p ∗ 2; }
        N ← 2 ∗ p − 1;
        A[0, ..., N − 1], C[0, ..., N − 1]; // Creating an arrays of A[ ] and C[ ]
        Populate(A[0, ..., N − 1], colour[1, ..., n], C[0, ..., N − 1]);
        k ← 1; // k stores the bombing number, initially k = 1 (for first bombing).
        for each bombing (l, r, c){
            Bomb(A[0, ..., N − 1], C[0, ..., N − 1], k, l, r, c);
            k++;
        }
        for (i = 1 to n){
            Update(A[0, ..., N − 1], C[0, ..., N − 1], colour[1, ..., n], i);
        }
    }
```

Note that, we are assuming bombings to be globally defined, i.e., all functions can use them, but don't need them as parameters.

Time Complexity Analysis:

Firstly, note that since $n \leq p \leq 2n$ and $N = 2p - 1$, $O(p) = O(N) = O(n)$. Also, $O(log(p)) = O(logN) = O(logn)$.
Now, let us look at the time complexities of the three functions **Populate**, **Bomb** and **Update**-

- **Populate(**$A[0, ..., N-1]$, $colour[1, ..., n]$, $C[0, ..., N-1]$**)**: Each function call consists of two **for** loops and some $O(1)$ time operations. First **for** loop has N iterations, and each iteration takes $O(1)$ time, so it takes total $O(N) = O(n)$ time. Second **for** loop has n iterations, and each iteration takes $O(1)$, so it takes $O(n)$ time. Therefore, total time taken in function call is $O(n) + O(n) = O(n)$.

- **Bomb(**$A[0, ..., N-1]$, $C[0, ..., N-1]$, k, l, r, c**)**: Each function call consists of some $O(1)$ time operations and a while loop. Through this while loop, we start from two nodes l and r, and move up the tree until we reach two nodes having the same parent. Hence, at max we can move upto the root. So, the while loop will have maximum $O(height) = O(logN) = O(logn)$ iterations. Also, each iteration of the while loop consists of $O(1)$ time operations. Therefore, time taken in one function call is $O(logn)$.

- **Update(**$A[0, ..., N-1]$, $C[0, ..., N-1]$, $colour[1, ..., n], x$**)**: Each function call consists of some $O(1)$ time operations and a while loop. Through this while loop, we start from the leaf node in our complete binary tree and go to the root taking one step in a time (from node to its parent). Since we have a complete binary tree, there will be $O(height) = O(logN)$ steps, where N is the total number of nodes in the tree. At each node (i.e., in each iteration of the loop) we perform some constant time operations. Therefore, time taken by the for loop $= O(logN) = O(logn)$. Therefore, time taken in one function call is $O(logn)$.

Now, we need to exactly one function call of **UpdateAll(**$colour[1, ..., n], m$**)**. In this function call- The while loop which calculates p runs for $logn$ iterations, and each iteration takes $O(1)$ time, hence it takes $O(logn)$ time.
We then have one function call of
Populate($A[0, ..., N-1]$, $colour[1, ..., n]$, $C[0, ..., N-1]$**)**. This will take $O(n)$ time.
Then we have a for loop which executes the bombings. This loop goes for m iterations since there are a total of m bombings. Each iterations calls **Bomb(**$A[0, ..., N-1]$, $C[0, ..., N-1]$, k, l, r, c**)** function once, this takes $O(logn)$ time. Rest operations take $O(1)$ time. Hence this for loop takes a total of $O(mlogn)$ time.
At last we have another for loop which goes for n iterations and in each iteration

calls **Update(**$A[0,...,N-1]$, $C[0,...,N-1]$, $colour[1,...,n], x$**)** function once. So each iteration takes $O(logn)$ time, and hence this for loop takes $O(nlogn)$ time.
All other operations take $O(1)$ time.
Hence, total time taken in one function call of **UpdateAll(**$colour[1,...,n], m$**)** $= O(logn) + O(n) + O(mlogn) + O(nlogn) + O(1) = O((m+n)logn)$.

Dhruv Gupta: 220361

Question 4

We are assuming that we are given the initial price of each sweet as an array $price[1, ..., n]$. We create a data structure, a complete binary tree to solve the given problem. The implementation of the complete binary tree will be done using an array $A[\,]$ (same as done in lectures).

- First, we need n to be a power of 2. If it is not we find the smallest number p, which is a power of 2 and is greater than n, i.e., $n \leq p = 2^k$, for some k. For this we can take $k = \lceil logn \rceil$. Clearly, we have $n \leq p \leq 2n$. Our complete binary tree will have p leaf nodes.
- Create an array $A[0, ..., N-1]$ of size $N = 2p - 1$.
- We populate our complete binary tree as follows. All the leaf nodes will store the price of each sweet. The first n leaf nodes will thus take values from $price[1, ...n]$, and the rest $p - n$ leaf nodes will have a dummy value 0. Every internal nodes will store the sum of the values of its children.
- For populating leaf nodes, in $A[\,]$ we store the values from $price[\,]$ at indices, $p - 1$ to $p + n - 2$ (both included), since these are the first n leaf nodes. And we set value to be 0 for indices $p + n - 1$ to $N - 1$ (both included).
- For populating internal nodes, we do the following. Start from the end of the array $A[\,]$ and move towards the front upto the index 1 (i.e., just before the root node, because, root does not have a parent). For each node, we increment its parent by the value of that node. So, for any index x, we increment value at index $\lfloor \frac{x-1}{2} \rfloor$, by $A[x]$. Since, every parent has a lower index than its children, by the time we reach any internal node, that node would correctly store the sum of the values of its children.

For each update query, we do the following-

- We start from the leaf node, update its value, then move to its parent, update the value of parent so that it now correctly stores the sum of the values of its children. Then repeat the above process, i.e., move to the parent of the parent, update its value accordingly, then move to its parent, and so on, until we reach the root.

For, each request query, we do the following-

- We need to find the sum of prices of l^{th} sweet to r^{th} sweet. These values are in our complete binary tree from l^{th} leaf node to r^{th} leaf node which are stored in $A[\,]$ at indices $p + l - 2$ to $p + r - 2$.
- For calculating this sum, we use our complete binary tree, and apply a similar algorithm as we used in lectures for the range minima problem.

- We create a variable sum, and set it to 0 initially, we add the values of the two starting leaf nodes to sum.
- Now, if the left node, is the left child of its parent, we add the value of its sibling to sum. Similarly, if the right node is the right child of its parent, we add the value of its sibling to sum.
- Then we move to their parents. Now the parent of the left node, becomes the left node, and parent of the right node becomes the right node. keep repeating the previous step until both left and right nodes have the same parent.
- For implementing this in $A[\,]$, we use the following property, that if node (index) x is odd, it must be the left child of its parent, and if it is even, it must be the right child of its parent. Also the parent of x will be the node (index) $\lfloor \frac{x-1}{2} \rfloor$.

Pseudo code for populating the array $A[0, ..., N-1]$:

```
Populate(A[0, ..., N-1], price[1, ..., n]){
// N = 2p - 1, means that p = (N + 1)/2.
    p ← (N + 1)/2;
    for (i = 1 to n){ A[p + i - 2] ← price[i]; }
    for (i = p + n - 1 to N - 1){ A[i] ← 0; }
    for (i = N - 1 to 1){
        par ← (x - 1)/2; //Assuming integer division.
        A[par] ← A[par] + A[i];
    }
}
```

Pseudo code for update query:

```
Update(A[0, ..., N-1], k, val){
// We are updating the price of k - th sweet to val.
// k - th sweet corresponds to index p + k - 2 in A[ ], where p = (N + 1)/2.
    p ← (N + 1)/2;
    i ← p + k - 2;
    while(i > 0){
        par ← (x - 1)/2; //Assuming integer division.
        A[par] ← A[par] + val - A[i];
    // If we change the value from A[i] to val,
    // value of parent increases by val - A[i].
    }
}
```

Pseudo code for request query:

```
Request(A[0, ..., N-1], l, r){
    p ← (N + 1)/2;
    sum ← 0;
    i ← l;
```

$j \leftarrow r$;
$par\_i \leftarrow (i-1)/2$; //Assuming integer division.
$par\_j \leftarrow (j-1)/2$; //Assuming integer division.
$sum \leftarrow sum + A[i] + A[j]$;
while$(par\_i \neq par\_j)\{$
if $(i$ *is odd*$)\{$ $sum \leftarrow sum + A[i+1]$; $\}$
//If i is the left child, its sibling is at $i+1$.
if $(j$ *is even*$)\{$ $sum \leftarrow sum + A[j-1]$; $\}$
//If j is the right child, its sibling is at $j-1$.
$i \leftarrow par\_i$;
$j \leftarrow par\_j$;
$\}$
return sum;
$\}$

Final Code:
(To calculate p, N for a given n and print output, making use of above defined functions)

//Assuming we are given array $price[1, ..., n]$ and M.
$p \leftarrow 1$;
while$(p < n)\{$ $p \leftarrow p * 2$; $\}$
$N \leftarrow 2 * p - 1$;
$A[0, ..., N-1]$; // Creating an empty array of size N
Populate($A[0, ..., N-1]$, $price[1, ..., n]$**)**;
//Let us assume each update query is given as $1, k, val$.
//Let us assume each request query is given as $2, l, r$.
//1 and 2 signify the type of query.
for each query$\{$
if(query is $1, k, val)\{$ **Update(**$A[0, ..., N-1]$, k, val**)**; $\}$
if(query is $2, l, r)\{$
$sum \leftarrow$ **Request(**$A[0, ..., N-1]$, l, r**)**;
if$(sum \leq M)\{$ **print**("YES"); $\}$
else$\{$ **print**("NO"); $\}$
$\}$
$\}$

Time Complexity Analysis:
Firstly, note that since $n \leq p \leq 2n$ and $N = 2p - 1$, $O(p) = O(N) = O(n)$. Also, $O(log(p)) = O(logN) = O(logn)$.
Now, let us look at the time complexities of the three functions defined before the Final Code.

- **Populate(**$A[0, ..., N-1]$, $price[1, ..., n]$**)**: Each function call consists of some constant ($O(1)$) time operations and three **for** loops. Each for loop consists of $O(1)$ time operations in each iteration. First for loop has n iterations, second has $p-n$ iterations, third has $N-1$ iterations. Total time taken in one function call is therefore $O(1)+O(n)+O(p-n)+O(N-1) = O(n)$.

- **Update(**$A[0, ..., N-1],\ k,\ val$**)**: Each function call consists of some $O(1)$ time operations and a for loop. Through this for loop, we start from the leaf node in our complete binary tree and go to the root taking one step in a time (from node to its parent). Since we have a complete binary tree, there will be $O(height) = O(logN)$ steps, where N is the total number of nodes in the tree. At each node (i.e., in each iteration of the loop) we perform some constant time operations. Therefore, time taken by the for loop $= O(logN) = O(logn)$. Therefore, time taken in one function call is $O(logn)$.

- **Request(**$A[0, ..., N-1],\ l,\ r$**)**: Each function call consists of some $O(1)$ time operations and a while loop. Through this while loop, we start from two nodes l and r, and move up the tree until we reach two nodes having the same parent. Hence, at max we can move upto the root. So, while loop will have maximum $O(height) = O(logN) = O(logn)$ iterations. Also, each iteration of the while loop consists of $O(1)$ time operations. Therefore, time taken in one function call is $O(logn)$.

In the final code, the while loop which calculates p runs for $logn$ iterations, and each iteration takes $O(1)$ time, hence it takes $O(logn)$ time.
One function call of **Populate(**$A[0, ..., N-1],\ price[1, ..., n]$**)** takes $O(n)$ time. The for loop which prints output for each query, runs for n iterations, since there are total n queries. Each iterations consists of either one function call of **Update(**$A[0, ..., N-1],\ k,\ val$**)** or one function call of **Request(**$A[0, ..., N-1],\ l,\ r$**)** and some $O(1)$ time operations. Hence each iteration will take $O(logn)$ time. Therefore, this for loop takes $O(nlogn)$ time.
Hence overall time complexity $= O(1) + O(n) + O(nlogn) = O(nlogn)$.

Dhruv Gupta: 220361

Question 5

For a given sequence of nodes of a tree to be in a valid BFS order, following two conditions must be met-
(1) The parent of every node must occur before the node itself.
(2) If a node a appears before node b in the sequence, then $parent(a)$ must also appear before $parent(b)$ in the sequence or in other words, if a node x appears before node y, all the children of x must come before all the children of y in the sequence.
(Whenever we say that a comes before b, we mean that index of a is less than index of b in the given sequence).
We know that BFS traversal algorithm traverses the nodes in a level wise order, i.e., in a non-decreasing order of distances. Therefore, for condition (1) to be true, we just need to check if the sequence has nodes in non-decreasing order of distance from the root. This is because if $v = parent(u)$ (v is not the root), $distance(v) < distance(u)$, hence if we ensure that the sequence has nodes in non-decreasing order of distance from the root, it is also ensured that parent of every node comes before the node in the sequence.
Conversely, if the non-decreasing order breaks at any point, we will get a node which appears before its parent.
Also, in a valid BFS order, the difference between distances pf two consecutive elements must not be greater than 1.
Whenever we visit a node x while applying BFS, we enqueue all the children of x. So if x comes before y in the BFS order, all the children of x must come before all the children of y. Furthermore, all the children of any particular node, will appear contiguously (together) in the sequence, in some order depending upon the adjacency list.

Pseudo code for checking condition (1):

```
Check_1(A[1,...,n], distance[1,...,n]){
//A[1,...,n] is the given sequence.
//distance array is obtained after applying BFS.
    flag ← 1;
    for (i = 1 to n − 1){
        if (distance[A[i]] > distance[A[i + 1]] or
          distance[A[i]] < distance[A[i + 1]] − 1){
            flag ← 0;
            break;
        }
    }
    return flag;
}
```

Algorithm for checking condition (2):

We create an array $par\_A[1, ..., n]$, which stores the parent of each node in the given sequence. For condition (2) to be true, the elements in par_A, must appear in the same order in A (ignoring the repetitions).
We assume that the sequence is a valid BFS order, and set the $flag$ to 1. Whenever we find that it is not a valid BFS order, we change the $flag$ to 0.
We take two pointers i and j to point at the arrays A, and par_A respectively. We compare the values of $A[i]$ and $par\_A[j]$.
If they are equal, it means that the element at index i is the parent of element at index j in the sequence. In this case we increment j because $A[i]$ can have multiple children.
If the values are not equal, it means that $A[i]$ is not the parent of $A[j]$ and we increment i assuming that we have exhausted all the children of $A[i]$ and so we should move to the next node.
This way, if any node a comes before node b, but children of b appear before that of a, node a will be skipped. And later when pointer j reaches at the index of children of a, pointer i will keep on increasing, until it reaches the end.
On the other hand, in a valid BFS ordering, i will reach only till the last internal node, after which j will keep on increasing because till the end, because par_A will not contain any leaf node and the last element in the sequence must have the last internal node as its parent.
So if i reaches the end first, then sequence is not a valid BFS order, and we change $flag$ to 0.
Else, if j reaches the end first, then sequence is a valid BFS order, and we let the $flag$ to be 1 (unchanged).

Pseudo code for checking condition (2):

Check_2($A[1, ..., n], parent[1, ..., n]$**)**{
//A[1,...,n] is the given sequence.
//parent array is obtained after applying BFS. We have assumed parent of root to be the root itself.
//Creating array $par\_A[1, ..., n]$, which stores the parent of each node in the given sequence.
    for $(i = 1$ **to** $n)$**{**
        $par\_A[i] \leftarrow parent[A[i]]$;
    }
    $flag \leftarrow 1$;
    $i \leftarrow 1$; $j \leftarrow 1$;
    while$(i \leq n$ **and** $j \leq n)$**{**
        if $(par\_A[j] = A[i])${ j++ };
        else{ i++ };
    }
    if $(i = n + 1)${ $flag \leftarrow 0$ };
    return $flag$;
}

Pseudo code to use **Check_1** and **Check_2** to give final answer:

// We are assuming graph to be globally defined.
Final_ans($A[1, ..., n]$**)**{
// First element in a valid BFS order is always the root. We are given (as per clarification mail) that 1 is the root, so apply BFS from 1 to get the distance and parent arrays.
// If the first element, $A[1]$, is not 1, we don't need to check any further.
if$(A[1] \neq 1)$**{ return** 0; **}**
BFS(1**)**;
$flag\_1 \leftarrow$ **Check_1(**$A[1, ..., n]$, $distance[1, ..., n]$**)**;
$flag\_2 \leftarrow$ **Check_2(**$A[1, 2, ..., n], parent[1, ..., n]$**)**;
$ans \leftarrow 0$;
if $(flag\_1 = 1)$ **and** $(flag\_2 = 1)$**{** $ans \leftarrow 1$ **}**;
// We set the ans to 1 only if both conditions are true, otherwise, we let it be 0 (unchanged).
return ans;
}

If **Final_ans(**$A[1, ..., n]$) returns 0, it means that the given sequence is not a valid BFS order, and if it returns 1, it means that the given sequence is a valid BFS order.

Time Complexity Analysis:

We need one call of **Final_ans(**$A[1, ..., n]$). This includes, one BFS traversal, one call of **Check_1(**$A[1, ..., n], distance[1, ..., n]$**)**, one call of **Check_2(**$A[1, ..., n], parent[1, ..., n]$**)** and some comparison and assignment operations which take constant, i.e., $O(1)$ time.
Since our graph is a tree with n nodes, it has $n - 1$ edges. So, BFS takes $O(n + n - 1) = O(2n - 1) = O(n)$ time.
One function call of **Check_1(**$A[1, ..., n], distance[1, ..., n]$**)** includes a for loop and some $O(1)$ time operations. Each iteration of the for loop takes $O(1)$ time and loop runs for $n - 1$ iterations, therefore total time of one function call is $O(n)$.
One function call of **Check_2(**$A[1, ..., n], parent[1, ..., n]$**)** includes a for loop, a while loop, and some $O(1)$ time operations. Each iteration of the for loop takes $O(1)$ time and loop runs for n iterations, hence it takes $O(n)$ time. Each iteration of while loop takes $O(1)$ time.
Also, in each iteration we increment atleast one of i and j. The loop ends when either reaches n, therefore, the loop cannot run for more than $2n$ iterations. So time taken by while loop is also $O(n)$.
So total time of one function call is $O(n)$.

Overall time complexity $= O(n) + O(n) + O(1) = O(n)$.