**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)