Tree Interval Queries Solution

First, some assumptions WLOG:

- The vertices are indexed according to start times
- N is the upper bound on everything (number of nodes, number of queries, sum of k)

Subtask 1

Let's denote by p_i the parent of i, and en_i the end time in dfs of node i (or equivalently the greatest node in the subtree of i).

The required number of connected components equals the number of vertices in S - the number of edges with both endpoints in S. Counting the number of vertices is easy. The number of edges equals the number of vertices $v \in S$ for which $p_v \in S$. This can be found in $O(n)$ by just iterating over all the vertices.

Subtask 3 and 5

Here, we treat this problem as a pure data structures problem.

Consider one query $[L_1, R_1], [L_2, R_2], \dots [L_k, R_k]$. We will do square root decomposition. Consider a value B (to be decided later). If $k > B$, just run the solution of subtask 1. This takes $O\left(\frac{N^2}{B}\right)$. We use a different approach if k is $< B$.

We need to find the number of v with $v \in S, p_v \in S$. This is equivalent to iterating over $1 \le i \le j \le k$ and finding then number of v with $p_v \in [L_j, R_j], p_v \in [L_i, R_i]$ (note that $p_v < v$). This is equivalent to adding the number of v with $v \le R_j$ and $p_v \in [L_i, R_i]$ and subtracting the number of v with $v < L_j, p_v \in [L_i, R_i]$.

To do the above, we do offline processing (that is, we don't process queries in order). For each query q, for each $[L_j, R_j]$ in that query, store $(j, q, 1)$ at index R_j and $(j, q, -1)$ at index $L_j - 1$.

Let's iterate on v from 1 to n, and for every v, add 1 to the index p_v in a fenwick tree. Suppose we are at index $v = x$, right now. Then consider all stored values (j, q, z) at index x. For each such value, iterate over all intervals $[L_i, R_i]$ in the $q-$ th query and add $z\times$ the sum in the range $[L_i, R_i]$ to the answer for query q.

This takes $O(NB \log N)$ time. So, our overall complexity is $O(N\sqrt{N \log N})$ if we choose $B = \sqrt{\frac{N}{\log N}}$. This is the intended solution for subtask 3.

Now, notice that we did $O(N)$ point update operations and $O(NB)$ range queries in the fenwick tree. But both update and query take $O(\log N)$ time. So, can we perhaps do queries faster at the cost of doing updates slower for a better total complexity? There is a simple solution for that. Divide the range $[1, N]$ in $\sqrt{N}$ blocks. Maintain the prefix sum in each block (sum of first j values inside a block), and also the prefix sums of blocks (sum of the first i blocks for each $i \le \sqrt{N}$). Clearly, we can do a point update at point i by just changing the prefix sums in i's block(say b) and also the prefix sums of blocks in $O(\sqrt{N})$. Also, it is really simple to get a prefix sum in $O(1)$ time, as the prefix sum of blocks $< b$ and some prefix sum of block b. This leads to a total complexity $O(N\sqrt{N} + \frac{N^2}{B} + NB)$ which equals $O(N\sqrt{N})$ for $B = \sqrt{N}$, which is the intended soln for subtask 5.

Subtask 2

One possible way is to just use $B = 1$ in subtask 3's solution to get $O(N \log N)$. But, that doesn't give any additional insights. So, let's look at a completely different $O(N \log N)$ solution:

Here, we want to find the number of connected components for a single interval $[L, R]$. Let's define $J_v = en_v + 1$ to be the smallest node $> v$ that we move to, after visiting all nodes in the subtree of v. Also, let's define $J_{n+1} = n + 1$. Note that J_v is either a sibling of v, or some node whose parent is an ancestor of v.

Let's consider the sequence $l_0 = L, l_1 = J_{l_0}, l_2 = J_{l_1}, \ldots$ and let c be the smallest index for which $l_c > R$. Then, we claim that the answer is c. For this, one can observe that for $j < c-1$, $[l_j, l_{j+1})$ is the subtree rooted at l_j, and is by default a connected component. The last range $[l_{c-1}, R]$ is also connected, as $p_v \in [l_{c-1}, R]$ for any $v \in (l_{c-1}, R]$. Also, note that for any i, j there is no edge between $[l_i, l_{i+1})$ and $[l_j, l_{j+1})$.

So, we can just iterate over this sequence, but that is takes $O(N)$ per query (for example in the case of a star graph, where $J_i = i + 1$ for all $i \neq 1$). Instead, we now store jump pointers, $J_v^{(i)}$ being $J(J(\ldots 2^i \text{times}(v))\ldots)$. Now, we can jump in powers of 2 to get to the first index $> R$, yielding an $O(\log N)$ per query algorithm.

Subtask 4

Consider two disjoint intervals $[L_i, R_i]$ and $[L_j, R_j]$ for some $i < j$. Also, let X be the sequence $l_0 = L_j, l_1 = J_{l_0}, l_2 = J_{l_1} \ldots$, as we did in the last subtask. Then, notice that for any $v \in [L_j, R_j]$ for which $p_v \in [L_i, R_i]$, v must be in X. Also, p_v is one of the ancestors of L_j (including L_j itself), which are $\leq D + 1$, if the diameter is D. So, we consider all different ancestors $x \in S$, and count all l_r, whose parent is x, using a similar method as in the last subtask (powers of 2 jumping). Overall complexity would be $O(ND \log N)$

Subtask 6

Let's break the line $[1, n]$ in $2k+1$ alternating intervals (one interval belongs to S, then the next doesn't and so on). For example if the query with $n = 12$ had intervals $[2, 3], [6, 7], [9, 9]$, we break into $[1, 1], [2, 3], [4, 5], [6, 7], [8, 8], [9, 9], [10, 12]$.

Note that the parents of the nodes in the sequence X; $l_0, l_1, \ldots$ are non-increasing. So, the index of the interval they lie in, is also non-increasing. Say currently, we are at node v in the sequence whose parent lies in the range $[L_i, R_i]$ (where $1 \leq i \leq 2k + 1$), then we can find the smallest node in the sequence whose parent is in $[L_r, R_r]$ for some $r < i$, in $O(\log N)$ time using binary jumping. This way, we only iterate over those i such that there is some edge connecting $[L_i, R_i]$ and $[L_j, R_j]$. Also, if $[L_i, R_i] \in S$, we can add some appropriate value (the number of jumps) to our answer.

Consider a graph G on $2k + 1$ nodes, where there is an edge between (i, j), if there is a tree edge, connecting some vertex in $[L_i, R_i]$ to some vertex in $[L_j, R_j]$. Let E be the set of edges of G. Then the time complexity is $O(|E| \log N)$. We note the following property in order to bound the number of edges:

Non-crossing-property: For any $p < q < r < s$, either $(p, r) \notin E$, or $(q, s) \notin E$.

This is to say that there are no sortof cross edges. Let a be a node in $[L_p, R_p]$, b be a node in $[L_q, R_q]$, c be a node in $[L_r, R_r]$, d be a node in $[L_s, R_s]$, such that there is a tree edge between a and c, and between b and d. Clearly $a < b < c < d$. Then c must be a child of a (as nodes are indexed according to start times), and d is a child of b. But b must also lie in the subtree of a, as $a < b < c$, and since d is a child of b, we can't move to c before covering d in the dfs, which means $d < c$, a contradiction.

So, now we'll prove that the number of edges is $O(k)$. There are multiple ways of proving this (for example, induction). One simple way is to see that G is planar, for we can draw all the nodes in a circle in clockwise order, and no two chords will intersect using the non-crossing property.

Therefore, we've solved each query in $O(k \log N)$, leading to a total complexity of $O(N \log N)$