Analysis

Tags: trees, data structures, tree linearization, Euler-tour tree, merge-sort tree, sqrt decomposition, amortized analysis, 2d-segment tree

The statement describes a rooted tree (the workers are the vertices) with bits in its vertices. The queries we are dealing with are updates or questions for a subtree of some vertex *x* limited to some depth *k*.

The first subtask is for 11 points. We do exactly what the statement ask us to do. First, we make the tree with the bits in the beginning. When we have update query (type 1) *x k*, we set 1's to all vertices of the *k*-subordinates of *x* with a simple *DFS*. When we have question query (type 2) *x k*, we count the 1's in the vertices of the *k*-subordinates of *x* with another *DFS*. Obviously, the complexity is $O(\boldsymbol{Q} * \boldsymbol{N})$.

The second subtask is for 15 points. Here the queries are simpler because they are for the whole subtrees of the vertices. One way to make optimal solution with complexity $O(\boldsymbol{Q} + \boldsymbol{N})$ is to adjust the previous solution. The main observations are that one vertex becomes 1 from 0 only one time and when we have update query for a subtree, all following question queries in it are with a clear answer (just the number of vertices in a subtree) and other update queries in it are irrelevant. We don't need such fast solution here and we will describe in details one other solution which is more relevant to the next solutions. When we have queries for a subtree a very useful technique is to linearize it by filling an array *T* with some special order of the vertices. We can do it in the following way. Let we make a *DFS* of the tree in which we push the vertices to *T* at in-time. By doing so, each subtree will be continuous segment in this array and each vertex will be included only once (so the size of *T* is $\boldsymbol{N}$). That means that the update query is to make all elements 1's in some segment of *T* and the question query is to count the 1's in some segment of *T*. So, we can build a segment tree from *T* and we can easily process these queries for the segment tree in $O(\log_2 \boldsymbol{N})$. The complexity here is $O((\boldsymbol{N} + \boldsymbol{Q}) * \log_2 \boldsymbol{N})$.

The third subtask is for 17 points. Here we have only queries of type 2 (question queries). Again, we will linearize the tree in an array *T*. But this time we will build the so-called *merge-sort tree* from the depths of the vertices in *T*. Of course, we will only include the vertices with bit 1 (because we need to count the 1's). In such a way, a node of the *merge-sort tree*, which corresponds to some segment $[l; r]$ in *T*, will have sorted list with the depths of the vertices with 1 in $T[l; r]$. Let's think we have some question query *x k*. The segment $[in[x]; out[x]]$ in *T* is with the vertices of the subtree at *x*. We will process the nodes, which correspond to this query, in the *merge-sort tree*. For such a node we can make binary search in its list for depth $\leq k$ and that will give us the count of the desired vertices. When we sum these counts, we will calculate the answer for the query. The complexity here can be $O(\boldsymbol{N} * \log_2 \boldsymbol{N} + \boldsymbol{Q} * \log_2^2 \boldsymbol{N})$ or $O((\boldsymbol{N} + \boldsymbol{Q}) * \log_2^2 \boldsymbol{N})$ if the *merge-sort tree* is built naively.

The fourth subtask is for 26 points. We can make the previous solution to process also update queries by having *treaps* in the nodes in the *merge-sort tree*. This time in the *merge-sort tree* will be included all vertices (not only those with 1's). Obviously, the *x-key* for the nodes in the *treaps* will be the depths of the vertices. We will need some more information but let us think about how can we make the update query in the first place. One big problem is that when we update some node *nd* in the *merge-sort tree*, we have to update that information for the nodes above him. But the update can be with rather huge information. So, one faster way to

deal with the update queries is for each such query to remove the vertices with 0's in the corresponding *treaps* (i.e. we save the depths of only the vertices with zeroes in the *treaps*). The update information that we have to carry above can be huge but because we remove certain zero only one time, if we sum all the work for this, it won't be above $\boldsymbol{N} * \log_2^2 \boldsymbol{N}$. Another approach which we won't discuss in detail is to do sqrt decomposition of the queries. The expected complexities for this subtask are $O\big((\boldsymbol{N}+\boldsymbol{Q}) * \log_2^2 \boldsymbol{N}\big)$ or $O\big(\boldsymbol{N} * \sqrt{\boldsymbol{Q}}\big)$ but with big constants.

The last subtask is for 31 points. Here the complexity won't be better than in the previous subtask but the solution will be much faster. Instead of making segment tree over the tree linearization and then building data structures for the depths in each node of the segment tree, we will make the opposite. We will build segment tree over the depths and for each node in this segment tree, we will save the vertices with the corresponding depths and make a segment tree over the in-times of the vertices. These segment trees will be for the sum of the vertices with 1's having in-times in some interval. If we make them smarter, they won't take much memory and time for building – the leaves in every segment tree can correspond to the different vertices in the node for which the segment tree is being build. So, the arguments used for the rather small time and memory of *merge-sort tree* also holds here! Because we don't have *treaps*, the question query will be much faster, although the complexity stays the same. For the update query, we will use another idea, taking advantage of the fact that one vertex becomes 1 from 0 only one time. We will use the segment trees to store information in every of their nodes whether they have a vertex with available 1 or not. When we have update query, we will search for each vertex with 1 in the query and update those vertices and corresponding nodes in the segment trees. If we sum all the work, it won't be above $\boldsymbol{N} * \log_2^2 \boldsymbol{N}$. The described solution corresponds to the provided full solution, but it can be also made for the segment tree build over the tree linearization, as in the previous subtask. The final complexity is also $O\big((\boldsymbol{N}+\boldsymbol{Q}) * \log_2^2 \boldsymbol{N}\big)$.

The task was intended to be harder, but in the end, we decided to make it easier so it suits better the contest day. Maybe someday, the original version will come to light.

Author: Iliyan Yordanov
Solutions: Iliyan Yordanov, Encho Mishinev