HandsOn 2

Segment Trees Representation

In both exercises, **Segment Trees** were represented as vectors $2n-1$ long, as this is the exact amount of nodes in the tree. To avoid creating vectors $4n$ long, I decided to enumerate the nodes not as in a binary heap ($2i+1$ for the left child and $2i+2$ for the right), but following the order of an Euler tour traversal.

So if the parent has index i, the left child will be numbered with $i+1$, the right child will have the number immediately following the last one used in the subtree with root the left child. In general, if the vector representing i goes from p to q it will have $2(q-p+1)-1$ nodes, the left child of i representing the subarray in the interval $[p, mid]$ (with mid equal to $\lfloor (q-p)/2 \rfloor$) will be $mid-p+1$ long and consequently will have $2(mid-p+1)-1$ nodes. So the index associated with the right child of i will be $i+2(mid-p+1)$. The -1 disappears because I have to take the next number.

Min-Max Problem

In this problem, I use a segment tree to store the maximum element of each subarray. The segment tree is constructed using the `build_tree()` procedure in a bottom-up manner. When the indices `my_i` and `my_j` are equal it means that I have arrived at a leaf of the segment tree, which represents a single element of the vector, which is inserted into the node. For intermediate nodes, the maximum is calculated inductively by comparing the maximum values between the left and right children (which would represent the first and second half of the segment). Since we touch all the nodes exactly once, and they are $2n-1$, the complexity of the `build_tree()` procedure is $O(n)$.

The first query `Max(i, j)` therefore has complexity $O(\log n)$ because, in the `max_inner()` procedure, I first call the `update_node_from_lazy()` procedure which I will explain later but which has constant cost, and then take the maximum of the two calls made in the first half and the second half of the subarray. These however as we have seen will touch at most $4 \log n$ nodes, which is therefore a logarithmic order cost.

The query `Update(i, j, T)`, on the other hand, being a *range query* that *should* touch all nodes i to j, requires the use of a **Lazy Tree** to keep the query cost logarithmic. The lazy tree is initially initialised with `u64::MAX` as this tree contains the minimum value in each node which replaces it in the main segment tree.

What the `update_inner()` procedure does is to update the maximum of each node touched in what is similar to a search query. The strategy of this procedure is to return the maximum

of the touched segment x, if it has no intersection with the $[i, j]$ portion you want to modify; otherwise if the x contains all elements that fall in $[i, j]$, there are two cases:

- If the maximum value in x is less than or equal to T then this remains unchanged because if T is greater than the maximum then so will all other values in x.
- If, on the other hand, T is less than the maximum of x, it will mean that the new maximum will necessarily be T: if there were in x another value greater than T this would be replaced by T! The values not replaced, on the other hand, are the smaller ones, and so the new maximum of x will be precisely T. To avoid continuing throughout the subtree, I only update the lazy tree in a symmetrical position with the new maximum value and return it.

As mentioned, this procedure for the update is very similar to a search query with some modification of constant cost, so here too the cost will be $O(\log n)$.

As we see in both queries, the `update_node_from_lazy()` procedure is called on the node that is tapped each time, this allows the value of the node to be updated with the correct one before visiting it. The procedure checks that if the value in the main segment tree is less than the value in the lazy tree then it must be replaced with a new maximum value (which is the minimum of the two). Also, if we are not in a leaf node (`m_i != m_j`) we need to propagate any change to the child nodes as well (if they contain a greater value), since they will also be part of the newly changed segment. After any change is made, the node's value in the lazy tree is reset to `u64::MAX`.

In conclusion, if we then have n elements and m queries to execute, the cost of the `build_tree()`, executed only once, being $O(n)$, and the cost of `max()` and `update()` $O(\log n)$, we will have as a total cost of $O(n + m \log n)$.