HandsOn 2

Studente: Luca Rizzo
Matricola: 598992

For both implementations I use a compact **2n layout array** for the segment tree representation instead of the classical 4n array. Since the tree is complete and each node's segment [l, r] determines the number of leaves in the left subtree (`num_left_leaves` = `middle - l + 1`), we can index children in constant time during the DFS build: the left child of node `i` is at `i + 1`, and the **right child** is at `i + (2 * num_left_leaves)`, since a subtree with `num_left_leaves` leaves contains exactly `2 * num_left_leaves - 1` nodes. This yields a contiguous, waste-free representation valid for both implementations.

Exercise 1

The solution uses a segment tree that stores, in each node, the maximum value of the corresponding segment, and supports range updates through lazy propagation.
During either a query or an update, before using the value of a node, we check whether it has any pending updates. If so, we apply the update to the node and propagate the lazy value to its children. In this way, the cost of propagation is amortized across operations, and the value stored in every node is always up-to-date whenever it is accessed. Unlike the case discussed in class, here we cannot perform an additive update on a partially overlapping node (such as applying "val · |segment|" to update the node) because the **`min`** operation is not additive. Its effect on a node depends on the values of its children, and therefore cannot be computed directly from the size of the segment. Therefore, in order to update the value of a node under **partial overlap**, we must recurse into its children and obtain their already-updated values before we can correctly recompute the value of the parent node. When we have a **total overlap** and the new value **T** is smaller than the maximum stored in the node, then all elements in that segment become ≤ **T**. This means that the node's maximum becomes exactly **T**, and all children must eventually be updated as well, since they also contain only values ≥ **T**. However, it is not necessary to update all children immediately: we store **T** as a pending lazy update for the children, and the update will be applied only when needed. If instead **T** is greater than or equal to the node's current maximum, the node's value does not change, but some of its children may still require the update (e.g., because they contain values > **T**). In this case as well, we propagate the lazy value to the children so that they can apply it when they are visited.
With this strategy, the parent node is updated immediately and can be used to compute values for its ancestors, while the propagation to children is deferred and handled lazily. With this approach, a **range update** has the same asymptotic cost as a query, namely **O(log n)**. During an update, we visit only the relevant nodes, and at each of them we apply the update and store the pending value for the children; all these operations take constant time. Similarly, a query only triggers the propagation of pending updates along the visited nodes, and this propagation is also constant-time, keeping the query cost at **O(log n)**. Since building the segment tree requires **O(n)** time, and each of the m queries(both updates and range queries) requires O(log n), the problem is solved in **O(n + m log n)** time.

Exercise 2

The solution is structured in two phases. In the first phase, we build a Segment Tree in average time **O(n log n)**, which allows us to answer each query in **O(log n)**. Since we must process *m* queries, the overall running time is therefore **O((n + m) log n)**. As a preliminary step, we build the **segment_coverage** array, which stores for each position in [0,n−1] the number of segments covering that position. To construct it, we use an approach similar to sweep line and counting sort for events: we initialize an array of length n with zeros in **O(n)** time; then, for each segment [l,r], we apply two updates (+1 at l and −1 at r+1) for a total cost of **O(#segments) = O(n)**. Finally, we compute the prefix sum in **O(n)**, obtaining the **segment_coverage** array of size n. This entire phase requires **O(n + #segments) = O(n)**.

Next, we build a **Segment Tree** over this coverage array, where each node stores a **HashSet** containing all distinct coverage values present in its interval. Each element of the coverage array contributes to **O(log n)** nodes of the tree (one per level), and inserting into a HashSet takes average **O(1)** time; therefore, the overall construction cost is **O(n log n)** on average. A query **IsThere(i, j, k)** is answered by visiting the Segment Tree on the interval [i,j]. Each visited node falls into one of the standard cases: with **no-overlap** we can immediately return *false*; with **total-overlap** we can directly check in average **O(1)** time whether k belongs to the node's HashSet; **partial-overlap**: we recurse into both children and combine their results with a logical OR.
At each level of the Segment Tree, at most four nodes intersect a query interval, and only two of them can be in total-overlap in which case we actually check the HashSet.
Since the tree has height **O(log n)** and each HashSet lookup costs **O(1)** on average, each query takes **O(log n)** time on average overall.