Hands-on 2 - Competitive Programming

Chiara Molinari

August 2023

1 Min and Max

The solution to this problem uses a binary segment tree. It is a variant of the standard segment tree used for the range-max-query problem. Each node is a couple (value, bool), where value is the max of the first entries of its children, while bool describes whether the node is "solved" or not (description of this condition will follow).
A max query on an interval I consists of a standard (almost) exploration of the tree: we look for nodes that cover sub-intervals of I.
An update (interval I, value M) consists of a similar exploration of the tree. This time, when a node covers an interval that needs to be updated, such node is updated as follows.
Its first entry is possibly lowered to M. If so, its second entry becomes false, meaning that the node is "unsolved". Indeed, the update will not propagate deeper, for now.
However, the update is propagated upwards.
What about unsolved nodes? In fact, in both kinds of queries, when going deeper, if we encounter a node that is unsolved, we solve it (propagating its update of one step).
This means that if some node is (value, false), it becomes (value, true) and its children's first entries are possibly lowered to 'value'. If so, their second entries will become false, making them 'unsolved'.

Time complexity: the tree's size is $O(n)$, making the time complexity for performing max-queries and updates $O(\log(n))$. Building it will cost $O(n)$. The resolution of unsolved nodes requires $O(1)$ time and this can happen only when a query or update happens. This gives us a time complexity of $O(m \log(n) + n)$.

Space complexity: the described segment tree requires $O(n)$ space. The queries, if stored, require $O(m)$ space. In my implementation they are stored, giving a space complexity of $O(n + m)$.

2 Queries and operations

Here the considered data structures are vectors.
Each query affects a specific interval. We store in two vectors (size m) their beginning and ending. Using such vectors, we construct a vector (size m) that tells how many times an operation is performed.
This is possible because of the queries being offline.
Then we reduce the problem to having exactly m operations. Indeed, if an operation that consists of adding k is applied t times, then a unique operation on the same interval, with constant $k * t$, can replace it.
As before, we store in two vectors (size n) where the intervals of the operations begin and end.
Using such vectors, we construct a vector (size n) to add (entry by entry) to the original vector.

Space complexity: $O(1)$ vectors are involved, of size either $O(n), O(m), O(k)$, giving a time complexity of $O(n + m + k)$.

Time complexity: the algorithm scans all the involved vectors $O(1)$ times, giving a total time complexity of $O(n + m + k)$.