Assignment 2 Competitive Programming and Contests

Dario Bekic

November 21, 2024

Contents

1 Min and Max **1**
1.1 Reasoning 1
1.2 Time Complexity 2
1.2.1 Max 2
1.2.2 Update 2
1.3 Space Complexity 2

2 Is There **2**
2.1 Reasoning and Time complexity 2
2.2 Space Complexity 3

1 Min and Max

1.1 Reasoning

First, note that if we considered only the `Max` query we could use a standard segment tree, where at each node we store the maximum for the interval represented by the node. Our implementation uses this approach and adds support for the updates.
The `Update(i, j, T)` query affects only those positions $i \leq k \leq j$ which are strictly greater than `T`, in a sense it *levels* the maximum to be, at best, `T`. If the maximum is strictly greater than `T` then the interval's maximum needs to be set to `T`. However, we might not want to update all the intervals, we can think therefore of a solution involving *lazy propagation* where we do not update subintervals until the moment a `Max`(or `Update`) query is queried

on them. Then, when we traverse the parent node, during the execution of a query, we push the minimum of all `T`s queried on the interval to the child nodes, which will potentially change their maximum and push the lazy value to their children.

1.2 Time Complexity

1.2.1 Max

A `Max(i, j)` query needs to access all those nodes whose intervals partially overlap with interval `[i, j]`(those that are totally wrapped in it immediately return). Since any node has at most two children intervals, in the worst case, both of them overlap with our target interval. But the number of nodes partially overlapping is bound to be at most 2 since indexes `i,j` belong to at most one of the intervals in a specific level of the tree, therefore the active nodes are only 4 per level, at worst. The lazy propagation overhead is constant: we just look up at `lazy[idx]` to see if there is any `T` not propagated, if that is the case the following is done:

- Set the maximum value of the node to the minimum between its current maximum value and the new lazy value propagated.
- Set `lazy[left(idx)]` and `lazy[right(idx)]` to the minimum of current node's lazy value and their respective lazy values. An empty lazy value has default value `usize::MAX`, indicating that it cannot lower any node.

The time complexity is $\mathcal{O}(log(n))$.

1.2.2 Update

A `Update(i, j, T)` query needs to update the lazy value of all the intervals partially or totally overlapping with interval `[i, j]` and set it to `min(T, lazy[idx])`. Since the update with the propagation represent a constant factor and the number of nodes visited is in the same order as the one visited by a `Max` query, the time complexity is $\mathcal{O}(log(n))$.

1.3 Space Complexity

We are required to store two segment trees: one for the actual values and one for the lazy propagation. Thus requiring $\mathcal{O}(n)$ storage.

2 Is There

2.1 Reasoning and Time complexity

To compute the number of segments containing a given point we can perform a stabbing query. A naive algorithm could compute a stabbing query for each point in the range queried by the `isThere(i, j, k)` query. However, since the range can span the union of all the segments and a stabbing query has logarithmic complexity in the number of nodes in the segment tree, this would result in a $\mathcal{O}(mnlog(n))$ overall complexity. This approach thus seems unreasonable.
We might use a form of storage, in this case we can use sets, for example Red-black trees or Binary search trees, in each node of the segment tree to store the result of the stabbing query for each of the indexes of the interval it represents. Then, for each `isThere(i, j, k)` query, we just need to find all the intervals in the segment tree containing sub-intervals of interval `[i, j]`, which takes $\mathcal{O}(log(n))$ time and perform for each such interval a $\mathcal{O}(log(m))$ look-up query where m is the length of the interval. Therefore a `isThere` query has complexity $\mathcal{O}(log(n) + log(m_1) + log(m_2) + ...)$ where $m_1, m_2, ..$ are lengths of subintervals of interval `[i, j]` whose sum is bound to be less than or equal to n therefore the complexity is at most $log(m_1 \cdot m_2 \cdot m_3...)$ which can be proven to be bounded to $log(\frac{n^2}{k^2})$, where k is the number of sub-intervals, therefore we have a per query complexity of $\mathcal{O}(log(n))$ which for m queries takes us to $\mathcal{O}(mlog(n))$ time complexity.
However, we have to take into account the building of the segment tree, in particular, the time needed for doing a stabbing query for each point $0 \leq i \leq n-1$, and the time to propagate its value throughout all the nodes which have that index in their interval.
Roughly, a stabbing query is done in $\mathcal{O}(log(n))$ time, so for n points that amounts to $\mathcal{O}(nlog(n))$ time; note that here we are not interested in retrieving the actual segments, just the number.
Then we need to build our segment tree with the actual sets. To do this we simply push the value of the index(meaning the number of segments which contain it) to all the segments who have the index in their interval, this is too done in logarithmic time and it is done for each point. Thus, this adds another $\mathcal{O}(nlog(n))$ factor which however, does not change the total complexity of the build operation which is $\mathcal{O}(nlog(n))$.
In total, the solution has $\mathcal{O}((n+m)(log(n)))$ as time complexity.

2.2 Space Complexity

We need space for the two segment trees, one for the stabbing queries, in our code named `SegmentTreeStab`, and another one which answers the `isThere` queries, named `SegmentTreeSet`. Our implementation uses a `BTreeSet` for each node of the `SegmentTreeSet`, which will store at most `r-l+1` elements; since each index can be in at most one node in a level of the segment tree, the total size occupied by the nodes in one level is bound to be less than n, therefore the total extra space needed is $\mathcal{O}(nlogn)$.