Second hands-on report

Francesco Scarfato

December 18, 2024

In this hands-on there are two problems to solve.

First problem: given an array $A[1, n]$ of positive integers, each integer is at most n, the goal is to implement a data structure that supports two operations:

- `Update(i, j, T)` that replaces every value $A[k]$ with $\min(A[k], T)$ with $i \leq T \leq j$,
- `Max(i,j)` that returns the largest number in the range $[i, j]$.

Second problem: given n segments as pairs (l, r) with $0 \leq l \leq r \leq n-1$, the goal is to support the operation `isThere(i,j,k)`. This operation return 1 if there exists a position p, with $0 \leq i \leq p \leq j \leq n-1$, that is contained in exactly k segments.

1 Min and max

The target time complexity for this problem is $O((n+m)\log n)$. The idea is to build a segment tree and put the answer of the query (the maximum of the range) in the nodes of the tree.

The update operation on one node for the segment tree takes $O(\log n)$ time. In this case, the update operations can affect multiple nodes, so the time depends on the number of nodes to change. To maintain the efficiency of the update, it is possible to use a *lazy propagation* that permits to postpone some updates and change only when necessary. The main data structure includes: the `tree` structure that represents the effective segment tree, the `upper_bound` of the interval for the whole tree and a `lazy` tree to track the pending updates. It is used an array to represents the tree data structure to have direct access to nodes without use of pointers. The left ad right child are accessed respectively with $2 \cdot c + 1$ and $2 \cdot c + 2$ with c index of the current node. The leaves are the elements of the input array so n.

```
pub struct SegmentTree {
    tree: Vec<i32>,
    upper_bound: usize,
    lazy: Vec<Option<i32>>,
}
```

The size of the tree is $4 \cdot n$ because this represents a perfect binary tree and is a safe upper-bound because it is possible to exploit the accessing methods described below. The size might be less but, in order to not have "out of bound" errors with the previous accessing methods, the size is always $4 \cdot n$.

```
fn build(&mut self, nums: &Vec<i32>, left_bound: usize, right_bound:
    usize, curr_index: usize) {
    if left_bound == right_bound {
        self.tree[curr_index] = nums[left_bound];
        return;
    }
    let mid = Self::mid(left_bound, right_bound);
    let left_child = Self::left(curr_index);
    let right_child = Self::right(curr_index);

    self.build(nums, left_bound, mid, left_child);
    self.build(nums, mid + 1, right_bound, right_child);

    self.tree[curr_index] = max(self.tree[left_child],
        self.tree[right_child]);
}
```

The tree is built recursively with a post-order visit in the `build()` function. When it arrives to leaves, so when the left bound and the right bound of the interval are equal (`left_bound == right_bound`), the function inserts the correspondent element of the input array. During the upward traversal to the root, each parent node is updated with the maximum value of its child nodes. This function takes $\Theta(n)$ because it visits all the node of the tree once.

The update query is divided in four parts:

1. **Update and propagation**: if there is a pending update in the current node, then it updates the current node and propagates to the children. The new value T is propagated to at least one child whose value is larger than T, otherwise is not propagated.

```
fn propagate(&mut self, curr_index: usize) {
    if let Some(lazy_val) = self.lazy[curr_index] {
        self.tree[curr_index] = lazy_val;
        if curr_index < self.tree.len() / 2 - 1 {
            if self.tree[left_child] > lazy_val {
                self.lazy[left_child] = Some(lazy_val);
            }
            if self.tree[right_child] > lazy_val {
                self.lazy[right_child] = Some(lazy_val);
            }
        }
        self.lazy[curr_index] = None;
    }
```

2. **No overlap**: if the current node doesn't cover the current range of the query then the function do nothing.

3. **Total overlap**: if the current node cover the entire range of the query and the new value T is less than the current node, the update and propagation is forced by setting a pending update on the current node.

```
if left_query <= left_bound && right_query >= right_bound {
    if new_val < self.tree[curr_index] {
        self.lazy[curr_index] = Some(new_val);
        self.propagate(curr_index);
    }
    return;
}
```

4. **Partial overlap**: if the current node doesn't cover entirely the current range of the query, then it starts the traversal of the tree with a post order visit. During the upwards traversal, it updates the current node with the maximum between the children because their contents could be changed. The middle index is the middle value in the current range covered by the node, and it is used to traverse the tree.

Thanks to the efficiency of the lazy propagation, when the algorithm traverses the tree it applies delayed updates, but exactly as much as necessary, in order to keep $O(\log n)$ complexity per query.

The max query visits the tree to find the nodes that cover entirely the query and returns the max. The structure is similar to the update query in fact:

1. it updates and propagates if there is a pending update on the current node;

2. return the current node if it totally overlapped the range of the query;

3. in partial overlap situation, it visits the tree with a post order visit and returns the max of the children.

```
fn max_query(&mut self, curr_index: usize, left_query: usize, right_query:
    usize, left_bound: usize, right_bound: usize) -> i32 {
    self.propagate(curr_index);

    if left_query > right_bound || right_query < left_bound {
        return -1;
    }
    if left_query <= left_bound && right_query >= right_bound {
        return self.tree[curr_index];
    }
```

```
    let mid_index = Self::mid(left_bound, right_bound);
    let left_child = Self::left(curr_index);
    let right_child = Self::right(curr_index);

    let max_left = self.max_query(left_child, left_query, right_query,
        left_bound, mid_index);
    let max_right = self.max_query(right_child, left_query, right_query,
        mid_index + 1, right_bound);

    max(max_left, max_right)
}
```

This query takes $\Theta(\log n)$ for one query like the previous function.

2 Is there

The target time complexity for this problem is $O((n+m)\log n)$. The goal is to build a segment tree where each node stores the number of segments overlapping in its range.

To compute the number of overlapping segments, it is used the sweep line technique.

```
fn compute_occurrences(segments: &Vec<(usize, usize)>, size: usize) ->
    Vec<i32> {
    let mut count = vec![0; size];

    for &(l, r) in segments {
        count[l] += 1;
        if r + 1 < size {
            count[r + 1] -= 1;
        }
    }

    for i in 1..size {
        count[i] += count[i - 1];
    }
    count
}
```

The key idea is to treat the segment start and end points as events that modify a running count of active segments. By iterating through the segments, the algorithm increments the count at the segment's starting point l and decrements it just after the segment's ending point $r+1$. A subsequent cumulative

sum of the `count` array propagates these incremental changes across the entire range, obtaining the total number of overlapping segments at each position in a single pass. This approach avoids explicitly processing each segment individually at every position, making it highly efficient with a time complexity of $O(2n) = O(n)$.

The `HashSet` data structure is used to efficiently represent the number of overlapping segments within a node. In fact, the query checks for the existence of exactly k overlapping segments within a specified range and, by exploiting the constant-time average complexity of `HashSet` operations, it is possible to verify quickly if the desired value k exists in the set associated with the node.

```
fn build(&mut self, occurrences: &Vec<i32>, left_bound: usize,
    right_bound: usize, curr_index: usize) {
     if left_bound == right_bound {
          self.tree[curr_index].insert(occurrences[left_bound]);
          return;
     }

     let mid = SegmentTree::mid(left_bound, right_bound);
     let left_child = SegmentTree::left(curr_index);
     let right_child = SegmentTree::right(curr_index);

     self.build(&occurrences, left_bound, mid, left_child);
     self.build(&occurrences, mid + 1, right_bound, right_child);

     let merged_set: HashSet<i32> =
         self.tree[left_child].union(&self.tree[right_child]).cloned().collect();

     self.tree[curr_index] = merged_set;
}
```

The building of the segment tree is very similar to the previous problem but without the lazy propagation concept. In the upward traversal, the algorithm merges the sets of the children with this:

```
self.tree[left_child].union(&self.tree[right_child]).cloned().collect();
```

The code merges two `HashSet` instances by using the `union` method to create an iterator over their elements references, then `cloned` to generate an iterator over referenced cloned values, and finally `collect` to gather the results into a new `HashSet`. This always takes $O(n)$.

The query method is trivial and takes $O(\log n)$ for each query.

```
fn is_there_query(&self, curr_index: usize, left_query: usize,
    right_query: usize, overlapped_seg: i32, left_bound: usize,
    right_bound: usize) -> bool {
     if left_query > right_bound || right_query < left_bound {
         return false;
     }

     if left_query <= left_bound && right_query >= right_bound {
         return self.tree[curr_index].contains(&overlapped_seg);
     }

     let mid_index = Self::mid(left_bound, right_bound);
     let (left_child, right_child) = (Self::left(curr_index),
        Self::right(curr_index));

     self.is_there_query(left_child, left_query, right_query,
        overlapped_seg, left_bound, mid_index)
     ||
     self.is_there_query(right_child, left_query, right_query,
        overlapped_seg, mid_index + 1, right_bound)

}
```