Competitive Programming and Contests Hands-On 2 report

Salvatore Salerno 2024/2025

Problem 1 - Min and Max

Given an array $A[1, n]$ of n positive integers, where each value is guaranteed to lie between 1 and n, the task is to construct a data structure that can answer the following queries in $O(\log_2 n)$ time:

- **Update(i, j, t)**: given $0 < i \leq j \leq n$, for every value $k \in$ `[i,j]` set $A[k] = \min(A[k], t)$.
- **Max(i, j)**: given $0 < i \leq j \leq n$, return the largest element in $A[i, j]$.

Solution using Max Segment Tree

A possible solution employs a *Max Segment Tree*, a type of *Segment Tree* where each node stores the maximum element within its indexed range. By constructing such a tree for a given array $A[1, n]$, the `Max(i, j)` query can be answered in $O(\log_2 n)$ time through a traversal of the tree. The `Update(`i`, `j`, `t`)` operation, which consists in a range update, may require visiting all nodes in the tree in the worst-case scenario (e.g., when the query range spans the entire array). To optimize this query, lazy propagation is used. Instead of immediately updating all affected nodes, the recursion halts at the first node whose range totally overlaps the query. This node is marked to indicate that its subtree requires updating. Subsequent queries or updates visiting the marked node will then propagate the pending changes to its children as needed. This strategy reduces the `Update(`i`, `j`, `t`)` query to a traversal of the tree, reducing the complexity to $O(\log_2 n)$. It is important to note that, while updating values is allowed, the tree does not support changing the number of elements in the array. In such cases, a new *Max Segment Tree* must be constructed.

Implementation of the Max Segment Tree

An implementation of the *Max Segment Tree* is provided in the file *lib.rs*. The tree is defined by the struct `MaxSTree`, which consists of:

- **nodes**: a vector that contains the nodes of the tree.
- **root**: the index of the root node in the vector.

Each node is represented by the struct `MaxNode`, which includes:

- **key**: The maximum value in the range indexed by the node.
- **children**: a pair of indices representing the node's children.
- **range**: a pair specifying the range of indices covered by the node.
- **pending**: A boolean flag indicating if the node has a pending update that needs to be propagated to its children.

Assume the cost of storing all this information in a single node is $\Theta(c)$, where $c \geq 1$. Since the tree is a complete binary tree with n leaves, it contains $2n - 1 = O(n)$ nodes. Therefore, the array `nodes`, which implements the tree, has a storage cost of $\Theta(n) \cdot O(c) = O(n \times c)$. Additionally, accounting for the cost of storing the index of the root, the total space complexity of the Max Segment Tree is: $O(c \times n) + 1 = O(c \times n)$, where $c \geq 1$.

Implementing the Queries

The `Update(`i`, `j`, `t`)` and `Max(`i`, `j`)` queries are implemented through the public methods `update(q_range, t)` and `max(q_range)` of the `MaxSTree` struct. Both methods follow a similar recursive process: they traverse the tree based on the range of the current node and the range specified by the query, identifying the nodes that contribute to the solution. During traversal, each node falls into one of three categories:

- **No Overlap:** The ranges do not intersect, so the current node and its subtree do not contribute to the query.
- **Total Overlap:** The query range fully encompasses the node's range, meaning the node entirely contributes to the query's solution.
- **Partial Overlap:** The node's range partially intersects with the query range, requiring further traversal of one or both of its subtrees.

This recursive traversal begins at the root and identifies the nodes that are relevant to the query. For the `update(q_range, t)` method, lazy propagation is applied to the relevant nodes using the `lazy_update(new_val)` method. This involves marking the node's `pending` flag as `true` and updating its key to $\min(\text{node's\_key}, t)$. For leaf nodes, there is no need to set the `pending` flag, since they have no children. For the `max(q_range)` method, the maximum value among the partial solutions contributed by the relevant nodes is computed and returned.

Problem 2 - Is There

Given a set of n segments $S = \{s_1, \ldots, s_n\}$, where each segment $s_i \in S$ is represented as a pair (l_i, r_i) such that $0 \leq l_i \leq r_i < n$, the goal is to construct a data structure that can answer the following query in $O(\log_2 n)$ time:

- **Is There(**i**,** j**,** k**)**: Returns 1 if there exists a position $p \in [i, j]$ (where $0 \leq i \leq j < n$) that is overlapped by exactly k segments from the set S, 0 otherwise.

Solving an harder problem

To find a solution, it was considered solving the more general query `How Many(`i`, `j`, `k`)` which returns the number of positions $p \in [i, j]$ (where $0 \leq i \leq j < n$) that are overlapped by exactly k segments. If the query returns a value greater than 0, then the answer to the original problem is 1; otherwise, it is 0.

The Position Array

To address the more complex problem, we need to determine, for each position $i \in [0, n-1]$, how many segments of S overlap at position i. This information will be used for constructing a Segment Tree. To store this data, we define a *position array* $A[0, n-1]$ such that for all $i \in [0, n-1]$, $A[i]$ represents the number of segments that pass through position i. To construct the array, we apply a *Sweep Line* approach on the set of segments S. For each $s_i = (l_i, r_i) \in S$, we increment $A[l_i]$ by 1 and decrement $A[r_i + 1]$ by 1. After processing all segments, we perform a cumulative sum scan over the array A, where for each $i \in [1, n-1]$, we update $A[i]$ as $A[i] += A[i-1]$, obtaining the final array.

Constructing the position array boils down to two linear scans, over sets of n elements, then the time complexity for constructing A is $O(n)$, and the space complexity is $\theta(n)$.

Solution using a Frequency Segment Tree

The solution utilizes a *Frequency Segment Tree*, a variant of the *Segment Tree*, where each node stores a map of pairs in the form (k, p) within its range. Specifically:

- **k**: A value between 0 and n inclusive, representing a number of overlapping segments.
- **p**: The number of positions within the node's range where exactly k segments overlap.

By using this data structure the problem reduces into one similar to the *Occurrences of x in a range* problem we have seen in previous lessons. By optimizing each node's key with a suitable data structure, such as a optimal hash map, each access on a node's table along the query path can be approximated to $O(1)$. Consequently, the query can be solved by navigating the tree, yielding an overall query time complexity of $O(\log_2 n)$.

Implementation of the Frequency Segment Tree

An implementation of the *Frequency Segment Tree* is provided in the file *lib.rs*. The tree is defined by the struct `FreqSTree`, which follows a similar structure to the `MaxSTree` described in earlier sections. The key difference lies in the type of nodes it contains. The tree uses `FreqNode`, which contain as key value an hash map. The hash map used is the one provided by Rust's standard library. Each map can store at most n entries, but any entry with a value of 0 (indicating no positions with that number of overlapping segments) is omitted. The entries in the map are represented as pairs of unsigned integers.

Each node can be constructed in one of two ways:

- **If it is a leaf:** The map contains a single entry, $< A[i], 1 >$, where i is the position indexed by the leaf.
- **If it is an internal node:** The map is constructed by merging the maps of its child nodes, summing overlapping entries and inserting distinct ones.

Compared to the `MaxSTree`, the space complexity increases due to the hash maps stored in each node and the additional array of position we constructed. The space complexity is therefore $O(n \times (c + \gamma)) + O(n) = O(n \times (c + \gamma))$, where $\gamma \geq 1$ represents the growth factor due to the hash maps.

Implementation of the Is There query

The `Is There(`i`, `j`, `k`)` query is implemented as the public method `is_there(q_range, k)`. This method traverses the tree in a manner similar to the queries in the previous problem. However, the key difference lies in the fact that we are interested in determining whether there exists a position, rather than retrieving the positions themselves. As a result, the execution can be terminated as soon as we encounter a node whose range is $\in [i, j]$ containing the pair (k, n_p) in its map with $n_p \geq 1$. This allows us to immediately return 1 if a solution is found. In the case of partial overlap, if one of the subtrees already provides a solution, there is no need to recurse into the other subtree, and we can return 1. Only if both subtrees return 0 then no solution exists and return 0.

Testing the Implemented Methods

A `main.rs` file is provided to verify the correctness of the implementation against the supplied test sets. Each set is associated to one of the problem and contains a collection of tests. Each test consist of a pair of input and output text files: the input file specifies a sequence of queries for the associated problem, while the output file contains the expected results. The `main` function executes each test by performing the operations described in the input file and comparing the produced results to the expected outputs. To ensure proper execution, the test cases must be placed in the root directory of the Cargo project under the folders `Testset_handson2_p1` and `Testset_handson2_p2`. The time complexity of processing each test case is $O((n+m)\log_2(n))$, where n is the size of the input array and m is the number of queries.