HANDS ON 2 REPORT

Competitive Programming and Contests

University of Pisa, Academic Year 2024-2025

SALMI ABDERRAHMANE

a.salmi1@studenti.unipi.it

Problem #1: Min and Max

Approach

We will use a **Segment Tree** to solve this problem efficiently, which will allow us to:

- Perform range updates for the **Update(i, j, T)** query.
- Perform range maximum queries for the **Max(i, j)** query.

The problem states that the solution must run in **O((n+m)log n)** time, where n is the length of the array, and m is the number of queries. To achieve this, we have to use lazy propagation in our segment tree.

For the **Update(i, j, T)** query, we will use lazy propagation which will help us propagate updates only when necessary.

For the **Max(i, j)** query, we will build the tree to store the maximum values within segments.

Data Structure

The data structure we will use is SegmentTree which has the following fields:

- **n**: The length of the original data array.
- **tree**: A vector representing the segment tree, where each node holds a calculated value (either min or max depending on the problem).
- **lazy**: A vector for storing pending updates (needed for lazy propagation).

Implementation

Segment Tree Construction

The **init** function initializes the Segment Tree object using an input array **a** and specifies if it should store minimum or maximum values (depending on the problem). It initializes the tree and lazy vectors, then calls build fun to fill the tree with values.

The **build** function recursively constructs the Segment Tree by dividing the input array into segments. For leaves, it directly assigns the value, and for segments, it combines the values of left and right child nodes using either **min** or **max**.

Max Query

The **max_query** function is just a public fun that calls a local recursive fun **max_query_rec.**

The **max_query_rec** function performs the core logic for finding the maximum value within a given range by checking overlaps.

- It starts by applying any pending updates that might exist in the lazy tree with the fun **apply_lazy_update** before processing the node.
- Then we check the overlaps, if no overlap, it skips the node.
- If there's a total overlap, it returns the node's value
- If its partial overlap, it recurses on the left and right children and returns the maximum of their values.

Range Update Query

The **update_range** just calls the recursive fun **update_range_rec** with the specified range and the T value.

The **update_range_rec** function carries out the range update by:

- Firstly applying any pending lazy updates using **apply_lazy_update**.
- Then it checks for overlaps, if there's no overlap with the current range, it skips the node.
- For total overlap, it updates the node to the minimum of its current value and the T value. And if the node has children, it propagates the update for its children by saving the update in the lazy tree, which will be applied only when these nodes are accessed the next time.
- For partial overlap, it recursively updates the left and right children, then sets the current node to the maximum of its children's values.

Time Complexity

- The **Build** operation takes **O(n log n).**
- Each **Update** and **Max** query takes **O(log n)** time because of the lazy propagation.

Space Complexity

- Segment Tree: **O(n)**, in reality it needs 4*n elements to ensure that the tree can accommodate all possible nodes and leaves, which results in O(n)

Problem #2: Is There

Approach

To solve this problem, we will also use the Segment Tree solution from the previous problem, and we'll also use difference array and prefix sum to represent how many segments cover each position in the array.

Implementation

Difference Array

We call the fun **build_frequency_array** with our segments array, for each segment [l, r], it'll increment the count at index **l** and decrement the count at index **r** + **1**. This will help in tracking coverage of positions in the array.

Prefix Sum

After processing all segments, we can build a prefix sum array where each element represents the number of segments covering that position.

Answer the Queries

For each query, call the fun **exists_exact_coverage**, which will call recursive fun **range_exact_check**, which does the following:

- Applies lazy updates if there are any
- Checks for overlaps, if there's no overlap with the specified range, it returns false
- If there's total overlap, it checks for an exact match by calling **find_first_match**, which is a fun that searches for an exact match of a given value in the tree starting from a specific node, if it finds it it returns true, otherwise false
- If there's partial overlap, it recursively checks the left and right children, returning true if either subtree has a match

Time Complexity

- Building the Frequency Array: **O(n)**
- Segment Tree Initialization: **O(n)** where n is the size of the frequency array
- Query Handling: **O(log n)**

Space Complexity

- Frequency Array: **O(n)**
- Segment Tree: **O(n)**