Competitive Programming and Contests

Hands-on 2

LEONARDO DA POZZO

A.A. 2022/2023

1 Introduction

The purpose of this hands-on is to solve two different problems based on range queries through the use of segment trees. Both solutions were implemented in the Rust programming language.

2 Tree Implementation

A segment tree struct with nodes stored in an array was used to solve both problems. The tree was completed with dummy nodes in order to make things simpler. Lazy propagation is implemented by storing a *pending* field wrapped in an *Option* object in each $Node$ struct. Pending updates are propagated to a node's children only when the node itself is visited.

3 Min and Max

3.1 Segment Tree Contents

For this problem, the segment tree's nodes contain both the minimum and the maximum elements of the underlying range.

3.2 Range Update

One of the two requested operations is the following: given two boundaries i and j, as well as a number T, substitute all the elements x included in the $[i, j]$ range with $min(x, T)$. To solve this query, we start from the tree's root and recursively traverse it downwards, propagating each touched node's pending updates. At each step we decide how to proceed based on how the current node's range overlaps with $[i, j]$:

- no overlap: the node is outside the desired range, so we return immediately;

- total overlap: we update the node's value and set the children's *pending* field to T (if applicable);
- partial overlap: we perform a recursive call to both children. The current node's *data* field is then recomputed by combining his children's.

In all cases we return the current's node *data* field.

3.3 Range Max Query

The second query is as follows: given two boundaries i and j, return the maximum value in the range $[i, j]$. Similarly to the update query, we start at the root and recursively traverse the tree, propagating any pending update. Nodes are treated as follows based on the range overlap:

- no overlap: the node is outside the desired range, so we return a dummy value.
- total overlap: return the node's *data.max* value.
- partial overlap: perform a recursive call on the children, then return the maximum between the two recursive calls' results.

3.4 Complexity

3.4.1 Building the Tree

Building the tree takes $\Theta(n)$ time, and the tree itself takes up $\Theta(n)$ space.

3.4.2 Queries

Each query takes $\Theta(log(n))$ time. The use of lazy propagation prevents update queries from taking $\Theta(n)$ time.

3.4.3 Total

We take $\Theta(n)$ time to build the tree and $\Theta(m \cdot log(n))$ time to solve m queries. Overall, this sums up to $\Theta(m \cdot log(n) + n)$.

4 Queries and Operations

4.1 Segment Tree Contents

For this problem, the segment tree contains the arrays' elements in its leaves and dummy data in the rest of the nodes. Since the segment tree is not being fully exploited, it is likely a better solution exists which either makes a better use of the ST or that uses another data structure altogether.

4.2 Range Update

This problem involves only one kind of operation on the tree: range updates. The difficulty stems from the fact that we have a set of potential updates and each query asks us to apply a subset of those to the tree. If done naively, this would require us to iterate over some (potentially all) the updates of each query's subset, leading to $O(k \cdot m)$ update operations on the tree, where k is the number of queries and m is the cardinality of the aforementioned set. The key steps of the chosen approach are the following:

- store a vector *ops* of m *Operation* structs, each of which contains *start* and *end* fields denoting a range and a *val* field holding the value we want to add to the elements of said range;
- create a vector $diff$ of $m + 1$ elements initialized to 0, which will be our difference array (see the References section at the bottom);
- for each query of range $[i, j]$, we modify the difference array by incrementing position i and decrementing position $j + 1$;
- create a final array *count* of size m and populate each position i with $diff[i]+diff[i-1]$ (except for the first element, which will be just $diff[0]$). The result is an array which stores at position i the number of times we need to perform update $ops[i]$;
- iterate over the *count* array and perform at each iteration the ith update with value equal to $count[i] \cdot ops[i].val$.

Note that these updates are done lazily: the propagation to the leaves only happens at the final step, when we traverse the whole tree once and print the leaves.

4.3 Complexity

4.3.1 Building the Tree

Building the tree takes $\Theta(n)$ time, and the tree itself takes up $\Theta(n)$ space.

4.3.2 Updates

Each update takes $\Theta(log(n))$ time.

4.3.3 Total

We take $\Theta(n)$ time to build the tree and take $\Theta(m \cdot log(n))$ time to perform all updates on it. We also need $\Theta(k)$ time to update the difference array. This results in $\Theta(m \cdot log(n) + n + k)$ time.

5 References

The Difference Array technique was taken from here: `https://www.geeksforgeeks.org/difference-array-range-update-query-o1/`.