HandsOn - 2

Simone Passèra

Nov 24, 2024

1 Introduction

The objective of this hands-on is to implement and play with Segment Trees in Rust. There are two problems to solve.

2 Code Implementation

2.1 Exercise 1

I implemented a segment tree [1] using a dynamically allocated `Node` struct wrapped in a Box. Each node stores the maximum value of the range it represents and a lazy value, which holds a deferred maximum value. The lazy value is used to optimize the update operation by deferring changes to child nodes until necessary, avoiding redundant recalculations.

The tree is built recursively in $O(n)$ time, where n is the size of the array. Both range queries and updates operate in $O(log(n))$, as they traverse the height of the tree, splitting the operation as needed between left and right children. The space complexity is $O(n)$, as the tree requires at most $2n - 1$ nodes, proportional to the size of the input array.

2.2 Exercise 2

For this exercise, as in the previous one, I implemented another segment tree named `NodeSegments` using the same design strategy. The key idea to solve the `is_there` operation is to store, at each node, all possible answers for the interval represented by that node. Specifically, each node contains a vector of size $n + 1$, where at position k, it stores the number of positions in the interval that are covered by exactly k segments.

The time complexity for building the tree is $O(n)$ because each node is initialized exactly once during the recursive process. Update and query operations have complexities of $O(k \cdot log(n))$ and $O(log(n))$ respectively, where $k = n + 1$ represents the maximum size of the counts vector in each node. In terms of space complexity, the tree requires $O(n^2)$, as each node stores a counts vector of size $n + 1$.

[1] Sources consulted: Link 1: geeksforgeeks. Link 2: cp-algorithms.