# Hands On 2 Report

This brief report outlines the solution for the first hands-on assignment of the Competitive Programming course, focusing on implementing recursive traversals of binary trees in Rust.

# Requirements

The assignment asked to solve 2 subproblems:

## Min-Max

Design a data structure that would efficiently support queries and updates over a range

Specifically, given an unsigned integer list $A[1,n]$ (note the 1-index base) numbers we had to device a data structure that would support 2 operations:

- `Update(i,j,v)`: would update all values in $A[k]$ given $i \leq k \leq j \leq n$ with $\min(A[k], v)$
- `Max(i,j)`: would return the max value in $A[i \dots j]$

Given then $m$ queries, we would have to compute the result in (at most) $O(n + m\log(n))$ time.

## Is-There

Given $n$ segments (0-index based) in the range $[0, n-1]$, we would have to be able to compute queries of the form:

- `IsThere(i,j,k)` that given a range would return $1$ if a position $i \leq p \leq j < n$ existed, that was covered by exactely $k$ segments, $0$ otherwise.
  Given then $m$ queries, we would have to compute the result in (at most) $O(n + m\log(n))$ time.

# Min-Max Solution

A Segment Tree is a data structure well-suited for efficiently performing updates and supporting queries over a range. In my implementation, I chose to store only the maximum value in each node, as the primary operation to support was the `Max` query.

To support generic types, I explicitly handled default values and enforced appropriate trait bounds to ensure the necessary operations could be performed on the stored data. I used `Option<T>` to represent each node value, which allowes to clearly represent empty or uninitialized nodes (e.g., padding needed to keep the tree balanced) as `None`, and to explicitly manage subtree updates.

# Queries Implementation

The segment tree is implemented using a classic recursive approach, where each node represent a segment of the original array. Queries and updates are performed by traversing the tree recursively, depending on how the current node range relates to the query provided range.

## Max Query

To perform a range max query we recursively traverse the tree and return, based on the overlap between the query range:

- **Total overlap**: the query range is totally included in the nodes range, so we return the `max` value that the node holds
- **No overlap**: the nodes range doesn't overlap with the query range, so we return a `None` value that will get ignored
- **Partial overlap**: we recursively traverse the children and return the composition of their values. If they're both `None` we return the same, else return the `max` initialized value.

## Conditional Min Update

For the `Update(i,j,v)` operation we essentially want to overwrite values in a range only if they're greater than the given value. The recursive update differenciates as the case before in:

- **Total overlap**: the update range is totally included in the node's range, so we conditionally perform the update, updating the `max` value in the node or we discard it. If we update the `max` value we have to propagate the update to the eventual children but we'll see how to do that in the next section
- **No overlap** the update range doesn't overlap with the node's range, so the update is discarded.
- **Partial overlap**: we apply the update recursively to the node children

The conditional update, that can be discarded at any tree level ensures correctness while avoiding unnecessary traversals.

## Lazy Propagation Support

Some Segment Trees, depending on the *composability* of their supported operation (regarding the update of the tree state), can implement a neat optimization: *lazy propagation*. This allows update operations to be stored and computed *on demand*, while still mantaining *linear space complexity* relative to the number of elements the tree holds.

Lazy propagation introduces a second tree, that we call the *lazy tree*. We exploit memoization, storing pending updates that are only applied to the relative subtree *when absolutely necessary*, for example when a query or further update overlaps with the affected range.

The main idea is to store the update in the corresponding lazy tree node when the **query range** totally overlaps with the current subtree. *Propagation* to the children nodes will be performed by future queries or updates that require accessing those nodes.

> This reduces unnecessary computations when multiple updates affect the same range, or when intermediate results are not needed immediately.

### Performing Lazy Propagation

Efficient lazy propagations relies heavily on *operation composability* to keep updates consistent without heavily impact space usage.

The idea is that we can implement lazy propagation in a straightforward way when the update operation are composable with one another:

For example if we support `tree.add(i,j,v)` that adds a specified value to all nodes in a range we could have 2 successive range `add` and instead of propagating both immediately, we can update the lazy value for that node.

In this specific case case our main segment tree stores the `max` value in the specific range (specifically an `Option<T>` to avoid default values for padding). A generic `Update(i,j,v)` has to overwrite all values in the range that are greater than the provided value. Suppose we store the value `v` in a lazy node (for the update to be deferred). If a second update operation `Update(i,j,v1)` is requested (and the range overlaps correctly), we can conditionally discard the new update value `v1` if greater than the value stored, or overwrite it in the alternative case.

Going a little more in detail, this specific lazy propagation implementation works in 2 steps:

- **Storing for later**
  If the update range totally covers the current subtree
  1. Apply hte update to the current segment tree node
  2. Store the update in the lazy tree node
  3. Do not push lower, the update will be pushed later *only if needed*
- **Committing on-demand**
  If the update range partially overlaps with the current subtree
  1. Push down any pending updates in the current lazy node to its children before proceeding. When pushing:
     - *Overwrite* the existing value if the new update takes precedence
     - *Discard* the new update if redundant. The value in the lazy node is less than the one provided
  2. Recursively process the update on children

## Time Complexity and Space Usage

Segment trees, in the general case, are not implicit data structures (like Fenwick Trees), so they require additional space beyond the original data.

In this specific implementation, we introduce **padding** to ensure a fully balanced tree. Padding is proportional to the number of leaves: we round the number of elements up to the nearest power of 2. In addition to the padded leaves, we need to store internal nodes — specifically, an extra $n-1$ nodes.

On top of the main segment tree, we also include support for **lazy propagation**, which introduces a second tree (the *lazy tree*) of the same shape, storing single values per node.

With this layout, in the worst case, we reach a space usage proportional to $8n \sim O(n)$, assuming 2 memory words per node (one for value, one for lazy metadata).

The tree is built recursively for simplicity. Since this is a classic balanced divide-et-impera pattern with a constant-time merge step, **the time for tree build** is:

$$\Theta(n)$$

Each query — whether a `Max` query or an `Update` — in the worst case needs to follow a full path from root to leaf. Since the tree is balanced, each such path has height $\log(n)$.
Thus, **the time per query is**:

$$\Omega(\log(n))$$

Lazy propagation, while effective in reducing redundant updates, **does not improve the asymptotic complexity**, as we can construct worst-case sequences that always require pushing lazy updates. For example:

- `Max(i, i)`
- `Update(i, i, v)`

Both always take $\Theta(\log n)$ time since they partially overlap at each level until reaching the leaf.

Accounting for tree construction and posing $m$ queries, the overall complexity is:

$$T(n) = \Theta(n + (m \log(n)))$$

## Test Suite

To support automated testing and validation, I included a dedicated test module in the `lib.rs` file, along with helper functions for input parsing. The solution is organized around a generic `solve(...)` function, which takes:

- An array of initial values.
- An array of structured queries, represented using a custom `enum`.
  This design allows the `min_max` logic to be tested both programmatically (by directly supplying parameters) and via parsed test files.

I also created a `test_min_max` directory containing input and expected output files (the one attached with the requirements). A dedicated test function reads all test cases from this folder, parses the inputs and expected outputs, runs the `solve(...)` function, and compares the results.

# Is-There Solution

In this section I will discuss the first approach I adopted to solve this problem. I later found out that due to implementation specific details, the time complexity didn't fully match requirements and the space usage was too high. The second approach solves these problems.

In both these approaches we treat the problem as one of overlapping segments. In fact we can interpret it as follows

> having $n$ segments in the range $[0, n-1]$, the `isThere(i,j,k)` has to find out if there exists a point $p$ that is covered by exactely $k$ segment.

Since both segment count and segment coordinates are always less than $n$ (total number of segments), we start by computing the coverage for each integer in the range $[0, n]$

## Computing coverage

Given $n$ segments, we can compute the coverage of each point in a given range by using a handful of approaches of which most rely on prefix sums.

In the one I chosen, we start from a vector $\text{cov}$ that has $n$ cells initialized to 0. At the end of the procedure the vector will hold for every cell $v[i]$ the number of overlapping segments at the point $0 \leq i < n$.

We then process each segment that is denoted as a tuple `(start,end)`:

- we **add 1** to $\text{cov}[\text{start}]$
- we **subtract 1** to $\text{cov}[\text{end} + 1]$ (if in bounds with n else we ignore it)

After this we compute the prefix sum of $\text{cov}$ ($\text{cov}[\text{i}] := \text{cov}[\text{i}] + \text{cov}[\text{i-1}]$).

Certainly a more ideomatic approach would have been to use *Fenwick Trees* that are specifically designed to work with prefix sum arrays, but due to later casting of types and the fact that this approach would result in no better time nor space complexity (which is $O(n)$ in both cases), I chose for the first one.

## SegmentTree-Hash

Since we have to answer online range queries, my first approach was to use a particular Segment Tree that for each interval would store all possible $k$ coverages. Reading on this Codeforces blog post I decided to start from the segments given and compute for each point in the range $[0, n]$ the associated coverage. We would construct the Segment Tree from this coverage, storing `HashSet<T>` in the internal nodes to merge different coverages, and achieving $O(1)$ lookup time for each interval. This results in a average case performant approach due to the massive storage duplication.

To minimize the `HashSet<T>` merge time, I decided to use an `enum Coverage` for node values. This would store either a single value or a `HashSet<T>`. Due to padding to keep the tree balanced the actual nodes use an `Option<Coverage<T>>` wrapper to process uninitialized values.

## Making a query

Suppose we already build the tree, and in each node, we store all different overlapping counts for a given range, a `isThere(i,j,k)` query is straighforward. We perform a recursive procedure to find the node which ranges totally overlaps the query range, and we perform a `HashSet.lookup(k)`.

> Right now I'm skipping over some details that I'll cover in the next section. To be specific the `HashSet` lookup isn't always performed, in some cases we can perform a direct comparison, due to the nature of the `Coverage` node data-type. We can overlook this, since either hash based lookup and direct comparison are performed in constant time.

So, using this approach a query gets executed in $\Omega(\log(n))$ time as usual for Segment Tree range queries.

## Building the Tree

Due to the problem requirements, we can think at the tree as an immutable structure. This is important because handling updates to the ranges after construction would not have been easy.

Same as the other solution, the tree build process can be though out as a divide-et-impera, with a key difference in the Merge step.

We start by allocating enough space to hold the tree, initializing each node to `None`. Each recursive calls operates on a slice that has half the length as the one given in input. After the recursion fully unwraps, we start by initializing the leaves.

As before mentioned the data type is:

```
enum Coverage<T> {
    Val(T),
    Set(HashSet<T>)
}
```

This allows us to exploit some input patterns where lots of consecutive coverage counts are equal, making possible to store single values in the internal nodes as well as the leaves.

When merging the data of 2 children in a parent node we perform a standard merge that works as follows:

```
use Coverage::{Val,Set};

fn merge_children(left: Coverage<T>,right: Coverage<T>) -> Coverage<T> {
    match (left,right) {
        (Val(v),Val(v1)) => {
            if v == v1 { Val(v) } else {
                let mut set = HashSet::new();
                set.insert(v);
                Set(set)
            }
        }
        (Set(s),Val(v))|(Val(v),Set(s)) => {
            let mut set = s.clone();
            s.insert(v);
            Set(Set)
        }
        (Set(s),Set(s1)) => {
            let mut (to_fill,filled) = if s.len() > s1.len() {
                (s.clone(),s1 } else { (s1.clone(),s) }
            for &e in filler.iter() {
                to_fill.insert(e);
            }
            Set(to_fill)
        }
    }
}
```

As above mentioned we try to not resort to `HashSet<T>` nodes until absolute necessary.

## Space Usage and Time Complexity

It's not hard to see that if we start with $n$ distinct coverage points (worst case), each level of the tree stores the same number of points, in half the number of sets per level bottom-up.

This means that our space complexity is:

$$S(n) = O(n \cdot h) = O(n \log(n))$$

This is not ideal and it's the trade-off that allows us (with this approach) to keep a worst case $\Omega(\log(n))$ time per query.

To compute time complexity we only have to reason about time complexity in building the tree.

In the last section I briefly discussed about the merge step complexity, that *assuming constant time set insertions* allows us to build the tree in $O(n \log(n))$ time. This is a bit controversial, because due to hashing function probabilstic nature, we theoretically can account for constant time insertions only in the average case, but due to hash collisions it degrades to linear time in the worst.

Most standard hash-tables (sets) implementations, use open addressing as *collision resolution techniques* that use specific heuristics about the table load factor to count on constant time insertions. Even with this, in most cases tables have to be resized to work this way and this requires hash recomputation for all keys stored, resulting in worst case linear time.

As I spent some time to implement about this solution, I didn't want to give up on this approach and I though of some ways that I could use to support the constant time insertions needed to match the time complexity of the requirements.
As I discovered, hash procedures that support real constant time insertion exist. The most popular one is *cockoo hashing* that keeps 2 support hash tables and can move an item from the first to the second table, if a collision. Even though theoretical, they're not widely used, since they don't adapt well with the modern distibuted cache model of general purpose architectures. The only case I found where they were used were real-time applications with strict time requirements.

The not high popularity doesn't enforce languages to support them, like rust.

Since this solution relies on rust stdlib HashSets and it would have been quite overkill and not worth it to build a constant time insertion hash map myself, we have to upper bound the tree building step as:

$$T_{\text{build}}(n) = \begin{cases} 1 \mid n = 1 \\ 2T\left(\frac{n}{2}\right) + O(n^2) \end{cases} \implies O(n^2)$$

So the total time complexity for solving $m$ queries is:

$$T(n) = T_{\text{build}}(n) + m \cdot \Omega(\log(n)) = O(n^2 + m \log(n))$$

## Sorted Lookup Table

For this approach I completely turned around how precomputation is stored. As a refresher, given $n$ segments a `isThere(i,j,k)` has to return true if there exist a point $p \in [i, j]$ so that $p$ is covered by exactely $n$ segments.

Since given an interval $[i, j]$ several $p$ points that map into different $k$ overlap counts can exist, so for the same query interval, different $k$ can result in a positive query,

The strategy in the other approach was to precompute the overlap count for each $i \in [0, n]$ then store them in hash-sets, one per range of the segment tree. We later could supply the range for the query and lookup in constant time to the `HashSet` of the corresponding range.

> It's later explained how this approach fails to meet time complexity requirements, due to how we precompute and store data.

Instead of storing all possible $k$ for every range, since they are not unique, I though of using a 2 level lookup table.

### Lookup Table Specifics

As the last case, we construct the *coverage* vector that for each point in the range $[0, n]$ stores the unmber of segments that cover that point. Note that we can't assume that the vector is sorted, nor that coverage items are unique. The only thing we can say is that every coverage item $\forall i : \text{cov}[i] \in [0, n]$, since in the worst case all $n$ segments overlap each other.

We can construct our lookup table, to have $n + 1$ cells, one for each possible coverage count and contain the list of the points that match that count.

Let's see an example:
Suppose we start with 5 segments:

- `[(0,0),(0,4),(1,1),(1,2),(1,3)]`.
  We compute the coverage for each point in the range $[0, 4]$ as
- `[2,4,3,2,1]` (i.e. the point `i = 0` is covered by `(0,0)(0,4)`)

We construct out lookup table as:

| coverage | point |
|---|---|
| 0 | $\varnothing$ |
| 1 | $\{4\}$ |
| 2 | $\{0, 3\}$ |
| 3 | $\{2\}$ |
| 4 | $\{1\}$ |

Since we have the guarantee that each vector of points contains all points with that specific coverage count, we can keep the vectors sorted and perform a binary search to look if a there exist a point that is contained in the query interval.

### Query Details

Assume we already built the lookup table that for every position $i$ contains all points of the range $[0, n]$ that are covered by exactely $i$ semgnets

Assuming a `isThere(i,j,k)` query we process it in this way:

- get all points covered by $k$ segments accessing $\text{lookup[k]}$
  - if no points then return `false`
  - else perform a binary search for the first value $x$ that falls under $[i, j]$. If $x$ exists then it's one of other possible $p$ points, so return `true` else return false.

### Space Usage and Time Complexity

The space required to answer $m$ queries is the space we need to preprocess the $n$ segments and build the lookup table.

1. To compute the *coverage* array we need $O(n)$ for the prefix-sum array (or Fenwick tree)
2. The lookup table outer level contains at most $n$ cells (one for each different coverage count)
3. For each lookup table ve store a vector. All lookup vectors only distribute store all values in the range $[0, n]$ so we need $O(n)$ total vectors slots.

With all this considered we find that $S(n) = \Theta(n)$

> Further optimizations can be implemented as using a `BtreeMap` for the outer lookup level that would automatically not map empty coverage counts, but giving a logarithmic time lookup access time, or coordinate normalization allowing us to discard outer empty coverage counts. All these would not impact the query time complexity.

To compute the total time complexity in answering $m$ queries, we split it in:

- **Building Lookup**
  1. $O(n \log(n))$ to build the sorted coverage range
  2. $O(n)$ to map each point in the respective lookup vectors
- **Query Time**:
  1. $O(1)$ to access the `lookup[k]` cell
  2. $O(\log(n))$ to perform the binary search of an appropriate $p$ point

with this we find, accounting for $m$ queries:

$$T(n) = O((n + m) \log(n))$$

## Test Suite

To support automated testing and validation, I included a dedicated test module in the `lib.rs` file, along with helper functions for input parsing. The solution is organized around two generic functions

- `solve_segment_tree()` : that implements the solution of the first approach
- `solve_binary_lookup()` : that implements the solution of the second approach
  Both of them take as parameters:
- An array of initial values.
- An array of structured queries, represented using custom types `is_there::Segment`,`is_there::Query` .
  This design allows the `min_max` logic to be tested both programmatically (by directly supplying parameters) and via parsed test files.

I also created a `test_is_there` directory containing input and expected output files (the one attached with the requirements). A dedicated test function reads all test cases from this folder, parses the inputs and

expected outputs, runs the `solve_...(...)` methods, and compares the results.