Min max and *queries and operations*: solutions and complexity

Giuseppe Muntoni

Contents

1 Introduction

In this report are discussed the solution of two competitive programming problems, *min max* and *queries and operations*. The solution of min max uses the segment tree data structure, so also the implementation of the segment tree is explained.

2 Segment tree implementation in Rust

The segment tree is implemented using two references combined.

The first explains how to implement an efficient segment tree in C++, using a vector of size $2 * N$, provided an array of size N. This implementation supports arrays of sizes that aren't a power of two, and range updates with lazy propagation [1].

The second explains how to generalize a segment tree to support any possible operation [2].

The idea is that the user of the segment tree, must implement:

- A type S which represents the values of the nodes, and the array.
- A type F which represents the lazy nodes.
- A function *op* to calculate the value of a subarray, based on the two subarrays of this subarray.
- An identity function that returns the identity element of *op*, useful for example in case of no overlaps.
- A compose function, useful in case of lazy propagation when for two updates there is an equal total overlap. Instead of saving each lazy update and then applying them when needed, which creates linear time complexity for each lazy value to propagate, the values are composed immediately, leaving the application constant.
- An apply function that applies the lazy value on a node.

2.1 Range update

The *update* function, executes a range update lazily.

Starting from the leaves the function updates every node that is a total overlap.

If the left extreme l is the right child of its parent, for sure the parent node is not a total overlap, so this node is updated lazily. Lazily means that the value is applied on this node and the lazy value is saved in order to be propagated to the children when needed. In this case, l is also incremented by one and divided by two in order to go to the parent.

The same reasoning applies to the right extreme.

Finally, from the total overlaps, the modified value is propagated upward in order to maintain consistent values on the nodes in the path from the root to the total overlaps.

2.2 Range query

The range query initially propagates lazy values in the path from the root to the extremes of the range, so also in the total overlaps in the middle. In this way is ensured that each node that is a total overlap, so used to compute the result, is updated.

Subsequently, the result is initialized with the identity, and every time the left extreme l is the right child of its parent, for sure the parent node is not a total overlap, so the result is updated with the output of the function *op* between the actual result and this node. The same reasoning applies to the right extreme.

3 Min max

3.1 Solution

The solution uses the segment tree implementation explained above.

Each node of the segment tree contains the maximum of the subarray that it represents.

For each $update(i, j, T)$ query, the update with the minimum is done lazily. This means that every time there is a total overlap, the update stops, applies the update to this node, and saves the lazy value T. The application of T in the total overlap nodes is made by updating the value of the node with the minimum between T and its value. In this way, if T is larger than the actual maximum, no nodes must change in the range, and T is discarded. Otherwise, if T is less than the actual maximum, certainly the maximum becomes T, so the node is updated with T. Also, some other nodes greater than T become T, eventually.

For each $max(i, j)$ query, simply the max is reported, but paying attention to propagate the lazy values encountered in partial overlaps, down to the tree.

3.2 Complexity

The solution runs m operations of update or query on a segment tree, that runs both in $O(log(n))$ time. The time to construct the segment tree is $\Theta(n)$, so the overall time complexity is $\Theta(mlog(n) + n)$.

The additional space used is for the segment tree, i.e., $\Theta(n)$, and for the array of maximums, i.e., $\Theta(m)$.

4 Queries and operations

4.1 Solution

For this problem, the segment tree is not strictly necessary. It can be solved only using static prefix sum.

The queries can be executed in any order, so an array of counters is created. This array contains, at index i, the number of times that $O[i]$ must be executed. This array is constructed using the static prefix sum.

For each operation that must be executed $k \geq 1$ times, an increment of $k * d$ is executed at index l, and an increment of $-(k + d)$ is executed at index $r + 1$, in a new array F initialized with all elements at zero. After this, the prefix sum of F is computed.

Finally, to report the updated array, each element i is incremented with $F[i]$.

4.2 Complexity

Initially, the algorithm constructs the array of counters of the operations, scanning the array of k queries. This costs $\Theta(k)$ in time, and $\Theta(m)$ in space, considering m operations. Next, it does the static prefix sum of this array, which costs $\Theta(m)$.

For each operation, executes two (or zero) updates, one in position l, and one in position $r + 1$ of F, that costs constant time, so $\Theta(m)$ overall.

Finally, to report the updated array, the prefix sum of F is executed, which cost is $\Theta(n)$, and the initial array is updated scanning A, in $\Theta(n)$ time.

Supposing that $k \leq n$ and $m \leq n$, the overall time complexity is $\Theta(n)$, and the space complexity is also $\Theta(n)$.

References

[1] Efficient and easy segment trees. https://codeforces.com/blog/entry/18051.

[2] Generalizing Segment Trees with Rust. https://codeforces.com/blog/entry/68419.