Handson 2

Competitive Programming and Contests

Silvio Martinico

November 20, 2022

Abstract

In this report we present the resolution of two problems: *Min and Max* and *Queries of operations*.

The solution of the first problem relies on *Segment Trees* and *Lazy Propagation*, while the solution of the second one exploits the *Difference Array*.
The solutions are implemented in `Rust` and have respectively a time complexity of $\Theta(n + m \log n)$ (where n is the size of the input vector and m is the number of queries) and $\Theta(n)$ (where n is the size of the input vector)).

Contents

1 Min and Max problem

Problem: Given an array $A[1,n] \in \mathbb{Z}_+^n$ such that $A[i] \leq n \; \forall \; i = 1, \ldots, n$. Answer m queries within a time complexity of $O((n+m)\log n)$. The queries are of two types:

- *Update(i,j,T)*: Replace $A[k]$ with $\min\big(A[k], T\big) \; \forall \; k \in [i,j]$;
- *Max(i,j)*: Return $\max\limits_{i \leq k \leq j} A[k]$

1.1 Data structure

In order to understand which is the most suitable data structure for this problem, we have to analyze what we need: we need to store an array and to solve two range operations on it: find the maximum of a given range and update a range. Since the queries are not offline and the two types of queries could alternate, the best thing to do is to minimize the time required by both queries.

The data structure we have chosen is the *Segment Tree*, which has a build cost of $\Theta(n)$.
The segment tree easily support the range max operation in $O(\log n)$ time, because the number of operations at each node is constant and the search in the tree stops at totally overlapping nodes, i.e. these nodes which range is totally covered by query range; we remind that the number of totally overlapping nodes for a given range is $O(\log n)$.
Regarding the update operation, a classic segment tree does not support it in $O(\log n)$ time. However, we can opt for the *Lazy Propagation* technique: we look for totally overlapping nodes as in the max query and, when we find them, we update their values and defer updating for their children until we visit them in the future (for another query). In this way, like in the max query, we stop at totally overlapping nodes, reaching a time complexity of $O(\log n)$. The lazy propagation is usually represented with a second segment tree, called *Lazy Tree*, where we store lazy values, but, as we will see later when we will describe the implementation, we do not actually need another tree as we can store lazy values in the main tree nodes.

1.2 Implementation

The segment tree for an input array of size ℓ was stored in an array of size $2^{k+1} - 1 = \sum\limits_{i=0}^{k} 2^i$, where k is a positive integer such that $2^{k-1} < \ell \leq 2^k$, or, alternatively, $k = \lceil \log_2 m \rceil$. Let's call $2^k = n$ for simplicity. The positions of the array which go from $\ell+1$ to n were filled with the value -1 because, having as merging operation for the nodes the function max and having only positive integers, the padding values vanish as soon as possible in the construction of the tree and never influence the functioning of the structure. In this way, for a node in a position i of the array which represents the segment tree, its children are in positions $2i$ (left) and $2i+1$ (right) (or $2i+1$ and $2i+2$ for 0-indexing).

For storing the nodes we created the structure `Node`, which has four attributes: the value inside the node (`key`), the interval covered by the node, given by the two attributes `left_interval` and `right_interval` and the value in the corresponding node of the lazy tree, `lazy_key`. In fact, the lazy tree was stored within the segment tree, by just adding the attribute `lazy_key` (initialized to -1) to the Node structure. Thus, the segment tree was stored as a structure with two attributes: an array of `Node` of length $2n-1$, called `seg_tree`, and the size of the array, called just `n`.

The update query was solved through the lazy propagation: it works like a standard update on a segment tree but, whenever we reach a total overlapping node, we eventually update this node and

then jump to the lazy tree in order to update the lazy values of its children (if the current node is not a leaf node). As first operation of the update function, we check for pending updates, i.e. we check if the node we are visiting has a lazy value and, if so, we update the current node and propagate to its children. In this way, we do these updates only when the nodes which require an update are visited, avoiding to visit too much nodes at each update.

The max query works exactly as the classic max query, the only different thing is that, again, when the function is called, the first thing it does is to check for pending updates on the current node.

1.3 Complexity analysis

The space complexity is clearly linear, because the only thing we store is the segment tree, which is stored as an array which length is less than 4 times the length of the original array (at most a 2 factor for reaching the next power of two and then the size is doubled for creating the actual vector). Furthermore, the size of each array cell is constant (it's a `Node`, so it has 4 fields, two integers and to `usize`).

The time complexity is divided in two parts: the segment tree building and the queries solving.

- The cost for building the segment tree is linear ($\Theta(n)$): in fact, we just take the original array, we extend it with the padding values and then we iterate over it from the end to the start, building the tree with a bottom-up approach: each node is created in constant time by just taking the max value of its children as `key`, the left interval of the left child as `left_interval` and the right interval of the right child as `right_interval`.
- The cost for each query is $O(\log n)$. As we said before, we stop at totally overlapping nodes and, in each node we visit, we just do a constant number of operations: for both queries we check for pending updates (by just checking if the lazy value is -1) and eventually we update the lazy values of its children; for the max query we just compare the interval of the node with the interval of the query and the value of the current maximum with the key of the node, while for the update we again compare intervals and the updating value T with the key of the node. Since the number of total overlapping nodes for a specific query is proportional to $\log n$, we have that the cost of a query is $O(\log n)$.

Summing up, we have a total cost of $\Theta(n)$ for the construction and $O(m \log n)$ for the queries, which gives us a total asymptotic time complexity of $O(n + m \log n)$.

2 Queries of operations problem

Problem: Given an array $A[1,n] \in \mathbb{Z}^n$ and an array $O[1,m]$ of operations, such that each operation is a triple of the form $\langle l, r, d\rangle$ (where $1 \leq l \leq r \leq n$), which consists of adding $d \in \mathbb{Z}$ to each element of the interval $A[l,r]$. Given k queries in the form $\langle a, b\rangle$ (where $1 \leq a \leq b \leq m$), we want, for each of them, to apply the operations in $O[a,b]$ to A. The aim is to report the updated array A (after all the k queries) within a time complexity of $\Theta((k+m)\log n + n)$. We assume that $k, m \leq n$.

2.1 Resolution of the problem

This problem, as the first one, could be solved using segment trees. However, there is a more efficient and simpler solution. The idea is that of using the *Difference Array*.

Given an array $A[1,n]$, its difference array is defined as an array $D[1,n]$ such that $D[1] = 0$ and $D[i] = A[i] - A[i-1]$ for $i = 2, ..., n$. This array could be exploited for updating ranges in constant time. In fact, once we have build it, adding a value d to a range $[i,j]$ in A consists of just two operations on D: adding d to $D[i]$ and subtracting d to $D[j+1]$ (if $j < n$, else we do just the first operation). Once we have done our updates on D, we can easily reconstruct the updated A starting from D, because it's easy to observe that $A[0] = D[0]$ and $A[i] = D[i] + A[i-1]$ for $i = 2, ..., n$.

This clearly solves the problem of applying the operations in O to A, but how can we solve our main problem? If we directly apply this idea to our problem, for each query in Q (an array containing the k queries), in the worst case we do m updates in a constant time each, so the total time cost would be $O(km + n)$ ($O(n)$ is the cost for building the difference array and reconstructing the original array). So, we need something else. Here comes another idea: the queries in Q can be seen as updates on O. In particular, we can see a query $\langle a, b\rangle$ as an operation $\langle a, b, 1\rangle$ on an array $C[1,m]$ (initialized with zeros) which counts how many times we have to execute each operation in O. In this way, we can use again the difference array, this time on the array C, and update it in constant time for k times.

Once we get the updated C (by reconstruction from its difference array), we can apply each operation $O[i]$ the number of times indicated by $C[i]$. Clearly, we don't really apply an operation $C[i]$ times, we just multiply its value d_i by the corresponding counter $C[i]$.

2.2 Complexity analysis

The space complexity is $\Theta(n)$ since $k, m \leq n$ and we store the four arrays A, O, Q, C and their corresponding difference arrays, which have the same size of their original arrays.

Regarding the time complexity, we proceed by steps:

1. Build $C \longrightarrow \Theta(m)$
2. Build D_C, the difference array of $C \longrightarrow \Theta(m)$
3. Update D_C for each query $q \in Q \longrightarrow \Theta(k)$
4. Reconstruct the updated C from the updated $D_C \longrightarrow \Theta(m)$
5. Build D, the difference array of $A \longrightarrow \Theta(n)$
6. Update D for each operation $O[i] = \langle i, j, d\rangle$, multiplying d by its counter $C[i] \longrightarrow \Theta(n)$
7. Reconstruct the updated A from the updated $D \longrightarrow \Theta(n)$

Summing up, it is clear that the asymptotic time complexity is $\Theta(n + m + k) = \Theta(n)$.