Segment Tree Range Updates

Executive Summary

CS 4501: Advanced Algorithms and Implementations,
Group 05 Final Project

1 Introduction

When we first introduced segment trees earlier in the semester, we had the overarching goal of allowing range queries and single-value updates on a list of integers $A = \{a_1, a_2, \ldots, a_n\}$ in $O(\log n)$ time and $O(n)$ space. We achieved these runtimes by constructing a binary tree, where each node represented a segment of the list and stored the query value corresponding to that segment.

In our original implementations, we only supported update queries of the form $update(i,\ x)$ for some single index $1 \leq i \leq n$ specifying which value $A[i]$ to update. Since we introduced seg-trees with range queries in mind, the next natural question to ask is whether we can support update queries of the form $update(\ell,\ r,\ x)$, where $1 \leq \ell \leq r \leq n$ specify a *range* of values $A[\ell \ldots r]$ to update.

More specifically, we explore three different seg-tree implementation variants that support different combinations of range updates and reading queries:

(i) The first seg-tree implementation allows for ranged addition updates, where $update(\ell,\ r,\ \delta)$ adds δ to each value in $A[\ell \ldots r]$ alongside a simple $get(i)$ reading query to return $A[i]$.

(ii) The second supports ranged assignment updates, where $update(\ell,\ r,\ x)$ sets each value in the range $A[\ell \ldots r]$ to the common value x, alongside the same simple $get(i)$ read query.

(iii) The final implementation supports the same ranged addition updates $update(\ell,\ r,\ \delta)$, but now with ranged reading queries of the form $max(\ell,\ r)$, returning the maximum value in the range $A[\ell \ldots r]$.

The primary challenge associated with supporting ranged queries like these is figuring out how to maintain the overall $O(\log n)$ runtime that we started with. We discuss how to approach this challenge next.

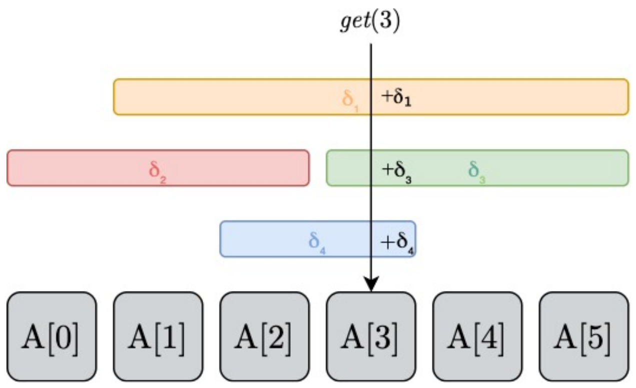
2 Approach

The key concept in supporting these new queries while keeping an efficient runtime is the concept of "lazy propagation." Broadly speaking, this is the idea that instead of working our way all the down the tree to update every single node that's been affected by a range update, we can instead stop at an earlier point and only update deeper nodes if and when we need to access them in the future, hence the "laziness." This approach will essentially give us a way to "update" entire ranges of A all at once and in constant time, much like we normally could for a single value of $A[i]$. The details for each of the three variants are laid out below.

2.1 Variant (i): Ranged Addition Updates with Point Queries

The first variant makes use of lazy propagation by deferring the addition updates until a *get* query is made. In each leaf node in our tree, we simply store the corresponding array value. In each non-leaf node, the value we store will correspond to the amount of pending additions to apply to the range of that node. For example, if $update(0, n-1, \delta)$ is called to add to the entire list, we simply set the root node's value to δ without touching any children nodes. If an update query's range doesn't match perfectly with the range of a segment, we leave the current node unchanged and recursively move down in the tree until finding children for which the range does match.

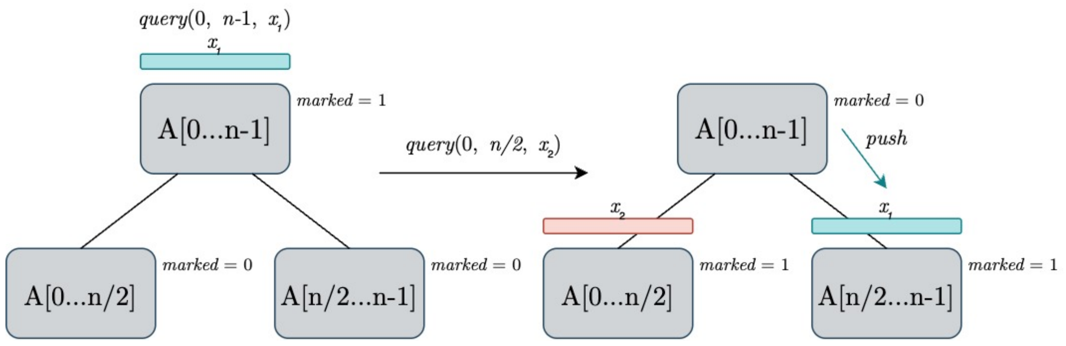
To perform *get* queries, we can just travel down the tree to find the specified index and add up all the pending additions we find along the way:



2.2 Variant (ii): Ranged Assignment Updates with Point Queries

Now we want to support updates that assign a value to an entire range alongside the usual *get* query. To support this, we add a boolean field to our nodes, *marked*, that indicates whether the given node's range has been covered entirely by a common value or not. If a node is marked, this common value will be stored in the node's value field. This allows us to be "lazy" with our updates: to set the value for a node's range, we can just mark it and not bother updating its children, even though in reality they should also be updated.

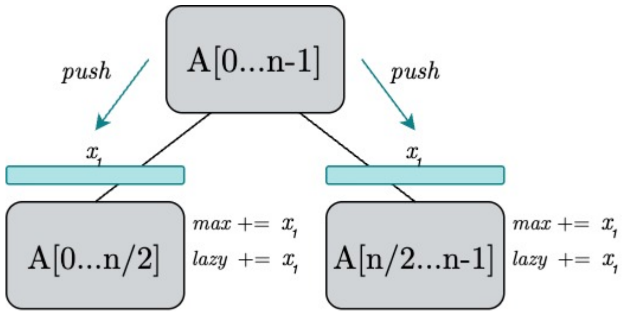
But what if a query doesn't fall precisely within the range of a node? We need to ensure that when we travel down the tree, we don't lose any information if the current node is marked. To do so, any time we recurse down the tree, we "push" down any marked data as necessary. For example, say two assignments $update(0, n-1, x_1)$ and $update(0, n/2, x_2)$ are called successively. The first call would mark the root node, set its value to x_1, and return. Now, since we only *lazily* set this range of values, on the second call we need to propagate the update down to the children before modifying further. So, we mark the root's children with its value, then reset set the root's *marked* and *value* flags. We then query recursively on the left child to update its range with x_2:



To perform *get* queries, we take a similar approach. To find a given index, the overall recursive structure to travel down the tree to find the desired value remains standard. But as we're travelling, we need to push any lazy updates that haven't been applied to a node's children before recursing further. This way we ensure we return a value that's up-to-date with all queries made thus far.

2.3 Variant (iii): Ranged Addition Updates with Ranged Max Queries

Finally, we want to design a segment tree to support ranged addition, but this time our reading query will return the maximum from a specified range. The maximum from a node's range is what will be stored in its *value* field. To handle addition updates, we combine the two previous approaches. We'll add a new field to our nodes, *lazy*, that stores the addends on the node's range that haven't been propagated down to its children yet. We perform the same lazy propagation as before, where we push these addends down to our children only when traversing further down in the tree, either on *update* or *max* calls. When performing the push, we need to add to our children's values *as well as* their own *lazy* fields:



The rest of the approach is very similar to the previous.

3 Implementation

We provide implementations for the update and read queries for each of the above three variants. The rest of the implementations consist of relatively standard segment tree structure, so we limit ourselves to implementing only the updates and reads. To perform the required queries, we simply need to call the implementations below on the root node of the corresponding seg-tree with the specified range arguments.

3.1 Implementing (i)

The update function is just as we described above—if the query range matches the node's, simply update the node's value. If not, call update on both children.

```
$update(node,\ \ell,\ r,\ \delta)$:
    if $\ell > r$:
        return
    if $\ell, r$ match $node$ range:
        $node.val \mathrel{+}= \delta$
    else:
        $mid = (\ell + r)/2$
        $update(node.leftChild,\ \ell,\ min(r,\ mid),\ \delta)$
        $update(node.rightChild,\ max(\ell,\ mid+1),\ r,\ \delta)$
```

For *get*, we travel down the tree recursively and add each node's value as we go, as described previously.

```
$get(node,\ i)$:
    if $node$ is leaf:
        return $node.val$
    if $i \le node.leftChild.r$:
        return $node.val + get(node.leftChild,\ i)$
    else:
        return $node.val + get(node.rightChild,\ i)$
```

3.2 Implementing (ii)

To implement variant (ii), in addition to the two query functions, we must write a method to perform the push operation that was described previously.

```
$push(node)$:
    if $!\,node.marked$:
        return
    $node.leftChild.val = node.val$
    $node.rightChild.val = node.val$
    $node.leftChild.marked =$ true
    $node.rightChild.marked =$ true
    $node.marked =$ false
```

We then use this method in our query implementations. For *update*, if the range matches the node's, we set the value and mark the node. Otherwise, we push any values down to our children, then make the necessary recursive calls.

```
$update(node, \ell, r, x)$:
    if $\ell > r$:
        return
    if $\ell, r$ match $node$ range:
        $node.val = x$
        $node.marked =$ true
    else:
        $push(node)$
        $mid = (\ell + r)/2$
        $update(node.leftChild,\ \ell,\ min(r,\ mid),\ x)$
        $update(node.rightChild,\ max(\ell,\ mid + 1),\ r,\ x)$
```

Similarly, in *get*, we must also make a call to *push* before travelling down the tree. This time, we're not keeping track of any additions as we go.

```
$get(node, i)$:
    if $node$ is leaf:
        return $node.val$
    $push(node)$
    if $i \leq node.leftChild.r$:
        return $get(node.leftChild,\ i)$
    else:
        return $get(node.rightChild,\ i)$
```

3.3 Implementing (iii)

As with the previous variant, (iii) also requires a push function. This time, instead of marking the children, we must update their stored and lazy values by adding the current node's lazy value.

```
$push(node)$:
    $node.leftChild.val \mathrel{+}= node.lazy$
    $node.rightChild.val \mathrel{+}= node.lazy$
    $node.leftChild.lazy \mathrel{+}= node.lazy$
    $node.rightChild.lazy \mathrel{+}= node.lazy$
```

Once again, we now use this function in the queries. The *update* function has almost an identical structure as before, but now since each node stores the ranged maximum, we have to merge each node's children values into its own after updating. We also need to make sure we update each node's lazy value accordingly.

```
$update(node,\ \ell,\ r,\ \delta)$:
    if $\ell > r$:
        return
    if $\ell, r$ match $node$ range:
        $node.val$ += $\delta$
        $node.lazy$ += $\delta$
    else:
        $push(node)$
        $mid = (\ell + r)/2$
        $update(node.leftChild,\ \ell,\ min(r,\ mid),\ \delta)$
        $update(node.rightChild,\ max(\ell,\ mid + 1),\ r,\ \delta)$
        $node.val = max(node.leftChild.val,\ node.rightChild.val)$
```

The maximum query is just a standard segment tree range query, just with an added call to *push*.

```
$query(node,\ \ell,\ r)$:
    if $\ell > r$:
        return $-\infty$
    if $\ell, r$ match $node$ range:
        return $node.val$
    else:
        $push(node)$
        $mid = (\ell + r)/2$
        $leftMax = query(node.leftChild, \ell, min(r, mid))$
        $rightMax = query(node.rightChild, max(\ell, mid + 1), r)$
        return $max(leftMax, rightMax)$
```

4 Programming Challenge Summary

The programming challenge asks students to write a program to support two different ranged queries on a list of integers $A = \{a_1, a_2, \ldots, a_n\}$ given as input. The first is a reading query of the form $GCD(\ell, r)$ that should return the greatest common divisor of the integers in the specified range, that is, $\gcd(a_\ell, \ldots, a_r)$. The second is an update query of the form $ADD(\ell, r, \delta)$ that should add δ to each of $a_\ell, \ldots, a_r$ in A. The ADD query is precisely the same as the *update* query discussed from the first and third segment tree variants above.

At first glance, it seems as if one might be able to just implement a seg-tree similar to the third variant from above, with the GCD query in place of *max*. But the key twist with this new query combination is that the GCD query *is not compatible* with the normal lazy propagation approach. Before, we were able to push down the addition updates to a node's children by simply adding to a child's stored value. This was because, after an addition of δ on a range, the new maximum in that range *will always be the old maximum plus* δ. But now, there is no way to easily calculate the new greatest common divisor across a range after an addition update.

We believe this programming challenge will not only test student's basic understanding of range updates by requiring them to implement a structure that supports such queries, but because of the aforementioned twist, will require them to understand the idea of lazy propagation at a deeper level. Many times, it's just as important to understand why an algorithm *doesn't* work as it is to understand why it does. Working through the issues that arise with the GCD query will ideally equip a student such an understanding with respect to lazy propagation by the time they have found a solution.

5 Programming Challenge Key Ideas

As mentioned above, simply implementing a seg-tree similar to the third variant from before will not suffice for the challenge. The key issue is that there is no way to easily recalculate the greatest common divisor over a range after an update using only the previous g.c.d. and the addition value:

$$\gcd(a_\ell, a_{\ell+1}, \dots, a_r), \quad \delta \quad \overset{?}{\Longrightarrow} \quad \gcd(a_\ell + \delta, a_{\ell+1} + \delta, \dots, a_r + \delta).$$

This leaves us in a tough spot. It seems as if there's no way to allow for addition updates without needing to iterate over the entire range to recalculate the new g.c.d., violating our precious $O(\log n)$ complexity. The key idea is to not just use a single seg-tree to accomplish the task, but instead two.

Since we have no way to recalculate the g.c.d. after a range update, we must somehow find a way to try and make these calculations *independent of the range updates.* Notice that, when δ is added to some range $a_\ell, \dots, a_r$, the *difference between consecutive integers remains unchanged.* That is,

$$(a_i + \delta) - (a_{i+1}\delta) = a_i - a_{i+1} + \delta - \delta = a_i - a_{i+1}.$$

This is the key idea at the heart of the solution, along with the identity

$$\gcd(a_\ell, a_{\ell+1}, \dots, a_r) = \gcd(a_\ell, a_{\ell+1} - a_\ell, a_{\ell+2} - a_{\ell+1}, \dots, a_r - a_{r-1}).$$

If we create two different segment trees, $Seg1$ that stores the original list A and supports the usual *update*(ℓ, r, δ) and *get*(i) queries, and $Seg2$ that stores the *differences* between elements of A and supports a *gcd*(ℓ, r) query on these differences as well as the usual single-value *update*(i, x) query, we can craft a solution. To answer a $GCD(\ell,\ r)$ query, we simply return

$$\begin{aligned}
\gcd(a_\ell, a_{\ell+1}, \dots, a_r) &= \gcd(a_\ell, a_{\ell+1} - a_\ell, a_{\ell+2} - a_{\ell+1}, \dots, a_r - a_{r-1}) \\
&= \gcd(a_\ell,\ \gcd(a_{\ell+1} - a_\ell, a_{\ell+2} - a_{\ell+1}, \dots, a_r - a_{r-1})) \\
&= \gcd(Seg1.\mathit{get}(\ell),\ Seg2.\mathit{gcd}(\ell, r-1)).
\end{aligned}$$

So, we only need two calls on our seg-trees to perform the GCD query. To deal with updates, we can let the *update* query on $Seg1$ do its job as normal, but we also need to update $Seg2$. As mentioned above, all differences within the query range will remain unchanged, but we still need to consider the differences on the boundary. That is, $a_\ell - a_{\ell-1}$ and $a_{r+1} - a_r$. These differences will change by δ and $-\delta$, respectively. But, these will be the only changes! So, to perform updates, we only need the following calls:

$$Seg1.\mathit{update}(\ell, r, \delta), \quad Seg2.\mathit{update}(\ell - 1, \delta), \quad Seg2.\mathit{update}(r, -\delta).$$

Each of the above calls are logarithmic, so our solution will also have an overall runtime of $O(\log n)$.

As far as difficulty, we believe the hints we have provided in the handout are sufficient in getting students on the right track. The hard part of the challenge is coming up with the idea. Once that's done, the implementation is quite simple.

6 Conclusion

Through this project, we explored how to extend segment trees beyond their classic use cases by supporting a variety of range update and query operations efficiently using lazy propagation. Each of the variants—ranged addition with point queries, ranged assignment with point queries, and ranged addition with range maximum queries— highlighted different nuances in lazy propagation, from marking and pushing values to merging updates during traversal. Lastly, the programming challenge we proposed introduces an edge case where lazy propagation fails to apply cleanly, reinforcing the importance of understanding both the strengths and limitations of these new techniques.