Segment Tree Guide

by fbrunodr

April 21, 2023

1 Introduction

Guide to use Segment Tree implemented in https://github.com/fbrunodr/CompetitiveProgramming.

2 Frozen Segment Tree

Frozen Segment Trees do not allow updating the internal data. They are used to perform operations in continuous arrays relatively fast. It is useful whenever the inverse operation is non-existent or impractical.

More formally, let $A = \{a_i \in T, i \in [1, n]\}$ be an array of type T. Let $\oplus$ be a binary operator over elements of T. Frozen Segment Trees answer queries in the following format:

$$Q(i, j) = a_i \oplus a_{i+1} \oplus a_{i+2} \oplus ... \oplus a_{j-1} \oplus a_j$$

The operator $\oplus$ must be associative for the Frozen Segment Tree to work. It does not need an inverse to exist, nor an identity element.

In other words, T and $\oplus$ must form a Semigroup (https://en.wikipedia.org/wiki/Semigroup).

If T and $\oplus$ forms a Group (https://en.wikipedia.org/wiki/Group_(mathematics)) and the inverse can be found fast, you should use a cumulative array instead of this data structure, like the following:

```
cumulative = vector<T>(n);
cumulative[0] = A[0]
for(int i = 1; i < n; i++)
    cumulative[i] = op(cumulative[i-1], A[i]);

auto query = [&cumulative](int i, int j){
    if(i == 0) return cumulative[j];
    return inv(cumulative[j], cumulative[i-1]);
}
```

Here, `op` stands for $\oplus$ and `inv` stands for the inverse of $\oplus$.

2.1 Space Complexity

In my implementation, I keep an array for the vertices of the tree that takes 4 times the space of the original array. Also, I keep a copy of the original array, to allow for $O(1)$ element retrieval. So my implementation takes about 5 times the space of the original array. Hence, it is $O(n \cdot \texttt{sizeof}(T))$.

The space complexity of the cumulative table is also $O(n \cdot \texttt{sizeof}(T))$, but it's only as large as A.

2.2 Time Complexity

Let T_{op} be the time complexity of $\oplus$. The time complexity to build the Frozen Segment Tree is $O(n \cdot T_{op})$. The time complexity of each query is $O(log(n) \cdot T_{op})$.

The time complexity to build the cumulative table is $O(n \cdot T_{op})$. The time to perform each query is $O(T_{inv})$, where T_{inv} is the time to perform the inverse of $\oplus$.

3 Lazy Segment Tree

If the data in the array is updated between queries, you may be able to use the Lazy Segment Tree. Notice the cumulative array becomes useless or it takes $O(n \cdot T_{up})$ for each update, where T_{up} is the time to update each element. Anyway, let $\otimes$ be the update operator. Let's define another query in the following format:

$$U(i, j, u) : a_x \leftarrow a_x \otimes u \; \forall x \in [i, j]$$

If you take a closer look at how the update works, you will see it only works if:

$$(a_i \otimes val) \oplus (a_{i+1} \otimes val) \oplus (a_{i+2} \otimes val) \oplus \ldots \oplus (a_j \otimes val) = (a_i \oplus a_{i+1} \oplus a_{i+2} \oplus \ldots \oplus a_j) \otimes val$$

It's then necessary and sufficient that $(u \otimes val) \oplus (v \otimes val) = (u \oplus v) \otimes val$, $\forall u, v \in T$. It is easy to see why this is necessary, just make $j = i + 1$ and we have:

$$(a_i \otimes val) \oplus (a_{i+1} \otimes val) = (a_i \oplus a_{i+1}) \otimes val$$

To see why it is sufficient, just take a look at the following procedure:

$$(a_i \otimes val) \oplus (a_{i+1} \otimes val) \oplus (a_{i+2} \otimes val) \oplus \ldots \oplus (a_j \otimes val) =$$

$$= ((a_i \oplus a_{i+1}) \otimes val) \oplus (a_{i+2} \otimes val) \oplus \ldots \oplus (a_j \otimes val) =$$

$$= ((a_i \oplus a_{i+1} \oplus a_{i+2}) \otimes val) \oplus \ldots \oplus (a_j \otimes val) =$$

$$\ldots$$

$$= (a_i \oplus a_{i+1} \oplus a_{i+2} \oplus \ldots \oplus a_j) \otimes val$$

Also, because the update happens *lazily*, we may end up updating a update, so we must have:

$$(u \otimes val_1) \otimes val_2 = u \otimes (val_1 \otimes val_2)$$

That is, $\otimes$ must be associative.
Summing up, these are all the conditions we must meet to use the Lazy Segment Tree:

- $\oplus$ must be associative (see Frozen Segment Tree section).
- $\otimes$ must be associative.
- $\otimes$ must be distributive over $\oplus$.

The operations we normally use are associative, so the third point is the hardest one to meet. Whenever using this data structure, pay special attention to it.

Readers who like math may have noticed T equipped with $\oplus$ and $\otimes$ has a structure similar to a ring (https://en.wikipedia.org/wiki/Ring_(mathematics)). But notice we don't need inverses, identities nor commutativity for $\oplus$ and $\otimes$, just the three conditions above.

This may seem a little too abstract, so here are some cases we can use Lazy Segment Tree:

- Update elements by increment and query max or min of range:

$$max(u + inc, v + inc) = max(u, v) + inc$$

- Update elements by setting new value and query max or min of range:

$$max(u \leftarrow val, v \leftarrow val) = max(u, v) \leftarrow val$$

- Update elements by multiplying them and query range sum.

$$(u * val) + (v * val) = (u + v) * val$$

3.1 Space Complexity

Here, we keep a lazy array as large as the array for vertices of the tree. At the same time, we drop the copy of the original array A, as we can't simply pick elements in $O(1)$ anymore because of the lazy update. So we take about 8 times the space of the original array. So $O(n \cdot \texttt{sizeof}(T))$.

3.2 Time Complexity

The time to build the Lazy Segment Tree is basically the same to build the Frozen Segment Tree (the only difference is the time to allocate and set the lazy flag). $O(n \cdot T_{op})$.

For the first type of queries $O(log(n) \cdot T_{op})$, just like the Frozen Segment Tree.

For the second type of queries, $O(log(n)(T_{op} + T_{up}))$.

4 Point Update Segment Tree

Remember that the update of a Lazy Segment Tree must be distributive over the query operation. This condition may be hard to meet sometimes. For example, let $\oplus$ be the usual $+$ and $\otimes$ be an increment operator. It is clear that $(u + val) + (v + val) \neq (u + v) + val$. So we can't have a Lazy Segment Tree that updates ranges through increment and queries the sum in ranges (for that we can use a clever combination of Fenwick Trees, as can be see in https://github.com/fbrunodr/CompetitiveProgramming/blob/main/algorithms/RURQ.cpp). But we can still use a Segment Tree if the update happens on points instead of ranges. That happens because we never have to update more than one point at once, so we never have to worry about doing $(u \oplus v) \otimes val$.

A Point Update Segment Tree is, in some sense, a more powerful version of a Fenwick Tree, because:

- $\oplus$ does not need to be commutative.
- $\oplus$ does not need an inverse $\ominus$.

Also, point updates simply transform a value a_i to another a'_i, so the `update` method I implemented simply takes the new value a'_i and update the data, instead of performing an operation $a_i \leftarrow a_i \otimes update$ as in the Lazy Segment Tree case. This makes the data structure easier to work with, as you don't have to worry about passing an updator functor (It becomes as easy to use as the Frozen Segment Tree).

That said, Fenwick Trees are still useful, as their multiplicative time complexities are small and they take less space.

4.1 Space Complexity

Same as Frozen Segment Tree, so $O(n \cdot \texttt{sizeof}(T))$.

4.2 Time Complexity

The time to build the Point Update Segment Tree is the same to build the Frozen Segment Tree, so $O(n \cdot T_{op})$.

For the first type of queries $O(log(n) \cdot T_{op})$, just like the Frozen Segment Tree.

For the second type of queries, $O(log(n)(T_{op} + T_{up}))$.

5 Extra

- We could call the Segment Trees above as "No Update Segment Tree", "Range Update Segment Tree" and "Point Update Segment Tree", but because the "Range Update Segment Tree" is normally known as Lazy Segment Tree, we chose the names above.
- In my implementation, Fenwick Trees and Point Update Segment Trees have an `update` method that takes a value val and sets an element to that value $a_i \leftarrow val$. The user of the Lazy Segment Tree may want another kind of update, such as range increment update. In that case we can't simply know the values of all the new elements beforehand. To avoid mistakes, I named the `update` method of this class as `rangeUpdate`.