Segment Tree Guide

by fbrunodr

April 16, 2023

1 Introduction

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

2 Frozen Segment Tree

The Frozen Segment Tree does not allow updating the internal data. It is 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. The Frozen Segment Tree answers 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

The space complexity is $O(n \cdot \texttt{sizeof}(T))$. To be more precise, we take about five times the size of A.

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 $O(n \cdot T_{up})$ to update it, 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 lazy update works, you will see it only works if:

$$(a_i \otimes u_1 \otimes u_2) \oplus (a_{i+1} \otimes u_1 \otimes u_2) = (a_i \oplus a_{i+1}) \otimes (u_1 \otimes u_2)$$

From that we draw the following conclusions:

- $\otimes$ must be associative.
- $\otimes$ must be distributive over $\oplus$.

Notice T equipped with $\oplus$ and $\otimes$ has a structure similar to a ring (https://en.wikipedia.org/wiki/Ring_(mathematics)), although we don't need inverses nor commutativity for $\oplus$ and $\otimes$.

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:

$$max(u + inc_1 + inc_2, v + inc_1 + inc_2) = max(u, v) + inc_1 + inc_2$$

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

$$max(u \leftarrow val_1 \leftarrow val_2, v \leftarrow val_1 \leftarrow val_2) = max(u, v) \leftarrow val_1 \leftarrow val_2$$

- Update elements by multiplying them and query range sum.

$$(u * val_1 * val_2) + (v * val_1 * val_2) = (u + v) * val_1 * val_2$$

3.1 Space Complexity

9/5 times the Frozen Segment Tree space (because of the lazy flag), 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, $O(log(n) \cdot T_{op} + T_{up})$.