Segment Trees 2

Veteran Track

Gabee De Vera

Last Week

Congratulations, You can Now Make a Segment Tree!

Review: The Basic Segtree Recipe

- What do you need to make a segtree work?

1. The **node state**, which contains the information necessary to answer a query.

2. A **combiner function** that updates the state of a node based on a combination of the states of its children.

3. An **update function** that traverses the segment tree, updates one of the leaf nodes, and propagates the changes up the tree.

4. A **query function** that computes the sum over a query range by combining the aggregate values over different ranges.

Segtrees are POWERFUL

Segtrees are POWERFUL

- Segtrees, in their basic form, can handle range sum and range maximum queries.

- In general, segtrees can handle any *associative binary operation*.

- Addition is one of them, since $(a + b) + c = a + (b + c)$

- However, we can also do multiplication, matrix multiplication, vector addition, and even other more "abstract" operations such as rotations and reflections on a square 🤯

- Let's see what other kinds of queries segtrees can handle! 👀

Counting The Number of Zeros in an Interval

- What if we want to count the number of times zero appears in an interval?

- As usual, we want to be able to update values in the segment tree and query ranges.

Counting The Number of Zeros in an Interval

- What if we want to count the number of times zero appears in an interval?

- As usual, we want to be able to update values in the segment tree and query ranges.

- To support these operations, you can **store 1 in a leaf node if the value at that index is 0, and store 0 otherwise**.

- Then, to count the number of zeros, we simply compute the sum over a range.

Counting the Number of Times the Maximum is Obtained

- What if we want to not only know what the maximum value over a range is, but also know how many times it occurs? Again, we want to also support point updates.

Counting the Number of Times the Maximum is Obtained

- What if we want to not only know what the maximum value over a range is, but also know how many times it occurs? Again, we want to also support point updates.

- This time, we store *both the maximum value and the number of times it occurs*. Let us represent this as $\langle m, c \rangle$, where m is the maximum value and c is the number of times m occurs.

- Of course, to make this work, we need a **combiner function**. In other words, given the maximum value and the number of times it occurs over two adjacent intervals, we must determine the maximum value and the number of times it occurs over the union of these two intervals.

Counting the Number of Times the Maximum is Obtained

- Let m_l and m_r be the maximum values over the left and right subintervals, respectively. Similarly, let c_l and c_r be the number of times these occur.

1. If $m_l > m_r$, then the maximum value over the entire interval is m_l, and it occurs c_l times.

2. If $m_l < m_r$, then the maximum value over the entire interval is m_r, and it occurs c_r times.

3. Finally, if $m_l = m_r$, then the maximum value over the entire interval is m_l (or m_r, it doesn't matter), and it occurs $c_l + c_r$ times.

Counting the Number of Times the Maximum is Obtained

- The implementation is quite long. See the GitHub for the implementation:

  https://github.com/RedBlazerFlame/reboot-materials/tree/main/compprog-materials/veteran/16-segment-trees-2/solutions/max-count.cpp

Finding the kth Zero

- Say we now want to find the kth zero in a segment.

- For instance, in the 0-indexed array $[1, 9, 0, 8, 0, 0, 3, 0, 5]$, the 3rd zero is at index 5.

Finding the kth Zero

- Say we now want to find the kth zero in a segment.

- For instance, in the 0-indexed array $[1, 9, 0, 8, 0, 0, 3, 0, 5]$, the 3rd zero is at index 5.

- It turns out that we can use a previous technique to solve this. If we *count* the number of zeroes in an interval, we can perform binary search on a prefix of the query interval.

- The kth zero can be found at the end of the smallest prefix containing exactly k zeroes.

- By doing binary search, we can find this prefix in $O(\log^2 n)$ per query, since we gain a log factor from both the segment tree and the binary search.

Finding the kth Zero

- It turns out that we can improve the $O(\log^2 n)$ time complexity to $O(\log n)$. The idea is to **perform the binary search directly on the tree**.

- This is known as **walking on the (segment) tree**.

- Implementation-wise, we simply define a function `kth_zero(ql, qr, k)`. Then, we attempt to find the kth zero. We return two values from the function: the number of zeroes found in total over the interval and the index of the kth zero.

- The purpose of also returning the number of zeroes found in the interval is to determine whether the kth zero is found in the interval or not. This is needed when we try to search over multiple intervals.

Finding the kth Zero

- The implementation is quite long. See the GitHub for the implementation:

  https://github.com/RedBlazerFlame/reboot-materials/tree/main/compprog-materials/veteran/16-segment-trees-2/solutions/kth-zero.cpp

Dynamic Maximum Subarray Sum

- Given a list of numbers a, find the maximum subarray sum over an interval $[l, r]$.

- Specifically, for $l \leq l' \leq r' \leq r$, find $\max \sum_{i=l'}^{r'} a[i]$.

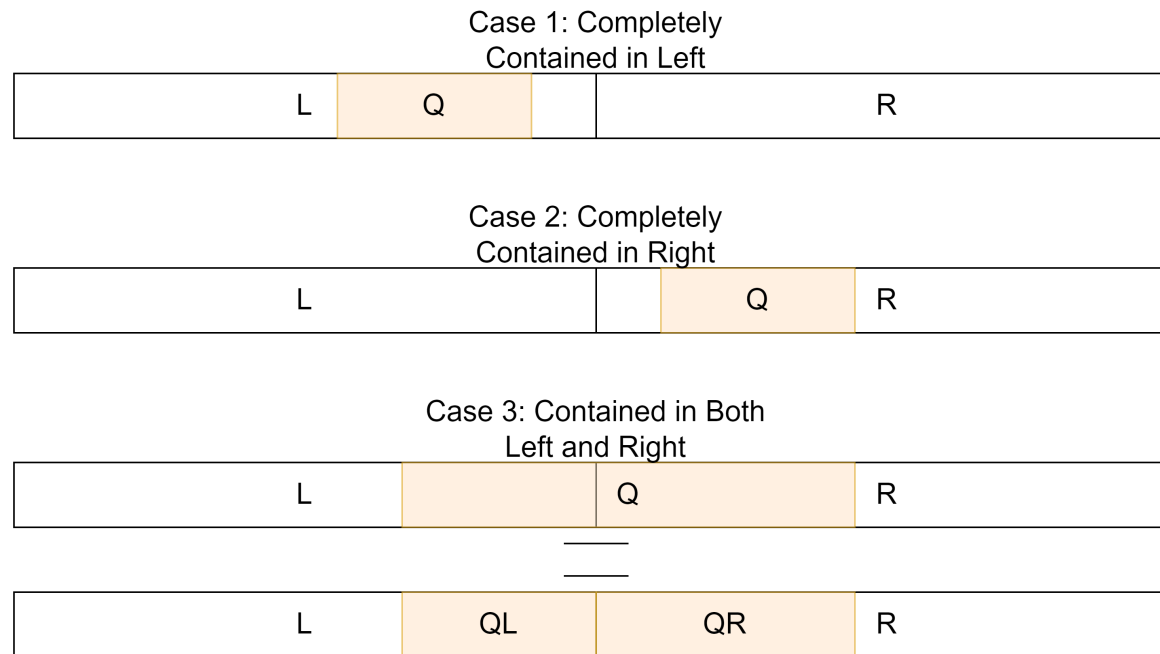- You also need to handle point updates.

Dynamic Maximum Subarray Sum

- Before we solve the dynamic version, note that there is an $O(n)$ greedy solution to the static version. Look up *Kadane's Algorithm* if you're curious.

- However, since we're going to implement a segment tree solution to this problem, we will instead investigate an $O(n \log n)$ divide-and-conquer solution to this problem.

Static Maximum Subarray Sum: Divide-and-Conquer

- What if we try to **divide** the interval into two?

- Then, the subarray with maximal sum is either completely contained in the left subinterval or the right subinterval.

- Alternatively, half of the subarray might be contained in the left subinterval while the other half is contained in the right subinterval.

Static Maximum Subarray Sum: Divide-and-Conquer

Case 1: Completely
Contained in Left

L	Q		R

Case 2: Completely
Contained in Right

L		Q	R

Case 3: Contained in Both
Left and Right

L		Q	R

___
___

L	QL	QR	R

- Notice that, in case 3, we can split the subarray into a suffix over the left subinterval and a prefix over the right subinterval.

Static Maximum Subarray Sum: Divide-and-Conquer

- To handle case 3, we must also know the *maximum prefix and suffix sum* over each interval. Fortunately, this is quite easy to compute, since the maximum prefix could either contain or not contain the left subinterval.

- This gives us an $O(n \log n)$ divide and conquer algorithm to find the maximum subarray sum over a static array.

Dynamic Maximum Subarray Sum

- To extend this to handle point updates, we simply simulate the divide-and-conquer algorithm on a segment tree.
- Specifically, for each interval, we will store:

    i. the maximum subarray sum,

    ii. the maximum prefix sum,

    iii. the maximum suffix sum, and

    iv. the sum of all values in that interval.

Dynamic Maximum Subarray Sum

- Then, the maximum prefix sum over an interval is either the maximum prefix sum over the left subinterval, or the maximum prefix sum over the right subinterval *plus* the sum of all the values in the left subinterval.

- The maximum suffix sum could be computed in a similar fashion as the maximum prefix sum.

- Finally, the maximum subarray sum is either the maximum subarray sum in the left subinterval or the right subinterval, or it is the sum of the maximum suffix sum over the left subinterval and the maximum prefix sum over the right subinterval.

- Using this, we can construct the segment tree.

Dynamic Maximum Subarray Sum

- To answer a range query, we do something similar to constructing the segment tree.

- At each step, we compute the maximum subarray sum, the maximum prefix sum, the maximum suffix sum, and the sum over the whole subarray.

Dynamic Maximum Subarray Sum: Implementation

- The implementation of the maximum subarray sum segment tree is quite long. Check the GitHub for the implementation: https://github.com/RedBlazerFlame/reboot-materials/tree/main/compprog-materials/veteran/16-segment-trees-2/solutions/max-subarray-sum.cpp

Homework

- Check the Reboot Website for the homework this week. Segment trees are advanced black magic reserved for the true masters of the CompProg art (/hj lol anyone can learn it), and it requires practice to learn to wield its power (okay this is actually true ^^). The techniques used this week are even more obscure and difficult to master. Thus, feel free to **collaborate and discuss with your fellow trainees**. You may also **ask for help from the trainers** and even **read the editorial (but only when you're really stuck)** 😄

Images Used

1. wheresgav. (2023, January 25). *Super Saiyan* [GIF]. Tenor.
   https://tenor.com/view/nft-shellz-shellz-orb-gif-27456375