Lecture 7: Lazy Segment Trees and Geometric Applications

Objectives of this lecture

- More examples of applying segtrees to design algorithms.
- Range updates and queries via lazy propagation.
- Coordinate compression techniques and geometric problems.

1 Philosophy

Segment trees open to the door to obtaining speedups for a huge number of algorithm design problems. Previous lecture you say point updates and range queries. In this class we give more examples of this, as well as describe how to build even more complicated data structures that can handle more advanced updates, such as range updates and range queries simultaneously, and how this additional power lets us solve certain geometric problems.

2 Example Problems

2.1 Decreasing sequences

This is just a warm-up example to get us in the headspace from the previous lecture.

Problem: Decreasing sequences

Let $a_1, \ldots, a_n$ be an array and let k be a given positive integer. Find the number of subsequences of length k that are decreasing, i.e., $1 \le i_1 < \cdots < i_k \le n$ such that $a_{i_1} > \cdots > a_{i_k}$.

Note that when $k = 2$ this is exactly counting inversions. We give an algorithm for this problem running in time $O(nk \log n)$ based on dynamic programming + range queries.

Step 1: Coordinate compression. Apply sorting to compress the a_i's to the range $\{1, 2, \ldots, n\}$ without changing the relative ordering. Clearly, this does not affect the answer to the problem. From now on, we assume that $1 \le a_i \le n$ for all i.

Definition of DP state. Let $DP[i][\ell]$ for $1 \le i \le n$ and $1 \le \ell \le k$ be the number of decreasing sequences of length ℓ which end at i. Then the DP transition is:

$$DP[i][\ell] = \sum_{j<i : a_j > a_i} DP[j][\ell-1].$$

Base cases are $DP[i][1] = 1$ for all i. The answer is $\sum_{i=1}^{n} DP[i][k]$.

Implementing the DP transition faster. Naïvely implementing the DP transition is $O(n)$ time per state. We will improve this to $O(\log n)$ by using a point update and range query data structure.

Recall from the previous lecture that there is a data structure that supports the following operations in $O(\log n)$ time per operation on an array $w[1], \ldots, w[n]$, initialized to all 0 say.

1. PointUpdate(i, x): sets $w[i] \leftarrow x$,
2. RangeSum(a, b): returns $\sum_{a \le i \le b} w[i]$.

Now we describe how to implement the DP transition using these operations. Initialize the initial array that the DS maintains to all 0.

1. Go from $i = 1, 2, \ldots, n$.
2. Set $DP[i][\ell] \leftarrow$ RangeSum$(a_i + 1, n)$.
3. Apply the update PointUpdate$(i, DP[i][\ell-1])$.

By the definition of the DP state, this solves the problem correctly. The runtime is $O(\log n)$ per state by the guarantees of the data structure. The total time is $O(nk \log n)$.

2.2 Range updates, queries, and lazy propagation

In this problem we build what is sometimes called a "lazy segment tree" or "segment tree with lazy propagation".

Problem: Range update and max query

Design a data structure that operates on an array $a[1], \ldots, a[n]$, initially all 0, supporting the following operations in time $O(\log n)$ per operation.

1. RangeAdd(ℓ, r, x): for all indices i satisfying $\ell \le i \le r$, set $a[i] \leftarrow a[i] + x$.
2. RangeMax(ℓ, r): return $\max_{i : \ell \le i \le r} a[i]$.

Approaches that don't work. You can't "pushdown" all the RangeAdd updates because they might be too expensive if the range is large. We need to be able to terminate once we have split the interval $[\ell, r]$ from the RangeAdd operations into intervals corresponding to segment tree nodes.

What information to maintain at each node? Recall that each node in the segment tree data structure corresponds to some interval. What information should we store at this node? Here is what we need to store (in addition the name of the node, and the corresponding interval):

1. The maximum array element in the interval (this helps to answer the RANGEMAX query. For notation, we will call this $\max[v]$ for a node labeled v.
2. "Unpropagated updates". This is an integer denoting RANGEADD updates to this node that have no been pushed down to its children. We need this because for runtime purposes we cannot afford to push down all updates. We call this $\text{lazy}[v]$ for a node labeled v.

Implementing updates and queries. Let's say that an update or query involves the interval $[\ell, r]$. The segment tree data structure processes this by walking down the tree. Here is the first critical point:

When moving to children of a node, push down all lazy updates and update its lazy value to 0.

In code, this amounts to doing:

1. $\text{lazy}[v.L] \leftarrow \text{lazy}[v.L] + \text{lazy}[v]$, and
2. $\text{lazy}[v.R] \leftarrow \text{lazy}[v.R] + \text{lazy}[v]$, and
3. $\text{lazy}[v] = 0$.

Now we describe updates and queries separately, starting with updates. When you end at a node (the intervals match), update $\text{lazy}[v] \leftarrow \text{lazy}[v] + x$. Also, update $\max[v] \leftarrow \max[v] + x$ (the max goes up by x if the whole interval goes up by x). Afterwards, go up the tree to all nodes you visited and update their $\max[v]$ values (simply by setting $\max[v] = \max(\max[v.L], \max[v.R])$).

Finally, queries. This is pretty simple, essentially the same as the case of point updates in the previous lecture. You still do the pushdowns, and just max over the $\max[\cdot]$ values of all nodes that you visited in the segment tree. Since each update and query visit $O(\log n)$ nodes, the cost is $O(\log n)$ per operation.

2.3 Coordinate compression

This is a warm-up problem that does not require any segment trees, but introduces a simple version of ideas we will need for the next problem.

Problem: Interval union

Intervals $[\ell_1, r_1], \ldots, [\ell_n, r_n]$ are given. Find the total length of the union of these intervals in time $O(n \log n)$.

Coordinate compression again. By sorting, etc., compress the values $\ell_1, \ldots, \ell_n$ and $r_1, \ldots, r_n$ to the range $\{1, 2, \ldots, 2n\}$ without changing the ordering. Also, remember which location each of $1, 2, \ldots, 2n$ correspond to – call these $d[1], \ldots, d[2n]$. Each $d[i]$ will be some original ℓ_i or r_i. In other words, you remember the mappings from ℓ_i's and r_i's to the range $\{1, 2, \ldots, 2n\}$.

Algorithm. Now for each interval, mark the corresponding range in $\{1, 2, \ldots, 2n\}$. In particular, mark its left endpoint (this is when the interval gets added in) and its right endpoint (this is when the interval gets removed). Now sweep from 1 to $2n$, and count the number of intervals which have been added but not removed (we will call such intervals "alive"). This can be done simply by maintaining a counter, and adding in intervals when their left endpoint arrives (increment counter), and removing them (decrement counter) when their right endpoint arrives. If a segment $(i, i+1)$ has a count of at least 1, some interval covers it. Now just add up the lengths of all covered segments.

2.4 Rectangle Area

Problem: Rectangle total area

n axis-aligned rectangles are given in the plane. Find the total area of the union of these rectangles in time $O(n \log n)$.

Formally, an axis aligned rectangle is the set of points (x, y) satisfying $a \le x \le b$ and $c \le y \le d$ for some real numbers $a \le b$ and $c \le d$.

Coordinate compression. Just like the third problem (Section 2.3), we first compress coordinates, so that all left/right/lower/upper boundaries of rectangles are compressed to be in the range $\{1, 2, \ldots, 2n\}$. We also remember the distance between adjacent vertical and horizontal coordinates, just like the above problem.

Left-to-right sweep. We sweep over the horizontal axis. For each rectangle, we remember its left and right boundaries. When we reach a left boundary, we mark that the vertical spread of the rectangle exists for now, i.e., its "alive". When we reach a right boundary, we remove the vertical spread of the rectangle. A vertical segment is alive if some rectangle containing that vertical segment is alive. Between two horizontal points, we compute the total area as the distance times the total length of the "alive" vertical spread.

Data structure problem. This requires designing a data structure for the following data structure problem. The input is an array $w[1], \ldots, w[n]$ of *weights* and an initial array $a[1], \ldots, a[n]$ of all 0's. These weights are distance between adjacent vertical markers. We want to support the following updates to a.

1. RANGEADD(ℓ, r, x): for all $\ell \le i \le r$, set $a[i] \leftarrow a[i] + x$. **It is guaranteed that $a[i] \ge 0$ at all times.**
2. POSITIVEWEIGHT(): Find the sum of $w[i]$ over all indices i where $a[i] > 0$.

Relation to the rectangle problem. The index $a[i]$ stores the number of alive rectangles covering the segment. We will set $x = -1$ or $+1$ in calls to RANGEADD depending on whether we are inserting a rectangle or deleting it. Calling POSITIVEWEIGHT tells us the total length of the "alive" vertical segments.

It turns out that it is possible to design a data structure to handle these operations in time $O(\log n)$ per update but is a bit more complicated than the above example of a lazy segment tree. The boldface is critical for this problem to be solvable in $O(\log n)$ time. Since this data structure needs to be called $O(n)$ times in the above algorithm, the total runtime is $O(n \log n)$.

The algorithm for this data structure is deferred to the oral homework.