DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Last week we discussed range query data structures. These data structures support in $O(\log n)$ time the following operations on an array:

1. Updating an element,
2. Query for the result of some computation on a range.

While we discussed only min/max and sum explicitly, these queries in general work for any kind of decomposable functions. That is, if we break $[l, r]$ into $[l, mid]$ and $[mid+1, r]$, there is a ‘merge’ function that we can piece together the information:

$$f([l \ldots r]) = \text{MERGE}(f([l \ldots mid]), f([mid+1 \ldots r]))$$

Then we define a tree whose nodes essentially correspond to

$$[2^k \cdot (i - 1), 2^k \cdot i - 1]$$

for some $0 \leq k \leq \log n$. The nesting structure of this node then gives the fast updates/queries.

This week we will extend this data structure further to allow updating entire ranges. Note that range sum can be couched as a range update, point query problem by storing the partial sums. That is, instead of storing the array $A[1 \ldots n]$, we can instead of maintain the partial sums array

$$S[i] = \sum_{1 \leq j \leq i} A[j]$$

Adding $x$ to some $A[i]$ adds $x$ to all $S[i], S[i + 1], \ldots S[n]$. So what we need to support is range add, and querying for the value of some $S[i]$.

We will show that this can be done in $O(\log n)$ time. Such schemes though, reply on propagating information downward: when we update a range that corresponds to a tree node, we simply mark that node with a flag indicating that it has been bulk updated. Pseudocode for such an access is:

```c
Update(int p, int l_p, int r_p, int l_u, int r_u) {
    if(l_p == l_u && r_p == r_u) {
        Update info on tree[p];
    }
```
else {
    int mid = (l_p + r_p) / 2;
    if(r_q <= mid) Update(p * 2, l_p, mid, l_u, r_u);
    else if(mid < l_q) Update(p * 2 + 1, mid + 1, r_p, l_u, r_u);
    else { \l_u <= mid < r_u
        Update(p * 2, l_p, mid, l_u, mid),
        Update(p * 2 + 1, mid + 1, r_p, mid + 1, r_u);
    }
}

That is, for tree[p], we track the flag of how much its corresponding range has been bulk edited. Then when querying for s[i], there are two options:

1. (more restrictive) because sum is additive, add together all tree[p] where p’s range contains i.

2. (more general) do a top-down (recursive) access on i, but push the tree[p] values downward. Specifically, we can guarantee that the path we access has no active flags via the operations

\[
\begin{align*}
\text{tree}[p \times 2] & += \text{tree}[p]; \\
\text{tree}[p \times 2 + 1] & += \text{tree}[p]; \\
\text{tree}[p] & = 0
\end{align*}
\]

That is, this is almost like a ‘reverse merge’ where we push all the range update information downward. It’s often referred to as ‘lazy propagation’ because the values are propagated downward as future updates encounter them.

The second approach is quite general. What’s more surprising is that it also composes well with range queries. Specifically, it’s possible to maintain things like ‘increment a segment’ and ‘query for the max of a segment’ in \(O(\log n)\) time per update, by equipping tree[p] with:

1. How much the range corresponding to p has been incremented,

2. The max of the range corresponding to p.

While this may seem intuitive, it leads to some fairly strange consequences.

**Problem 7.1.** Maintain two lists of numbers \(A[1 \ldots n], B[1 \ldots n]\) under operations:

1. Set an entire range of A to x.

2. Set an entire range of B to x.

Such data structures have some rather non-obvious uses. As a result, I recommend developing a fairly high level understanding of what kind of range operations can be supported, and try to manipulate problems toward them. Below is an example of a problem that can be solved using the two operations above.

**Problem 7.2.** Consider the message delivering problem from PS3, but messages can be delivered in ANY order, and you must come back to the origin. Specifically, you’re walking in a $n$-by-$n$ grid either horizontally or vertically, and want to deliver messages to a series of locations $(x_i, y_i)$ by either being in the same $x$, or $y$ coordinate. Compute minimum distance walked in $O(n \log n)$ time.

Note that this version of the problem essentially asks to find an $x$-interval $[x_{\text{min}}, x_{\text{max}}]$, and a $y$-interval $[y_{\text{min}}, y_{\text{max}}]$ (both containing 0) such that for all points, either

$$x_i \in [x_{\text{min}}, x_{\text{max}}],$$

or

$$y_i \in [y_{\text{min}}, y_{\text{max}}].$$

and the cost is just the absolute values of the coordinates.

For the algorithm, consider decreasing values of $x_{\text{max}}$, and storing the goal value of $x_{\text{min}}$ in an array. That is, for each $x$, we store

$$\min_{i : x_i < OR x_i > x_{\text{max}}} y_i$$

and try to dynamically maintain these as we decrease $x_{\text{max}}$. The set with $x_i < x$ is fixed for each $x$ value, so we need to consider what happens when we introduce a new $y$.

By symmetry, let’s only consider the max case first. This new $y$ value must be more than the max of all previous $x_i$ with $x_i > x_{\text{max}}$ for it to matter. But once it does matter, it will change the max value for all $i$ s.t.

$$y > \max_{i : x_i < x} y_i.$$

Note that these are increasing sets: so what we modify is going to be a suffix of all $x_i$ values, which forms a range.

Similarly, the min value can be maintained via range updates as well. Then by the data structure above, we can get the sum maintained too, and query for the min sum gives the answer.

This is an example of the ‘think with range updates’ method of problem solving. Below I will give three more examples of such approach to problem solving (hopefully to be discussed on Wednesday).
Problem 7.3. Given a static sequence of numbers \( x[1 \ldots n] \), define the unique sum of a range \( x[l \ldots r] \) to be the sum of the values that appear at least once in the range. That is, if \( x \) appears twice, or ten times, it still only gets summed once. Answer \( n \) queries of the following form in \( O(n \log n) \) time: given \( \hat{l} \) and \( \hat{r} \), find the range contained in it \((\hat{l} \leq l \leq r \leq \hat{r})\) with maximum unique sum.

This problem is available at [https://www.spoj.com/problems/GSS2](https://www.spoj.com/problems/GSS2).

Let’s start by considering an easier version of this problem: just computing the unique sum of the \( n \) ranges given, \([\hat{l}_i, \hat{r}_i]\). For this, consider processing the queries offline, in increasing order of their \( \hat{r}_i \) values.

Note that for a fixed right end point \( r \), the contribution of each distinct value \( x \) is to a prefix of \( l \) values: that is, any \( l \) to the left of the first point of appearance of \( x \) will have \( x \) added to it.

In other words, if we maintain \( s[i] \) to be the unique sum of the range \( i \ldots \hat{r} \) for the current value of \( r \), the effect of each \( x \) on such values is that it adds \( x \) to some range of it. Furthermore, note that as we move \( \hat{r} \) to the right by one, only one of these intervals may change. So we can update these sums using a single range update.

In this setting, finding the value for a single \( \hat{l} \) is querying the value of a single point. Finding the max unique sum over all \([\hat{l}, \hat{r}]\) is also doable with a single range minimum query. That is, querying for

\[
\min_{\hat{l} \leq i \leq \hat{r}} s[i].
\]

What about also allowing \( r \) to move? That’s actually querying for the historical maximum, that is, the max that any value in a range has been over the entire history of the data structure. This (plus range addition) is surprisingly also doable in \( O(\log n) \) time via lazy propagating: we store in each node:

1. The historical max sum since it was last propagated downward
2. The total sum added to it since the last propagated downward.
3. The max historical sum of something below it, assuming nothing in its ancestor has been added.

And the key reason this propagates well is that anything added to the ancestors of some node \( p \) happened after we last did some operation involving \( p \). So the operations don’t interleave in a time sense. Such operations are fairly magical though: personally I have to code this to believe that it works.

Problem 7.4. Break a sequence of \( n \) numbers, \( x[1 \ldots n] \), into groups of length at most \( k \) to minimize the sum of the max values in each group.

The prefix based dynamic program for this problem defines \( DP[i] \) as the best total cost of breaking up \( x[1 \ldots i] \). This leads to the transition:

\[
DP[i] = \min_{i-k \leq j < i} DP[j] + \max \{x[j+1 \ldots i]\}
\] (1)
This problem is available at https://dmoj.ca/problem/ccc02s4hard. The $O(n^2)$ version of it is also on problem set 3.

We claim that the values

$$DP[j] + \max \{x[j+1 \ldots i]\}$$

can be maintained using a total of $O(n)$ range additions as we increase the value of $i$. The reason is that if we want to introduce $x[i+1]$, we need to increment all values for $j < i$ such that

$$\max \{x[j+1 \ldots i]\} < x[i+1].$$

To understand how these change, we need to look at the structure of these values. At location $i$, there can be a whole bunch of distinct values, each corresponding to some $j$ such that

$$x[j+1] = \max \{x[j+1 \ldots i]\}.$$ 

That is, $x[j+1]$ is a local maxima when we start from $i$ and go in reverse order. Let these maxima be $j_1, j_2, \ldots, j_k$. Now consider what happens when we add in $x[i+1]$: if $x[i+1] > x[j_t+1]$ for some $t < k$, then all the other local maxima after $t$ will not be there after we’re done with item $i$. This is because for any $t > t$, we get

$$x[i+1] > x[j_t+1] > x[j_{\hat{t}}+1].$$

So we can go through the list $j$s one at a time, backwards (essentially off of a stack). Furthermore, note that adding $i$ only adds one local maxima to this list. So we can keep everything on a stack, and only perform a total of $O(n)$ range increment operations over the entire course of going through the $i$s.

The min among all $j$ s.t. $i - k \leq j < i$ can in turn be found using a range min query. This then gives a total running time of $O(n \log n)$.

**Problem 7.5.** In a sequence of $n$ (possibly negative) numbers, find $k$ non-overlapping segments whose total sum is maximized in $O(n \log n)$ time.

The idea here is to do greedy, except once we picked numbers, we negate them so that future updates can ‘reverse’ these choices. It leads to one of the most complex ‘naturally’ occurring range update data structure that I’m aware of:

1. Negate an interval of numbers.
2. Find the subinterval of maximum sum

This gets terribly complex because to just maintain max sum, one needs to maintain in addition the max prefix & suffix sums, along with the sum of segments. The negation operation means we also need to store the min prefix & suffix sums, leading to a whopping 7 intermediate values maintained per tree node.