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.

All the augmented search tree structures we’ve talked about so far are for intervals of 1-D, linear lists. This week we will discuss ways of composing them to build data structures for queries with multiple parameters.

An example of how such queries can happen is that each entry have two attributes, $x[i]$ and $y[i]$, and the queries want to ask about all points such that

$$x_l \leq x[i] \leq x_h$$
$$y_l \leq y[i] \leq y_h$$

This can be visualized as a rectangle on the 2-D plane. Hence, such data structures are often referred to as range queries, with the case above called a 2-D range query.

We can also consider the dense case for further simplicity:

**Problem 9.1.** Given a $\sqrt{n} \times \sqrt{n}$ square with coordiantes $[1 \ldots \sqrt{n}] \times [1 \ldots \sqrt{n}]$, support the following operations in $O(\log^2 n)$ time:

1. Add $v$ to the entry in location $(x, y)$.
2. Compute the sum of $[x_l, x_h] \times [y_l, y_h]$

To get some intuition for this problem, consider what happens if the $y$ interval is the full interval, that is $[y_l, y_h] = [1, \sqrt{n}]$. Then this becomes a single range query on the $x$ values, which can be handled with a 1-D range query data structures in $O(\log n)$ time. One way to think about this is that we can break $[x_l, x_r]$ into $O(\log n)$ disjoint ranges, one corresponnding to each node of the tree. Summing over them then gives the union of values we wanted for this range.

What if instead, on each of the nodes, we query for some general range $[y_l, y_h]$? The natural answer is to answer it via another 1-D range query data structure. That is, for each node of the 1-D search data structure, we maintain another range search data structure on the $y$ coordinates. This gives a query cost of $O(\log^2 n)$.

What about update cost? For each $x$ value, there are $O(\log n)$ intervals corresponding to tree nodes that contain it. So each change leads to updates on $O(\log n)$ 1-D trees, giving a total cost of $O(\log^2 n)$. 

1
For memory, because there are only $O(\sqrt{n})$ distinct $y$ coordinates, each of the 1-D trees take up size $O(\sqrt{n})$. This combined with the $O(\sqrt{n})$ trees for $x$ coordinates gives a total memory of $O(n)$.

What about the more general setting of sparse coordinates?

**Problem 9.2.** Given $n$ points on the 2D plane located at $(x[i], y[i])$, support the following operations in $O(\log^2 n)$ time and $O(n \log n)$ memory:

1. Add $v$ to the value of the $i$th point.
2. Compute the sum of $[x_l, x_h] \times [y_l, y_h]$

Here the differences are:

1. one needs to build the 1-D range search data structure based on sorted values of $x$ (and $y$),
2. for each node, the total size of the $y$ structure is no longer $\sqrt{n}$, but rather the size of the corresponding subtree of $x$.

The update/query times remain $O(\log^2 n)$. However, the memory increases to $O(n \log n)$: there are two ways of seeing this:

1. There are: 1 node of size $n$, 2 nodes of size $n/2$, 4 nodes of size $n/4$, ..., $n$ nodes of size 1. Each level has total size $n$, times the $O(\log n)$ layers gives the total.
2. Each node has $O(\log n)$ parents, so can contribute to this sum at most $O(\log n)$ times.

The somewhat strange extension of this is that the dynamic version, where new points are inserted/deleted, is still doable with same performance. However, the tree balancing mechanism matters there: treap is the simplest that gives the right properties.

This 2-D structure can also be used in other ways: often the second level structure can be replaced with something as simple as set, or queues.

**Problem 9.3.** An chord graph (not to be confused with chordal, or interval, graphs) is a graph where each vertex corresponds to a line between $l[i]$ and $r[i]$ ($l[i] < r[i]$), and an edge $i - j$ exists if their lines intersect. (note that line containments don’t mean intersections). Given such an interval representation, 2-color it in $O(n \log n)$ time.

Note that there are two ways for intervals to intersect:


So if we want to perform depth first search on this graph, what we need to do (up to symmetry) is a data structure that supports:
1. Query for any point such that \( l < x[i] < r \) and \( y[i] > z \).

2. Delete such a point.

Then note that in any range, if there is some \( y[i] > z \), the maximum must also exceed \( z \). So all we need to maintain, at every node, is a sorted list of the \( y \)'s in decreasing order. Any time we perform a query, we delete some tail of this list until the max is below the threshold we want. This takes \( O(n \log n) \) time total after construction, because each element is deleted once.

To get preprocessing to also work in \( O(n \log n) \) time, observe that we can construct the sorted list at \( p \) by merging the sorted lists for \( p.left \) \( (p * 2) \) and \( p.right \) \( (p * 2 + 1) \).

We can also use this type of structure to solve range distinct element problem. Such problem can be ‘soupied up’ to things like computing the lowest common multiple (LCM) of ranges of numbers.

**Problem 9.4.** Range LCM: maintain a list of \( n \) positive integers \( A[1 \ldots n] \), each up to \( n \), and support:

1. Update a number.

2. Query for the lowest common multiple of an interval (modulo some word-sized integer \( M \)).

Both in \( O(\log^3 n) \) time.

This problem, in its static version, recently showed up at [https://codeforces.com/contest/1422/problem/F](https://codeforces.com/contest/1422/problem/F). Note that’t it’s really the \( \pmod{M} \) that prevents us from treating it standardly (as one would treat GCD) via 1-D range search: given two LCMs mod \( M \), we cannot find their LCM mod \( M \).

So what we instead need to do is for each prime factor, find the maximum number of times it appears in the query range \( [l, r] \). Note that any positive integer up to \( n \) has at most \( O(\log n) \) prime factors. We can obtain these factors by pre-processing everything in \( [1, n] \) in \( O(n \log n) \) time (using sieve based factorization).

Furthermore, we can treat each different power of \( p \) as a separate number that contributes \( p \) to the overall product. That is, if \( p^j \) appears in the range \( [l, r] \), it will multiply the overall LCM by a factor of \( p \). This works because if \( p^3 \) is present, we also record \( p^2 \) and \( p \) as present as well. So up to a factor of \( O(\log n) \), the problem becomes the following involving key/value pairs:

**Problem 9.5.** Support insertion/deletion of keys, each associated with some value, on an array, and the querying for the product \( \pmod{M} \) of all values corresponding to keys that appear at least once in the range.

We solve this by storing for each occurrence of a key, its previous occurrence location. Then we only count the first appearance of a key in the range. That is, if this location is at \( i \), we must have

\[
\text{PREV}(i) < l \leq i \leq r. \tag{1}
\]
This is exactly the same as the condition above about chords intersecting: it can be handled by building a structure on location \(i\) first, and then a prefix sum/product query structure on the PREV values, with values associated with keys. Querying/modifyng this data structure takes \(O(\log^2 n)\) time, with the caveat that the second level structure needs to be a complete binary search tree as the PREV locations are dynamic. Also, note that as we modify a key, we also change the PREV value of the successor, so two updates are needed.

Range update / queries in 2-D becomes much more nuanced. I’m aware of a total of two such combinations that are doable in \(O(\log^2 n)\) time:

**Problem 9.6.** Support 2-D range add, and 2-D range sum, in \(O(\log^2 n)\) time each.

One simplification that we can make to this is to assume all ranges include (1,1). Aka, they are prefixes in both dimensions. The trick is to directly store the ‘2-D prefix sums’. That is, we store

\[
S[i][j] = \sum_{1 \leq k \leq i} \sum_{1 \leq l \leq j} A[i][j].
\]  

Adding \(x\) to everything in the rectangle \([1 \ldots k] \times [1 \ldots l]\) changes \(S[i][j]\) in four ways:

1. If \(k \leq i, l \leq j\), it changes by \(x \cdot k \cdot l\): this is a range tree add! That is, we simply need to increase all \(S[i][j]\) with \(i \geq k, j \geq l\) by \(x \cdot k \cdot l\).

2. If \(k \geq i, l \geq j\), it changes by \(x \cdot i \cdot j\). This is a range add on coefficients. That is, we store for each \(ij\), the coefficient of \(i \cdot j\) in \(S[i][j]\), and increase all \(i\) and \(j\) with \(i \leq k, j \leq l\) by \(x\).

3. If \(k \leq i, l \geq j\), it changes by \(x \cdot k \cdot j\). Note that the only dependence is on \(j\). So we need another tree to track the coefficient of \(j\) added to each entry. In this tree, we increase all with \(i \geq k, j \leq l\) by \(x \cdot k\).

4. If \(k \geq i, l \leq j\), it changes by \(x \cdot i \cdot l\). This is the mirror case above, except we track coefficients on \(i\).

In other words, we break \(S[i][j]\) into 4 terms:

\[
S[i][j] = S0[i][j] + i \cdot S1[i][j] + j \cdot S2[i][j] + i \cdot j \cdot S3[i][j].
\]  

In each of these trees, we need to support range increment, point query. That is far more manageable: once we break a rectangle into the \(O(\log^2 n)\) rectangles involved, we simply add \(x\) to each of them. Then we can query the value of a point in \(O(\log^2 n)\) time by summing over all tree nodes whose corresponding rectangles contain it.

**Problem 9.7.** Support 2-D range set, and 2-D range max, in \(O(\log^2 n)\) time each.