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.

In this lecture we revisit dynamic programming with the highly powerful ‘for loops’ / range operations that we developed over the previous month.

First, consider the following problem that’s oddly similar to the ‘walk in a grid with non-overlapping obstacles’ problem:

**Problem 10.1.** One is traveling on the 2D plane: going upward by 1 unit per second, but can move left/right by at most 1 unit per second as well. Given n prizes, collect as much of them as possible.

This problem is a simplified version of [https://icpcarchive.ecs.baylor.edu/external/73/7374.pdf](https://icpcarchive.ecs.baylor.edu/external/73/7374.pdf).

Note that this problem is a prefix based dynamic program: we can let $DP[i]$ represent the maximum number we can collect while passing through $i$, and we can go from point $i$ to point $j$ (with coordinates $(x[i], y[i])$ and $(x[j], y[j])$ respectively) if and only if:

$$y[j] \geq y[i]$$

$$|x[i] - x[j]| \leq y[j] - y[i]$$

We can get rid of the absolute value on the second term using

$$|x[i] - x[j]| = \max \{x[i] - x[j], x[j] - x[i]\}$$

which gives the conditions

$$x[i] - x[j] \leq y[j] - y[i]$$

$$x[j] - x[i] \leq y[j] - y[i].$$

Isolating terms involving $i$ and $j$ separately, we get

$$x[i] + y[i] \leq x[j] + y[j]$$

$$y[i] - x[i] \leq y[j] - x[j].$$

Furthermore, note that adding these conditions gives $2y[i] \leq 2y[j]$, which implies the first condition. So we can discard the first condition altogether. That is, for each $j$, we consider all $i$ such that $a[i] \leq a[j]$, $b[i] \leq b[j]$, where

$$a[k] = x[k] + y[k]$$

$$b[k] = y[k] - x[k].$$
This can be done by first sorting all points in order of $a[k]$, and then doing a prefix max query for all points with 

$$b[i] \leq b[j],$$

which with range searching is $O(\log n)$ time. The update operation is once we find the value of $DP[j]$, we update it onto the location $b[j]$.

There is an even more direct correspondence between this problem and finding the longest increasing subsequence: the $a$ and $b$ values are essentially obtained by rotating the plane by 45 degrees.

The most common type of problem where such range operations are applicable are ones with clear prefix-like structures. In fact, one can even create problems by reverse engineering them from the range update/search data structures.

**Problem 10.2.** Given a sequence of $n$ positive pairs $A[i]$ and $B[i]$, partition it into segments so that:

1. In each part, the sum of the $B[i]$s is at most $L$.
2. In every segment, each of the $B[i]$s are greater than all $A[j]$s in previous segments.

and minimize the sum of the maximums of the $A[i]$s among the pieces.

This problem, with an extra layer of binary search, is at https://www.spoj.com/problems/SEQPAR2/.

Clearly, the problem is asking us to implement a dynamic program where $DP[i]$ is the min sum of the maxes of segments after we partition up $[1 \ldots i]$.

There are two things we need to do: make sure that the transition from $j$ to $i$ meets the two conditions, and maintaining the values of

$$DP[j] + \max \{A[j + 1], A[j + 2], \ldots A[i]\},$$

as we increase the $i$s.

For the first portion, observe that for each $j$ representing a split of $1 \ldots j$, there is a clear point of expiry: its the earlier of:

1. The first index after $j$ where the sum of $B[i]$s exceeds $L$: this is a threshold because all the $A$s are positive.
2. The first index after $j$ where $B[i]$ is less than $\min_{1 \leq k \leq j} A[k]$.

The first can be calculated using binary search on prefix sums, while the second can be calculated using range searching, and the prefix-min on the $A$s.

This means that as we increase the indices $j$, we can zero-out ‘expired’ entries of $i$. Which leaves the issue of updating the max. Note that for each new value of $A[j]$, we update all the max entries until the first one before it that’s greater. We can find this location again via a binary search / prefix min, after which the operations becomes the two-sequence min with set that we discussed before. That is, at each index $i$, we store:
1. $DP[i]$ for all active $i$ is in the first array.

2. $\max_{i \leq k \leq j} A[k]$ in the second array.

Then the update on max values is in a suffix, and is a lazy range set, while the first operation of removing the expired $j$s is a single entry set.

Finally, at each $j$, we just query for the minimum sum of these two entries among the entire tree, giving $O(\log n)$ time each $j$, and $O(n \log n)$ total.

Recall from before there was also a way to do this with a stack, where we gradually ‘erase’ the other minimas. However, the bulk range set operation is much simpler.

**Problem 10.3.** There are $n$ events happening, each on day $T[i]$ at location $L[i]$, and will yield a profit of $M[i]$ if attended. Traveling is insantenous, but each unit in the $+1$ direction costs $D$, while each unit in the $-1$ direction costs $U$. Events on the same day can be attended in any order, but events on different days can only be attended in chronological order. Find the max profit that can be made when starting at time 0 in location $S$.

This problem is at [https://dmoj.ca/problem/ioi09p8](https://dmoj.ca/problem/ioi09p8).

First, consider the case where all events happen on different days. Then the time provides a clear ordering, and $DP[i]$ represents the max profit that can be made after attending event $i$ (plus an optimal subste of the ones occurring before it).

The transitions are then

$$DP[i] = M[i] + \max \left\{ \max_{j \leq i, L[j] \leq L[i]} DP[j] + D (L[i] - L[j]), \max_{j \leq i, L[j] \geq L[i]} DP[j] + U (L[j] - L[i]) \right\}.$$

In both cases, we can isolate out the terms involving $L[i]$, giving:

$$\max \left\{ D \cdot L[i] + \max_{j \leq i, L[j] \leq L[i]} DP[j] - D \cdot L[j], -U \cdot L[i] + \max_{j \leq i, L[j] \geq L[i]} DP[j] + U \cdot L[j] \right\}.$$

In both cases, the term inside the range maximum is a range max involving $DP[i] - D \cdot L[i]$ and $DP[i] + U \cdot L[i]$ respectively. So both can be answered (and maintained) in $O(\log n)$ time.

It remains to handle the case where events happen on the same day. Note that for the first event attended that day, we can still use the range queries above, as long as the trees only store events from previous days. Then we need to make two additional observations:

1. The events visited that day form an interval.

2. It’s ok to visit one end of that interval first, and end at the other end: the rest can be charged toward ‘additional’ travel between the days.
In other words, at some point, we went from one end of the visited interval to the other, and we can use that ‘pass’ as the time in which we visited the objects.

So once we sorted everything by location, we can propagate the best values using

\[ DP[i] = \max \{ DP[i], DP[\text{Prev}(i)] + D \cdot (L[i] - L[\text{Prev}(i)]) \} \]

for going downstream, and a mirror array for the upstream sweep (but make sure to use a ‘stale’ copy containing only the transitions from previous days).

**Problem 10.4.** There are \( n \) people in a line: each person has a hat of size \( w[i] \), and want a hat of size \( v[i] \). Starting from person 1, until person \( n - 1 \), each person can either:

1. Do nothing.
2. Swap hat with person \( i + 1 \).

after which their hat choice is considered finalized. Let \( h[i] \) be the final hat sizes, find the minimum value of

\[ \sum_i |h[i] - v[i]|. \]

This problem is at [https://dmoj.ca/problem/dmopc18c5p5](https://dmoj.ca/problem/dmopc18c5p5).

Let \( DP[i] \) be the optimum value among \( 1 \ldots i \) when person \( i \) does nothing (aka. keeps whatever hat was given to them). Note that person \( n \) must do this, so \( DP[n] \) is well defined.

To calculate \( DP[i] \), let \( j \) be the last previous person who did nothing. Then the hats that

\( j + 1, j + 2, \ldots i \)

have are respectively

\( j + 2, j + 3, \ldots i, j + 1 \)

So the resulting total cost is

\[ |w[j + 1] - v[i]| - \sum_{k=j+1}^{i-1} |w[k + 1] - v[k]| \]

(some effort is needed to verify that this formula is still ok if \( j = i - 1 \)).

Note that \( |w[k + 1] - v[k]| \) are fixed values: we can pull these values of out of prefix sums. That is, if we define

\[ S[j] = \sum_{k \leq j} |w[k + 1] - v[k]|, \]

the DP transition becomes

\[ Dp[i] = \min_{j < i} \{ DP[j] + |w[j + 1] - v[i]| + S[i - 1] - S[j] \}. \]
As with the moving upstream/downstream case, the majority of effort will be spent untangling this absolute value function: there are two cases, \( w[j+1] \leq v[i] \), or \( w[j+1] \geq v[i] \). Separating them gives:

\[
\min \begin{cases} 
\min_{j<i, w[j+1] \leq v[i]} DP[j] + v[i] - w[j+1] + S[i-1] - S[j], \\
\min_{j<i, w[j+1] \geq v[i]} DP[j] + w[j+1] - v[i] + S[i-1] - S[j]. 
\end{cases}
\]

Manipulating these to move out terms dependent on \( i \) then leads to:

\[
\min \begin{cases} 
v[i] + S[i-1] + \min_{j<i, w[j+1] \leq v[i]} DP[j] - w[j+1] - S[j], \\
-v[i] + S[i-1] \min_{j<i, w[j+1] \geq v[i]} DP[j] + w[j+1] - S[j].
\end{cases}
\]

So they become range min queries (with ranged given by \( w[j+1] \)) over values \( DP[j] - w[j+1] - S[j] \) and \( DP[j] + w[j+1] - S[j] \) respectively. Both take \( O(\log n) \) time (assuming the \( ws \) and \( vs \) are pre-sorted), so the total runtime is \( O(n \log n) \).