**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.

I will spend this class showing something strange that connects everything we did so far together.

Consider the following problem:

**Problem 14.1.** Given a sequence of \( n \) numbers, \( x_1 \ldots x_n \), find a non-decreasing sequence of numbers \( y_1 \ldots y_n \) that minimizes

\[
\sum_i |x_i - y_i|
\]

We first get an \( O(n^2) \) time algorithm via the following observation.

**Lemma 14.2.** There exists an optimum solution where all \( y_i \)'s equal to some \( x_j \) (for some \( j \) not necessarily equal to \( x_i \)).

**Proof.** First, consider what happens if all \( y_i \)'s are distinct: for some particular \( y_i \), if it’s not the same as \( x_i \), we can move it toward \( x_i \) until it hits either the predecessor, or the successor, or \( x_i \) itself.

Then consider what happens if we move a ‘block’ of the \( y_i \)'s (with same values) around. We either get to the medians of the \( x \)'s in the same block, or merge with previous blocks. In the latter case, we decrease the number of blocks, so this will eventually terminate, and we get a solution whose value is no more than what we started, and has all \( x_i \)'s equalling to the medians of their corresponding blocks. \( \square \)

This gives a dynamic program: we use

\[
DP[i][j]
\]

to denote the minimum possible value of choosing \( y[1] \leq \ldots \leq y[i] \) such that \( y[i] = x[j] \).

Then the transition is

\[
DP[i][j] = |x[j] - y[i]| + \min_{k:x[k] \leq x[j]} DP[i - 1][k].
\]

Implemented directly, this takes \( O(n^3) \) time. However, we can do faster: instead of letting \( j \) denote the value \( x[j] \), we sort \( x \) to become \( z \), so things are increasing, and use the state
to represent the case where $y[i] = z[j]$. The minimum are then over prefixes, and can be calculated in $O(n)$ time. So the total running time is $O(n^2)$.

We will try to do better: the only method we talked about so far is to reduce number of states. We will instead do something different: use the fact that the function has convexity in $j$ to represent it implicitly using data structures.

Formally, our definition of concavity is:

Definition 14.3. An array $A$ is convex if the partial differences

$$A[i] - A[i - 1]$$

is decreasing, and concave if this is increasing.

Lemma 14.4. Consider a modification of the DP table where $DP[i][z]$ is the min value if $x[i] = z$. For any $i$, the table $DP[i][z]$ is concave in $z$.

Now consider what happens if we have such a table $a[\ldots]$, and we want to create the table

$$b[i] = |i - x| + \min_{j \leq i} b[j].$$

There are two steps: first to take prefix minimum, and then add $|i - x|$ to each $b[i]$. We consider the second operation first: it adds $x - i$ to everything before $x$, and $i - x$ to everything after. There is another way to think about this, in terms of the second-order partial difference/derivative. We’re adding a value of $x$ at location 0 (assuming $x > 0$), and everything after gets a derivative of $-1$. Then at $x$, this derivative changes from $-1$ to 1. So if we only store the second order derivative, as well as $b[0]$, there are only three locations we need to change:

1. $b[0]+ = x$,
2. $b''[0]− = 1$,
3. $b''[x]+ = 2$.

So all we have to do to store this function is to store the value of $b[0]$, the assumption of $b'[-1] = 0$, and a treemap for all values of $b''[x]$. In particular, discrete integral gives

$$b'[i] = \sum_{j \leq i} b''[i],$$

and in turn

$$b[i] = b[0] + \sum_{j < i} b'[j] = b[0] + \sum_{j < i} (i - j) b''[j].$$

So with this representation, it becomes super easy to add $|i - x|$ to some value $x$.

The only other thing we need to do is to set everything before the minimum before we get to the minimum. However, by what we know from derivatives, the minimum happens
when $b'[i] = 0$. So because $b'[z_{max}]$ is just the sum of all the $b''$s we store, we have this final value, and we can work backward (subtract off $b''$s) until we get to the point of 0 derivative. Everything after that point can be removed, so this amortizes to a small total cost.

This algorithm takes $O(n \log n)$ time: inserting a value into $b''$ takes $O(\log n)$ time, and everything gets removed at most once when we compute the minimum.

We can also use this approach to solve even more sophisticated problems.

**Problem 14.5.** Given $n$ blocks, each with a weight $w[i] \geq 1$, and load $s[i]$, find the maximum number of a subset of these blocks that can be formed into a tower: that is, each block’s load is at least the total weight of the blocks above it.

This problem can be found at [https://ncna17.kattis.com/problems/atlantis](https://ncna17.kattis.com/problems/atlantis). The $O(n^2)$ version of this, but with additional values of blocks, was on Problem Set 10 ([https://dmoj.ca/problem/dpx](https://dmoj.ca/problem/dpx)).

There are two steps to this problem: we first show that in the optimum configuration, the $w[i] + s[i]$ values are non-decreasing. This is by a swapping argument: suppose blocks $i$ and $i + 1$ have

$$w[i] + s[i] > w[i + 1] + s[i + 1].$$

Then we can swap these two blocks because:

1. block $i + 1$ can support blocks $1 \ldots i - 1$ because that’s a subset of blocks $1 \ldots i$, which it currently supports.

2. block $i$ can support blocks $1 \ldots i - 1$ plus $i + 1$ because

$$s[i] > w[i + 1] + s[i + 1] - w[i] \geq w[i + 1] + \sum_{j=1}^{i} w[j] - w[i] = w[i + 1] + \sum_{j=1}^{i-1} w[j].$$

So after this sort, we maintain $DP[i][j]$: the minimum weight of a tower of $j$ blocks using blocks $1 \ldots i$. The key claim is that for each $i$, $DP[i][j]$ is convex in $j$. Then this function can be updated using either a map/treeset similar to above, or a full balanced binary search tree.