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.

We continue with the theme of optimizing dynamic programming, but with more specialized data structures that more closely fit the transition function.

Once again, we will start with prefix based transition functions:

\[
DP[i] = \min_j DP[j] + f(i, j)
\]

The main approaches that we will go into in more detail are:

1. Prove that the optimum \( j \) is always increasing.

2. Show that one can quickly find the argmin of \( DP[j] + f(i, J) \), either in worst case time per \( i \) (as we did before with data structures), or amortized time over all \( i \) via some kind of pointer based sweep.

Problem 12.1. Divide a sequence of \( n \) positive numbers into segments, minimize the sum of squares of the segments, plus \( C \) times the number of segments.

This problem is available at [https://open.kattis.com/problems/coveredwalkway](https://open.kattis.com/problems/coveredwalkway).

Let \( S \) denote partial sums. The DP formula then becomes:

\[
DP[i] = \min_j DP[j] + (S[i] - S[j])^2
\]

Expanding the formula gives

\[
DP[j] + (S[i] - S[j])^2 = S[i]^2 + DP[j] + S[j]^2 - 2S[i] \cdot S[j]
\]

So if we rename the variables:

\[
\begin{align*}
x[j] &= 2x[j] \\
\end{align*}
\]

we’re actually looking for the minimizer of

\[
y[j] - x[j] \cdot S[i]
\]

This means our interpretation of \( x \) and \( y \) is not an accident: this value is the \( y \)-intercept of a line through the point \((x[i], y[i])\), with slope \( S[i] \). In other words, we’re querying for the lowest point when we move up a line with slope \( S[i] \) upwards!
The most direct way to use this fact is that the $S[i]$s are monotonically increasing. So the standard convex hull algorithm of putting a point on a stack, and popping any previous point that’s too low compared to the previous one, works. This means that we can maintain the hull of all the $(x[i], y[i])$ points in $O(n)$ time.

There are two ways of using this hull:

1. Binary search for the line with closest slope to $S[i]$ in $O(\log n)$ time, and return value of that point.

2. Use the fact that the queries are also monotonically increasing in slope, and just walk a pointer upward on the hull, as long as things are better. This takes $O(n)$ time total.

The second condition is significant on its own: it proves that the ‘decision point’ for each $i$, that is the optimum $j$ for each $i$, is monotonically increasing. This fact alone is sufficient for speeding up dynamic programming.

This is actually easier to use in the 2-D case, and was first introduced there.

**Problem 12.2.** Solve the optimum binary search tree problem, this time with internal nodes as well (minimize $\sum_i \text{depth}[i] \cdot w[i]$) in $O(n^2)$ time.

Here the DP transition is

$$DP[l][r] = \text{SUM}(w[l \ldots r]) + \min_{l \leq m \leq r} (DP[l][m-1] + DP[m+1][r]).$$

The key fact is that $m[l][r]$, the optimum mid point for $[l \ldots r]$ is monotonic: as we move $r$ to the right, it should also move to the right (or remain still). Similarly, as we move $l$ to the left, it does not move to the right.

Formally, this gives

$$m[l][r-1] \leq m[l][r] \leq m[l+1][r],$$

that is, the opt decision point of each length $k$ segment is ‘sandwiched’ between the opt decision points of the two immediate $k - 1$ segments. The decision points of these length $k - 1$ segments are also monotonically increasing, and the total lengths of the gaps is at most $O(n)$.

So we can solve the decision points for all length $k$ segments in $O(n)$ time. As $k \leq n$, the total runtime is $O(n^2)$.

**Problem 12.3.** Given $n$ points on a line, build $k$ post office to minimize the total distance everyone need to travel to their closest post office.

This problem, with small bounds, is available at [https://dmoj.ca/problem/ioi00p5](https://dmoj.ca/problem/ioi00p5).
First, sort the locations. Then note that if we assign locations \( j + 1 \ldots i \) to the same office, the optimum office is placed in the median location. That is, the opt post office location is \( \text{loc}[k] \) for \( \text{mid} = \lfloor \frac{j + 1 + i}{2} \rfloor \), and the optimum cost is

\[
\sum_{t=j+1}^{\text{mid}} (\text{loc}[\text{mid}] - \text{loc}[t]) + \sum_{t=\text{mid}+1}^{i} (\text{loc}[t] - \text{loc}[\text{mid}]).
\]

With prefix sums, this simplifies to

\[
\text{Cost}(j, i) = (\text{mid} - j) \cdot \text{loc}[\text{mid}] - S[\text{mid}] + S[j] + S[i] - S[\text{mid}] - (i - \text{mid}) \cdot \text{loc}[\text{mid}]
\]

Even though this looks complicated, it can be done in \( O(1) \) time.

So naively, this dynamic program takes \( O(n^2k) \) time to compute: there are \( n \) states for each of the \( k \) numbers of offices build, for a total of \( O(nk) \).

To speed this up, the key is that the optimum decision point, \( j[i][\hat{k}] \), is monotonic in each \( i \). Formally, as \( i \) grows, the last interval should cover more. To take advantage of this in 1-D, we will need to do a divide-and-conquer on the median \( i \): once we know \( j[n/2] \), we know that \( 1 \ldots n/2 - 1 \) have decision points before \( j[n/2] \), and \( n/2 + 1 \ldots n \) have decision points after. Note that this exactly splits up the possible choices \( 1 \ldots n \): so we can pass the two halves of potential \( j \) to the two halves, and repeat. This gives a recursion of the form of

\[
T(n, l) = O(l) \max_k [T(n/2, k) + T(n/2, l - k)]
\]

which solves to \( T(n, n) \leq O(n \log n) \). So the total cost is \( O(nk \log n) \).