• **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.

• Main topics:
  - Review for Test 3.
  - *(optional)* speedups using monotonic decision point

• Last time:
  - Recursive view of dynamic programming made more sense than iterative.
  - How to define dynamic programming states.
  - More knapsack.

The plan is to discuss a quick trick for speeding up the dynamic program for optimal binary search trees (Exercise 6.20 in textbook).

The problem asks to arrange a set of keys on a binary tree in order to minimize the access cost. Here each key has a frequency that it will be accessed, $f(i)$, and the goal is to minimize

$$\sum_i \text{depth}(i) \times f(i).$$

We start by obtaining an $O(n^3)$ time dynamic programming solution for this problem. The key observation is that subtrees of nodes correspond to intervals of nodes, and the total cost can be rewritten as:

$$\sum_i \text{depth}(i) \times f(i) = \sum_{\text{Tree node } p} \sum_{i \in \text{SubTree}(p)} f(i).$$

When we’re given some interval $[l, r]$, picking a root at $k$ divides it into subintervals $[l, k - 1]$ and $[k + 1, r]$. This naturally leads to our dynamic program:

1. State: $OPT[l, r]$: optimum cost of a binary tree built from entries $l \ldots r$.

2. Base case: if $r < l$, $OPT[l, r] = 0$. 
3. Transition:

\[ OPT[l][r] = \sum_{k=l}^{r} f(k) + \min \{OPT[l][k-1], OPT[k+1][r]\}. \]

4. Ordering of states: in increasing order of length, \( j - i \).

This runs in \( O(n^3) \) since there are \( O(n^2) \) states and each makes \( O(n) \) transitions.

This is a case where better algorithms can be found by observing the monotonicity of the choices made by the dynamic program. Consider a variable \( k^*[l][r] \) that is the best choice of \( k \) for a value of \( l \) and \( r \).

With some effort, one can prove that as the interval gets wider, the midpoint only moves rightwards.

**Lemma 0.1.** \( k^*[l][r] \) is monotonic in both \( l \) and \( r \).

In particular, this implies

\[ k^*[l][r-1] \leq k^*[l][r] \leq k^*[l+1][r], \]

which means when we look for the optimum \( k \) for \( OPT[l][r] \), it suffices to loop through only the values between \([k^*[l][r-1], k^*[l+1][r]]\).

How much are we saving from doing this? Notice that if we compute the interval of the same length, but shifted one to the right, \( OPT[l+1][r+1] \), we will examine \([k^*[l+1][r+1], k^*[l+2][r+1]]\).

This means that the optimum \( k \) values of the intervals that are 1 shorter partitions the space \( 1 \ldots n \), and we will visit each interval at most once when solving for the current interval length.

This lets us solve all intervals of a fixed length in \( O(n) \) time, bringing the total down to \( O(n^2) \).