• 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:
  – Longest increasing subsequence.
  – (optional) speedups using binary search.

• Textbook:
  – Chapter 6.2.

The longest increasing subsequence problem is given a list of length \( n \), \( a_1 \ldots a_n \), find
the longest subsequence, \( i_1 < i_2 < \ldots < i_k \) such that the corresponding values in \( a \) are
increasing, aka.
\[
a_{i_j} < a_{i_{j+1}}
\]
for all \( 1 \leq j < k \).

This can be solved via dynamic programming in two ways: either view it as a longest
path problem in a DAG, or by designing states corresponding to \( a_k = i \).

For the first approach, create vertices \( 1 \ldots n \), and put an edge \( i \rightarrow j \) if \( a_i < a_j \), with
length 1. Then any longest increasing subsequence corresponds to a path in this graph,
and it suffices to find the longest path.

For the dynamic program, first consider the brute force search for finding the longest
sequence ending at \( i \):

```
LONGEST(i)
1. Initialize \( L[i] \leftarrow 1 \).
2. For \( j = 1 \) to \( i - 1 \)
   (a) If \( a[j] < a[i] \)
      i. \( L[i] = \max \{ L[i], \text{LONGEST}(j) + 1 \} \).
   (b) Return \( L[i] \).
```

This lends itself to a dynamic program fairly directly: note that only the location of
\( i \), matters for the transition. Iteratively, this leads to the dynamic program:
1. $L[i]$: longest increasing subsequence ending at $i$.

2. Base case: $L[i] \geq 1$.

3. Transition:

$$L[i] = \max_{j<i, a_j < a_i} L[j] + 1.$$ 

This algorithm runs in $O(n^2)$ time. From the longest path in a DAG perspective, this is as fast as we can make it, since there can be $n^2$ edges. This iterative view on the other hand allows us to make the algorithm run even faster, to $O(n \log n)$ time in fact.

The main idea is to compute the maximum value of $L[j]$ for some $a_j < a_i$ faster, in $O(\log n)$ time specifically. There is a simpler way of doing this, but the most systematic is to use the augmented binary search tree discussed in lecture 7 (which was also optional).

We make a balanced binary tree using all the $a_i$ values as keys, while attaching the values of $L[i]$ to the corresponding elements. Then computing the value of $L[i]$ equals to querying for the maximum value of $L[j]$ in the part of the tree to the left of $a_i$. This takes $O(\log n)$ time if we pre-build an augmented search tree with the $a_i$s as keys.