• **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 path on a DAG.
  – Recursive View.
  – Edit Distance.

• Textbook:
  – Chapter 6.2.
  – Relevant exercises 6.2, 6.3, 6.4, 6.10, and 6.11.

• From last time:
  – Can there be multiple base cases.
  – How to measure running time of a dynamic program: usually bounded by the total costs of computing the transitions.
  – More practice questions for test?

Last time we discussed finding longest paths on a grid where edges are oriented rightwards and downwards through the use of dynamic programming. This approach can also be extended to work on directed acyclic graphs (DAGs).

Such graphs don’t have cycles, and can be viewed as ordering the vertices as 1…n such that all edges $u \rightarrow v$ has $u < v$. Note that this ordering imposes a natural ordering of the states: a path ending at $v$ must come from some $u < v$. For the square grid that we considered last time, the row-major order (which incidentally is how we processed the states) is in fact one such ordering.

The longest path problem asks to find the path with maximum sum of lengths in a graph. The problem is often difficult because there could be cycles in the graph. On DAGs it is much easier for the same reason: if the last edge is $u \rightarrow v$, the path length is maximized by taking the longest path ending at $u$. This means we can design a dynamic program whose states are the endpoints of the paths.

1. State: $L[i]$: longest path ending at $i$. 

1
2. Base case: for all $i$, $L[i] \geq 0$ (path can now start anywhere)

3. Transition:

$$L[i] = \max_{j \to i} (L[j] + l_{j \to i}).$$

The base case can be set up in two alternate ways:

1. Only one base case: create a new vertex 0, have $0 \to i$ with length 0 ($l_{0 \to i} = 0$), and base-case being $L[0] = 0$.

2. Have no base case, and the transition being

$$L[i] = \max \left\{ 0, \max_{j \to i} (L[j] + l_{j \to i}) \right\}.$$

In general there is quite a bit of flexibility in designing the base case: it’s just there to get the dynamic program started. Most of the intricacies are in designing the states and the transitions.

The running time of such a dynamic program is dominated by the cost of the transitions. In this case here, the number of states is $n$, while the number of transitions is $m$: one per edge (ignoring the base case). So the total cost is $O(m)$ if we can ignore degree 0 vertices.

DAGs essentially summarize the iterative view of dynamic programming: the states are arranged in some ordering, the algorithm iterates through the states, and computes them based on the results of previous ones, whose answers have already been computed. This view requires in addition to the states, base case, and transition, and ordering of the states. Such an ordering is crucial for correctness: when we compute on each state, it needs to draw upon information that we’ve already computed.

The other problem to discuss is edit distance: the minimum number of inserts, deletes, and substitutions needed to transform one string into another. For example, the edit distance between $ABCD$ and $DABC$ is 2: insert a $D$ at the start, and delete the last $D$.

We will show an algorithm that computes the edit distance of two length $n$ strings in $O(n^2)$ time. To derive the state, consider what happens to the last character of $x[1 \ldots n]$ and $y[1 \ldots n]$:

1. They are the same, in which case we can find the best way to edit $x[1 \ldots n - 1]$ to $y[1 \ldots n - 1]$.

2. We transformed $x[n]$ into $y[n]$, in which case we also need to edit $x[1 \ldots n - 1]$ to $y[1 \ldots n - 1]$, but perform 1 more update for $x[n]$ and $y[n]$.

3. (at least) one of them was inserted / deleted, in which case we either need to edit $x[1 \ldots n]$ to $y[1 \ldots n - 1]$, or edit $x[1 \ldots n - 1]$ to $y[1 \ldots n]$. 

2
Note that the creation of these transitions is **lossy**: we are implicitly allowing for the case where \(x[n]\) was deleted, as well as \(y[n]\) was inserted. This is fine because we will covered all possible solution sequences.

This reasoning leads to a dynamic program:

1. **States**: \(E[i, j]\): minimum edit distance to transform \(x[1 \ldots i]\) to \(y[1 \ldots j]\).
2. **Base case**: \(E[0, 0] = 0\).
3. **Transition**:

\[
E[i, j] = \min \begin{cases} 
E[i - 1, j - 1] + 0 & \text{if } x[i] = y[j] \\
E[i, j - 1] + 1 & \text{otherwise} \\
E[i - 1, j] + 1 & 
\end{cases}
\]

(hiding out of bounds cases)

4. **Ordering of states**: row major order, \([1, 1] \ldots [1, n], [2, 1] \ldots [2, n] \ldots\)

This can be turned into code as:

1. Initialize \(E[0, 0] = 0\), \(E[i, j] = \infty\) for all other \(i, j\).
2. For \(i = 1 \ldots n\)
   
   (a) For \(j = 1 \ldots n\)
      
      i. If \(i > 0\) and \(j > 0\) \(E[i][j] = \min\{E[i][j], E[i-1][j-1] + 1\}\).
      ii. If \(i > 0\) and \(j > 0\) and \(x[i] = y[j]\) \(E[i][j] = \min\{E[i][j], E[i-1][j-1]\}\).
      iii. If \(i > 0\) \(E[i][j] = \min\{E[i][j], E[i-1][j]\}\).
      iv. If \(j > 0\) \(E[i][j] = \min\{E[i][j], E[i][j-1]\}\).

This array for the example of \(x = DABC\) and \(y = ABCD\) is:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Note that the entries with \(x[i] = y[j]\) are underlined to indicate the different type of transition (\(+0\) from \(E[i-1][j-1]\)).

The text book runs through this with a more thorough example.