In this lecture, we continue our exploration of the dynamic programming technique by examining two more problems where the technique is useful.

Note. The problem of finding a longest common sequence in two strings that was covered at the beginning of the lecture is described in Lecture 8.

1. Edit distance

The length of the longest common subsequence can be interpreted as a measure of how similar two strings are. A more sophisticated measure of the similarity between different strings is known as edit distance.

Definition 9.1 (Edit distance). The edit distance between two strings \( x_1, \ldots, x_m \) and \( y_1, \ldots, y_n \) is the minimum number of edit operations required to transform \( x \) into \( y \), where the 3 possible edit operations are:

- **Adding**: a letter to \( x \),
- **Deleting**: a letter from \( x \), and
- **Replacing**: a letter in \( x \) with another one.

For example, the edit distance between the strings POLYNOMIAL and EXPONENTIAL is 6, as this is the minimum number of edit operations required to go from one string to the other:

\[
\text{--POLYNOMIAL} \quad \text{EXPONENTIAL}
\]

In this example, the full list of operations that transformed POLYNOMIAL into EXPONENTIAL is as follows.

\[
\text{POLYNOMIAL} \\
\text{E-POLYNOMIAL} \quad \text{(Add E)} \\
\text{EXPOLYNOMIAL} \quad \text{(Add X)} \\
\text{EXPONYNOMIAL} \quad \text{(Replace L with N)} \\
\text{EXPONENOMIAL} \quad \text{(Replace Y with E)} \\
\text{EXPONEN-MIAL} \quad \text{(Delete O)} \\
\text{EXPONENTIAL} \quad \text{(Replace M with T)}
\]

A basic computational problem is to find the edit distance between two strings.

Definition 9.2 (Edit distance problem). An instance of the edit distance problem is two strings \( x_1, \ldots, x_m \) and \( y_1, \ldots, y_n \); the valid solution to an instance is the edit distance between \( x \) and \( y \).
We can again use the dynamic programming method to solve this problem by considering the subproblems obtained by computing the edit distance between prefixes of $x$ and $y$. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i, j) = \text{edit distance between } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$ 

When $i = 0$ or $j = 0$, the edit distance is easy to compute: it is exactly the length of the other string. So $M(i, 0) = i$ and $M(0, j) = j$.

What about for the other entries? If $x_i = y_j$, then we can match those characters together and we get $M(i, j) = M(i-1, j-1)$. If not, we have three choices:

1. Replace $x_i$ with $y_j$. In this case, we get $M(i, j) = M(i-1, j-1) + 1$.
2. Delete $x_i$. With this choice, $M(i, j) = M(i-1, j) + 1$.
3. Add the character $y_j$ to the string $x$ right before $x_i$. This choice gives us $M(i, j) = M(i, j-1) + 1$.

To compute $M(i, j)$, we want to choose the option among the ones above that has minimum value. So this means that we get

$$M(i, j) = \min \begin{cases} M(i-1, j-1) & \text{if } x_i = y_j \\ M(i-1, j-1) + 1 & \text{if } x_i \neq y_j \\ M(i-1, j) + 1 \\ M(i, j-1) + 1 \end{cases}.$$ 

As long as we compute the values of $M$ in an order where $M(i-1, j-1)$, $M(i-1, j)$, and $M(i, j-1)$ have all been computed before we compute $M(i, j)$, computing this value takes $\Theta(1)$ time. We can again proceed row by row, obtaining an algorithm that looks very similar to the one we used to compute the length of the longest common subsequence.

**Algorithm 1: EditDistance**($x_1, \ldots, x_m, y_1, \ldots, y_n$)

for $i = 1, \ldots, m$ do $M[i, 0] = i$;

for $j = 1, \ldots, n$ do $M[0, j] = j$;

for $i = 1, \ldots, m$ do

for $j = 1, \ldots, n$ do

if $x_i = y_j$ then

$r = M[i-1, j-1]$;

else

$r = M[i-1, j-1] + 1$;

$M[i, j] = \min\{M[i-1, j] + 1, M[i, j-1] + 1, r\}$;

return $M[m, n]$;

The time complexity of this algorithm is again $\Theta(mn)$.

2. **Weighted Interval Scheduling**

Let’s revisit the interval scheduling problem, with a slight twist.
Definition 9.3 (Weighted interval scheduling). In the weighted interval scheduling problem, an instance is a set of \(n\) pairs of intervals that have start and finish times \((s_1, f_1), \ldots, (s_n, f_n)\) and positive weights \(w_1, \ldots, w_n\). A valid solution is a subset \(I \subseteq \{1, 2, \ldots, n\}\) of non-overlapping intervals that maximizes \(\sum_{i \in I} w_i\).

Let’s even simplify the problem slightly to only aim to find the total weight of a valid solution to the weighted interval scheduling problem.

Without weights, we saw that the greedy algorithm that sorts the intervals by finish time and considers them in that order solves the problem. It’s not clear whether greedy algorithms can also solve the weighted version of the problem, but as it turns out it is still useful to sort the intervals by finish times and consider them in order. But instead of making greedy decisions, we can define the following subproblems.

- **Subproblems.** For \(k = 0, 1, \ldots, n\), define \(W[k]\) to be the maximum sum of weights of a set \(I \subseteq \{1, \ldots, k\}\) of non-overlapping intervals taken from the \(k\) intervals with earliest finish times.

We can now design a dynamic programming algorithm that computes \(W[1], W[2], \ldots, W[n]\) in that order and returns \(W[n]\).

The values of the first subproblems are easy to determine: \(W[0] = 0\) (since in this case there is no interval to select) and \(W[1] = w_1\). Our task is now to figure out how to compute \(W[k]\) given the values of \(W[j]\) for each \(1 \leq j < k\). Examining this problem carefully, we realize there are exactly two possibilities to consider:

**Case 1:** \(k \notin \text{OPT}(k)\): The first possibility is that the \(k\)th interval is not in the maximum-weight subset \(I \subseteq \{1, 2, \ldots, k\}\) of non-overlapping intervals. If that’s the case, then

\[ W[k] = W[k-1] \]

since the optimal set is the same whether or not we consider the \(k\)th interval.

**Case 2:** \(k \in \text{OPT}(k)\): Otherwise, the \(k\)th interval is in the maximum-weight subset \(I \subseteq \{1, 2, \ldots, k\}\) of non-overlapping intervals. Then, letting \(j^* < k\) be the largest index such that \(f_{j^*} < s_k\), we have that

\[ W[k] = W[j^*] + w_k \]

since any set \(I \subseteq \{1, \ldots, k\}\) of non-overlapping intervals that contains interval \(k\) cannot include any of the intervals \(j^* + 1, \ldots, k - 1\).

We can compute the value of both candidates to determine which is correct. The resulting dynamic programming algorithm is as follows.

**Algorithm 2:** \textsc{WeightedIS}((\(s_1, f_1\), \ldots, \(s_n, f_n\), \(w_1, \ldots, w_n\))

\begin{algorithmic}
\State (Sort intervals so that \(f_1 \leq f_2 \leq \cdots \leq f_n\));
\State \(W[0] \leftarrow 0\);
\For{\(k = 1, \ldots, n\) do}
\State \(j^* \leftarrow \max\{0 \leq i < k : f_i < s_k\}\);
\State \(W[k] = \max\{W[k-1], W[j^*] + w_k\}\);
\EndFor
\State return \(W[n]\);
\end{algorithmic}

What is the running time of this algorithm? The naïve algorithm for identifying \(j^*\) in each iteration of the for loop is a linear search that takes time \(O(k)\), for a total time
complexity $O(n^2)$; but since the intervals are sorted by finish time we can use binary search instead. The total time complexity of the resulting implementation of the algorithm is then $O(n \log n)$.

As a final step, we should see how to revise the algorithm to find the maximum-weight set of non-overlapping intervals itself, and not just its total weight. We leave this as an exercise for now.

**Exercise 9.1.** Modify the dynamic programming solution to solve the (original version of the) Weighted Interval Scheduling problem.