4.1 Overview

Dynamic programming is a method that follows a similar theme to other techniques learned this semester: In order to solve a large, complicated problem, we first split it into smaller sub-problems. With dynamic programming, the basic idea is to break the problem down into many closely related sub-problems, solve them, and then store their results for later use. In this way, dynamic programming avoids recomputing the results of the sub-problems, allowing it to achieve better runtimes than naive approaches. In this lecture, we will demonstrate the technique through two examples: the longest increasing subsequence problem and the knapsack problem.

4.2 Longest Increasing Subsequence

Definition 4.1 Given an input array $A$, a subsequence is a list of numbers that appears in the same order as the elements of $A$, though not necessarily consecutively. A subsequence $x_1, x_2, \ldots, x_k$ is increasing if for all $1 \leq i < k$, $x_i < x_{i+1}$. The longest increasing subsequence of $A$ is then the increasing subsequence in $A$ with maximal length.

For instance, consider the array $\{4, 2, 5, 3, 9, 7, 8, 10, 6\}$. An example of a subsequence is $\{4, 2, 5\}$, an example of an increasing subsequence is $\{2, 3, 8\}$, and the longest increasing subsequence is $\{2, 5, 7, 8, 10\}$ (or $\{2, 3, 7, 8, 10\}$).

In this example, we will try to find the length of the longest increasing subsequence of the following array:

$$A = \{4, 2, 3, 5, 1, 7, 10, 8\}$$

The first step in creating a dynamic programming solution is to relate the problem recursively to smaller sub-problems. We will therefore begin by focusing on just the last element of this sequence, 8. We then have two options to consider for this element:

Option 1: 8 is not in the longest increasing subsequence.

Option 2: 8 is in the longest increasing subsequence.

Dealing with option 1 is easy. We just recurse on all of the other elements in $A$, i.e. $\{4, 2, \ldots, 10\}$. Option 2 is trickier to deal with. To see why, consider that in this example, the LIS of $\{4, 2, 3, 5, 1, 7, 10\}$ is $\{2, 3, 5, 7, 10\}$. 10 > 8 so we clearly cannot add 8 to the end of this sequence. Our goal then, should be to find a transition function that properly relates the solution for this sub-problem to that of other sub-problems.
To this end, we will define \( a[i] \) to be the length of the longest increasing subsequence of \( A \) that ends at the \( i \)th element of \( A \). We can determine the value of \( a[i] \) in the following way. Consider all of the \( i - 1 \) elements in \( A \) both previous to \( A[i] \) and smaller than it, i.e. \( \{ j \in [1, i - 1] \mid A[i] > A[j] \} \). These are the elements that \( A[i] \) could be appended to in an increasing subsequence. Choose the \( a[j] \) with maximal value, and set \( a[i] = a[j] + 1 \) (effectively adding element \( A[i] \) to the end of the longest increasing subsequence possible). So we have:

\[
a[i] = \begin{cases} 
1 & \text{if } A[i] < A[j] \forall j < i \\
1 + \max_{j<i, A[j]<A[i]} A[j] & \text{otherwise}
\end{cases}
\]

\( a[i] \) depends on all of the elements before it, so when we create our dynamic programming table, we will start at \( a[1] \) and then progressively fill it in from left to right. Once we’ve determined values for all \( a[i] \), we just select the one with the maximum value, and the algorithm is complete.

**Algorithm 1** Dynamic programming method for LIS

**Require:** \( A \) is an array of length \( n \).

**Ensure:** \( LIS \) is the length of the longest increasing subsequence of \( A \).

**procedure** LONGESTINCREASINGSUBSEQUENCE(A)

\[
LIS = 0 \\
\text{for } i \text{ in } \{1, 2, \cdots, n\} \text{ do} \\
\quad a[i] = 1 \\
\quad \text{for } j \text{ in } \{1, 2, \cdots, i - 1\} \text{ do} \\
\qquad \text{if } A[j] < A[i] \text{ and } a[j] + 1 > a[i] \text{ then} \\
\qquad \quad a[i] = a[j] + 1 \\
\qquad \text{end if} \\
\quad \text{end for} \\
\quad \text{if } a[i] > LIS \text{ then} \\
\qquad LIS = a[i] \\
\quad \text{end if} \\
\text{end for} \\
\text{return LIS}
\]

**end procedure**

### 4.3 Knapsack

The knapsack problem is stated as follows. There is a knapsack that can hold items of total weight at most \( W \). There is also a set \( I \) of \( n \) items available. Each item \( i \in I \) has an associated weight \( w_i \) and value \( v_i \). The goal is to select a subset of the items to place in the knapsack, so that the total weight is less than \( W \) and the total value is maximized. Stated in another way, we wish to choose the subset \( K \subseteq I \) that maximizes \( \sum_{i \in K} v_i \), subject to \( \sum_{i \in K} w_i \leq W \).

As before, we will begin by breaking the problem down into smaller sub-problems. We look at the last item, and consider two possible options:

**Option 1:** The last item is not in the knapsack.

**Option 2:** The last item is in the knapsack.
To compare these two options, we will define \( a[i, j] \) to be the maximum total value that can be obtained from using only the first \( i \) items, with a weight capacity of \( j \). We see that if we choose option 1, and do not add item \( i \) to the knapsack, we can just maximize value over the remaining \( i - 1 \) items, i.e. \( a[i, j] = a[i - 1, j] \). If we choose option 2, we add value \( v_i \) to the knapsack, and then maximize value over the remaining \( i - 1 \) items, keeping in mind that the capacity must also be decreased by weight \( w_i \), i.e. \( a[i, j] = v_i + a[i - 1, j - w_i] \). We will choose the option that provides maximal value, so we have:

\[
a[i, j] = \max \left\{ \begin{array}{ll}
a[i - 1, j] & \text{(do not put item } i \text{ in knapsack)} \\
v_i + a[i - 1, j - w_i] & \text{(put item } i \text{ in knapsack)}
\end{array} \right.
\]

We must also define base cases, namely whenever \( i = 0 \), or \( j \leq 0 \), \( a[i, j] = 0 \) (because we can’t add items if we have no items left or if the capacity is spent). To construct the dynamic programming table, we make a two-dimensional table, with \( i \) on the horizontal axis going from 1 to \( n \), and \( j \) on the vertical axis going from 1 to \( W \). We then fill in the table, starting at \( a[1, 1] \) and filling in each row from left to right. Once we have completely filled in the table, our answer will be the value \( a[n, W] \).

\[\text{Algorithm 2 Dynamic programming method for knapsack problem}\]

\begin{algorithm}
\textbf{Require:} \( I \) contains \( n \) items. Each \( i \in I \) has a weight \( w_i \) and a value \( v_i \). \( W \) is maximum capacity.
\textbf{Ensure:} \( a[n, W] \) is the maximum possible value we can place into knapsack.

\begin{procedure}
\caption{Knapsack(\( I, W \))}
\For{\( i \in \{1, 2, \ldots, n\} \)}
\For{\( j \in \{1, 2, \ldots, W\} \)}
\State \( \text{optionOne} = a[i - 1, j] \)
\State \( \text{optionTwo} = v_i + a[i - 1, j - w_i] \)
\State \( a[i, j] = \max\{\text{optionOne, optionTwo}\} \)
\EndFor
\EndFor
\end{procedure}
\end{algorithm}