We now turn our attention to another general technique for designing algorithms: dynamic programming. The main idea for this technique can be summarized as follows: We can solve a big problem by

1. Breaking it up into smaller sub-problems;
2. Solving the sub-problems from smallest to largest; and
3. Storing solutions along the way to avoid repeating our work.

Note the similarity with the greedy method technique: there also we break up the problem into smaller sub-problems and solve them one by one in order. The main difference of the dynamic programming technique lies in how we solve these sub-problems: whereas with the greedy algorithm we usually resort to a simple local decision criterion, with dynamic programming we will be critically relying on using the work we have done so far to solve the current sub-problem. All of this discussion is best explained with illustrations. We begin with a simple example—computing Fibonacci numbers—and then will consider a range of other problems where dynamic programming is particularly useful over the next few lectures.

### 1. Fibonacci numbers

**Definition 8.1.** The *Fibonacci sequence* of numbers is the famous sequence of integers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ defined by the rule

\[
F_n = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F_{n-1} + F_{n-2} & \text{if } n \geq 2.
\end{cases}
\]

The definition can be turned directly into a simple algorithm:

**Algorithm 1:** 

```plaintext
Fib1(n)
if n = 0, return 0;
if n = 1, return 1;
return Fib1(n - 1) + Fib1(n - 2);
```

Is this a good algorithm for computing Fibonacci numbers? Not at all! It’s correct, but horribly inefficient. Drawing the recurrence tree, we see that the amount of work performed by this algorithm is exponential in $n$. 

In fact, the time complexity of this algorithm satisfies
\[ T(n) \geq T(n-1) + T(n-2). \]
so \( T(n) \geq F_n \) so the time complexity of the algorithm is bounded below by the Fibonacci numbers themselves. (If you’re interested, you can prove directly that \( F_n \geq 2^n/2 \) for every \( n > 6 \) by induction; you can also read up on Binet’s formula to obtain a closed-form expression for \( F_n \).)

We can do better than Fib1 by examining the recurrence tree we already drew to spot a glaring inefficiency: the intermediate values \( F_{n-1}, F_{n-2}, F_{n-3}, \ldots \) are each recomputed multiple times within this tree. Eliminating this inefficiency can be done easily with a simple application of the dynamic programming technique:

- **Subproblems:** Compute \( F_1, F_2, F_3, \ldots, F_n \) in that order.

**Algorithm 2:** Fib2 \((n)\)

\[
\begin{align*}
\text{if } n &= 0 \text{ return } 0; \\
A[0] &\leftarrow 0; \\
A[1] &\leftarrow 1; \\
\text{for } i &= 2, \ldots, n \text{ do} \\
\text{return } A[n];
\end{align*}
\]

The algorithm we obtain now run in time that is \textit{linear} in \( n \). It’s hard to overstate how much of an improvement this represents over the original algorithm: whereas Fib1 can’t even compute reasonably small values like \( F_{50} \) even on the world’s biggest supercomputers, Fib2 can easily compute much larger values—say, \( F_{5000000} \)—on any computer.

2. Text segmentation

Can the following sequence of letters be split up into (actual English) words?

\textit{thecustomofdrinkingorangejuicewithbreakfastisnotverywidespread}

In this case, yes:

\textit{the custom of drinking orange juice with breakfast is not very widespread}

And in general, we can ask the same question about any string of letters to obtain the \textit{text segmentation} problem.
Definition 8.2 (Text segmentation problem). Given an array $A$ of $n$ letters from the sets \{a, b, \ldots, z\} and access to a function $\text{IsWord}$ that for any sequence $w$ of letters returns True if $w$ is a word in the English language and False otherwise, determine whether $A$ can be split into a sequence of English words or not.

The example we saw above suggests that a greedy algorithm would provide a good solution to the text segmentation problem: just keep calling $\text{IsWord}$ on the words $A[1]$, $A[1..2]$, $A[1..3]$, \ldots, $A[1..k]$ until it returns True, make that the first word, then continue from $A[k+1]$ onwards. But if you try to prove that this algorithm works, you will find that no argument can establish its correctness... for good reason! Consider the input

them examples

that can be separated into the two words them examples—if you run the simple greedy algorithm on this input, it will first identify the word the, then me, but it will then be stuck with the non-word examples and incorrectly report that the string cannot be split into words.

Similarly, an alternative greedy algorithm that always try to extract the longest possible word at the beginning of the string will extract the word theme and be stuck with examples on the input above, so this algorithm does not work either. In fact, I’m not aware of any greedy algorithm that correctly solves the text segmentation problem. Instead, to get a correct and efficient solution, we need to turn to dynamic programming.

The key, as always with the dynamic programming technique, is to identify the right subproblems. And here, there is one option that seems quite natural: determining if the initial part $A[1..k]$ of the array can be split into words.

- **Subproblems:** For $k = 1, 2, \ldots, n$, determine if $A[1..k]$ can be split into words.

Solving these subproblems in order, how can we determine if $A[1..k]$ can be split into words or not? Simple: It can be split if and only if there is some index $j < k$ for which $A[1..j]$ can be split into words and $A[j + 1..k]$ is a word! This is enough to complete our dynamic programming algorithm.

\begin{algorithm}
\begin{algorithmic}
\State $S[0] \leftarrow \text{True}$;
\For {$k = 1, \ldots, n$} \Do
  \State $S[k] \leftarrow \text{False}$;
  \For {$j = 0, \ldots, k - 1$} \Do
    \If {$S[j]$ and $\text{IsWord}(A[j+1..k])$} \Then
      \State $S[k] \leftarrow \text{True}$;
    \EndIf
  \EndFor
\EndFor
\Return $S[n]$;
\end{algorithmic}
\end{algorithm}

If we assume that each call to $\text{IsWord}$ has time complexity $O(1)$, then the total time complexity of the algorithm is $O(n^2)$.

Note that the algorithm can also be easily modified so that when a string can be converted into a sequence of words, the algorithm not only returns true but outputs a valid segmentation of the string. This is left as an exercise.
3. LONGEST INCREASING SUBSEQUENCE

An increasing subsequence in a given sequence of numbers is a subset of those numbers (not necessarily all next to each other in the original sequence) in increasing order. For instance, in the sequence

5 2 1 4 3 1 6 9 2

we can find the increasing subsequence 2 3 6 9 since those numbers are increasing and are in that order within the original sequence:

5 2 1 4 3 1 6 9 2

Definition 8.3 (Longest increasing subsequence problem). Given a sequence $A$ of $n$ positive integers, find the length of the longest increasing subsequence of $A$.

Once again, we can use the dynamic programming technique to design an efficient algorithm for the longest increasing subsequence problem. But we have to be careful in how we choose which subproblems to solve. The most natural idea is to find the longest subsequence within each prefix of the original sequence.

- **Candidate Subproblems:** For $k = 1, 2, \ldots, n$, find the length of the longest increasing subsequences in $A[1..k]$.

But how can we compute the value of $A[1..k]$ when we have solved the earlier subproblems? Unfortunately, just knowing the length of the longest common subsequence in $A[1..j]$ for every $j < k$ does not give us enough information to compute $A[1..k]$ because we don’t know to which subsequence we can append the number $A[k]$ and make a longer increasing subsequence. So for this problem it turns out that we want to consider slightly different subproblems.

- **Subproblems:** For $k = 1, 2, \ldots, n$, find the length of the longest increasing subsequences in $A[1..k]$ that includes $A[k]$ itself.

The small change makes it much easier to solve the subproblems: find the value $j < k$ where (i) $A[1..j]$ has the longest increasing subsequence; and (ii) $A[j] < A[k]$. This solution yields the following algorithm.

**Algorithm 4: LIS($A[1..n]$)**

```plaintext
for $k = 1, \ldots, n$ do
    $S[k] \leftarrow 1$;
    for $j = 1, \ldots, k - 1$ do
        if $A[j] < A[k]$ then
            $S[k] \leftarrow \max\{S[k], S[j] + 1\}$;
    return $\max\{S[1], S[2], \ldots, S[n]\}$;
```

Note that because of our choice of subproblems, the value to return at the end is not $S[n]$ (as this would correspond to the longest increasing subsequence of $A[1..n]$ that includes $A[n]$ itself) but rather the maximum value $S[k]$ over all $k = 1, 2, \ldots, n$.

It’s helpful to visualize what the algorithm does on an example:

<table>
<thead>
<tr>
<th>Input</th>
<th>5 2 1 4 3 1 6 9 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S[k]$</td>
<td>1 1 1 2 2 1 3 4 2</td>
</tr>
<tr>
<td>Coming from $j$</td>
<td>0 0 2 2 0 4 7 3</td>
</tr>
</tbody>
</table>
Exercise 8.1. Show how to modify the LIS algorithm to return a longest increasing sequence instead of just its length. (Hint. Consider storing the information from the bottom line of the table in a separate array.)

The time complexity of the LIS algorithm is $O(n^2)$. Note that this is not best possible: it is also possible to solve the longest increasing subsequence in time $O(n \log n)$.

4. Longest Common Subsequence

A common subsequence between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is a string $z_1, \ldots, z_k$ for which there are indices $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ where for each $\ell \leq k$, $z_\ell = x_{i_\ell} = y_{j_\ell}$. For example, in the pair of strings $x = \text{POLYNOMIAL}$, $y = \text{EXPONENTIAL}$, the string $z = \text{PONIAL}$ is a subsequence of $x$ and $y$. (As are the strings $\text{POL}$, $\text{PNA}$, etc.)

Definition 8.4 (Longest common subsequence problem). An instance of the longest common subsequence (LCS) problem is a pair of strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. The valid solution to an instance is the length of the longest common subsequence of $x$ and $y$.

The natural way to break down the LCS problem into smaller subproblems is to consider the longest common subsequence of prefixes of $x$ and $y$. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i, j) = \text{length of LCS of } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When $i$ or $j$ is 0 (which corresponds to $x$ or $y$ being an empty string), then

$$M(0, j) = 0 \quad \text{and} \quad M(i, 0) = 0.$$

Given that we have computed $M(i - 1, j)$, $M(i, j - 1)$, and $M(i - 1, j - 1)$, can we now compute $M(i, j)$? Indeed we can!

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

This gives the following algorithm.

Algorithm 5: LCS($x_1, \ldots, x_m, y_1, \ldots, y_n$)

\[
\begin{array}{l}
\text{for } i = 1, \ldots, m \text{ do } M[i, 0] = 0; \\
\text{for } j = 1, \ldots, n \text{ do } M[0, j] = 0; \\
\text{for } i = 1, \ldots, m \text{ do } \\
\hspace{1em} \text{for } j = 1, \ldots, n \text{ do } \\
\hspace{2em} M[i, j] = \max\{M[i - 1, j], M[i, j - 1]\}; \\
\hspace{2em} \text{if } x_i = y_j \text{ then } M[i, j] = \max\{M[i, j], M[i - 1, j - 1] + 1\}; \\
\text{return } M[m, n];
\end{array}
\]

We can picture the algorithm as filling out the table of values $M[i, j]$, row by row. With the instance $x = \text{ALGORITHM}$, $y = \text{ANALYSIS}$, the table looks as follows.

The time complexity of the algorithm is $\Theta(mn)$. 
The LCS algorithm determines the length of the longest common subsequence between the two strings given as input, but it does not identify the subsequence itself. What if we want to identify it? There are a number of different ways we can modify the algorithm to do so. Or, if we have already computed the matrix $M$ of LCS lengths for all prefixes, we can identify the LCS itself by working backwards from $M(m, n)$.

**Algorithm 6: PrintLCS($x, y, M, i, j$)**

```algorithm
if $i > 1$ and $M[i, j] = M[i - 1, j]$ then
    PrintLCS($x, y, M, i - 1, j$);
else if $j > 1$ and $M[i, j] = M[i, j - 1]$ then
    PrintLCS($x, y, M, i, j - 1$);
else
    /* We must have matched $x_i = y_j$ */
    PrintLCS($x, y, M, i - 1, j - 1$);
    Print $x_i$;
```

Calling PrintLCS($x, y, M, m, n$) will print the longest common subsequence of $x$ and $y$. Interestingly, by calling it with any other $i \leq m$ and $j \leq n$ we can also print the longest common subsequence of any prefixes of $x$ and $y$ just as efficiently.