Lecture 11: Dynamic Programming
First Example: Stairs Climbing

Problem: Suppose you can take 1 or 2 stairs with one step. How many different ways can you climb \( n \) stairs?

Solution: Let \( f(n) \) be the number of different ways to climb \( n \) stairs.

\[
\begin{align*}
f(1) &= 1, \\
f(2) &= 2, \\
f(3) &= 3, \\
&\quad \ldots \\
f(n) &= f(n - 1) + f(n - 2)
\end{align*}
\]

Q: How to compute \( f(n) \)?
Solving the recurrence by recursion

\[ f(1) = 1, \ f(2) = 2, \ f(3) = 3, \ldots \]
\[ f(n) = f(n-1) + f(n-2) \]

Running time?

Between \(2^{n/2}\) and \(2^n\).

A more complicated analysis yields \(\Theta(\varphi^n)\) where \(\varphi \approx 1.618\) is the golden ratio.

Q: Why so slow?

A: Solving the same subproblem many many times.
Solving the recurrence by recursion

\[ f(1) = 1, f(2) = 2, f(3) = 3, \ldots \]
\[ f(n) = f(n-1) + f(n-2) \]

**Dynamic programming:**
- Used to solve recurrences
- Avoid solving a subproblem more than once by memorization
- Can be either top-down or bottom-up
  - Bottom-up is usually more efficient in practice
- “Programming” here means “planning”, not coding!

```plaintext
F(n):
allocate an array A of size n
A[1] ← 1
for i = 3 to n
return A[n]
```

**Running time:** \( \Theta(n) \)

**Space:** \( \Theta(n) \) but can be improved to \( \Theta(1) \) by freeing array entries that are no longer needed.
The Rod Cutting Problem

**Problem:** Given a rod of length $n$ and prices $p_i$ for $i = 1, \ldots, n$, where $p_i$ is the price of a rod of length $i$. Find a way to cut the rod to maximize total revenue.

<table>
<thead>
<tr>
<th>length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

- (a) 9
- (b) 1 8
- (c) 5 5
- (d) 8 1
- (e) 1 1 5
- (f) 1 5 1
- (g) 5 1 1
- (h) 1 1 1 1

The correct way to cut the rod is shown in (c).
Rod Cutting: The Algorithm

**Define:** Let $r_n$ be the maximum revenue obtainable from cutting a rod of length $n$.

**Recurrence:** $r_n = \max\{p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}, r_0 = 0$
- $p_n$ if we do not cut at all
- $p_1 + r_{n-1}$ if the first piece has length 1
- $p_2 + r_{n-2}$ if the first piece has length 2
- ...

```java
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← -∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j - i])
    r[j] ← q
return r[n]
```

**Running time:** $\Theta(n^2)$
Reconstructing the Solution

**Idea:** Remember the optimal decision for each subproblem.

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j - i] then
            q ← p[i] + r[j - i]
            s[j] ← i
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j - s[j]
```

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[i]</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>r[i]</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>13</td>
<td>17</td>
<td>18</td>
<td>22</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>s[i]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
Dynamic Programming: Summary

**Structure:** Analyze structure of an optimal solution, and thereby choose a definition of subproblems.

**Recurrence:** Establish the relationship between the optimal value of the problem and those of some subproblems (optimal substructure).

**Bottom-up computation:** Compute the optimal values of the smallest subproblems first, save them in the table. Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.

**Construction of optimal solution:** Record the optimal decisions made for each subproblem. At the end, assemble the optimal solution by tracing the back computation in the previous step.

**Remark:** The first two steps are interdependent. And they are the most important steps. The last two steps are usually straightforward.
Lecture 12: 2D Dynamic Programming
Longest Common Subsequence

Problem: Given two sequences $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$, we say that $Z$ is a common subsequence of $X$ and $Y$ if $Z$ has a strictly increasing sequence of indices $i$ and $j$ of both $X$ and $Y$ such that we have $x_{i_p} = y_{j_p} = z_p$ for all $p = 1, 2, \ldots, k$. The goal is to find the longest common subsequence of $X$ and $Y$.

Ex:

$X$: A B A C B D A B

$Y$: B D C A B A

$Z$: B C B A

Application: diff
The Recurrence

**Def:** Let $c[i, j]$ to be the length of the longest common subsequence of $X[1..i]$ and $Y[1..j]$.

**Observations:** The problem is equivalent to finding the maximum matching between $X$ and $Y$ such that matched pairs don’t cross.

```
X:  1  2  \cdots m
    \   i
    \  =
Y:  1  2  \cdots n
```

The recurrence:

- **Case 1:** If $x_i = y_j$, then we match $x_i$ and $y_j$. By doing so, we will not miss the optimal solution. (If OPT doesn’t match them, we can change it so that they are matched.)
- **Case 2:** If $x_i \neq y_j$, then either $x_i$ or $y_j$ is not matched. So the problem reduces to either $c[i - 1, j]$ or $c[i, j - 1]$. 
The Recurrence and Algorithm

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j
\end{cases}
\]

**Running time:** $\Theta(mn)$

**Space:** $\Theta(mn)$, can be improved to $\Theta(m + n)$ if we only need to return the optimal length.
Reconstruct the Optimal Solution

Print-LCS\((b, i, j)\):
\[
\text{if } i = 0 \text{ or } j = 0 \text{ then return } \\
\text{if } b[i, j] = " \leftarrow " \text{ then } \\
\quad \text{Print-LCS}(b, i - 1, j - 1) \\
\quad \text{print } x_i \\
\text{else if } b[i, j] = " \uparrow " \\
\quad \text{Print-LCS}(b, i - 1, j) \\
\text{else Print-LCS}(b, i, j - 1)\]
Longest Common Substring

**Problem:** Given two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$, we wish to find their longest common substring $Z$, that is, the largest $k$ for which there are indices $i$ and $j$ with $x_{i}x_{i+1} \ldots x_{i+k-1} = y_{j}y_{j+1} \ldots y_{j+k-1}$.

**Ex:**

$X: \textbf{DEADBEEF}$

$Y: \textbf{EATBEEF}$

$Z: \textbf{BEEF} \ //\text{pick the longest contiguous substring}$

**Note:** Brute-force algorithm takes $O(n^4)$ time.
The Recurrence

**Def:** \( d[i, j] \) = the length of the longest common substring of \( X[1..i] \) and \( Y[1..j] \). (Does this work?)

**Def:** \( d[i, j] \) = the length of the longest common substring of \( X[1..i] \) and \( Y[1..j] \) that ends at \( x_i \) and \( y_j \).

Q: Wait, are we changing the problem?

A: Yes, but it’s OK. The optimal solution to the original is just \( \text{max}\{d[i, j]\} \)

Recurrence:

- If \( x_i = y_j \), then the LCS of \( X[1..i] \) and \( Y[1..j] \) is just the LCS of \( X[1..i-1] \) and \( Y[1..j-1] \), plus \( x_i = y_j \)
- If \( x_i \neq y_j \), then there can’t be a common substring ending at \( x_i \) and \( y_j \)!

\[
d[i, j] = \begin{cases} 
    d[i-1, j-1] + 1 & \text{if } x_i = y_j \\
    0 & \text{if } x_i \neq y_j 
\end{cases}
\]
The Algorithm

let \( d[0..m,0..n] \) be a new array of all 0

\[ l_m \leftarrow 0, p_m \leftarrow 0 \]

for \( i \leftarrow 1 \) to \( m \)
  
  for \( j \leftarrow 1 \) to \( n \)
    
    if \( x_i = y_j \) then
      \[ d[i,j] \leftarrow d[i-1,j-1] + 1 \]
    
    if \( d[i,j] > l_m \) then
      \[ l_m \leftarrow d[i,j] \]
      \[ p_m \leftarrow i \]

for \( i \leftarrow p_m - l_m + 1 \) to \( p_m \)

print \( x_i \)

Note: For this problem, reconstructing the optimal solution just needs the location of the LCS.

Running time: \( \Theta(mn) \)

Space: \( \Theta(mn) \) but can be improved to \( \Theta(m + n) \).