Dynamic Programming

Another Technique

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn’t really refer to computer programming.
- Used for optimization problems:
  - Find a solution with the optimal value.
  - Minimization or maximization. (We’ll see both.)

Four-step method
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Assembly-Line Scheduling

Automobile factory with two assembly lines.

Problem: Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has \( n \) stations.
- \( 2^n \) subsets.
- Infeasible when \( n \) is large.
Dynamic Programming

Assembly-Line Scheduling

Note: a fastest way through station $S_{i,j}$ MUST have made use of a fastest way through either $S_{i,j-1}$ or $S_{i-1,j}$ - if not, we could reduce the time to finish by just picking a better previous route.

This is an example of a general idea applicable to all Dynamic Programming problems: an optimal solution to a problem contains within it optimal solutions to its subproblems. We denote this state of affairs as having the property of **optimal substructure**.

Recursive solution

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through = $f^*$.

\[
f^* = \min(f_1[n] + x_1, f_2[n] + x_2)
\]

\[
f_1[1] = c_1 + \alpha_{1,1}
\]

\[
f_2[1] = c_2 + \alpha_{2,1}
\]

For $j = 2, \ldots, n$:

\[
f_1[j] = \min(f_1[j-1] + \alpha_{1,j}, f_2[j-1] + \beta_{1,j-1} + \alpha_{1,j})
\]

\[
f_2[j] = \min(f_3[j-1] + \alpha_{2,j}, f_2[j-1] + \beta_{2,j-1} + \alpha_{2,j})
\]
Dynamic Programming

Assembly-Line Scheduling

For example:

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>9</td>
<td>18</td>
<td>20</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>$f_1$</td>
<td>12</td>
<td>16</td>
<td>22</td>
<td>25</td>
<td>30</td>
</tr>
</tbody>
</table>

$f^* = 35$

$p = 1$

Go through optimal way given by $i$ values. (Shaded path in earlier figure.)

Dynamic Programming

Assembly-Line Scheduling

If we try to compute $f_1[n]$ and $f_2[n]$ recursively, how many references to $f_1[1]$ and $f_2[1]$ are we going to have?

....

This is an instance of the "overlapping subproblems property": we would recompute the same subproblems over and over...

Dynamic Programming

Assembly-Line Scheduling

How do we solve the exponential explosion?

We can rewrite the algorithm so that

1. We compute $f_j(j)$ "bottom up": every time $f_j[j]$ accesses $f_j[j-1]$, the latter value is already computed;

2. We can compute $f_j[j]$ "top down": every time $f_j[j]$ accesses $f_j[j-1]$, and the latter value is not already computed, it computes it (recursively) and saves it (memoization technique - each value is computed only once)

Dynamic Programming

Assembly-Line Scheduling

We can rewrite the algorithm so that

1. We compute $f_j(j)$ "bottom up": every time $f_j[j]$ accesses $f_j[j-1]$, the latter value is already computed;

2. We can compute $f_j[j]$ "top down": every time $f_j[j]$ accesses $f_j[j-1]$, and the latter value is not already computed, it computes it (recursively) and saves it (memoization technique - each value is computed only once)
Dynamic Programming

Longest Common Subsequence

**Problem:** Given 2 sequences,

\[ X = \langle x_1, \ldots, x_m \rangle \text{ and } Y = \langle y_1, \ldots, y_n \rangle, \]

find a subsequence common to both whose length is longest. A subsequence does not have to be consecutive (like a substring), but must be in order.

**Potential application:** check to what extent two nucleotide sequences "match". A proportionately longer match should indicate larger similarity (in some sense).

---

**Example:**

```
springtime  horseback
pioneer    snowflake
maelstrom  heroically
becalm     scholarly
```

---

**Dynamic Programming**

Longest Common Subsequence

Brute-force algorithm:

For every subsequence of \( X \), check whether it's a subsequence of \( Y \).

Time: \( \Theta(n2^n) \).

- \( 2^n \) subsequence of \( X \) to check.
- Each subsequence takes \( \Theta(n) \) time to check: scan \( Y \) for first letter, from there scan for second, and so on.

---

**Dynamic Programming**

Longest Common Subsequence

How do we identify the "optimal substructure" property?

**Def.:** given a sequence \( X = \langle x_1, x_2, \ldots, x_m \rangle \), we define the \( i^{th} \) prefix of \( X \) for \( i = 0, 1, \ldots, m \), as \( X_i = \langle x_1, x_2, \ldots, x_i \rangle \).

**Theorem:** Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \), and \( Y = \langle y_1, \ldots, y_n \rangle \) be sequences, and let \( Z = \langle z_1, \ldots, z_k \rangle \) be any LCS of \( X \) and \( Y \). Then:

1. If \( x_m = y_n \), then \( z_k = x_m = y_n \) and \( Z \) is an LCS of \( X_{m-1} \) & \( Y_{n-1} \).
2. If \( x_m \neq y_n \), then \( z_k \neq x_m \rightarrow Z \) is an LCS of \( X_{m-1} \) & \( Y \).
3. If \( x_m \neq y_n \), then \( z_k \neq y_n \rightarrow Z \) is an LCS of \( X \) & \( Y_{n-1} \).
Dynamic Programming
Longest Common Subsequence

Proof:
1. If \( z_k \neq x_m \) (and \( y_n \)), we could append \( x_m = y_n \) to \( Z \) and obtain a common subsequence of \( X \) and \( Y \) of length \( k+1 \), contradicting the assumption on \( Z \). Thus we must have \( z_k = x_m = y_n \). The prefix \( Z_{k-1} \) is a length \( k-1 \) common subsequence of \( X_{m-1} \) and \( Y_{n-1} \). We show that it is an LCS. Assume not. Then there is a common subsequence \( W \) of \( X_{m-1} \) and \( Y_{n-1} \), of length > \( k-1 \). Appending \( x_m = y_n \) to \( W \) produces a common subsequence of \( X \) and \( Y \) longer than \( Z \) - contradiction.

2. If \( z_k \neq x_m \), then \( Z \) is a common subsequence of \( X_{m-1} \) and \( Y \).
   If \( Z \) were not an LCS of \( X_{m-1} \) and \( Y \), then there would be a longer common subsequence \( W \), that would also be a common subsequence of \( X \) and \( Y \), violating the assumption on \( Z \).


NOTE: this shows that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. The LCS problem has an optimal substructure property. We will also see an "overlapping subproblems property".

To find an LCS for \( X = \langle x_1, \ldots, x_m \rangle \) and \( Y = \langle y_1, \ldots, y_n \rangle \), we have two possibilities:
1. If \( x_m = y_n \), find an LCS for \( X_{m-1} \) and \( Y_{n-1} \) and append \( x_m = y_n \).
2. If \( x_m \neq y_n \), find an LCS for \( X_{m-1} \) and \( Y \) and one for \( X \) and \( Y_{n-1} \).
   Whichever of the two is longer is an LCS for the original problem.

The "overlapping subproblem property" arises because finding LCSs for \( X_{m-1} \) and \( Y \) and for \( X \) and \( Y_{n-1} \), both may involve finding an LCS for \( X_{m-1} \) and \( Y_{n-1} \) (and all other shorter ones).

Define \( c[i,j] \) to be the length of an LCS for the sequences \( X_i \) and \( Y_j \). We can compute it as:
\[
c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
\max(c[i-1,j-1]+1, \max(c[i-1,j],c[i,j-1])) & \text{if } i,j > 0 \text{ and } x_i = y_j.
\end{cases}
\]

This would lead to an exponential time algorithm… Not very good. How can we make it low degree polynomial (or better -- well, maybe not...)?
Longest Common Subsequence

Ex.: \( X = \text{bozo}; Y = \text{bat} \). Think of these as sequences of characters rather than strings. The full tree would look like (partial):

Once the LCS has been identified, and saved (via the three kinds of arrows), we can print it out:

- Initial call is \( \text{PRINT-LCS}(0, 0) \).
- \( b[i, j] \) points to table entry whose subproblem we used in solving LCS of \( X_i \) and \( Y_j \).
- When \( b[i, j] = \_ \), we have extended LCS by one character. So longest common subsequence \( z \) entries with \( \_ \) in them.
Dynamic Programming

Longest Common Subsequence

LCS-LENGTH(x, y, m, n, i, j) for i ← 1 to m do; for j ← 1 to n do;
if x[i] = y[j] then L[i][j] = L[i-1][j-1] + 1
else if |i - j| ≤ |i - j| then L[i][j] = L[i-1][j-1] + 1
else L[i][j] = \max_i L[i-1][j], L[i][j-1]
return L[m][n]