Dynamic Programming

Solving a problem by decomposing it into smaller subproblems and solving them from smallest to largest.

Introduced by Richard Bellman (of Bellman-Ford)

“Programming”: scheduling/planning

“Dynamic”: because it sounds cool
Fibonacci

0, 1, 1, 2, 3, 5, 8, ...

\( F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \)

\[ \text{Fib}(n) = \begin{cases} 
0 & \text{if } n=0 \\
1 & \text{if } n=1 \\
\text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise}
\end{cases} \]

Lecture 1: \( \text{Fib}(n) \) takes \( \geq 1.5^n \) calls to \( \text{Fib}(\cdot) \)

Fix: Store values of \( \text{Fib}(n) \) once computed so we don't have to recompute.
Fibonacci with memoization

FibMemo(n)
if n = 0 return 0
if n = 1 return 1
if n in Memo
    return Memo[n]
Memo[n] = FibMemo(n-1) + FibMemo(n-2)
return Memo[n]

# of recursive calls = O(n)

Subproblems: computing F(0), ..., F(n)

“top-down dynamic programming”
The underlying DAG

View each subproblem as a node.
Add a directed edge from \( i \rightarrow j \) if subproblem \( j \) directly depends on solution to subproblem \( i \).

This is a DAG

Should do topological sort, then solve subproblems in that order.
FibBottomUp(n)

```plaintext
Mem[0] = 0
Mem[1] = 1
for i from 2 to n
    Mem[i] = Mem[i-1] + Mem[i-2]
Return Mem[n]
```

this is "bottom-up dynamic programming"
Outline for designing DP algorithms

1. Identify subproblems \((F(0), \ldots, F(n))\)

2. Compute recurrence \((F(n) = F(n-1) + F(n-2))\)

3. Design algorithm \((\text{bottom-up Fib})\)
Shortest paths in DAGs

Input: DAG $G = (V, E)$, edge lengths $l(u,v)$ (positive or negative) $s \in V$

Output: Compute $\text{dist}(t) =$ length of shortest $s \rightarrow t$ path

1. Identify subproblems: $\forall v \in V$, compute $\text{dist}[v]$
2. Compute recurrence: $\text{dist}(D) = \min_{c} \text{dist}(c) + 3$, $\text{dist}(B) + 1$
Recurrence: \( \text{dist}[v] = \min_{u:(u,v) \in E} \text{dist}[u] + l(u,v) \) 

3. Design algorithm

initialize all \( \text{dist}(u) \) values to 0
\( \text{dist}(s) = 0 \)
for each \( v \in V \setminus \{s\} \) in linearized order
\( \text{dist}[v] = \min \{ \text{dist}[u] + l(u,v) \} \) \((*)\)
return \( \text{dist}[t] \)

Runtime: \( O(n+m) \)

Underlying DAG: \( G \) itself

What if we had wanted to compute shortest path?
introduce \( \text{prev}[v] = \text{the } u \text{ that minimizes } (*) \)
Longest increasing subsequences

Input: Sequence of n integers \(a_1, \ldots, a_n\)

Output: Length of longest increasing subsequences (non-contiguous)

\(a = 5 \ 2 \ 8 \ 6 \ 3 \ 6 \ 9 \ 7\)

DP design steps

1. Identify subproblems: for each prefix \(a_1, \ldots, a_j\)
   Compute: \(L[j] = \text{length of LIS in } a_1, \ldots, a_j \text{ ends in } a_j\)

2. Compute recurrence: \(L[j] = \max \{L[i] : a_i < a_j \} + 1\)
   \(L[j] = 1\) if no \(a_i < a_j\)
3. Design algorithm
   for \( j = 1, \ldots, n \)
   if \( \exists \ i < j \) s.t. \( a_i < a_j \)
   \[ L[j] = 1 + \max_{\ell < j} L[\ell] \]
   else
   \[ L[j] = 1 \]
return \( \max_{\ell} L[\ell] \) over all \( j \)

Runtime: \( O(n) \) subproblems \( \times \) \( O(n) \) time/subproblem
\( = O(n^2) \) time
Edit distance

Input: two strings \( S[1...m], T[1...n] \)

Goal: Find smallest number of edits to turn \( S \) into \( T \)

Edits allowed:
1. Insert character into \( S \)
2. Delete character from \( S \)
3. Substitute one character for another

Example:
\( S = "\text{Snowy}" \)
\( T = "\text{Sunny}" \)

\( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)  \( \text{Snowy} \)

- Insert \( u \) after \( S \)
- Replace \( o \) with \( n \) in \( S \)
- Delete \( w \) from \( S \)
Alignment

$S = \text{“Snowy”}$

$T = \text{“Sunny”}$

Alignment

\[
\begin{array}{cccc}
S & - & n & o
\end{array}
\quad
\begin{array}{cccc}
\text{nowy} & \quad & \text{Sunny}
\end{array}
\]

**Cost:** # of columns where symbols differ

$\text{Edit distance} = \text{minimum cost of an alignment}$
DP design steps

1. Identify subproblems: for all $0 \leq i \leq m$, $0 \leq j \leq n$

   \[ E(i, j) = \text{Edit Distance} \left( S[1, \ldots, i], T[1, \ldots, j] \right) = \text{cost of best alignment between } S[1, \ldots, i], T[1, \ldots, j] \]

2. Compute recurrence

   \[ S[1, \ldots, i] T[1, \ldots, j] \]

   look at last column in optimal alignment

   \[
   E(i, j) = \min \begin{cases} 
   E(i, j) & \text{match} \
   1 + E(i-1, j) & \text{insert} \
   1 + E(i, j-1) & \text{delete} \
   E(i-1, j-1) + \text{diff}(c_{ij}) & \text{substitution} 
   \end{cases}
   \]

   \[ E(i, 0) = i, \quad E(0, j) = j \]

   \[ \text{diff}(c_{ij}) = \begin{cases} 
   0 & \text{if } S(i) = T(j) \
   1 & \text{otherwise} 
   \end{cases} \]
\[ a \quad b \quad a \quad b \quad a - b \]
3 Design algorithms

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$E^{(-1)}_{j-j}$</td>
<td>$E^{(-1)}_{i-j}$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>$E^{(-1)}_{j-n}$</td>
<td>$E^{(-1)}_{i-j}$</td>
<td>$E^{(-1)}_{i-j}$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>$E^{(-1)}_{j-n}$</td>
<td>$E^{(-1)}_{i-j}$</td>
<td>$E^{(-1)}_{i-j}$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for $i = 0, 1, 2, \ldots, m$
\[ E(i, 0) = i \]

for $j = 0, 1, 2, \ldots, n$
\[ E(0, j) = j \]

for $i = -1, 2, \ldots, m$

for $j = 1, 2, \ldots, n$
\[ E(i, j) = \min \{ E(i-1, j) + 1, E[i, j-1] + 1 \} \]

return $E(m, n)$

Runtime: $O(mn)$

\[
\begin{align*}
S &= 9 \\
T &= 9 \\
T' &= 6 \\
E(0, 0) &= 0 \\
E(0, 1) &= E(0, 0) + \text{diff}(1, 1)
\end{align*}
\]