Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems.

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS.

“Programming” here means “planning”

Main idea:
- Set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- Solve smaller instances once
- Record solutions in a table
- Extract solution to the initial instance from that table
Example: Fibonacci numbers

• Recall definition of Fibonacci numbers:

\[
\begin{align*}
    f(0) &= 0 \\
    f(1) &= 1 \\
    f(n) &= f(n-1) + f(n-2)
\end{align*}
\]

• Computing the \( n \)th Fibonacci number recursively (top-down):

\[
\begin{align*}
    f(n) &= f(n-1) + f(n-2) \\
    &= f(n-2) + f(n-3) + f(n-3) + f(n-4) \\
    &\vdots
\end{align*}
\]

Example: Fibonacci numbers

Computing the \( n \)th Fibonacci number using bottom-up iteration and recording results:

\[
\begin{align*}
    F(0) &= 0 \\
    F(1) &= 1 \\
    F(2) &= 1 + 0 = 1 \\
    \vdots \\
    F(n-2) &= \\
    F(n-1) &= \\
    F(n) &= F(n-1) + F(n-2)
\end{align*}
\]

| 0 | 1 | 1 | \ldots | \( F(n-2) \) | \( F(n-1) \) | \( F(n) \) |

Efficiency:
- time
- space
Examples of Dynamic Programming Algorithms

- Computing binomial coefficients
- Optimal chain matrix multiplication
- Constructing an optimal binary search tree
- Warshall’s algorithm for transitive closure
- Floyd’s algorithms for all-pairs shortest paths
- Some instances of difficult discrete optimization problems:
  - Travelling salesman
  - Knapsack
- Coin change problem
- Manhattan tourist problem
- DNA/Protein sequence alignment

Computing a binomial coefficient by DP

- Binomial coefficients are coefficients of the binomial formula:
  \[(a + b)^n = C(n, 0)a^n b^0 + \ldots + C(n, k)a^{n-k}b^k + \ldots + C(n, n)a^0b^n\]

- Recurrence: \(C(n, k) = C(n-1, k) + C(n-1, k-1)\) for \(n > k > 0\)
  \(C(n, 0) = 1, C(n, n) = 1\), for \(n \geq 0\)

- Value of \(C(n,k)\) can be computed by filling a table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>k-1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>(C(n-1, k-1))</td>
<td>(C(n-1, k))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>(C(n, k))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing $C(n,k)$: pseudocode and analysis

**Algorithm**  
*Binomial*(n, k)  

//Computes $C(n, k)$ by the dynamic programming algorithm  
//Input: A pair of nonnegative integers $n \geq k \geq 0$  
//Output: The value of $C(n, k)$

for $i \leftarrow 0$ to $n$ do

  for $j \leftarrow 0$ to $\min(i, k)$ do

    if $j = 0$ or $j = i$

      $C[i, j] \leftarrow 1$

    else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

return $C[n, k]$

**Time efficiency:** $\Theta(nk)$  

**Space efficiency:** $\Theta(nk)$

Copyright © 2007 Pearson Addison-Wesley. All rights reserved