Dynamic Programming

- Outline:
  - Example 1 – Licking Stamps
  - General Principles
  - Example 2 – Knapsack (§ 5.10)
  - Example 3 – Sequence Comparison (§ 6.8)

Licking Stamps

- Given:
  - Large supply of 5¢, 4¢, and 1¢ stamps
  - An amount $N$
  - Problem: choose fewest stamps totaling $N$

How to Lick 27¢

<table>
<thead>
<tr>
<th># of 5¢ Stamps</th>
<th># of 4¢ Stamps</th>
<th># of 1¢ Stamps</th>
<th>Total Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Moral: Greed doesn’t pay

A Simple Algorithm

- At most $N$ stamps needed, etc.
  - for $a = 0, \ldots, N$
    - for $b = 0, \ldots, N$
      - for $c = 0, \ldots, N$
        - if $(5a + 4b + c = N \land a + b + c$ is new min)
          - retain $(a, b, c)$
  - output retained triple;
- Time: $O(N^3)$
  (Not too hard to see some optimizations, but we’re after bigger fish…)
Better Idea

**Theorem:** If last stamp licked in an optimal solution has value v, then previous stamps form an optimal solution for N-v.

**Proof:** if not, we could improve the solution for N by using opt for N-v.

\[
M(i) = \min \begin{cases} 
0 & i = 0 \\
1 + M(i-5) & i \geq 5 \\
1 + M(i-4) & i \geq 4 \\
1 + M(i-1) & i \geq 1 
\end{cases}
\]

where \( M(i) \) = min number of stamps totaling \( i \)

New Idea: Recursion

\[
M(i) = \min \begin{cases} 
0 & i = 0 \\
1 + M(i-5) & i \geq 5 \\
1 + M(i-4) & i \geq 4 \\
1 + M(i-1) & i \geq 1 
\end{cases}
\]

Another New Idea: Avoid Recomputation

- Tabulate values of solved subproblems
  - Top-down: "memoization"
  - Bottom up:
    
    \[
    \text{for } i = 0, \ldots, N \text{ do } M(i) = \min \begin{cases} 
0 & i = 0 \\
1 + M(i-5) & i \geq 5 \\
1 + M(i-4) & i \geq 4 \\
1 + M(i-1) & i \geq 1 
\end{cases}
    \]

- Time: \( O(N) \)

Finding How Many Stamps

\[
1 + \text{Min}(3, 1, 3) = 2
\]

Finding Which Stamps: Trace-Back

\[
1 + \text{Min}(3, 1, 3) = 2
\]

Complexity Note

- \( O(N) \) is better than \( O \left( N^3 \right) \) or \( O \left( 3^{N/5} \right) \)

- But still \textit{exponential} in input size (\log N bits)

  (E.g., miserably slow if \( N \) is 64 bits – \( 2^{64} \) steps for 64 bit input.)

- Note: can do in \( O(1) \) for 5¢, 4¢, and 1¢ but not in general. See “NP-Completeness” later
Elements of Dynamic Programming

- What feature did we use?
- What should we look for to use again?
- “Optimal Substructure”
  - Optimal solution contains optimal subproblems
- “Repeated Subproblems”
  - The same subproblems arise in various ways

The Knapsack Problem (§ 5.10)

Given positive integers $W, w_1, w_2, \ldots, w_n$.
Find a subset of the $w_i$’s totaling exactly $W$.
Alternate (Easier?) Problem: Is there one?

(Like stamp problem, but limited supply of each.)

Motivation: simple 1-d abstraction of packing boxes, trucks, VLSI chips, …

Knapsack Example

$w_1, \ldots, w_4 = 2, 5, 9, 11$

- $W = 14$:
  - YES: $5+9 = 14$
- $W = 15$:
  - NO: all singletons up to 11 too small,
    - all pairs too small, except $9+11$, $5+11$ too big
    - all triples $\leq 16$: too big
    - all quadruples too big

Solve by Induction? Try 1

- Defn: Let $P(i, X)$ be true iff there is a subset of first $i$ weights $w_1, w_2, \ldots, w_i$ totaling $X$
- Assume we know how to evaluate $P(n-1, X)$ for all $X$
  - Case 1: $P(n-1, W) = True$ – done; $w_n$ unneeded
  - Case 2: $P(n-1, W) = False$ – may or may not be a solution, but if there is one, it includes $w_n$, and other included weights total $W-w_n$, so $P(n, W) = P(n-1, W-w_n)$

Algorithm:
  - $P(n, W) = P(n-1, W) \lor P(n-1, W-w_n)$
  - Basis: $P(0, X) = True$ iff ($X = 0$)

Solve by Induction? Try 2

- Defn: Let $P(i, X)$ be true iff there is a subset of first $i$ weights $w_1, w_2, \ldots, w_i$ totaling $X$
- Assume we know $P(n-1, X)$ for all $X \leq W$
  - Case 1: $P(n-1, W) = True$ – done; $w_n$ unneeded
  - Case 2: $P(n-1, W) = False$ – may or may not be a solution, but if there is one, it includes $w_n$, and other included weights total $W-w_n$, so $P(n, W) = P(n-1, W-w_n)$

Algorithm:
  - $P(n, W) = P(n-1, W) \lor P(n-1, W-w_n)$
  - Basis: $P(0, X) = True$ iff ($X = 0$)

Knapsack Example

$w_1, \ldots, w_4 = 2, 5, 9, 11$, $W = 15$

$W = 14$: YES
$W = 15$: NO
Dynamic Programming?

\[ P(n,W) = P(n-1, W) \lor P(n-1, W-w_n) \]

- Optimal substructure?
  - Best/only way to fill a big knapsack implicitly fills smaller ones with fewer objects in the best or only way
- Repeated subproblems?
  - Smallest cases potentially common to many bigger instances

Complexity Notes

- Time is \( O(NW) \)
- May or may not beat naïve \( 2^N \)
- But still partially exponential in input size (\( N \log W \) bits)
  - E.g. 100 weights, 64 bits each – \( 100 \times 2^{64} \) array elements.
  - C.v., e.g., Skyline 100 bids, 64 bit coords – \( c \times 100 \times \log 100 \) steps.
- See “NP-Completeness” later