Licking Stamps

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

<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, ..., N {
    - for b = 0, ..., N {
      - for c = 0, ..., N {
        - if (5a + 4b + c == N && a + b + c is new min) {
          - retain (a, b, c);}
        }
      }
    }
  }
- Time: \(O(N^3)\)

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 \(N-v\).
  - \(M(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 + M(i-5) & \text{if } i < 5 \\ 1 + M(i-4) & \text{if } i < 4 \\ 1 + M(i-1) & \text{if } i \geq 1 \end{cases}\)

where \(M(N) = \text{min number of stamps totaling } N\)
New Idea: Recursion

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

Time: \( > 3^{N/5} \)

Another New Idea: Avoid Recomputation

- Tabulate values of solved subproblems
  - Top-down: “memoization”
  - Bottom up:
    \[
    \text{for } i = 0, \ldots, N \text{ do } M(i) = \min \begin{cases} 
0 & \text{if } i = 0 \\
1 + M(i-5) & \text{if } i \geq 5 \\
1 + M(i-4) & \text{if } i < 5 \\
1 + M(i-1) & \text{if } i < 1 
\end{cases} 
\]
- 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 \( O(N^3) \) or \( O(3^{N/5}) \)
- But still exponential in input size (log \( N \) bits)
  - (E.g., miserably slow if \( N \) is 64 bits.)
- 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
Matrix-chain Products

Given: \( p_1 \times p_2 \) matrices \( A_i, 1 \leq i \leq n \)

Problem: Compute \( A_1 \cdot A_2 \cdot \ldots \cdot A_n \)

In general:

\[ p \times q \) times \( q \times r \)

\[ \text{costs } p \times q \times r \]

\[ \text{Work } = 3 \times 20 \times 4 \]

Simple Algorithm

Just try all possible parenthesizations

How many are there?

\[ P(1) = 1 \]

\[ P(n) = \sum_{k=1}^{n-1} P(k)P(n-k), n > 1 \]

\[ P(n) = \frac{1}{n} \left(2n - 2\right) \cdot \Omega\left(\frac{4^n}{n^{3/2}}\right) \]

Optimal Substructure:

Theorem: if the last multiply is \((A_1 \cdots A_i)(A_{i+1} \cdots A_n)\), then \(A_1 \cdots A_i\) is
optimally parenthesized, as is \(A_{i+1} \cdots A_n\).

Proof: Could improve if not.

Let \( M[i,j] = \min \text{ ops to multiply } A_i \cdots A_j \)

\[ M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k<i,j} (M[i,k] + M[k+1,j] + p_i \cdot p_k \cdot p_j) & \text{otherwise} \end{cases} \]

Repeated Subproblems

All 5 Parenthesizations of \( A_1 \cdot A_2 \cdot A_3 \cdot A_4 \):

An \( O(n^3) \) Algorithm

// Goal: \( M[i,j] = \min \text{ ops to multiply } A_i \cdots A_j \)

for \( j := 1 \) to \( n \) do

\( M[j,j] := 0; \)

for \( i := (j - 1) \) downto 1 do

\( M[i,j] := \min_{k<i,j} (M[i,k] + M[k+1,j] + p_i \cdot p_k \cdot p_j); \)
Example:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & & & \\
2 & & & \\
3 & & & \\
4 & & & \\
\end{array}
\]

\[
\begin{align*}
p_0 &= 2 \quad A_0: 2x3 \\
p_1 &= 3 \quad A_1: 3x1 \\
p_2 &= 1 \quad A_2: 1x5 \\
p_3 &= 5 \quad A_3: 5x1 \\
p_4 &= 1 \\
\end{align*}
\]