**Algorithm Design Techniques**
- Induction
- Divide and Conquer
- Dynamic Programming
- Greedy
- Reduction

**Outline**
- **Longest Common Subsequence**
  - Inductive Approach
  - Dynamic Programming
  - Backtracking
- Matrix Chain Multiplication

**Longest Common Subsequence**
- **Input:** Two sequences (lists) of integers
  \[ X = x_1, x_2, \ldots, x_j \text{ and } Y = y_1, y_2, \ldots, y_m \]
- **Output:** A longest subsequence of \( X \) that is also a subsequence of \( Y \)

**LCS - Example**
- **Input:** \( X = 1, -2, 3, 4, 9, 18 \)
  \( Y = 3, 9, 1, -2, 5, -2, 22, 18 \)
- **Output:** \( Z = 1, -2, 18 \)

**LCS\(_{n+1}\) Algorithm**
Finds LCS of sequences with \( n+1 \) or fewer elements (total)

**Easy cases:**
- \( X \) or \( Y \) is empty
  - \( \text{LCS}(\Phi, Y[1\ldots m]) = \) 
  - \( \text{LCS}(X[1\ldots j], \Phi) = \)
Harder case
(Assume j>0, m>0)

\[ X = x_1, x_2, \ldots, x_{j-1}, x_j \quad \text{and} \quad Y = y_1, y_2, \ldots, y_{m-1}, y_m \]

1. \( x_j = y_m \)

2. \( x_j \neq y_m \)

Run Time

\[ T(j, m) = \max(T(j-1, m) + T(j, m-1)), T(m-1, n-1)) + c \]

\[ \geq 2T(j-1, m-1) + c \]

\[ = \Omega(2^{\min(j, m)}) \]

Run Time Analysis
Many duplicated subtrees

Dynamic Programming
Don’t Recalculate

Dynamic Programming

• \( A(i, k) \) is the length of a longest common subsequence of \( X[1 \ldots i] \) and \( Y[1 \ldots k] \)

• \( A(i, 0) = A(0, k) = 0 \) for \( 0 \leq i \leq j \) and \( 0 \leq k \leq m \)

• \( A(i, k) \) = maximum of
  - \( A(i-1, k) \)
  - \( A(i, k-1) \)
  - \( A(i-1, k-1) + \text{match}(x_i, y_k) \)

• \( A(j, m) \) is the length of a longest common subsequence of \( X \) and \( Y \)
A(i,j)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>-2</th>
<th>3</th>
<th>4</th>
<th>9</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

LCS - Algorithm

LCS(X=x_1,x_2,...,x_j;Y=y_1,y_2,...,y_m)
For i=0 to j: A(i,0)=0
For i=0 to m: A(0,i)=0
For i=1 to j
  For k=1 to m
    If x_i=y_k then match=1 else match=0
    A(i,k) =max(A(i -1,k),A(i,k -1),A(i-1,k-1)+match))
Return A(j,m)

Run Time Analysis
- Number of table entries:
- Time to compute one entry:
- Run time:

Backtracking

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>-2</th>
<th>3</th>
<th>4</th>
<th>9</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Outline
- Longest Common Subsequence
  - Inductive Approach
  - Dynamic Programming
  - Backtracking
- Matrix Chain Multiplication

Matrix Chain Multiplication
- A is an n x m matrix
- B is an m x k matrix
- How many scalar multiplications are needed to compute AB?
Matrix Chain Multiplication

- A is a $2 \times 5$ matrix
- B is a $5 \times 1000$ matrix
- C is a $1000 \times 2$ matrix.
- How many scalar multiplications are needed to compute $ABC$?
  - $(AB)C$
  - $A(BC)$

Matrix Chain Multiplication

- Input: A list of $n+1$ integers $p_1, p_2, \ldots, p_{n+1}$
- Output: The minimum number of scalar multiplications needed to compute $\prod_{i=1}^{n} A_i$ where $A_i$ is a $p_i \times p_{i+1}$ matrix.

(Assume a standard matrix multiplication procedure is used; i.e. no Strassen-like improvements.)

**MCM$_{n+1}$ Algorithm**

Inductive Approach

- Consider an input: $p_1, p_2, p_3, p_4, p_5, p_6$
- Imagine an optimal way of multiplying matrices $A_1, A_2, A_3, A_4, A_5$:
  $$(A_1(A_2A_3))(A_4A_5)$$

Inductive Approach

- There is some last multiplication
  $$(A_1(A_2A_3))(A_4A_5)$$
Inductive Approach cont.

• There is some last multiplication
  \((A_1(A_2 A_3)) \| (A_4 A_5)\)

• So \(\text{OPT}(A_1, A_2, A_3, A_4, A_5) = \text{OPT}(A_1, A_2, A_3) + \text{OPT}(A_4, A_5) + p_1 p_4 p_5\)

Inductive Approach

• We don’t know where the top split occurs … but clearly \(\text{OPT}(A_1, A_2, A_3, A_4, A_5) = \min_{0<k<5} \text{OPT}(A_1, \ldots, A_k) + \text{OPT}(A_{k+1}, \ldots, A_5) + p_1 p_{k+1} p_5\)
  
• where \(\text{OPT}(A) = 0\)

Running Time

• \(T(n) = \sum_{0<k<n} T(k) + T(n-k) + c\)
  \[\geq 2T(n-1) + c\]
  \[= \Omega(2^n)\]

Dynamic Programming

• Use a table to store results
• What kind of results?
  – \(M(k,j) = \text{Minimum number of multiplications to compute } \prod_{i=k}^{j} A_i\)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(M(3,5)\) needs \(M(3,3), M(4,5), M(3,4), M(5,5)\)

\(M(k,j)\) needs \(M(i,m)\)
  where \(m-i < j-k\)
**Dynamic Programming Algorithm**

\[ M(k,k) = 0 \]

For \( j, k \) such that \( j-k = 1, 2, \ldots, n-1 \)

\[ M(k,j) = \min_{i=k}^{j-1} M(k,i) + M(i+1,j) + p_k p_{i+1} p_j \]

Return \( M(1,n) \)

---

**Input:** 2, 3, 1, 5, 4, 8

(A_1 is 2x3, A_2 is 3x1, …)

\[
\begin{array}{|c|c|c|c|}
\hline
0 & 6 & & \\
\hline
0 & 15 & & \\
\hline
0 & 20 & & \\
\hline
0 & 160 & & 0 \\
\hline
\end{array}
\]

---

**MCM Algorithm**

- Recursive Algorithm takes exponential time.
- Dynamic Programming takes ________.