Dynamic Programming: Finding Recursive Structures

Hao WU

Department of Computer Science and Engineering
Chinese University of Hong Kong
A string $s$ is a **subsequence** of another string $t$ if either $s = t$ or we can convert $t$ to $s$ by deleting characters.

$t = ABCDE$

- $s = ACE$ is not a subsequence because $D$ is missing.
- $s = ACD$ is not a subsequence because $E$ is missing.
The Longest Common Subsequence Problem

Given two strings $x$ and $y$, find a common subsequence $z$ of $x$ and $y$ with the maximum length.

- $z$ is a **longest common subsequence** (LCS) of $x$ and $y$.

$$
\begin{array}{cccccc}
x &=& A & B & D & A & B \\
y &=& B & C & D & A & B \\
z &=& B & D & A & B \\
\end{array}
$$

**Remark:** If $x = \emptyset$ (empty string) or $y = \emptyset$, their (only) LCS is $\emptyset$. 
The key to solving the problem is to identify its underlying recursive structure.

Specifically, how the original problem is related to subproblems.
The length of $x$ is $n$; the length of $y$ is $m$.

**Theorem (LCS Theorem):** Let $z$ be any LCS of $x$ and $y$, and $k$ be the length of $z$. Then:

1. If $x[n] = y[m]$ then $z[k] = x[n]$ (hence, also $= y[m]$) and $z[1 : k - 1]$ is an LCS of $x[1 : n - 1]$ and $y[1 : m - 1]$.
2. If $x[n] \neq y[m]$, then at least one of the following holds:
   - $z$ is an LCS of $x[1 : n - 1]$ and $y$
   - $z$ is an LCS of $x$ and $y[1 : m - 1]$.

Next, we will prove the theorem.
Lemma 1: If $z[k] \neq x[n]$, then $z$ is a subsequence of $x[1 : n - 1]$.

**Proof:** As $z$ is a subsequence of $x$, we can convert $x$ to $z$ by deleting characters repeatedly. The conversion must have deleted $x[n]$; otherwise, $x[n]$ must be the last character of $z$, which contradicts $z[k] \neq x[n]$.

It thus follows that we can obtain $z$ by repeatedly deleting characters from $x[1 : n - 1]$ and, hence, $z$ is a subsequence of $x[1 : n - 1]$. □
Proof of Statement 1 (in the LCS Theorem):

**Claim:** If \( x[n] = y[m] \), then \( z[k] = x[n] \).

Assume that \( x[n] = y[m] \) but \( z[k] \neq x[n] \). By Lemma 1, \( z \) is a common subsequence of \( x[1 : n - 1] \) and \( y[1 : m - 1] \). Now, we can obtain a common subsequence \( z' = z \circ x[n] \) of \( x \) and \( y \). However, \( z' \) will be a length-\((k + 1)\) common subsequence of \( x \) and \( y \), contradicting the fact that \( z \) is an LCS of \( x \) and \( y \).

**Remark:** \( \circ \) means string concatenation. For example, \( ABC \circ DEF = ABCDEF \).
Proof of Statement 1:

Claim: If $x[n] = y[m]$, then $z[1 : k - 1]$ is an LCS of $x[1 : n - 1]$ and $y[1 : m - 1]$.

Assume that $z[1 : k - 1]$ is not an LCS of $x[1 : n - 1]$ and $y[1 : m - 1]$. Thus, $x[1 : n - 1]$ and $y[1 : m - 1]$ have an LCS $z'$ with length at least $k$.

However, $z' \circ x[n]$ will be a length-$(k + 1)$ common subsequence of $x$ and $y$, contradicting the fact that $z$ is an LCS of $x$ and $y$.  ■
Proof of Statement 2:
Because $x[n] \neq y[m]$, at least one of the following is false:

- $z[k] = x[n]$
- $z[k] = y[m]$.

Consider first $z[k] \neq x[n]$.

We argue that $z$ must be an LCS of $x[1 : n - 1]$ and $y$.

- By Lemma 1, $z$ is a subsequence of $x[1 : n - 1]$. Since $z$ is also a subsequence of $y$, $z$ is a common subsequence of $x[1 : n - 1]$ and $y$.
- Suppose that $z$ is not an LCS of $x[1 : n - 1]$ and $y$. Thus, $x[1 : n - 1]$ and $y$ have an LCS $z'$ of length at least $k + 1$. This means that $x$ and $y$ have a common subsequence of length $k + 1$, contradicting the fact that $z$ is an LCS of $x$ and $y$.

A symmetric argument proves the statement when $z[k] \neq y[m]$. □
Matrix-Chain Multiplication

You are given an algorithm \( A \) that, given an \( a \times b \) matrix \( A \) and a \( b \times c \) matrix \( B \), can calculate \( AB \) in \( O(abc) \) time. You need to use \( A \) to calculate the product of \( A_1A_2\ldots A_n \) where \( A_i \) is an \( a_i \times b_i \) matrix for \( i \in [1, n] \). This implies that \( b_{i-1} = a_i \) for \( i \in [2, n] \), and the final result is an \( a_1 \times b_n \) matrix.

A trivial strategy is to apply \( A \) to evaluate the product from left to right. However, we may be able to reduce the cost by following a different multiplication order.
Example

Consider $A_1 A_2 A_3$ where $A_1$ and $A_2$ are $m \times m$ matrices, but $A_3$ is $m \times 1$.

There are two multiplication orders:

- $(A_1 A_2) A_3$.
  The cost of computing $B = A_1 A_2$ is $O(m \cdot m \cdot m) = O(m^3)$ and $B$ is an $m \times m$ matrix. The cost of $BA_3$ is $O(m \cdot m \cdot 1) = O(m^2)$. The total cost is $O(m^3)$.

- $A_1 (A_2 A_3)$.
  The cost of computing $B = A_2 A_3$ is $O(m \cdot m \cdot 1) = O(m^2)$ and $B$ is an $m \times 1$ matrix. The cost of $A_1 B$ is $O(m \cdot m \cdot 1) = O(m^2)$. The total cost is $O(m^2)$.
Parenthesizing $A_1A_2...A_n$ at $A_k$ for some $k \in [1, n-1]$ converts the expression to $(A_1...A_k)(A_{k+1}...A_n)$, after which you can parenthesize each of $A_1...A_i$ and $A_{i+1}...A_n$ recursively.

A **fully parenthesized product** is

- either a single matrix or
- the product of two fully parenthesized products.

For example, if $n = 4$, then $(A_1A_2)(A_3A_4)$ and $((A_1A_2)A_3)A_4$ are fully parenthesized, but $A_1(A_2A_3A_4)$ is not.

A fully parenthesized product determines a multiplication order that, in turn, determines the computation cost.

**Goal:** Design an algorithm to find in $O(n^3)$ time a fully parenthesized product with the smallest cost.
Recursive Structure

By parenthesizing at \( A_k \), we obtain

\[
\underbrace{(A_1 \ldots A_k)}_{B_1} \underbrace{(A_{k+1} \ldots A_n)}_{B_2},
\]

where \( B_1 \) is an \( a_1 \times b_k \) matrix and \( B_2 \) is an \( a_{k+1} \times b_n \) matrix.

The total cost is

\[
\text{cost of computing } B_1 + \text{cost of computing } B_2 + O(a_1 b_k b_n).
\]
We define \( \text{cost}(i, j) \), where \( 1 \leq i \leq j \leq n \), to be the smallest achievable cost for calculating \( A_i \ldots A_j \). Our objective is to calculate \( \text{cost}(1, n) \).

If we parenthesize \( A_i \ldots A_j \) at \( A_k \), we obtain

\[
\left( A_i \ldots A_k \right) \left( A_{k+1} \ldots A_j \right).
\]

The total cost is

\[
\text{cost}(i, k) + \text{cost}(k + 1, j) + O(a_ib_kb_j).
\]
To attain $\text{cost}(i, j)$, we should try all possible parenthesizations of $A_i \ldots A_j$. This implies:

\[
\text{cost}(i, j) = \begin{cases} 
  O(1) & \text{if } i = j \\
  \min_{k=i}^{j-1} (\text{cost}(i, k) + \text{cost}(k + 1, j) + O(a_i b_k b_j)) & \text{if } i < j
\end{cases}
\]

By dyn. programming, we can compute $\text{cost}(1, n)$ in $O(n^3)$ time.
Consider $A_1 A_2 A_3 A_4$ where $A_1$ and $A_2$ are $m \times m$ matrices, $A_3$ is $m \times 1$, and $A_4$ is $1 \times m$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$cost(1, 4)$
After solving all subproblems, we obtain:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(1)$</td>
<td>$O(m^3)$</td>
<td>$O(m^2)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>$O(1)$</td>
<td>$O(m^2)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>$O(1)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Next, we apply the “piggyback technique” to generate an optimal parenthesization.
Define \( bestSub(i, j) = \)

- nil, if \( i = j \);

- \( k \), if the best parenthesization for \( A_i A_{i+1} \ldots A_j \) is \((A_i \ldots A_k)(A_{k+1} \ldots A_j)\).

\[
\begin{array}{cccc}
  & 1 & 2 & 3 & 4 \\
 1 & O(1) & O(m^3) & O(m^2) & O(m^2) \\
 2 & 0 & O(1) & O(m^2) & O(m^2) \\
 3 & 0 & 0 & O(1) & O(m^2) \\
 4 & 0 & 0 & 0 & O(1) \\
\end{array}
\]

After \( cost(i, j) \) is ready for all \( i, j \), we can compute all \( bestSub(i, j) \) in \( O(n^3) \) time.
### Example:

**bestSub(1, 4) = 3**, i.e., the best way to calculate \(A_1 A_2 A_3 A_4\) is \((A_1 A_2 A_3) A_4\).

Similarly, **bestSub(1, 3) = 1**, i.e., the best way to calculate \(A_1 A_2 A_3\) is \(A_1 (A_2 A_3)\).

Therefore, an optimal fully parenthesized product of \(A_1 A_2 A_3 A_4\) is \((A_1 (A_2 A_3)) A_4\).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(O(1))</td>
<td>(O(m^3))</td>
<td>(O(m^2))</td>
<td>(O(m^2))</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>(O(1))</td>
<td>(O(m^2))</td>
<td>(O(m^2))</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>(O(1))</td>
<td>(O(m^2))</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

\(A_1: m \times m\)
\(A_2: m \times m\)
\(A_3: m \times 1\)
\(A_4: 1 \times m\)