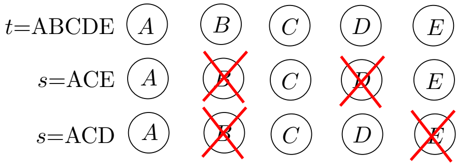
Dynamic Programming: Finding Recursive Structures

Hao WU

Department of Computer Science and Engineering
Chinese University of Hong Kong

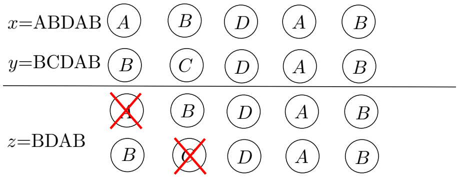
A string s is a **subsequence** of another string t if either $s = t$ or we can convert t to s by deleting characters.



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.



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.

n = the length of x; m = the length of y

Theorem (LCS Theorm): Let z be any LCS of x and y, and k 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 $x[1:n-1]$. $\square$

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. $\square$

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 $z[k] \neq y[m]$. $\square$

Matrix-Chain Multiplication

You are given an algorithm $\mathcal{A}$ that, given an $a \times b$ matrix $\boldsymbol{A}$ and a $b \times c$ matrix $\boldsymbol{B}$, can calculate $\boldsymbol{AB}$ in $O(abc)$ time. You need to use $\mathcal{A}$ to calculate the product of $\boldsymbol{A}_1\boldsymbol{A}_2...\boldsymbol{A}_n$ where $\boldsymbol{A}_i$ 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 $\mathcal{A}$ 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 $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3$ where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are $m \times m$ matrices, but $\boldsymbol{A}_3$ is $m \times 1$.

There are two multiplication orders:

- $(\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3$.
  The cost of computing $\boldsymbol{B} = \boldsymbol{A}_1\boldsymbol{A}_2$ is $O(m \cdot m \cdot m) = O(m^3)$ and $\boldsymbol{B}$ is an $m \times m$ matrix. The cost of $\boldsymbol{B}\boldsymbol{A}_3$ is $O(m \cdot m \cdot 1) = O(m^2)$. The total cost is $O(m^3)$.
- $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3)$.
  The cost of computing $\boldsymbol{B} = \boldsymbol{A}_2\boldsymbol{A}_3$ is $O(m \cdot m \cdot 1) = O(m^2)$ and $\boldsymbol{B}$ is an $m \times 1$ matrix. The cost of $\boldsymbol{A}_1\boldsymbol{B}$ is $O(m \cdot m \cdot 1) = O(m^2)$. The total cost is $O(m^2)$.

Parenthesizing $\boldsymbol{A}_1\boldsymbol{A}_2...\boldsymbol{A}_n$ at $\boldsymbol{A}_k$ for some $k \in [1, n-1]$ converts the expression to $(\boldsymbol{A}_1...\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}...\boldsymbol{A}_n)$, after which you can parenthesize each of $\boldsymbol{A}_1...\boldsymbol{A}_i$ and $\boldsymbol{A}_{i+1}...\boldsymbol{A}_n$ recursively.

A **fully parenthesized product** is

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

For example, if $n = 4$, then $(\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4)$ and $((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)\boldsymbol{A}_4$ are fully parenthesized, but $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4)$ 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 $\boldsymbol{A}_k$, we obtain

$$\underbrace{(\boldsymbol{A}_1 \ldots \boldsymbol{A}_k)}_{\boldsymbol{B}_1} \underbrace{(\boldsymbol{A}_{k+1} \ldots \boldsymbol{A}_n)}_{\boldsymbol{B}_2},$$

where $\boldsymbol{B}_1$ is an $a_1 \times b_k$ matrix and $\boldsymbol{B}_2$ is an $a_{k+1} \times b_n$ matrix.

The total cost is

$$\text{cost of computing } \boldsymbol{B}_1 + \text{cost of computing } \boldsymbol{B}_2 + O(a_1 b_k b_n).$$

We define $cost(i, j)$, where $1 \leq i \leq j \leq n$, to be the smallest achievable cost for calculating $\boldsymbol{A}_i ... \boldsymbol{A}_j$. Our objective is to calculate $cost(1, n)$.

If we parenthesize $\boldsymbol{A}_i ... \boldsymbol{A}_j$ at $\boldsymbol{A}_k$, we obtain

$$\underbrace{(\boldsymbol{A}_i ... \boldsymbol{A}_k)}_{cost(i,k)} \underbrace{(\boldsymbol{A}_{k+1} ... \boldsymbol{A}_j)}_{cost(k+1,j)}.$$

The total cost is

$$cost(i, k) + cost(k + 1, j) + O(a_i b_k b_j).$$

To attain $cost(i, j)$, we should try all possible parenthesizations of $\boldsymbol{A}_i ... \boldsymbol{A}_{\boldsymbol{j}}$. This implies:

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

By dyn. programming, we can compute $cost(1, n)$ in $O(n^3)$ time.

Consider $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are $m \times m$ matrices, $\boldsymbol{A}_3$ is $m \times 1$, and $\boldsymbol{A}_4$ is $1 \times m$.

i \ j	1	2	3	4
1				$cost(1,4)$
2	0			
3	0	0		
4	0	0	0	

After solving all subproblems, we obtain:

$i \backslash j$	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	$O(1)$

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 $\boldsymbol{A}_i \boldsymbol{A}_{i+1} \ldots \boldsymbol{A}_j$ is $(\boldsymbol{A}_i \ldots \boldsymbol{A}_k)(\boldsymbol{A}_{k+1} \ldots \boldsymbol{A}_j)$.

$i \backslash j$	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	$O(1)$

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

i \ j	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	$O(1)$

$\boldsymbol{A}_1$: $m \times m$
$\boldsymbol{A}_2$: $m \times m$
$\boldsymbol{A}_3$: $m \times 1$
$\boldsymbol{A}_4$: $1 \times m$

Example:
$bestSub(1,4) = 3$, i.e., the best way to calculate $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ is $(\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3)\boldsymbol{A}_4$.

Similarly, $bestSub(1,3) = 1$, i.e., the best way to calculate $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3$ is $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3)$.

Therefore, an optimal fully parenthesized product of $\boldsymbol{A}_1\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4$ is $(\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3))\boldsymbol{A}_4$.