CS 2009
Design and Analysis of Algorithms

Waheed Ahmed

Lecture 22,23, 24 Dynamic Programing (Longest common subsequence, Matrix-Chain Multiplication)

RECIPE FOR APPLYING DYNAMIC PROGRAMMING

Steps for applying DP to design algorithms

(Dynamic Programing)

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called “memoization”)

e.g. **Lots of different entries in the row $d^{(k)}$ may ask for $d^{(k-1)}[v]$**

Longest Common Subsequence

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

LONGEST COMMON SUBSEQUENCE

A sequence **Z** is a **SUBSEQUENCE** of **X** if **Z** can be obtained from **X** by deleting symbols

BDFH is a subsequence of **ABCDEFGH**

C is a subsequence of **ABCDEFGH**

ABCDEFGH is a subsequence of **ABCDEFGH**

A sequence **Z** is a **LONGEST COMMON SUBSEQUENCE (LCS)** of **X** and **Y**

if **Z** is a subsequence of both **X** and **Y**

and any sequence longer than **Z** is not a subsequence of at least one of **X** or **Y**.

ABDFGH is the LCS of

ABCDEFGH and **ABDFGHI**

APPLICATIONS OF LCS

Bioinformatics!

Detect similarities between DNA or protein sequences

Computational linguistics!

Extract similarities in words/word-forms and determine how words are related

the diff unix command!

Identify differences between the contents of two files

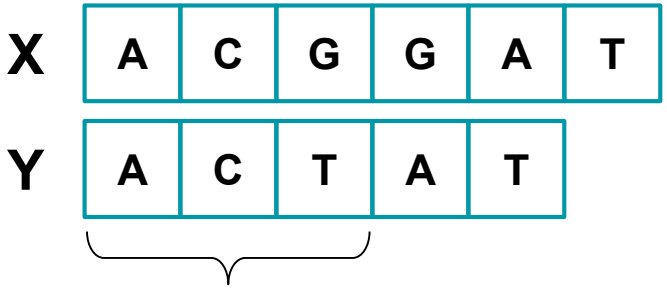
and so much more...

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM: Find the length of LCS's of *prefixes* to X and Y.



Notation: denote this prefix **ACT** by Y_3

Examples:

C[2,3] = 2 (LCS of X_2 and Y_3 is AC)

C[5,4] = 3 (LCS of X_5 and Y_4 is ACA)

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

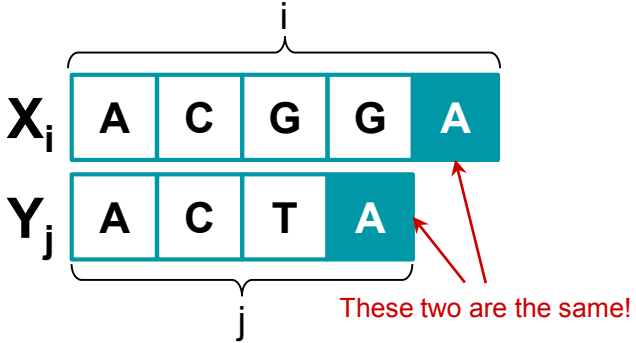
Why is this a good choice?

STEP 1: OPTIMAL SUBSTRUCTURE

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

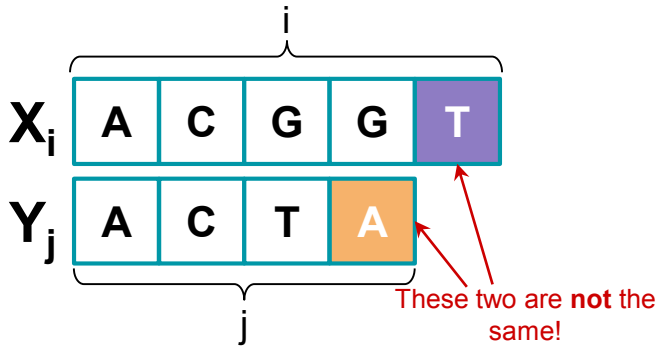
Case 1: X[i] = Y[j]



Then, **C[i, j] = 1 + C[i–1, j–1]**

because $LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1})$ followed by **A** .

Case 2: X[i] ≠ Y[j]



Then, **C[i, j] = max{ C[i–1, j], C[i, j–1] }**

Give **A** a chance to "match": $LCS(X_i, Y_j) = LCS(X_{i-1}, Y)$

Give **T** a chance to "match": $LCS(X_i, Y_j) = LCS(X_i, Y_{j-1})$

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 2: RECURSIVE FORMULATION

Our recursive formulation:

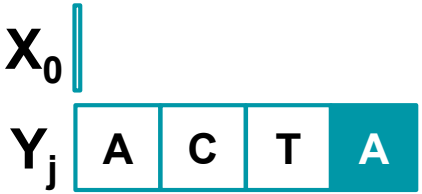
$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i-1, j], C[i, j-1] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

STEP 2: RECURSIVE FORMULATION

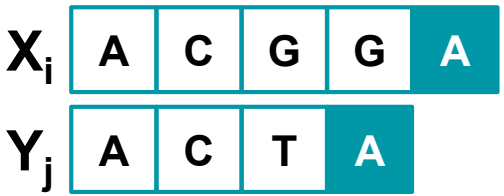
Our recursive formulation:

$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i-1, j], C[i, j-1] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

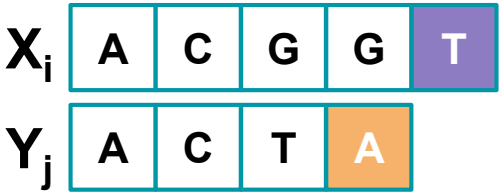
CASE 0 (base case)



CASE 1



CASE 2



STEP 3: WRITE A DP ALGORITHM

```
LCS-LENGTH(X, Y)
 1   m = X.length
 2   n = Y.length
 3   let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4   for i = 1 to m
 5       c[i, 0] = 0
 6   for j = 0 to n
 7       c[0, j] = 0
 8   for i = 1 to m
 9       for j = 1 to n
10           if x_i == y_j
11               c[i, j] = c[i − 1, j − 1] + 1
12               b[i, j] = "↖"
13           elseif c[i − 1, j] ≥ c[i, j − 1]
14               c[i, j] = c[i − 1, j]
15               b[i, j] = "↑"
16           else c[i, j] = c[i, j − 1]
17               b[i, j] = "←"
18   return c and b
```

Example - 1

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Example - 2

i		j	0	1	2	3	4	5	6	7	8	9	10
				p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0			0	0	0	0	0	0	0	0	0	0	0
1	*p*		0	1	1	1	1	1	1	1	1	1	1
2	*r*		0	1	2	2	2	2	2	2	2	2	2
3	*e*		0	1	2	2	2	2	2	3	3	3	3
4	*s*		0	1	2	2	2	2	2	3	3	3	3
5	*i*		0	1	2	2	2	3	3	3	3	3	3
6	*d*		0	1	2	2	2	3	4	4	4	4	4
7	*e*		0	1	2	2	2	3	4	5	5	5	5
8	*n*		0	1	2	2	2	3	4	5	6	6	6
9	*t*		0	1	2	2	2	3	4	5	6	6	6

Running time and memory: O(*mn*) and O(*mn*).

Output: Priden

Matrix Chain Multiplication

Matrix-chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied
  - That is, we want to compute the product $A_1A_2\ldots A_n$
- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions
- Can you write the algorithm to multiply two matrices?

Matrix-chain Multiplication

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 = 2{,}500$ scalar multiplications

Total: 7,500

  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 = 50{,}000$ scalar multiplications

Total: 75,000

Matrix chain multiplication: Dynamic Programming

The optimum time to compute $A_{i..k}$ is $m[i, k]$ and optimum time for $A_{k+1..j}$ is in $m[k + 1, j]$. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \times p_k \times p_j$. This suggests the following recursive rule:

$$m[i, i] = 0$$
$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j)$$

Matrix chain multiplication: Dynamic Programming

We do not want to calculate m entries recursively. So how should we proceed? We will fill m along the diagonals. Here is how. Set all $m[i, i] = 0$ using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are $m[1, 2], m[2, 3], m[3, 4], \ldots, m[n-1, n]$. $m[1, 2]$, for example is

$$m[1, 2] = m[1, 1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2$$

For example, for m for product of 5 matrices at this stage would be:

$m[1,1]$	$\leftarrow m[1,2]$ $\downarrow$			
	$m[2,2]$	$\leftarrow m[2,3]$ $\downarrow$		
		$m[3,3]$	$\leftarrow m[3,4]$ $\downarrow$	
			$m[4,4]$	$\leftarrow m[4,5]$ $\downarrow$
				$m[5,5]$

Matrix chain multiplication: Dynamic Programming

Next, we compute cost of multiplication for sequences of three matrices. These are $m[1,3], m[2,4], m[3,5], \ldots, m[n-2,n]$. $m[1,3]$, for example is

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 \end{cases}$$

We repeat the process for sequences of four, five and higher number of matrices. The final result will end up in $m[1,n]$.

Matrix chain multiplication: Dynamic Programming

Example: Let us go through an example. We want to find the optimal multiplication order for

$$\underset{(5\times4)}{A_1} \cdot \underset{(4\times6)}{A_2} \cdot \underset{(6\times2)}{A_3} \cdot \underset{(2\times7)}{A_4} \cdot \underset{(7\times3)}{A_5}$$

We will compute the entries of the m matrix starting with the base condition. We first fill that main diagonal:

0				
	0			
		0		
			0	
				0

- Here A = { A1,A2,A3,A4,A5} ; starts from index 1
- and P = {5,4,6,2,7,3}; starts from index 0

Matrix chain multiplication: Dynamic Programming

Next, we compute the entries in the first super diagonal, i.e., the diagonal above the main diagonal:

$$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2 = 0 + 0 + 5 \cdot 4 \cdot 6 = 120$$
$$m[2,3] = m[2,2] + m[3,3] + p_1 \cdot p_2 \cdot p_3 = 0 + 0 + 4 \cdot 6 \cdot 2 = 48$$
$$m[3,4] = m[3,3] + m[4,4] + p_2 \cdot p_3 \cdot p_4 = 0 + 0 + 6 \cdot 2 \cdot 7 = 84$$
$$m[4,5] = m[4,4] + m[5,5] + p_3 \cdot p_4 \cdot p_5 = 0 + 0 + 2 \cdot 7 \cdot 3 = 42$$

The matrix m now looks as follows:

0	120			
	0	48		
		0	84	
			0	42
				0

Matrix chain multiplication: Dynamic Programming

We now proceed to the second super diagonal. This time, however, we will need to try two possible values for k. For example, there are two possible splits for computing $m[1,3]$; we will choose the split that yields the minimum:

$$m[1,3] = m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 == 0 + 48 + 5 \cdot 4 \cdot 2 = 88$$
$$m[1,3] = m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 120 + 0 + 5 \cdot 6 \cdot 2 = 180$$

the minimum $m[1,3] = 88$ occurs with $k = 1$

Matrix chain multiplication: Dynamic Programming

Similarly, for $m[2,4]$ and $m[3,5]$:

$$m[2,4] = m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 = 0 + 84 + 4 \cdot 6 \cdot 7 = 252$$
$$m[2,4] = m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 = 48 + 0 + 4 \cdot 2 \cdot 7 = 104$$
$$\text{minimum } m[2,4] = 104 \text{ at } k = 3$$

$$m[3,5] = m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 = 0 + 42 + 6 \cdot 2 \cdot 3 = 78$$
$$m[3,5] = m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 = 84 + 0 + 6 \cdot 7 \cdot 3 = 210$$
$$\text{minimum } m[3,5] = 78 \text{ at } k = 3$$

Matrix chain multiplication: Dynamic Programming

With the second super diagonal computed, the m matrix looks as follow:

0	120	88		
	0	48	104	
		0	84	78
			0	42
				0

Matrix chain multiplication: Dynamic Programming

We repeat the process for the remaining diagonals. However, the number of possible splits (values of k) increases:

$$m[1,4] = m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 104 + 5 \cdot 4 \cdot 7 = 244$$
$$m[1,4] = m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 120 + 84 + 5 \cdot 6 \cdot 7 = 414$$
$$m[1,4] = m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 88 + 0 + 5 \cdot 2 \cdot 7 = 158$$
$$\text{minimum } m[1,4] = 158 \text{ at } k = 3$$

$$m[2,5] = m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 = 0 + 78 + 4 \cdot 6 \cdot 3 = 150$$
$$m[2,5] = m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 = 48 + 42 + 4 \cdot 2 \cdot 3 = 114$$
$$m[2,5] = m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 = 104 + 0 + 4 \cdot 7 \cdot 3 = 188$$
$$\text{minimum } m[2,5] = 114 \text{ at } k = 3$$

Matrix chain multiplication: Dynamic Programming

0	120	88	158	
	0	48	104	114
		0	84	78
			0	42
				0

That leaves the $m[1,5]$ which can now be computed:

$$m[1,5] = m[1,1] + m[2,5] + p_0 \cdot p_1 \cdot p_5 = 0 + 114 + 5 \cdot 4 \cdot 3 = 174$$
$$m[1,5] = m[1,2] + m[3,5] + p_0 \cdot p_2 \cdot p_5 = 120 + 78 + 5 \cdot 6 \cdot 3 = 288$$
$$m[1,5] = m[1,3] + m[4,5] + p_0 \cdot p_3 \cdot p_5 = 88 + 42 + 5 \cdot 2 \cdot 3 = 160$$
$$m[1,5] = m[1,4] + m[5,5] + p_0 \cdot p_4 \cdot p_5 = 158 + 0 + 5 \cdot 7 \cdot 3 = 263$$
$$\text{minimum } m[1,5] = 160 \text{ at } k = 3$$

Matrix chain multiplication: Dynamic Programming

- **Matrix “m”**

We thus have the final cost matrix.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

and the split k values that led to a minimum m[i, j] value

- **Matrix “s”**

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Matrix chain multiplication: Dynamic Programming

- Matrix “m” top right value is minimum cost for multiplying five matrices and Matrix ”s” is used to put parenthesis or order in which they will be multiplied to get that minimum cost.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Based on the computation, the minimum cost for multiplying the five matrices is 160 and the optimal order for multiplication is

$$((A_1(A_2A_3))(A_4A_5))$$

Matrix chain multiplication: Dynamic Programming

- **How to put parenthesis by “s” matrix :**
- To find order or parenthesis, start from top right value s[1,5]. At s[1,5], we have k=3 where row value is A1 and column value is A5 so this means that divide A1,A2,A3,A4,A5 into (A1A2A3)(A4A5).
- Now it has been divided into two parts. First part is from 1 to 3 and second from 4 to 5. So look for s[1,3] and s[4,5].
- At s[1,3], we have k=1 where row value is A1 and column value is A3 so this means that divide A1,A2,A3 into (A1)(A2A3) so the complete becomes (A1(A2A3))(A4A5)
- At s[4,5], we have k=4 where row value is A4 and column value is A5 so this means that divide A4,A5 into (A4)(A5) which wont make any effect so the complete becomes (A1(A2A3))(A4A5)

- **Matrix “s”**

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Matrix chain multiplication: Dynamic Programming

Here is the dynamic programming based algorithm for computing the minimum cost of chain matrix multiplication.

```
MATRIX-CHAIN(p, N)
 1  for i = 1, N
 2  do m[i, i] ← 0
 3  for L = 2, N
 4  do
 5     for i = 1, n − L + 1
 6     do j ← i + L − 1
 7        m[i, j] ← ∞
 8        for k = 1, j − 1
 9        do t ← m[i, k] + m[k + 1, j] + p_{i−1} · p_k · p_j
10           if (t < m[i, j])
11              then m[i, j] ← t;   s[i, j] ← k
```

Analysis: There are three nested loops. Each loop executes a maximum n times. Total time is thus $\Theta(n^3)$.

Matrix chain multiplication: Dynamic Programming

The s matrix stores the values k. The s matrix can be used to extracting the order in which matrices are to be multiplied. Here is the algorithm that caries out the matrix multiplication to compute $A_{i..j}$:

```
MULTIPLY(i, j)
1  if (i = j)
2     then return A[i]
3     else  k ← s[i, j]
4           X ← MULTIPLY(i, k)
5           Y ← MULTIPLY(k + 1, j)
6           return X · Y
```