

# What is Dynamic Programming

- Like DaC, **Dynamic Programming** is another useful method for designing efficient algorithms.
- Why the name?

## Eye of the Hurricane: An Autobiography - A quote from Richard Bellman

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision process. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. ... I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. .... Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object. So I used it as an umbrella for my activities.

# General Description of Dynamic Programming

- Divide the problem into smaller subproblems (of the same type).
- Solve each subproblem.
- Combine the solutions of subproblems into the solution of the original problem.
- Looks familiar? It's identical to DaC!
- But they differ substantially in details.
- For DaC:
  - The sub-problems are independent, they do not overlap.
  - We solve sub-problems in a top-down fashion. Namely, solve the largest sub-problem first, then the second largest ...
  - Usually by recursive calls.
- For Dynamic Programming:
  - The sub-problems are intermingled, they do overlap.
  - We solve sub-problems in a bottom-up fashion. Namely, solve the smallest sub-problem first, then the second smallest ...

## MergeSort

Sort array  $A[1..n]$

Divide it into two subproblems: Sort  $A[1..n/2]$  and Sort  $A[(n/2 + 1)..n]$

- The two sub-problems **do not overlap**. They are totally **independent**.
- We solve these two largest sub-problems, by recursive calls.
- Move to smaller problems Sort  $A[1..n/4]$ , Sort  $A[(n/4 + 1)..n/2]$  etc.

# Example for Dynamic Programming

$$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$$

- We divide  $\text{Fib}(n)$  into two sub-problems  $\text{Fib}(n - 1)$  and  $\text{Fib}(n - 2)$ .
- They **overlap, not totally independent**:  $\text{Fib}(n - 1)$  **contains**  $\text{Fib}(n - 2)$ .
- If we solve the largest sub-problems  $\text{Fib}(n - 1)$  and  $\text{Fib}(n - 2)$  first by using recursive calls, as we did before, we get exp time algorithm.

## **Fib( $n$ )**

```
1: Fib[0] = 0; Fib[1] = 1;  
2: for  $i = 2$  to  $n$  do  
3:   Fib[ $i$ ] = Fib[ $i - 1$ ] + Fib[ $i - 2$ ]  
4: end for  
5: output Fib[ $n$ ]
```

- Solves the smallest sub-problem  $\text{Fib}[0]$ , then  $\text{Fib}[1]$  ... bottom-up.
- It clearly takes  $O(n)$  time.

# Matrix Chain Product Problem

## Matrix Chain Product Problem

Input:  $n$  matrices  $A_1, A_2, \dots, A_n$ , the size of  $A_i$  is  $p_{i-1} \times p_i$  for  $i = 1, \dots, n$ .

Compute the **chain product**:  $A_1 \times A_2 \times \dots \times A_n$

- This is a basic operation in linear algebra.
- To calculate

$$p_{i-1} \left\{ \begin{array}{ccc} \leftarrow & p_i & \rightarrow \\ & A_i & \end{array} \right\} \times p_i \left\{ \begin{array}{ccc} \leftarrow & p_{i+1} & \rightarrow \\ & A_{i+1} & \end{array} \right\} = p_{i-1} \left\{ \begin{array}{ccc} \leftarrow & p_{i+1} & \rightarrow \\ & C & \end{array} \right\}$$

We need to calculate  $p_{i-1} \cdot p_{i+1}$  entries, and each entry takes  $p_i$  **scalar multiplications**. So it totally takes  $p_{i-1} \cdot p_i \cdot p_{i+1}$  scalar multiplications.

- $\times$  is associative. So we can compute the chain product in different ways, **with different total cost**.

# Example

## Example

$$100 \left\{ \begin{array}{c} \leftarrow 2 \rightarrow \\ A_1 \end{array} \right\}, \quad 2 \left\{ \begin{array}{c} \leftarrow 50 \rightarrow \\ A_2 \end{array} \right\}, \quad 50 \left\{ \begin{array}{c} \leftarrow 6 \rightarrow \\ A_3 \end{array} \right\}$$

There are two ways to compute  $A_1 \times A_2 \times A_3$ :

- $(A_1 \times A_2) \times A_3$ :
  - Calculate  $X = A_1 \times A_2$  takes  $100 \cdot 2 \cdot 50 = 10000$  ops.
  - Calculate  $X \times A_3$  takes  $100 \cdot 50 \cdot 6 = 30000$  ops.
  - Total cost is  $10000 + 30000 = 40000$  ops.
- $A_1 \times (A_2 \times A_3)$ :
  - Calculate  $Y = A_2 \times A_3$  takes  $2 \cdot 50 \cdot 6 = 600$  ops.
  - Calculate  $A_1 \times Y$  takes  $100 \cdot 2 \cdot 6 = 1200$  ops.
  - Total cost is  $600 + 1200 = 1800$  ops.
- The total costs are very different.

# Matrix Chain Product Problem

## Matrix Chain Product Problem

Input:  $n$  matrices  $A_1, A_2, \dots, A_n$ , the size of  $A_i$  is  $p_{i-1} \times p_i$  for  $i = 1, \dots, n$ .

Find: **The best way** to compute the **chain product**:  $A_1 \times A_2 \times \dots \times A_n$  so that the total cost is **minimum**.

- A **way** to calculate the product is a **parenthesization** of the chain.
- A simple algorithm:

### Brute-Force

- 1 Enumerate all parenthesizations of  $A_1 \times \dots \times A_n$ .
- 2 For each, compute the total cost.
- 3 Pick the one with the lowest total cost.

# Matrix Chain Product Problem

- But how many possible solutions?
- Let  $C_i$  be the number of different ways to put parenthesis into a  $n$  term chain.
- $C_0$  and  $C_1$  are meaningless. For simplicity, define  $C_0 = 0$  and  $C_1 = 1$ .
- It's easy to see:  $C_2 = 1$  and  $C_3 = 2$ .
- $C_4 = 5$ :  
 $((A_1 \times A_2) \times A_3) \times A_4, (A_1 \times A_2) \times (A_3 \times A_4), (A_1 \times (A_2 \times A_3)) \times A_4,$   
 $A_1 \times ((A_2 \times A_3) \times A_4), A_1 \times (A_2 \times (A_3 \times A_4))$



# Matrix Chain Product Problem

We will show:

$$C_n = \frac{1}{n} \binom{2n-2}{n-1} = \frac{(2n-2)!}{n!(n-1)!}$$

- $C_n$  is called the  $n$ th Catalan number.
- $C_4 = \frac{6!}{4! \cdot 3!} = 5$ .
- $C_{10} = 4862$ ,  $C_{15} = 2,674,440$ .
- By using Stirling's approximation, we have:

$$C_n = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

- Thus, the Brute-Force algorithm takes exponential time. This is not acceptable.

# Generating Function Method

- **Generating Function Method** is a systematic way for solving recursive sequences (linear or non-linear)
- You may wonder why our method for solving linear recursive sequences works. Here we will see why.
- Let  $\{f_0, f_1, f_2 \dots\} = \{f_n\}_{n \geq 0}$  be a recursively defined sequence (linear or non-linear.)

# Generating Function Method

- 1 Define a formal series  $f(x) = \sum_{n=0}^{\infty} f_n x^n$   
 $f(x)$  is called the **generating function** of  $\{f_n\}_{n \geq 0}$ .
- 2 Using the recursive definition of  $\{f_n\}_{n \geq 0}$ , try to get an equation that only involves  $f(x)$  and  $x$ , without mentioning  $f_n$ .
  - This is the key step of this method.
  - For linear recursive sequences, we have an easy systematic way to do this.
  - For other cases, we will need good luck!
- 3 Solving this equation for  $f(x)$  in terms of  $x$ .
- 4 Find the **Taylor Series** of  $f(x)$ :

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n$$

- 5 Then

$$f_n = \frac{f^{(n)}(0)}{n!}$$

# Examples

Fib numbers:  $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$

Step 1: define  $f(x) = \sum_{n=0}^{\infty} f_n x^n$

Step 2: Try to get an equation of  $f(x)$  and  $x$ ;

$$\begin{array}{rcccccccc} f(x) & = & f_0 & + & f_1 x & + & f_2 x^2 & \cdots & f_n x^n & \cdots \\ x f(x) & = & & & f_0 x & + & f_1 x^2 & \cdots & f_{n-1} x^n & \cdots \\ x^2 f(x) & = & & & & & f_0 x^2 & \cdots & f_{n-2} x^n & \cdots \\ f(x) - x f(x) - x^2 f(x) & = & f_0 & + & (f_1 - f_0)x & + & (f_2 - f_1 - f_0)x^2 & \cdots & (f_n - f_{n-1} - f_{n-2})x^n & \cdots \\ & = & x & & & & & & & \end{array}$$

The last line is because:  $f_0 = 0, f_1 = 1, f_2 - f_1 - f_0 = 0$  and  $f_n - f_{n-1} - f_{n-2} = 0$ .  
This implies  $f(x)(1 - x - x^2) = x$ . Hence:

$$f(x) = \frac{x}{1 - x - x^2}$$

# Examples

Step 3: Find the Taylor Series of  $f(x)$ . Because  $f(x)$  is a fraction of polynomials, instead of using the formula  $f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}}{n!} x^n$ , we have an easier way.

## Partial Fraction

Let  $p(z)$  be a polynomial of degree  $k$ . Let  $q(z)$  be a polynomial of degree at most  $k - 1$ . Let  $\alpha_1, \alpha_1, \dots, \alpha_k$  be the roots of the equation  $p(z) = 0$ .

- If all roots are distinct, then for some constants  $a_1, a_2, \dots, a_k$ :

$$\frac{q(z)}{p(z)} = \frac{a_1}{z - \alpha_1} + \frac{a_2}{z - \alpha_2} + \dots + \frac{a_k}{z - \alpha_k}$$

## Partial Fraction - Continued:

- If there are repeated roots, say  $\alpha_1 = \alpha_2 = \dots = \alpha_t$  repeat  $t$  times, then the portion in the above formula corresponding to the roots  $\alpha_1 \dots \alpha_t$  becomes:

$$\dots \frac{a_1}{(z - \alpha_1)} + \frac{a_2 z^1}{(z - \alpha_1)^2} + \dots + \frac{a_t z^{t-1}}{(z - \alpha_1)^t} + \dots$$

- This is a fact from algebra.
- Extensively used in Calculus.

# Examples

Back to the generating function for Fib numbers:

$$\begin{aligned}f(x) &= \frac{x}{1-x-x^2} = \frac{1/x}{(1/x)^2 - (1/x) - 1} = \frac{z}{z^2 - z - 1} \quad (\text{here } z = 1/x) \\&= \frac{a_1}{z - \alpha_1} + \frac{a_2}{z - \alpha_2} \quad (\text{Partial Fraction, } \alpha_1, \alpha_2 \text{ are the roots of } z^2 - z - 1) \\&= \frac{a_1}{1/x - \alpha_1} + \frac{a_2}{1/x - \alpha_2} = \frac{a_1 x}{1 - \alpha_1 x} + \frac{a_2 x}{1 - \alpha_2 x} \\&= a_1 x \sum_{n=0}^{\infty} (\alpha_1 x)^n + a_2 x \sum_{n=0}^{\infty} (\alpha_2 x)^n \quad (\text{sum of geometric series}) \\&= \sum_{n=0}^{\infty} (a_1 \alpha_1^n + a_2 \alpha_2^n) \cdot x^{n+1}\end{aligned}$$

This is the Taylor Series of  $f(x)$ . Thus  $f_{n+1} = a_1(\alpha_1)^n + a_2(\alpha_2)^n$ . Or

$$f_n = a_1(\alpha_1)^{n-1} + a_2(\alpha_2)^{n-1}$$

## Linear Recursive Sequences

By using this method, we can find the solution of any linear recursive sequences. The result is the procedure we discussed before.

# Catalan Number

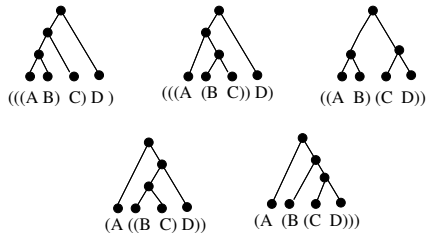
- Let  $c_n$  be the  $n$ th Catalan number. Namely:  
 $c_n$  = the number of different parenthesizations of  $n$  terms.
- We have  $c_0 = 0$ ,  $c_1 = 1$  (these two are defined for convenience),  
 $c_2 = 1$ ,  $c_3 = 2$ ,  $c_4 = 5 \dots$
- We want to show:

$$c_n = \frac{1}{n} \binom{2n-2}{n-1} = \frac{(2n-2)!}{n!(n-1)!}$$

- Actually  $c_n$  is also the number of distinct  $n$ -leaf binary trees.
- This is because there is a 1-1 correspondence between the set of  $n$ -leaf binary trees and the set of parenthesizations of  $n$  terms.



# Correspondence with Binary Trees



The case of  $n = 4$ .

# Catalan Number

In order to find the solution for the sequence  $\{c_n\}_{n \geq 0}$ , first we need to find a recursive formula for  $c_n$ .

Consider a particular kind parenthesization:

$$(A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n)$$

where the last two pairs of parenthesis are  $(A_1 \cdots A_k)$  and  $(A_{k+1} \cdots A_n)$

- The number of parenthesizations of  $A_1 \cdots A_k$  is  $c_k$ .
- The number of parenthesizations of  $A_{k+1} \cdots A_n$  is  $c_{n-k}$ .
- For each parenthesization of  $A_1 \cdots A_k$  and each parenthesization of  $A_{k+1} \cdots A_n$ , we get a valid parenthesization of  $A_1 \cdots A_n$ .
- So the number of parenthesization of  $A_1 \cdots A_n$  that satisfies this condition is  $c_k \cdot c_{n-k}$ .
- The possible values for  $k$ :  $1 \leq k < n$ . Therefore:

# Catalan Number

$$c_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} c_k \cdot c_{n-k} & \text{if } n > 1 \end{cases}$$

Check a few cases:

- $c_2 = c_1 \cdot c_1 = 1 \cdot 1 = 1$
- $c_3 = c_1 \cdot c_2 + c_2 \cdot c_1 = 1 \cdot 1 + 1 \cdot 1 = 2$
- $c_4 = c_1 \cdot c_3 + c_2 \cdot c_2 + c_3 \cdot c_1 = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = 5$

Now we use the **generating function method** to find the solution for  $\{c_n\}_{n \geq 0}$ .

Step 1. Define  $f(x) = \sum_{n=0}^{\infty} c_n x^n$ .

Step 2: Try to get an equation involving only  $f(x)$  and  $x$ . Since  $\{c_n\}_{n \geq 0}$  is **not a linear recursive sequence**, this is much harder to do.

# Catalan Number

$$\begin{aligned}(f(x))^2 &= (c_1x + c_2x^2 + c_3x^3 \cdots) \cdot (c_1x + c_2x^2 + c_3x^3 \cdots) \\&= (c_1 \cdot c_1)x^2 + (c_1 \cdot c_2 + c_2 \cdot c_1)x^3 + (c_1 \cdot c_3 + c_2 \cdot c_2 + c_3 \cdot c_1)x^4 + \\&\quad \cdots + (\sum_{k=1}^{n-1} c_k \cdot c_{n-k})x^n + \cdots \\&= c_2x^2 + c_3x^3 + c_4x^4 + \cdots + c_nx^n + \cdots \\&= f(x) - c_1x = f(x) - x\end{aligned}$$

It's pure luck we get this simple equation!

Step 3. Solve this equation for  $f(x)$  in terms of  $x$  (note: this is a quadratic equation for  $f(x)$ ):

$$f(x) = \frac{1}{2}(1 - \sqrt{1 - 4x})$$

Step 4. Find Taylor series for  $f(x)$ . There is no short cut this time.

# Catalan Number

$$\begin{aligned}f(x) &= \frac{1}{2}(1 - \sqrt{1 - 4x}) \\f'(x) &= \frac{1}{2} \frac{1}{2}(1 - 4x)^{-\frac{1}{2}} \cdot 4 \\&= (1 - 4x)^{-\frac{1}{2}}\end{aligned}$$

$$f^{(2)}(x) = \frac{1}{2}(1 - 4x)^{-\frac{3}{2}} 4$$

$$f^{(3)}(x) = \frac{1}{2} \frac{3}{2}(1 - 4x)^{-\frac{5}{2}} 4^2$$

$$f^{(4)}(x) = \frac{1}{2} \frac{3}{2} \frac{5}{2}(1 - 4x)^{-\frac{7}{2}} 4^3$$

$$f^{(5)}(x) = \frac{1}{2} \frac{3}{2} \frac{5}{2} \frac{7}{2}(1 - 4x)^{-\frac{9}{2}} 4^4$$

...

$$\begin{aligned}f^{(n)}(x) &= \frac{1}{2} \frac{3}{2} \cdots \frac{2n-3}{2}(1 - 4x)^{-\frac{2n-1}{2}} 4^{n-1} \\&\dots\end{aligned}$$

$$f(0) = 0$$

$$f'(0) = 1 = 1 \cdot 2^0$$

$$f^{(2)}(0) = 1 \cdot 2^1$$

$$f^{(3)}(0) = 1 \cdot 3 \cdot 2^2$$

$$f^{(4)}(0) = 1 \cdot 3 \cdot 5 \cdot 2^3$$

$$f^{(5)}(0) = 1 \cdot 3 \cdot 5 \cdot 7 \cdot 2^4$$

$$f^{(n)}(0) = 1 \cdot 3 \cdot 5 \cdots (2n-3) \cdot 2^{n-1}$$

# Catalan Number

Step 4: Hence:

$$\begin{aligned}f_n &= \frac{f^{(n)}(0)}{n!} = \frac{1 \cdot 3 \cdot 5 \cdots (2n-3)}{n!} \cdot 2^{n-1} \\&= \frac{1 \cdot 3 \cdot 5 \cdots (2n-3)}{n!} \cdot \frac{2 \cdot 4 \cdots (2n-2)}{2 \cdot 4 \cdots (2n-2)} \cdot 2^{n-1} \\&= \frac{(2n-2)!}{n!(n-1)!2^{n-1}} \cdot 2^{n-1} \\&= \frac{(2n-2)!}{n!(n-1)!}\end{aligned}$$

# Matrix Chain Product Problem

## Recursive Formulation

- Let  $A_{i..j}$  ( $1 \leq i \leq j \leq n$ ) be the chain product:  $A_i \times A_{i+1} \times \cdots \times A_j$ .
- Let  $m[i,j]$  be the minimum number of scalar multiplications needed to compute  $A_{i..j}$ .
- We derive a recursive formula to compute  $m[i,j]$ .
- For  $i > j$ ,  $m[i,j]$  is undefined and not needed.
- For  $i = j$ ,  $A_{i..j} = A_i$ , we have nothing to compute. So  $m[i,i] = 0$  for all  $1 \leq i \leq n$ .
- For  $i < j$ : Suppose that we know the optimal parenthesization in  $A_i \times \cdots \times A_j$ , that gives the minimum cost.

# Matrix Chain Product Problem

- Further assume that

$$\underbrace{(A_i \times \cdots \times A_k)}_{A_{i..k}} \times \underbrace{(A_{k+1} \times \cdots \times A_j)}_{A_{(k+1)..j}}$$

are the last two pairs of parenthesis in the optimal parenthesization.

- The minimum cost for calculating  $A_{i..k}$  is  $m[i, k]$  by definition.
- The minimum cost for calculating  $A_{(k+1)..j}$  is  $m[(k+1), j]$  by definition.
- The size of  $A_{i..k}$  is  $p_{i-1} \times p_k$ . The size of  $A_{(k+1)..j}$  is  $p_k \times p_j$ . The cost of calculating  $A_{i..k} \times A_{(k+1)..j}$  is  $p_{i-1} \cdot p_k \cdot p_j$ .
- Thus the total cost is  $m[i, k] + m[(k+1), j] + p_{i-1} \cdot p_k \cdot p_j$ .
- Of course, we do not know where the last two pairs of parenthesis are located. So we consider all possible positions  $i \leq k < j$ , and take the minimum.



# Recursive Algorithm

$$m[i,j] = \begin{cases} \text{undefined} & \text{if } i > j \\ 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[(k+1),j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{if } i < j \end{cases}$$

- $m[1,n]$  is the minimum cost for computing  $A_1 \times \cdots \times A_n$ .
- The following recursive top-down algorithm calculates  $m[i,j]$ . We call MCP-REC(1,  $n$ ) to solve our problem.

## MCP-REC( $i,j$ )

- 1 If  $i > j$ , return “undefined”
- 2 If  $i = j$ , return 0
- 3 If  $i < j$ , return  $\min_{i \leq k < j} \{ \text{MCP-REC}(i,k) + \text{MCP-REC}(k+1,j) + p_{i-1} \cdot p_k \cdot p_j \}$

The algorithm is simple. But is it efficient?

# Recursive Algorithm

- Say we call MCP-REC(1, 5) to compute  $m[1, 5]$ .
- It will need to solve each of the following recursively:  
 $m[1, 1], m[2, 5], m[1, 2], m[3, 5], m[1, 3], m[4, 5], m[1, 4], m[5, 5]$ .
- Each of these will make several recursive calls.
- Many of these subproblems **overlap**. And we make repeated calls to solve the **same subproblem over and over again!**
- It can be shown this algorithm takes  $\Theta(C_n) = \Omega(4^n/n^{3/2})$  time. Not acceptable.
- However, there are only at most  $n^2$  subproblems! Why would it take exp time?
- All we have to do is calculate the 2D array  $m[1..n, 1..n]$  according to the above formula.
- Must make sure when calculating  $m[i, j]$ , all entries needed have been calculated already.
- This gives the dynamic programming algorithm.

# Dynamic Programming Algorithm

## DynamicProg

- ➊ Fill all entries below main diagonal by  $-\infty$ .
  - ➋ Fill all entries on the main diagonal (from  $m[1, 1]$  to  $m[n, n]$ ) by 0.
  - ➌ Fill the 2nd main diagonal according to above formula.
  - ➍ Fill the 3rd main diagonal ...
  - ➎ Output  $m[1, n]$ .
- 
- There are  $n - 1$  entries on the 2nd main diagonal, each is the min of 1 terms.
  - There are  $n - 2$  entries on the 3rd diagonal, each is the min of 2 terms.
  - .....
  - Only one entry ( $m[1, n]$ ) on the last ( $n$ th) diagonal, it is the min of  $n - 1$  terms.
  - So the total runtime is  $\Theta$  of  $\sum_{i=1}^{n-1} i \cdot (n - i) = \Theta(n^3)$ .

# Dynamic Programming Algorithm

- This algorithm only calculates the min cost of the MCP. It doesn't tell us how to put parenthesis into the chain.
- In order to do so, we must keep additional information.
- Define a 2D array  $S[1..n, 1..n]$  where, for  $i \leq j$ ,  $S[i, j] = k$  if  $k$  corresponds to the term in the min sign that gives the value of  $m[i, j]$ .
- Using  $S[*, *]$ , the following algorithm puts parenthesis into the product chain.

## MCP-Multiply( $A, S, i, j$ )

```
1: if  $i < j$  then
2:    $X = \text{MCP-Multiply}(A, S, i, S[i, j])$ 
3:    $Y = \text{MCP-Multiply}(A, S, S[i, j] + 1, j)$ 
4:   return  $X \times Y$ 
5: else
6:   return  $A_i$ 
7: end if
```

# Example

## Example: $A_1 \times \cdots \times A_6$

The dimensions are given below.

$i$	0	1	2	3	4	5	6
$p_i$	10	20	1	40	5	30	15

The matrix  $m[*,*]$  is as follows (the  $S[i,j]$  value is given in ( )):

$m_{ij}$	1	2	3	4	5	6
1	0	200(1)	600(2)	450(2)	850(2)	1150(2)
2	-	0	800(2)	300(2)	950(2)	1100(2)
3	-	-	0	200(3)	350(4)	800(5)
4	-	-	-	0	6000(4)	5250(4)
5	-	-	-	-	0	2250(5)
6	-	-	-	-	-	0

So the optimal way is:  $(M_1 M_2)((M_3 M_4) M_5) M_6$ , which needs 1150 scalar multiplications.

The calculations are given below:

$$m_{11} = m_{22} = m_{33} = m_{44} = m_{55} = m_{66} = 0$$

$$m_{12} = p_0 p_1 p_2 = 10 \times 20 \times 1 = 200$$

$$m_{23} = p_1 p_2 p_3 = 20 \times 1 \times 40 = 800$$

$$m_{34} = p_2 p_3 p_4 = 1 \times 40 \times 5 = 200$$

$$m_{45} = p_3 p_4 p_5 = 40 \times 5 \times 30 = 6000$$

$$m_{56} = p_4 p_5 p_6 = 5 \times 30 \times 15 = 2250$$

# Example

$$m_{13} = \min\{m_{11} + m_{23} + p_0 p_1 p_3 = 0 + 800 + 10 \times 20 \times 40 = 8800, m_{12} + m_{33} + p_0 p_2 p_3 = 200 + 0 + 10 \times 1 \times 40 = 600\} = 600$$
$$S_{13} = 2$$

$$m_{24} = \min\{m_{22} + m_{34} + p_1 p_2 p_4 = 0 + 200 + 20 \times 1 \times 5 = 300, m_{23} + m_{44} + p_1 p_3 p_4 = 800 + 0 + 20 \times 40 \times 5 = 4800\} = 300$$
$$S_{24} = 2$$

$$m_{35} = \min\{m_{33} + m_{45} + p_2 p_3 p_5 = 0 + 6000 + 1 \times 40 \times 30 = 7200, m_{34} + m_{55} + p_2 p_4 p_5 = 200 + 0 + 1 \times 5 \times 30 = 350\} = 350$$
$$S_{35} = 4$$

$$m_{46} = \min\{m_{44} + m_{56} + p_3 p_4 p_6 = 0 + 2250 + 40 \times 5 \times 15 = 5250, m_{45} + m_{66} + p_3 p_5 p_6 = 6000 + 0 + 40 \times 30 \times 15 = 24000\} = 5250$$
$$S_{46} = 4$$

$$m_{14} = \min\{m_{11} + m_{24} + p_0 p_1 p_4 = 0 + 300 + 10 \times 20 \times 5 = 1300, m_{12} + m_{34} + p_0 p_2 p_4 = 200 + 200 + 10 \times 1 \times 5 = 450, m_{13} + m_{44} + p_0 p_3 p_4 = 600 + 0 + 10 \times 40 \times 5 = 2600\} = 450$$
$$S_{14} = 2$$

$$m_{25} = \min\{m_{22} + m_{35} + p_1 p_2 p_5 = 0 + 350 + 20 \times 1 \times 30 = 950, m_{23} + m_{45} + p_1 p_3 p_5 = 800 + 6000 + 20 \times 40 \times 30 = 30800, m_{24} + m_{55} + p_1 p_4 p_5 = 300 + 0 + 20 \times 5 \times 30 = 3300\} = 950$$
$$S_{25} = 2$$

$$m_{36} = \min\{m_{33} + m_{46} + p_2 p_3 p_6 = 0 + 5250 + 1 \times 40 \times 15 = 5850, m_{34} + m_{56} + p_2 p_4 p_6 = 200 + 2250 + 1 \times 5 \times 15 = 2525, m_{35} + m_{66} + p_2 p_5 p_6 = 350 + 0 + 1 \times 30 \times 15 = 800\} = 800$$
$$S_{36} = 5$$

# Example

$$m_{15} = \min\{m_{11} + m_{25} + p_0p_1p_5 = 0 + 950 + 10 \times 20 \times 30 = 6950, m_{12} + m_{35} + p_0p_2p_5 = 200 + 350 + 10 \times 1 \times 30 = 850, \\ m_{13} + m_{45} + p_0p_3p_5 = 600 + 6000 + 10 \times 40 \times 30 = 18600, m_{14} + m_{55} + p_0p_4p_5 = 450 + 0 + 10 \times 5 \times 30 = 1950\} = 850 \\ S_{15} = 2$$

$$m_{26} = \min\{m_{22} + m_{36} + p_1p_2p_6 = 0 + 800 + 20 \times 1 \times 15 = 1100, m_{23} + m_{46} + p_1p_3p_6 = 800 + 5250 + 20 \times 40 \times 15 = 18050, \\ m_{24} + m_{56} + p_1p_4p_6 = 300 + 2250 + 20 \times 5 \times 15 = 4050, m_{25} + m_{66} + p_1p_5p_6 = 950 + 0 + 20 \times 30 \times 15 = 9950\} = 1100 \\ S_{26} = 2$$

$$m_{16} = \min\{m_{11} + m_{26} + p_0p_1p_6 = 0 + 1100 + 10 \times 20 \times 15 = 4100, m_{12} + m_{36} + p_0p_2p_6 = 200 + 800 + 10 \times 1 \times 15 = 1150, \\ m_{13} + m_{46} + p_0p_3p_6 = 600 + 5250 + 10 \times 40 \times 15 = 11850, m_{14} + m_{56} + p_0p_4p_6 = 450 + 2250 + 10 \times 5 \times 15 = 3450, \\ m_{15} + m_{66} + p_0p_5p_6 = 850 + 0 + 10 \times 30 \times 15 = 5350\} = 1150 \\ S_{16} = 2$$

# Elements of Dynamic Programming

If a problem has the following properties, it's likely that **dynamic programming technique** will work.

## Optimal Substructure Property

An optimal solution to the problem  $Q$  contains within it optimal solutions of subproblems.

## Example

MCP: Let  $(A_1 \cdots A_k) \times (A_{k+1} \cdots A_n)$  be the optimal parenthesization of  $A_1 \cdots A_n$  where  $(A_1 \cdots A_k)$  and  $(A_{k+1} \cdots A_n)$  are the last two pairs of parenthesis in the optimal solution. Then

- The optimal solution restricted to  $A_1 \cdots A_k$  is an optimal parenthesization of  $A_1 \times \cdots \times A_k$ .
- The optimal solution restricted to  $A_{k+1} \cdots A_n$  is an optimal parenthesization of  $A_{k+1} \times \cdots \times A_n$ .

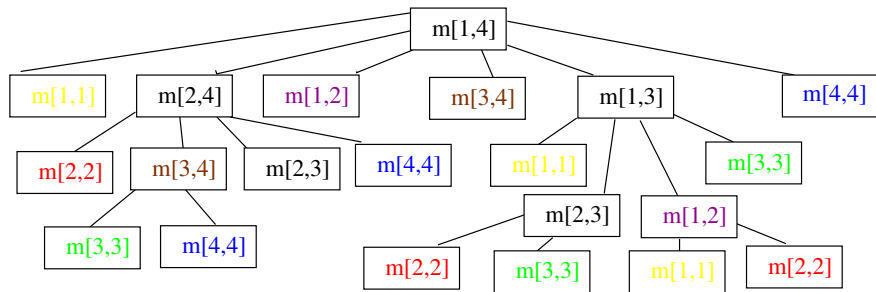


# Elements of Dynamic Programming

## Overlapping Subproblems Property

If the **Optimal Substructure** property holds, then the problem can be divided into sub-problems. If the sub-problems **overlap**, or depend on each other, we say the problem exhibit the **Overlapping Subproblems Property**.

MCP: Consider the recursion tree for computing  $m[1, 4]$



# Elements of Dynamic Programming

- The number of sub-problems is small ( $n^2$ ).
- But the number of recursive calls is exp (because many repeated calls).
- In this case, we should use dynamic programming. Namely solve sub-problems bottom up, starting from the smallest sub-problems.
- We solve all possible sub-problems, even without knowing if we really need to solve them or not. (Do we really need to find  $m[2, 3]$ ? Have no idea!)
- But the total runtime is smaller!
- For MCP, the smallest problems are  $m[i, i]$  for  $1 \leq i \leq n$ .
- For other problems, the smallest sub-problem might mean something else.
- Another way to do this is by Memorization:

# Memorization

- First, fill the entire array  $m[1..n, 1..n]$  by  $-$ .
- Then call the following recursive procedure.

## MCP-Mem( $i, j$ )

```
1: if  $m[i, j] \neq -$  then  
2:   return  $m[i, j]$   
3: else  
4:   if  $i = j$  then  
5:      $m[i, j] = 0$  and return 0  
6:   else  
7:      $m[i, j] = \min_{i \leq k < j} \{ \text{MCP-Mem}(i, k) + \text{MCP-Mem}(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j \}$   
8:     return  $m[i, j]$   
9:   end if  
10: end if
```

- Once we have computed  $m[i, j]$ , it is memorized. So we will not make repeated call to solve it again.

# Summary of Dynamic Programming

**Matrix Chain Product** is a representative of a group of other similar problems. We will discuss other versions in class.

## Summary of Dynamic Programming

- Divide problem into sub-problems. Derive a recursive formula for getting the solution of the original problem from the solutions of sub-problems.
- Verify that the **Optimal Substructure Property** holds.
- Draw the recursion tree for several levels. If it shows the **Overlap Subproblems Property**, then the problem should be solved by dynamic programming.
- Select the proper order for solving subproblems: When solving a sub-problem, the solutions of all needed other sub-problems have been obtained.

# 0/1 Knapsack Problem

## 0/1 Knapsack Problem

**Input:**  $n$  items. Each item  $i$  ( $1 \leq i \leq n$ ) has a *weight*  $w[i]$  (pounds) and a *profit*  $p[i]$  (dollars). We also have a *Knapsack* with *capacity*  $K$  (pounds).

**Problem:** Choose a subset of items and put them into the knapsack so that:

- The total weight of the items we put into the knapsack is at most  $K$ .
- The total profit of the items we put into the knapsack is maximum.

## Application 1:

You win a prize from your favorite candy shop. You are given a knapsack with capacity  $K$  pounds. You can fill the knapsack with any candy box you want. (But if the knapsack breaks, you get nothing). The box  $i$  weighs  $w[i]$  pounds and costs  $p[i]$  dollars. How to pick boxes to maximize the total price?

# 0/1 Knapsack Problem

## Application 2:

The knapsack is a super-computer at a computing center. On a particular day, there are  $K$  seconds computing time available for outside users. Each item is a user job. The job <sub>$i$</sub>  needs  $w[i]$  seconds computing time, and the user will pay  $p[i]$  dollars to the computing center, if job <sub>$i$</sub>  is run by the computing center. How should the computing center select the jobs, in order to maximize its revenue?

## Application 3:

The knapsack is a hard disk drive, with  $K$  bytes capacity. Each item is a file. The size of the file <sub>$i$</sub>  is  $w[i]$  bytes. The file owner will pay  $p[i]$  cents to the disk owner if file <sub>$i$</sub>  is stored on the disk. How should the disk owner select the files in order to maximize his income?

# Mathematical Description of the Problem:

## Mathematical Description of the Problem:

**Input:**  $2n + 1$  positive integers,  $p[1], \dots, p[n], w[1], \dots, w[n], K$ .

**Output:** Find a 0/1 vector  $(x_1, x_2, \dots, x_n)$  such that:

- 1  $\sum_{i=1}^n x_i w[i] \leq K$ ;
- 2  $\sum_{i=1}^n x_i p[i]$  is maximized.

(Here, we put the item <sub>$i$</sub>  into the knapsack iff  $x_i = 1$ .)

# Simple Algorithm

The knapsack problem can be easily solved by the following algorithm.

## Simple algorithm:

- 1 Enumerate all possible  $n$ -bit 0/1 vectors  $(x_1, x_2, \dots, x_n)$ ;
- 2 For each vector  $(x_1, x_2, \dots, x_n)$ , calculate the total weight and the total profit of the subset of the items represented by the vector;
- 3 Select the vector with total weight  $\leq K$  and maximum profit.

This algorithm works fine, except that there are  $2^n$   $n$ -bit 0/1 vectors and the algorithm must loop thru all of them. So the run time is at least  $\Omega(2^n)$ . **This is not acceptable.**



# Dynamic Programming Algorithm:

- Define a 2D array  $\text{Profit}[0..n][0..K]$ .
- The value of the entries in  $\text{Profit}[*][*]$  is defined as:  
 $\text{Profit}[i][j]$  = the maximum profit if the knapsack capacity is  $j$ , and we can fill the knapsack only with a subset of  $\text{item}_1, \text{item}_2, \dots, \text{item}_i$ .
- By this definition, the maximum profit for the original problem is  $\text{Profit}[n][K]$ .
- So, we only need to calculate this entry. To do so, use the following recursive formula.

# Dynamic Programming Algorithm:

$$\text{Profit}[i][j] = \begin{cases} 0 & \text{if } i = 0 & (1) \\ 0 & \text{if } j = 0 & (2) \\ \text{Profit}[i-1][j] & \text{if } i \neq 0, j \neq 0 \text{ and } w[i] > j & (3) \\ \max \left\{ \underbrace{\text{Profit}[i-1][j]}_{(4a)}, \underbrace{p[i] + \text{Profit}[i-1][j-w[i]]}_{(4b)} \right\} & \text{if } i \neq 0, j \neq 0 \text{ and } w[i] \leq j & (4) \end{cases}$$

Explanation:

(1)  $i = 0$ : we cannot put any item into the knapsack. So, the max profit is 0.

(2)  $j = 0$ : the capacity of the knapsack is 0. Thus we cannot put any item into it. So the max profit is again 0.

# Dynamic Programming Algorithm:

(3)  $w[i] > j$ : We are allowed to use  $\text{item}_1, \dots, \text{item}_{i-1}, \text{item}_i$ . However, since  $w[i] > j$ , we cannot put  $\text{item}_i$  into the knapsack (its weight exceeds the knapsack capacity). Thus, we can actually only choose from  $\text{item}_1, \dots, \text{item}_{i-1}$ . Therefore, the max profit is  $\text{Profit}[i-1][j]$ .

(4) There are two choices: Either we put  $\text{item}_i$  into the knapsack, or we don't.

(4a) Do not put  $\text{item}_i$  into the knapsack. Then the capacity of the knapsack remains the same ( $j$ ), and now we can only use  $\text{item}_1, \dots, \text{item}_{i-1}$ . So the max profit for this case is  $\text{Profit}[i-1][j]$ .

(4b) Put  $\text{item}_i$  into the knapsack. The remaining capacity is reduced by the weight of  $\text{item}_i$  so it becomes  $j - w[i]$ , and now we can only use  $\text{item}_1, \dots, \text{item}_{i-1}$ . On the other hand, since we do put  $\text{item}_i$  into the knapsack, we gain its profit  $p[i]$ . So the max profit for this case is:  $p[i] + \text{Profit}[i-1][j - w[i]]$ .

Because we do not know which of the cases (4a) and (4b) gives larger profit, we take the maximum of the two cases.

# Recursive Algorithm

RecKS(int  $i$ , int  $j$ )

- ❶ **if**  $((i = 0) \text{ or } (j = 0))$  return 0
- ❷ **else if**  $(w[i] > j)$  return RecKS( $i - 1, j$ )
- ❸ **else** return **max** { RecKS( $i - 1, j$ ),  $p[i] + \text{RecKS}(i - 1, j - w[i])$  }

- This algorithm is very slow. The reason is that it makes many repeated recursive calls. It takes exponential time.
- The problem shows the **Overlapping Sub-problem Property**. So we should use dynamic programming.

# Dynamic Programming algorithm

## Dynamic Programming algorithm

**Input:**  $p[1..n]$ ,  $w[1..n]$ ,  $K$

- ① **for** ( $j = 0; j \leq K$ )  $\text{Profit}[0][j] = 0;$
- ② **for** ( $i = 0; i \leq n$ )  $\text{Profit}[i][0] = 0;$
- ③ **for** ( $i = 1; i \leq n$ )
- ④     **for** ( $j = 1; j \leq K$ )
- ⑤         **if** ( $w[i] > j$ )  $\text{Profit}[i][j] = \text{Profit}[i - 1][j];$
- ⑥         **else**  $\text{Profit}[i][j] = \max (\text{Profit}[i - 1][j], p[i] + \text{Profit}[i - 1][j - w[i]]);$
- ⑦ **output**  $\text{Profit}[n][K];$

- We calculate the entries of Profit array row by row according to formula (1) - (4).
- When calculating an entry  $\text{Profit}[i][j]$ , it only depends on other entries **that have been calculated already**.
- Thus each entry needs  $O(1)$  time to calculate.
- Since there are  $(n + 1)(K + 1) = O(nK)$  entries in Profit array, the total run time of the algorithm is  $O(nK)$ .

# Construct Solution Set

- This algorithm only calculates the max profit. It doesn't tell us the subset to be put into the knapsack.
- In order to do so, we need to keep additional information.
- We need another 2D array  $\text{Dir}[1..n][1..K]$ . The definition of  $\text{Dir}[j][k]$  is given below and its calculation should be included in the lines (5) and (6) in the above algorithm.

$$\text{Dir}[i][j] = \begin{cases} 1 & \text{if Profit}[i][j] \text{ is set to Profit}[i-1][j]. \\ & \text{(It gets its value from above, i.e. from (3) or (4a).)} \\ 2 & \text{if Profit}[i][j] \text{ is set to } p[i] + \text{Profit}[i-1][j-w[i]]. \\ & \text{(It gets its value from upper left, i.e. from (4b).)} \end{cases}$$

- After calculating the arrays  $\text{Profit}[*][*]$  and  $\text{Dir}[*][*]$ , the following code segment will print out items (in the reverse order) that should be included into the knapsack in order to achieve the max profit.

# Construct Solution Set

## Printout Items:

- 1  $j = K;$
  - 2 **for** ( $i = n$  **to** 1 **by**  $-1$ )
  - 3     **if** ( $\text{Dir}[i][j] = 2$ ) {  $j = j - w[i];$  and **print out**  $\text{item}_i;$ }
- $j$  keeps the remaining knapsack capacity, initialized to  $K$ .
  - Start at the lower right corner  $\text{Dir}[n][k]$ .
  - $\text{Dir}[i][j] = 2$ :  $\text{Profit}[i][j]$  gets its value from (4b) which means we put the  $\text{item}_i$  into the knapsack. So we print out this item, and reduce the capacity by its weight  $w[i]$  (the line  $j = j - w[i]$ ).
  - $\text{Dir}[i][j] = 1$ :  $\text{Profit}[i][j]$  gets its value from formula (3) or (4a). In either case,  $\text{item}_i$  is not included in the knapsack, so we do nothing.



# Example

## Example

Input:  $K = 13$ ,  $n = 5$ ,  $W[1] = 5$ ,  $W[2] = 4$ ,  $W[3] = 3$ ,  $W[4] = 2$ ,  $W[5] = 4$ ,  $P[1] = 4$ ,  $P[2] = 2$ ,  $P[3] = 4$ ,  $P[4] = 1$ ,  $P[5] = 5$

Profit array:		$j = 0$	1	2	3	4	5	6	7	8	9	10	11	12	13
	$i = 0$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	4	4	4	4	4	4	4	4	4
	2	0	0	0	0	2	4	4	4	4	6	6	6	6	6
	3	0	0	0	4	4	4	4	6	8	8	8	8	10	10
	4	0	0	1	4	4	5	5	6	8	8	9	9	10	10
	5	0	0	1	4	5	5	6	9	9	10	10	11	13	13
Dir array:		$j = 0$	1	2	3	4	5	6	7	8	9	10	11	12	13
	$i = 0$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	2	2	2	2	2	2	2	2	2
	2	0	1	1	1	2	1	1	1	2	2	2	2	2	2
	3	0	1	1	2	2	1	1	2	2	2	2	2	2	2
	4	0	1	2	1	1	2	2	1	1	1	2	2	1	1
	5	0	1	1	1	2	1	2	2	2	2	2	2	2	2

Items in the knapsack ("\*" means the item is NOT in the knapsack).

Item	1	2	3	4	5
Weight	5	*	3	*	4
Profit	4	*	4	*	5

When printing out the items in the knapsack, we set  $j = K = 13$  and start at the lower-right entry  $\text{Dir}[5][13]$ . Since this entry is 2, item<sub>5</sub> is in the knapsack. So we reduce  $j$  to  $j - w[5] = 13 - 4 = 9$ . Then we look at the entry  $\text{Dir}[4][9]$  which is 1. So item<sub>4</sub> is not in the knapsack and  $j$  remains 9. Next we look at  $\text{Dir}[3][9]$ , and so on. (The red entries are visited by the printing algorithm.)

# Longest Common Subsequence (LCS) Problem

- Let  $\Sigma$  be an **alphabet set**. (Ex:  $\Sigma = \{a, b, \dots, z\}$ )
- $X = \langle x_1, x_2, \dots, x_m \rangle$  is a **sequence** over  $\Sigma$  (i.e. each  $x_i \in \Sigma$ .)
- $|X| = m$  denotes **the length of  $X$** .  $X[i] = x_i$  denotes the  $i$ th letter.
- $Z = \langle z_1, z_2, \dots, z_k \rangle$  is another **sequence** of  $\Sigma$ .
- We say “ $Z$  is a **subsequence** of  $X$ ” if  $Z$  can be obtained by deleting some letters from  $X$ .

## Example

$Z = \langle BCDB \rangle$  is a subsequence of  $X = \langle \underline{D}BC\underline{B}D\underline{C}\underline{B} \rangle$

# Longest Common Subsequence (LCS) Problem

## Definition

Let  $X$ ,  $Y$  and  $Z$  be three sequences. If  $Z$  is a subsequence of both  $X$  and  $Y$ , we say “ $Z$  is a common subsequence of  $X$  and  $Y$ ”.

## Longest Common Subsequence (LCS) Problem

Input: Given  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ .

Find: a common subsequence  $Z$  of  $X$  and  $Y$  with maximum length.

## Brute Force Approach

- Enumerate all subsequences of  $Y$ . (There are  $2^n$  of them.)
- Check each of them to see if it is a subsequence of  $X$ .
- Pick a longest one.

This takes  $\Omega(2^n)$  time.

# Dynamic Programming Algorithm

## Definition

A **prefix** of the sequence  $X = \langle x_1 \dots x_m \rangle$  is  $X_i = \langle x_1 \dots x_i \rangle$  ( $1 \leq i \leq m$ )

## Theorem

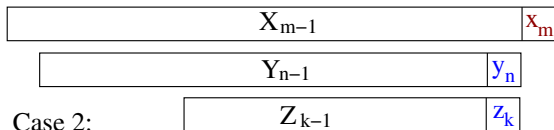
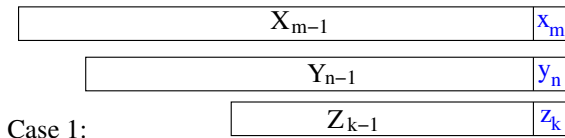
### Optimal Substructure Property of LCS

Let  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ . Let  $Z = \langle z_1 \dots z_k \rangle$  be a LCS of  $X$  and  $Y$ .

- 1 If  $x_m = y_n$ , then  $z_k = x_m = y_n$  and  $Z_{k-1}$  is a LCS of  $X_{m-1}$  and  $Y_{n-1}$ .
- 2 If  $x_m \neq y_n$ , then  $z_k \neq x_m$  implies  $Z$  is a LCS of  $X_{m-1}$  and  $Y$ .
- 3 If  $x_m \neq y_n$ , then  $z_k \neq y_n$  implies  $Z$  is a LCS of  $X$  and  $Y_{n-1}$ .

This theorem says: an LCS ( $Z$ ) of  $X$  and  $Y$  contains within it an LCS of the prefixes of  $X$  and  $Y$ .

# Recursive Formulation



Case 3 is similar.

# Recursive Formulation

Define a 2D array  $c[0..m, 0..n]$ , where  $c[i, j]$  is defined to be the **length of the LCS of  $X_i$  and  $Y_j$** . Then:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } x_i = y_j \\ \max\{c[i, j - 1], c[i - 1, j]\} & \text{if } x_i \neq y_j \end{cases}$$

- If  $i = 0$ ,  $X_i = \emptyset$ . The  $\text{LCS}(\emptyset, Y_j) = \emptyset$ . Its length is 0.
- If  $j = 0$ ,  $Y_j = \emptyset$ . The  $\text{LCS}(X_i, \emptyset) = \emptyset$ . Its length is 0.
- if  $x_i = y_j$ , then we find the length of  $\text{LCS}(X_{i-1}, Y_{j-1})$ . This is  $c[i - 1, j - 1]$ . Since  $x_i = y_j$  match, this is the last letter in LCS, so we add 1.
- If  $x_i \neq y_j$ , we find  $\text{LCS}(X_i, Y_{j-1})$  (the length is  $c[i, j - 1]$ ) and  $\text{LCS}(X_{i-1}, Y_j)$  (the length is  $c[i - 1, j]$ ) and pick the longer one.

# Dynamic Programming Algorithm

## DynPro-LCS

- Fill the 2D array  $c[*,*]$  row by row according to the recursive formula.
  - Output  $c[m,n]$ .
- 
- There are  $(m+1) \cdot (n+1) = \Theta(nm)$  entries in  $c[*,*]$ .
  - When we calculate  $c[i,j]$ , we may need the values  $c[i-1, j-1]$ ,  $c[i-1, j]$  and  $c[i, j-1]$ . They have been computed already. So each entry takes  $O(1)$  time.
  - So the algorithm takes  $\Theta(nm)$  time.
  - This algorithm only computes the length of the LCS, not the actual LCS. To do so, we need to keep more information.
  - Define an array  $b[1..m, 1..n]$  where  $b[i,j] = \uparrow, \leftarrow, \text{ or } \nwarrow$ , pointing to the direction where  $c[i,j]$  gets its value.

# Dynamic Programming Algorithm

$X = \langle ABCBDAB \rangle$  and  $Y = \langle BDCABA \rangle$

$i$	$j$	0	1	2	3	4	5	6
		$y_j$	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

To construct LCS (in reverse order)

- Starts at  $b[m, n]$ , follow the arrows.
- For a ↖, the corresponding letter is in LCS.
- For a ← or a ↑, do nothing.
- Stop when reaching the first row or column.
- In the above, blue indicates the LCS, and the path for constructing it.



# Summary on Using Dynamic Programming Algorithm

- Step 1: Analyze the structure of the problem, **prove or convince yourself** it has the **Optimal Substructure Property**.
- Step 2: Derive a recursive formulation of the problem. Namely, express the solution of the bigger problem in terms of the subproblems.
- Note: In most cases, we want to **construct an optimal solution**. However, it is often easier to concentrate on the **value** of the optimal solution. Once we have an alg. for computing the **value**, it's pretty easy to **construct it**.  
**All we have to do is to memorize how the optimal value is obtained.**
- Step 3: Write a recursive procedure according to the recursive formulation obtained in Step 2. Draw the recursion tree for a few levels. If the algorithm is making recursive calls to solve **overlapping subproblems, or repeatedly solving the same subproblems**, then you should use dynamic programming (i.e. bottom-up approach).
- Write a bottom up alg for solving subproblems. **Pay attention to the order: When solving a subproblem, the solution of other subproblems needed by it have been obtained already.**