

# Dynamic Programming

**Joy Mukherjee**

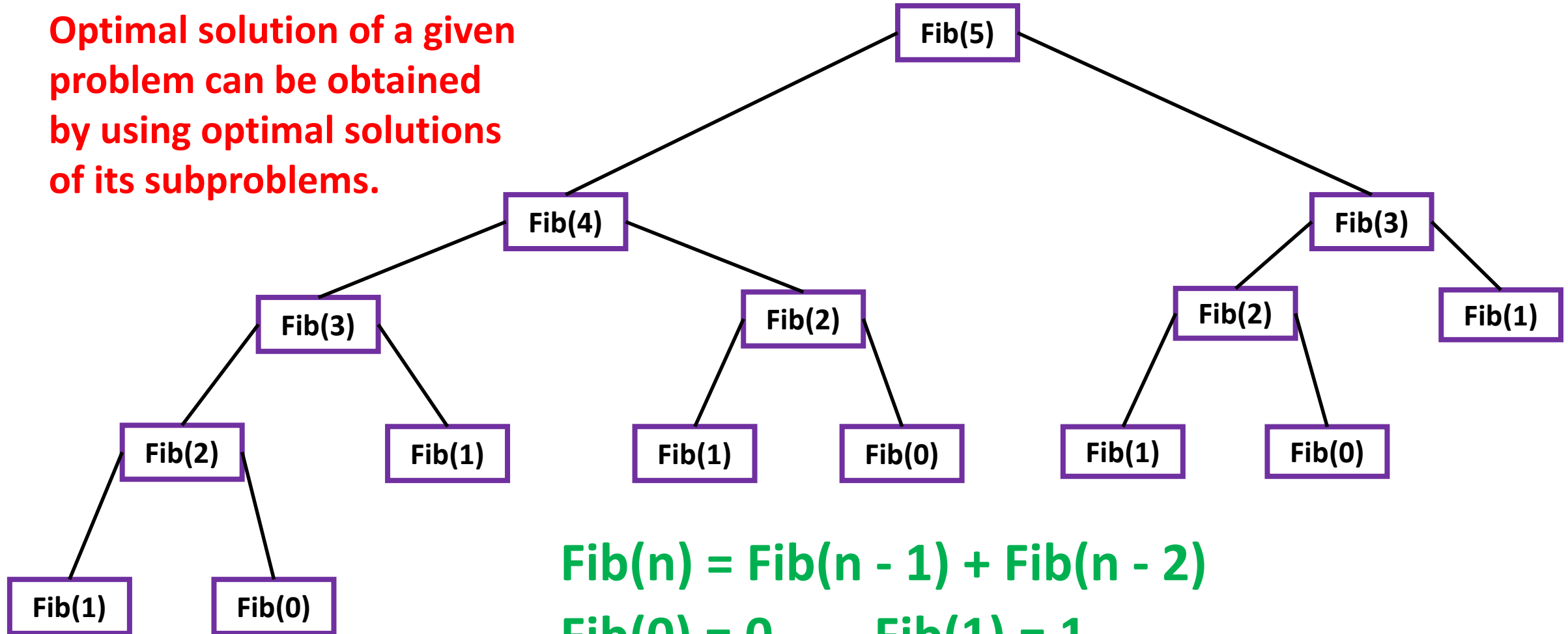
# n-th Fibonacci Number: Recursion

0 1 1 2 3 5 8 13 21 34

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

# Optimal Substructure

Optimal solution of a given problem can be obtained by using optimal solutions of its subproblems.

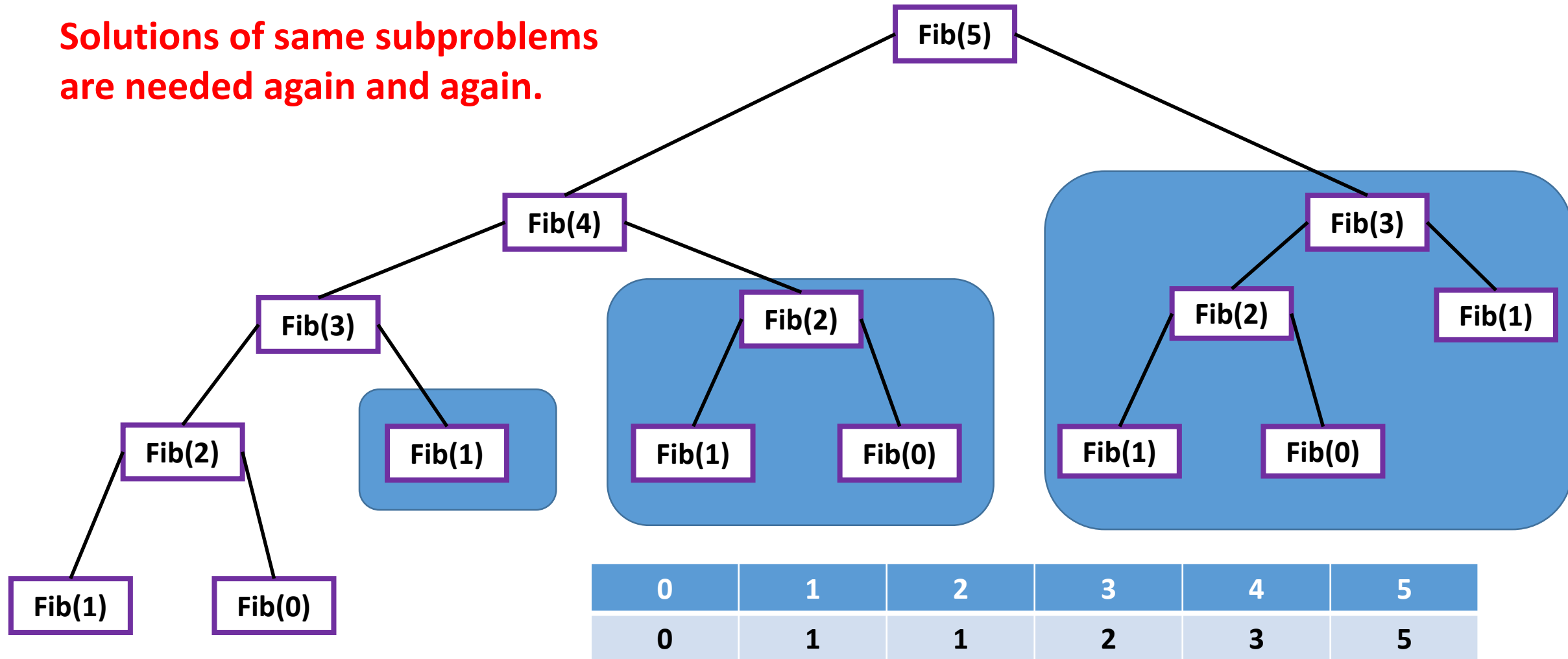


$$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$$

$$\text{Fib}(0) = 0 \quad \text{Fib}(1) = 1$$

# Overlapping Subproblems

Solutions of same subproblems  
are needed again and again.



# Dynamic Programming

- Break a problem into subproblems:
  - Find a recurrence relation with base cases (Optimal Substructure Property)
- Stores the results of subproblems in a table
  - Avoid computing the same subproblem again. (Overlapping Subproblem Property)
  - Size of the table/array can be determined from the recursion tree.
- How to populate the table?
  - **Top-down approach**: Memoization (Recursive)
  - **Bottom-up approach**: Tabulation (Iterative)

# Memoization: A Top-Down Approach

1. Recursion with lookup table.
2. Initialize a lookup table with all initial values as -1
3. If (lookup = -1)
  - a. Calculate the value and put the result in the lookup table
4. Else
  - a. Return precomputed value

```
int fib(int n)
{
    if (T[n] == -1) {
        if (n <= 1)
            T[n] = n;
        else
            T[n] = fib(n-1) + fib(n-2);
    }
    return T[n];
}
```

# Tabulation: A Bottom-Up Approach

1. Iteration with lookup table.
2. Compute and store the result of the subproblems in the table in a bottom-up fashion.
3. Return last entry of the table

```
int fib(int n)
{
    int i, f[n+1];
    f[0] = 0; f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

# Dynamic Programming

- Recursion
- **Memoization**: Recursion + Table
  - It is better than Tabulation only when a subset of the subproblems are needed to solve the original problem.
- **Tabulation**: Table (typically used methodology to solve a DP)
  - It is better than Memoization only when all of the subproblems are needed to solve the original problem.



# Substring vs Subsequence

- A **subsequence** of a string is a sequence of characters in the string that maintains the same relative order as in the string **with one or more characters left out**.
- **Example:** “CBCDB” is a subsequence in “ACBCCDAB”
- A **substring** of a string is a sequence of **contiguous** characters in the string .
- **Example:** “CBCCD” is a substring in “ACBCCDAB”
- A substring is a subsequence, but a subsequence may not be a substring.

# Longest Common Subsequence (LCS)

- **Input:**

X = "ACBCCDAB"

Y = "DABCADB"

- **Output:** Length of LCS

5 (Z = "ABCDB" or Z = "ABCAB")

# Length of LCS: Optimal Substructure

Input:  $x[] = \text{"ACBCCDAB"}$        $y[] = \text{"DABCADB"}$

Since  $x[7] = y[6]$ , then

$$\text{LCS}(\text{"ACBCCDAB"}, \text{"DABCADB"}) = 1 + \text{LCS}(\text{"ACBCCDA"}, \text{"DABCAD"})$$

Input:  $x[] = \text{"ACBCCDAB"}$        $y[] = \text{"DABCADA"}$

Since  $x[7] \neq y[6]$ , then

$$\begin{aligned} \text{LCS}(\text{"ACBCCDAB"}, \text{"DABCADA"}) = \text{MAX}\{ \\ \text{LCS}(\text{"ACBCCDAB"}, \text{"DABCAD"}), \text{LCS}(\text{"ACBCCDA"}, \text{"DABCADA"}) \} \end{aligned}$$

# Length of LCS: Optimal Substructure

**Input:**  $X_m = x_0, x_1, \dots, x_{m-1}$

$Y_n = y_0, y_1, \dots, y_{n-1}$

1. If  $x[m-1] = y[n-1]$ , then

$$\text{LCS} ( X_m, Y_n ) = 1 + \text{LCS} ( X_{m-1}, Y_{n-1} )$$

2. If  $x[m-1] \neq y[n-1]$ ,

$$\begin{aligned} \text{LCS} ( X_m, Y_n ) = \text{MAX}\{ & \text{LCS} ( X_{m-1}, Y_n ), \\ & \text{LCS} ( X_m, Y_{n-1} ) \} \end{aligned}$$

**Important Observation:**

$\text{LCS} ( X_m, Y_n )$  depends on

Either  $\text{LCS} ( X_{m-1}, Y_{n-1} )$

Or  $\text{LCS} ( X_{m-1}, Y_n )$  and  $\text{LCS} ( X_m, Y_{n-1} )$

# Length of LCS ( $X[m]$ , $Y[n]$ )

L	Y=	A	C	B	A	B
X=		0	0	0	0	0
C	0	0	1	1	1	1
C	0	0	1	1	1	1
A	0	1	1	1	2	2
B	0	1	1	2	2	3

```
int L[m+1][n+1];  
L[1][2] = Length of  
LCS("C", "AC");  
L[4][5] = Length of  
LCS("CCAB", "ACBAB");
```

$L[i][j]$  = Length of LCS of string  $X[0..i-1]$  and string  $Y[0..j-1]$

# Length of LCS ( $X[m]$ , $Y[n]$ )

	Y=	A	C	B	A	B
X=	0	0	0	0	0	0
C	0	0	1	1	1	1
C	0	0	1	1	1	(2,5)1
A	0	1	1	1	(3,4)2	(3,5)2
B	0	1	1	(4,3)2	(4,4)2	(4,5)3

**Input:**  $X_m = x_0, x_1, \dots, x_{m-1}$

$Y_n = y_0, y_1, \dots, y_{n-1}$

1. If  $x_{m-1} = y_{n-1}$ , then

$$\text{LCS} ( X_m, Y_n ) = 1 + \text{LCS} ( X_{m-1}, Y_{n-1} )$$

2. If  $x_{m-1} \neq y_{n-1}$ ,

$$\text{LCS} ( X_m, Y_n ) = \text{MAX} \{ \text{LCS} ( X_{m-1}, Y_n ), \\ \text{LCS} ( X_m, Y_{n-1} ) \}$$

$L[i][j] = \text{LCS}(X, i, Y, j) = \text{Length of LCS of string } X[0..i] \text{ and string } Y[0..j]$

# LCS: Assignment

1. Write a Recursive program for longest common subsequence (LCS).
2. Write a Memoized DP for LCS.
3. Write a Tabulated DP for LCS.
4. Print an LCS.

# Length of LCS ( X[m], Y[n] ) (Tabulation)

```
int lcs(char *X, char *Y)
{
    int i, j, m = strlen(X), n = strlen(Y), L[m+1][n+1];
    /* L[i][j] is length of LCS of string X[0..i] and string Y[0..j] */
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)          L[i][j] = 0;
            else if (X[i-1] == Y[j-1])     L[i][j] = L[i-1][j-1] + 1;
            else if (L[i-1][j] > L[i][j-1]) L[i][j] = L[i-1][j];
            else                            L[i][j] = L[i][j-1];
        }
    }
    return L[m][n];
}
```



# Longest Palindromic Subsequence (LPS)

$$\begin{aligned} & \text{LPS}(\text{"ACBCCDA"}) \\ &= 2 + \text{LPS}(\text{"CBCCD"}) \\ &= 2 + \max(\text{LPS}(\text{"CBCC"}), \text{LPS}(\text{"BCCD"}) ) \\ &= 2 + \max(2 + \text{LPS}(\text{"BC"}), \max(\text{LPS}(\text{"BCC"}), \text{LPS}(\text{"CCD"}) ) ) \\ &= 2 + \max(2 + 1, \max(\max(\text{LPS}(\text{"BC"}), \text{LPS}(\text{"CC"}) ) , \max(\text{LPS}(\text{"CC"}), \text{LPS}(\text{"CD"}) ) )) \\ &= 2 + \max(3, 2) = 5 \end{aligned}$$

- **Output:** Length of LPS

5 (Z = "ACBCA" or Z = "ACCCA")

# Length of LPS: Optimal Substructure

**Input:**  $x[8] = \text{"ACBCCDBA"}$

Since  $x[0] = x[7]$ , then

$$\text{LPS}(x, 0, 7) = 2 + \text{LPS}(x, 1, 6)$$

$$\text{LPS}(\text{"ACBCCDBA"}) = 2 + \text{LPS}(\text{"CBCCDB"})$$

**Input:**  $x[8] = \text{"ACBCCDAB"}$

Since  $x[0] \neq x[7]$ , then

$$\text{LPS}(x, 0, 7) = \text{MAX}\{ \text{LPS}(x, 0, 6), \text{LPS}(x, 1, 7) \}$$

$$\text{LPS}(\text{"ACBCCDAB"}) = \text{MAX}\{ \text{LPS}(\text{"ACBCCDA"}), \text{LPS}(\text{"CBCCDAB"}) \}$$

# Length of LPS: Optimal Substructure

**Input:**  $X_m = x_0, x_1, \dots, x_{m-1}$

**Output:**  $Z, 0, k-1 = \text{LPS} ( X, 0, m-1 )$

1. If  $x_0 = x_{m-1}$ , then  $z_0 = x_0$  and  $z_{k-1} = x_{m-1}$

$Z, 1, k-2 = \text{LPS} ( X, 1, m-2 )$

1. If  $x_0 \neq x_{m-1}$ , then

$Z, 0, k = \text{MAX}\{ \text{LPS} ( X, 0, m-2 ),$   
 $\text{LPS} ( X, 1, m-1 ) \}$

**Important Observation:**

$\text{LPS} ( X, i, j ) = \text{Length of LPS } X[i..j]$

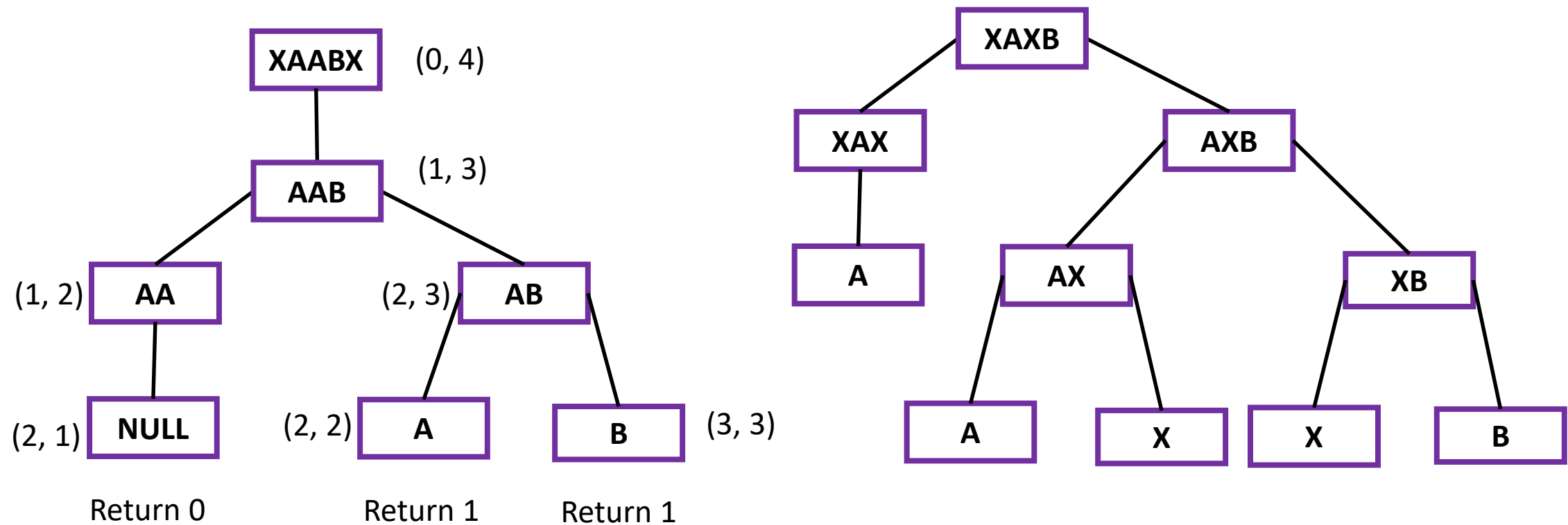
$\text{LPS} ( X, i, j )$  depends on

Either  $\text{LPS} ( X, i+1, j-1 )$  if  $X[i]=X[j]$

Or Maximum of

$\{ \text{LPS} ( X, i, j-1 ) \text{ and } \text{LPS} ( X, i+1, j ) \}$

# Optimal Substructure: LPS



# Length of LPS ( X[m] )

	A	C	B	B	A
A	0, 0	0, 1	0, 2	<b>0, 3</b>	<b>0, 4</b>
C		1, 1	1, 2	(1, 3)= max{(1,2),( 2,3)}	<b>1, 4</b>
B			2, 2	2, 3	<b>2, 4</b>
B				3, 3	3, 4
A					4, 4

	A	C	B	B	A
A	1	1	1	2	4
C	0	1	1	2	2
B		0	1	2	2
B			0	1	1
A				0	1

A	C	B	B	A
0	1	2	3	4

0	1	2	3
A	B	B	A

$$L[i][j] = \text{LPS}(X, i, j) = \text{Length of LPS } X[i..j]$$

# Reduction: Solving LPS using LCS

$$\text{LPS}(X) = \text{LCS}(X, X_R) = \text{LCS}(\text{ACBBA}, \text{ABBCA}) = \text{ABBA}$$

# LPS: Assignment

1. Write a Recursive program for longest palindromic subsequence (LPS).
2. Write a Memoized DP for LPS.
3. Write a Tabulated DP for LPS.
4. Print an LPS.

# Length of LPS ( X[m] ) (Tabulation)

```
int lps(char X[], int m) {
    int i, j, k, l, L[m][m];
    for(i = 0; i < m; i++) L[i][i] = 1;
    for(l = 1; l < m; l++) {
        for(i = 0; i < m-l; i++) {
            if(X[i] == X[i+l]) {
                if(i+1 <= i+l-1) L[i][i+l] = 2 + L[i+1][i+l-1];
                else L[i][i+l] = 2;
            } else {
                int a = L[i][i+l-1];
                int b = L[i+1][i+l];
                L[i][i+l] = (a > b)? a : b;
            }
        }
    }
    return L[0][m-1];
}
```



# Longest Increasing Subsequence (LIS)

- **Input:**

$X[10] = \{3, 2, 1, 3, 5, 4, 4, 5, 6, 3\}$

$X[10] = \{3, 2, 1, 3, 5, 4, 4, 5, 6, 3\}$

- **Output:** Length of LIS

5     $\{1, 3, 4, 5, 6\}$  or  $\{2, 3, 4, 5, 6\}$

# Length of LIS ( A[n] )

<b>A</b>	3	2	1	3	5	4	4	5	6	3
<b>i</b>	0	1	2	3	4	5	6	7	8	9

**Length[i] = Length of LIS that ends with A[i]**

**Previous[i] = Previous index of LIS that ends with A[i]**

**Optimal Substructure:**

$\text{Length}[i] = \text{MAX} (0, \text{Length}[k]) + 1, \text{ where } k < i \text{ AND } A[i] > A[k]$

**Objective:**

$\text{Max}(\text{Length}(i)) \text{ for all } i = 0 \text{ to } n-1$

# Initialization

**A**

<b>3</b>	<b>2</b>	<b>1</b>	<b>3</b>	<b>5</b>	<b>4</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>3</b>
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>

## Initialization

**Length**

<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>

**Previous**

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

i = 1

No update:  $A[1] < A[0]$

Length	1	1	1	1	1	1	1	1	1	1
Previous	0	1	2	3	4	5	6	7	8	9

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

i = 2

No update:  $A[2] < A[1]$  AND  $A[2] < A[0]$

Length	1	1	1	1	1	1	1	1	1	1
Previous	0	1	2	3	4	5	6	7	8	9

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

i = 3

Update:  $A[2] < A[3]$

Length	1	1	1	2	1	1	1	1	1	1
Previous	0	1	2	2	4	5	6	7	8	9

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

i = 4

Update: A[3] < A[4]

Length	1	1	1	2	3	1	1	1	1	1
Previous	0	1	2	2	3	5	6	7	8	9

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

**i = 5**

**No Update:**  $A[4] > A[5]$   $j = 4$

**Update:**  $A[3] < A[5]$   $j = 3$  (If get a longer subsequence)

Length	1	1	1	2	3	3	1	1	1	1
Previous	0	1	2	2	3	3	6	7	8	9

Since  $i = 5$ , the variable  $j$  iterates from  $j = 4$  to  $j = 0$ , and check whether  $A[j] < A[i]$  and  $\text{Length}[j] + 1 > \text{Length}[i]$ . If so then update  $\text{Length}[i] = \text{Length}[j] + 1$  and  $\text{Previous}[i] = j$



# Execution: LIS

**A**

3	2	1	3	5	4	4	5	6	3
0	1	2	3	4	5	6	7	8	9

**i = 6**

**Update: A[3] < A[6]**

<b>Length</b>	1	1	1	2	3	3	3	1	1	1
<b>Previous</b>	0	1	2	2	3	3	3	7	8	9

# Execution: LIS

A	3	2	1	3	5	4	4	5	6	3
	0	1	2	3	4	5	6	7	8	9

**i = 7**

**Update:  $A[6] < A[7]$**

Length	1	1	1	2	3	3	3	4	1	1
Previous	0	1	2	2	3	3	3	6	8	9

Since  $i = 7$ , the variable  $j$  iterates from  $j = 6$  to  $j = 0$ , and check whether  $A[j] < A[i]$  and  $\text{Length}[j] + 1 > \text{Length}[i]$ . If so then update  $\text{Length}[i] = \text{Length}[j] + 1$  and  $\text{Previous}[i] = j$

# Execution: LIS

**A**

3	2	1	3	5	4	4	5	6	3
0	1	2	3	4	5	6	7	8	9

**i = 8**

**Update: A[7] < A[8]**

<b>Length</b>	1	1	1	2	3	3	3	4	5	1
<b>Previous</b>	0	1	2	2	3	3	3	6	7	9

# Execution: LIS

**A**

3	2	1	3	5	4	4	5	6	3
0	1	2	3	4	5	6	7	8	9

**i = 9**

**Update:**  $A[2] < A[9]$

<b>Length</b>	1	1	1	2	3	3	3	4	5	2
<b>Previous</b>	0	1	2	2	3	3	3	6	7	2

# LIS

A

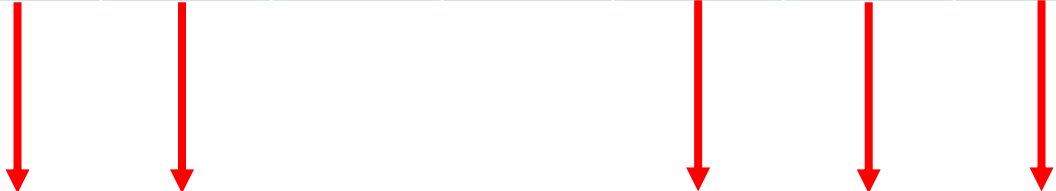
3	2	1	3	5	4	4	5	6	3
0	1	2	3	4	5	6	7	8	9

Length

1	1	1	2	3	3	3	4	5	2
---	---	---	---	---	---	---	---	---	---

Previous

0	1	2	2	3	3	3	6	7	2
---	---	---	---	---	---	---	---	---	---



# Algorithm: LIS (Tabulation)

```
For each i = 1 to n-1 {  
    For each j = i-1 to 0 {  
        if( A[i] > A[j] AND  
            length[j] + 1 > length[i]) {  
            length[i] = length[j] + 1;  
            previous[i] = j;  
            if(length[i] > maximum) {  
                maximum = length[i];  
                LastIndexLIS = i;  
            }  
        }  
    }  
}
```

```
printLIS(int A[], int previous[], int i)  
{  
    if(previous[ i ] != i)  
        printLIS(A, previous, previous[ i ]);  
    print(A[ i ]);  
}
```