Chapter 4:
Dynamic Programming

Outline

- Introduction
- Matrix chain multiplication method
- Longest common subsequence

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems
- However, it applies when the subproblems overlap, i.e. when subproblems share subsubproblems
- It solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Dynamic programming

Two ways to implement dynamic programming:

1. Memoization - Top down approach
   a. Maintain a map of already solved sub problems

2. Tabulation - Bottom up approach
   a. Solve all related sub-problems first
   b. Based on the results in the table, compute the solution to the "top" / original problem

Fibonacci numbers

Recall definition of Fibonacci numbers:

```
F(n) = F(n-1) + F(n-2)

F(0) = 0

F(1) = 1
```

The beginning of the sequence is thus:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```

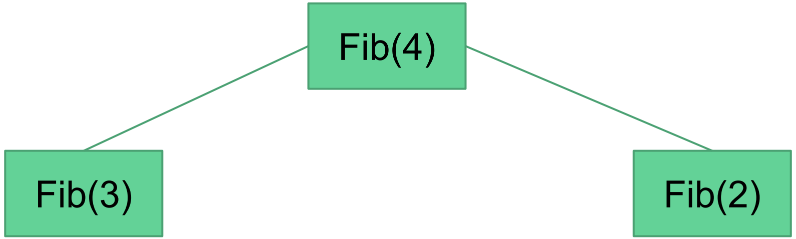
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```

Fib(4)

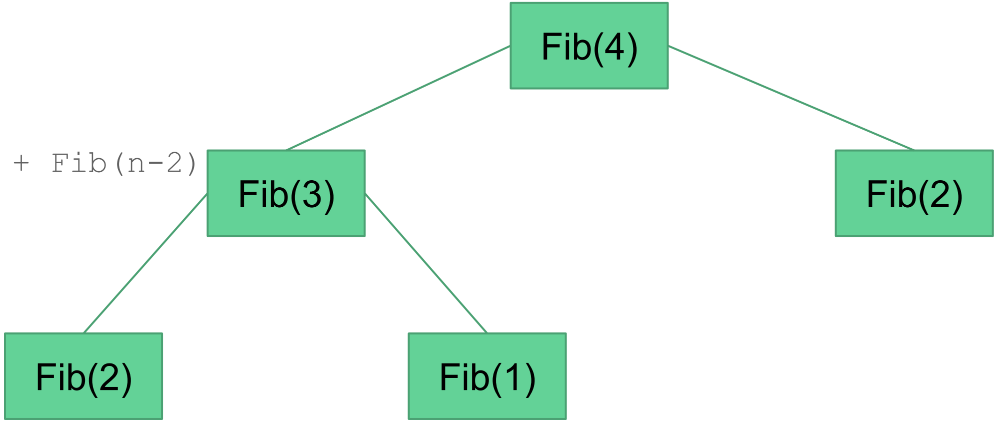
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



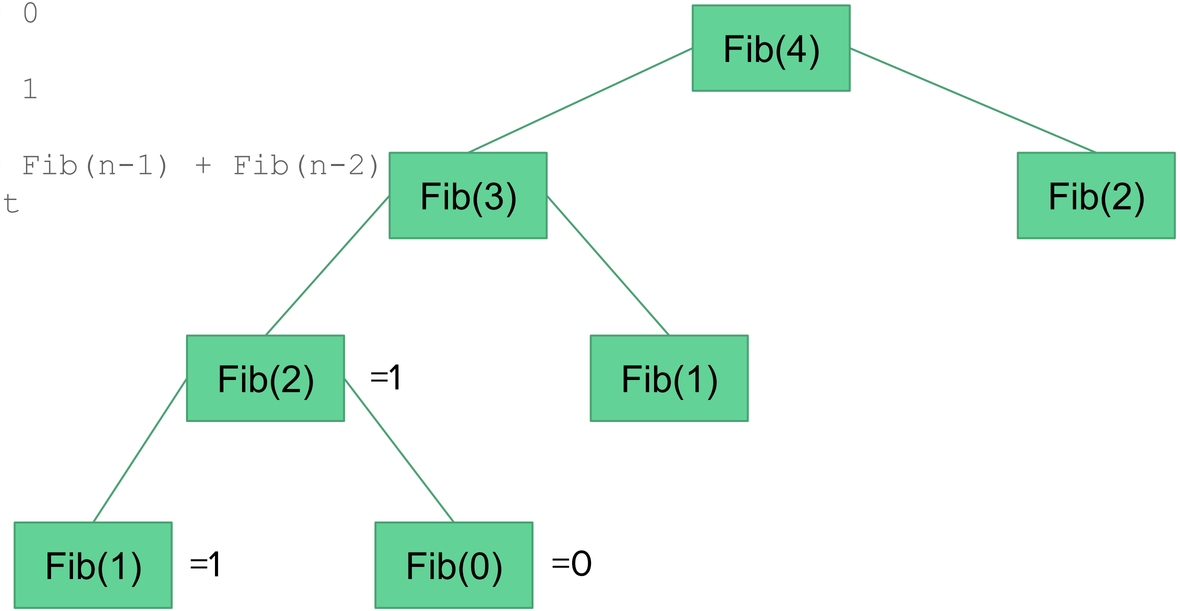
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



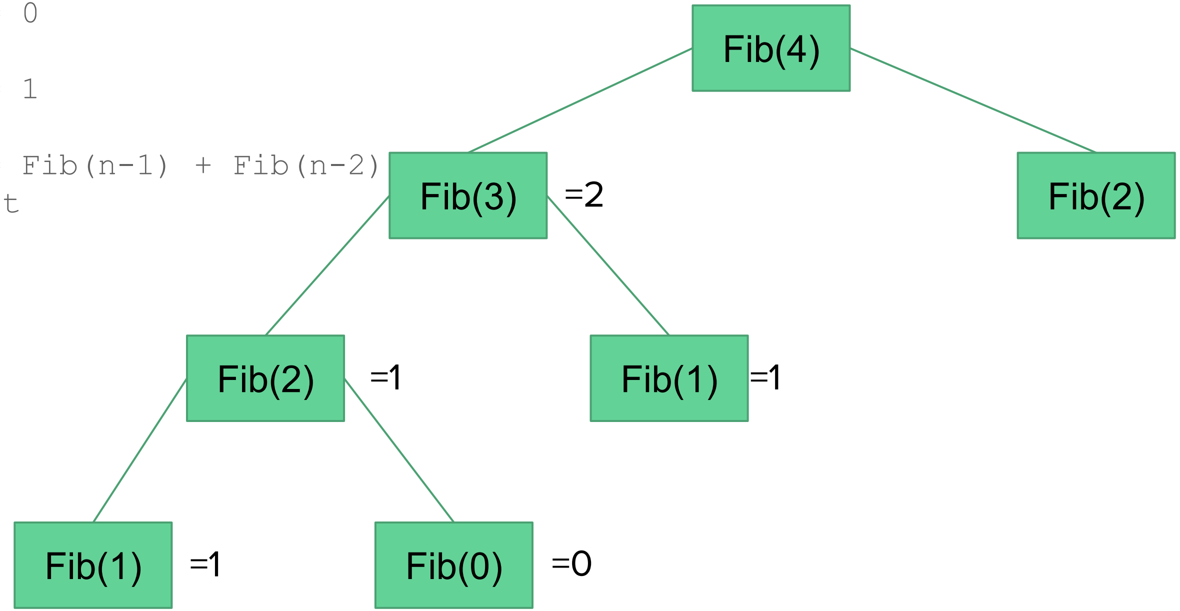
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



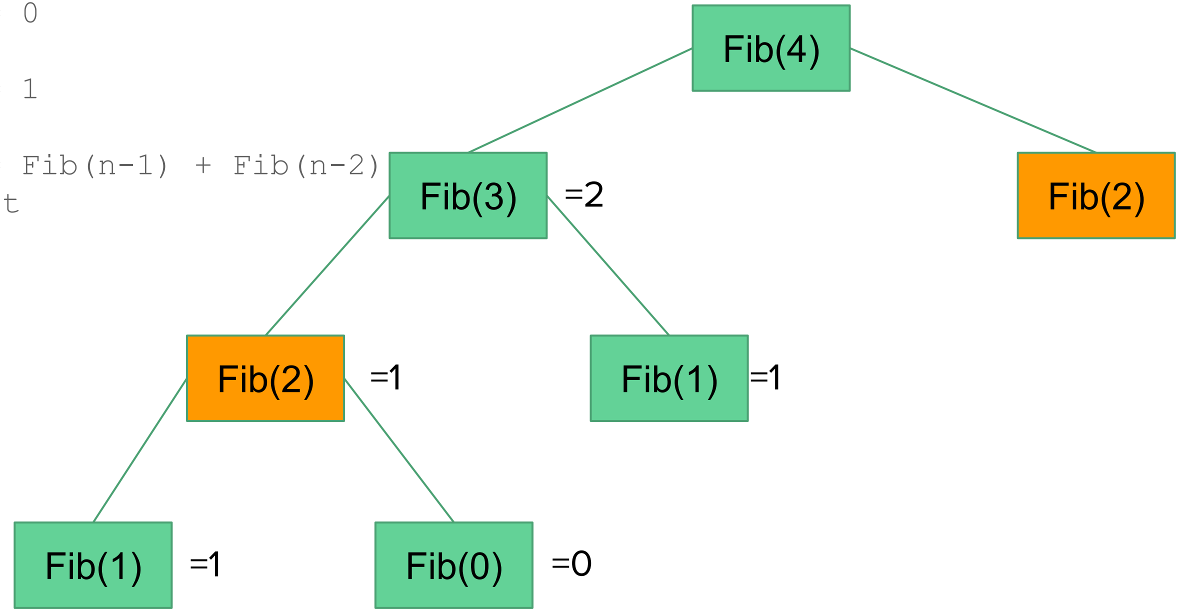
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



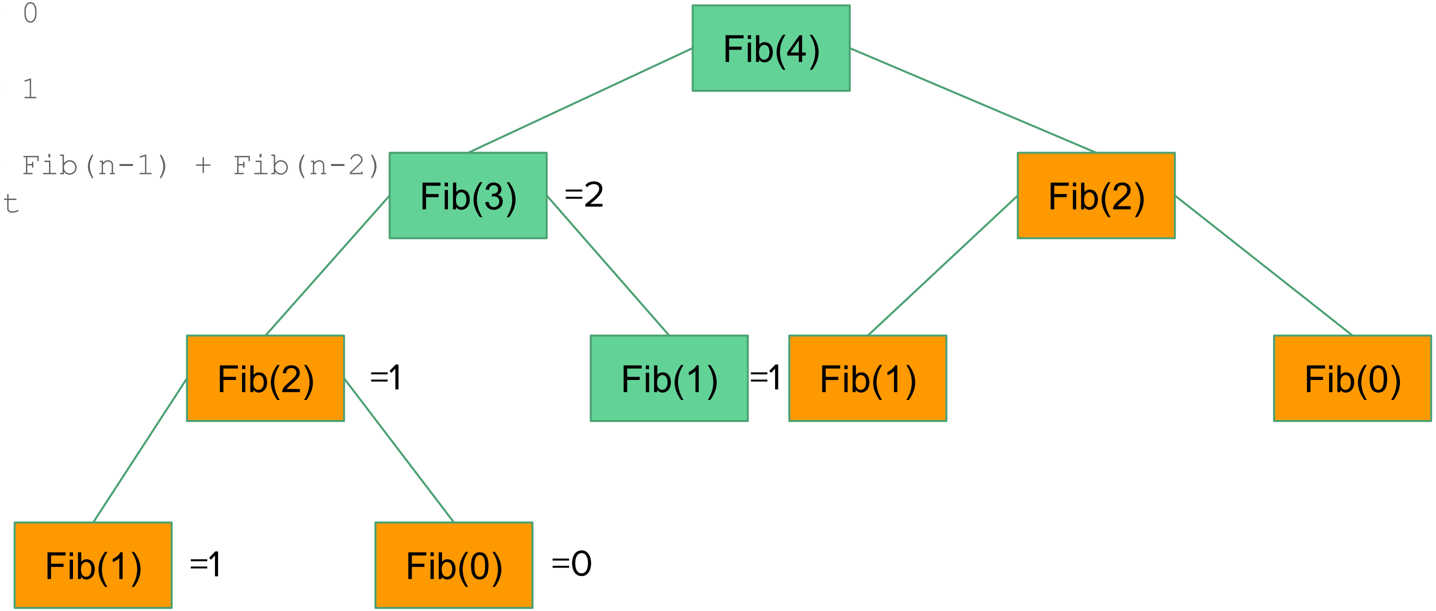
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



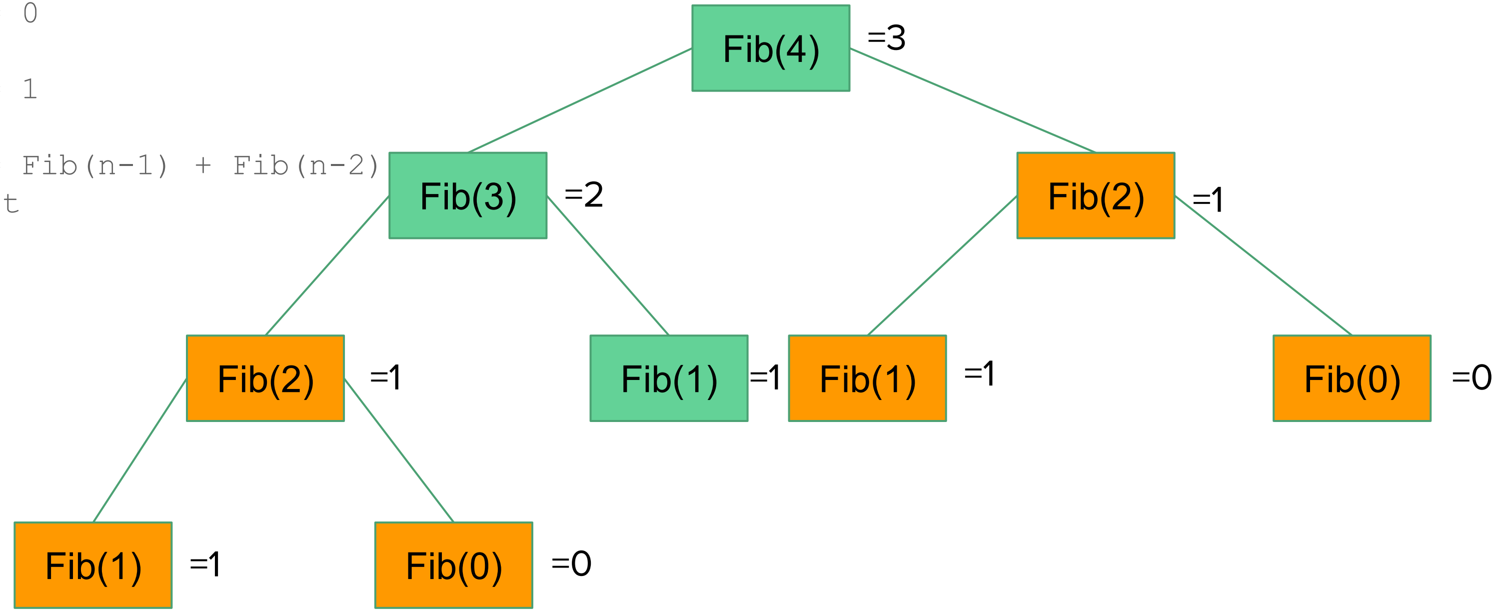
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



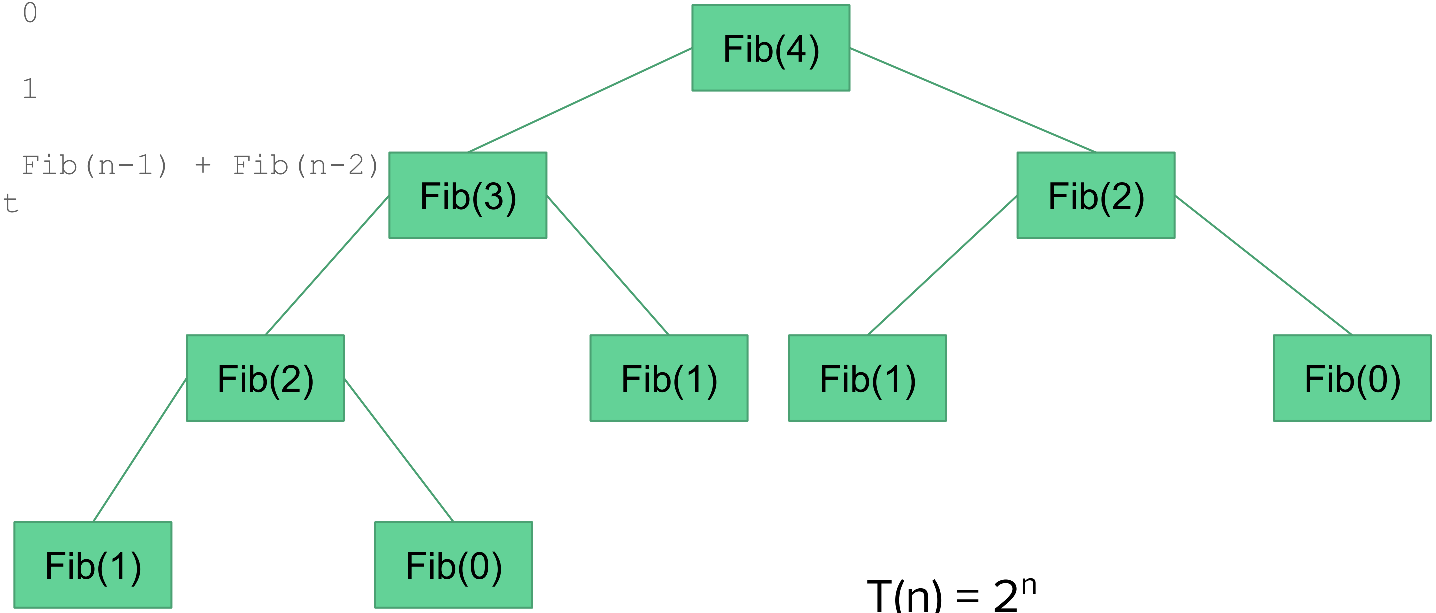
Fibonacci numbers: Recursion

```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```



Fibonacci numbers: Recursion

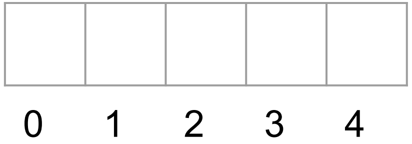
```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```

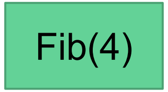


$T(n) = 2^n$

Fibonacci numbers: Memoization

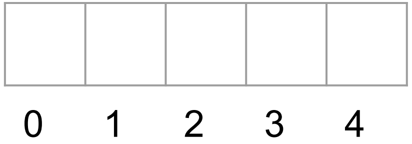
Storing the results of subsubproblems so as to avoid recomputing them

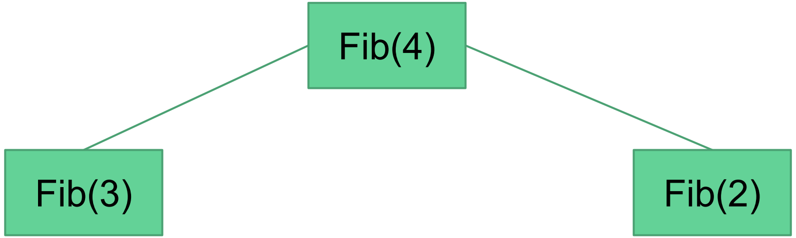




Fibonacci numbers: Memoization

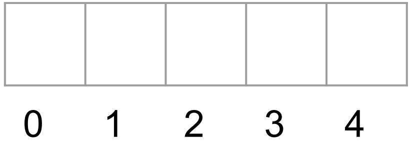
Storing the results of subsubproblems so as to avoid recomputing them

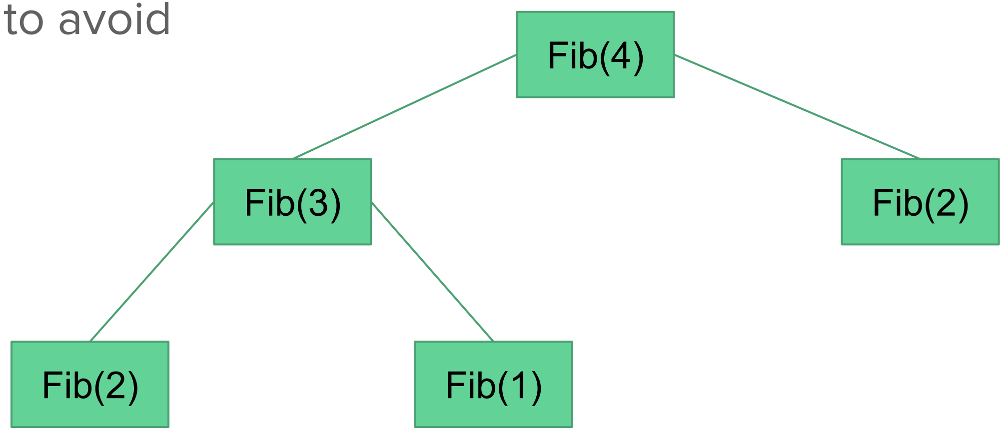




Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

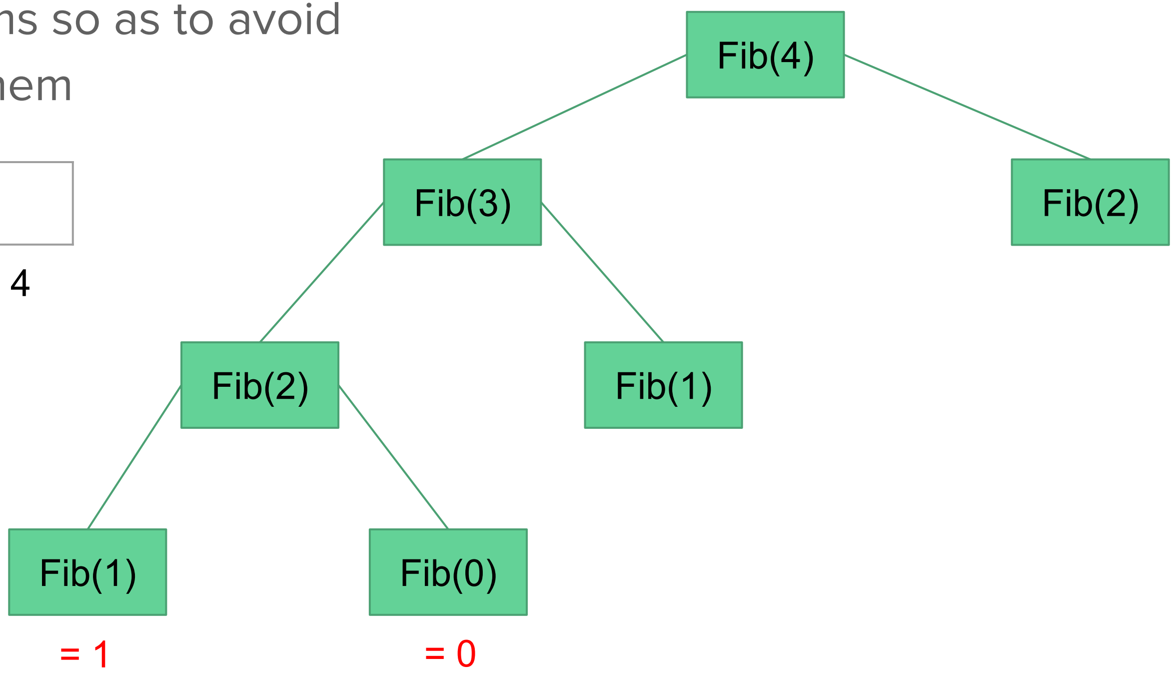




Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

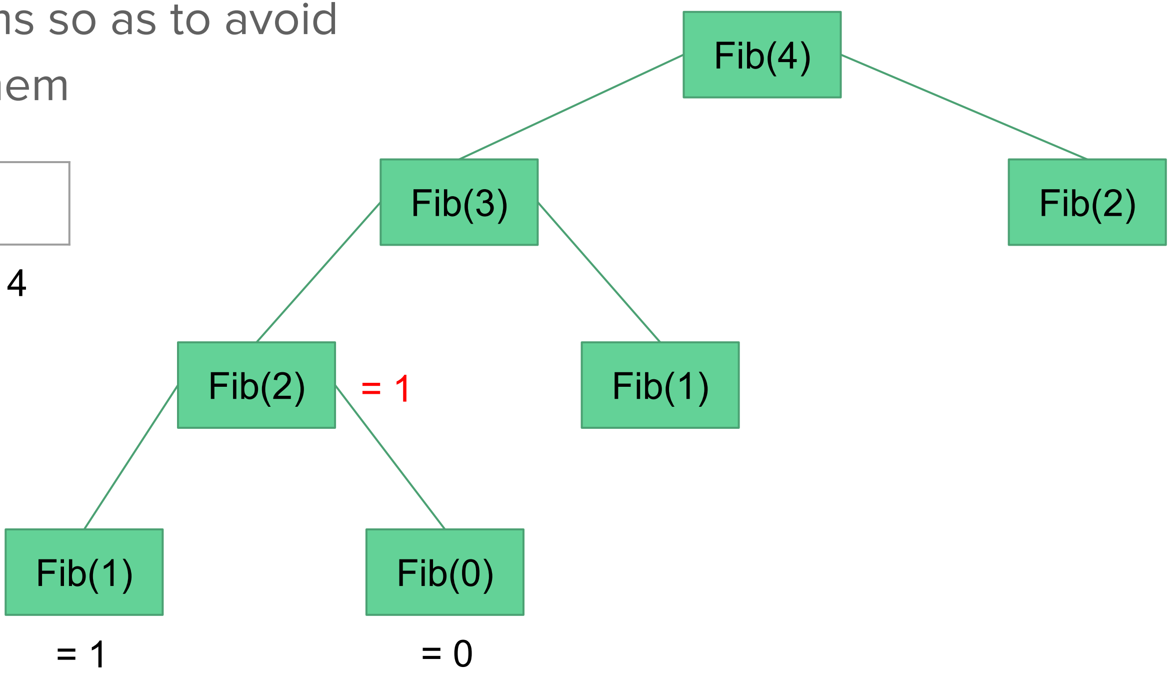
0	1			
0	1	2	3	4



Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

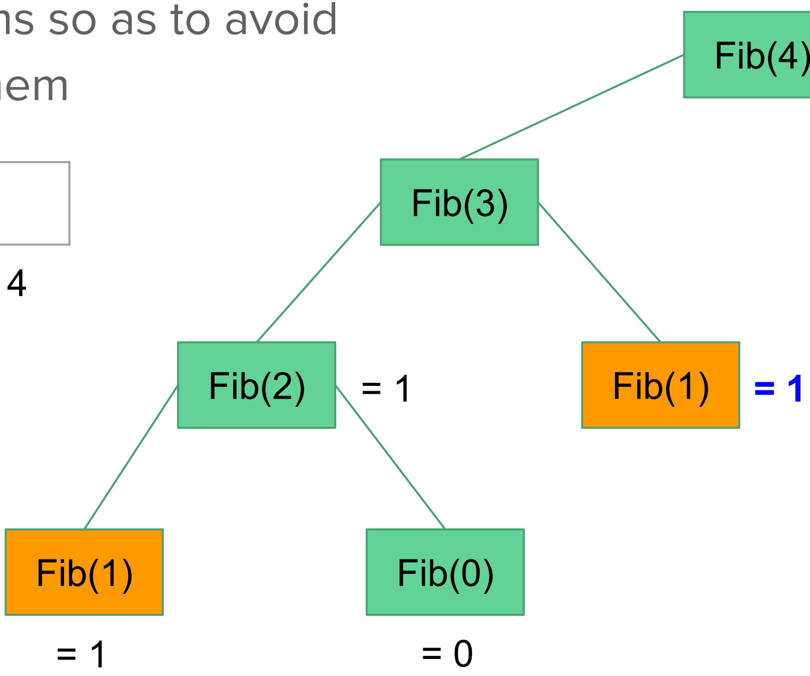
0	1	1		
0	1	2	3	4



Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

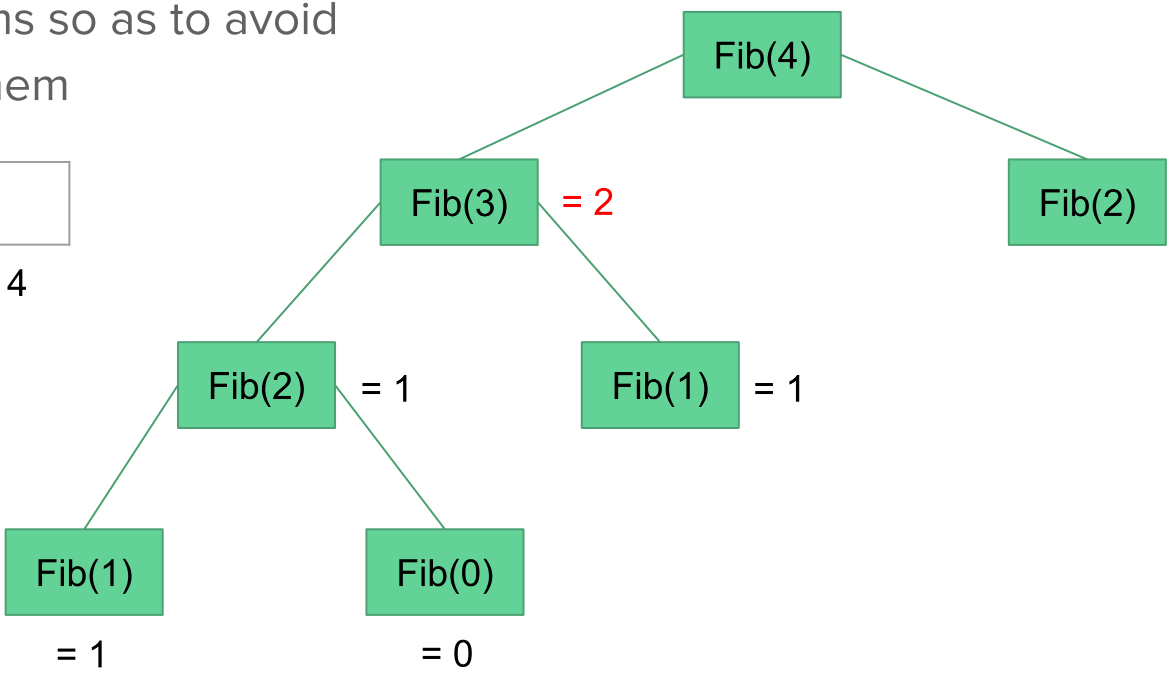
0	**1**	1		
0	1	2	3	4



Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

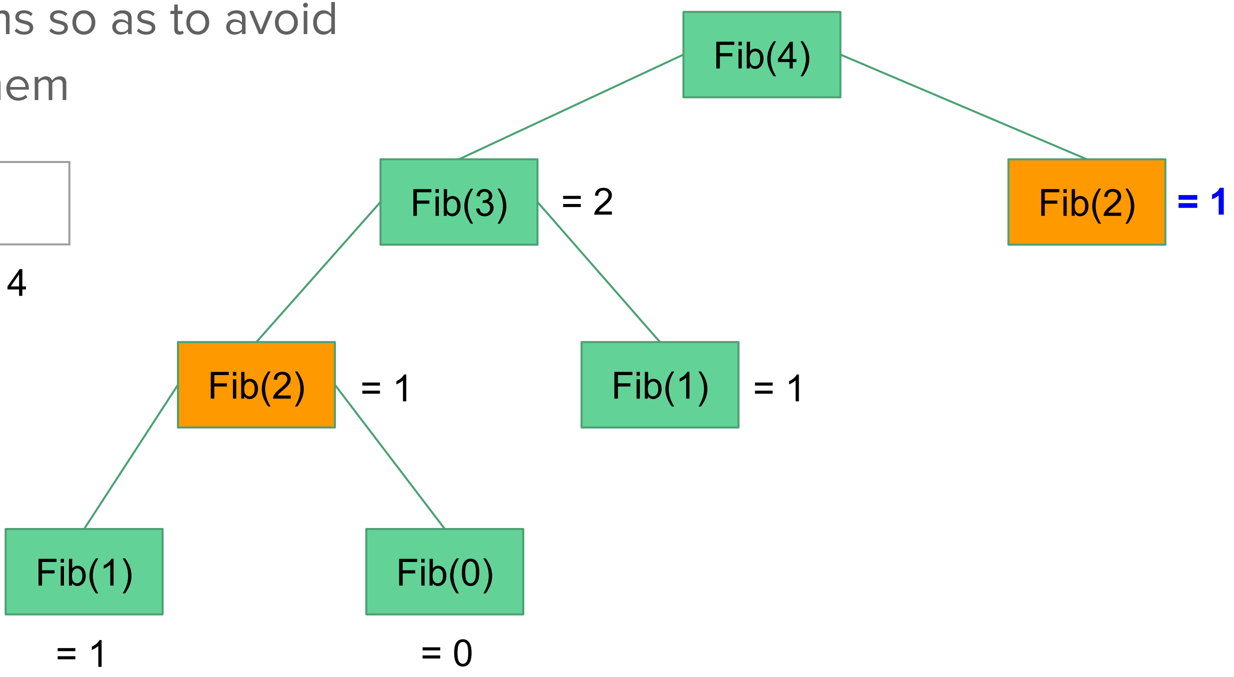
0	1	1	2	
0	1	2	3	4



Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

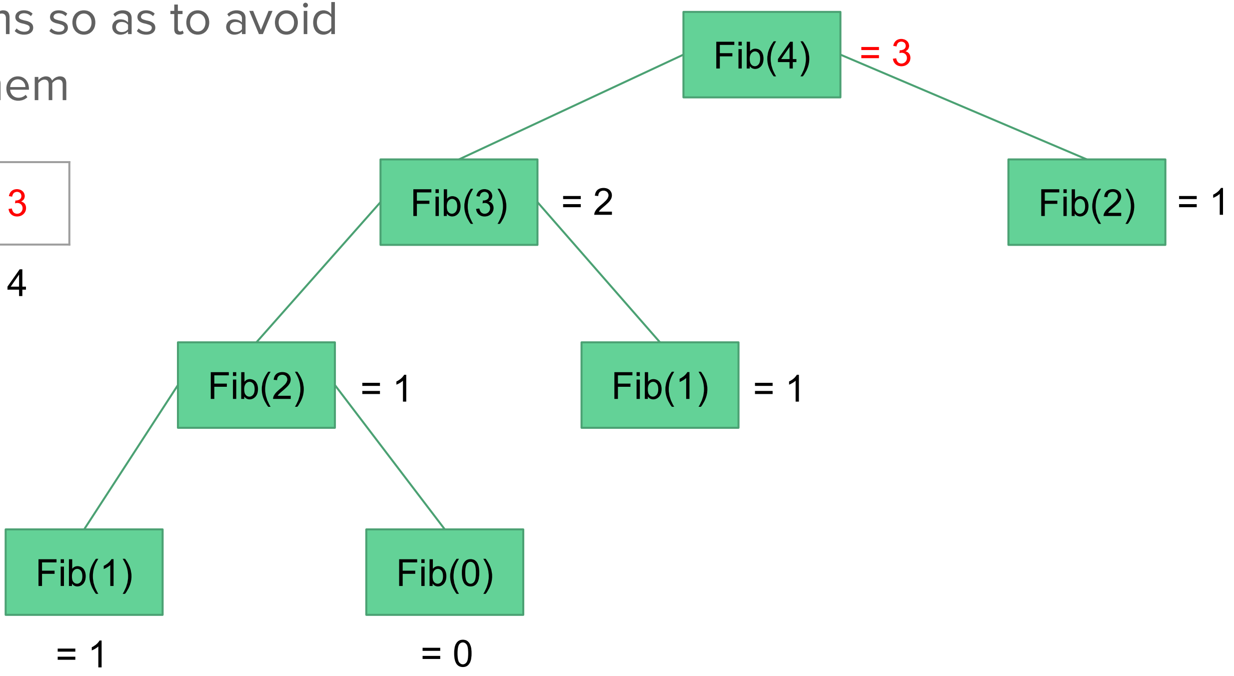
0	1	1	2	
0	1	2	3	4



Fibonacci numbers: Memoization

Storing the results of subsubproblems so as to avoid recomputing them

0	1	1	2	3
0	1	2	3	4



Fibonacci numbers: Memoization

```
def Fib(n, memo):
    if memo[n] != null:
        return memo[n]
    elif n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1, memo) + Fib(n-2, memo)
    memo[n] = result
    return result
```

$T(n) = O(n)$

Fibonacci numbers: bottom-up approach

- Solve all related sub-problems first
- Based on the results in the table, compute the solution to the "top" / original problem

  For example, to find F(4), first find out F(0), F(1), and so on, i.e. if the array from left to right

  F(0) = 0, F(1) = 1

0	1			
0	1	2	3	4

Fibonacci numbers: bottom-up approach

- Solve all related sub-problems first
- Based on the results in the table, compute the solution to the "top" / original problem

For example, to find F(4), first find out F(0), F(1), and so on, i.e. if the array from left to right

F(0) = 0, F(1) = 1

0	1			
0	1	2	3	4

F(2) = F(1) + F(0)

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

Fibonacci numbers: bottom-up approach

- Solve all related sub-problems first
- Based on the results in the table, compute the solution to the "top" / original problem

For example, to find F(4), first find out F(0), F(1), and so on, i.e. if the array from left to right

F(0) = 0, F(1) = 1

0	1			
0	1	2	3	4

F(2) = F(1) + F(0)

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

F(3) = F(2) + F(1)

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

Fibonacci numbers: bottom-up approach

- Solve all related sub-problems first
- Based on the results in the table, compute the solution to the "top" / original problem

For example, to find F(4), first find out F(0), F(1), and so on, i.e. if the array from left to right

F(0) = 0, F(1) = 1

0	1			
0	1	2	3	4

F(2) = F(1) + F(0)

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

F(3) = F(2) + F(1)

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

F(4) = F(3) + F(2)

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

Fibonacci numbers: bottom-up approach

```
def fib_bottom_up(n):
    if n <= 1:
        return n
    else:
        arr = [0 for i in range(n+1)]
        arr[1] = 1
        for i in range(2, n+1):
            arr[i] = arr[i-1] + arr[i-2]
        return arr[n]
```

$T(n) = O(n)$

No overhead for recursion

Longest Common Subsequence

A subsequence of a sequence/string S is obtained by deleting zero or more symbols from S.

Examples: subsequences of “algorithms” are alg, lits, oms, agihs etc.

The letters of a subsequence of S appear in order in S, but they are not required to be consecutive

Longest Common Subsequence

The longest common subsequence problem is to find a maximum length common subsequence between two sequences

The LCS of the sequences “president” and “providence” is “priden”

One of the LCSs of the sequences “algorithm” and “alignment” is “algm”

LCS: recursion

```
# Find LCS of two subsequences X and Y of length m and n
# respectively
def lcs(X, Y, m, n):
    if m == len(X) or n == len(Y):
        return 0
    elif X[m] == Y[n]:
        return 1 + lcs(X, Y, m+1, n+1)
    else:
        return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

LCS: recursion

X[0], Y[0]
A, T

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

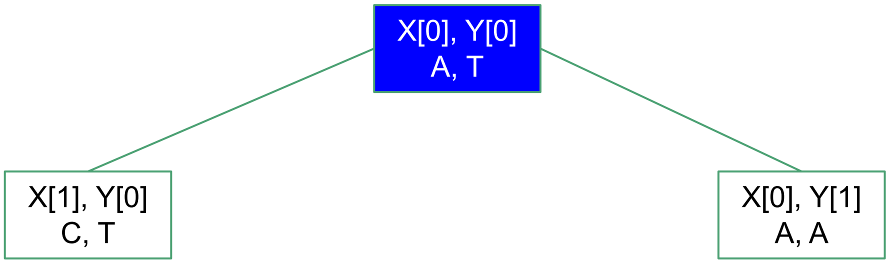
Y = “TAGC”

LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

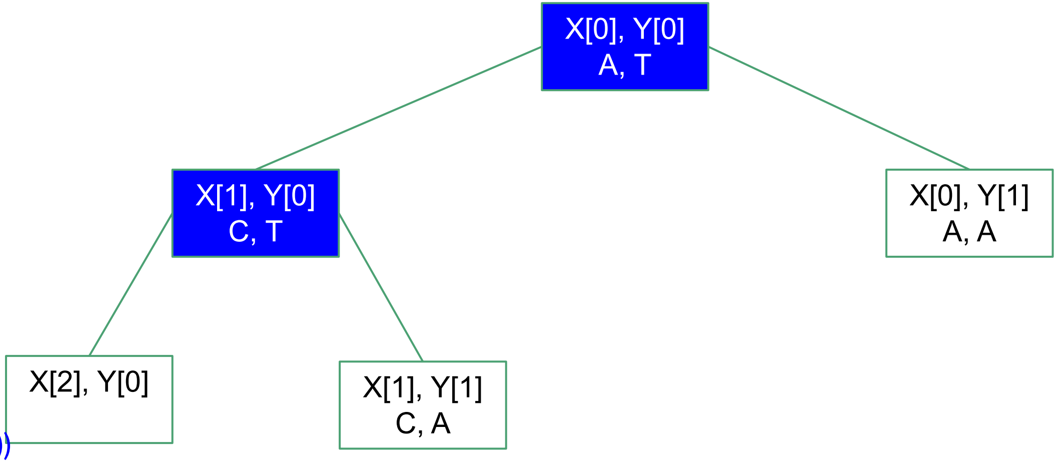
X = “AC”

Y = “TAGC”



LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

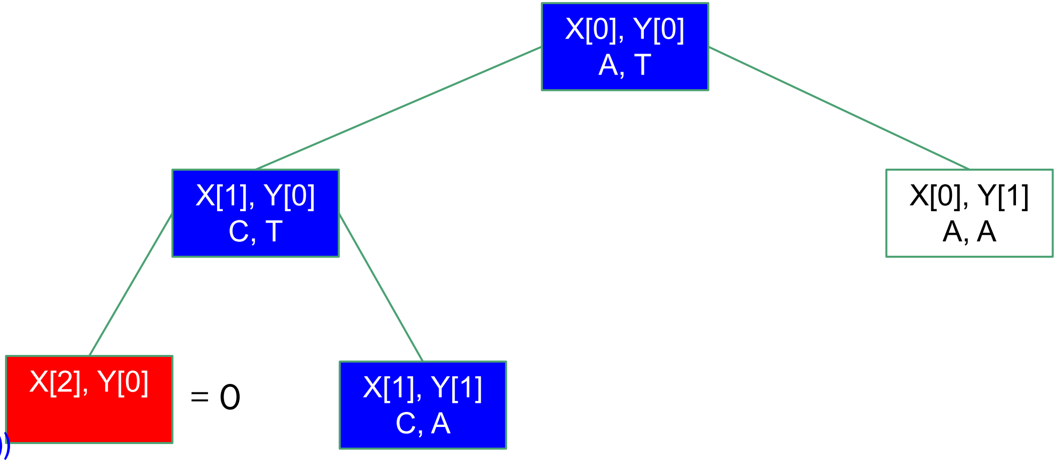


X = “AC”

Y = “TAGC”

LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

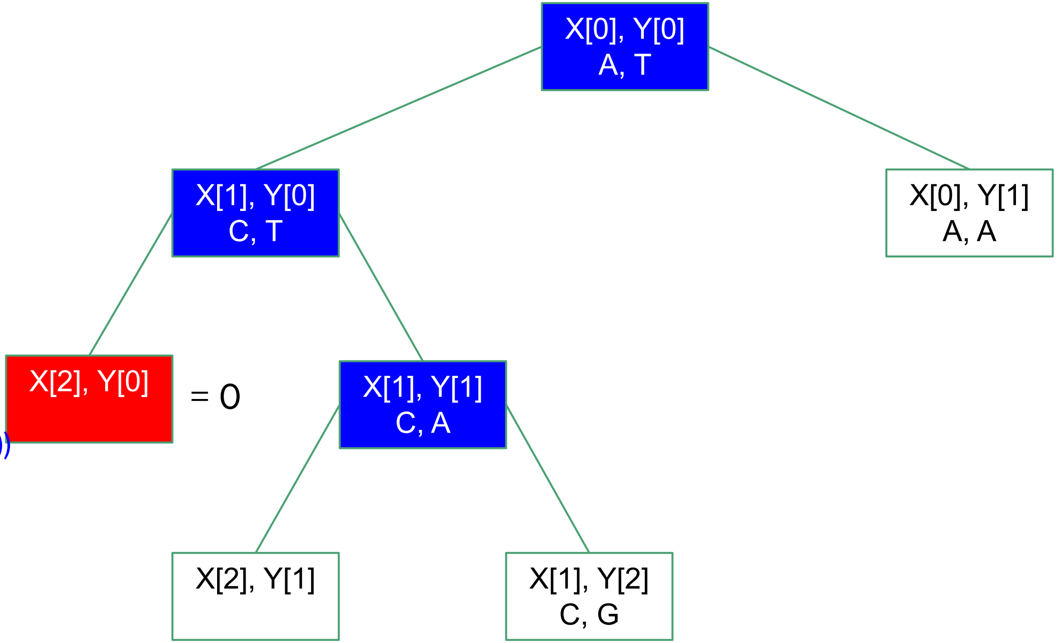


X = “AC”

Y = “TAGC”

LCS: recursion

```
if m == len(X) or n == len(Y):
  return 0
elif X[m] == Y[n]:
  return 1 + lcs(X, Y, m+1, n+1)
else:
  return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```



X = “AC”

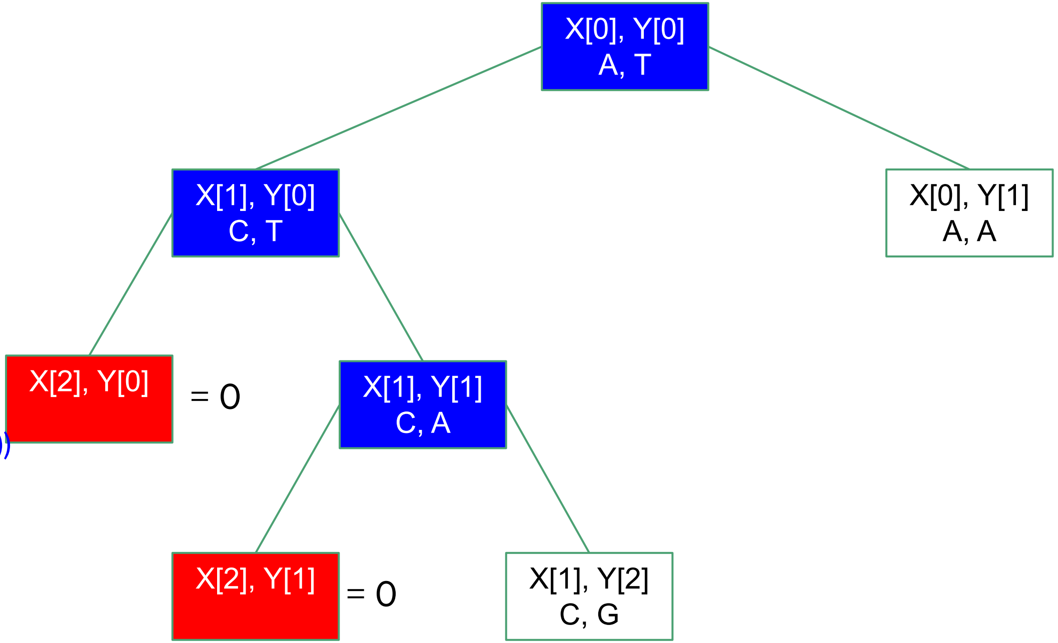
Y = “TAGC”

LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

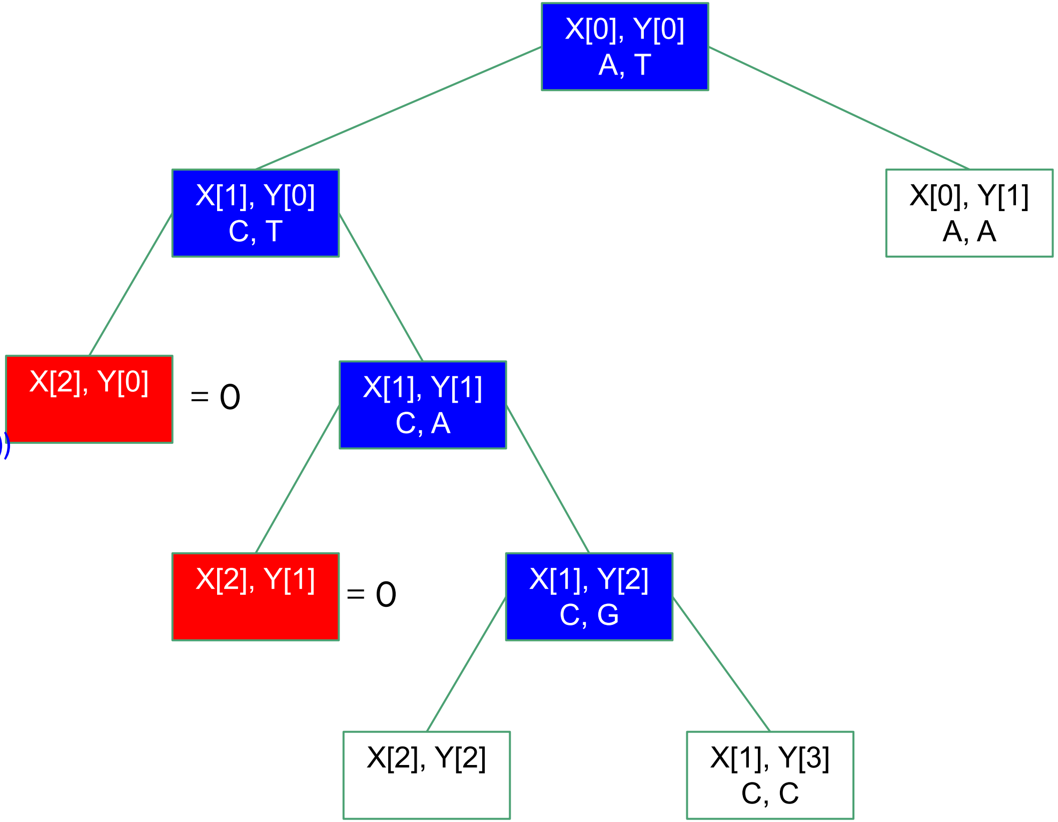


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

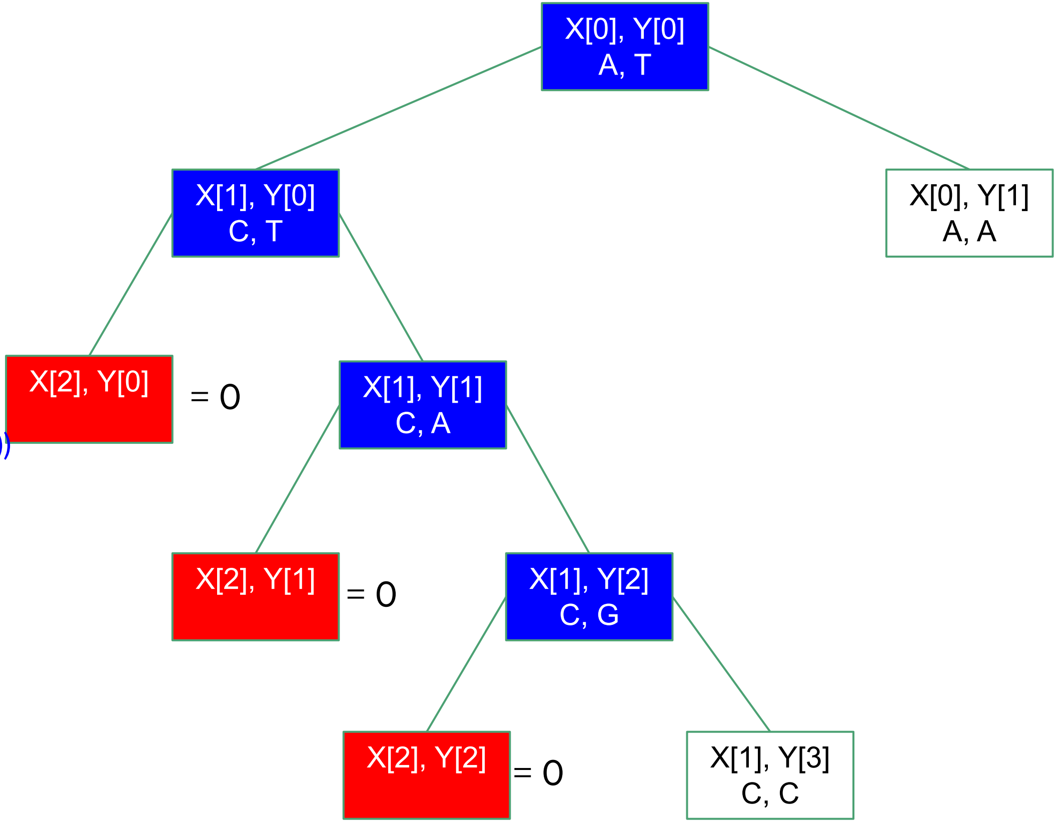


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

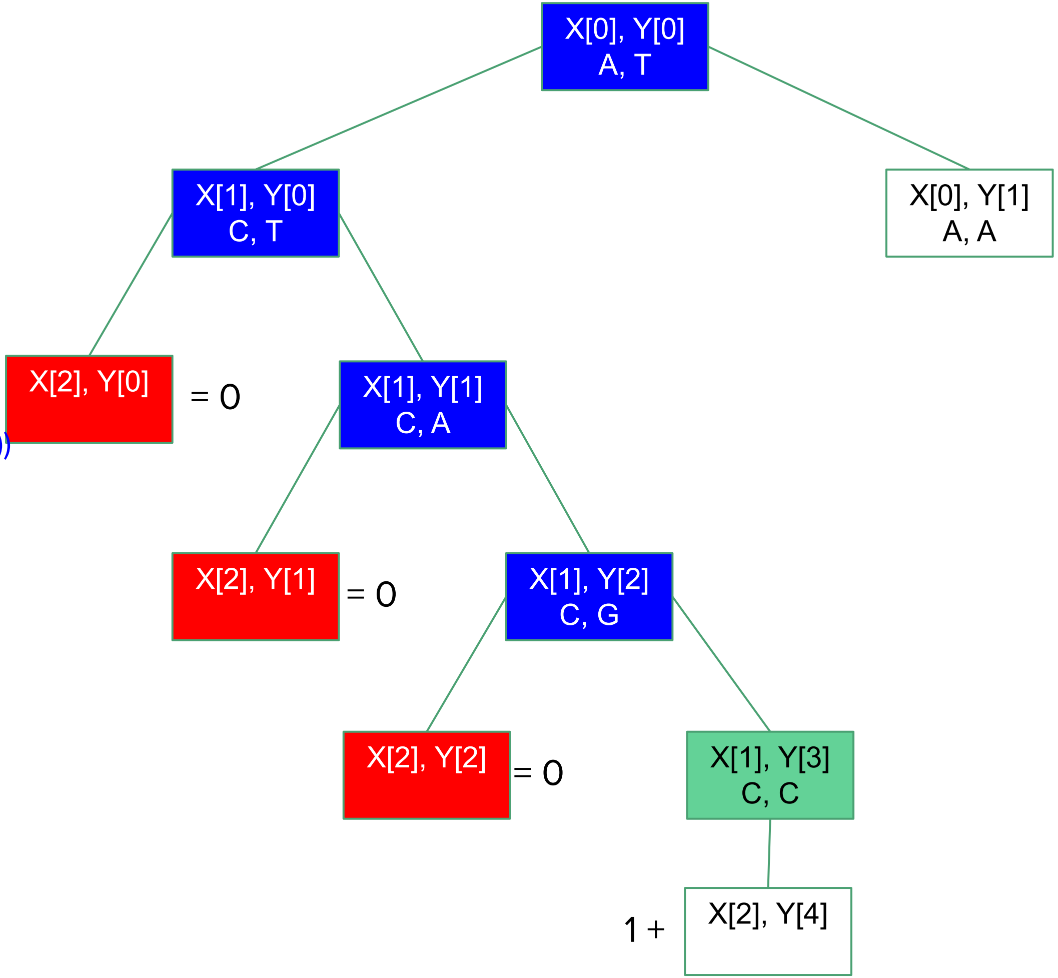


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

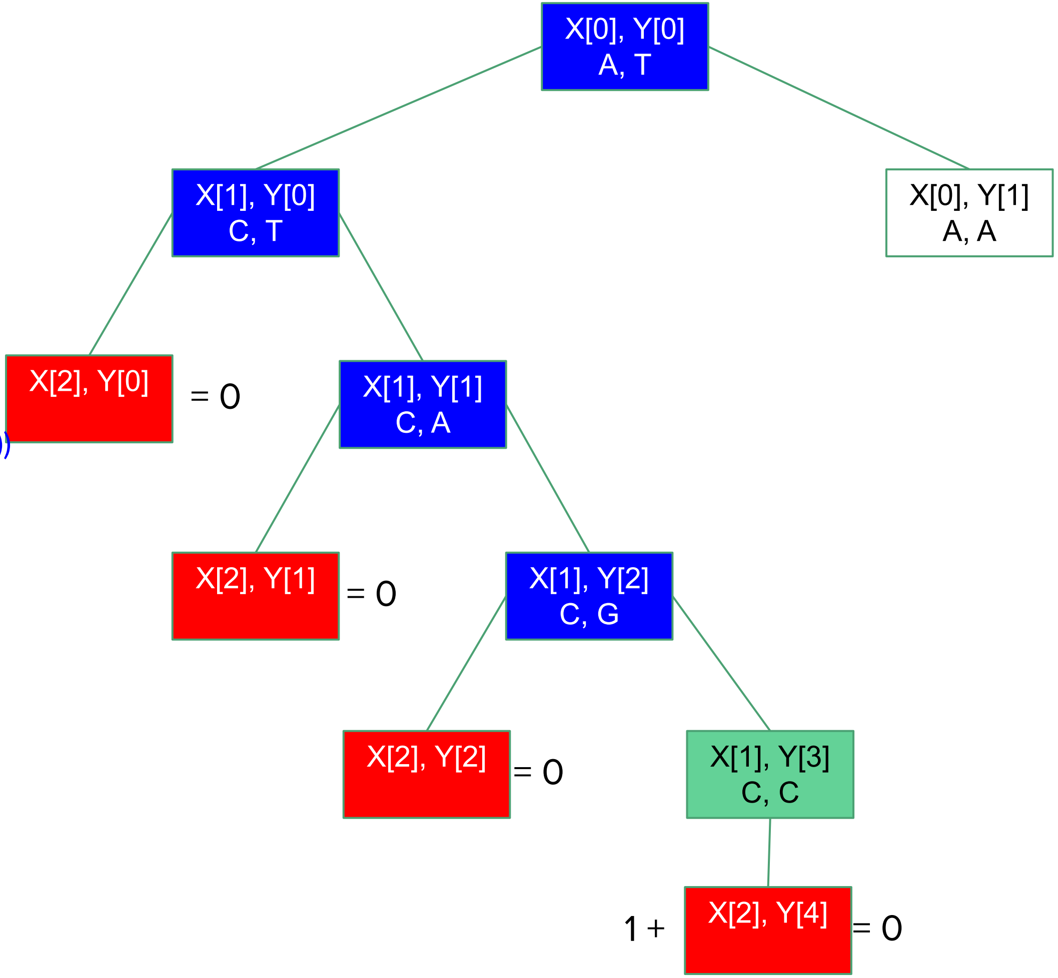


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

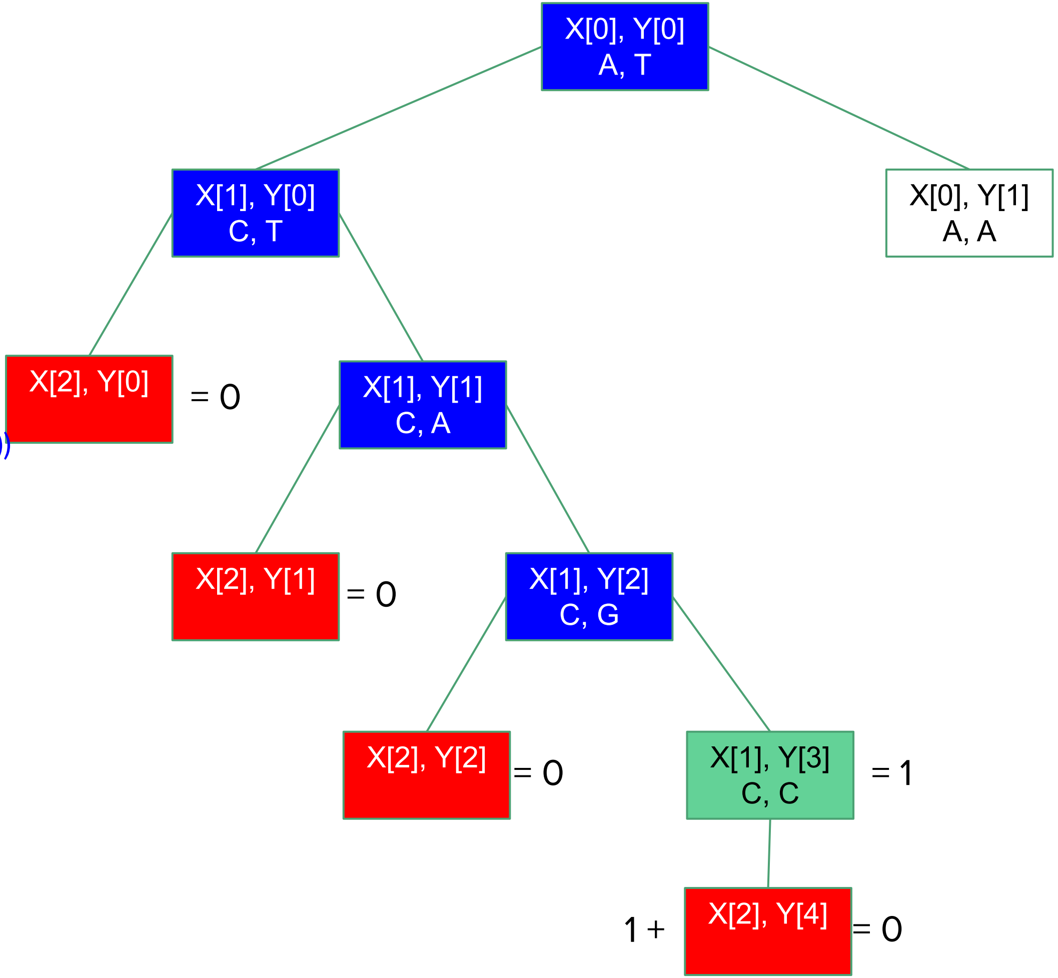


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

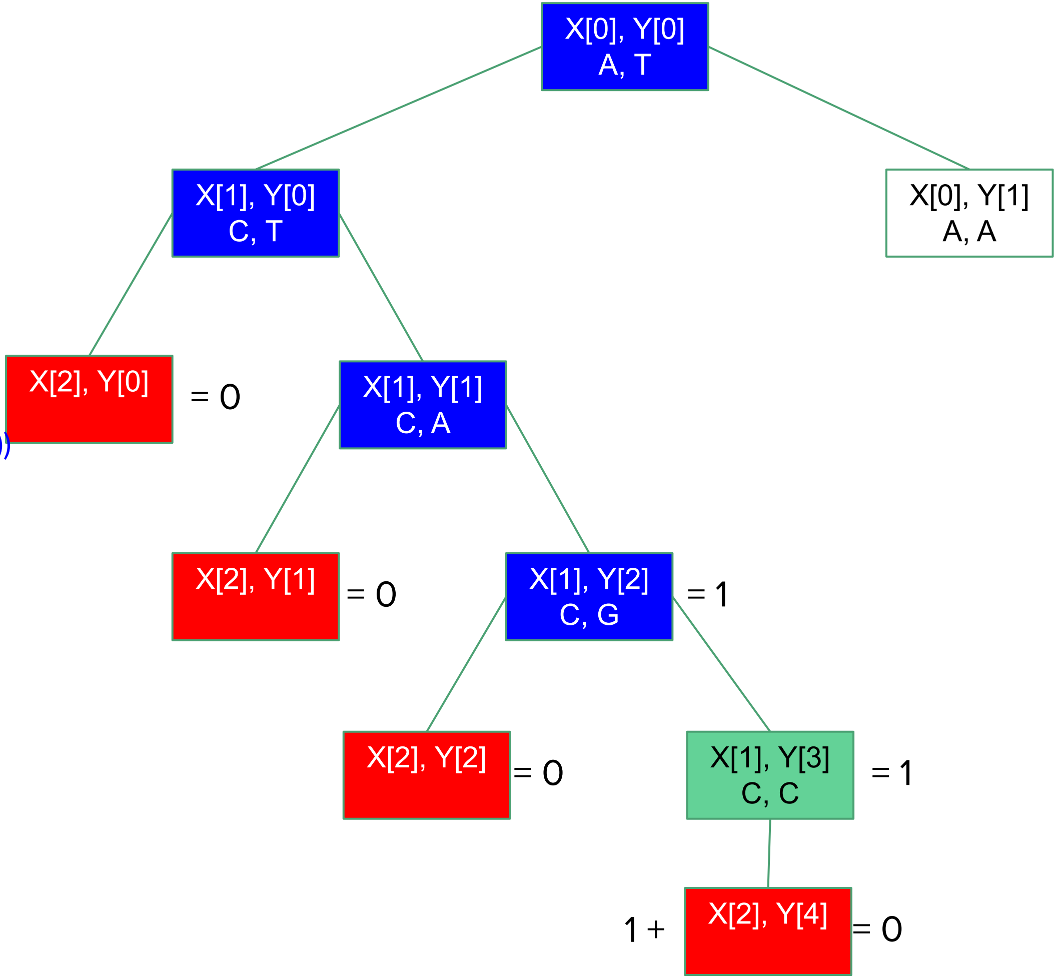


LCS: recursion

```
if m == len(X) or n == len(Y):
    return 0
elif X[m] == Y[n]:
    return 1 + lcs(X, Y, m+1, n+1)
else:
    return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

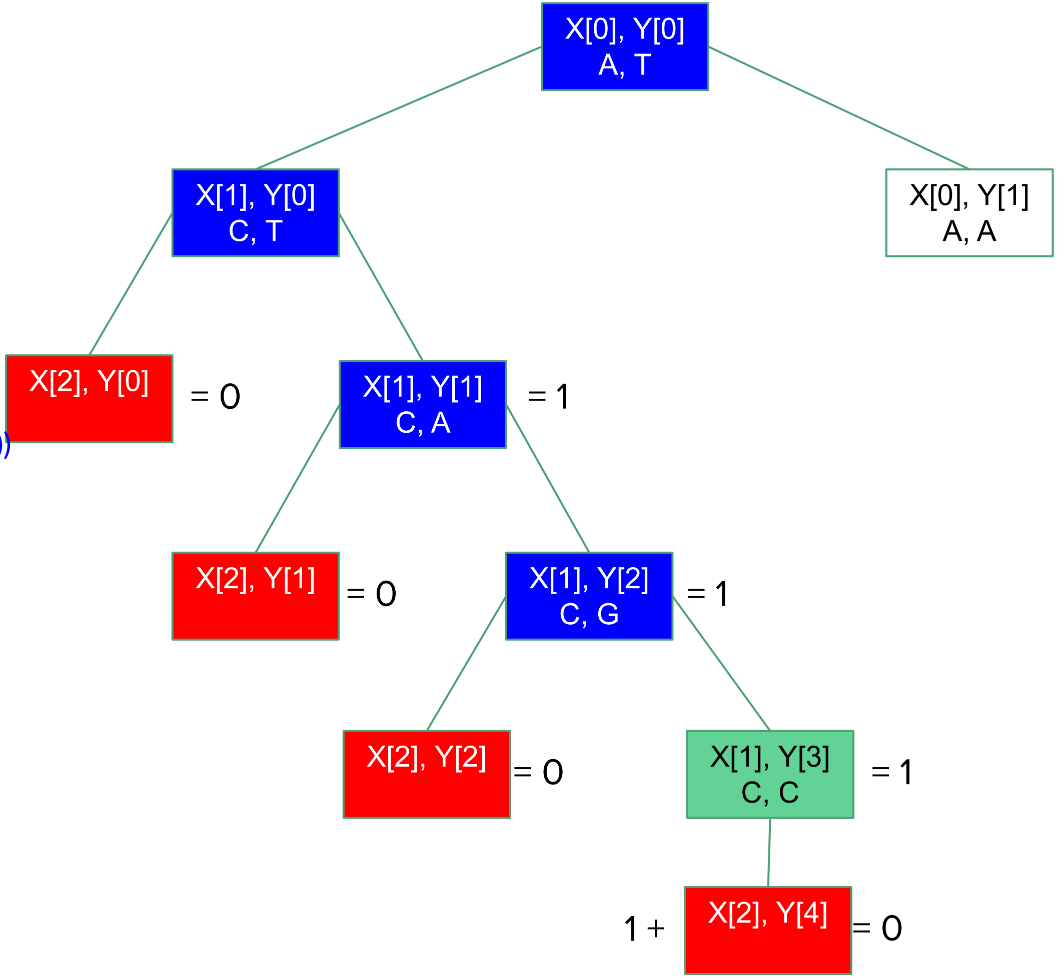


LCS: recursion

```
if m == len(X) or n == len(Y):
    return 0
elif X[m] == Y[n]:
    return 1 + lcs(X, Y, m+1, n+1)
else:
    return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

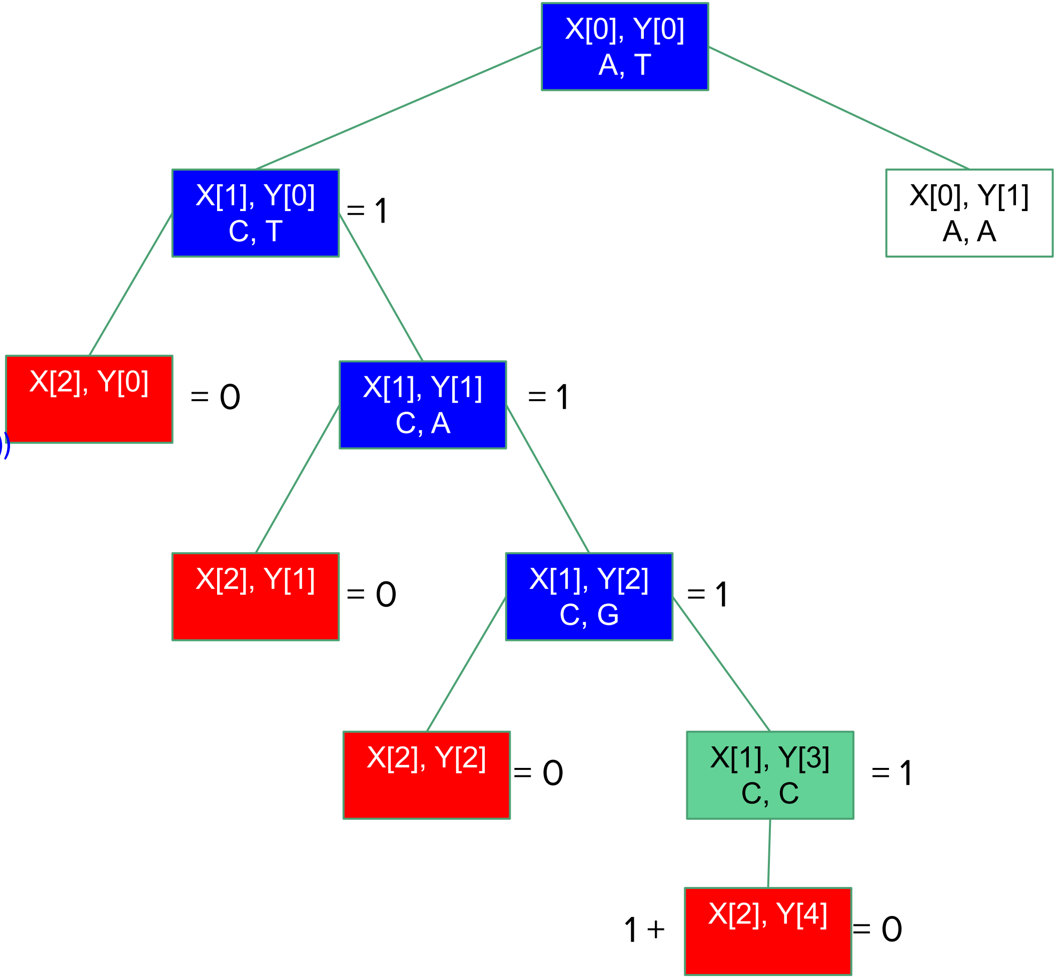


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

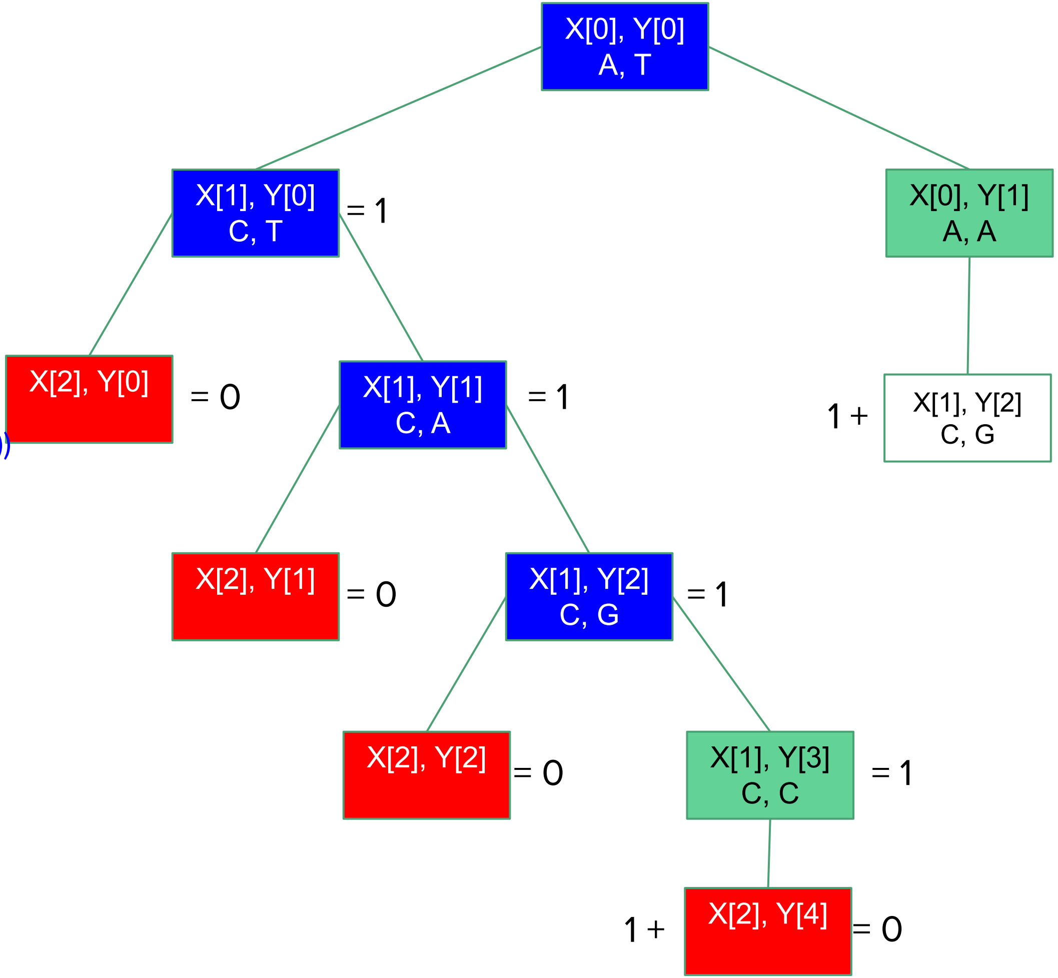


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

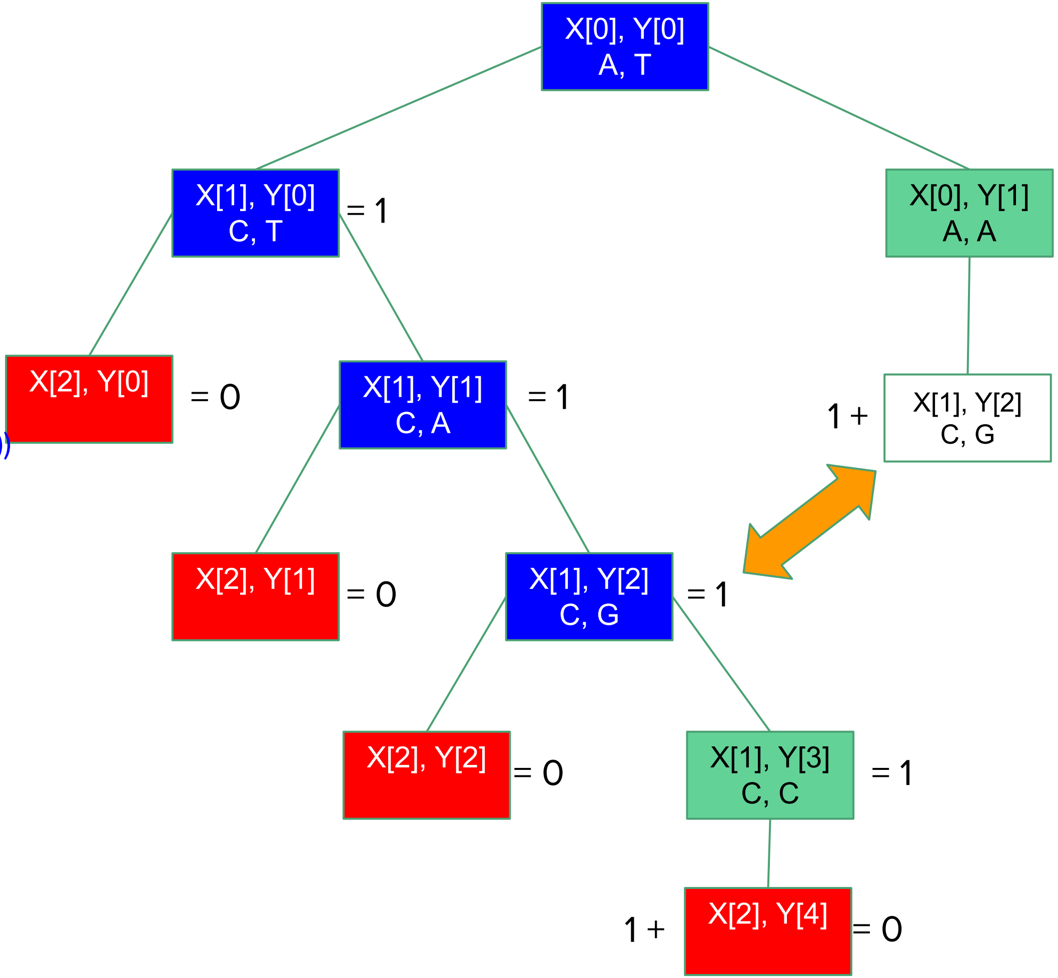


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

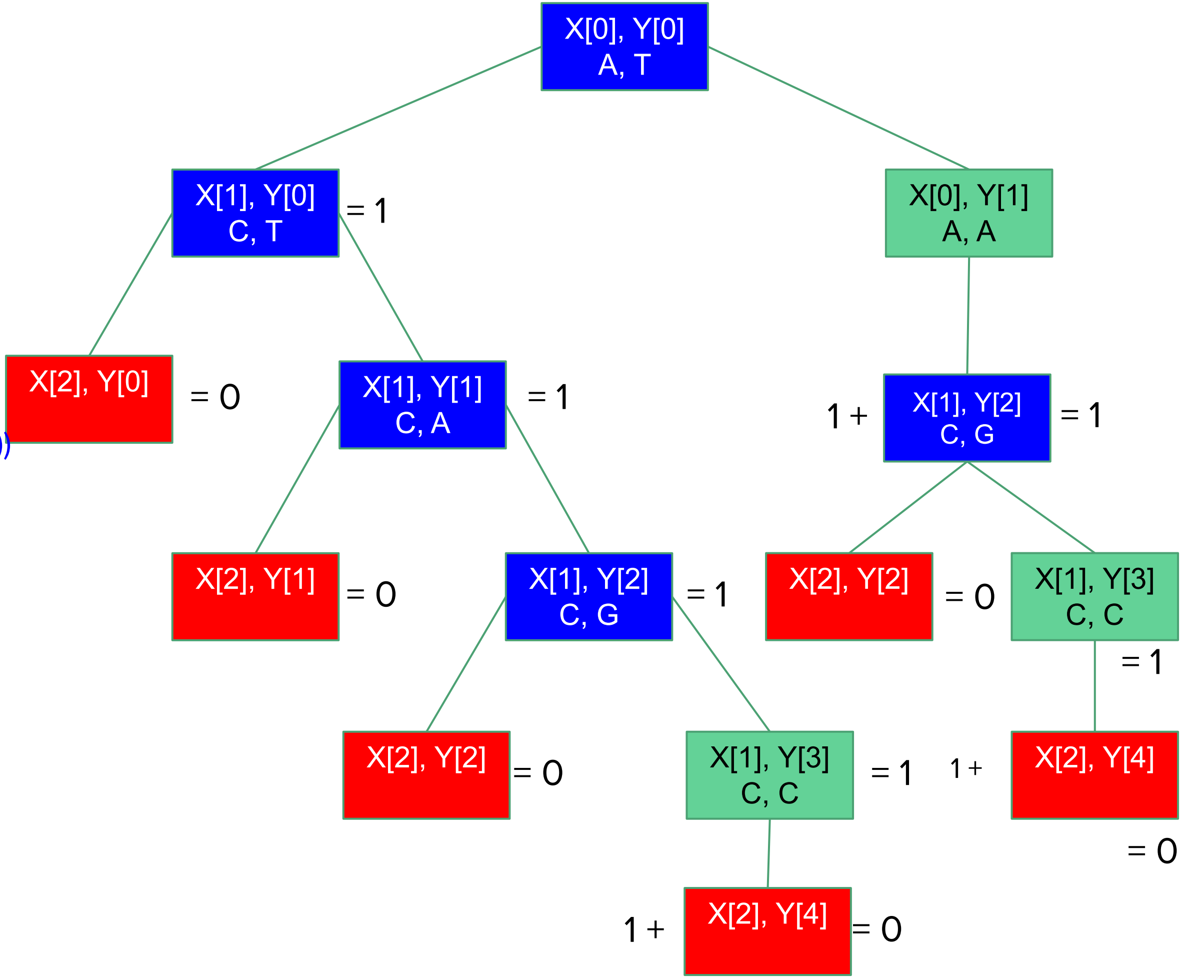


LCS: recursion

```
if m == len(X) or n == len(Y):
    return 0
elif X[m] == Y[n]:
    return 1 + lcs(X, Y, m+1, n+1)
else:
    return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

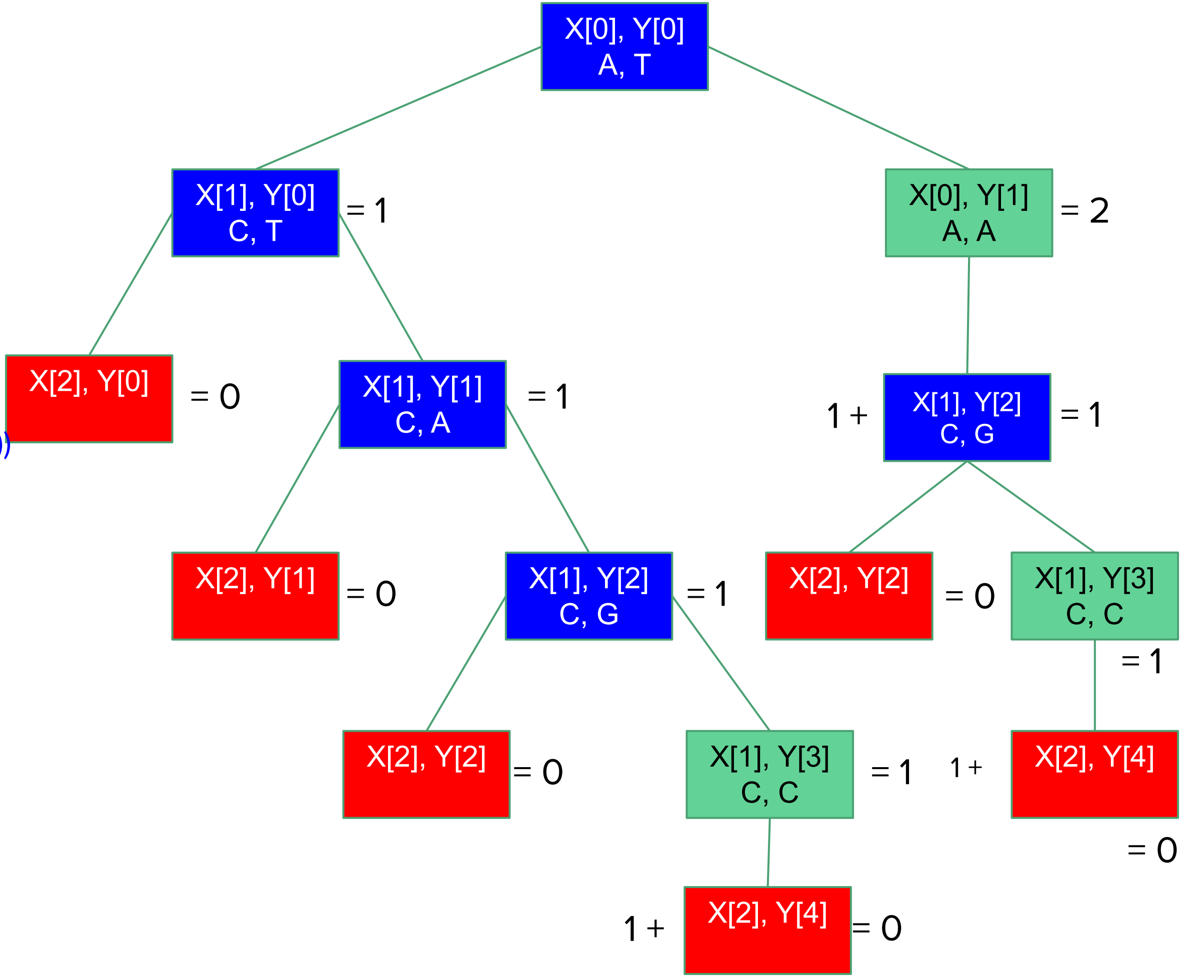


LCS: recursion

```
if m == len(X) or n == len(Y):
    return 0
elif X[m] == Y[n]:
    return 1 + lcs(X, Y, m+1, n+1)
else:
    return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”

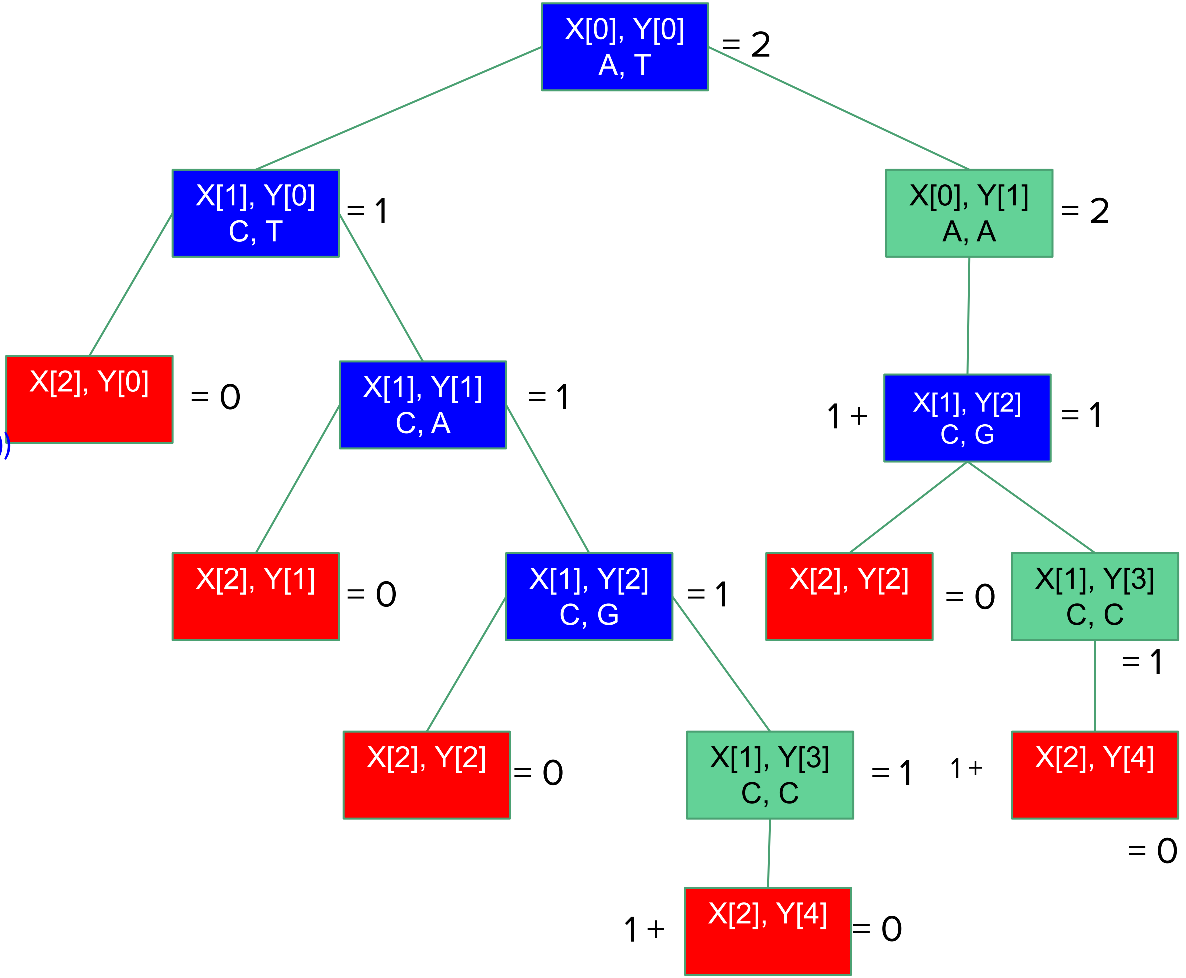


LCS: recursion

```
if m == len(X) or n == len(Y):
   return 0
elif X[m] == Y[n]:
   return 1 + lcs(X, Y, m+1, n+1)
else:
   return max(lcs(X, Y, m, n+1), lcs(X, Y, m+1, n))
```

X = “AC”

Y = “TAGC”



LCS: memoization

Assignment:

Write an algorithm for computing longest common subsequence of two subsequences, A and B, using memoization.

Hint: Store the intermediate results in an m x n matrix in top-down order, where m = length of A + 1 and n = length of B + 1. For example, if A = “AC” and B = “TAGC”, then store the intermediate results in the following array according to the previous tree

	0 (T)	1 (A)	2 (G)	3 (C)	4 (\0)
0 (A)					
1 (C)					
2 (\0)					

LCS: bottom-up approach

Let A = $a_1a_2...a_m$ and B=$b_1b_2...b_n$

len(i, j): the length of an LCS between A and B

With proper initializations, len(i, j) can be computed as follows:

$$len(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ len(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j , \\ \max(len(i, j-1), len(i-1, j)) & \text{if } i, j > 0 \text{ and } a_i \neq b_j . \end{cases}$$

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0										
r	0										
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

Here, A[1] == B[1]
Therefore,
len(1, 1) = len(0, 0) + 1 = 1

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1									
r	0										
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

Here, A[1] != B[2]
Therefore,
len(1, 2) = max(len(0, 1), len(1, 1)
= 1

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	**1**								
r	0										
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
r	0										
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	1	1	1	1	1	1	1	1
r	0	1									
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

Here, A[2] != B[1]
Therefore,
len(2, 1) = max(len(1, 1), len(2, 0)) = 1

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	1	1	1	1	1	1	1	1
r	0	1	2								
e	0										
s	0										
i	0										
d	0										
e	0										
n	0										
t	0										

Here, A[2] == B[2]
Therefore,
len(2, 2) = len(1, 1) + 1 = 2

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	1	1	1	1	1	1	1	1
r	0	1	2	2	2	2	2	2	2	2	2
e	0	1	2	2	2	2	2	3	3	3	3
s	0	1	2	2	2	2	2	3	3	3	3
i	0	1	2	2	2	3	3	3	3	3	3
d	0	1	2	2	2	2	4	4	4	4	4
e	0	1	2	2	2	2	4	5	5	5	5
n	0	1	2	2	2	2	4	5	6	6	6
t	0	1	2	2	2	2	4	5	6	6	6

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

Length of LCS = len(m, n) = 6

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	1	1	1	1	1	1	1	1
r	0	1	2	2	2	2	2	2	2	2	2
e	0	1	2	2	2	2	2	3	3	3	3
s	0	1	2	2	2	2	2	3	3	3	3
i	0	1	2	2	2	3	3	3	3	3	3
d	0	1	2	2	2	2	4	4	4	4	4
e	0	1	2	2	2	2	4	5	5	5	5
n	0	1	2	2	2	2	4	5	6	6	6
t	0	1	2	2	2	2	4	5	6	6	6

LCS: bottom-up approach

$\boxed{T(n) = O(mn)}$

```
def lcs_bottom_up(X, Y):
    m = len(X) + 1
    n = len(Y) + 1
    arr = [[0 for i in range(n)] for j in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            if (X[i-1] == Y[j-1]):
                arr[i][j] = 1 + arr[i-1][j-1]
            else:
                arr[i][j] = max(arr[i-1][j], arr[i][j-1])
    return arr[m-1][n-1]
```

LCS: bottom-up approach

Example:

A = “president”,

B = “providence”

LCS = “priden”

		p	r	o	v	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0
p	0	1	1	1	1	1	1	1	1	1	1
r	0	1	2	2	2	2	2	2	2	2	2
e	0	1	2	2	2	2	2	3	3	3	3
s	0	1	2	2	2	2	2	3	3	3	3
i	0	1	2	2	2	3	3	3	3	3	3
d	0	1	2	2	2	2	4	4	4	4	4
e	0	1	2	2	2	2	4	5	5	5	5
n	0	1	2	2	2	2	4	5	6	6	6
t	0	1	2	2	2	2	4	5	6	6	6

Matrix chain multiplication

Aka Matrix Product Parenthesization problem

Given : a chain/sequence of matrices $\{A_1, A_2, \ldots, A_n\}$

Goal : find the most efficient way to multiply these matrices, i.e. determine an order for multiplying matrices that has the lowest cost

Matrix chain multiplication

Matrix multiplication is **associative**, i.e. no matter how the product is parenthesized, the result obtained will remain the same.

For example, for four matrices A1, A2, A3, and A4, we would have:

(A1 (A2 (A3A4))),
(A1 ((A2A3) A4)),
((A1A2) (A3A4)),
((A1 (A2A3)) A4),
(((A1A2) A3) A4)

Recall matrix multiplication

We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B

If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix

The time to compute C is dominated by the number of scalar multiplications, which is p.q.r.

Example: When multiplying A_{3x4} and B_{4x5} total number of elements in the resulting matrix will be 3x5, and to get each of those elements, we need 4 multiplications. Thus, total number of multiplications will be 3x5x4 = 60

Matrix chain multiplication

The way the chain is parenthesized can have a dramatic impact on the cost of evaluating the product.

For example, if A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then

computing (AB)C needs $(10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500$ operations, while

computing A(BC) needs $(30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000$ operations

Matrix chain multiplication

How to optimize:

- Brute force – look at every possible way to parenthesize
- Dynamic programming

Matrix chain multiplication: Dynamic programming

Steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

1. The structure of an optimal solution

We can build an optimal solution into two suubproblems finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

That is, to figure out how to best multiply A_i x ... x A_j, we consider all possible middle points k and select the one that minimizes:

Optimal cost to multiply A_i ... A_k
+ Optimal cost to multiply A_{k+1} ... A_j
+ Cost to multiply the results

2. Recursive solution

Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix A_i x … x A_j

We can define m[i, j] recursively as follows:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases}$$

Our recursive definition for the minimum cost of parenthesizing the product A_i x A_{i+1} x … x A_j is

3. Computing the optimal costs

Let us assume that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$.

Dimensions sequence = $\langle p_0, p_1, p_2, \ldots p_n \rangle$

We define two tables:

1. m of dimensions n x n for storing the m[i,j] costs
2. s of dimensions n-1 x n-1 for recording which index of k achieved the optimal cost in computing m[i,j]

3. Computing the optimal costs

Example:

Given the following matrices:

1. A_1 of dimension 5 x 4
2. A_2 of dimension 4 x 6
3. A_3 of dimension 6 x 2
4. A_4 of dimension 2 x 7

Dimensions sequence = $\langle 5, 4, 6, 2, 7 \rangle$

3. Computing the optimal costs

Example (Continued):

We define two tables, m and s as follows:

m =

	1	2	3	4
1				
2				
3				
4				

s =

	2	3	4
1			
2			
3			

Recall m[i, j] = the minimum number of scalar multiplications needed to multiply A_i x … x A_j

3. Computing the optimal costs

Example (Continued):

m[1, 1] = 0 because the chain consists of just one matrix A1, so no multiplications

Similarly, m[2, 2] = m[3, 3] = m[4, 4] = 0

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

	2	3	4
1			
2			
3			

3. Computing the optimal costs

Example (Continued):

m[1, 2] = the minimum number of multiplications to compute $A_1 \times A_2$

$= 5 \times 4 \times 6 = 120$

Recall: Dimensions of $A_1 = 5 \times 4$

$A_2 = 4 \times 6$

	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

	2	3	4
1	1		
2			
3			

3. Computing the optimal costs

Example (Continued):

m[2, 3] = the minimum number of multiplications to compute $A_2 \times A_3$

$= 4 \times 6 \times 2 = 48$

Recall: Dimensions of $A_2 = 4 \times 6$

$A_3 = 6 \times 2$

	1	2	3	4
1	0	120		
2		0	48	
3			0	
4				0

	2	3	4
1	1		
2		2	
3			

3. Computing the optimal costs

Example (Continued):

m[3, 4] = the minimum number of multiplications to compute $A_3 \times A_4$

$= 6 \times 2 \times 7 = 84$

Recall: Dimensions of $A_3 = 6 \times 2$

$A_4 = 2 \times 7$

	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

	2	3	4
1	1		
2		2	
3			3

3. Computing the optimal costs

Example (Continued):

m[1, 3] = the minimum number of multiplications to compute $A_1 \times A_2 \times A_3$

$A_1 \times A_2 \times A_3 = (A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$

Dimensions of $(A_1 \times A_2) = 5 \times 6$
$(A_2 \times A_3) = 4 \times 2$

Number of multiplications in $(A_1 \times A_2) \times A_3$ = m[1,2] + m[3,3] + 5 x 6 x 2

Number of multiplications in $\mathbf{A_1 \times (A_2 \times A_3)}$ = m[1,1] + m[2,3] + 5 x 4 x 2

∴ m[1, 3] = min {120 + 0 + 60, **0 + 48 + 40**} = 88

	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

	2	3	4
1	1	1	
2		2	
3			3

3. Computing the optimal costs

Example (Continued):

m[2, 4] = the minimum number of multiplications to compute $A_2 \times A_3 \times A_4$

$A_2 \times A_3 \times A_4 = (A_2 \times A_3) \times A_4 = A_2 \times (A_3 \times A_4)$

Dimensions of $(A_2 \times A_3) = 4 \times 2$

$(A_3 \times A_4) = 6 \times 7$

Number of multiplications in $\mathbf{(A_2 \times A_3) \times A_4}$ = m[2, 3] + m[4,4] + 4 x 2 x 7

Number of multiplications in $A_2 \times (A_3 \times A_4)$ = m[2,2] + m[3,4] + 4 x 6 x 7

$\therefore$ m[1, 3] = min {**48 + 0 + 56**, 0 + 84+ 168} = 104

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

	2	3	4
1	1	1	
2		2	3
3			3

3. Computing the optimal costs

Example (Continued):

$m[1, 4]$ = the minimum number of multiplications to compute $A_1 \times A_2 \times A_3 \times A_4$

$$
\begin{aligned}
A_1 \times A_2 \times A_3 \times A_4 &= A_1 \times ((A_2 \times A_3) \times A_4) \\
&= (A_1 \times A_2) \times (A_3 \times A_4) \\
&= A_1 \times (A_2 \times (A_3 \times A_4)) \\
&= ((A_1 \times A_2) \times A_3) \times A_4 \\
&= (A_1 \times (A_2 \times A_3)) \times A_4
\end{aligned}
$$

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

	2	3	4
1	1	1	
2		2	3
3			3

3. Computing the optimal costs

Example (Continued):

Since we have already computed m[2, 4] and m[1, 3], we consider the following three multiplications only

$$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times A_2) \times (A_3 \times A_4)$$
$$= A_1 \times (A_2 \times A_3 \times A_4)$$
$$= (A_1 \times A_2 \times A_3) \times A_4$$

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

	2	3	4
1	1	1	
2		2	3
3			3

3. Computing the optimal costs

Example (Continued):

Dimensions of $(A_1 \times A_2 \times A_3) = 5 \times 2$

$(A_2 \times A_3 \times A_4) = 4 \times 7$

Number of multiplications in $(A_1 \times A_2 \times A_3) \times A_4$

$= m[1, 3] + m[4, 4] + 5 \times 2 \times 7$

$= 88 + 0 + 70$

Number of multiplications in $A_1 \times (A_2 \times A_3 \times A_4)$

$= m[1, 1] + m[2, 4] + 5 \times 4 \times 7$

$= 0 + 104 + 140$

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

	2	3	4
1	1	1	
2		2	3
3			3

3. Computing the optimal costs

Example (Continued):

Dimensions of $(A_1 \times A_2) = 5 \times 6$
$(A_3 \times A_4) = 6 \times 7$

Number of multiplications in $(A_1 \times A_2) \times (A_3 \times A_4)$
$= m[1, 2] + m[3, 4] + 5 \times 6 \times 7$
$= 120 + 84 + 210$

$\therefore$ m[1, 4] = min { **88 + 0 + 70**, 0 + 104 + 140, 120 + 84 + 210 } = 158

That is the number of multiplications is minimum for $(A_1 \times A_2 \times A_3) \times A_4$

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

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

4. Constructing an optimal solution

The table s[1...n-1, 2...n] gives us the information needed to construct an optimal solution

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

4. Constructing an optimal solution

The table s[1...n-1, 2...n] gives us the information needed to construct an optimal solution

$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times A_2 \times A_3) \times A_4$

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

4. Constructing an optimal solution

The table s[1...n-1, 2...n] gives us the information needed to construct an optimal solution

$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times (A_2 \times A_3)) \times A_4$

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

4. Constructing an optimal solution

$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times (A_2 \times A_3)) \times A_4$