Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University

a.khan@innopolis.ru

Recap

- What is an algorithmic strategy?

- Learn about commonly used Algorithmic Strategies
  - ❖ Max-SubArray Problem
  - ❖ Brute-force
  - ❖ Divide-and-conquer
  - ❖ Master Theorem

Today’s objectives

- Learn about
  - ❖Dynamic programming
  - ❖Tabulation and Memoization
  - ❖Max-subarray with DP
  - ❖Longest Common Subsequence Problem

Dynamic Programming

Dynamic Programming

- Similar to divide-and-conquer, it solves the problem by combining solutions to the sub-problems
- But it applies when sub-problems overlap
- That is, sub-problems **share sub-sub-problems**!
- To avoid solving the same sub-problems more than once, their results are stored (often in a data structure) that is updated dynamically

Example

- Fibonacci Numbers

```
Fibonacchi(N) = 0                                         for n=0
              = 1                                         for n=1
              = Fibonacchi(N-1)+Finacchi(N-2)              for n>1
```

Example

- Fibonacci Numbers using Recursion

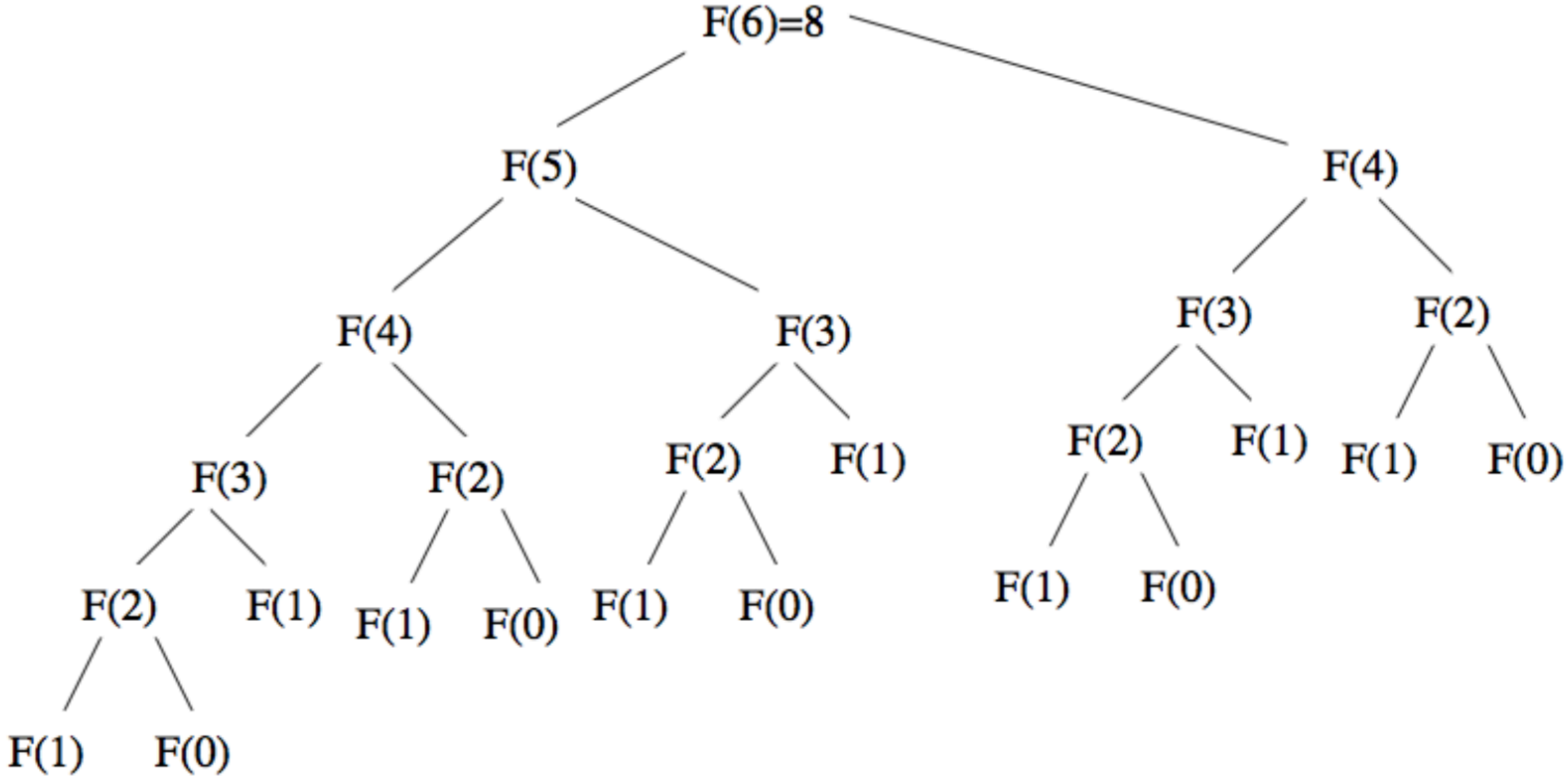
```
public int fibRecur(int x) {
        if (x == 0)
            return 0;
        if (x == 1)
            return 1;
        else {
            int f = fibRecur(x - 1) + fibRecur(x - 2);
            return f;
        }

    }
```

$$T(n) = O(2^n)$$

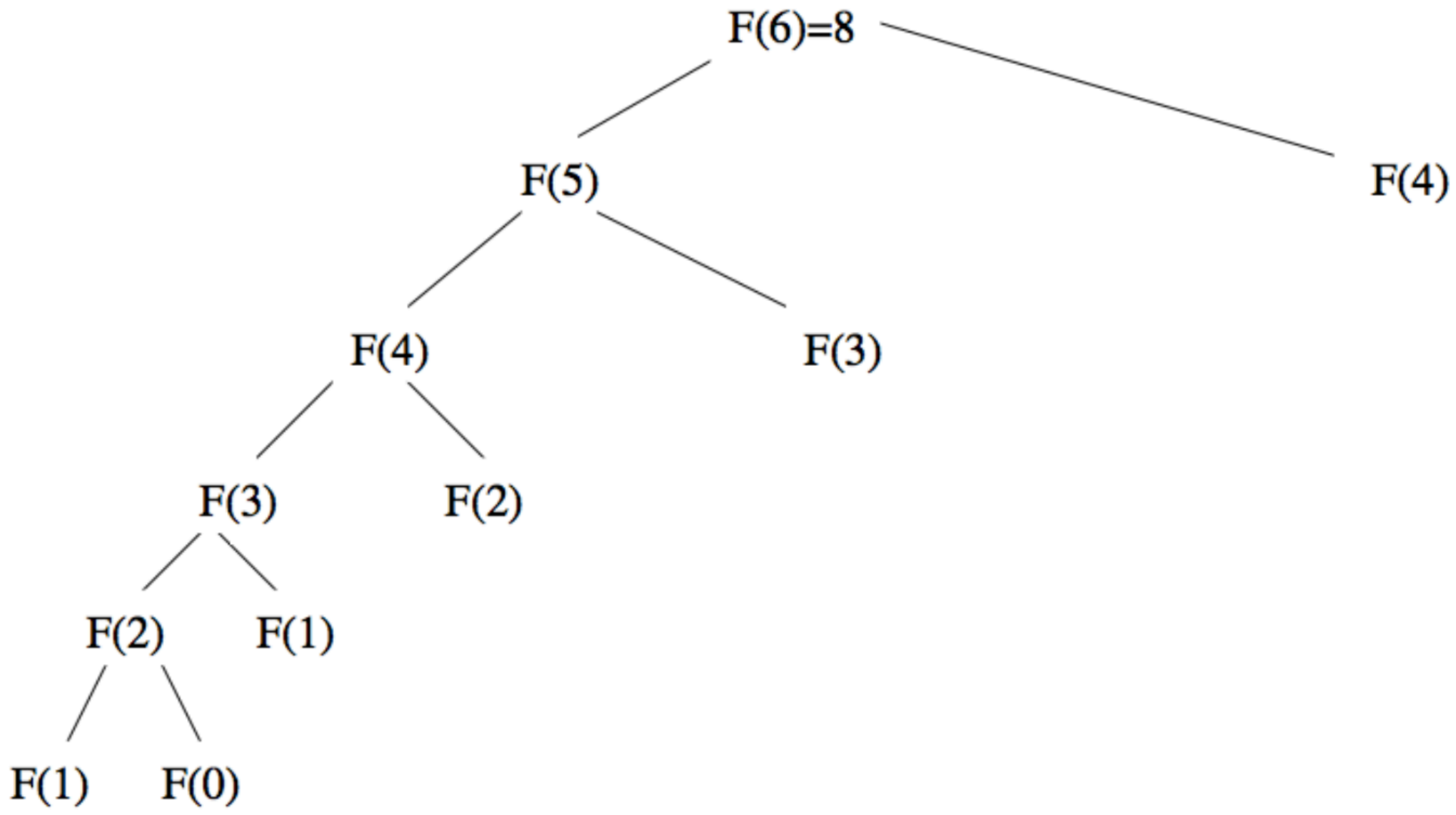
Recursion Tree

- $n - th$ Fibonacci Numbers



Dynamic Programming

- $n - th$ Fibonacci Numbers

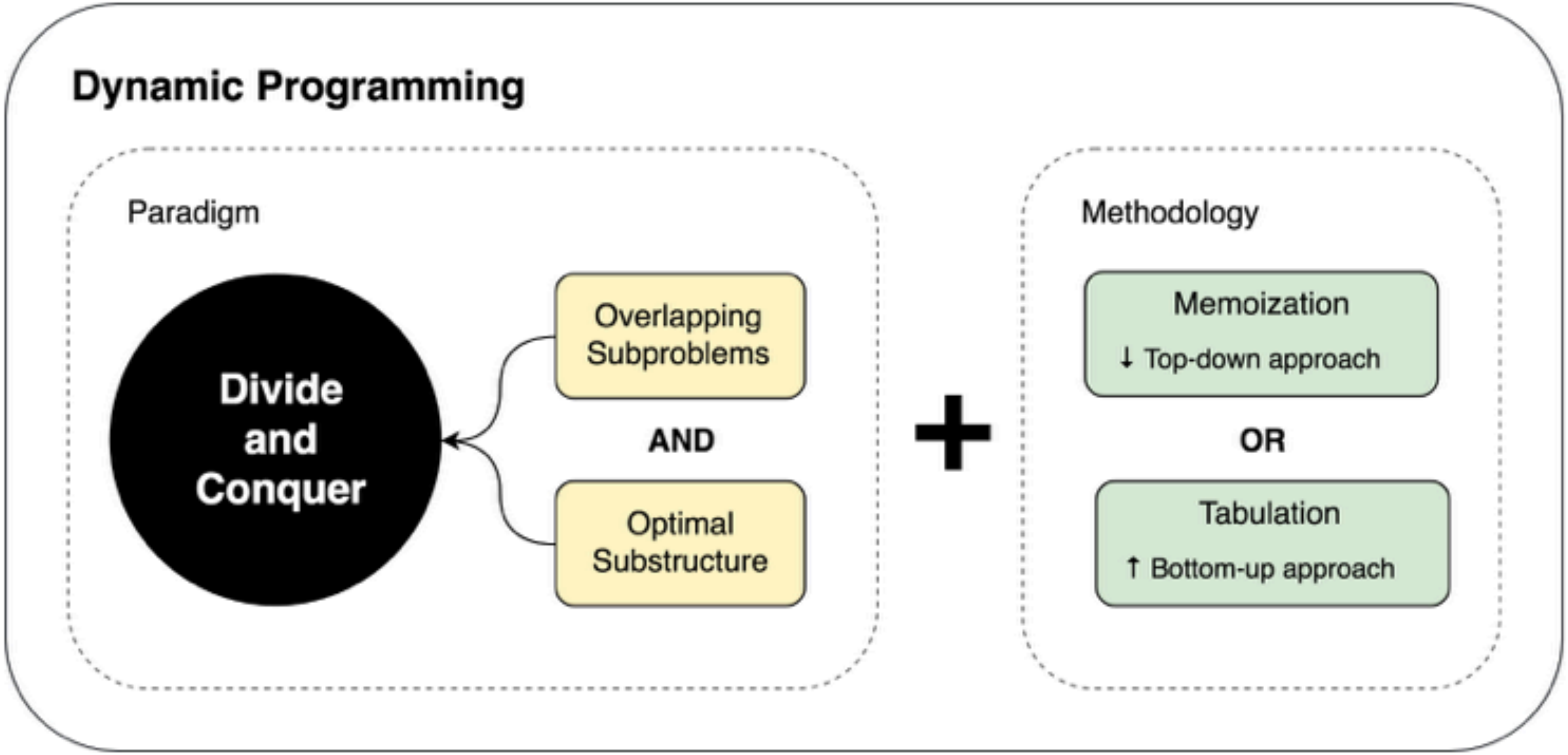


Dynamic Programming

- Thus

"Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again."

Dynamic Programming



https://dzone.com/articles/dynamic-programming-or-quotdivide-and-conquerquot

Dynamic Programming: Two Approaches

- Tabulation

"The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table."

Dynamic Programming

- Fibonacci Numbers – Bottom-up (Tabulation)

```
/* Java program for Tabulated version */

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

Time Complexity: O(n)

Dynamic Programming

- Memoization

"The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions."

Dynamic Programming

- Fibonacci Numbers – Top-down (Memoization)

```
/* Java program for Memoized version
public class Fibonacci {
        final int MAX = 100;
        final int NIL = -1;
        int lookup[] = new int[MAX];

        void _initialize() {
                for (int i = 0; i < MAX; i++)
                        lookup[i] = NIL;
        }
```

Dynamic Programming

- Fibonacci Numbers – Top-down (Memoization)

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

Time Complexity: O(n)

Summary: Dynamic Programming

- Key is to relate the solution of the whole problem and the solutions of subproblems.

    - ❖ Same is true of divide & conquer, but here the subproblems need not be disjoint. – they need not divide the input (i.e., they can “overlap”)

- A dynamic programming algorithm computes the solution of every subproblem needed to build up the solution for the whole problem.
    - compute each solution using the above relation
    - store all the solutions in an array (or matrix)
    - algorithm simply fills in the array entries in some order

Dynamic Programming

- Max-SubArray

- Let's work with the following example

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

- Now, for every index, we will calculate a quantity called its ***local-maximum-sum***

Dynamic Programming: Max-SubArray

- Lacal-Maximum-sum for index 4, and 5

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

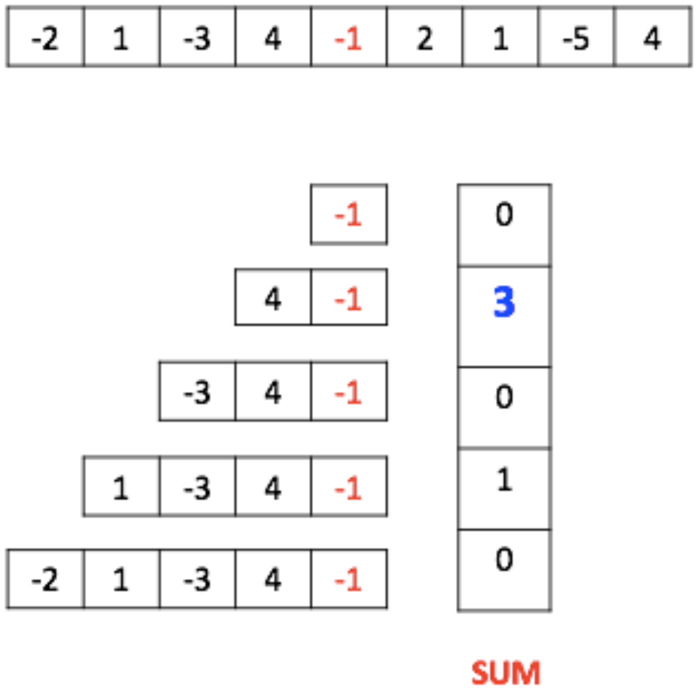
					SUM
				-1	0
			4	-1	**3**
		-3	4	-1	0
	1	-3	4	-1	1
-2	1	-3	4	-1	0

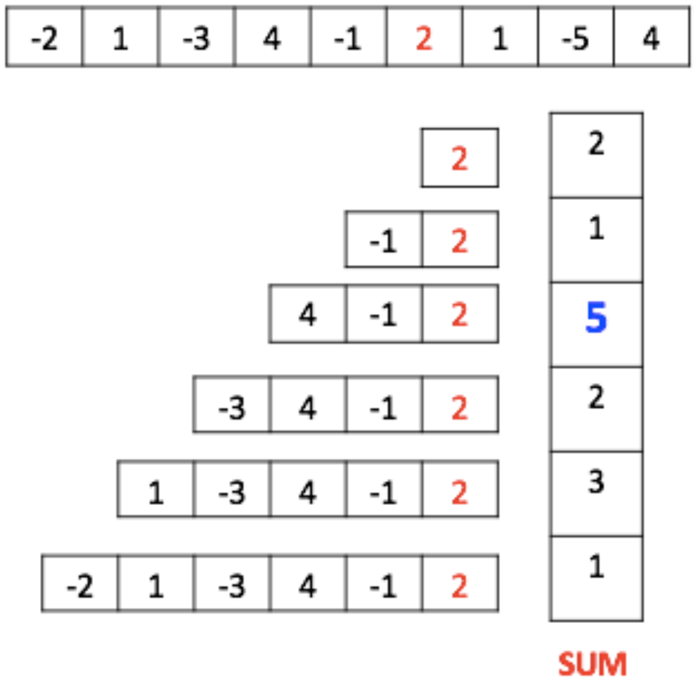
-2	1	-3	4	-1	2	1	-5	4

						SUM
					2	2
				-1	2	1
			4	-1	2	**5**
		-3	4	-1	2	2
	1	-3	4	-1	2	3
-2	1	-3	4	-1	2	1

Dynamic Programming: Max-SubArray

- Lacal-Maximum-sum for index 4, and 5





Now, Ask yourself, is there a way to compute local-max-sum at index 5 using the local-max-sum at index 4?

Dynamic Programming

- Max Subarray Problem

- Let $S(i)$ be the local-max-sum at ith-index

A[0]	A[1]			…				A[n-1]

- Then it can be recursively defined as

$$S(i) = \max((S(i-1) + A[i]), 0)$$

Dynamic Programming

- Max Subarray Problem

```
Max-Subarray-Sum (A, n)
sum ← 0, sum' ← 0
for i←1 to n
    sum' ← max{0, sum' + A[i]}
    sum ← max{sum, sum'}
return sum
```

Elements of Dynamic Programming

- So we just learned how DP works

- But, given a problem, how do we know:

  - Whether we can use DP
  - How to attack the problem with DP

Will be covered in detail in the tutorial

Longest Common Subsequence

Subsequence

A subsequence of a sequence/string $X = \langle x_1, x_2, \ldots, x_m \rangle$ is a sequence obtained by deleting 0 or more elements from X.

Example: "sudan" is a subsequence of "sesquipedalian".

So is "equal".

There are 2^m subsequences of X.

A common subsequence Z of two sequences X and Y is a subsequence of both.

Example: "ua" is a common subsequence of "sudan" and "equal".

Longest Common Subsequence

Input: $X = < x_1, x_2, \ldots, x_m >$

$Y = < y_1, y_2, \ldots, y_n >$

Output: a *longest common subsequence* (LCS) of X and Y.

Example a) X = abcbdab Y = bdcaba

LCS_1 = bcba LCS_2 = bdab

b) X = enquiring Y = sequipedalian

LCS = equiin

c) X = empty bottle Y = nematode knowledge

LCS = emt ole

The Longest Common Subsequence (LCS) Problem

- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y

- Has applications to DNA similarity testing

A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.

- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

A Dynamic-Programming Approach to the LCS Problem

- Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively.

- And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y.

- Let's come up with a ***Recursive Definition of the Problem***

Recursive Definition of the Problem

- **Case 1:** If last characters of both sequences match $(X[m-1] == Y[n-1])$
  - ❖ Then $L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$
- Example

L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")

Recursive Definition of the Problem

- Case 2: If last characters of both sequences do not match (X[m-1] != Y[n-1])

  - Then
    L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

- Example

L("ABCDGH", "AEDFHR") = MAX (L("ABCDG", "AEDFH**R**"), L("ABCDG**H**", "AEDFH"))

Recursive Definition of the Problem

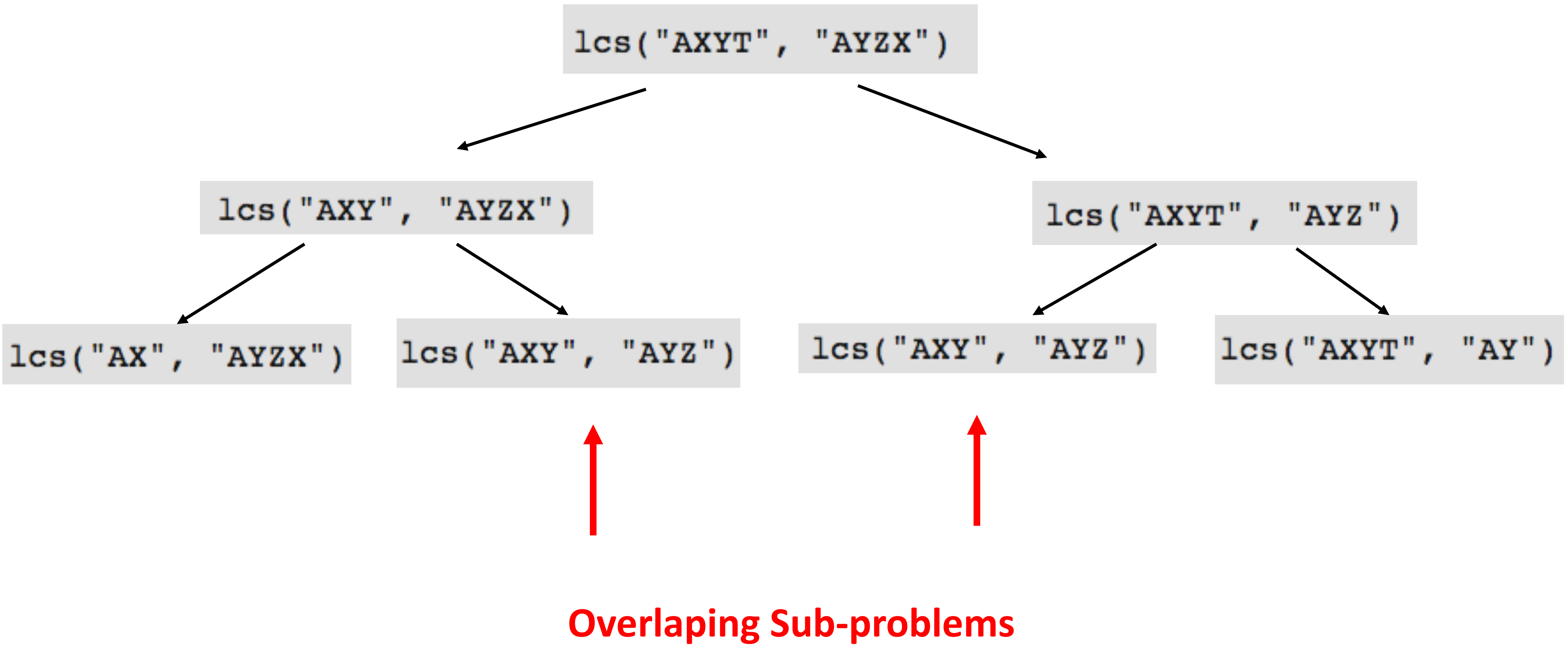
- Case 1: If last characters of both sequences match (X[m-1] == Y[n-1])

  - ❖ Then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

- Case 2: If last characters of both sequences do not match (or X[m-1] != Y[n-1])

  - ❖ Then
    L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Recursion Tree



Dynamic Programming Solution

```
/** Returns table such that L[j][k] is length of LCS for X[0..j−1] and Y[0..k−1]. */
public static int[ ][ ] LCS(char[ ] X, char[ ] Y) {
  int n = X.length;
  int m = Y.length;
  int[ ][ ] L = new int[n+1][m+1];
  for (int j=0; j < n; j++)
    for (int k=0; k < m; k++)
      if (X[j] == Y[k])              // align this match
        L[j+1][k+1] = L[j][k] + 1;
      else                           // choose to ignore one character
        L[j+1][k+1] = Math.max(L[j][k+1], L[j+1][k]);
  return L;
}
```

Goodrich and Tamasia – Chap: 12, P: 525

Analysis of DP LCS Algorithm

- We have two nested loops

  - The outer one iterates *n* times
  - The inner one iterates *m* times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is O(*nm*)

- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).