Sequence Alignment

Algoritmos em Bioinformática

Cátia Vaz

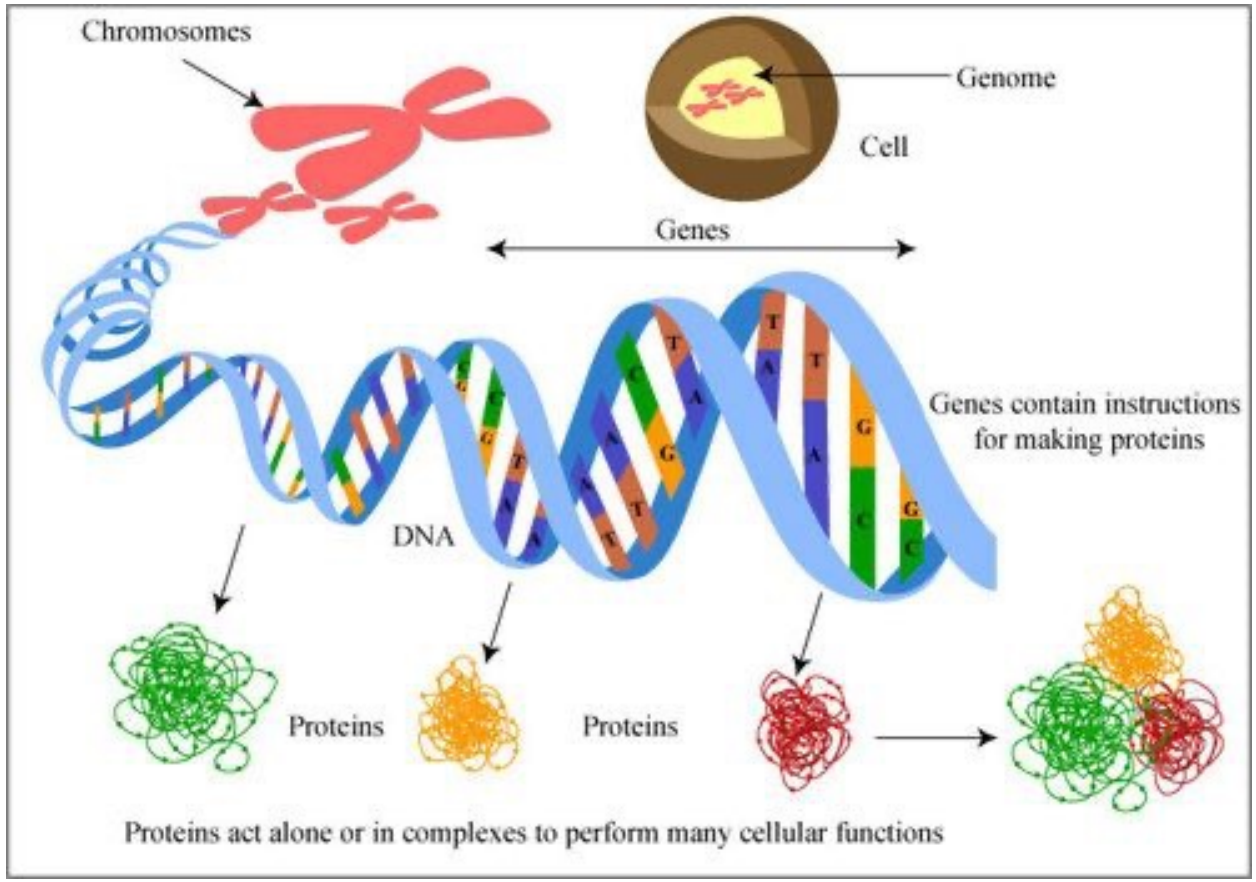
MEIC Semestre Verão 2023/2024

Adaptado do livro: Bioinformatics Algorithms
An Active Approach

Genome, Chromosomes and DNA, Genes

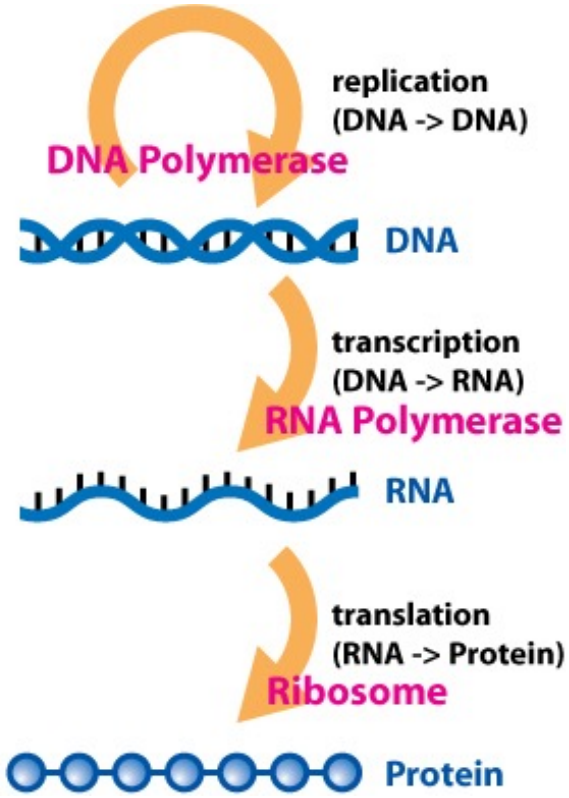
Four choices for the nucleic acid of a nucleotide:

1. Adenine (A)
2. Cytosine (C)
3. Guanine (G)—bonds to C
4. Thymine (T)—bonds to A



DNA and RNA

- a **gene** from a genome is first **transcribed** into a strand of RNA composed of four **ribonucleotides**:
  - Adenine (A),
  - Guanine (G) ,
  - Cytosine (C),
  - Uracil, (U).
- A strand of RNA can be represented as an **RNA string**, formed over the four-letter alphabet {A, C, G, U}
- Then, the RNA transcript is **translated** into an amino acid sequence of a protein.



Why sequencing alignment?

- **Sequence alignments** are useful in bioinformatics for identifying:
  - sequence similarity,
  - producing phylogenetic trees,
  - developing homology models of protein structures.
- Sequenced RNA, for instance, can be aligned to a sequenced genome to find where there are genes/proteins with common origin
  - Allows to predict function and structure
- In **Sequence Assembly,** for instance, it helps to locate sequences that might overlap

Example of application

- **platelet derived growth factor** (**PDGF**) gene and **v-sis gene** are very similar when comparing their sequence.
  - But their functions differ greatly;
    - the PDGF gene encodes a protein stimulating cell growth
    - the v-sis is an **oncogene**, or a gene in viruses that causes a cancer-like transformation of infected human cells.
    - **Conclusion**: some forms of cancer might be caused by a good gene doing the right thing at the wrong time

Comparing Genes is a Fundamental Problem in Biology

Comparing Genes Problem: *Compare two similar genes.*

- **Input:** Two genes.
- **Output:** How "similar" these genes are.

Goal: Convert this important biological question into a well-defined computational problem.

Try 1: Hamming Distance

Hamming Distance Problem: *Compute the number of mismatching symbols between two strings.*

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

Try 1: Hamming Distance

Hamming Distance Problem: *Compute the number of mismatching symbols between two strings.*

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

```
ATGCATGC
TGCATGCA
```

Hamming distance = 8

Try 1: Hamming Distance

Hamming Distance Problem: *Compute the number of mismatching symbols between two strings.*

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

```
ATGCATGC
TGCATGCA
```

Hamming distance = 8

STOP: What are the issues with this approach?

Try 1: Hamming Distance

Hamming Distance Problem: *Compute the number of mismatching symbols between two strings.*

- **Input:** Two strings.
- **Output:** The number of "mismatched" symbols in the two strings.

ATGCATGC
TGCATGCA

Hamming distance = 8

Note that these strings **have a long shared substring,** it just doesn't line up perfectly.

Try 2: Longest Substring

Longest Shared Substring Problem:

- **Input:** Two strings.
- **Output:** The longest substring shared by both substrings.

STOP: What are the weaknesses of using the length of a longest shared substring to represent the similarity between two strings?

Try 2: Longest Substring

Longest Shared Substring Problem:

- **Input:** Two strings.
- **Output:** The longest substring shared by both substrings.

Consider the strings AAACAAACAAACAAACAAACAAA and AAAGAAAGAAAGAAAGAAAGAAAGAAA. These strings are very similar, but they don't have a long shared substring in common.

Try 3: Counting Shared *k*-Mers

Instead of finding a longest shared substring of two strings, we will count the number of shared substrings.

For simplicity, we restrict to substrings of the same length; recall that a ***k*-mer** is the term for a string of length ***k***.

Try 3: Counting Shared *k*-Mers

s1 = ACGTATACACGTAT

String	Count
ACA	1
ACG	2
ATA	1
CAC	1
CGT	2
GTA	2
TAC	1
TAT	2

s2 = TATCGGTATATCCTAC

String	Count
ATA	1
ATC	2
CCT	1
CGG	1
CTA	1
GGT	1
GTA	1
TAC	1
TAT	3
TCC	1
TCG	1

STOP: How should we count the # of shared 3-mers of these two strings?

Try 3: Counting Shared *k*-Mers

s1 = ACGTATACACGTAT

String	Count
ACA	1
ACG	2
ATA	1
CAC	1
CGT	2
GTA	2
TAC	1
TAT	2

s2 = TATCGGTATATCCTAC

String	Count
ATA	1
ATC	2
CCT	1
CGG	1
CTA	1
GGT	1
GTA	1
TAC	1
TAT	3
TCC	1
TCG	1

Take minimum counts for each shared *k*-mer:

1 + 1 + 1 + 2 = 5

Toward a Better Approach

STOP: What similarities do you see in these strings?

ATGCTTA
TGCATTAA

Toward a Better Approach

STOP: What similarities do you see in these strings?

ATGCTTA
TGCATTAA

Key Point: we can find similarities if we "slide" the strings, letting symbols shift (but stay in same order).

A**TGC**-**TTA**-
-**TGC**A**TTA**A

Toward a More Accurate Problem

Symbol Matching Problem: *Match as many symbols as possible between two strings.*

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

```
ATGCTTA
TGCATTAA
```

```
ATGC-TTA-
-TGCATTAA
```

Toward a More Accurate Problem

Symbol Matching Problem: *Match as many symbols as possible between two strings.*

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

Exercise: How many matches can you find if the strings are `ATGTTATA` and `ATCGTCC`? What algorithm did you use?

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A **A**	T G T T A T A T C G T C C	**+1**

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A **A**	T G T T A T A T C G T C C	+1
A T **A T**	G T T A T A C G T C C	+1

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	

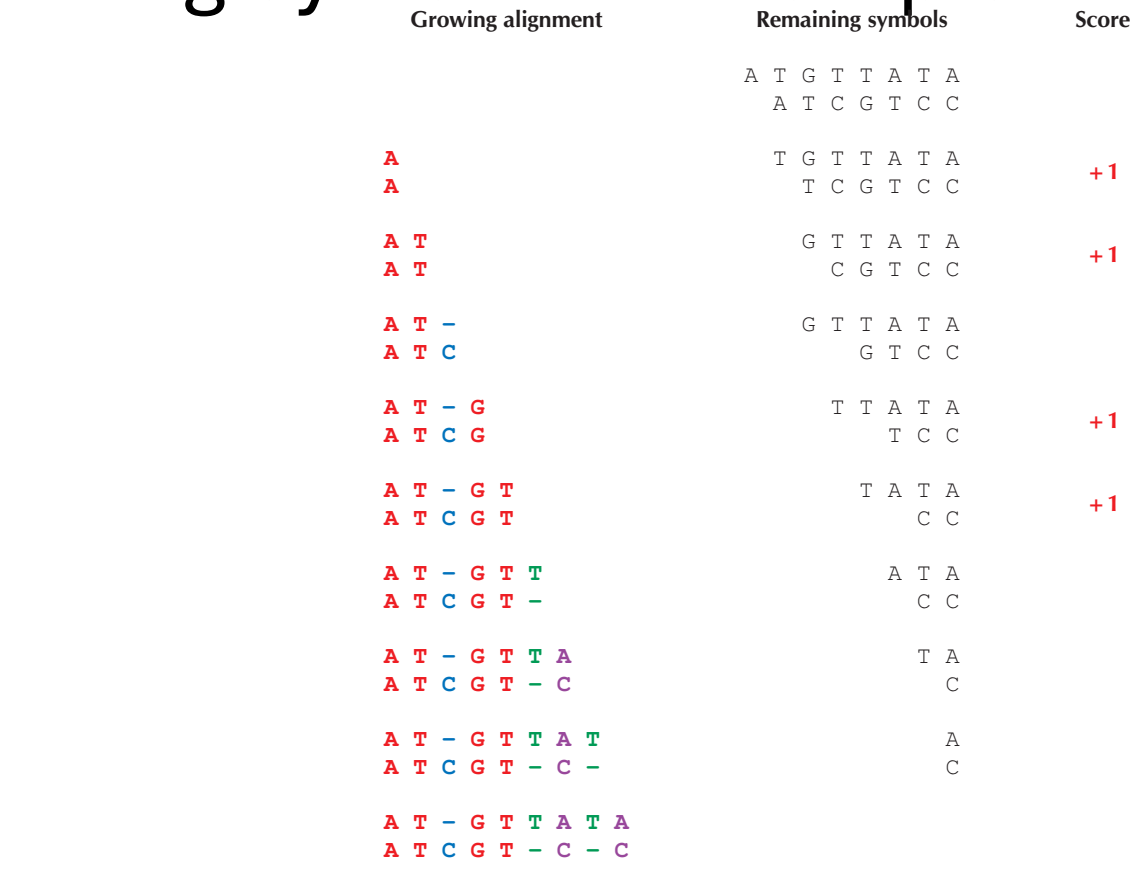
Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	
A T – G T T A A T C G T – C	T A C	

Matching Symbols of Example

Growing alignment	Remaining symbols	Score
	A T G T T A T A A T C G T C C	
A A	T G T T A T A T C G T C C	+1
A T A T	G T T A T A C G T C C	+1
A T – A T C	G T T A T A G T C C	
A T – G A T C G	T T A T A T C C	+1
A T – G T A T C G T	T A T A C C	+1
A T – G T T A T C G T –	A T A C C	
A T – G T T A A T C G T – C	T A C	
A T – G T T A T A T C G T – C –	A C	

Matching Symbols of Example



To a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")
- no column has two gap symbols

```
A T - G T T A T A
A T C G T - C - C
```

To a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")
- no column has two gap symbols

A T - G T T A T A
A T C **G T** - C - C

Matches

To a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")
- no column has two gap symbols

A T - G T T **A** T **A**
A T C G T - **C** - **C**

Mismatches

To a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")
- no column has two gap symbols

```
A T - G T T A T A
A T C G T - C - C
```

Insertions

To a Definition

Given two strings *v* and *w*, an **alignment** of *v* and *w* is a two-row matrix such that:

- the first row contains symbols of *v*
- the second row contains symbols of *w*
- each row may also contain **gap symbols** ("-")
- no column has two gap symbols

```
A T - G T T A T A
A T C G T - C - C
```

Deletions

Finding a Longest Common Subsequence

A **common subsequence** of *v* and *w* is a sequence of symbols occurring (not necessarily contiguously) in both *v* and *w*.

Finding a Longest Common Subsequence

A **common subsequence** of *v* and *w* is a sequence of symbols occurring (not necessarily contiguously) in both *v* and *w*.

The **matches** in an alignment of *v* and *w* form a common subsequence of *v* and *w*.

A T – **G T** T A T A

A T C **G T** – C – C

The Problems are the Same!

Longest Common Subsequence (LCS) Length Problem:.

- **Input:** Two strings.
- **Output:** The length of a longest common subsequence of these strings.

Symbol Matching Problem:

- **Input:** Two strings.
- **Output:** The greatest number of matched symbols in any "alignment" of the two strings.

Formally Formulate the problem of the Length of LCS

- Let X and Y two strings. Ex:

$$X = ATGCTTA$$
$$Y = TGCATTAA$$

- Let X_i the prefix of X of dimension i. Ex:

$$X_4 = ATGC$$

- LCS(i , j) = length of a longest common subsequence of X_i and Y_j

Example

```
X = ATGCTTA
    1234567
Y = TGCATTAA
    12345678
```

LCS(7,8) = 1 + LCS (6,7) “they share A”

= 1+ Max(LCS(6, 6) , LCS(5, 7))

1+LCS(5,5) , MAX(LCS(5,6), LCS(4,7)) **(…)**

1+LCS(4,4) **(…)**

MAX(LCS(4,3) , LCS(3,4))

1+LCS(3,2) **(…..)**

1+LCS(2,1)

1+ LCS(1,0)

0

Find a recursive formulation for the of the Length of LCS

$$LCS(X_i,Y_j)= \begin{cases} 0 & \text{,if } i=0 \text{ or } j=0 \\ 1 + LCS(X_{i-1},Y_{j-1}), & \text{if } x_i=y_j \\ MAX(LCS(X_i,Y_{j-1}), LCS(X_{i-1},Y_j)) & \text{,if } x_i \neq y_j \end{cases}$$

Example

X = ATGCTTA
1234567
Y = TGCATTAA
12345678

LCS(7,8) = 1 + LCS (6,7) “they share A”=1+ Max(LCS(6, 6), LCS(5, 7)) =6

LCS(6,6)= 1+LCS(5,5)=1+ 4=5

LCS(5,5)= 1+LCS(4,4) =4

LCS(4,4)= Max(LCS(4,3) , LCS(3,4))=3

LCS(4,3)= 1+LCS(3,2) =1+2=3

LCS(3,4)=MAX(LCS(3,3),LCS(2,4))=2

LCS(3,3)=MAX(LCS(3,2),LCS(1,3))=2

LCS(2,4)=MAX(LCS(2,3),LCS(1,4))=1

LCS(2,3)=MAX(LCS(2,2),LCS(1,3))=1

LCS(5,7)= MAX(LCS(5,6), LCS(4,7))=4

LCS(5,6)=1+ LCS(4,5) =4

LCS(4,5)=MAX(LCS(4,4),LCS(3,5))=3

LCS(3,5)=MAX(LCS(3,4),LCS(2,5))=2

LCS(2,5)=1+ LCS(1,4)=1+1=2

LCS(4,7)=MAX(LCS(3,7),LCS(4,6))=3

LCS(3,7)=MAX(LCS(3,6),LCS(2,7))=2

LCS(3,6)=MAX(LCS(3,5),LCS(2,6))=2

LCS(2,7)=MAX(LCS(2,6),LCS(1,7))=2

LCS(4,6)=MAX(LCS(4,5),LCS(3,6))=3

LCS(1,0)=0

LCS(0,2)=0

LCS(0,1)=0

LCS(0,7)=0

LCS(1,1)=MAX(LCS(0,1),LCS(1,0))=0

LCS(2,1) = 1+ LCS(1,0)=1

LCS(3,2)= 1+LCS(2,1)=2

LCS(1,2)=MAX(LCS(1,1),LCS(0,2))=0

LCS(1,3)=MAX(LCS(1,2),LCS(0,3))=0

LCS(1,4)=1+LCS(1,3)=1+0=1

LCS(3,2)=1+LCS(2,1)=1+1=2

LCS(1,7)=1+LCS(0,7)=1

LCS(1,5)=MAX(LCS(1,4),LCS(0,5))=1

LCS(2,2)=MAX(LCS(2,1),LCS(1,2))=1

LCS(2,6)=1+LCS(1,5)=2

Brute Force Approach for LCS Length

- Start at the first character of each string
- For every pair of characters, we have two choices:
  1. If the characters match, then the current LCS length will be 1 plus the LCS length of the remaining sections of both strings.
  2. If they don't match, the current LCS length is the maximum of the LCS lengths obtained by disregarding the current character of either string.

Example

X = ATGCTTA
1234567
Y = TGCATTAA
12345678

LCS(7,8) = 1 + LCS (6,7) “they share A”=1+ Max(LCS(6, 6), LCS(5, 7)) =6
LCS(6,6)= 1+LCS(5,5)=1+ 4=5
LCS(5,5)= 1+LCS(4,4) =4
LCS(4,4)= Max(LCS(4,3) , LCS(3,4))=3
LCS(4,3)= 1+LCS(3,2) =1+2=3
LCS(3,4)=MAX(LCS(3,3),LCS(2,4))=2
LCS(3,3)=MAX(LCS(3,2),LCS(1,3))=2
LCS(2,4)=MAX(LCS(2,3),LCS(1,4))=1
LCS(2,3)=MAX(LCS(2,2),LCS(1,3))=1
LCS(5,7)= MAX(LCS(5,6), LCS(4,7))=4
LCS(5,6)=1+ LCS(4,5) =4
LCS(4,5)=MAX(LCS(4,4),LCS(3,5))=3
LCS(3,5)=MAX(LCS(3,4),LCS(2,5))=2
LCS(2,5)=1+ LCS(1,4)=1+1=2
LCS(4,7)=MAX(LCS(3,7),LCS(4,6))=3
LCS(3,7)=MAX(LCS(3,6),LCS(2,7))=2
LCS(3,6)=MAX(LCS(3,5),LCS(2,6))=2
LCS(2,7)=MAX(LCS(2,6),LCS(1,7))=2
LCS(4,6)=MAX(LCS(4,5),LCS(3,6))=3

How much time is calculated LCS(3,2)?

LCS(1,0)=0
LCS(0,2)=0
LCS(0,1)=0
LCS(0,7)=0
LCS(1,1)=MAX(LCS(0,1),LCS(1,0))=0
LCS(2,1) = 1+ LCS(1,0)=1
LCS(3,2)= 1+LCS(2,1)=2
LCS(1,2)=MAX(LCS(1,1),LCS(0,2))=0
LCS(1,3)=MAX(LCS(1,2),LCS(0,3))=0
LCS(1,4)=1+LCS(1,3)=1+0=1
LCS(3,2)=1+LCS(2,1)=1+1=2
LCS(1,7)=1+LCS(0,7)=1
LCS(1,5)=MAX(LCS(1,4),LCS(0,5))=1
LCS(2,2)=MAX(LCS(2,1),LCS(1,2))=1
LCS(2,6)=1+LCS(1,5)=2

Example

X = ATGCTTA
1234567
Y = TGCATTAA
12345678

LCS(7,8) = 1 + LCS (6,7) “they share A”=1+ Max(LCS(6, 6), LCS(5, 7)) =6

LCS(6,6)= 1+LCS(5,5)=1+ 4=5

LCS(5,5)= 1+LCS(4,4) =4

LCS(4,4)= Max(LCS(4,3) , LCS(3,4))=3

LCS(4,3)= 1+LCS(3,2) =1+2=3

LCS(3,4)=MAX(LCS(3,3),LCS(2,4))=2

LCS(3,3)=MAX(LCS(3,2),LCS(1,3))=2

LCS(2,4)=MAX(LCS(2,3),LCS(1,4))=1

LCS(2,3)=MAX(LCS(2,2),LCS(1,3))=1

LCS(5,7)= MAX(LCS(5,6), LCS(4,7))=4

LCS(5,6)=1+ LCS(4,5) =4

LCS(4,5)=MAX(LCS(4,4),LCS(3,5))=3

LCS(3,5)=MAX(LCS(3,4),LCS(2,5))=2

LCS(2,5)=1+ LCS(1,4)=1+1=2

LCS(4,7)=MAX(LCS(3,7),LCS(4,6))=3

LCS(3,7)=MAX(LCS(3,6),LCS(2,7))=2

LCS(3,6)=MAX(LCS(3,5),LCS(2,6))=2

LCS(2,7)=MAX(LCS(2,6),LCS(1,7))=2

LCS(4,6)=MAX(LCS(4,5),LCS(3,6))=3

How much time is calculated LCS(1,2)?

LCS(1,0)=0

LCS(0,2)=0

LCS(0,1)=0

LCS(0,7)=0

LCS(1,1)=MAX(LCS(0,1),LCS(1,0))=0

LCS(2,1) = 1+ LCS(1,0)=1

LCS(3,2)= 1+LCS(2,1)=2

LCS(1,2)=MAX(LCS(1,1),LCS(0,2))=0

LCS(1,3)=MAX(LCS(1,2),LCS(0,3))=0

LCS(1,4)=1+LCS(1,3)=1+0=1

LCS(1,7)=1+LCS(0,7)=1

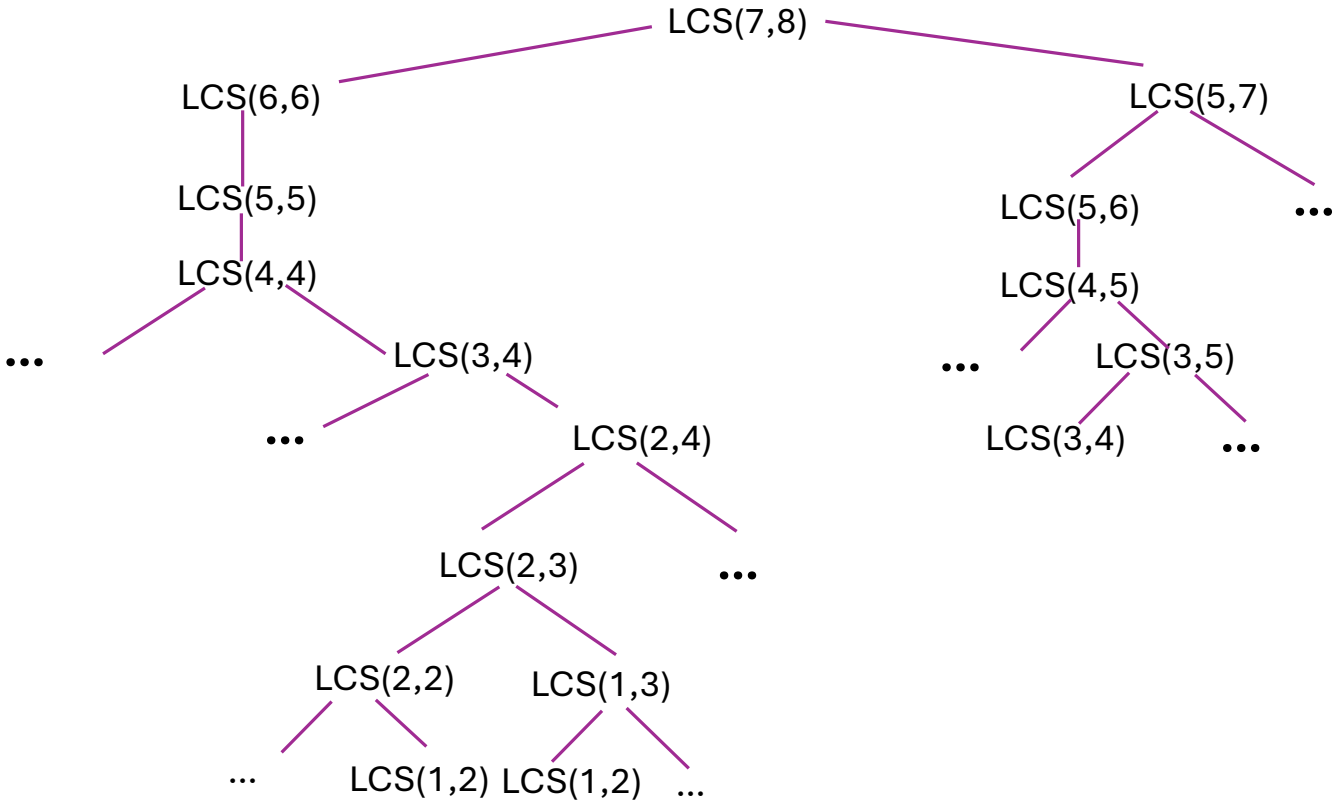
LCS(1,5)=MAX(LCS(1,4),LCS(0,5))=1

LCS(2,2)=MAX(LCS(2,1),LCS(1,2))=1

LCS(2,6)=1+LCS(1,5)=2

LCS(6,7)= Max(LCS(6, 6), LCS(5, 7)) =5

Recursive Calls Can Grow Exponentially



When X = ATGCTTA and Y = TGCATTAA, LCS(1,2) is computed 4 times.

Brute Force Approach for LCS Length

- Start at the first character of each string
- For every pair of characters, we have two choices:
  1. If the characters match, then the current LCS length will be 1 plus the LCS length of the remaining sections of both strings.
  2. If they don't match, the current LCS length is the maximum of the LCS lengths obtained by disregarding the current character of either string.

How to change it to get the Longest Common Subsequence?

Depending on the implementation:
Time Complexity $O(2^{n*m})$ or $O(2^{n+m})$

Overlapping subproblems

- The recursive solution contains a “small” number of distinct subproblems repeated many times
- The number of distinct LCS subproblems for two strings of lengths m and n is only **mn**

Dynamic Programming

The Longest Common Subsequence (LCS) problem exhibits two main characteristics that make it an ideal candidate for optimization using dynamic programming, namely:

1. **Optimal Substructure**: The optimal solution for the LCS problem can be constructed from the optimal solutions of its subproblems. If we know the longest common subsequence of the two substrings, we can easily figure out the LCS of the two whole strings.
2. **Overlapping Subproblems**: If we draw a recursion tree for brute force solution, you will see that many subproblems are solved multiple times in a simple recursive solution. This redundancy leads to unnecessary computation and poor performance.

Dynamic Programming versus Divide and Conquer

- **Dynamic programming**, like the **Divide and Conquer technique**, solves problems by combining the solutions of subproblems.
- However **Dynamic programming**, it applies when subproblems overlap. Therefore, with dynamic programming, we only calculate the solution to the subproblem once and memorize it:
  - top-down approach: memorization;
  - down-top approach: tabulation.

How to apply dinamic programming?
A simple example: Fibonnacci number

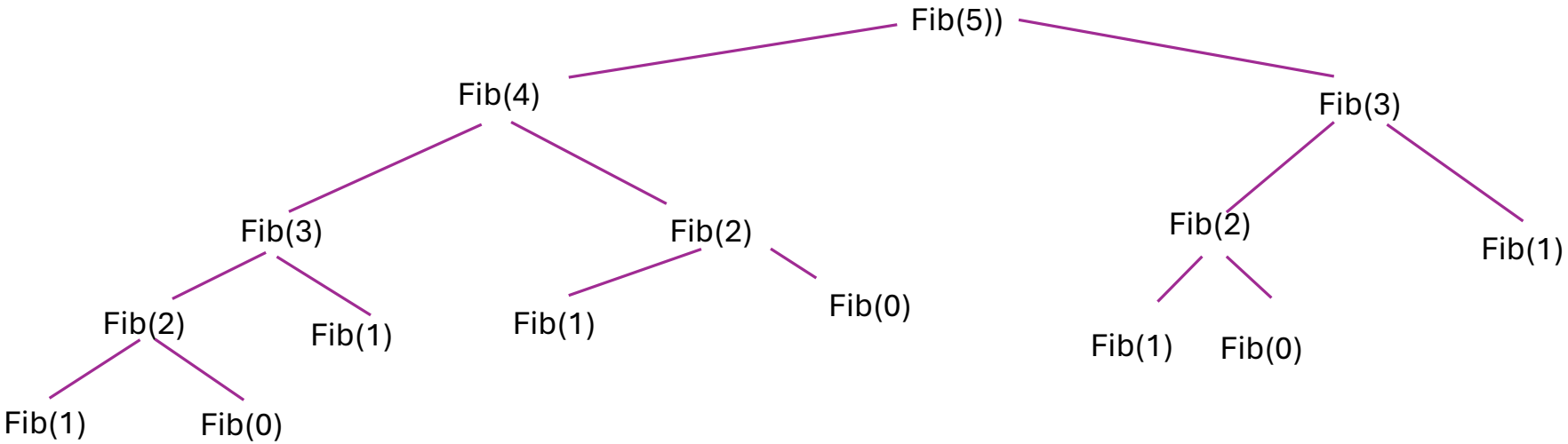
Fibonnacci Sequence:

$$F(n)=\begin{cases} 0 & , n=0 \\ 1 & , n=1 \\ F(n-1)+F(n-2) & , n>1 \end{cases}$$

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Temporal Complexity of Solution?

Recursive calls for Fib(5)



Time Complexity:
Exponencial! $O(2^n)$

Dynamic Programming Soluction

- Key Idea: Use an array to store que processed numbers until now!

```
def fibonacci(n):

    # Taking 1st two fibonacci numbers as 0 and 1
    f = [0, 1]

    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

Dynamic Programming Soluction

- Key Idea: Use an array to store que processed numbers until now!

```
def fibonacci(n):

    # Taking 1st two fibonacci numbers as 0 and 1
    f = [0, 1]

    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

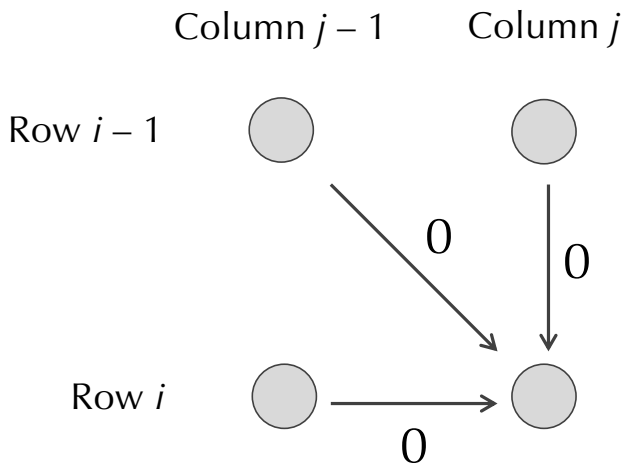
Time Complexity: O(n).
Can we do better?

Formally Formulate the problem of LCS length

		0	1	2	3	4	5	6	7	8
			T	G	C	A	T	T	A	A
0		0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	0	0	1	
2	T	0	1	1	1	1	2	2	2	
3	G	0		2	2	2	2	2	2	
4	C	0			3	3	3	3	3	
5	T	0					4	4	4	
6	T	0						5	5	
7	A	0								6

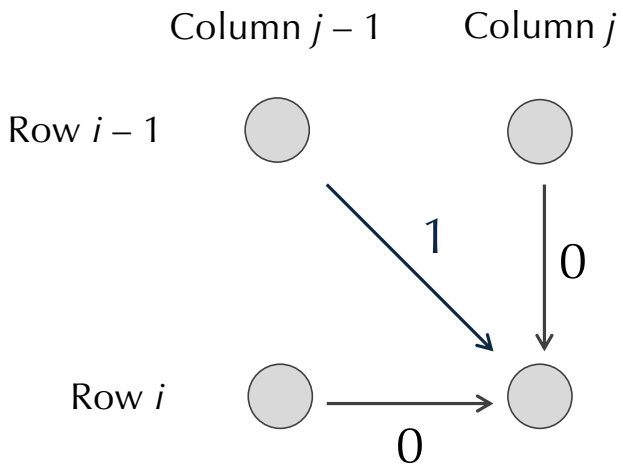
Our Recurrence Has Two Cases

Case 1



length(i, j) = maximum of:
- *length*$(i-1, j) + 0$
- *length*$(i, j-1) + 0$
- *length*$(i-1, j-1) + 0$

Case 2



length(i, j) = maximum of:
- *length*$(i-1, j) + 0$
- *length*$(i, j-1) + 0$
- *length*$(i-1, j-1) + 1$

Formally Formulate the problem of LCS length

		0	1	2	3	4	5	6	7	8
			T	G	C	A	T	T	A	A
0		0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	0	0	1	
2	T	0	1	1	1	1	2	2	2	
3	G	0		2	2	2	2	2	2	
4	C	0			3	3	3	3	3	
5	T	0					4	4	4	
6	T	0						5	5	
7	A	0								6

A diagonal jump means that we found an element of the LCS!

Solution : top-down approach

Once we've filled this in, we can work backwards.

```
def lcs(X, Y, m, n, dp):
  if (m == 0 or n == 0):
        return 0

  if (dp[m][n] != -1):
        return dp[m][n]

  if X[m - 1] == Y[n - 1]:
        dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp)
        return dp[m][n]

  dp[m][n] = max(lcs(X, Y, m, n - 1, dp), lcs(X, Y, m - 1, n, dp))
  return dp[m][n]
```

Time complexity: O(mn).

Solution : botton up approach

```
def lcs(X, Y, m, n):
    # Declaring the array for storing the dp values
    L =  [[None]*(n+1) for i in range(m+1)]

    # Following steps build L[m+1][n+1] in bottom up fashion
    # Note: L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```