

# Dynamic Programming:

Longest Common Subsequences

1

# Application: Alignment

DNA

# Sequence

DNA sequences can be

viewed

as strings of

**A**, **C**,

**G**, and

**T** characters,

which represent

nucleotides.

Finding

the

similarities

between two

DNA

sequences is

an important

computation

performed in bioinformatics.

* For

instance, when

comparing

the DNA of

different organisms, such alignments can highlight

the

locations where those

organisms have

identical DNA patterns.



# Application: Alignment

DNA

# Sequence

Finding the best alignment

between two

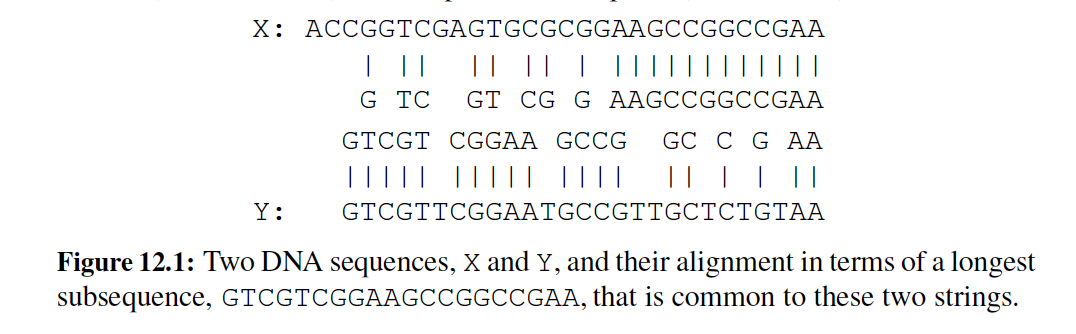
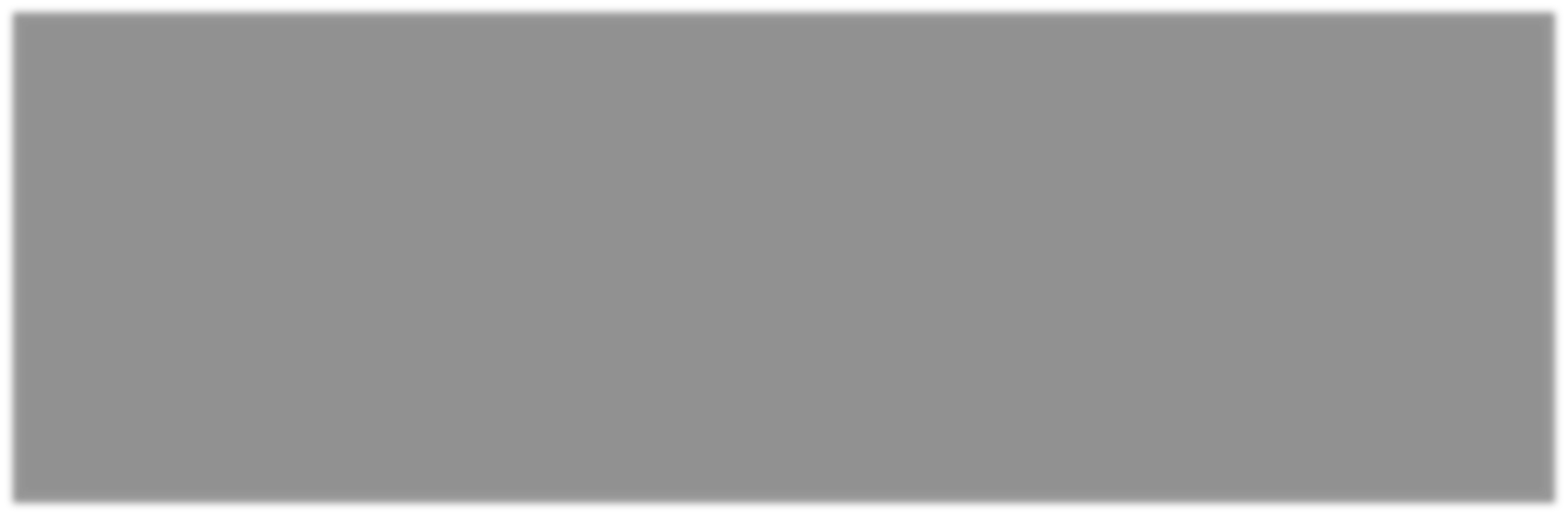
DNA strings

involves

minimizing

the number of changes to

convert one string to the other.



A brute-force

search would take exponential

time,

but we can do much better using

**programming**.

**dynamic**



# The General Dynamic Programming Technique

Applies to a problem that

at first seems to

require a lot provided we

of time have:

(possibly exponential),

* **Simple subproblems:** the subproblems can be



defined in terms of a few variables, such as j, k, l, m, and so on.

* **Subproblem optimality:** the global optimum value

can be defined in terms of optimal subproblems

* **Subproblem overlap:** the subproblems are not

independent, but instead they overlap (hence,

should be constructed bottom-up).

# Subsequences

A **subsequence** of

a character

string

x0x1x2…xn-1 is a string

of the

form

xi1xi2…

xik, where ij < ij+1.

Not the same as substring!

Example String: ABCDEFGHIJK

* + Subsequence:
  + Subsequence:

ACEGJIK DFGHK

* + Not subsequence: DAGH



# 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 (alphabet is {A,C,G,T})

Example: ABCDEFG and XZACKDFWGH

have ACDFG as a longest

common

subsequence



# A Poor Approach to the

LCS Problem

A Brute-force solution:

* Enumerate all subsequences of X
* Test which ones are also
* Pick the longest one.

Analysis:

subsequences of Y

* If

X is

of length

n, then it

has 2n

subsequences

* This is an

exponential-time

algorithm!



# A Dynamic-Programming

Approach to the LCS Problem

Define L[i,j] to be the length of the

longest

common

subsequence of X[0..i] and

Y[0..j].

Allow for -1 as an index, so L[-1,k] = 0 and L[k,-1]=0, to

indicate that the null part of X or Y has no match with the other.

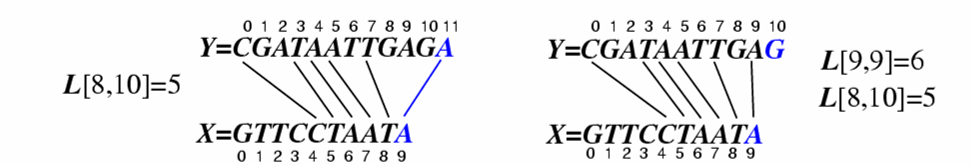
Then

we can

define L[i,j] in the general case as follows:

1. If xi=yj, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
2. If xi≠yj, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we match here)

have no

Case 1: Case 2:



# An LCS Algorithm

**Algorithm** LCS(X,Y ):

**Input:**

Strings X

and Y with n and

m elements, respectively

**Output:** For i = 0,…,n-1, j

= 0,...,m-1, the length L[i, j] of a longest

string

that is a subsequence of both the string X[0..i] = x0x1x2…xi

and the

string Y [0.. j] = y0y1y2…yj

**for** i =1 to n-1 **do**

L[i,-1] = 0

**for** j =0 to m-1 **do**

L[-1,j] = 0

**for** i =0 to n-1 **do**

**for** j =0 to m-1 **do**

**if** xi = yj

**else**

**then**

L[i, j]

= L[i-1, j-1] + 1

**return** array L

L[i, j]

= max{L[i-1, j] , L[i,

j-1]}



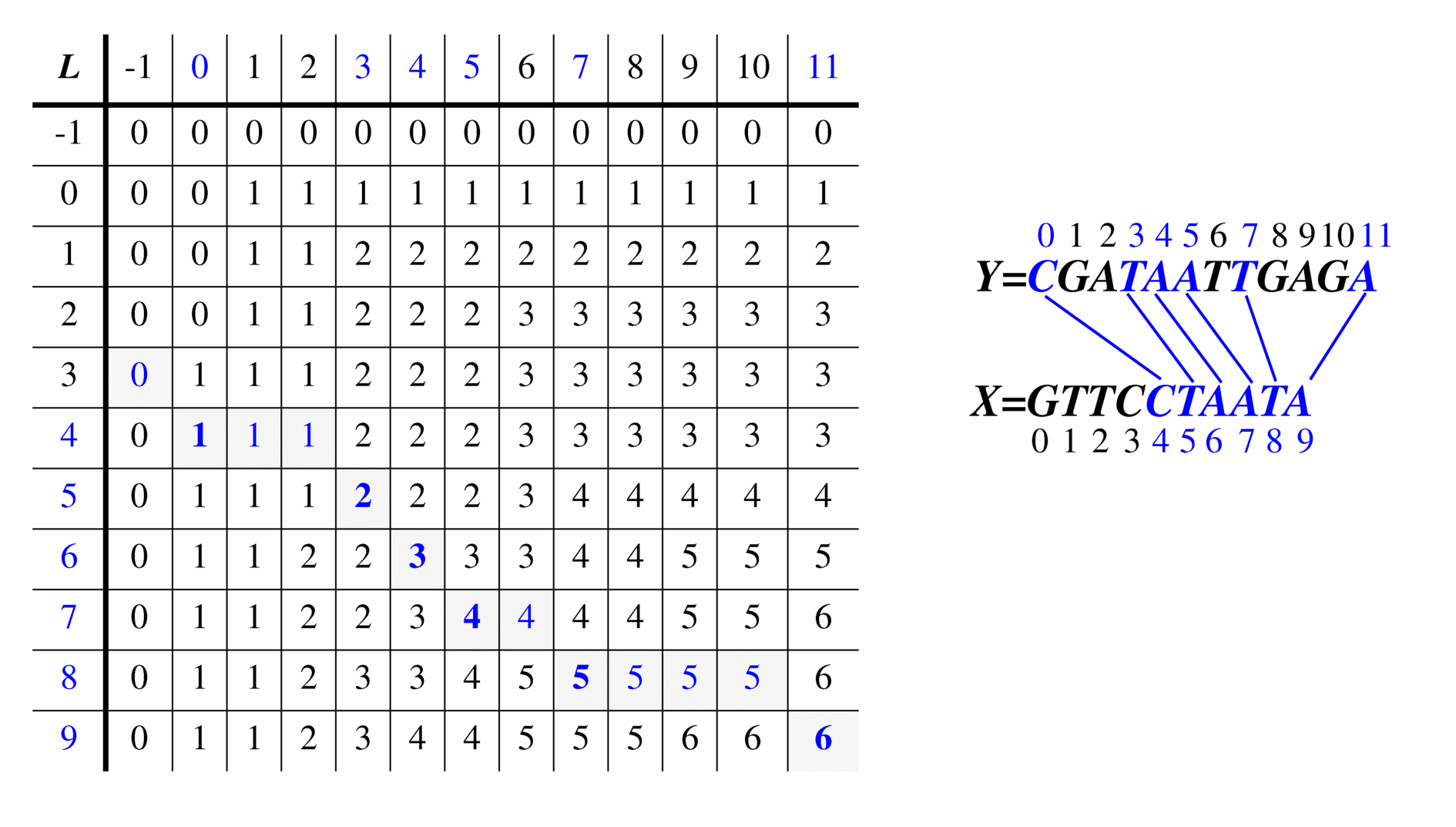
# Visualizing

the

LCS

# Algorithm





# Analysis of

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).