

# Dynamic Programming for Sequence Alignment

# An Important Algorithm Design Technique

## Dynamic Programming

Give a solution to a problem using smaller sub-problems, e.g. a recursive solution

Useful when the same sub-problems show up again and again in the solution

# Sequence Similarity

G G A C C A

T A C T A A G

T C C A A T

# Sequence Similarity: What

G G A C C A

T A C T A A G

| : | : | | :

T C C - A A T

# Sequence Similarity: Why

Most widely used comp. tools in biology

New sequence always compared to  
sequence data bases

**Similar sequences often have similar  
origin or function**

Recognizable similarity after  $10^8 - 10^9$  yr

not to mention:

- Unix “diff”,
- module histories in version control systems,
- programming assignment pairs with sadly questionable evolutionary history,
- etc., etc., ...

# Terminology

*String*: ordered list of letters TATAAG

*Prefix*: consecutive letters from front  
empty, T, TA, TAT, ...

*Suffix*: ... from end  
empty, G, AG, AAG, ...

*Substring*: ... from ends or middle  
empty, TAT, AA, ...

*Subsequence*: ordered, nonconsecutive  
TT, AAA, TAG, ...

# Sequence Alignment

a c b c d b  
  /  \  
c a d b d

a c - - b c d b  
  |          |  |  
- c a d b - d -

**Defn:** An *alignment* of strings  $S$ ,  $T$  is a pair of strings  $S'$ ,  $T'$  (with spaces or '-') s.t.

(1)  $|S'| = |T'|$ , and  $(|S| = \text{"length of } S\text{"})$

(2) removing all spaces leaves  $S$ ,  $T$

# Alignment Scoring

E.g.:  
Mismatch = -1  
Match = 2

a c b c d b  
c a d b d d

a c - - b c d b  
- c a d b - d d  
-1 2 -1 -1 2 -1 2 -1

Value = 3\*2 + 5\*(-1) = +1

The *score* of aligning (characters or spaces) x & y is  $\sigma(x,y)$ .

← in general

*Value* of an alignment  $\sum_{i=1}^{|S'|} \sigma(S'[i], T'[i])$

An *optimal alignment*: one of max value



# Analysis of brute force

Assume  $|S| = |T| = n$

Time to evaluate one alignment:  $O(n)$

How many alignments are there:

S = abcde  
T = vwxyz

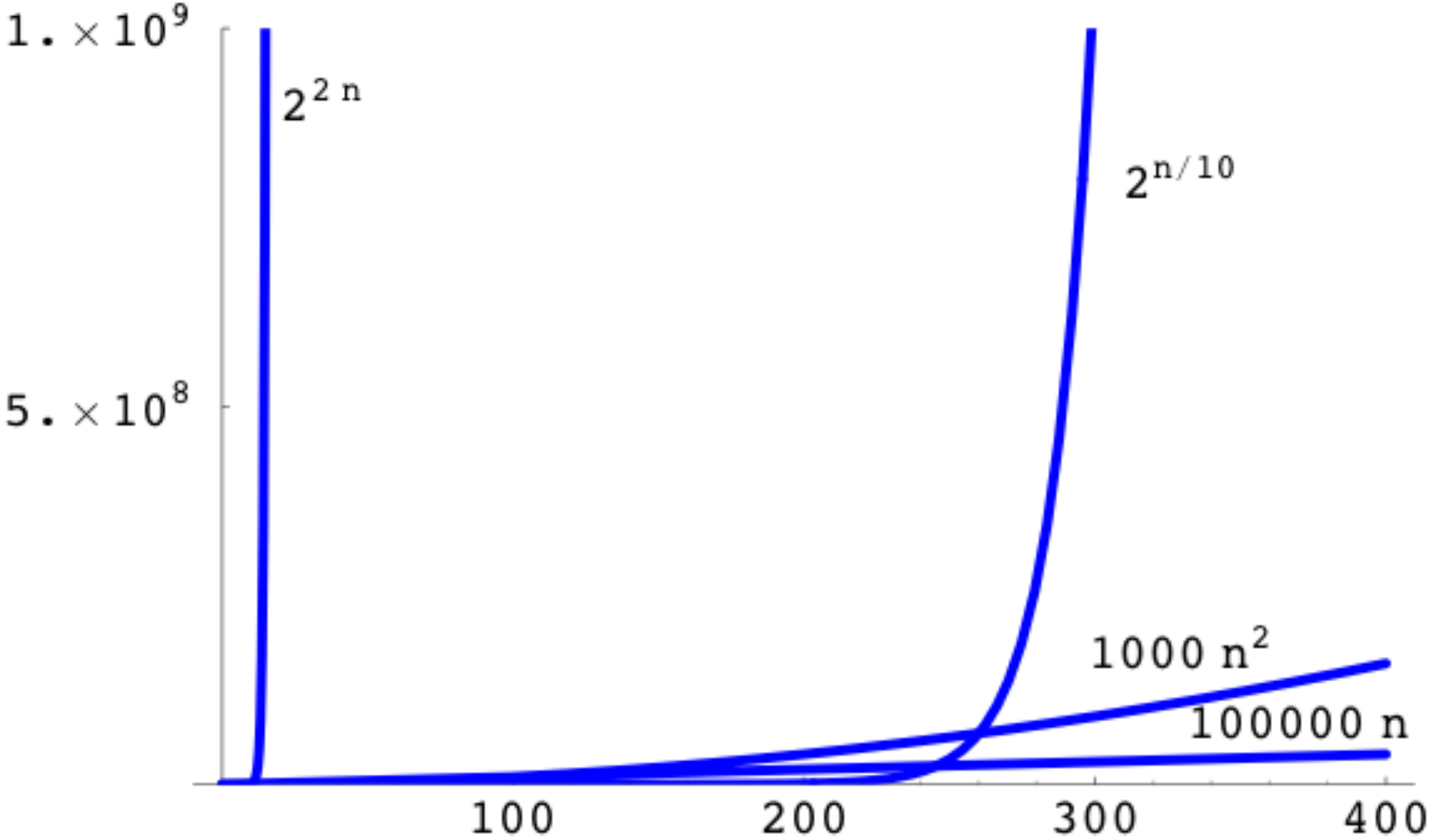
$$\sum_{k=0}^n \binom{n+k}{k} \binom{n}{k}$$

--abc-de      ab--c-de  
vw---xyz-      ---vwxyz-  
(two alignments, but same value)

Total time more than  $2^{2n}$ , for  $n > 3$

E.g., for  $n = 20$ , time is  $> 2^{40} > 10^{12}$  operations

# Polynomial vs Exponential Growth



# Dynamic Programming Alg: The Key Idea

Optimal alignment *ends* in 1 of 3 ways:

last chars of S & T aligned with each other

last char of S aligned with space in T

last char of T aligned with space in S

( never align space with space;  $\sigma(-, -) < 0$  )

In each case, the *rest* of S & T should be

*optimally* aligned to each other

(else you could improve by doing so)

# Optimal Alignment in $O(n^2)$ via “Dynamic Programming”

Input:  $S, T, |S| = n, |T| = m$

Output: **value** of optimal alignment

Common: first solve for value of opt.

$V(i,j)$  = value of optimal alignment of  
 $S[1], \dots, S[i]$  with  $T[1], \dots, T[j]$   
for **all**  $0 \leq i \leq n, 0 \leq j \leq m$ .

# Base Cases

$V(i,0)$ : first  $i$  chars of  $S$  all match spaces

$$V(i,0) = \sum_{k=1}^i \sigma(S[k], -)$$

$V(0,j)$ : first  $j$  chars of  $T$  all match spaces


$$V(0,j) = \sum_{k=1}^j \sigma(-, T[k])$$

# General Case

Opt align of  $S[1], \dots, S[i]$  vs  $T[1], \dots, T[j]$ :

$$\left[ \begin{array}{c} \sim\sim\sim\sim T[j] \\ \sim\sim\sim\sim S[i] \end{array} \right], \quad \left[ \begin{array}{c} \sim\sim\sim\sim T[j] \\ \sim\sim\sim\sim - \end{array} \right], \quad \text{or} \quad \left[ \begin{array}{c} \sim\sim\sim\sim - \\ \sim\sim\sim\sim S[i] \end{array} \right]$$

Opt align of  
 $S_1 \dots S_{i-1}$  &  
 $T_1 \dots T_{j-1}$

$$V(i,j) = \max \left\{ \begin{array}{l} V(i-1, j-1) + \sigma(S[i], T[j]) \\ V(i-1, j) + \sigma(S[i], -) \\ V(i, j-1) + \sigma(-, T[j]) \end{array} \right\},$$


for all  $1 \leq i \leq n, 1 \leq j \leq m$ .

Mismatch = -1  
Match = 2

# Example

i \ j	0	1	2	3	4	5
0	0	-1	-2	-3	-4	-5
1	-1					
2	-2					
3	-3					
4	-4					
5	-5					
6	-6					

← T

↑ S

c
-

 Score(c,-) = -1

Mismatch = -1  
Match = 2

# Example

i \ j	0	1	2	3	4	5
0	0	-1	-2	-3	-4	-5
1	a	-1				
2	c	-2				
3	b	-3				
4	c	-4				
5	d	-5				
6	b	-6				

← T

↑ S

-  
a      Score(-,a) = -1



Mismatch = -1  
Match = 2

# Example

	j	0	1	2	3	4	5
i			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1					
2	c	-2					
3	b	-3					
4	c	-4					
5	d	-5					
6	b	-6					

← T

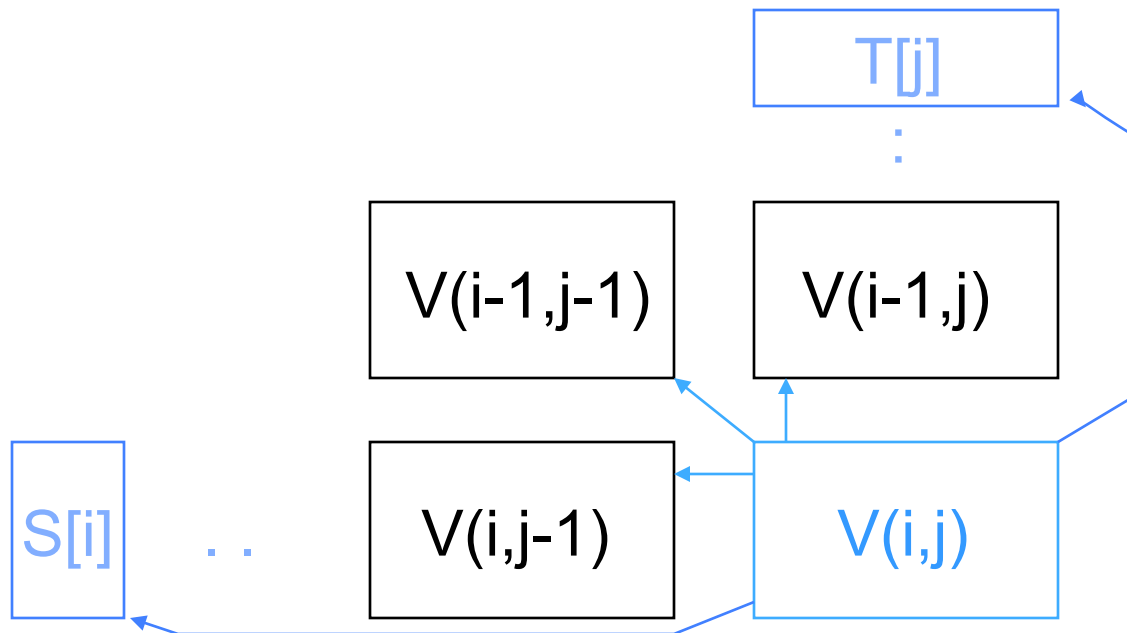
-	-
a	c
-1	

Score(-,c) = -1

↑ S

# Calculating One Entry

$$V(i,j) = \max \left\{ \begin{array}{l} V(i-1,j-1) + \sigma(S[i],T[j]) \\ V(i-1,j) + \sigma(S[i], -) \\ V(i,j-1) + \sigma(-, T[j]) \end{array} \right\}$$



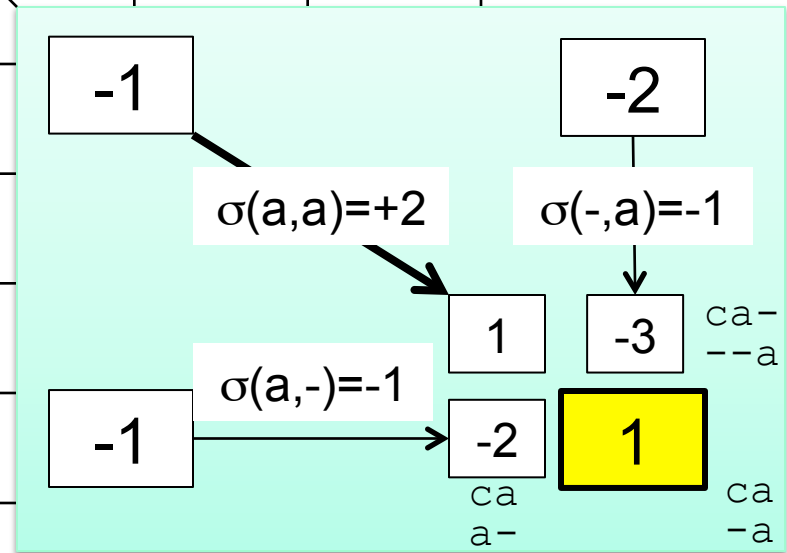
Mismatch = -1  
Match = 2

# Example

i \ j	0	1	2	3	4	5
		c	a	d	b	d
0	0	-1	-2	-3	-4	-5
1	a	-1	-1	1		
2	c	-2				
3	b	-3				
4	c	-4				
5	d	-5				
6	b	-6				

← T

↑ S



Mismatch = -1  
 Match = 2

# Example

	j	0	1	2	3	4	5
i			c	a	d	b	d
0		0	-1	-2	-3	-4	-5
1	a	-1	-1	1			
2	c	-2	1				
3	b	-3					
4	c	-4					
5	d	-5					
6	b	-6					

←T

Time =  
 $O(mn)$

↑S

Mismatch = -1  
Match = 2

# Example

	j	0	1	2	3	4	5	
i			c	a	d	b	d	←T
0		0	-1	-2	-3	-4	-5	
1	a	-1	-1	1	0	-1	-2	
2	c	-2	1	0	0	-1	-2	
3	b	-3	0	0	-1	2	1	
4	c	-4	-1	-1	-1	1	1	
5	d	-5	-2	-2	1	0	3	
6	b	-6	-3	-3	0	3	2	

↑S

# Finding Alignments: Trace Back

Arrows = (ties for) max in  $V(i,j)$ ; 3 LR-to-UL paths = 3 optimal alignments

	j	0	1	2	3	4	5	
i			c	a	d	b	d	←T
0		0	-1	-2	-3	-4	-5	
1	a	-1	-1	1	0	-1	-2	
2	c	-2	1	0	0	-1	-2	
3	b	-3	0	0	-1	2	1	
4	c	-4	-1	-1	-1	1	1	
5	d	-5	-2	-2	1	0	3	
6	b	-6	-3	-3	0	3	2	

↑S

# Complexity Notes

Time =  $O(mn)$ , (value and alignment)

Space =  $O(mn)$

Easy to get **value** in Time =  $O(mn)$  and  
Space =  $O(\min(m,n))$

Possible to get value *and alignment* in  
Time =  $O(mn)$  and Space =  $O(\min(m,n))$   
but tricky.

# Summary

Sequence similarity has important

Surprisingly simple scoring often works well in practice: score positions separately & add

Simple “dynamic programming” algorithms can find *optimal* alignments under these assumptions in polynomial time (product of sequence lengths)

Keys to D.P. are to

- a) identify the subproblems (usually repeated/overlapping)
- b) be sure opt solutions to subproblems are needed for opt solution globally
- c) solve in a careful order; solve all small ones before needed by bigger ones
- d) build table with solutions to the smaller ones so bigger ones just need to do table lookups (*no* recursion, despite recursive formulation implicit in (a))