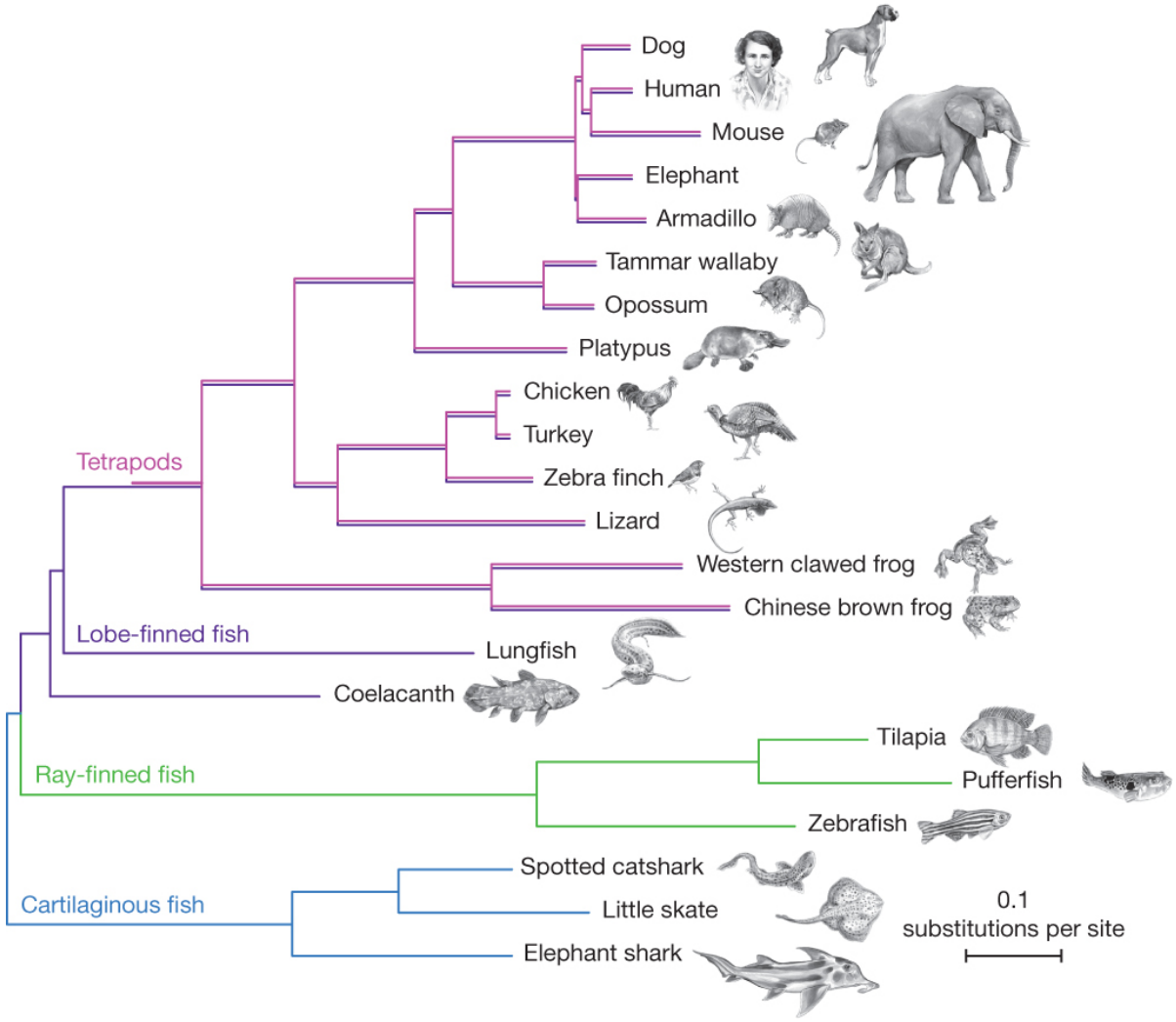
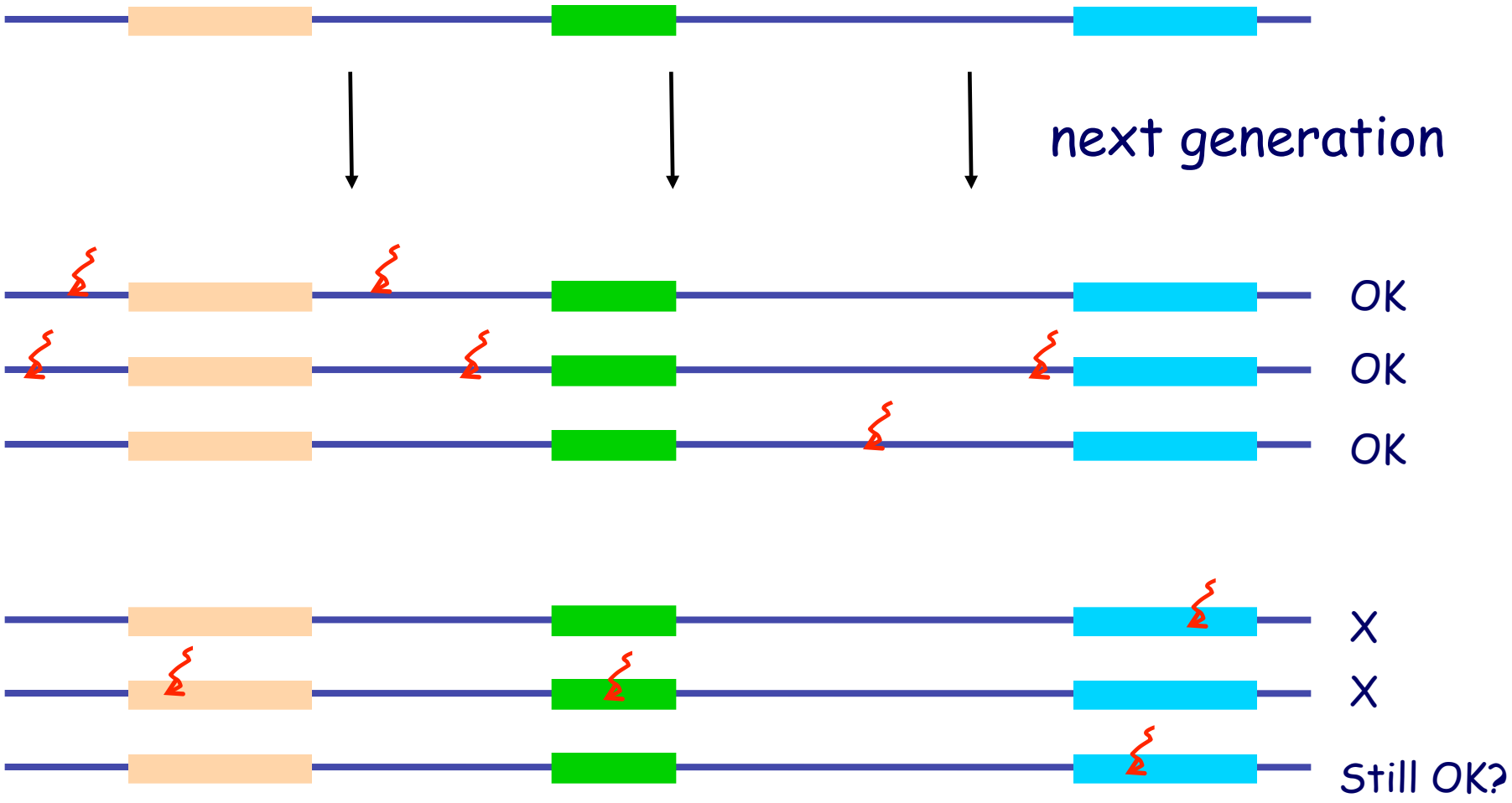
Sequence Alignment

Evolution

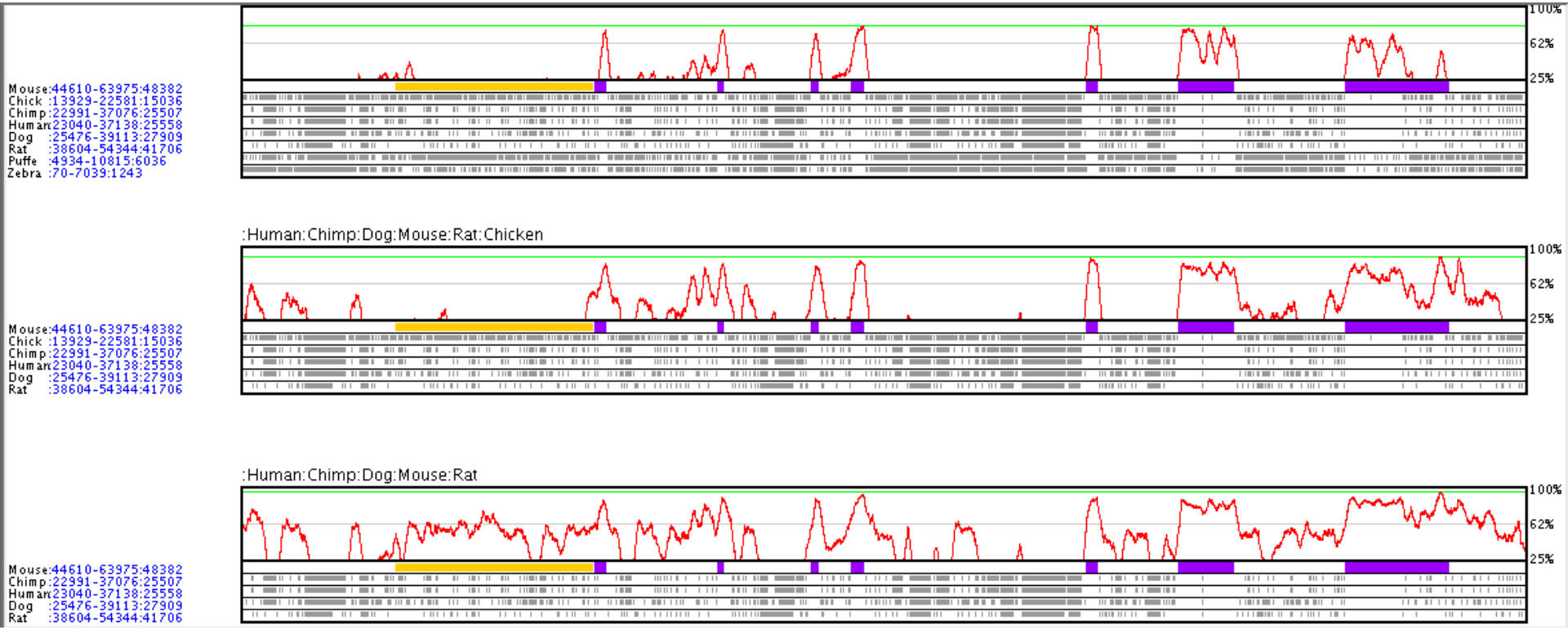


CT Amemiya *et al. Nature* **496**, 311-316 (2013)
doi:10.1038/nature12027

Evolutionary Rates

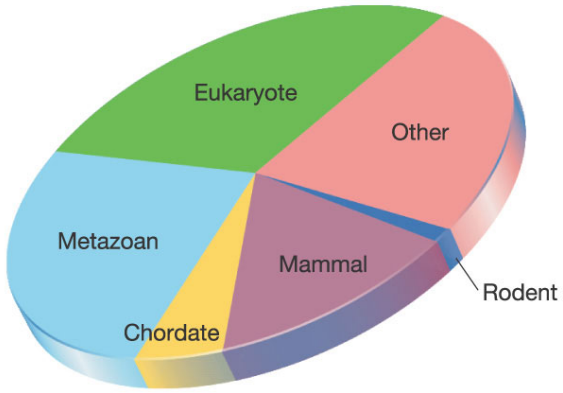


Sequence conservation implies function

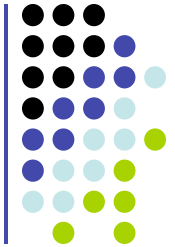


Alignment is the key to

- Finding important regions
- Determining function
- Uncovering evolutionary events



Sequence Alignment



AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

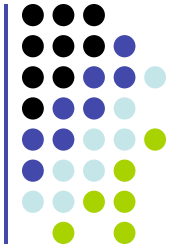
-AGG**CTATCAC**CT**GACC**T**C**CA**GG**C**CGA--TGCCC**---
T**AG-CTATCAC--GACC**G**C--GG**T**CGA**TT**TGCCC**GAC

Definition

Given two strings $x = x_1x_2...x_M$, $y = y_1y_2...y_N$,

an alignment is an assignment of gaps to positions 0,…, N in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

What is a good alignment?



```
AGGCTAGTT,
AGCGAAGTTT
```

```
AGGCTAGTT-
AGCGAAGTTT
```

6 matches, 3 mismatches, 1 gap

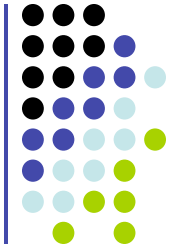
```
AGGCTA-GTT-
AG-CGAAGTTT
```

7 matches, 1 mismatch, 3 gaps

```
AGGC-TA-GTT-
AG-CG-AAGTTT
```

7 matches, 0 mismatches, 5 gaps

Scoring Function



- Sequence edits:

  AGGCCTC

  - Mutations **AGGACTC**
  - Insertions **AGGGCCTC**
  - Deletions **AGG . CTC**

Scoring Function:

Match: +m
Mismatch: -s
Gap: -d

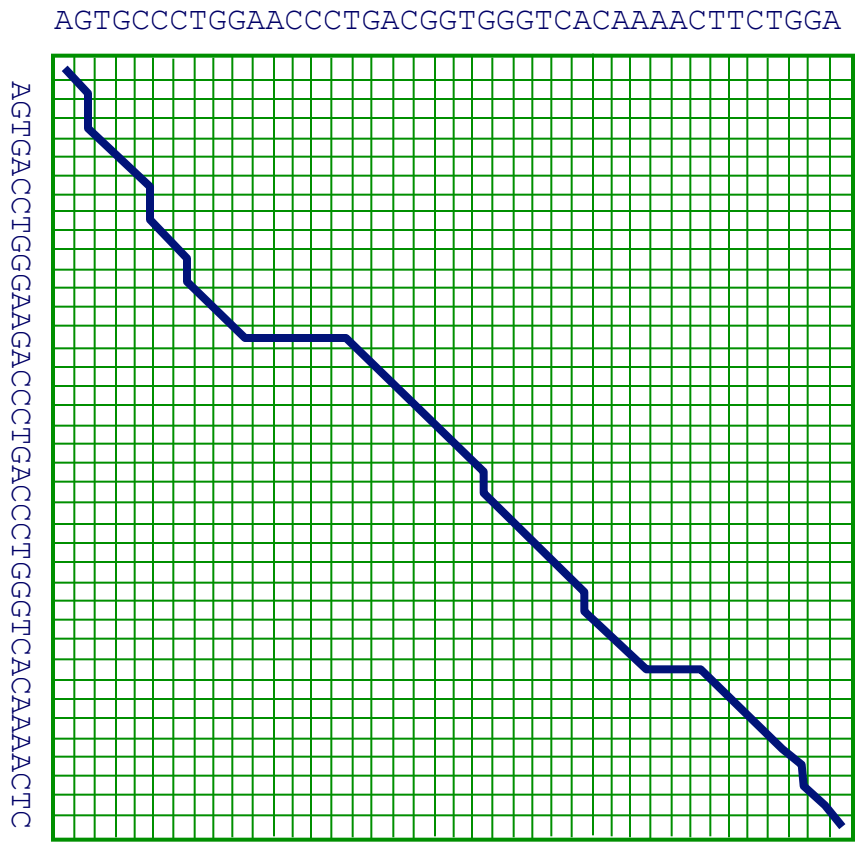
Alternative definition:

minimal edit distance

"Given two strings x, y, find minimum # of edits (insertions, deletions, mutations) to transform one string to the other"

Score F = (# matches) × m - (# mismatches) × s – (#gaps) × d

How do we compute the best alignment?

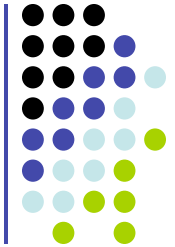


Too many possible alignments:

$>> 2^N$

(exercise)

Alignment is additive



Observation:

The score of aligning $x_1 \ldots\ldots x_M$ and $y_1 \ldots\ldots y_N$

is additive

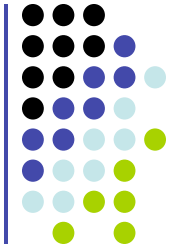
Say that $x_1 \ldots x_i$ $x_{i+1} \ldots x_M$

aligns to $y_1 \ldots y_j$ $y_{j+1} \ldots y_N$

The two scores add up:

$$F(x[1:M], y[1:N]) = F(x[1:i], y[1:j]) + F(x[i+1:M], y[j+1:N])$$

Dynamic Programming



- There are only a polynomial number of subproblems
  - Align $x_1 \ldots x_i$ to $y_1 \ldots y_j$
- Original problem is one of the subproblems
  - Align $x_1 \ldots x_M$ to $y_1 \ldots y_N$
- Each subproblem is easily solved from smaller subproblems
  - We will show next

Let

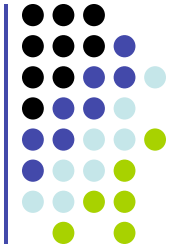
$F(i, j)$ = optimal score of aligning

$x_1 \ldots\ldots x_i$

$y_1 \ldots\ldots y_j$

F: Dynamic Programming table

Dynamic Programming (cont'd)



Notice three possible cases:

1. x_i aligns to y_j

   $x_1 \ldots\ldots x_{i-1}$ x_i

   $y_1 \ldots\ldots y_{j-1}$ y_j

   $$F(i, j) = F(i-1, j-1) + \begin{cases} m, \text{ if } x_i = y_j \\ -s, \text{ if not} \end{cases}$$

2. x_i aligns to a gap

   $x_1 \ldots\ldots x_{i-1}$ x_i

   $y_1 \ldots\ldots y_j$ -

   $$F(i, j) = F(i-1, j) - d$$

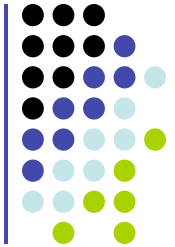
3. y_j aligns to a gap

   $x_1 \ldots\ldots x_i$ -

   $y_1 \ldots\ldots y_{j-1}$ y_j

   $$F(i, j) = F(i, j-1) - d$$

Dynamic Programming (cont'd)



How do we know which case is correct?

<u>Inductive assumption:</u>

$F(i, j-1), F(i-1, j), F(i-1, j-1)$ are optimal

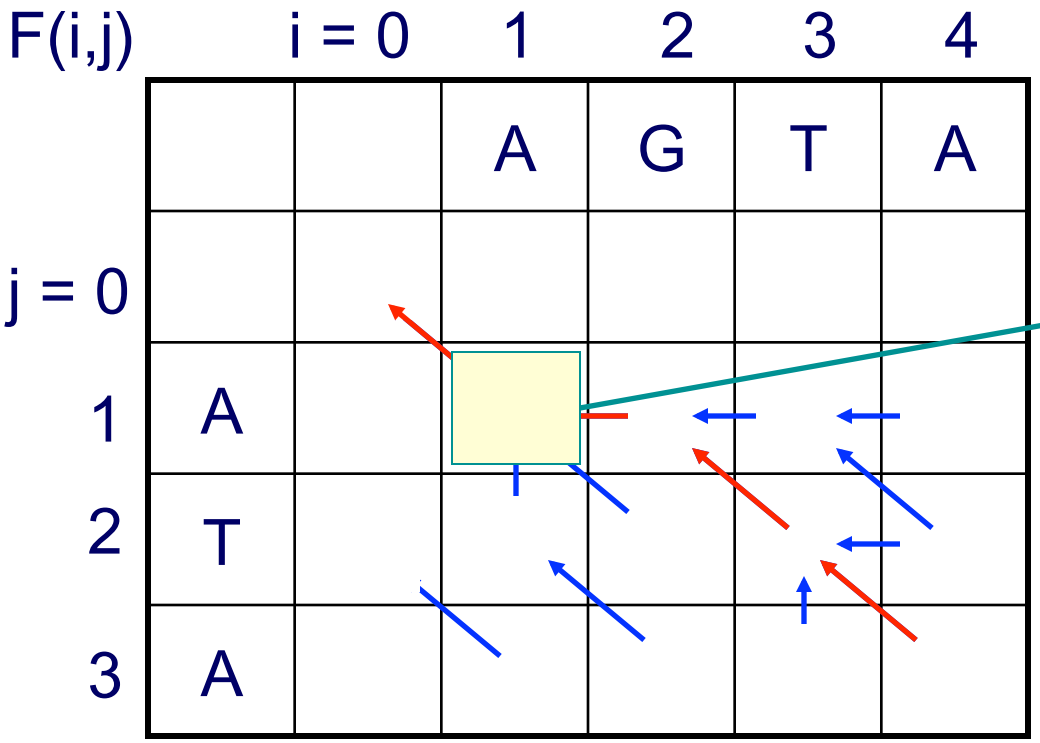
Then,

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

Where $s(x_i, y_j) = m$, if $x_i = y_j$; -s, if not

Example

x = AGTA
y = ATA

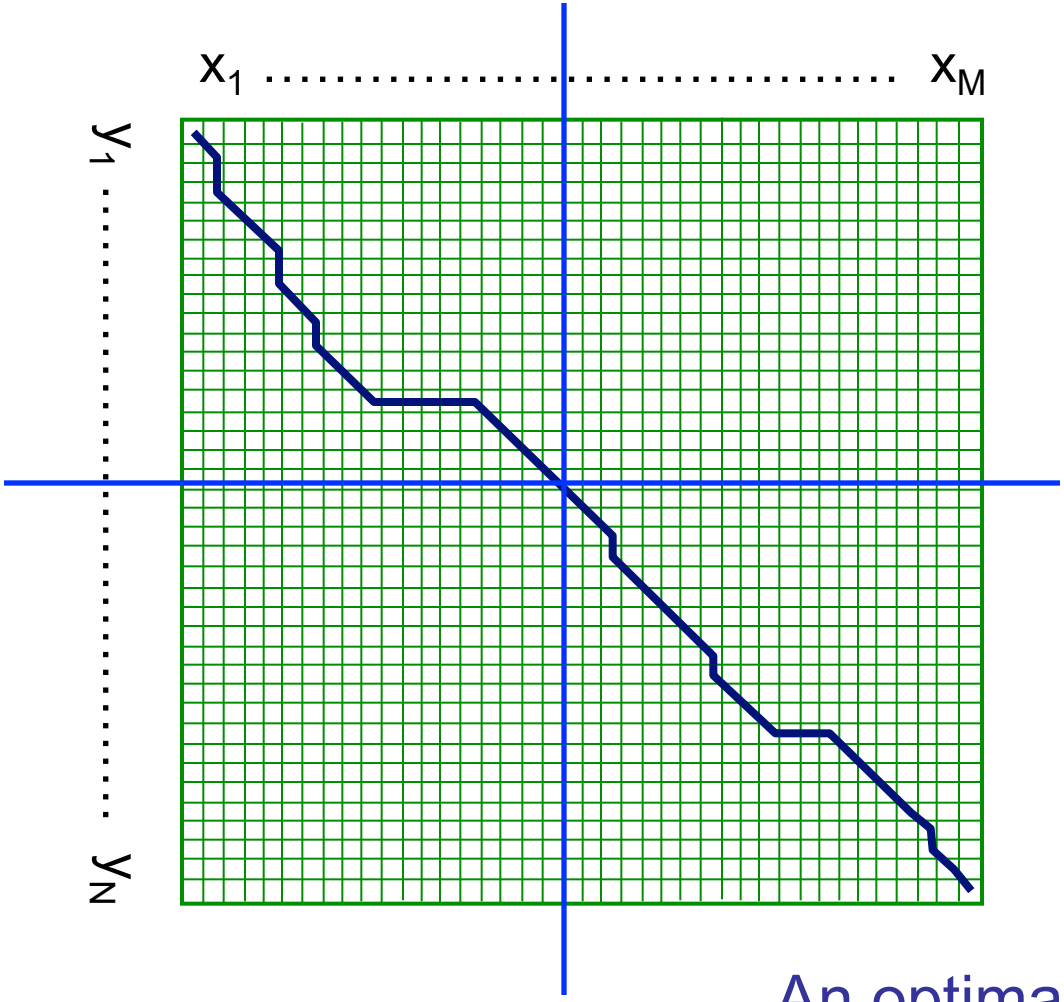


Procedure to output Alignment

- Follow the backpointers
- When diagonal,
  OUTPUT x_i, y_j
- When up,
  OUTPUT y_j
- When left,
  OUTPUT x_i

A G T A
A - T A

The Needleman-Wunsch Matrix



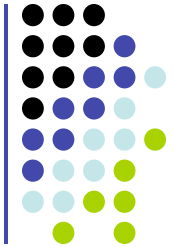
Every nondecreasing path

from (0,0) to (M, N)

corresponds to an alignment of the two sequences

An optimal alignment is composed of optimal subalignments

The Needleman-Wunsch Algorithm



Initialization.

$F(0, 0) = 0$

$F(0, j) = -j \times d$

$F(i, 0) = -i \times d$

Main Iteration. Filling-in partial alignments

For each i = 1……M

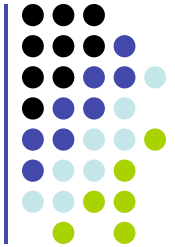
For each j = 1……N

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) & \text{[case 1]} \\ F(i-1, j) - d & \text{[case 2]} \\ F(i, j-1) - d & \text{[case 3]} \end{cases}$$

$$Ptr(i, j) = \begin{cases} \text{DIAG}, & \text{if [case 1]} \\ \text{LEFT}, & \text{if [case 2]} \\ \text{UP}, & \text{if [case 3]} \end{cases}$$

3. Termination. F(M, N) is the optimal score, and from Ptr(M, N) can trace back optimal alignment

Performance



- Time:

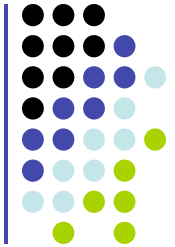
  O(NM)

- Space:

  O(NM)

- Later we will cover more efficient methods

A variant of the basic algorithm:

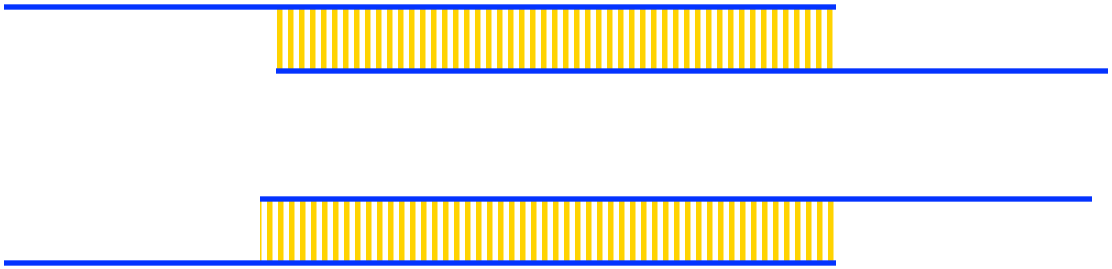


- Maybe it is OK to have an unlimited # of gaps in the beginning and end:

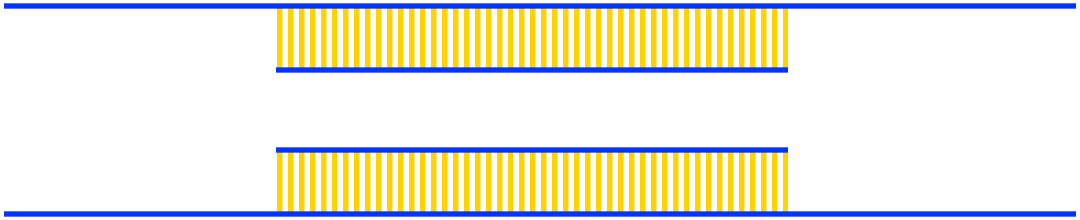
```
----------CTATCACCTGACCTCCAGGCCGATGCCCCTTCCGGC
          |||||||  ||||  |  ||  ||
GCGAGTTCATCTATCAC--GACCGC--GGTCG--------------
```

- Then, we don't want to penalize gaps in the ends

Different types of overlaps

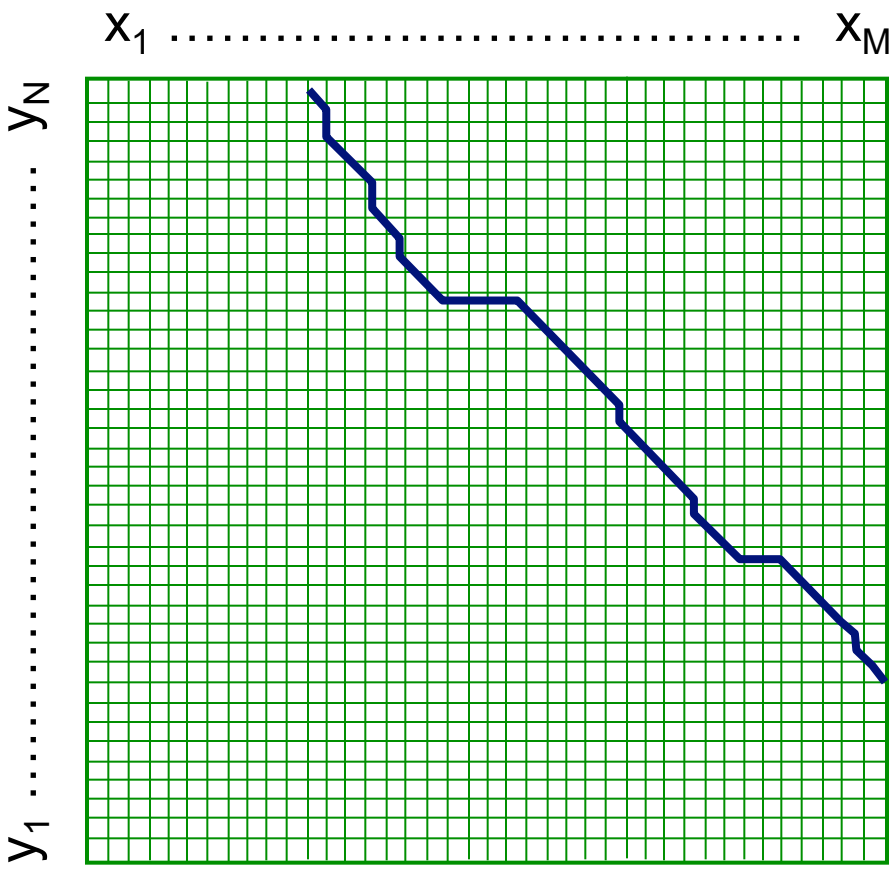


Example:
2 overlapping“*reads*” from a sequencing project



Example:
Search for a mouse gene within a human chromosome

The Overlap Detection variant



Changes:

1. Initialization

   For all i, j,

   $F(i, 0) = 0$

   $F(0, j) = 0$

2. Termination

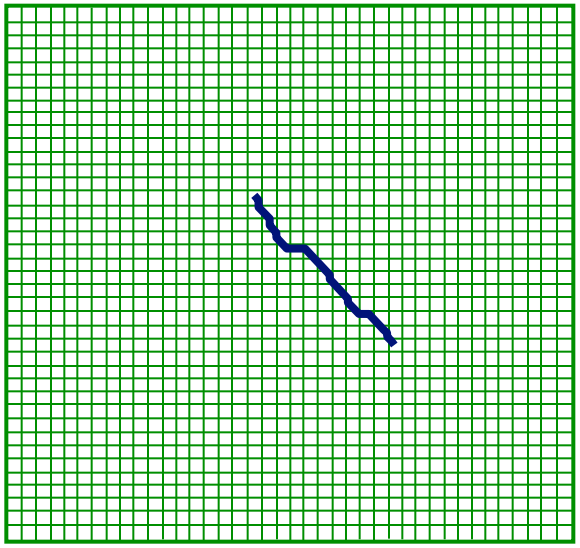
$$F_{OPT} = \max \begin{cases} \max_i F(i, N) \\ \max_j F(M, j) \end{cases}$$

The local alignment problem

Given two strings $x = x_1 \ldots\ldots x_M$,
$y = y_1 \ldots\ldots y_N$

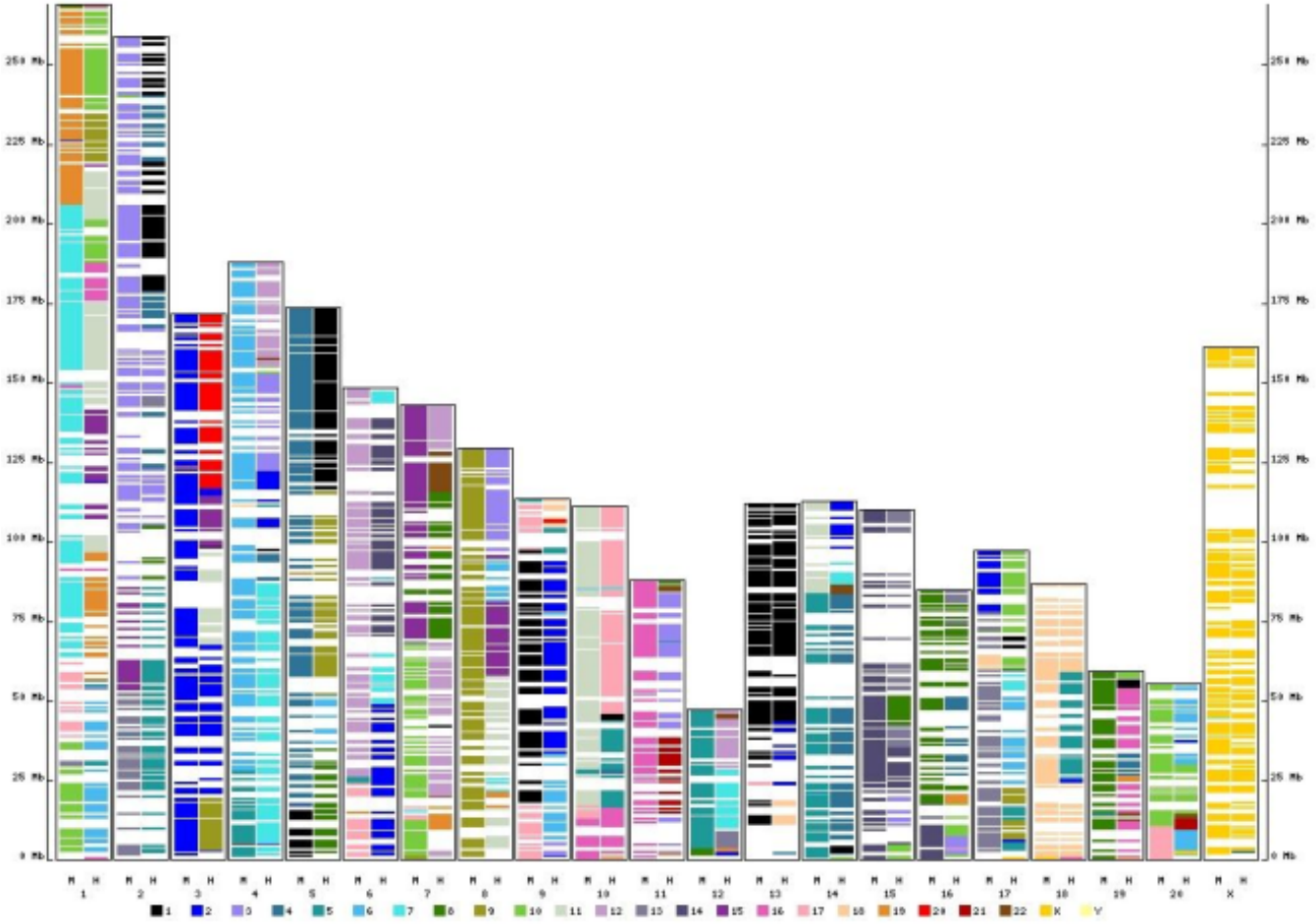
Find substrings x', y' whose similarity
(optimal global alignment value)
is maximum

x = aaaacccccgggggtta
y = ttcccgggaaccaacc

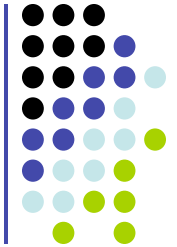


Why local alignment

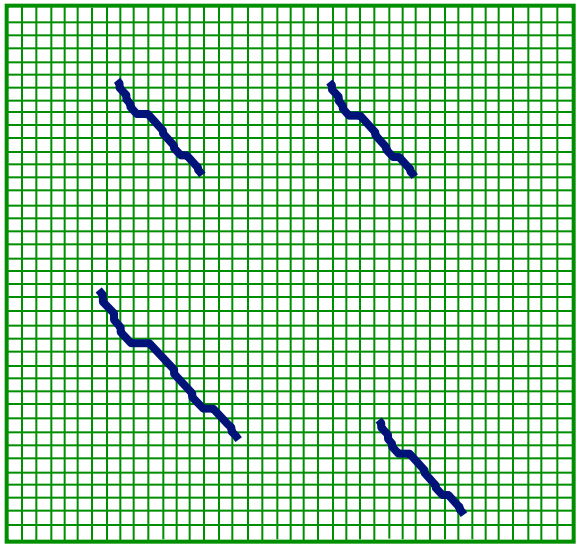
- Genes are shuffled between genomes



The Smith-Waterman algorithm



Idea: Ignore badly aligning regions



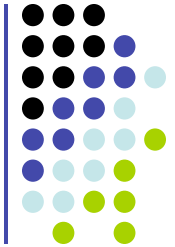
Modifications to Needleman-Wunsch:

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration:

$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$

The Smith-Waterman algorithm

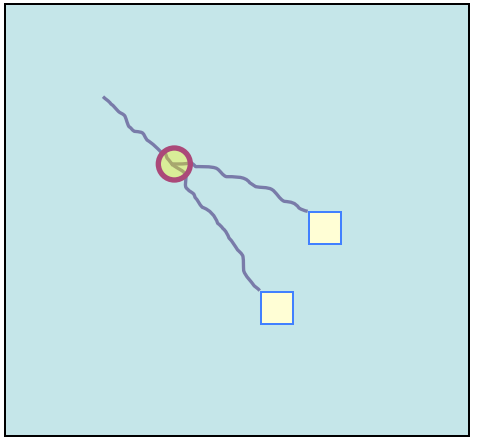


Termination:

1. If we want the best local alignment…

$$F_{OPT} = \max_{i,j} F(i, j)$$

Find F_{OPT} and trace back



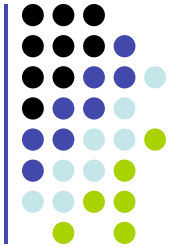
2. If we want all local alignments scoring > t

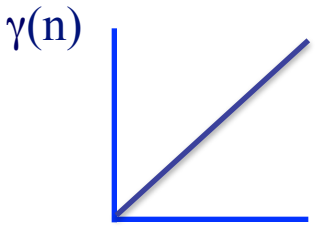
?? For all i, j find F(i, j) > t, and trace back?

Complicated by overlapping local alignments

Waterman–Eggert '87: ***find all non-overlapping local alignments with minimal recalculation of the DP matrix***

Scoring the gaps more accurately

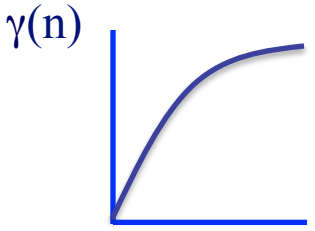




Current model:

Gap of length n
incurs penalty n×d

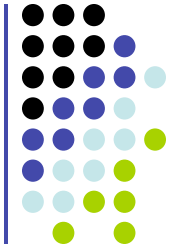
However, gaps usually occur in bunches



Concave gap penalty function $\gamma(n)$
(aka Convex $-\gamma(n)$):

$\gamma(n)$:
for all n, $\gamma(n + 1) - \gamma(n) \leq \gamma(n) - \gamma(n - 1)$

Convex gap dynamic programming



Initialization: same

Iteration:

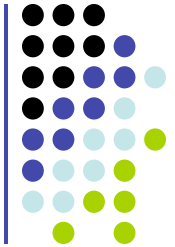
$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\ldots i-1} F(k, j) - \gamma(i-k) \\ \max_{k=0\ldots j-1} F(i, k) - \gamma(j-k) \end{cases}$$

Termination: same

Running Time: $O(N^2M)$ (assume N>M)

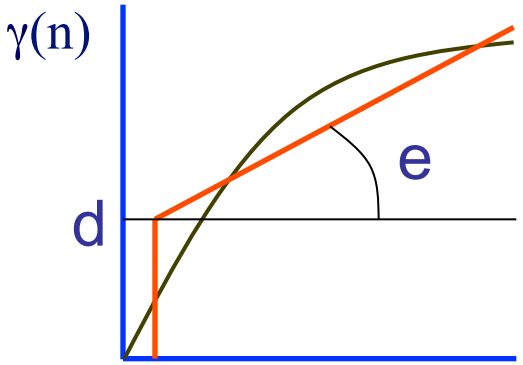
Space: $O(NM)$

Compromise: affine gaps



$\gamma(n) = d + (n - 1) \times e$

gap open	gap extend



To compute optimal alignment,

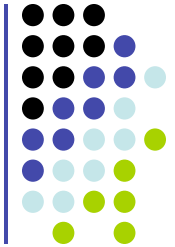
At position i, j, need to "remember" best score if gap is open
best score if gap is not open

F(i, j): score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$
if x_i aligns to y_j

G(i, j): score **if** x_i aligns to a gap after y_j
H(i, j): score **if** y_j aligns to a gap after x_i

V(i, j) = best score of alignment $x_1 \ldots x_i$ to $y_1 \ldots y_j$

Needleman-Wunsch with affine gaps



Why do we need matrices F, G, H?

Because, perhaps

G(i, j) < **V**(i, j)

(it is best to align x_i to y_j if we were aligning only $x_1 \ldots x_i$ to $y_1 \ldots y_j$ and not the rest of x, y),

but on the contrary

G(i, j) – e > **V**(i, j) – d

(i.e., had we "fixed" our decision that x_i aligns to y_j, we could regret it at the next step when aligning $x_1 \ldots x_{i+1}$ to $y_1 \ldots y_j$)

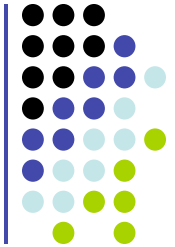
Add -d

G(i+1, j) = **F**(i, j) – d

Add -e

G(i+1, j) = **G**(i, j) – e

Needleman-Wunsch with affine gaps



Initialization:

$$V(i, 0) = - d - (i - 1) \times e$$
$$V(0, j) = - d - (j - 1) \times e$$

Iteration:

$$V(i, j) = \max\{ F(i, j), G(i, j), H(i, j) \}$$

$$F(i, j) = V(i - 1, j - 1) + s(x_i, y_j)$$

$$G(i, j) = \max \begin{cases} V(i - 1, j) - d \\ G(i - 1, j) - e \end{cases}$$

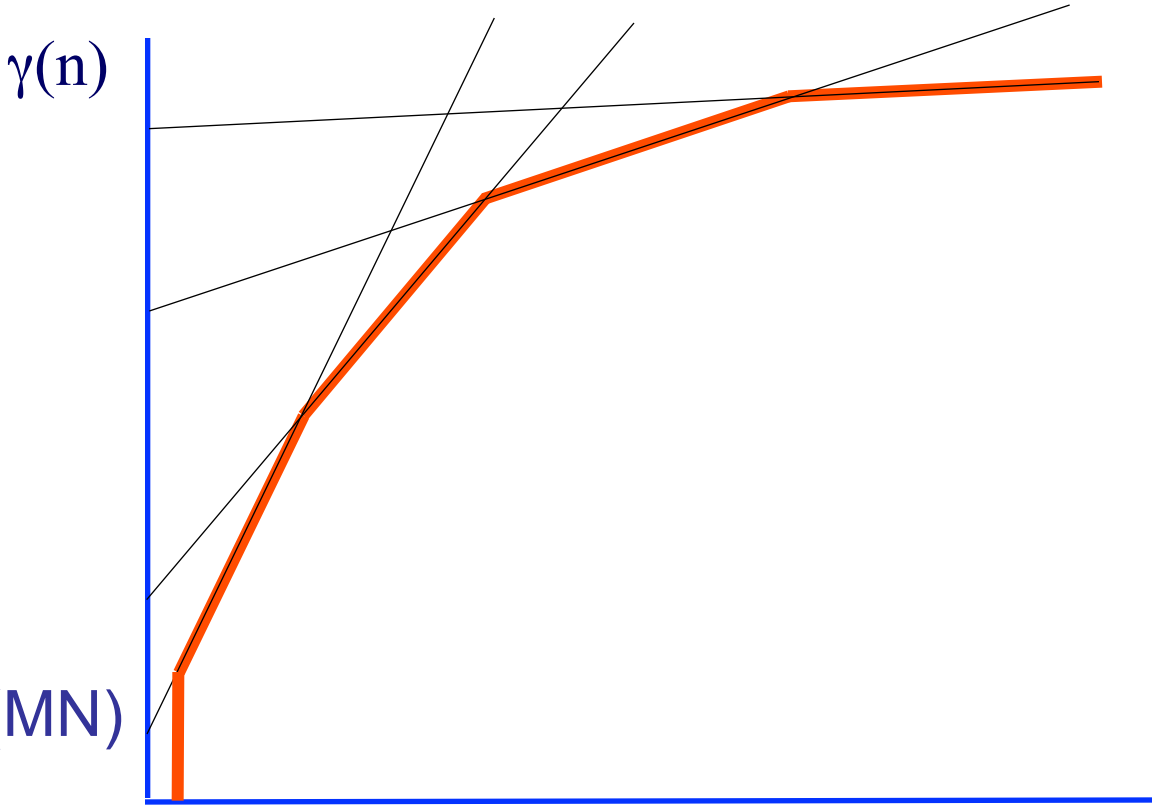
$$H(i, j) = \max \begin{cases} V(i, j - 1) - d \\ H(i, j - 1) - e \end{cases}$$

Termination: V(i, j) has the best alignment

Time?
Space?

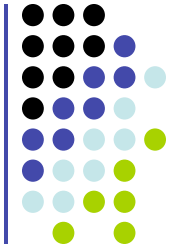
To generalize a bit…

… think of how you would compute optimal alignment with this gap function



….in time O(MN)

Bounded Dynamic Programming



Assume we know that x and y are very similar

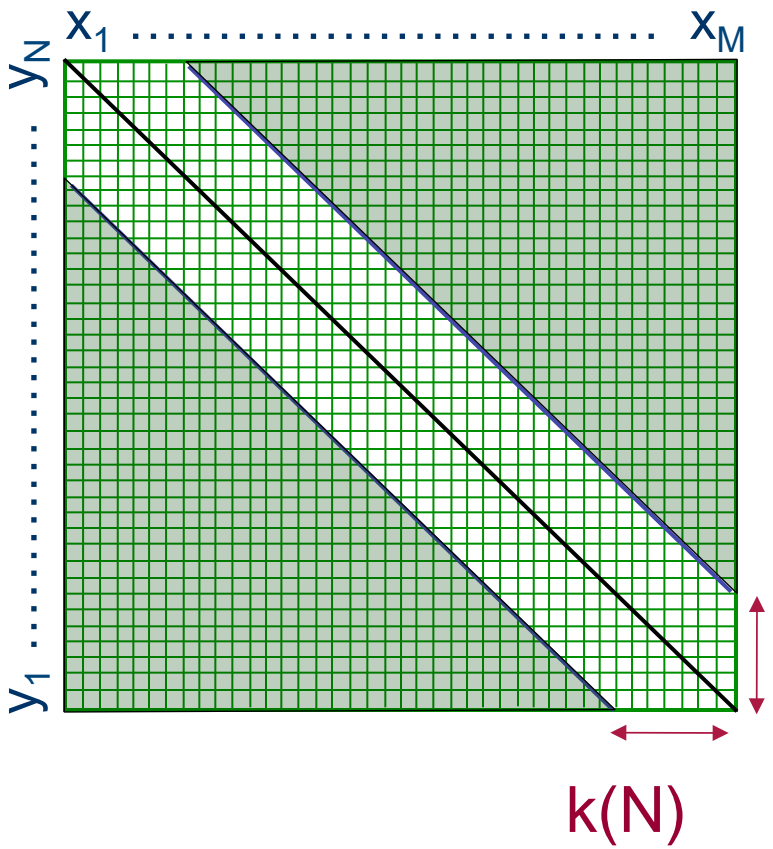
Assumption: # gaps(x, y) < k(N)

Then, $x_i \mid y_j$ implies $| i - j | < k(N)$

We can align x and y more efficiently:

Time, Space: $O(N \times k(N)) << O(N^2)$

Bounded Dynamic Programming



Initialization:

F(i,0), F(0,j) undefined for i, j > k

Iteration:

For i = 1…M

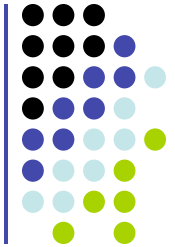
For j = max(1, i – k)…min(N, i+k)

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i, j-1) - d, \text{ if } j > i - k(N) \\ F(i-1, j) - d, \text{ if } j < i + k(N) \end{cases}$$

Termination: same

Easy to extend to the affine gap case

Outline



- Linear-Space Alignment
- BLAST – local alignment search
- Ultra-fast alignment for (human) genome resequencing