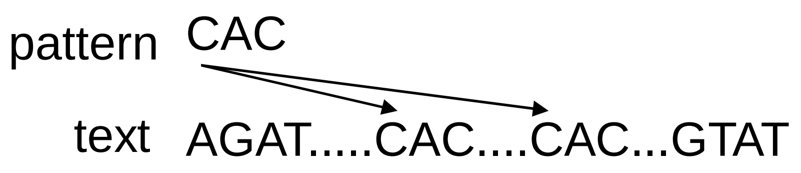
Prelude to Sequence Alignment

- Content
  - General results in Combinatorial Pattern Matching / Stringology
    - Knuth-Morris-Pratt
    - Boyer-Moore
    - Suffix tree, Suffix array
    - Edit distance
    - Dynamic Programming
    - Approximate pattern search, k-mismatches, k-errors
  - Solutions specific to Bioinformatics
    - Needleman-Wunsch (global alignment, score matrixes)
    - Smith-Waterman (local alignment)
    - FASTA, BLAST
    - ...

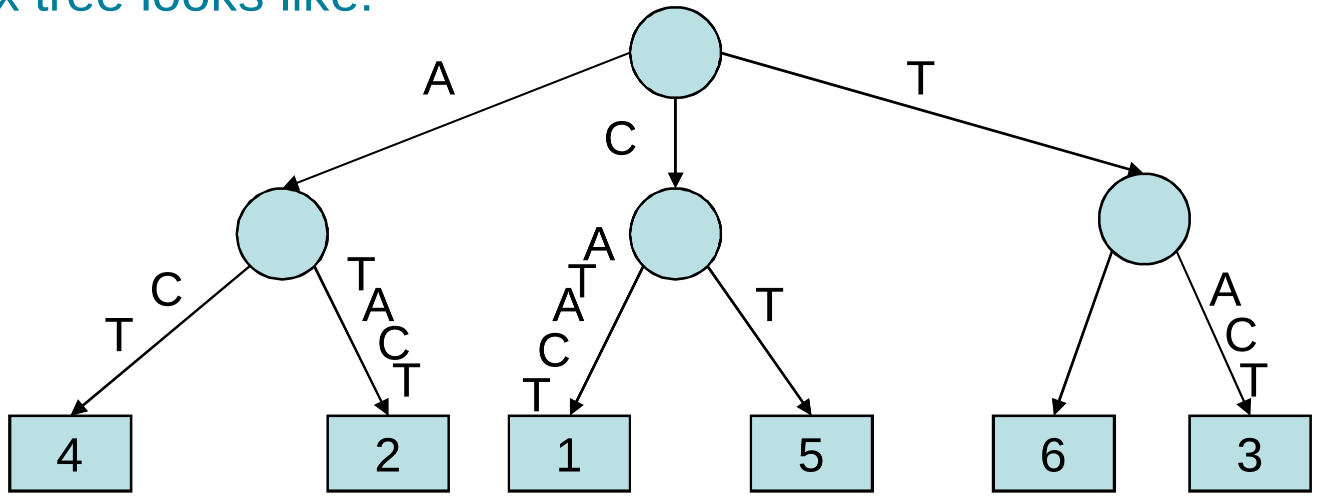
Classical results from Stringology

- The world of "text of length *n*, pattern of length *m*".
- Knuth-Morris-Pratt: *O(n)* time exact pattern search.
- Boyer-Moore: *O(n/m)* time exact pattern search on average.
- Powerful general tools: *Suffix tree* and *suffix array*
- Numerous theoretical results on approximate pattern matching

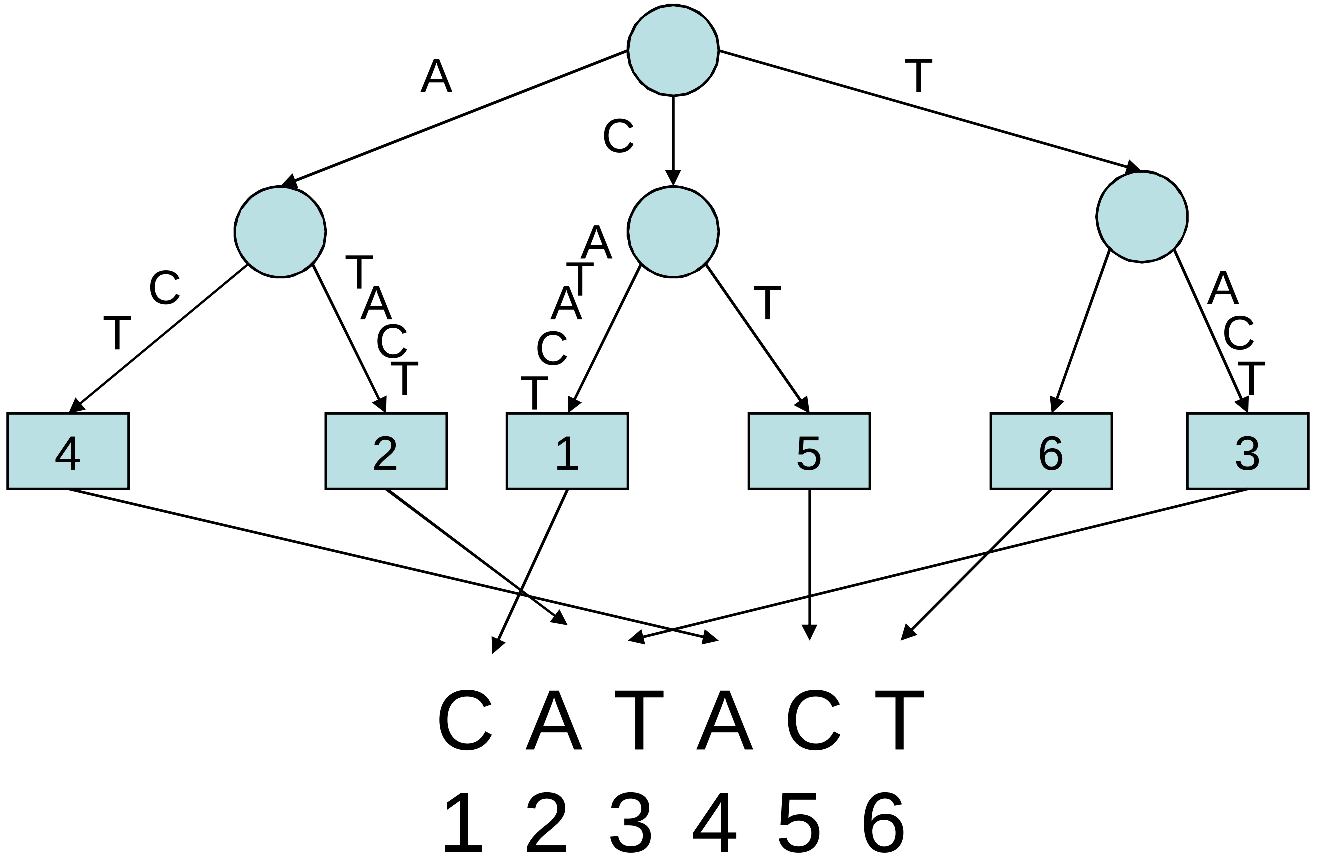


Suffix tree

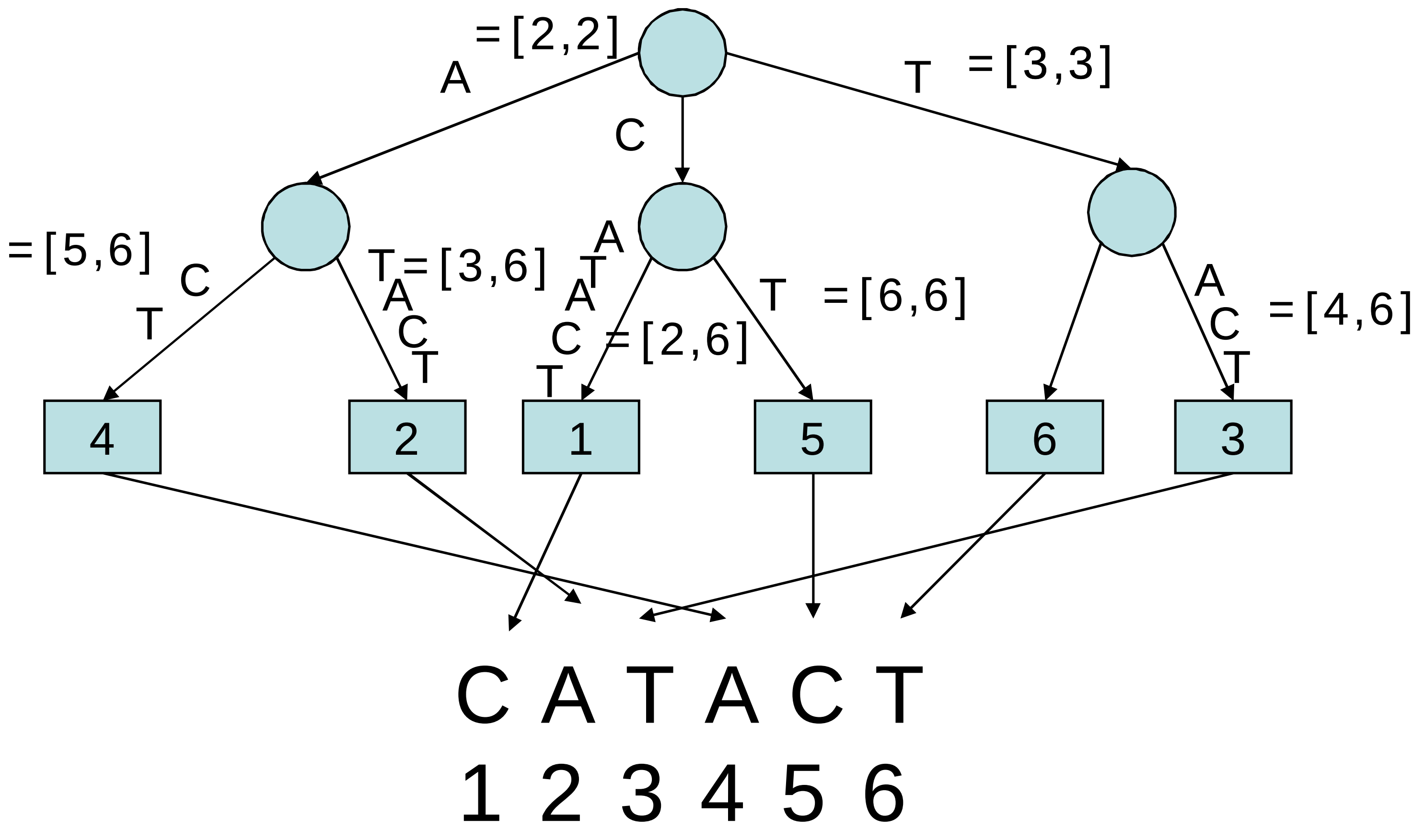
- Suffix tree is a compressed keyword trie of all *suffixes* of a sequence
- E.g. suffixes of sequence CATACT are CATACT, ATACT, TACT, ACT, CT, T.
  - suffix tree looks like:



Suffix tree



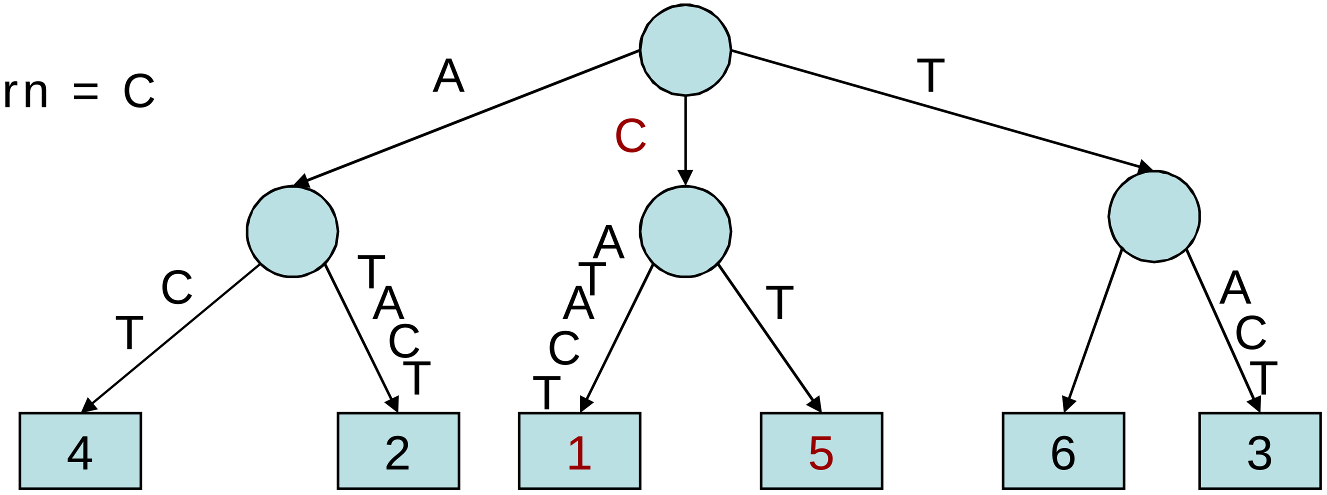
Suffix tree



A = [2,2]
C
T = [3,3]
CT = [5,6]
TACT = [3,6]
ATACT
ACT = [2,6]
T = [6,6]
ACT = [4,6]
4
2
1
5
6
3
C A T A C T
1 2 3 4 5 6

Exact search on suffix tree

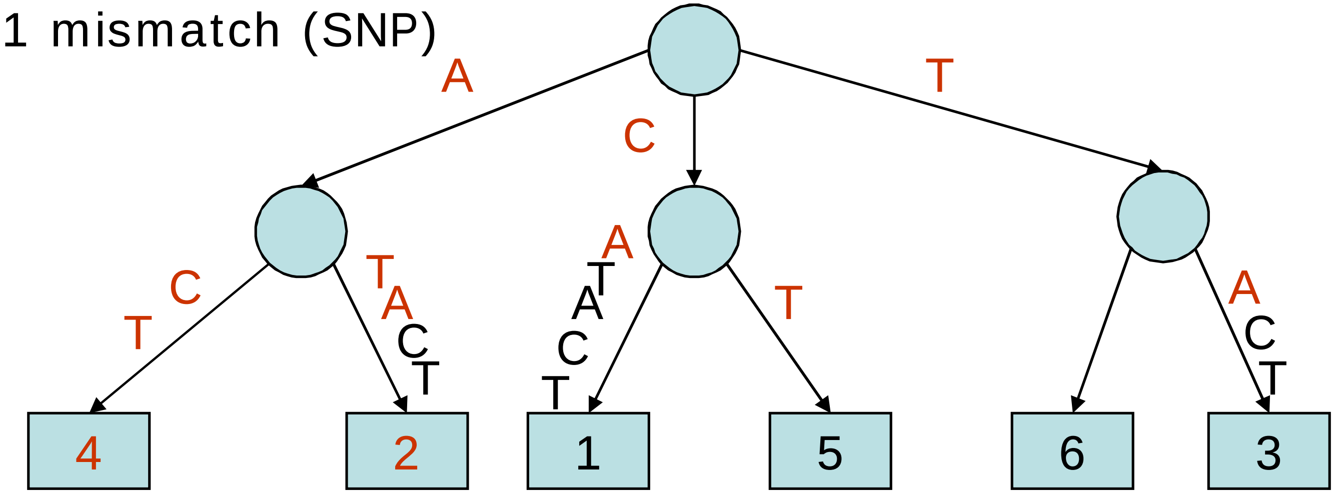
pattern = C



C A T A C T
1 2 3 4 5 6

Backtracking on suffix tree

ACA, 1 mismatch (SNP)



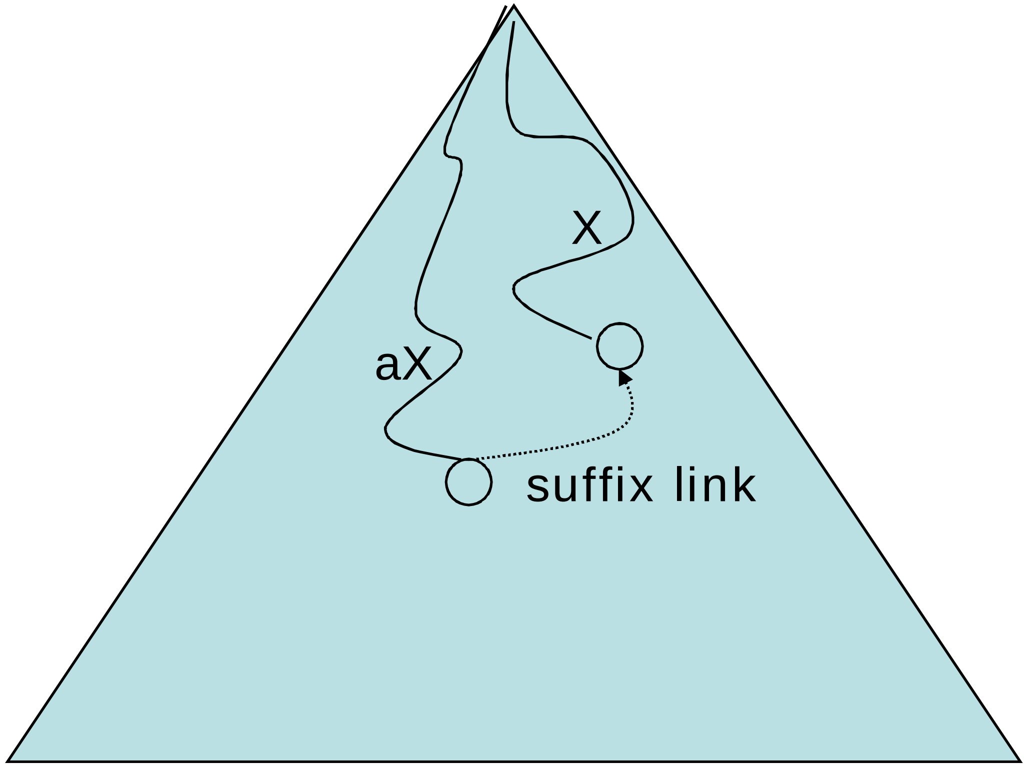
C A T A C T

1 2 3 4 5 6

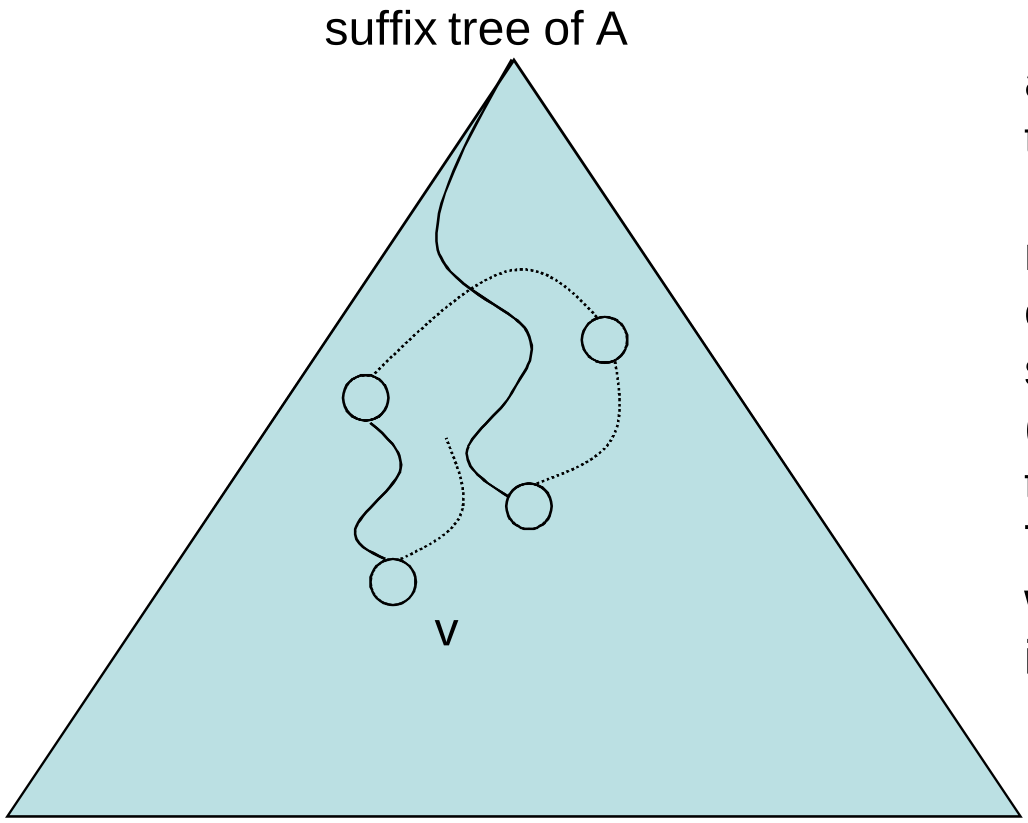
Simple analysis task: LCSS

- Let LCSS(A,B) denote the longest common substring of two sequences A and B. E.g.:
  - LCSS(AGATCTATCT,CGCCTCTATG)=TCTAT.
- A good solution is to build suffix tree for the shorter sequence and make a *descending suffix walk* with the other sequence.

Suffix link

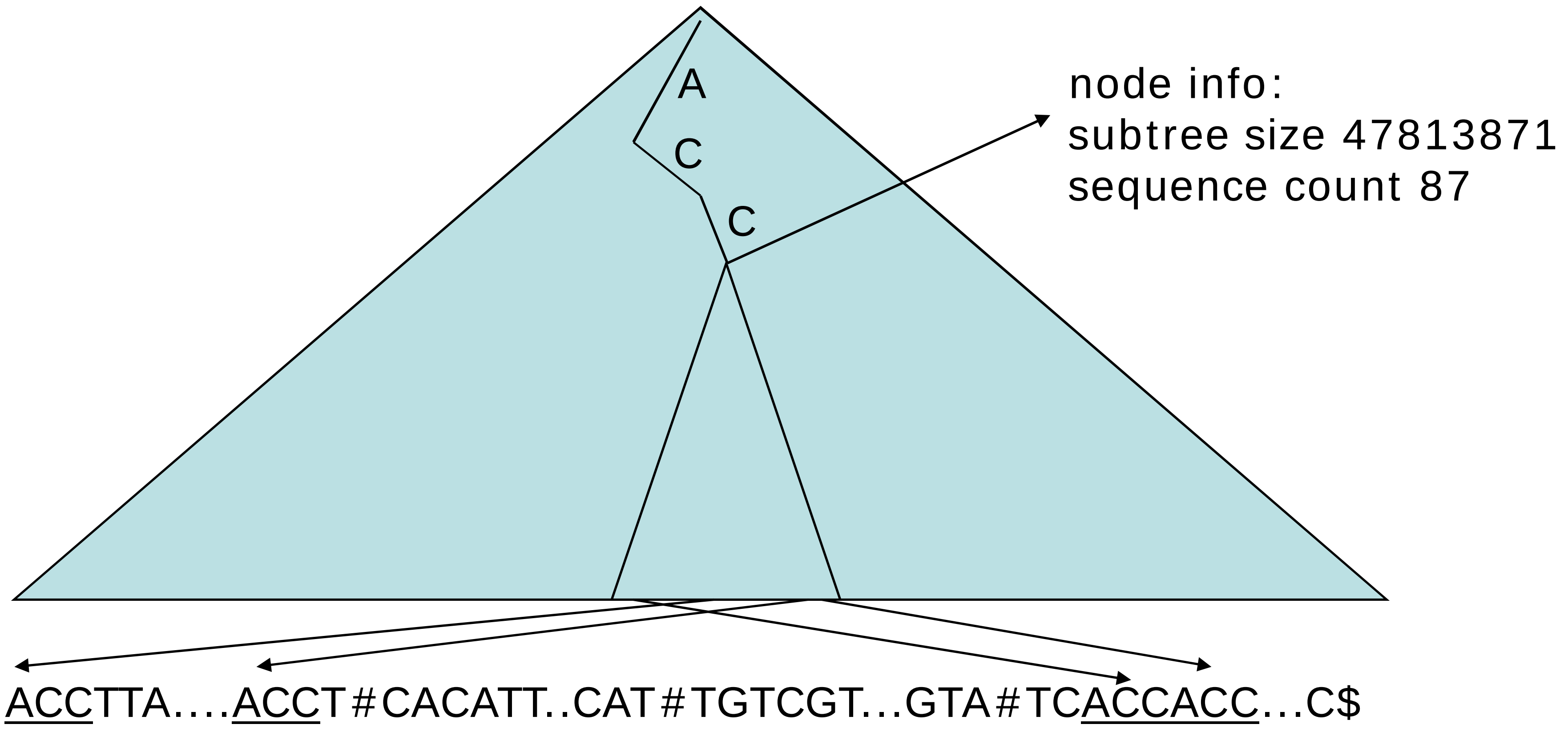


Descending suffix walk

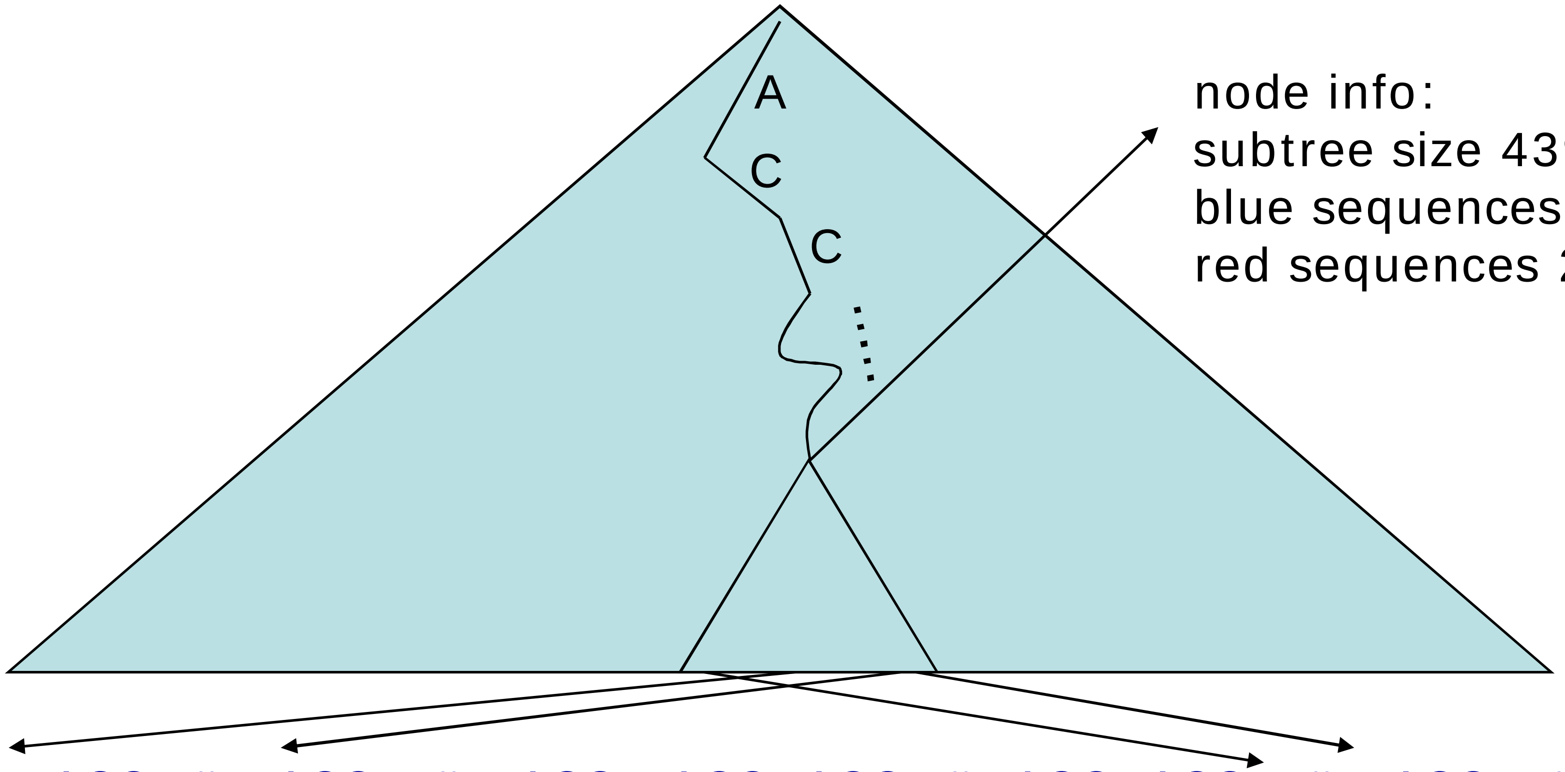


Read B left-to-right, always going down in the tree when possible.
If the next symbol of B does not match any edge label on current position, take suffix link, and try again. (Suffix link in the root to itself emits a symbol).
The node v encountered with largest string depth is the solution.

Another common tool: Generalized suffix tree



Generalized suffix tree application



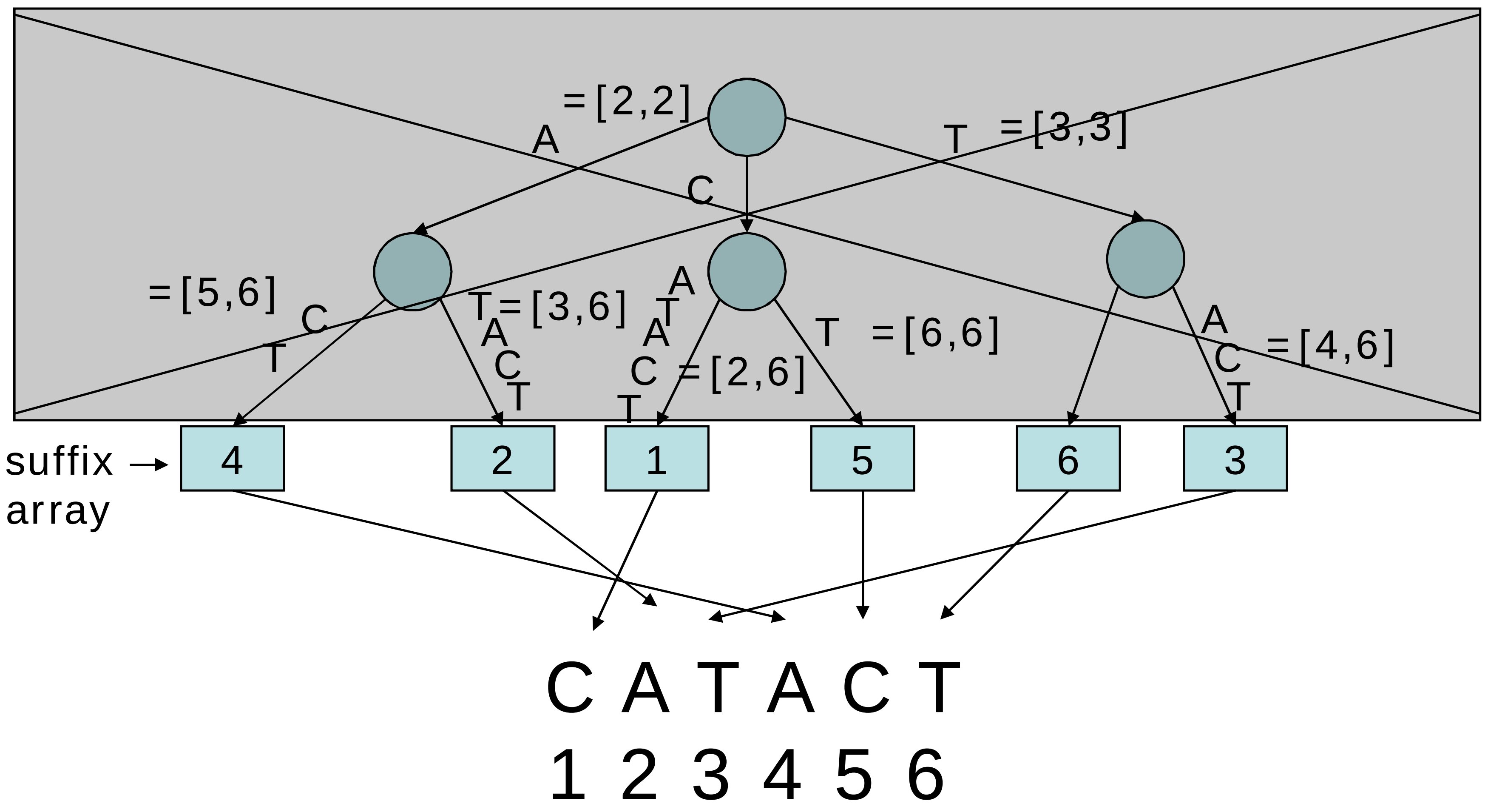
…ACC..#…ACC…#…ACC…ACC..ACC..#..ACC..ACC…#…ACC…#…

…#….#…#…#…ACC…#…#…#…#…#…#..#..ACC..ACC…#……#…

Properties of suffix tree

- Suffix tree has *n* leaves and at most *n-1* internal nodes, where *n* is the total length of all sequences indexed.
- Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).
- Can be constructed in linear time (e.g. Ukkonen's online linear time construction).
- In practice: Huge overhead due to pointer structure:
  - Standard implementation of suffix tree for human genome requires over 200 GB memory!

Reducing space: suffix array



Suffix array

- Many algorithms on suffix tree can be simulated using suffix array.
- For example, exact pattern search works using binary search on suffix array.
- Suffix array is the basis a popular bioinformatics tool called *Mummer*.
- Suffix array can be constructed easily from suffix tree, but there are also direct linear time construction algorithms that take less space (e.g. Kärkkäinen & Sanders algorithm).

Approximate string matching

- *k-mismatches problem*: Search all occurrences *O* of pattern *P[1,m]* in text *T[1,n]* such that *P* differs in at most *k* positions from the occurrence substring:
  - More formally: $j \in O$ is a k-mismatch occurrence position of *P* in *T* if and only if $d_H(P,T[j,j+m-1]) \leq k$, where $d_H(A,B)$ is the *Hamming distance* of *A* and *B*.
  - $d_H(A,B)=|\{ i : A[i] \neq B[i]\}|$.
  - Theory: $O(kn)$ time algorithm is easy to achieve (using suffix trees and some advanced data structure techniques) and very sophisticated algorithms exist to solve the problem even faster.
  - Practice: naive algorithm or backtracking on suffix tree (slide 7) work well for small *k*.

Approximate pattern matching: filtering

- Best practical algorithms for approximate string matching use *filtering*:
  - Sweep the text with a fast algorithm to detect possible candidate occurrence positions.
  - Check all candidates for real occurrences.
  - There are *noisy filters* (that may fail to find some candidates that are real occurrences) and *noiseless filters* (that are guaranteed to find all real occurrences).
- Simple noiseless filter for k-mismatch search:
  - Partition the pattern into *k+1* pieces.
  - Take all exact occurrences of the pieces as candidates.
  - Check all candidates with naive algorithm.

Approximate string matching: filtering example

- Text T=CGAGCGATAGCTACCGT
- Pattern P=ACAG, $k=1$
- Partition P into e.g. P^1=AC, P^2=AG
- Search P^1 and P^2 in T: CG$\underline{\text{AG}}$CGAT$\underline{\text{AG}}$CT$\underline{\text{AC}}$CGT
- Check the candidates: $\underline{\text{CGAG}}$CG$\underline{\text{ATAG}}$CT$\underline{\text{ACCG}}$T
- Running time:
  - Build suffix tree of T: $O(n)$ time.
  - Search P^1 and P^2 in suffix tree of T: $O(m+\#candidates)$ time.
  - Checking $O(\#candidates \times m)$ time.
- The challenge: *#candidates >> #occurrences*
  - Better filters than above exist (with smaller *#candidates*)

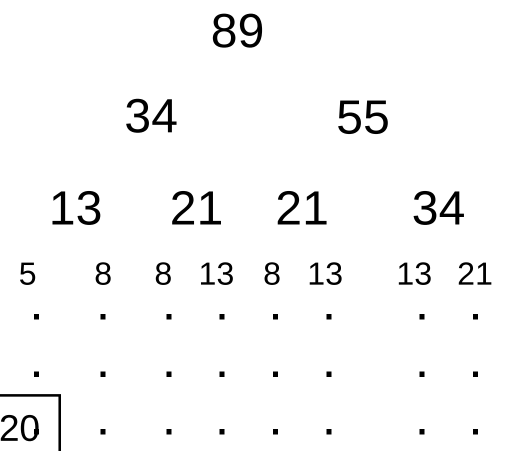
Approximate what?

- Different versions of approximate pattern matching can be defined modifying the distance function *$d(A,B)$*.
- The most studied distance function is *unit cost edit distance* or *Levenshtein distance*.
  - *$d_L(A,B)$* is the minimum amount of single symbol insertions, deletions, and substitutions required to convert *A* into *B*.
  - For example, on *A="stockholm"* and *B="tukholma"* we have *$d_L(A,B)=4$*:
    - delete s, substitute o->u, delete c, insert a
    - .. or delete s, delete o, substitute c->u, insert a
    - .. or is there better sequence of edits???

```
s t o c k h o l m -
- t u - k h o l m a
```

Dynamic programming

- Way to compute edit distance optimally.
- General algorithm principle:
  - Can be seen as a variant of *Dijkstra's shortest path algorithm*.
- Abstract idea: Use induction to break the problem into smaller subproblems and suitable evaluation order so that subproblem solutions are available when needed.
- Concrete example, Fibonacci numbers:
  - 0,1,1,2,3,5,8,13,21,<u>34</u>,<u>55</u>,89,...
  - *F(i)=F(i-2)+F(i-1)* with *F(0)=0*, *F(1)=1*
  - The recursion to compute *F(i)* contains many identical subproblems.



Edit distance

- Let $A=a_1a_2...a_m$ and $B=b_1b_2...b_n$ be two strings.
- Consider an optimal listing of edits to convert the prefix $a_1a_2...a_i$ of A into prefix $b_1b_2...b_j$ of B corresponding to $d_L(a_1a_2...a_i,b_1b_2...b_j)$:
  - If $a_i=b_j$ we know that $d_L(a_1a_2...a_i,b_1b_2...b_j)=d_L(a_1a_2...a_{i-1},b_1b_2...b_{j-1})$
  - Otherwise either a_i is substituted by b_j, or a_i is deleted or b_j is inserted in the optimal list of edits.
  - Hence, we have $d_L(a_1a_2...a_i,b_1b_2...b_j)=$
    $min(d_L(a_1a_2...a_{i-1},b_1b_2...b_{j-1})+(\text{if } a_i=b_j \text{ then } 0 \text{ else } 1),$
    $d_L(a_1a_2...a_{i-1},b_1b_2...b_j)+1,$
    $d_L(a_1a_2...a_i,b_1b_2...b_{j-1})+1)$.

Edit distance matrix D[i,j]

- Let $D[i,j]$ denote $d_L(a_1a_2\ldots a_i, b_1b_2\ldots b_j)$.
- Obviously $D[0,j]=j$ and $D[i,0]=i$.
- The induction from previous slide gives
  $D[i,j]=\min(D[i-1,j-1]+\text{if } (a_i=b_j) \text{ then } 0 \text{ else } 1,$
  $D[i-1,j]+1, D[i,j-1]+1)$.
- Matrix D can be computed row-by-row, column-by-column (or in many other evaluation orders) so that $D[i-1,j-1]$, $D[i-1,j]$, and $D[i,j-1]$ are available when computing $D[i,j]$.
- Running time to compute $D[m,n]$ is $O(mn)$.

Edit distance example

		s	t	o	c	k	h	o	l	m
	0	1	2	3	4	5	6	7	8	9
t	1	1	1	2	3	4	5	6	7	8
u	2	2	2	2	3	4	5	6	7	8
k	3	3	3	3	3	3	4	5	6	7
h	4	4	4	4	4	4	3	4	5	6
o	5	5	5	4	5	5	4	3	4	5
l	6	6	6	5	5	6	5	4	3	4
m	7	7	7	6	6	6	6	5	4	3
a	8	8	8	7	7	7	7	6	5	4

k-errors problem

- *k-errors problem* is the approximate string matching problem with edit distance:
    - More formally: $j \in O$ is a k-errors occurrence (end)position of P in T if and only if $d_L(P,T[j',j]) \leq k$ for some j'.
- Can be solved with the "zero the first row trick":
    - $D[0,j]=0$ for all j.
    - Otherwise the computation is identical to edit distance computation using matrix D.
    - Intuition: $D[i,j]$ then equals the minimum number of edits to convert $P[1,i]$ into *some suffix of* $T[1,j]$.
    - If $D[m,j] \leq k$, then P can be converted to some substring $T[j',j]$ with at most k edit operations.

Current applications

- Short-read sequencing (454, Solexa, SOLiD) has raised again the issue of doing fast k-mismatches and k-errors matching.
- Some popular software packages exploit the suffix tree backtracking idea (bowtie, bwa, SOAP2):
  - Instead of suffix tree, a *compressed suffix array* based on so-called *Burrows-Wheeler transform* is used as backbone of the search.
  - The index size for e.g. human genome can be kept in ~3 GB.
  - Compression does not affect the running time significantly.

More on general string processing techniques...

- Gusfield's book: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology
- 58093-3 Merkkijonomenetelmät (String Processing Algorithms)
  - Lectured previously Autumn 2008.
  - Next time Autumn 2010 in English?
- ISMB 2009 tutorial on compressed index structures applied to short-read mapping (http://www.cs.helsinki.fi/u/vmakinen/ismb09tutorial_vm.pdf)

Sequence alignment

- *The biological problem*
- Global alignment
- Local alignment
- Multiple alignment

Background: comparative genomics

- Basic question in biology: *what properties are shared among organisms?*
- Genome sequencing allows comparison of organisms at DNA and protein levels
- Comparisons can be used to
  - Find evolutionary relationships between organisms
  - Identify functionally conserved sequences
  - Identify corresponding genes in human and model organisms: develop models for human diseases

Homologs

- Two genes (sequences in general) g_B and g_C evolved from the same ancestor gene g_A are called *homologs*
- Homologs usually exhibit conserved functions
- Close evolutionary relationship => expect a high number of homologs

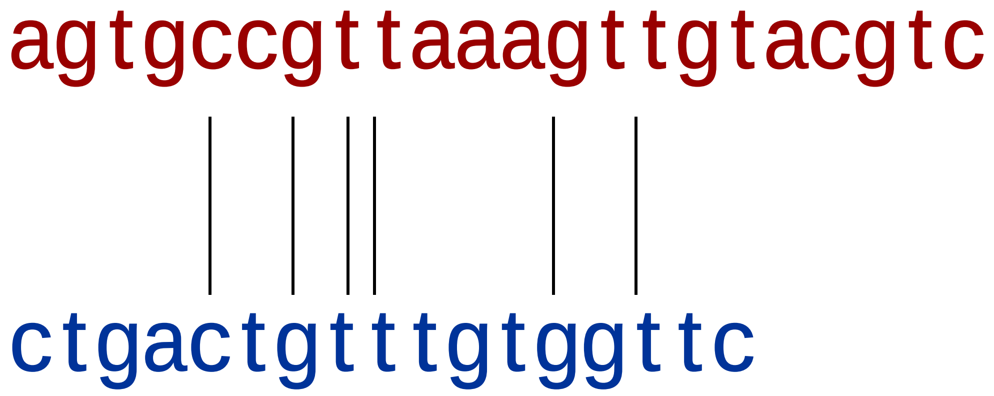
g_A = agtgtccgttaagtgcgttc

g_B = agtgccgttaaagttgtacgtc

g_C = ctgactgtttgtggttc

Sequence similarity

- We expect homologs to be ”similar” to each other
- Intuitively, similarity of two sequences refers to the degree of match between corresponding positions in sequence



- What about sequences that differ in length?

Similarity vs homology

- Sequence similarity is not sequence homology
  - If the two sequences g_B and g_C have accumulated enough mutations, the similarity between them is likely to be low

#mutations	
0	agtgtccgttaagtgcgttc
1	agtgtccgttatagtgcgttc
2	agtgtccgcttatagtgcgttc
4	agtgtccgcttaagggcgttc
8	agtgtccgcttcaaggggcgt
16	gggccgttcatggggt
32	gcagggcgtcactgagggct

#mutations	
64	acagtccgttcgggctattg
128	cagagcactaccgc
256	cacgagtaagatatagct
512	taatcgtgata
1024	acccttatctacttcctggagtt
2048	agcgacctgcccaa
4096	caaac

Homology is more difficult to detect over greater evolutionary distances.

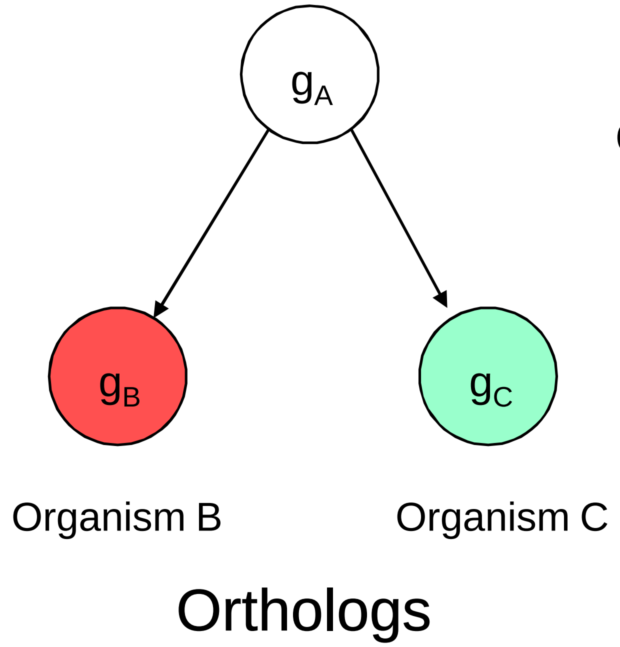
Similarity vs homology (2)

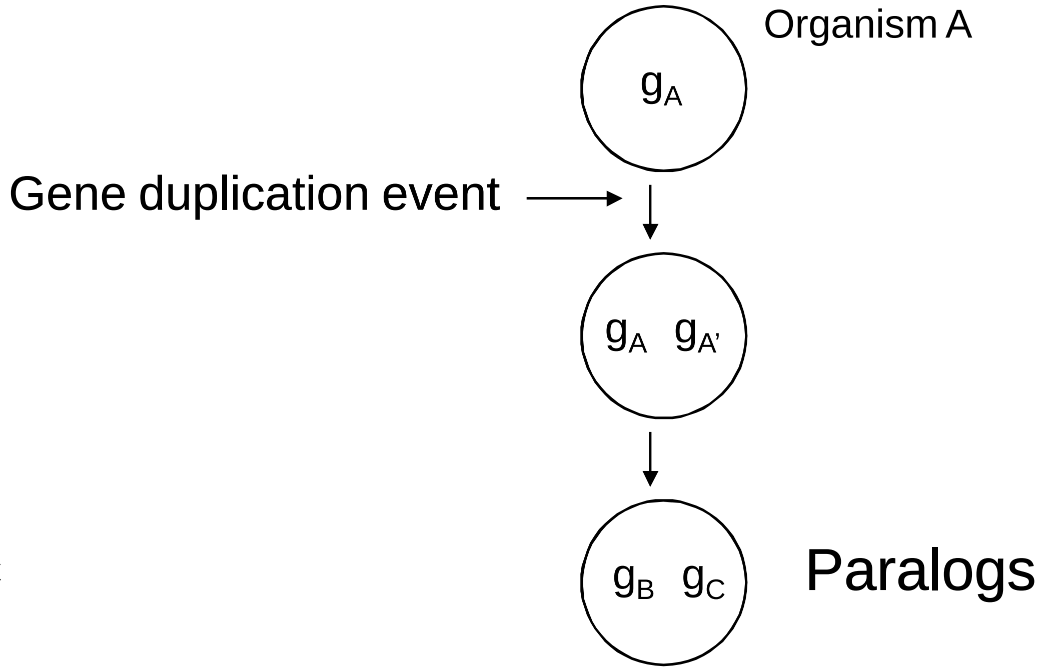
- Sequence similarity can occur by chance
  - *Similarity does not imply homology*

- Consider comparing two short sequences against each other

Orthologs and paralogs

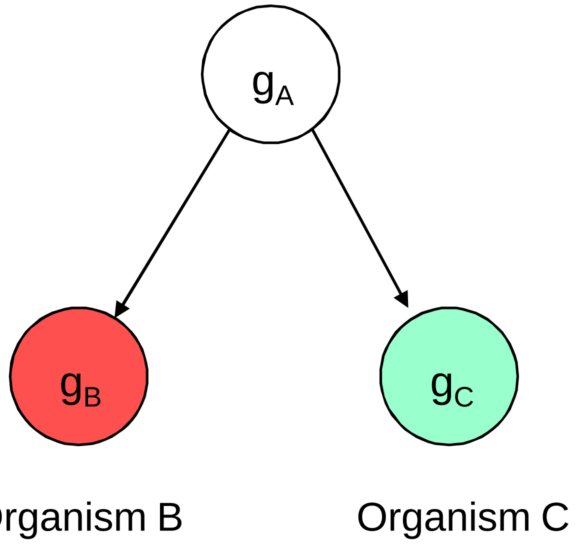
- We distinguish between two types of homology
  - Orthologs: homologs from two different species, separated by a *speciation* event
  - Paralogs: homologs within a species, separated by a *gene duplication* event

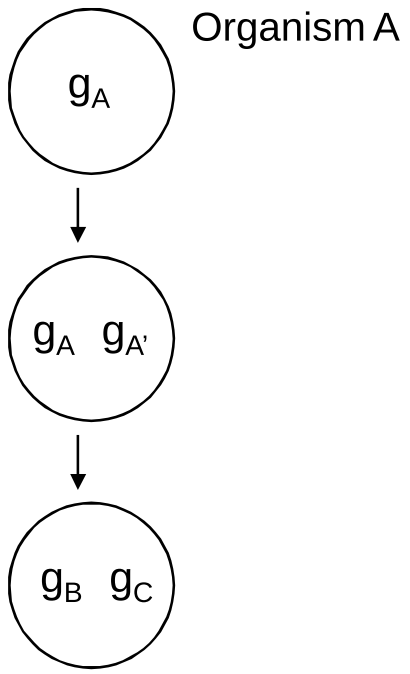




Orthologs and paralogs (2)

- Orthologs typically retain the original function
- In paralogs, one copy is free to mutate and acquire new function (no selective pressure)



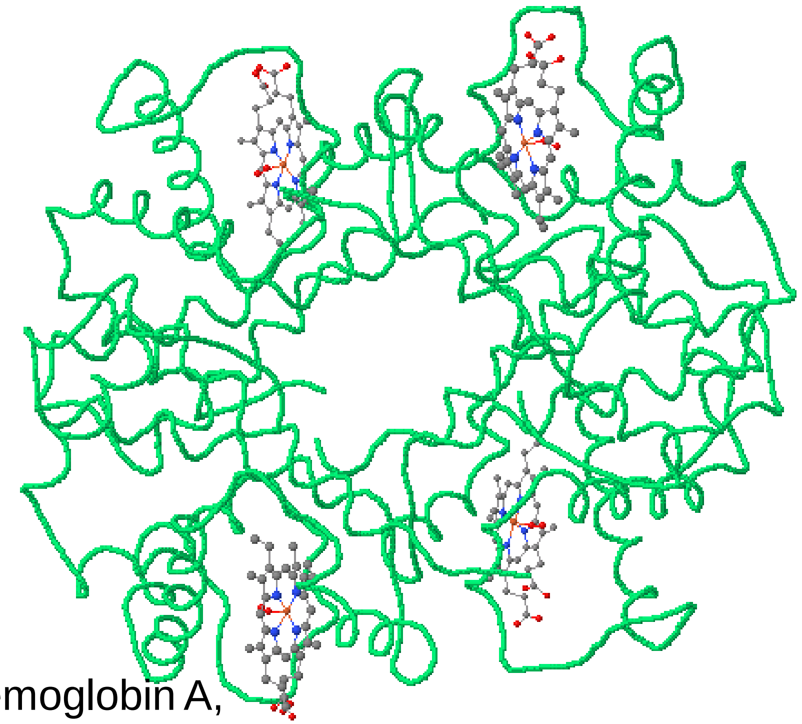


Paralogy example: hemoglobin

- Hemoglobin is a protein complex which transports oxygen
- In humans, hemoglobin consists of four protein subunits and four non-protein heme groups

Sickle cell diseases are caused by mutations in hemoglobin genes

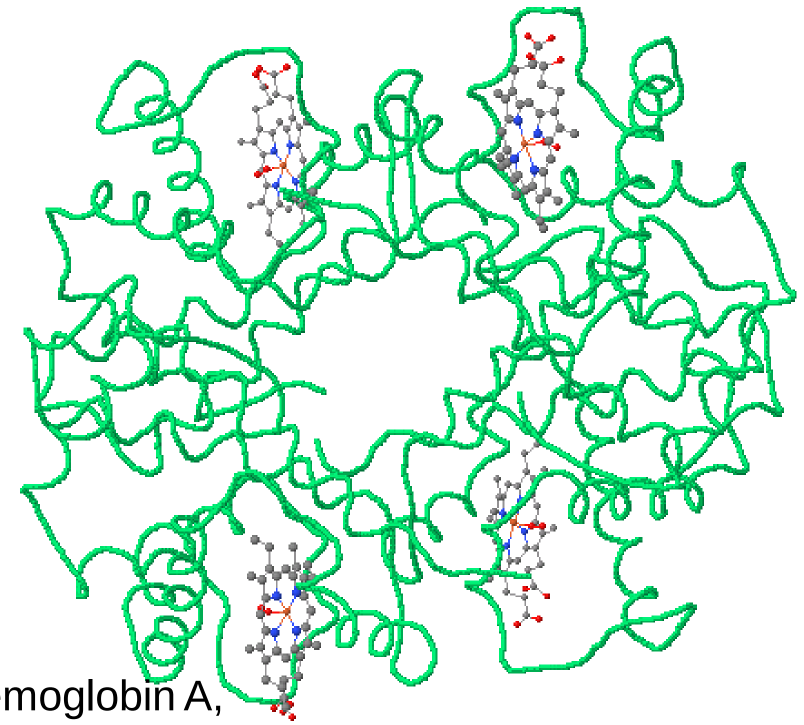
http://en.wikipedia.org/wiki/Image:Sicklecells.jpg



Hemoglobin A, www.rcsb.org/pdb/explore.do?structureId=1GZX

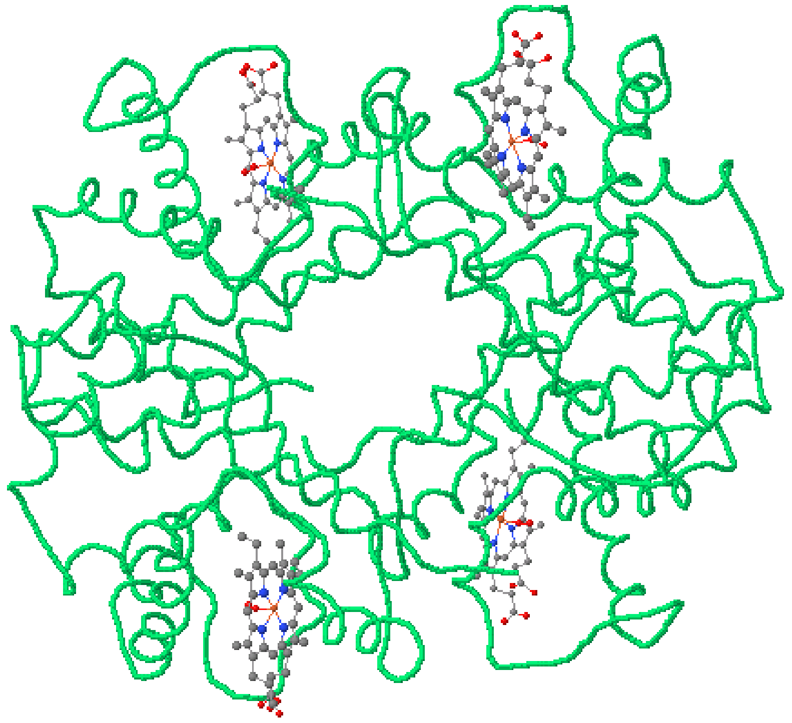
Paralogy example: hemoglobin

- In adults, three types are normally present
  - Hemoglobin A: 2 alpha and 2 beta subunits
  - Hemoglobin A2: 2 alpha and 2 delta subunits
  - Hemoglobin F: 2 alpha and 2 gamma subunits
- Each type of subunit (alpha, beta, gamma, delta) is encoded by a separate gene


Hemoglobin A, www.rcsb.org/pdb/explore.do?structureId=1GZX

Paralogy example: hemoglobin

- The subunit genes are paralogs of each other, i.e., they have a common ancestor gene
- Exercise: hemoglobin human paralogs in NCBI sequence databases
  http://www.ncbi.nlm.nih.gov/sites/entrez?db=Nucleotide
  - Find human hemoglobin alpha, beta, gamma and delta
  - Compare sequences



Hemoglobin A, www.rcsb.org/pdb/explore.do?structureId=1GZX

Orthology example: insulin

- The genes coding for insulin in human (*Homo sapiens*) and mouse (*Mus musculus*) are orthologs:
  - They have a common ancestor gene in the ancestor species of human and mouse
  - Exercise: find insulin orthologs from human and mouse in NCBI sequence databases

Sequence alignment

- Alignment specifies which positions in two sequences match

```
acgtctag   acgtctag   acgtctag
||            |||||   || |||||
actctag-   -actctag   ac-tctag
```

2 matches	5 matches	7 matches
5 mismatches	2 mismatches	0 mismatches
1 not aligned	1 not aligned	1 not aligned

Sequence alignment

- Maximum alignment length is the total length of the two sequences

```
acgtctag-------     -------acgtctag

--------actctag     actctag--------
```

0 matches
0 mismatches
15 not aligned

0 matches
0 mismatches
15 not aligned

Mutations: Insertions, deletions and substitutions

Indel: insertion or deletion of a base with respect to the ancestor sequence

```
acgtctag
   |||||
-actctag
```

Mismatch: substitution (point mutation) of a single base

- Insertions and/or deletions are called *indels*
  - *We can't tell whether the ancestor sequence had a base or not at indel position!*

Problems

- What sorts of alignments should be considered?
- How to score alignments?
- How to find optimal or good scoring alignments?
- How to evaluate the statistical significance of scores?

In this course, we discuss each of these problems briefly.

Sequence Alignment (chapter 6)

- The biological problem
- *Global alignment*
- Local alignment
- Multiple alignment

Global alignment

- Problem: find optimal scoring alignment between two sequences (Needleman & Wunsch 1970)
- Every position in both sequences is included in the alignment
- We give score for each position in alignment

Identity (match)	+1
Substitution (mismatch)	$-\mu$
Indel	$-\delta$

- Total score: sum of position scores

Scoring: Toy example

- Consider two sequences with characters drawn from the English language alphabet: WHAT, WHY

```
WHAT
||
WH-Y
```

S(WHAT/WH-Y) = 1 + 1 – δ – μ

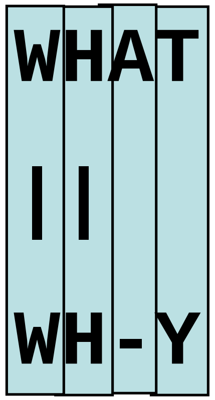
```
WHAT

-WHY
```

S(WHAT/-WHY) = – δ – μ – μ – μ

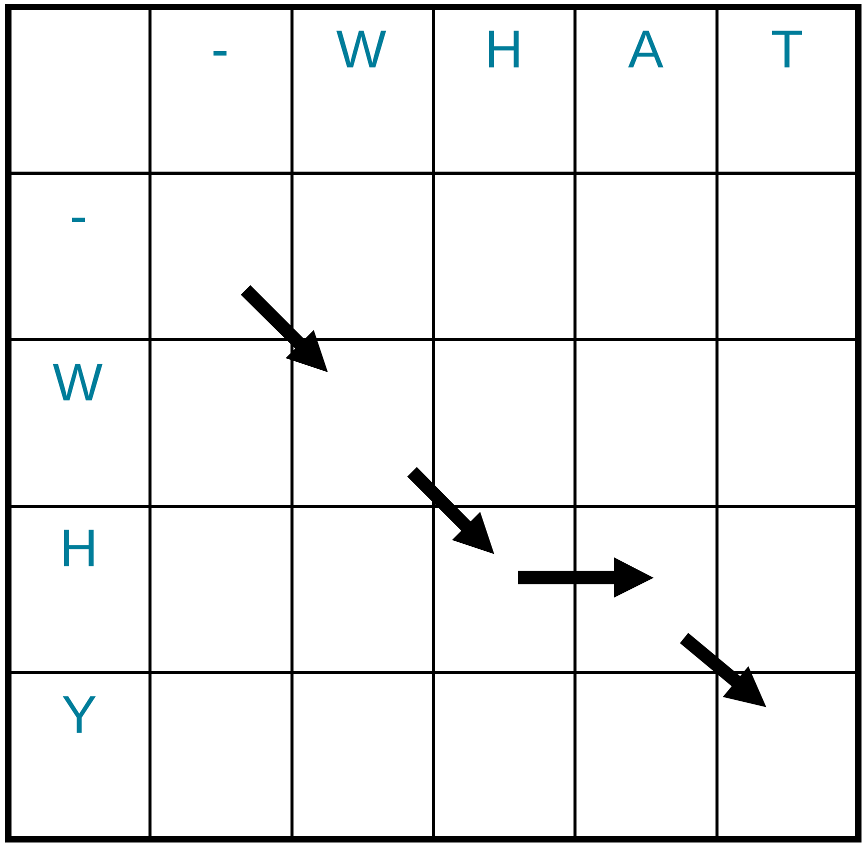
Representing alignments and scores

Alignments can be represented in the following tabular form.

Each alignment corresponds to a path through the table.



```
WHAT
||
WH-Y
```

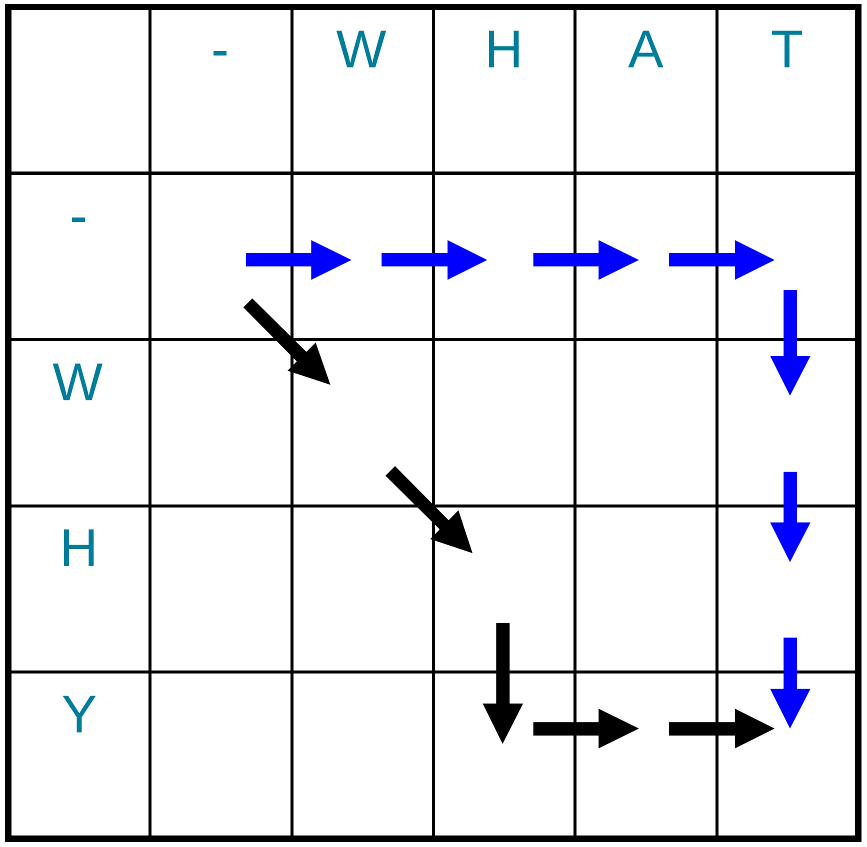


Representing alignments and scores

```
WH-AT
||
WHY--
```

```
WHAT---
```

```
----WHY
```



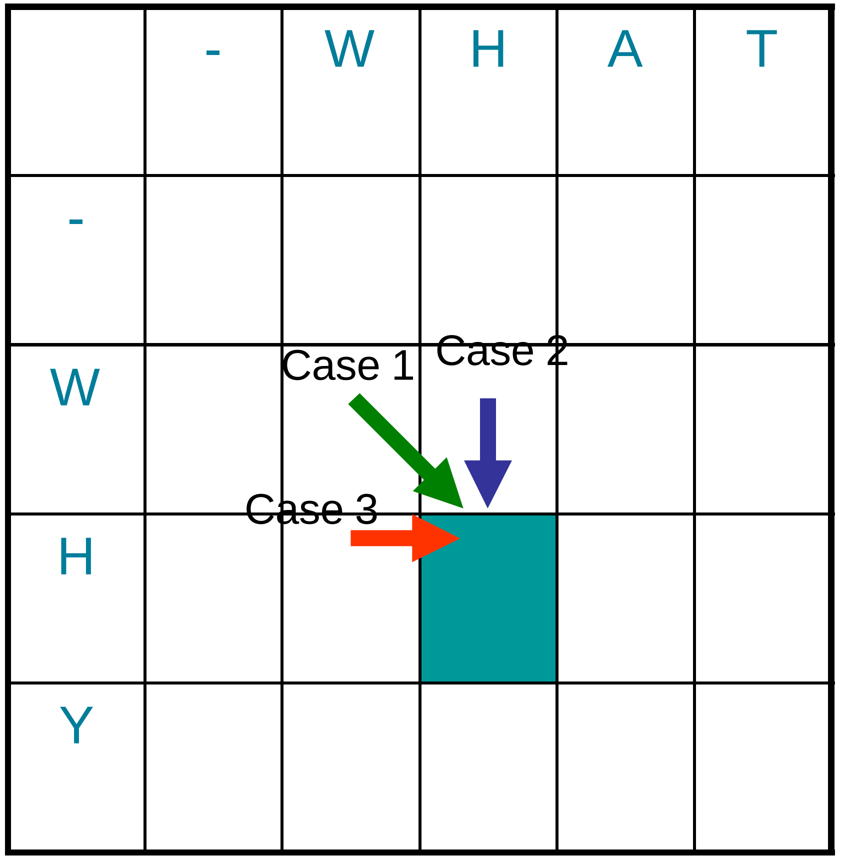
Representing alignments and scores

```
WHAT
||
WH-Y
```

	-	W	H	A	T
-	0				
W		1			
H			2	2-δ	
Y					2-δ-µ

Global alignment score $S_{3,4} = 2\text{-}\delta\text{-}\mu$

Filling the alignment matrix



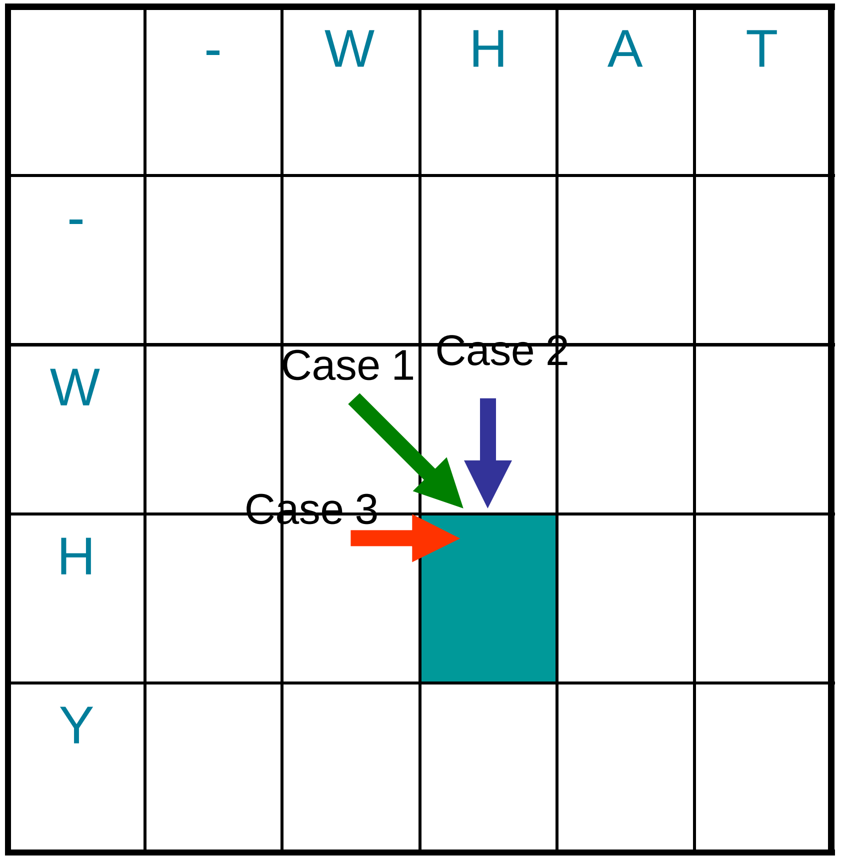
Consider the alignment process at shaded square.

Case 1. Align H against H (match)

Case 2. Align H in WHY against – (indel) in WHAT

Case 3. Align H in WHAT against – (indel) in WHY

Filling the alignment matrix (2)



Scoring the alternatives.

Case 1. $S_{2,2} = S_{1,1} + s(2, 2)$

Case 2. $S_{2,2} = S_{1,2} - \delta$

Case 3. $S_{2,2} = S_{2,1} - \delta$

$s(i, j) = 1$ for matching positions,

$s(i, j) = -\mu$ for substitutions.

Choose the case (path) that yields the maximum score.

Keep track of path choices.

Global alignment: formal development

$A = a_1a_2a_3\ldots a_m,$

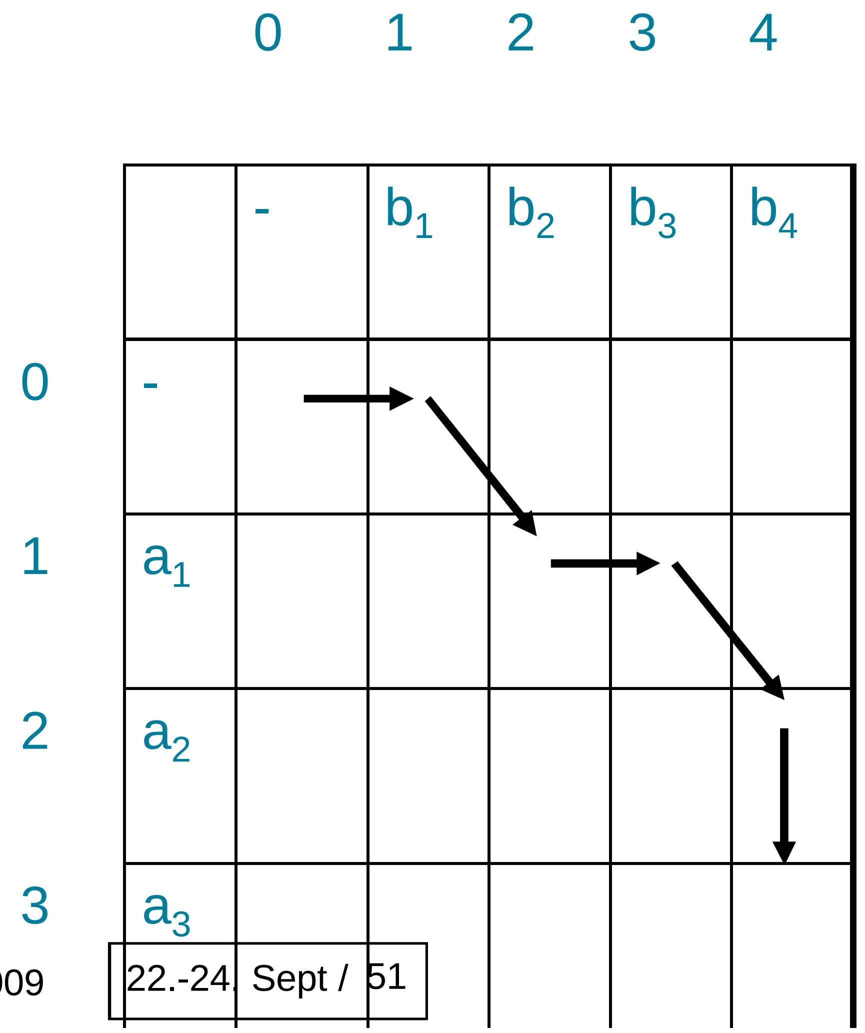
$B = b_1b_2b_3\ldots b_n$

$$\begin{array}{ccccc} b_1 & b_2 & b_3 & b_4 & - \\ - & a_1 & - & a_2 & a_3 \end{array}$$

- Any alignment can be written as a unique path through the matrix

- Score for aligning A and B up to positions i and j:

$S_{i,j} = S(a_1a_2a_3\ldots a_i,\ b_1b_2b_3\ldots b_j)$



Scoring partial alignments

- Alignment of A = $a_1a_2a_3\ldots a_i$ with B = $b_1b_2b_3\ldots b_j$ can be end in three possible ways
  - Case 1: $(a_1a_2\ldots a_{i-1})\ a_i$
    $(b_1b_2\ldots b_{j-1})\ b_j$
  - Case 2: $(a_1a_2\ldots a_{i-1})\ a_i$
    $(b_1b_2\ldots b_j)$ -
  - Case 3: $(a_1a_2\ldots a_i)$ –
    $(b_1b_2\ldots b_{j-1})\ b_j$

Scoring alignments

- Scores for each case:
  - Case 1: $(a_1a_2\ldots a_{i-1})\ a_i$
    $(b_1b_2\ldots b_{j-1})\ b_j$
  - Case 2: $(a_1a_2\ldots a_{i-1})\ a_i$
    $(b_1b_2\ldots b_j)$ –
  - Case 3: $(a_1a_2\ldots a_i)$ –
    $(b_1b_2\ldots b_{j-1})\ b_j$

$$s(a_i, b_j) = \begin{cases} +1 \text{ if } a_i = b_j \\ -\mu \text{ otherwise} \end{cases}$$

$$s(a_i, -) = s(-, b_j) = -\delta$$

Scoring alignments (2)

- First row and first column correspond to initial alignment against indels:

  $S(i, 0) = -i\,\delta$

  $S(0, j) = -j\,\delta$

- Optimal global alignment score $S(A, B) = S_{m,n}$

		0	1	2	3	4
		-	b_1	b_2	b_3	b_4
0	-	0	$-\delta$	-2δ	-3δ	-4δ
1	a_1	$-\delta$				
2	a_2	-2δ				
3	a_3	-3δ				

Algorithm for global alignment

```
Input sequences A, B, $m = |A|$, $n = |B|$
Set $S_{i,0} := -\delta i$ for all i
Set $S_{0,j} := -\delta j$ for all j
for i := 1 to m
    for j := 1 to n
        $S_{i,j} := \max\{S_{i-1,j} - \delta, S_{i-1,j-1} + s(a_i,b_j), S_{i,j-1} - \delta\}$
    end
end
```

Algorithm takes O(mn) time

Global alignment: example

$\mu = 1$

$\delta = 2$

	-	T	G	G	T	G
-	0	-2	-4	-6	-8	-10
A	-2					
T	-4					
C	-6					
G	-8					
T	-10					?

Global alignment: example

$\mu = 1$

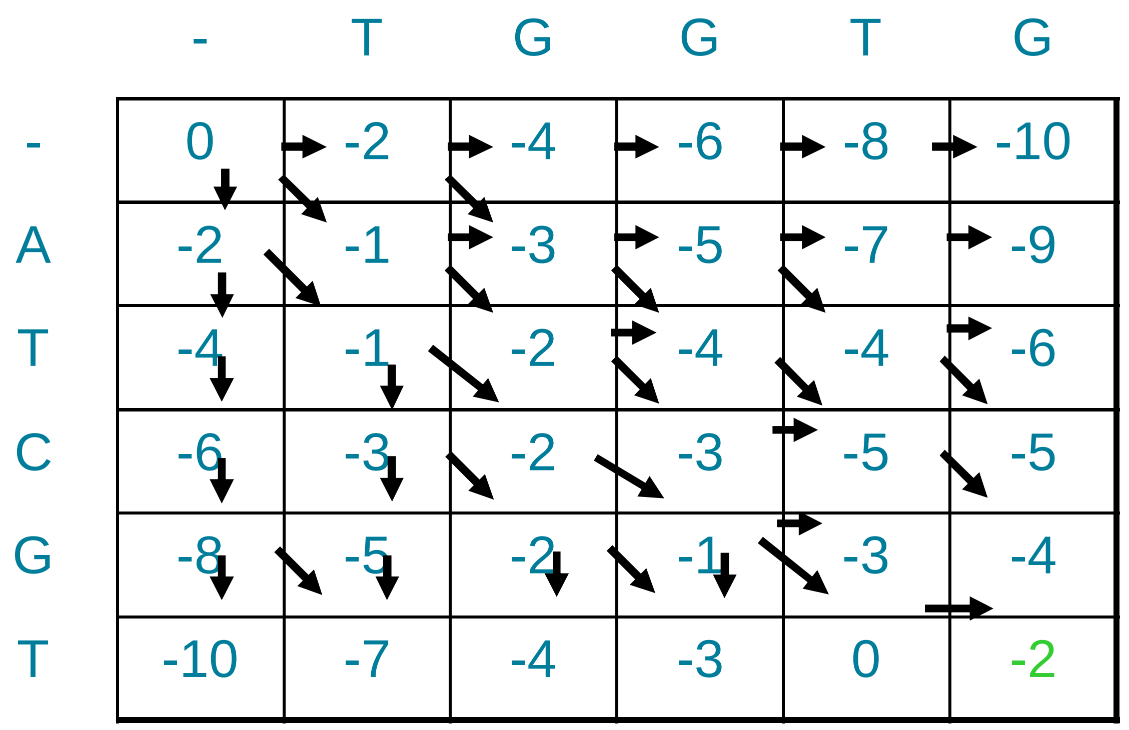
$\delta = 2$

	-	T	G	G	T	G
-	0	-2	-4	-6	-8	-10
A	-2	$\searrow \downarrow \rightarrow$ -1	$\searrow \downarrow \rightarrow$ -3			
T	-4					
C	-6					
G	-8					
T	-10					?

Global alignment: example (2)

$\mu = 1$

$\delta = 2$



	-	T	G	G	T	G
-	0	-2	-4	-6	-8	-10
A	-2	-1	-3	-5	-7	-9
T	-4	-1	-2	-4	-4	-6
C	-6	-3	-2	-3	-5	-5
G	-8	-5	-2	-1	-3	-4
T	-10	-7	-4	-3	0	-2

```
ATCGT-
 | ||
-TGGTG
```

Sequence Alignment (chapter 6)

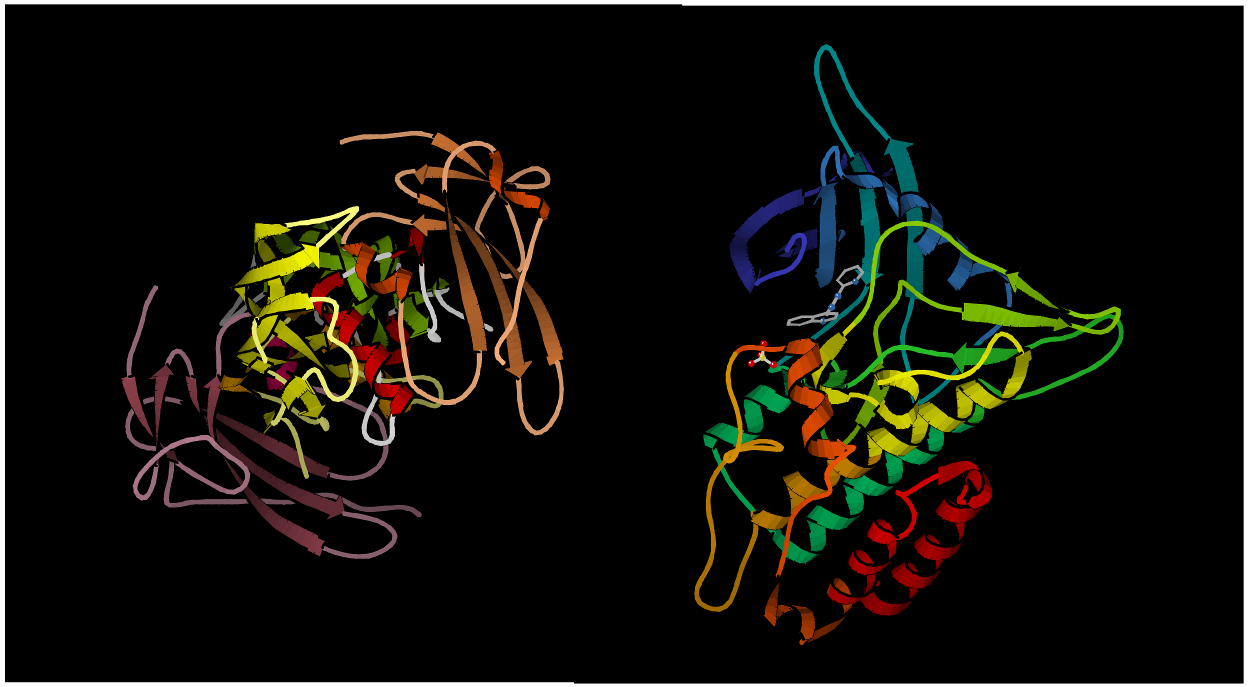
- The biological problem
- Global alignment
- *Local alignment*
- Multiple alignment

Local alignment: rationale

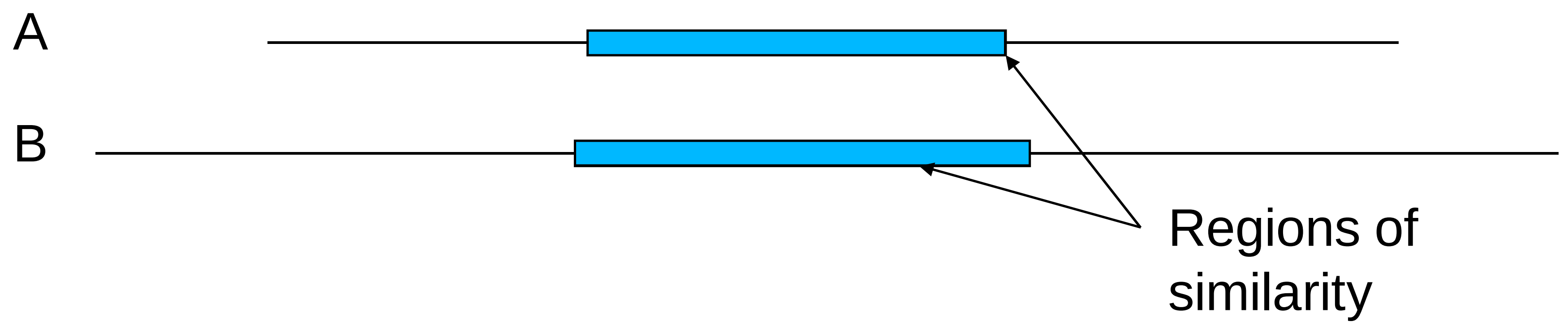
- Otherwise dissimilar proteins may have local regions of similarity
  -> Proteins may share a function

Human bone morphogenic protein receptor type II precursor (left) has a 300 aa region that resembles 291 aa region in TGF-β receptor (right).

The shared function here is protein kinase.



Local alignment: rationale



- Global alignment would be inadequate
- Problem: find the highest scoring *local* alignment between two sequences
- Previous algorithm with minor modifications solves this problem (Smith & Waterman 1981)

From global to local alignment

- Modifications to the global alignment algorithm
  - Look for the highest-scoring path **in** the alignment matrix (not necessarily through the matrix), or in other words:
  - Allow preceding and trailing indels without penalty

Scoring local alignments

$A = a_1a_2a_3\ldots a_m$, $B = b_1b_2b_3\ldots b_n$

Let I and J be intervals (substrings) of A and B, respectively:

$$I \subset A \qquad J \subset B$$

Best local alignment score:

$$M(A, B) = \max\{S(I, J) : I \subset A, J \subset B\}$$

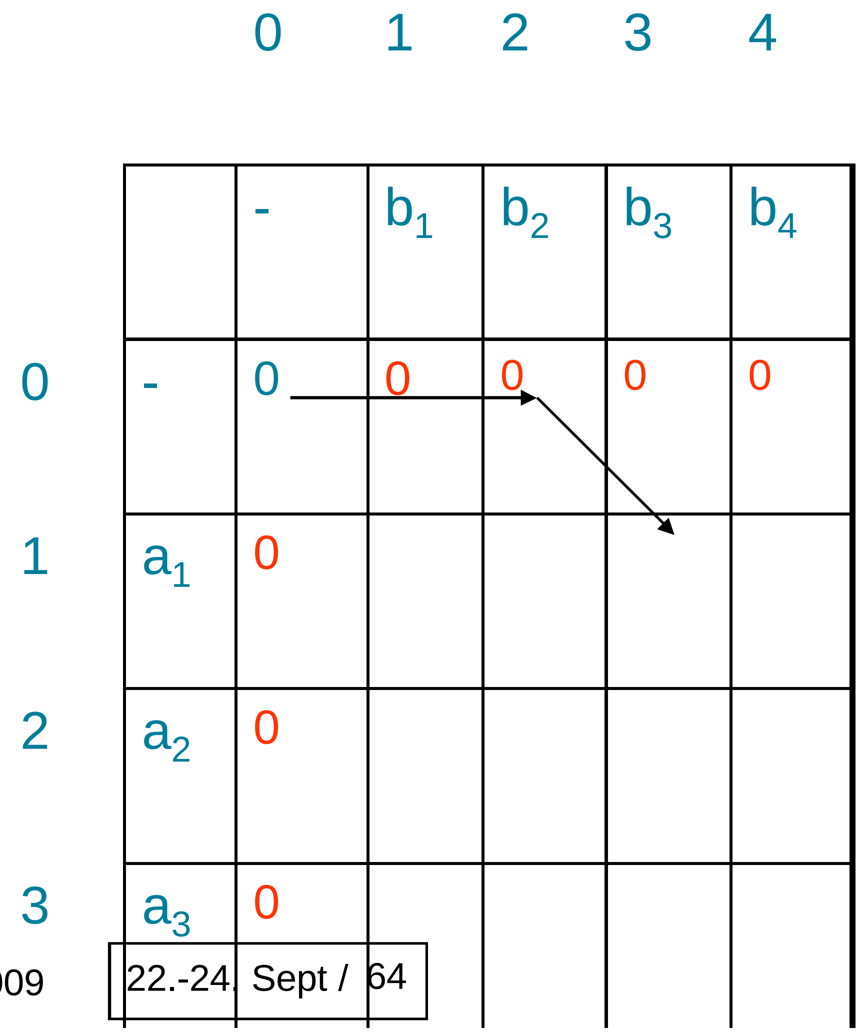
where S(I, J) is the alignment score for substrings I and J.

Allowing preceding and trailing indels

- First row and column initialised to zero:

  $M_{i,0} = M_{0,j} = 0$

```
b1 b2 b3
-  -  a1
```



		0	1	2	3	4
		-	b_1	b_2	b_3	b_4
0	-	0	0	0	0	0
1	a_1	0				
2	a_2	0				
3	a_3	0				

Recursion for local alignment

- $M_{i,j} = \max \{$
  $M_{i-1,j-1} + s(a_i, b_j),$
  $M_{i-1,j} - \delta,$
  $M_{i,j-1} - \delta,$
  0
  $\}$

Allow alignment to start anywhere in sequences

	-	T	G	G	T	G
-	0	0	0	0	0	0
A	0	0	0	0	0	0
T	0	1	0	0	1	0
C	0	0	0	0	0	0
G	0	0	1	1	0	1
T	0	1	0	0	2	0

Finding best local alignment

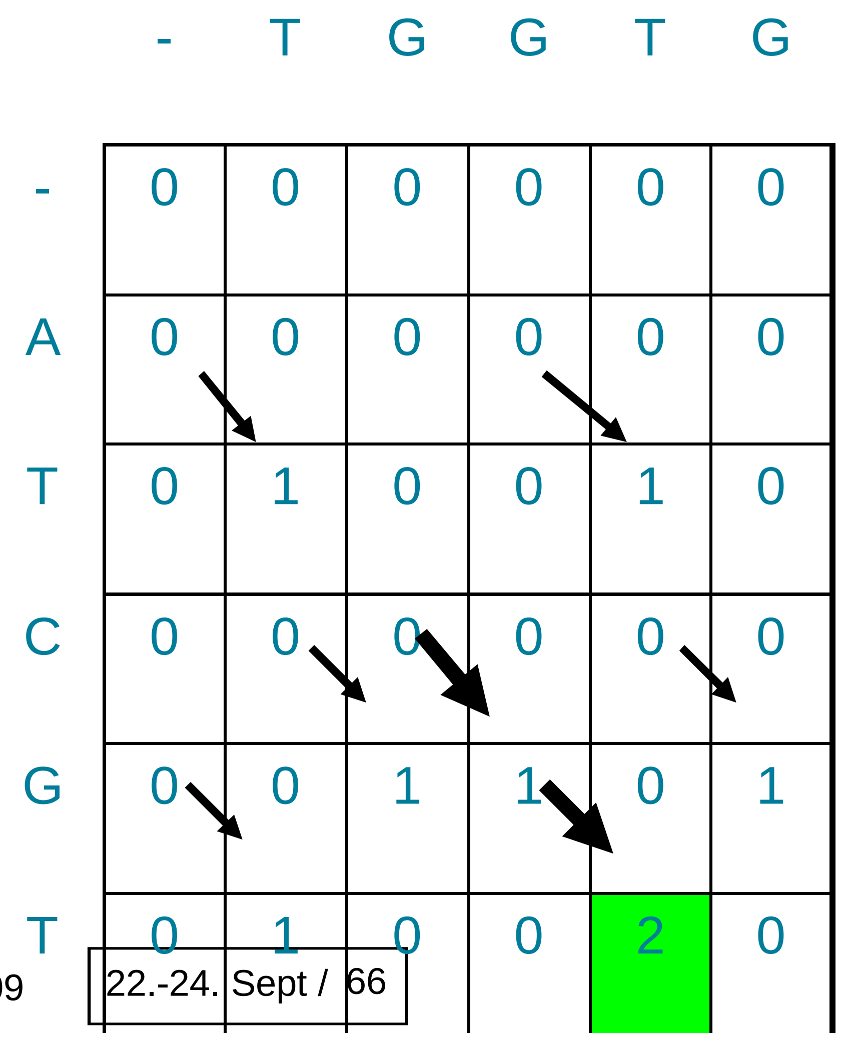
- Optimal score is the highest value in the matrix

$$M(A,B) = \max\{S(I,J) : I \subset A, J \subset B\}$$

$$= \max_{i,j} M_{i,j}$$

- Best local alignment can be found by backtracking from the highest value in M

- What is the best local alignment in this example?



	-	T	G	G	T	G
-	0	0	0	0	0	0
A	0	0	0	0	0	0
T	0	1	0	0	1	0
C	0	0	0	0	0	0
G	0	0	1	1	0	1
T	0	1	0	0	2	0

Local alignment: example

$$
M_{i,j} = \max \{
\begin{array}{l}
M_{i-1,j-1} + s(a_i, b_i), \\
M_{i-1,j} - \delta, \\
M_{i,j-1} - \delta, \\
0
\end{array}
\}
$$

		0	1	2	3	4	5	6	7	8	9	10
		-	G	G	C	T	C	A	A	T	C	A
0	-	0	0	0	0	0	0	0	0	0	0	0
1	A	0	↘ ↓ → 0									
2	C	0										
3	C	0										
4	T	0										
5	A	0										
6	A	0										
7	G	0										
8	G	0										

Scoring (for example)
Match: +2
Mismatch: -1
Indel: -2

Local alignment: example

$$M_{i,j} = \max \{ \\ M_{i-1,j-1} + s(a_i, b_j), \\ M_{i-1,j} - \delta, \\ M_{i,j-1} - \delta, \\ 0 \\ \}$$

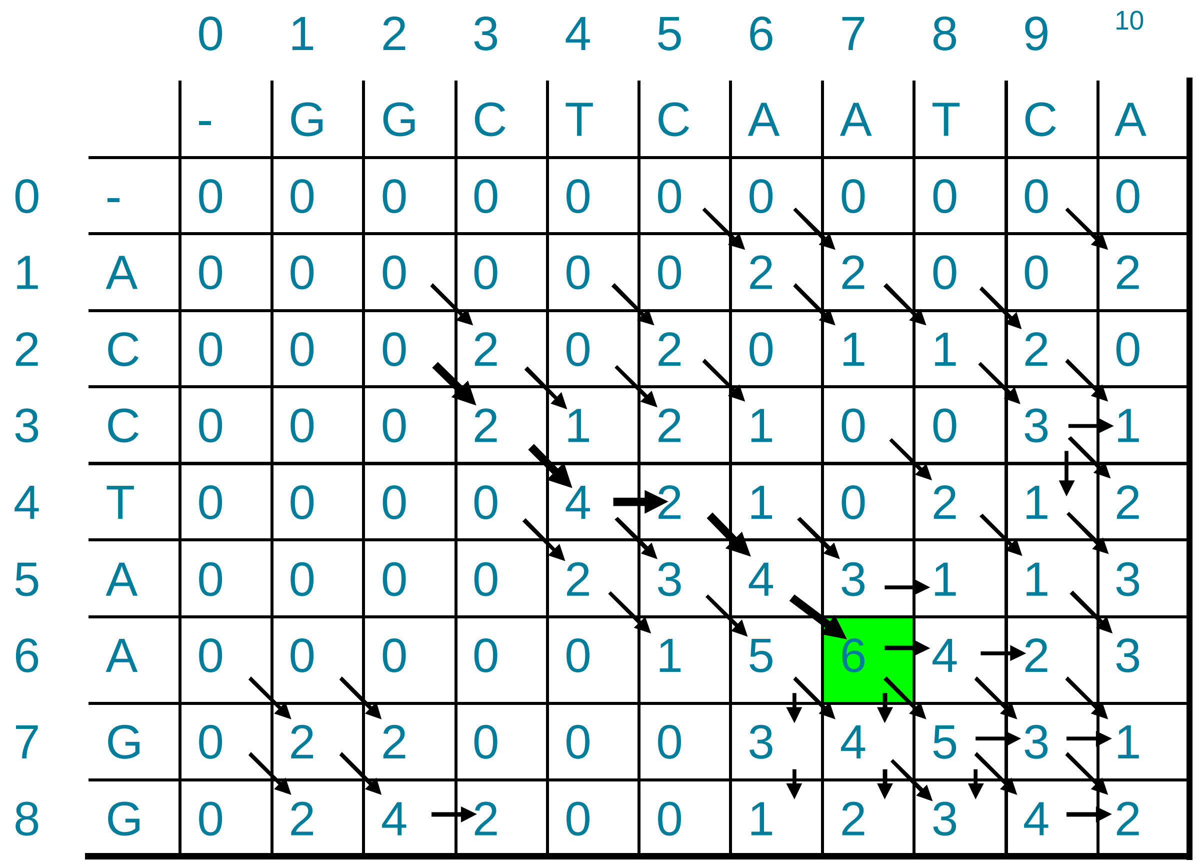
Scoring (for example)
Match: +2
Mismatch: -1
Indel: -2

		0	1	2	3	4	5	6	7	8	9	10
		-	G	G	C	T	C	A	A	T	C	A
0	-	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	0 $\rightarrow$	$\searrow\downarrow$ 2				
2	C	0										
3	C	0										
4	T	0										
5	A	0										
6	A	0										
7	G	0										
8	G	0										

Local alignment: example

Optimal local alignment:

C T – A A
C T C A A



		0	1	2	3	4	5	6	7	8	9	10
		-	G	G	C	T	C	A	A	T	C	A
0	-	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	0	0	2	2	0	0	2
2	C	0	0	0	2	0	2	0	1	1	2	0
3	C	0	0	0	2	1	2	1	0	0	3	1
4	T	0	0	0	0	4	2	1	0	2	1	2
5	A	0	0	0	0	2	3	4	3	1	1	3
6	A	0	0	0	0	0	1	5	6	4	2	3
7	G	0	2	2	0	0	0	3	4	5	3	1
8	G	0	2	4	2	0	0	1	2	3	4	2

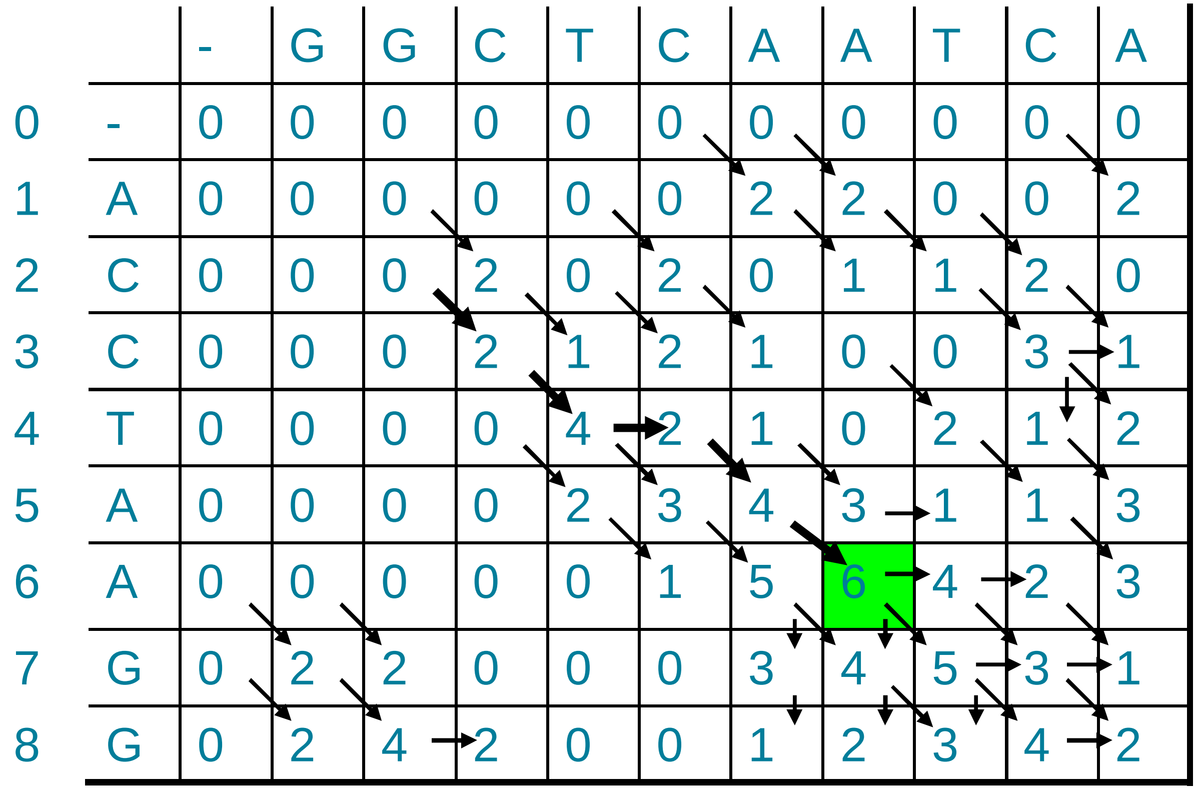
Scoring (for example)
Match: +2
Mismatch: -1
Indel: -2

Multiple optimal alignments
Non-optimal, good-scoring alignments

How can you find

1. Optimal alignments if more than one exist?

2. Non-optimal, good-scoring alignments?



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