Introduction to Bioinformatics for Computer Scientists

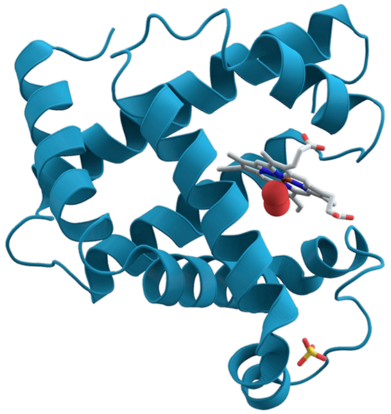
Lecture 3

Pair-wise Sequence Alignment

Lukas Hübner, PhD student
lukas.huebner@h-its.org

slides by
Benoit Morel, postdoc

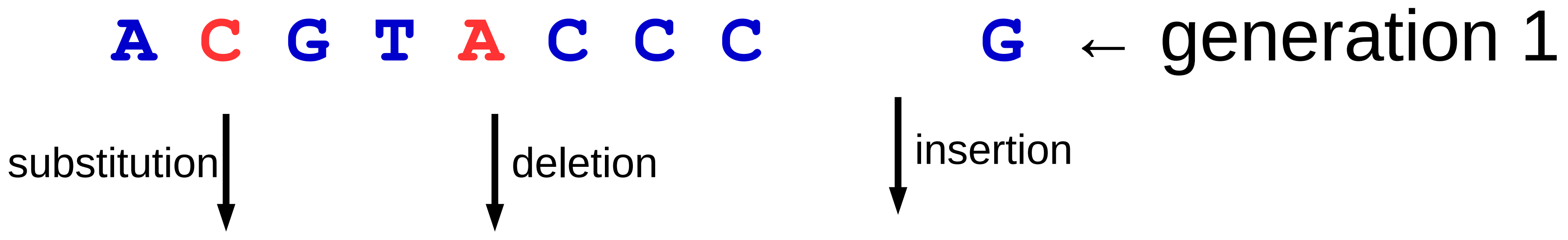
DNA and protein sequences are strings

- DNA: AACCTGTTGTCAAATG
- Protein:  TTETTSFLIFETAVKNT

Sequences evolve

A C G T A C C C G ← generation 1

Sequences evolve


A C G T A C C C G ← generation 1
substitution
deletion
insertion
A T G T C C C T A G ← generation 2

(and their lengths change)

Pair-wise sequence alignment

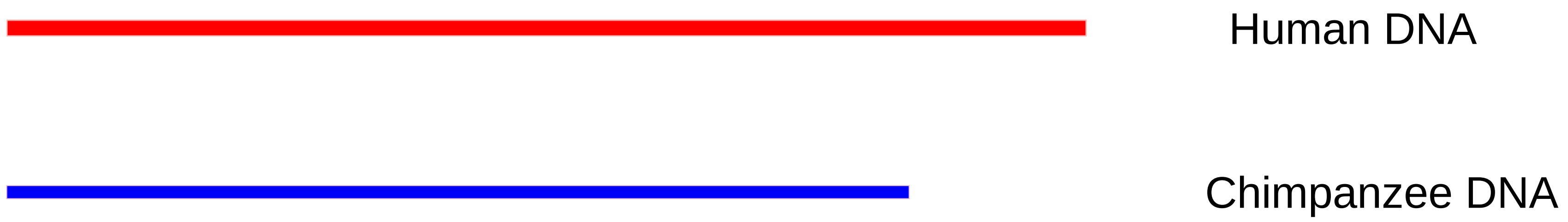
Compare two sequences to infer their similarity

Example: alignment between ‘GCGACGTCC’ and ‘GCGATAC’

$$\begin{array}{cccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ x = & \mathtt{G} & \mathtt{C} & \mathtt{G} & \mathtt{A} & \mathtt{C} & \mathtt{G} & \mathtt{T} & \mathtt{C} & \mathtt{C} \\ & | & | & | & | & & & | & . & | \\ y = & \mathtt{G} & \mathtt{C} & \mathtt{G} & \mathtt{A} & - & - & \mathtt{T} & \mathtt{A} & \mathtt{C} \end{array}$$

Example 1: Measure DNA similarity

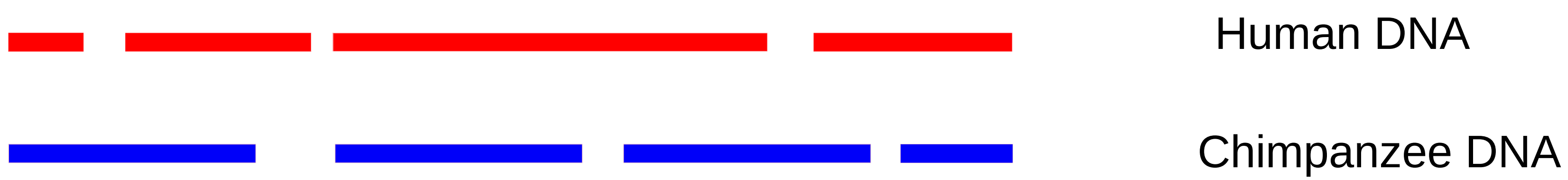
How similar are human and chimpanzee at the DNA level?



Example 1: Measure DNA similarity

How similar are human and chimpanzee at the DNA level?

We first need to align their DNA sequences!

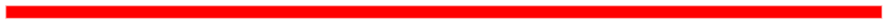


(now nucleotids at the same position are comparable)

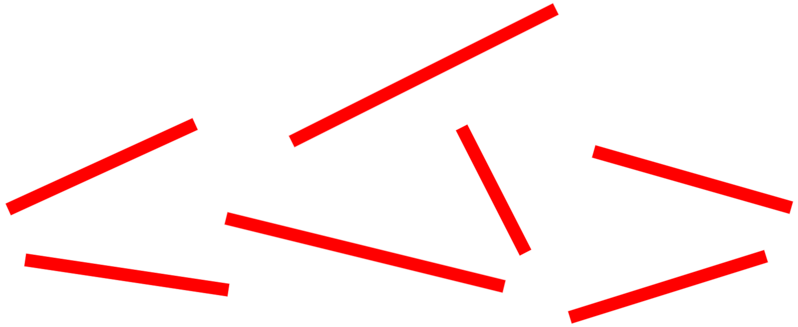
Example 2: genome assembly

Individual to sequence

Sequencing machine

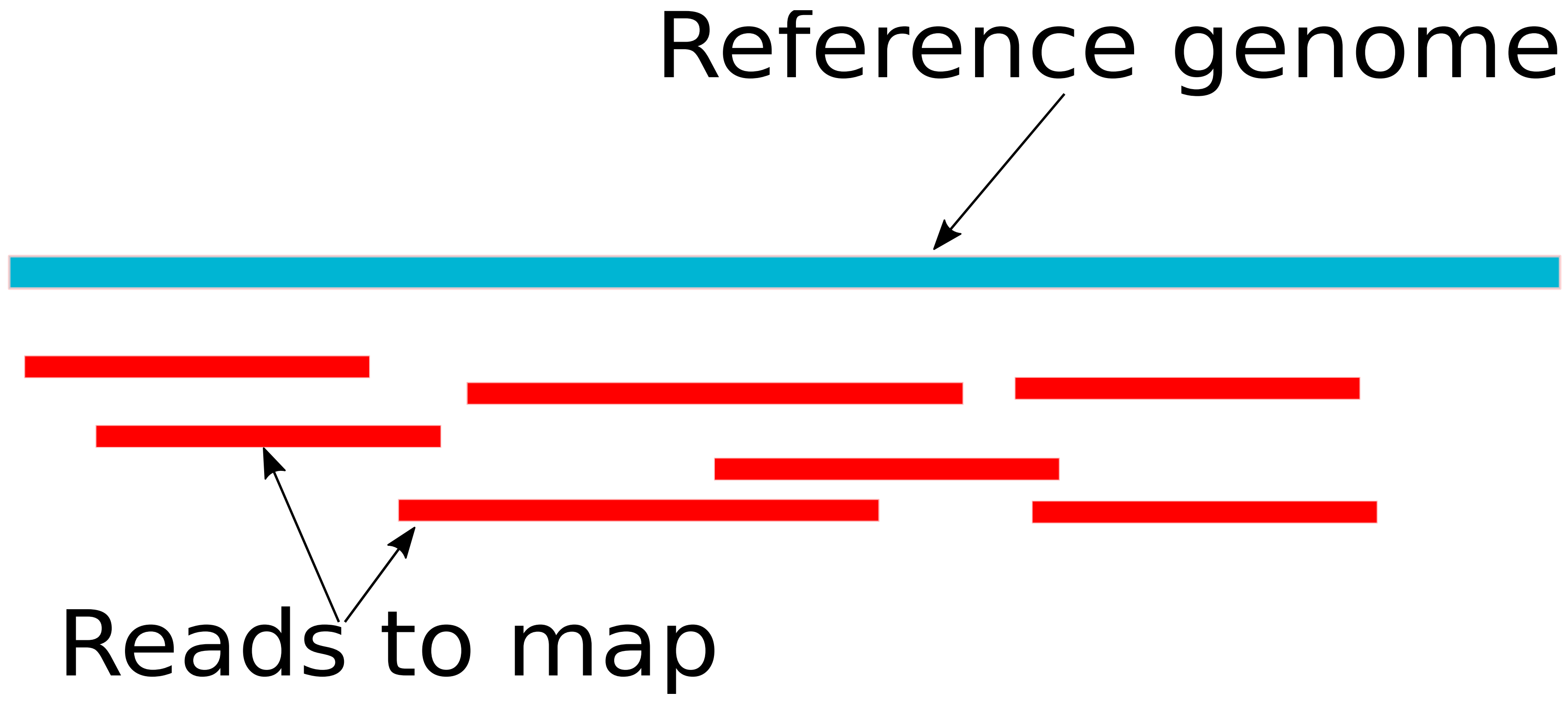


Assembled genome



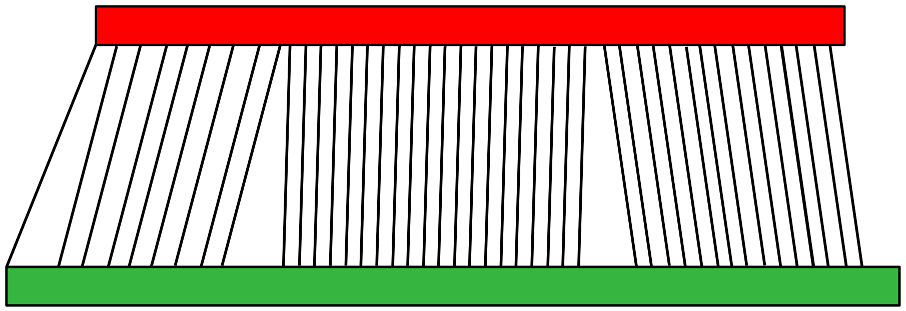
Unordered reads
(short DNA sequences)

Example 2: genome assembly

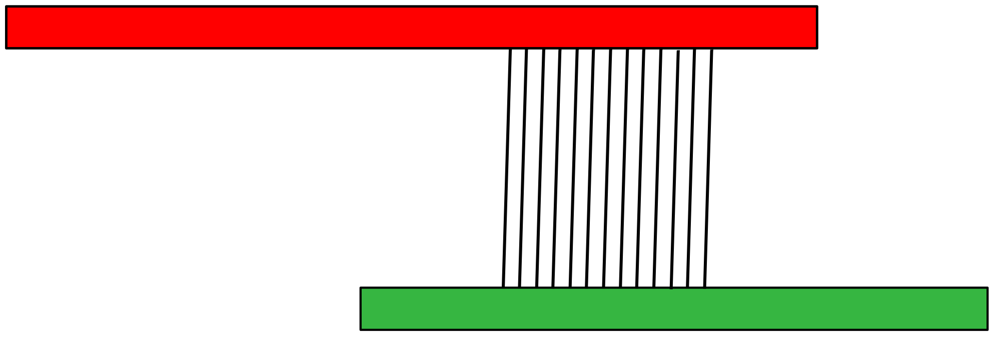


Global and local alignments

Global alignment: align the full strings



Local alignment: align similar substrings



Dynamic programming

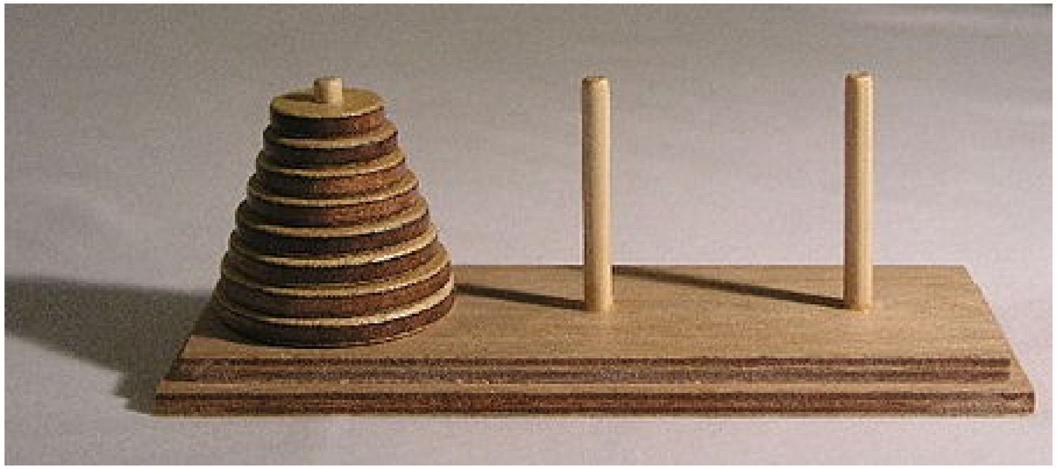
- Break down a complicated problem into simpler subproblems (typically with recursion)
- Cache the results of the recursive calls

Dynamic programming

Break down a complicated problem into simpler subproblems (typically with recursion)

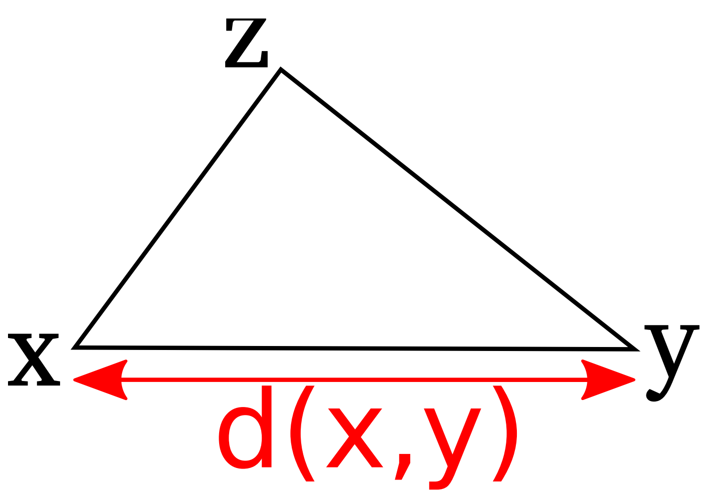
Example:

Tower of Hanoi algorithm



Distance

- A function d is a distance if the following conditions are satisfied for all element x and y:
  - Positivity: d(x,y) >= 0
  - Separation: d(x,y) = 0 <=> x = y
  - Symmetry: d(x,y) = d(y,x)
  - Triangle inequality: d(x,y) <= d(x,z) + d(z,y) for all elements z



Hamming distance

- Defined for strings of same length
- Counts the number of characters that differ
- Linear complexity

PING	**HAMMING**	**ACGTTGGGTT**
PONG	**LEMMING**	**ACGATGCATT**
d = 1	d = 2	d = 3

Is hamming biologically relevant?

`X = ACTATATATACG`

`Y = CTATATATACGT`

d = 12, the distance between **`X`** and **`Y`** is maximal

Is hamming biologically relevant?

```
X = ACTATATATACG
Y = CTATATATACGT
```

d = 12, the distance between **X** and **Y** is maximal

… but **X** and **Y** are very similar!

```
X = ACTATATATACG-
Y = -CTATATATACGT
```

(Hamming distance is actually relevant, but not to compare “raw” sequences)

Edit operations: substitution, insertion and deletion

Substition of one letter x by another letter y

A C G T G C

$\downarrow$

A C G A G C

Edit operations: substitution, insertion and deletion

Insertion of one letter

A C G – G C

$\downarrow$

A C G A G C

Edit operations: substitution, insertion and deletion

Deletion of one letter

$$\begin{tikzcd} A & C & G & T \arrow[d] & G & C \\ A & C & G & - & G & C \end{tikzcd}$$

Edit distance

$\delta e(x,y)$ = <u>minimum</u> number of (edits) operations to transform x into y

Edit distance

Example: compute the edit distance between "SALADS" and "BALLAD"

Edit distance

Example: compute the edit distance between

“SALADS” and “BALLAD”

SALADS → BALADS	Subs S → B
BALADS → BALLADS	Insert L
BALLADS → BALLAD	Del S

Edit distance = 3

Alignment definition

Result of inserting gaps in both strings such that they have the same length

Alignments between

x='ACGA' and y='ATGCTA':

```
ACGA--          A------CGA      A--CGA
ATGCTA          -ATGCTA--       ATGCTA
```

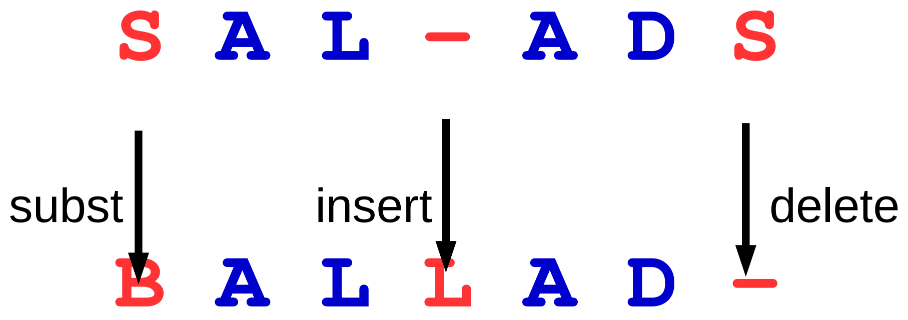
What is a good alignment?

We can (for instance) score an alignment with the hamming distance.

```
ACGA--          A------CGA          A--CGA
ATGCTA          -ATGCTA---          ATGCTA
  d=4              d=10               d=3
```

Aligning sequences using edit distance

Computing the alignment with minimum hamming distance = Computing the edit distance and storing the sequence of edit operations



hamming(SAL-ADS,BALLAD-) = edit(SALADS, BALLAD)

How to compute the edit distance between two strings **X** and **Y**?

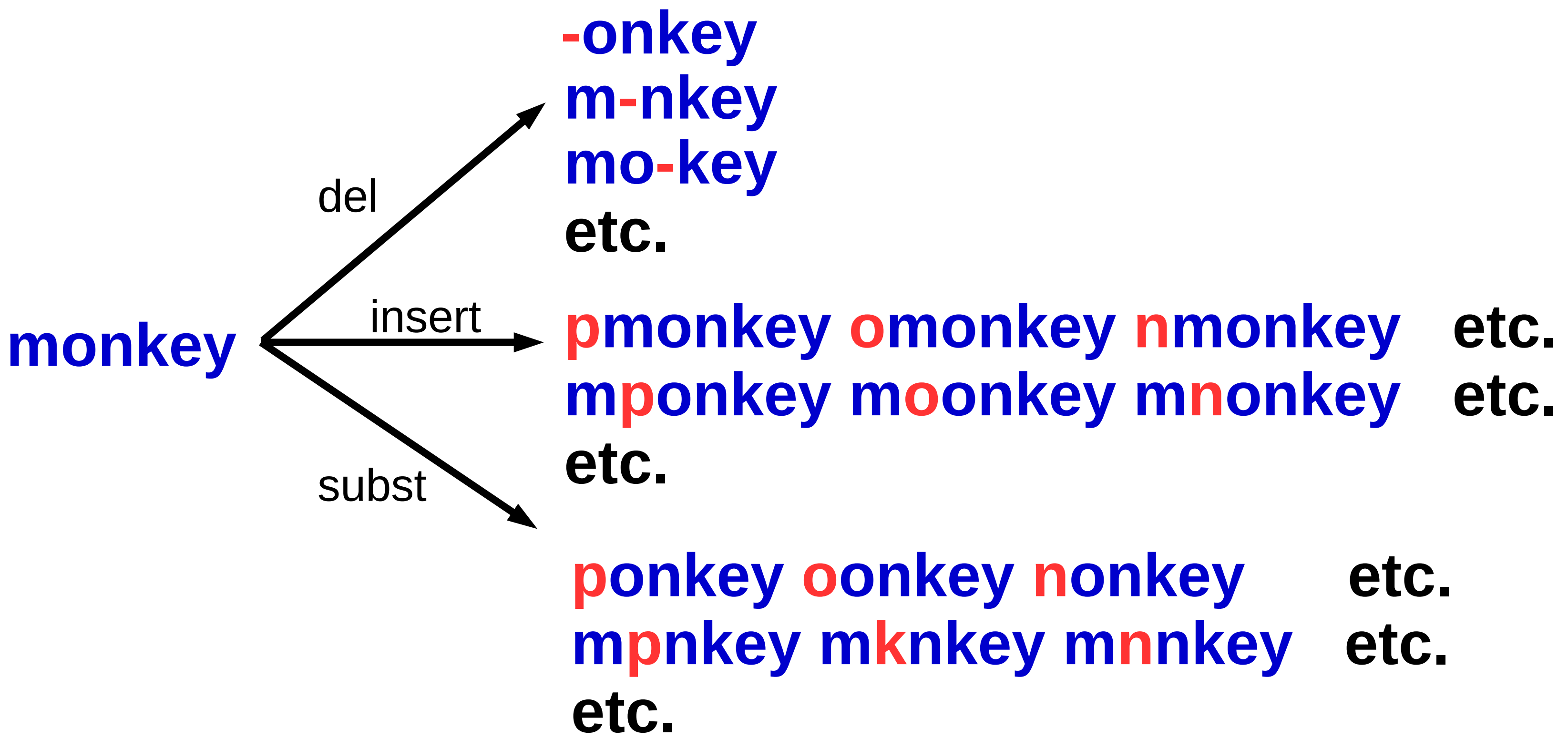
For instance, how to compute the distance between **poney** and **monkey**?

Naive bruteforce

For each distance from 1 to |X|, try every possible combinations of edit operations

Naive bruteforce

First iteration:



monkey

-onkey
m-nkey
mo-key
etc.

pmonkey omonkey nmonkey etc.
mponkey moonkey mnonkey etc.
etc.

ponkey oonkey nonkey etc.
mpnkey mknkey mnnkey etc.
etc.

Naive bruteforce

Second iteration: repeat the same procedure from each output of the previous iteration

→ Non-polynomial time complexity (with respect to |X| and |Y|)

Dynamic programming

Recursion on the prefixes of X and Y

We need:

- A recursion formula
- Trivial edge case to end the recursion

Edge case

$\delta e(\mathbf{X}, \varepsilon) = |\mathbf{X}|$ (delete all letters of **X**)

$\delta e(\boldsymbol{\varepsilon}, \mathbf{Y}) = |\mathbf{Y}|$ (insert all letters of **Y**)

Dynamic programming

Recursion on the prefixes of x and y

We need:

- A recursion formula
- Trivial edge case to end the recursion

Levenstein formula

Let F(i,j) be the edit distance between the prefixes of X of size i and the prefix of Y of size j.

```
F(i,j) =   min (
                   F(i, j – 1) + 1,
                   F(i – 1, j) + 1,
                   F(i – 1, j – 1) + 1 * (X[i] != Y[j])
               )
```

Levenstein formula

```
δe(monkey,poney) = min (
        δe(monkey,pone) + 1,
        δe(monke,poney) + 1,
        δe(monke,pone) + 1 * (y != y)
)
```

First term

$$\text{monkey}\ \overset{\text{recursion}}{\boxed{\rightarrow \ldots \rightarrow}}\ \text{pone}\ \overset{\text{Insertion}}{\boxed{\rightarrow}}\ \text{poney}$$

$$\delta e(\text{monkey},\text{pone}) + 1$$

$$F(i, j - 1) + 1$$

Second term

$$\text{monkey} \overset{\text{deletion}}{\boxed{\rightarrow}} \text{monke} \overset{\text{recursion}}{\boxed{\rightarrow \dots \rightarrow}} \text{poney}$$

$$\delta e(\text{monke}, \text{poney}) + 1$$

$$F(i - 1, j) + 1$$

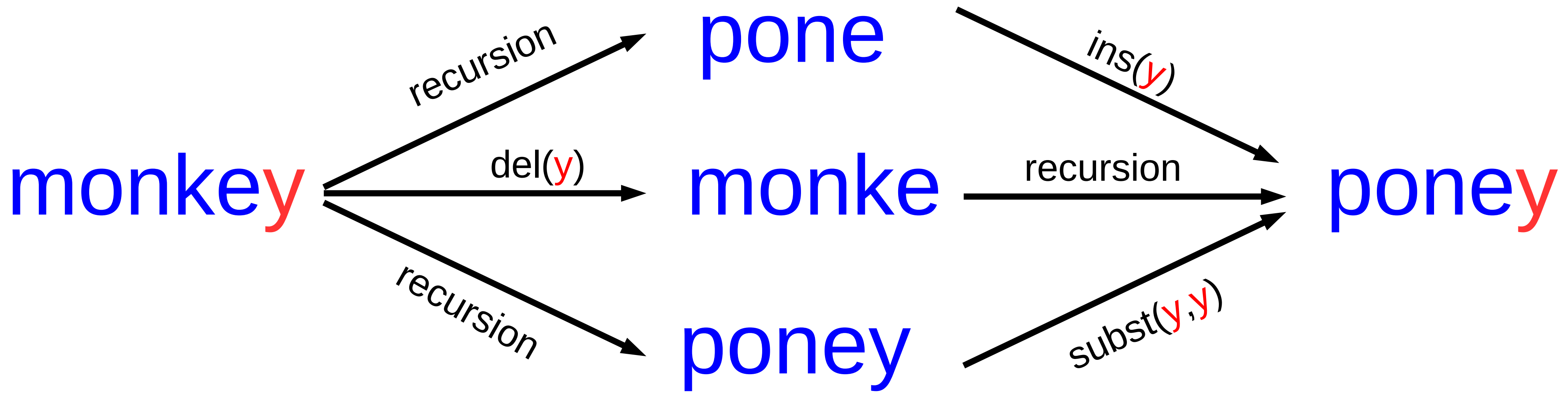
Third term

$$\text{monke(y)}\ \overset{\text{recursion}}{\boxed{\rightarrow \ \ldots \ \rightarrow}}\ \text{pone(y)}\ \overset{\text{substitution?}}{\boxed{\rightarrow}}\ \text{poney}$$

δe(monke,pone) + 0

F(i - 1, j – 1) + 1 * (X[i] != Y[j])

Take the best of the three paths



When to stop?

Stop the recursion when one prefix is empty:

F(0, j) = j

F(i, 0) = i

remember: $\delta e(\mathbf{X}, \varepsilon) = |\mathbf{X}| = \delta e(\varepsilon, \mathbf{X})$

Levenstein formula recursion convergence

```
F(i,j) =   min (  F(i, j – 1) + 1,
                  F(i – 1, j) + 1,
                  F(i – 1, j – 1) + 1 * (X[i] != Y[j]) )
```

The quantity (i+j) decreases at each step, until i=0 or j=0, which ends the recursion

Let’s have a look at the functions calls

$$\begin{tikzcd}
& \boxed{\delta(\text{‘ABC’}, \text{‘ABE’})} \arrow[dl] \arrow[d] \arrow[dr] & \\
\delta(\text{‘ABC’}, \text{‘AB’}) & \boxed{\delta(\text{‘AB’}, \text{‘ABE’})} & \delta(\text{‘AB’}, \text{‘AB’})
\end{tikzcd}$$

Naive implementation → exponential complexity

$$\begin{tikzcd}
 & \boxed{\delta(\text{‘ABC’, ‘ABE’})} \arrow[dl] \arrow[d] \arrow[dr] & \\
\delta(\text{‘ABC’, ‘AB’}) & \boxed{\delta(\text{‘AB’, ‘ABE’})} \arrow[dl] \arrow[d] \arrow[dr] & \delta(\text{‘AB’, ‘AB’}) \\
\delta(\text{‘A’, ‘ABE’}) & \boxed{\delta(\text{‘AB’, ‘AB’})} \arrow[dl] \arrow[d] \arrow[dr] & \delta(\text{‘A’, ‘AB’}) \\
\delta(\text{‘AB’, ‘A’}) & \boxed{\delta(\text{‘A’, ‘AB’})} & \delta(\text{‘A’, ‘A’})
\end{tikzcd}$$

Redundant computations !

$$\begin{tikzcd}
 & \boxed{\delta(\text{‘ABC’, ‘ABE’})} \arrow[dl] \arrow[d] \arrow[dr] & \\
\delta(\text{‘ABC’, ‘AB’}) & \boxed{\delta(\text{‘AB’, ‘ABE’})} \arrow[dl] \arrow[d] \arrow[dr] & \delta(\text{‘AB’, ‘AB’}) \\
\delta(\text{‘A’, ‘ABE’}) & \boxed{\delta(\text{‘AB’, ‘AB’})} \arrow[dl] \arrow[d] \arrow[dr] & \delta(\text{‘A’, ‘AB’}) \\
\delta(\text{‘AB’, ‘A’}) & \boxed{\delta(\text{‘A’, ‘AB’})} & \delta(\text{‘A’, ‘A’})
\end{tikzcd}$$

Dynamic programming

Be careful not to blindly implement the recursion!

Store intermediate results in a table to avoid redundant computations

Needleman-Wunsch algorithm

→ Computes the edit distance

→ Finds the best alignment

Stores intermediate results in a table to save computations

Needleman-Wunsch algorithm

Store in a table T the edit distance between all the possible prefixes.

		M	O	N	K	E	Y
M							
O							
N							
E					🟦		
Y							

T[4][4] = Edit distance between MONE and MONK

Needleman-Wunsch algorithm

Edit distance between x=money and y=monkey with dynamic programming

Let’s compute T:

		M	O	N	K	E	Y
M							
O							
N							
E							
Y							

Needleman-Wunsch algorithm

Initialization of first raw and first column: trivial.

Rationale: δe(X, ε) = |X| = δe(ε, X)

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1						
O	2						
N	3						
E	4						
Y	5						

T[0][3] = Edit distance between ε and MON = 3

Needleman-Wunsch algorithm

Fill the rest of the table :

T[i][j] = min(T[i-1][j] + 1,   1 + 1 = 2

T[i][j-1] + 1,   1 + 1 = 2

T[i-1][j-1]) + subst(i,j))   0 + 0 = 0

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	??					
O	2						
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

Fill the rest of the table :

T[i][j] = min(T[i-1][j] + 1, 1 + 1 = 2
T[i][j-1] + 1, 1 + 1 = 2
T[i-1][j-1]) + subst(i,j)) 0 + 0 = 0

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0					
O	2						
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

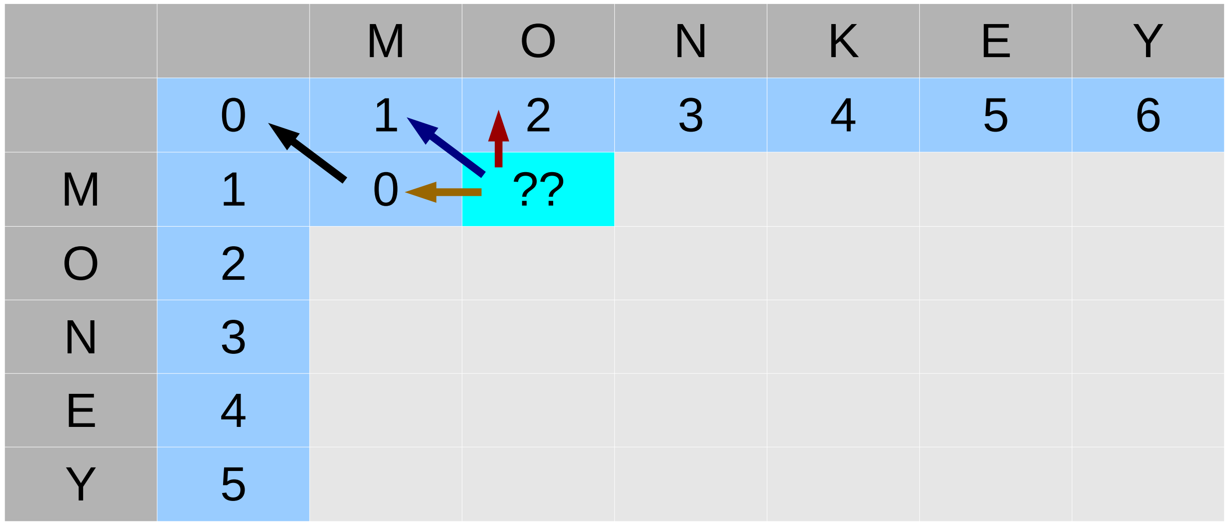
After filling a cell, keep track of the best path

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0					
O	2						
N	3						
E	4						
Y	5						

Diagonal path: substitution M → M

Needleman-Wunsch algorithm

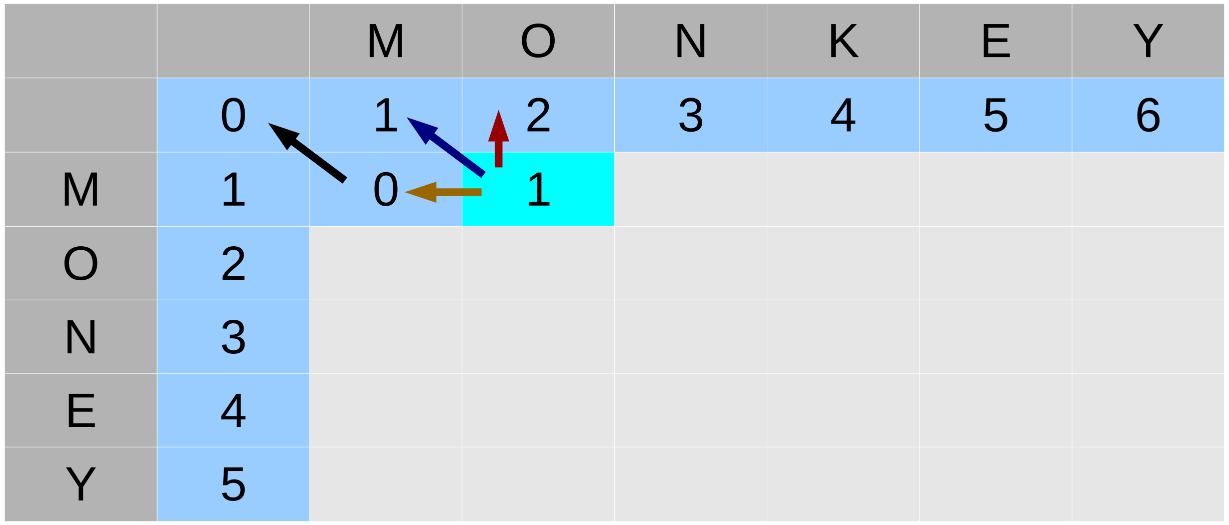
$$\begin{array}{lll} T[i][j] = \min(& T[i-1][j] + 1, & 0 + 1 = 1 \\ & T[i][j-1] + 1, & 2 + 1 = 3 \\ & T[i-1][j-1]) + \text{subst}(i,j)) & 1 + 1 = 2 \end{array}$$



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	??				
O	2						
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

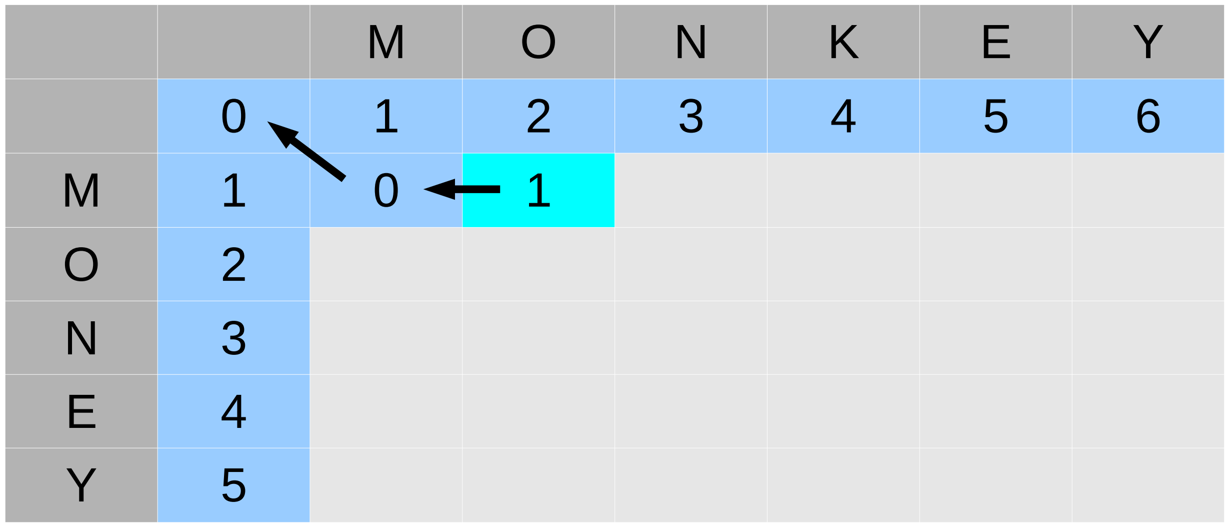
T[i][j] = min(T[i-1][j] + 1, 0 + 1 = 1
T[i][j-1] + 1, 2 + 1 = 3
T[i-1][j-1]) + subst(i,j)) 1 + 1 = 2



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1				
O	2						
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

After filling a cell, keep track of the best path



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1				
O	2						
N	3						
E	4						
Y	5						

Horizontal path: insertion of ‘O’

Needleman-Wunsch algorithm

Iterate...

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2			
O	2						
N	3						
E	4						
Y	5						

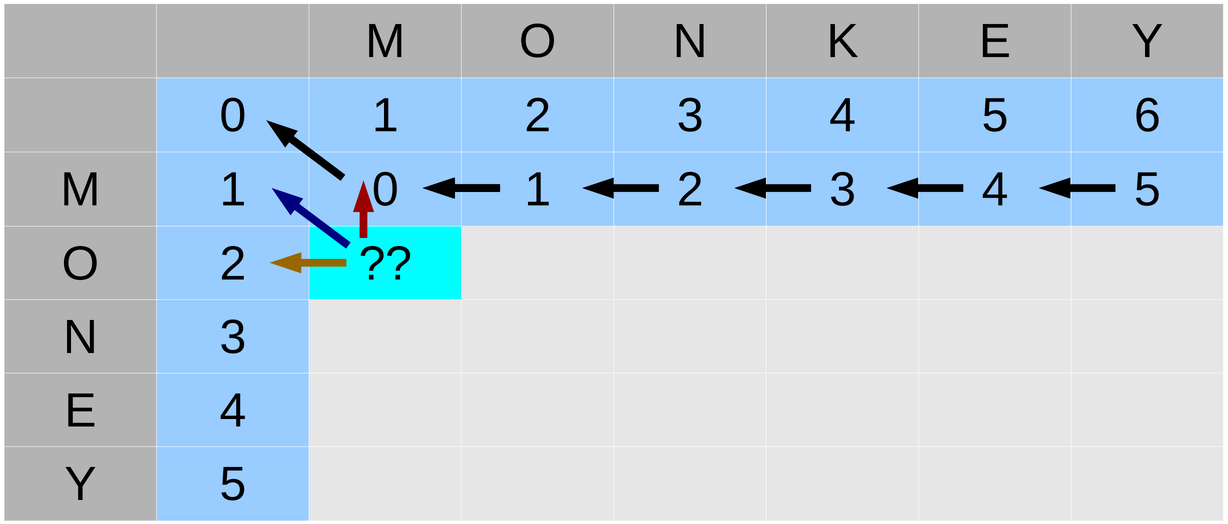
Needleman-Wunsch algorithm

Iterate...

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2						
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

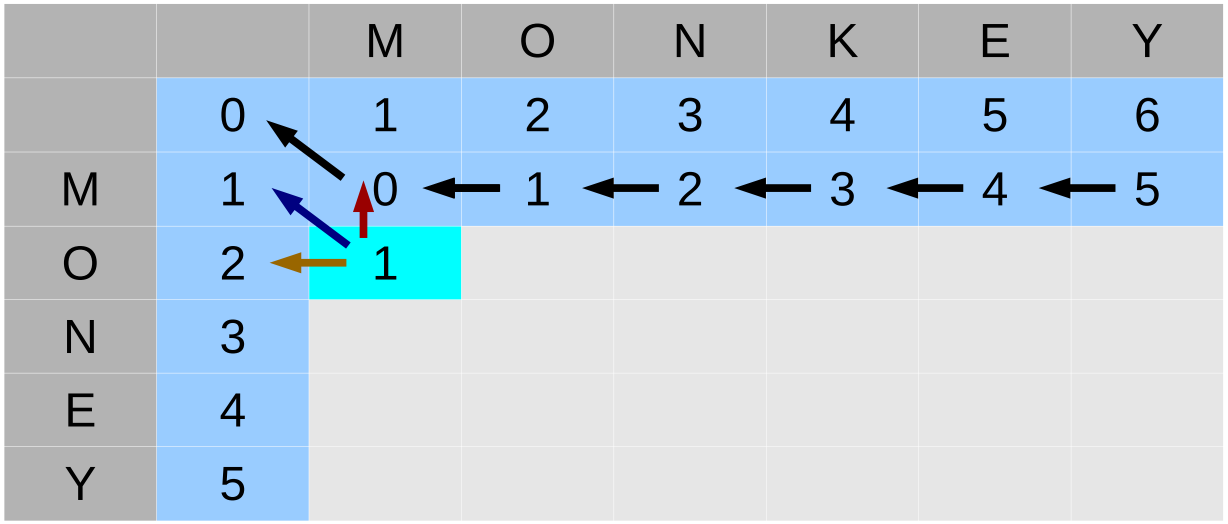
T[i][j] = min(T[i-1][j] + 1,   2 + 1 = 3
               T[i][j-1] + 1,   0 + 1 = 1
               T[i-1][j-1]) + subst(i,j))   1 + 1 = 2



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	??					
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

$$\begin{aligned} T[i][j] = \min(\ & T[i-1][j] + 1, && 2 + 1 = 3 \\ & T[i][j-1] + 1, && 0 + 1 = 1 \\ & T[i-1][j-1]) + \mathrm{subst}(i,j)) && 1 + 1 = 2 \end{aligned}$$



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1					
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

After filling a cell, keep track of the best path

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2	↑ 1					
N	3						
E	4						
Y	5						

Vertical path: deletion of ‘O’

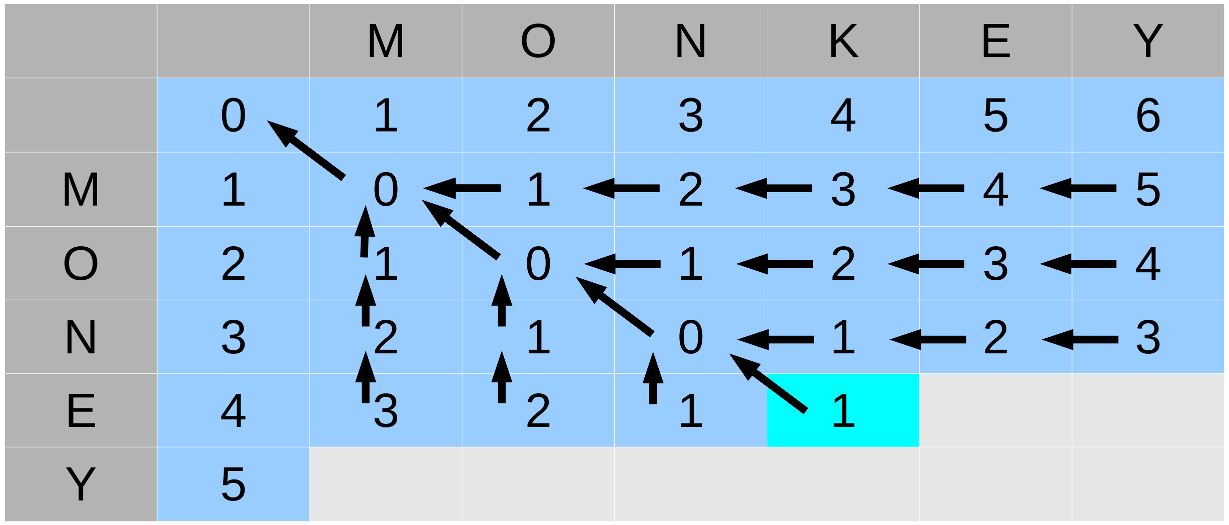
Needleman-Wunsch algorithm

Iterate...

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2	↑ 1	↖ 0				
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

Iterate...



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2	↑ 1	↖ 0	← 1	← 2	← 3	← 4
N	3	↑ 2	↑ 1	↖ 0	← 1	← 2	← 3
E	4	↑ 3	↑ 2	↑ 1	↖ 1		
Y	5						

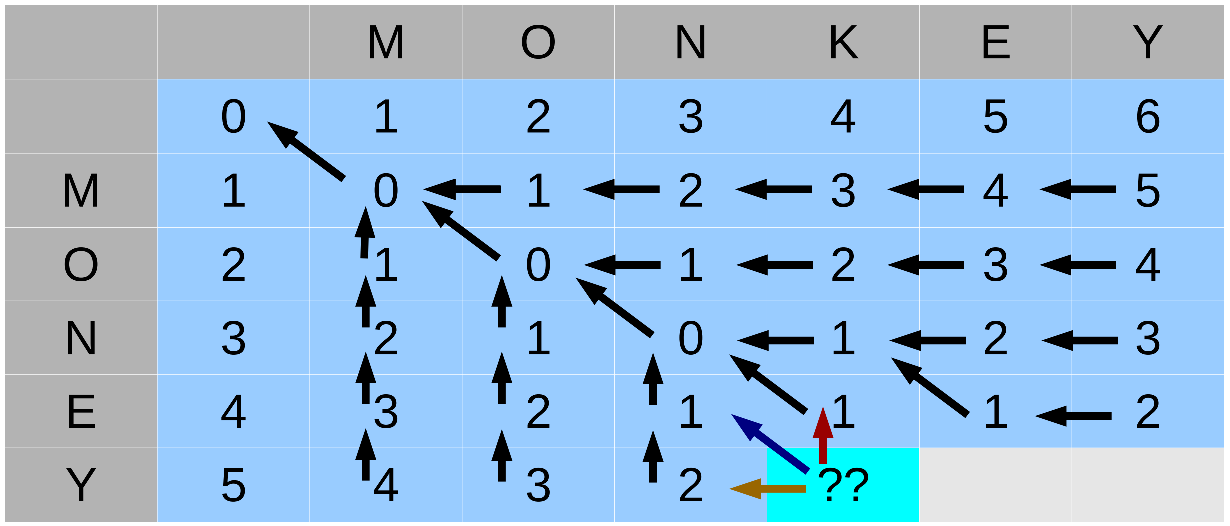
Needleman-Wunsch algorithm

Iterate...

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2	↑ 1	↖ 0	← 1	← 2	← 3	← 4
N	3	↑ 2	↑ 1	↖ 0	← 1	← 2	← 3
E	4	↑ 3	↑ 2	↑ 1	↖ 1	↖ 1	← 2
Y	5	↑ 4	↑ 3	↑ 2			

Needleman-Wunsch algorithm

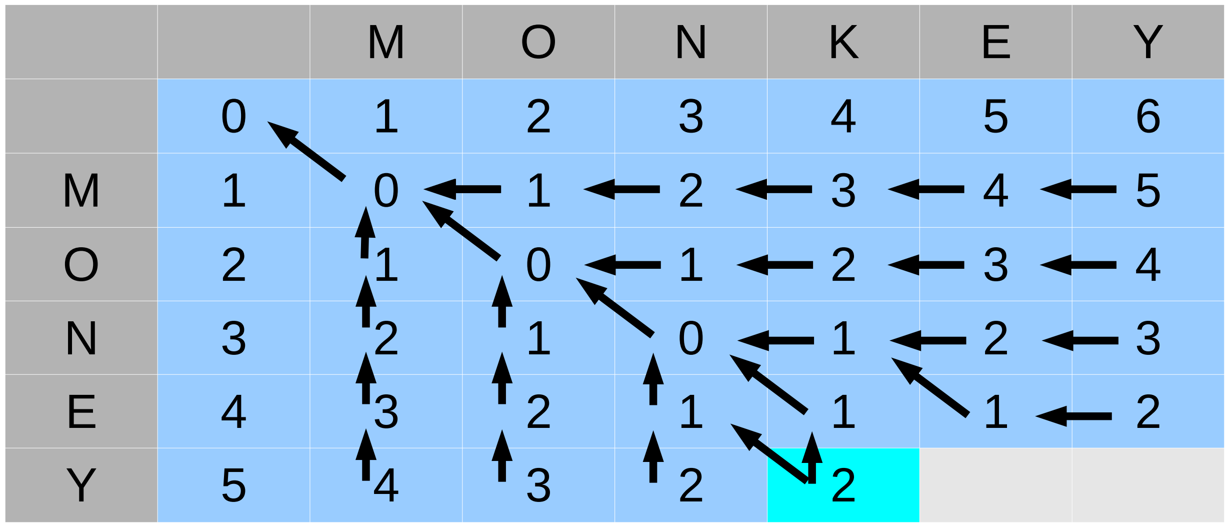
T[i][j] = min(T[i-1][j] + 1, 2 + 1 = 3
T[i][j-1] + 1, 1 + 1 = 2
T[i-1][j-1]) + subst(i,j)) 1 + 1 = 2



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	??		

Needleman-Wunsch algorithm

Sometimes, there are several best paths



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2		

Ambiguity!

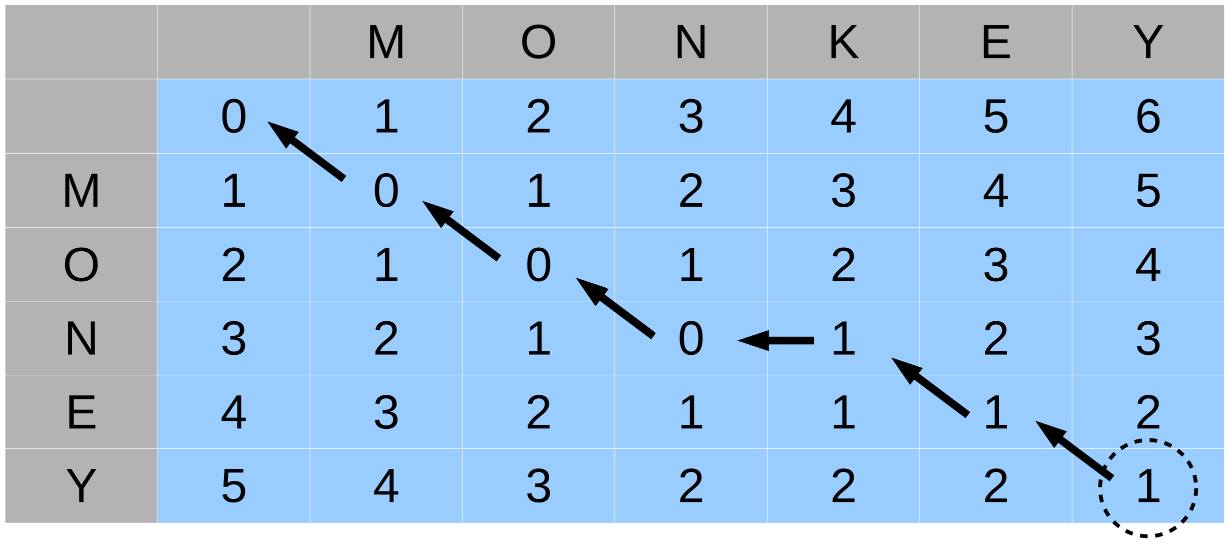
Needleman-Wunsch algorithm

Edit distance between monkey and money = 1

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖ 0	← 1	← 2	← 3	← 4	← 5
O	2	↑ 1	↖ 0	← 1	← 2	← 3	← 4
N	3	↑ 2	↑ 1	↖ 0	← 1	← 2	← 3
E	4	↑ 3	↑ 2	↑ 1	↖ 1	↖ 1	← 2
Y	5	↑ 4	↑ 3	↑ 2	↖ ↑ 2	↖ ↑ 2	↖ 1

Backtrace the best alignment

Follow the lines!

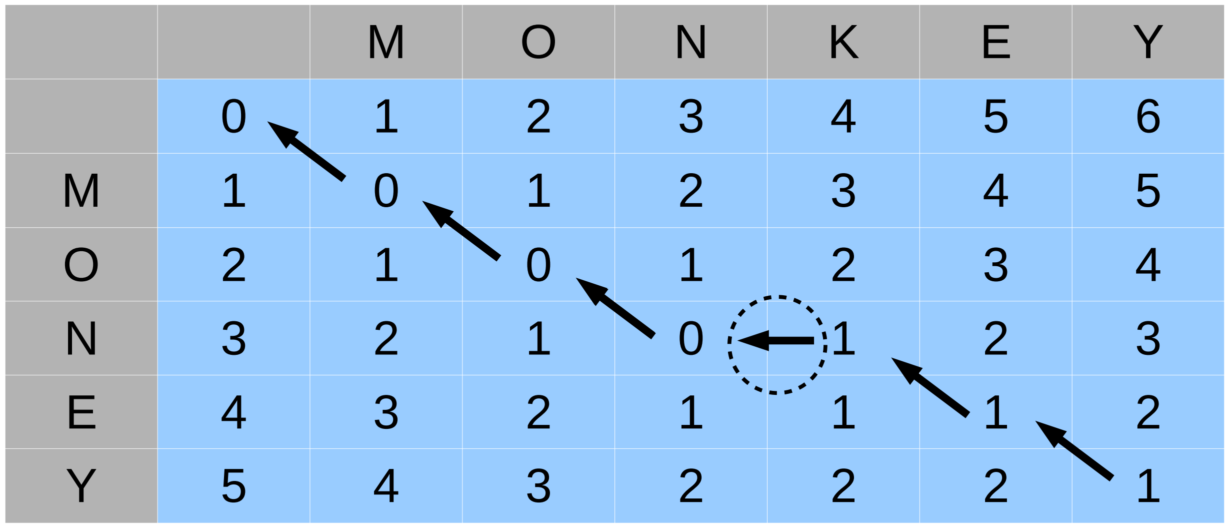


		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

Backtrace the best alignment

- Vertical line: deletion
- Horizontal line: insertion
- Diagonal line: substitution

```
M O N - E Y
M O N K E Y
```



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

Recap: Needleman-Wunsch algorithm

- Initialize first row and first column

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1						
O	2						
N	3						
E	4						
Y	5						

Recap: Needleman-Wunsch algorithm

- Initialize first row and first column
- Fill each element with the minimum of the three previous values. Store the best path.

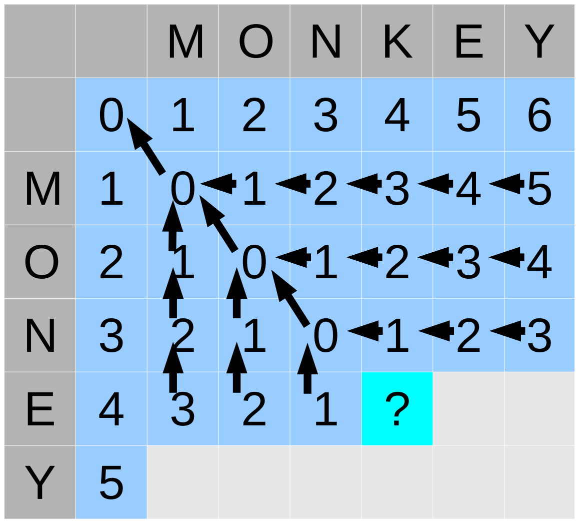
		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1						
O	2						
N	3						
E	4						
Y	5						

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖0	←1	←2	←3	←4	←5
O	2	↑1	↖0	←1	←2	←3	←4
N	3	↑2	↑1	↖0	←1	←2	←3
E	4	↑3	↑2	↑1	?		
Y	5						

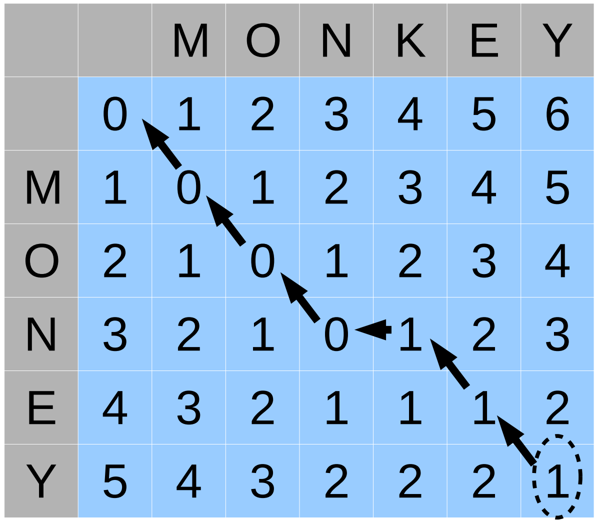
Recap: Needleman-Wunsch algorithm

- Initialize first row and first column
- Fill each element with the minimum of the three previous values. Store the best path.
- Backtrace to get the chain of operations

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1						
O	2						
N	3						
E	4						
Y	5						



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	?		
Y	5						



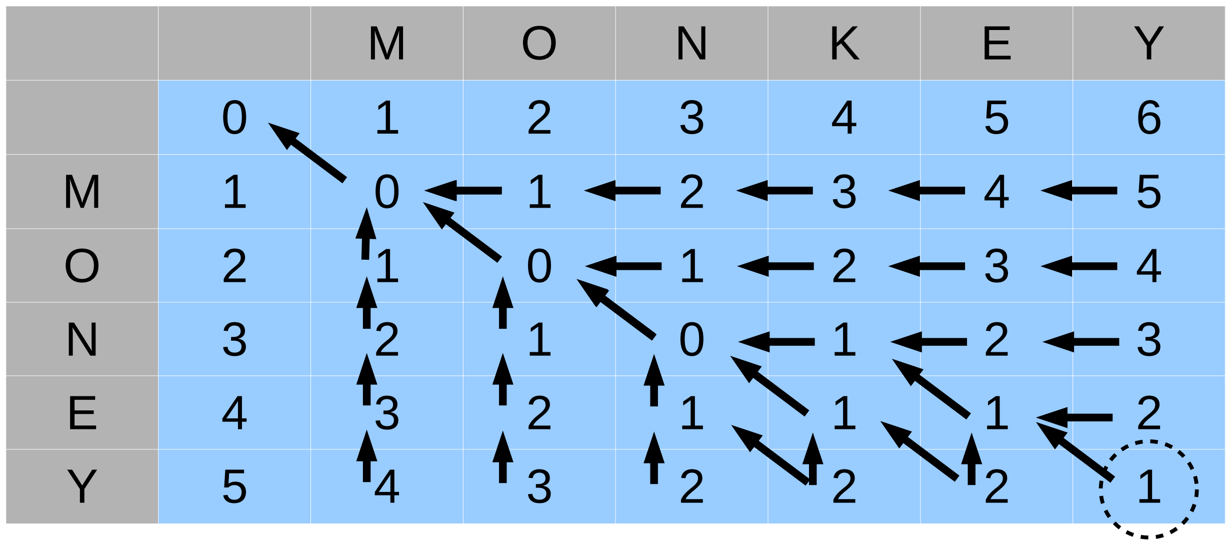
		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

Let n=|X| and m=|Y|. What is the complexity of the Needleman-Wunsch algorithm?

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	↖0	←1	←2	←3	←4	←5
O	2	↑1	↖0	←1	←2	←3	←4
N	3	↑2	↑1	↖0	←1	←2	←3
E	4	↑3	↑2	↑1	↖1	↖1	←2
Y	5	↑4	↑3	↑2	↖↑2	↖↑2	↖1

Complexity

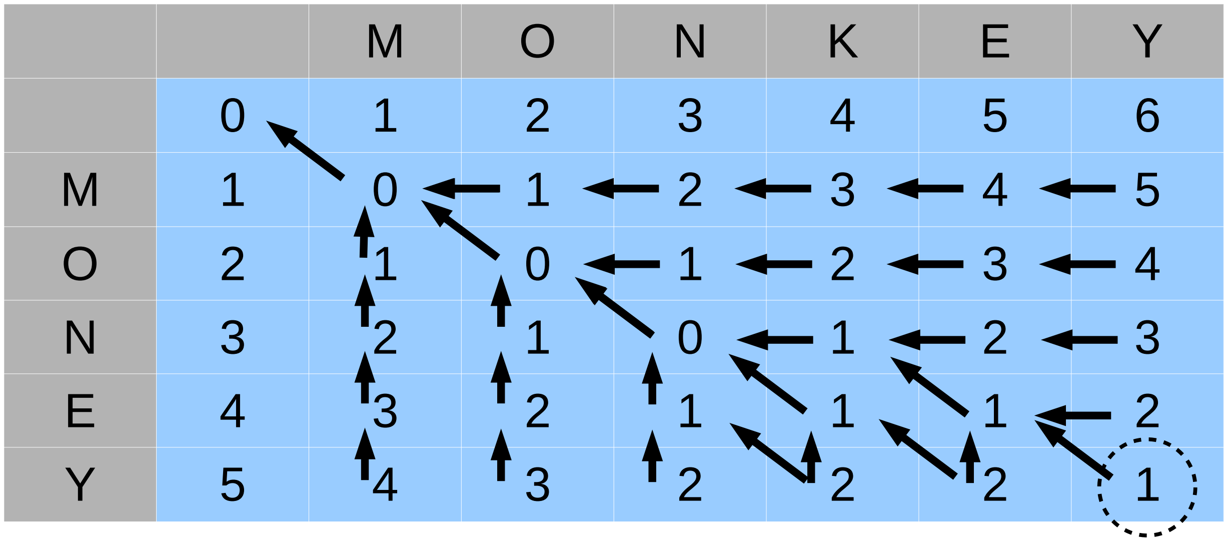
- Most expensive step: filling the table
- O(n * m) where n = |X| and m = |Y|



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

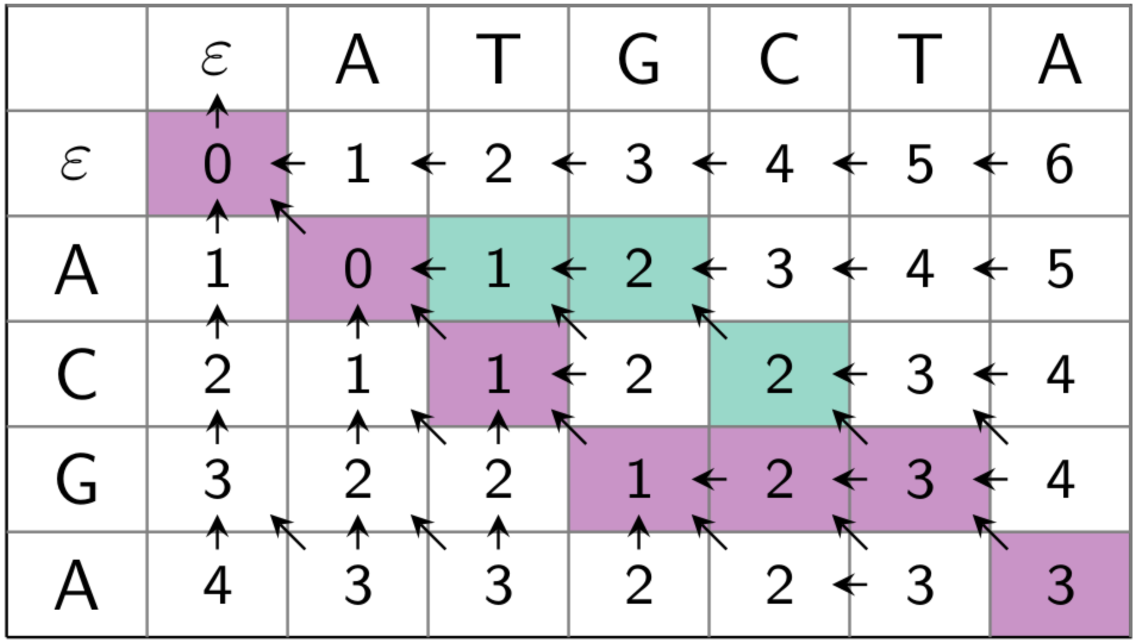
Complexity

- Most expensive step: filling the table
- O(n * m) where n = |X| and m = |Y|
- Much quicker than naive recursion or bruteforce!



		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

Ambiguities



	ε	A	T	G	C	T	A
ε	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
C	2	1	1	2	2	3	4
G	3	2	2	1	2	3	4
A	4	3	3	2	2	3	3

$$\begin{pmatrix}\texttt{A--CGA}\\ \texttt{ATGCTA}\end{pmatrix} \begin{pmatrix}\texttt{ACG--A}\\ \texttt{ATGCTA}\end{pmatrix}$$

Should we really assign the same cost to substitutions, deletions and insertions?

Should we really assign the same cost to substitutions, deletions and insertions?

→ No, there is not reason to think that these events are equally likely.

Adding weights

- We can penalize some less likely operations with weights.

Example: assume that substitutions happen 5 times more often than additions and deletions:

```
F(i,j) =   min (
                F(i, j – 1) + 5,
                F(i – 1, j) + 5,
                F(i – 1, j – 1) + 1 * (X[i] != Y[j])
                )
```

Remark

If the insertion and deletion costs are different, the edit distance is not a distance anymore:

The symetry d(X,Y) = d(Y,X)

is not respected anymore

In coding genes, what could make an insertion less likely than a substitution?

(coding genes code for a protein)

Insertion VS substitution in coding genes

Insertions are less likely to produce a fit organism

- Insertion of a non-multiple of three number of nucletids → shift the whole sequence!

  TTT CCC AAA GGG → TAT TCC CAA AGG

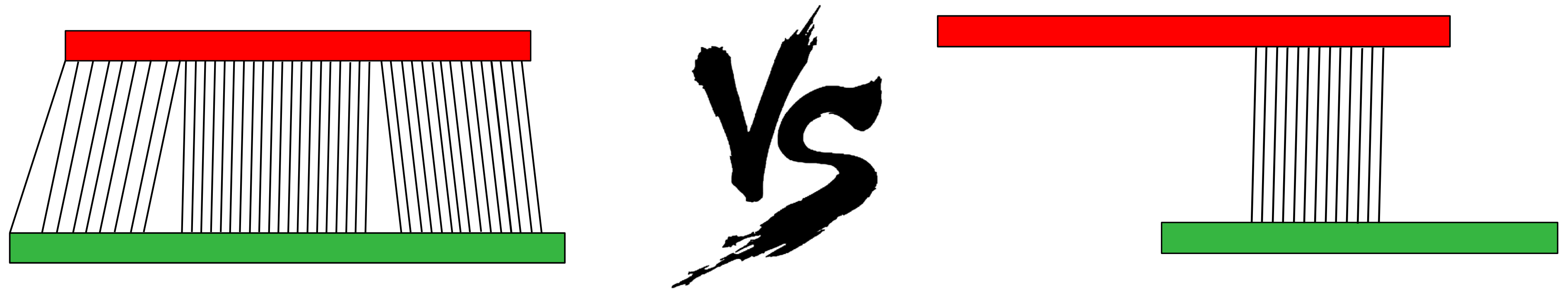
- Some nucleotid substitutions are silent. For instance, TTT and TTC both code for the same amino acid Lys

What is the ratio between subsitutions and insertions/deletions?

It depends!

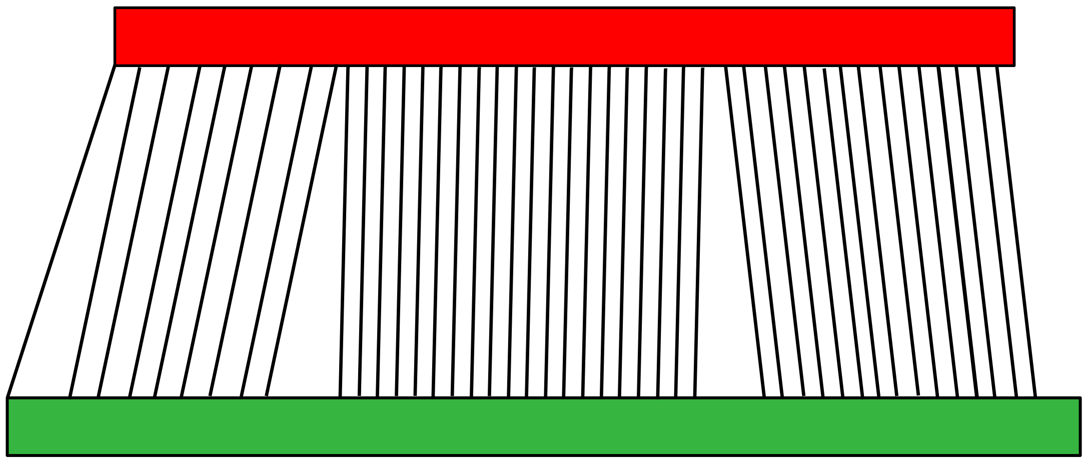
- Type of organism
- Coding/non coding genes
- Etc.

What kind of alignment problem did we solve with the Needleman-Wunsch algorithm?



Answer: global alignment

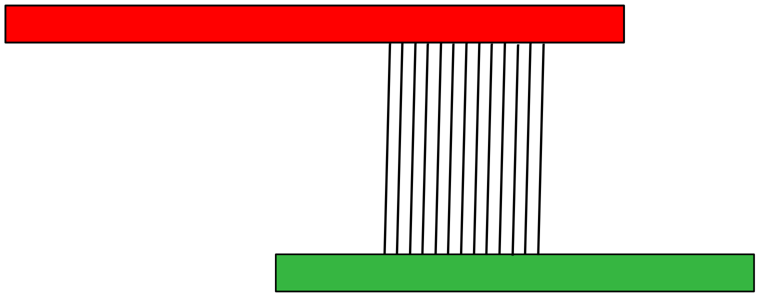
- Needleman-Wunsch gives us the best sequence of operations to get Y from X.
- This corresponds to the best global alignment.



Local alignment

Input: two strings X and Y

Output: two aligned substrings



Local alignment problem

Example: TTTACCACAACT and GACCATCAACGGGG

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

Formulate local alignment problem with distances

“*Find the substrings with minimum distance*”

Issue: distances increase with the lengths of the strings. Short strings will be selected

δe(ACCACAAC, ACCATCAAC) = 1

δe(ACCA, ACCA) = 0

Similarity functions

- Assign positive and negative weights, to favor similarities
  - Insertion, deletion, non-identity substitution have a negative weight
  - Identity substitution has a positive weight

Similarity functions are NOT distances. They do not verify:

Positivity: $s(x,y) >= 0$

Formulate local alignment problem with similarities

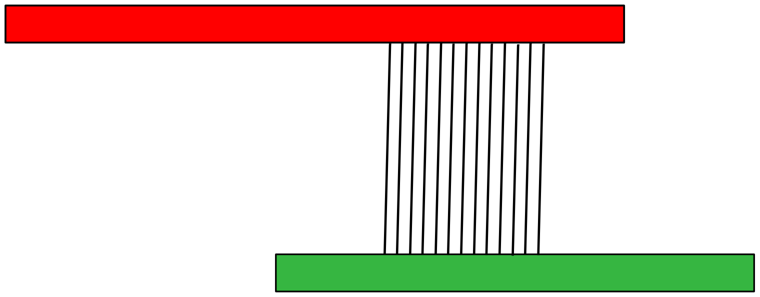
“*Find the substrings with maximum similarity*”

S(ACCACAAC, ACCATCAAC) = 8 – 1 = 7

S(ACCA, ACCA) = 4

Local alignment problem

Find the two substrings of X and Y with the maximal <u>similarity</u>.



Smith-Waterman algorithm to solve local alignment

Same as Needleman-Wunsch algorithm, but:

- Use a similarity function for the Levenstein formula
- Negative values are set to 0.
- Initialize first row and column with 0.
- Find the largest value in the table and traceback until a 0 is reached.

Align EAWACQGKL and ERDAWCQPGKWY

S	-	E	R	D	A	W	C	Q	P	G	K	W	Y
-	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	1	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	1	0	0	0	0	0	0	0	0
W	0	0	0	0	0	2	1	0	0	0	0	1	0
A	0	0	0	0	1	1	0	0	0	0	0	0	0
C	0	0	0	0	0	0	2	1	0	0	0	0	0
Q	0	0	0	0	0	0	1	3	2	1	0	0	0
G	0	0	0	0	0	0	0	2	1	3	2	1	0
K	0	0	0	0	0	0	0	1	0	2	4	3	2
L	0	0	0	0	0	0	0	0	0	1	3	2	1

$$\begin{pmatrix}\texttt{AWACQ-GK}\\ \texttt{AW-CQPGK}\end{pmatrix}$$

Gap penalties

One large deletion is more likely than many small deletions:

```
GAAAAAT        GAAAAAT
GAA---T        G-A-A-T
```

→different scores for gap-start and gap-extension

Substitution matrices

- Are all substitution equally likely?

- Is a nucleotid A more likely to be replaced with a G or a T?

Substitution matrices

- Are all substitution equally likely?

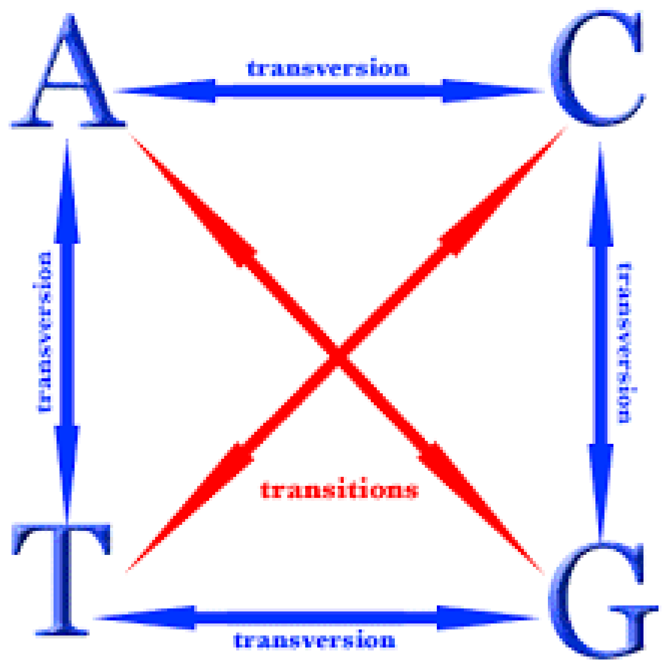
  No!

- Is a nucleotid A more likely to be replaced with a G or a T?

  Yes!

Substitution matrices

For instance, transitions (A ↔ G and C ↔ T) happen more often than transversions



Substitution matrices

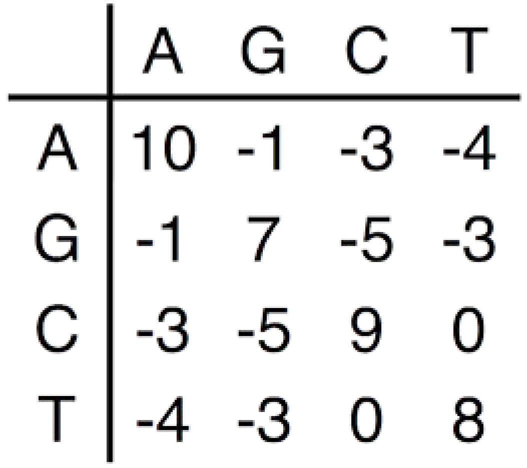
- The substitutions costs in our alignment algorithms should take these rates into account:

  subst(A,G) < subst(A,C)

- $$F(i,j) = \min \begin{pmatrix} F(i, j-1) + \text{ins}(b), \\ F(i-1, j) + \text{del}(a), \\ F(i-1, j-1) + \text{subst}(a,b) \end{pmatrix}$$

Substitution matrices

A substitution matrix describes the rate at which one character in a sequence changes to other character states over the time.

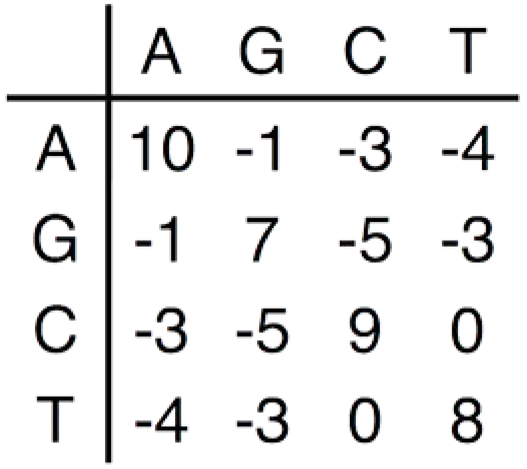


	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

$\longleftarrow$ Example of a DNA substitution matrix

Substitution matrices

Substitution matrices are crucial to build reliable alignments!



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

Key points of the lecture (exam-relevant!)

- Local VS global alignment
- Hamming/Edit distances, similarity functions
- Needleman-Wunsch algorithm (and variations):
  - Write the recursion formula
  - Fill a dynamic programming table
  - Backtrace to build the alignment
- Substitution matrices, gap penalties

Questions, feedback?

lukas.huebner@h-its.org