Introduction to Bioinformatics for Computer Scientists

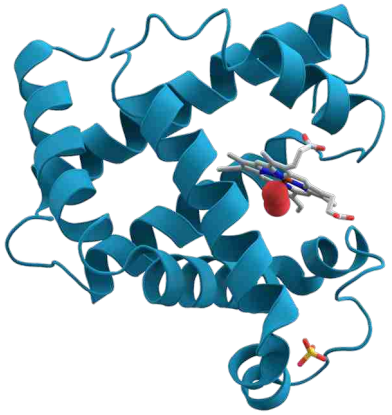
Lecture 3

Pair-wise Sequence Alignment

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

slides by
Benoit Morel, postdoc

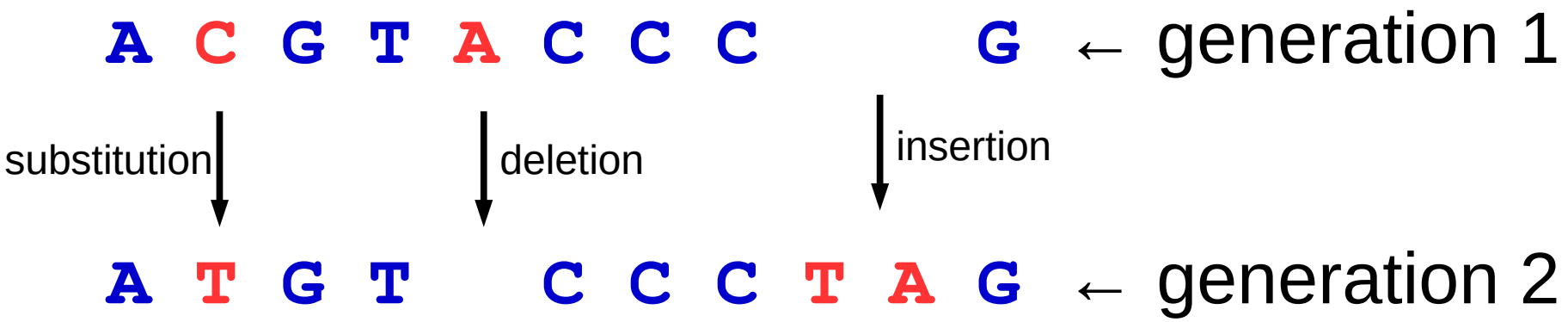
DNA and protein sequences are strings

- DNA:  AACCTGTTGTCAAATG
- Protein:  TTETTSFLIFETAVKNT

Sequences evolve

A C G T A C C C G ← generation 1

Sequences evolve

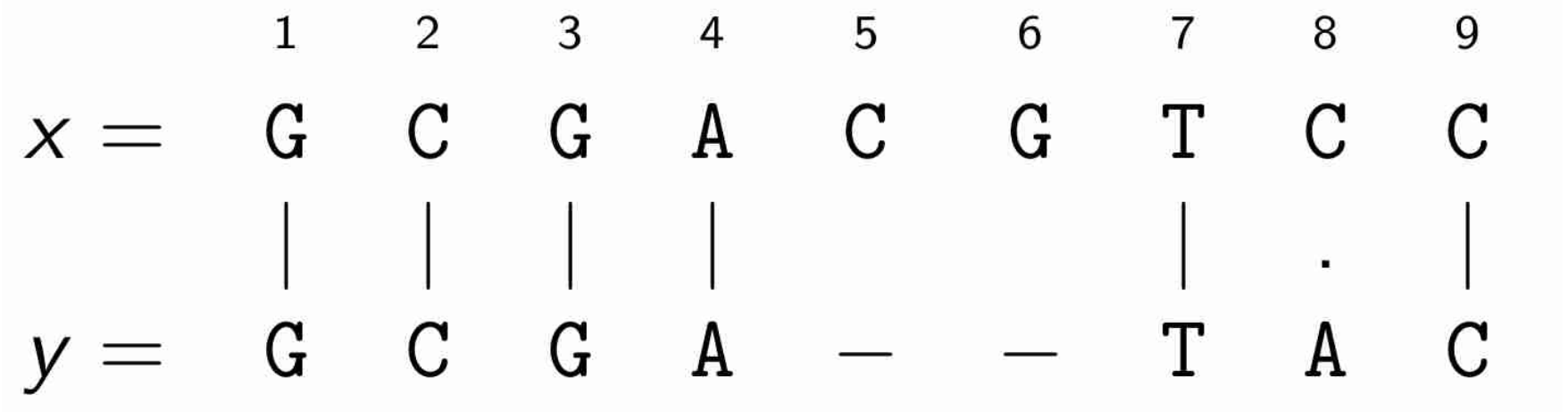


(and their lengths change)

Pair-wise sequence alignment

Compare two sequences to infer their similarity

Example: alignment between ‘GCGACGTCC’ and ‘GCGATAC’



Example 1: Measure DNA similarity

How similar are human and chimpanzee at the DNA level?

Human DNA

Chimpanzee DNA

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

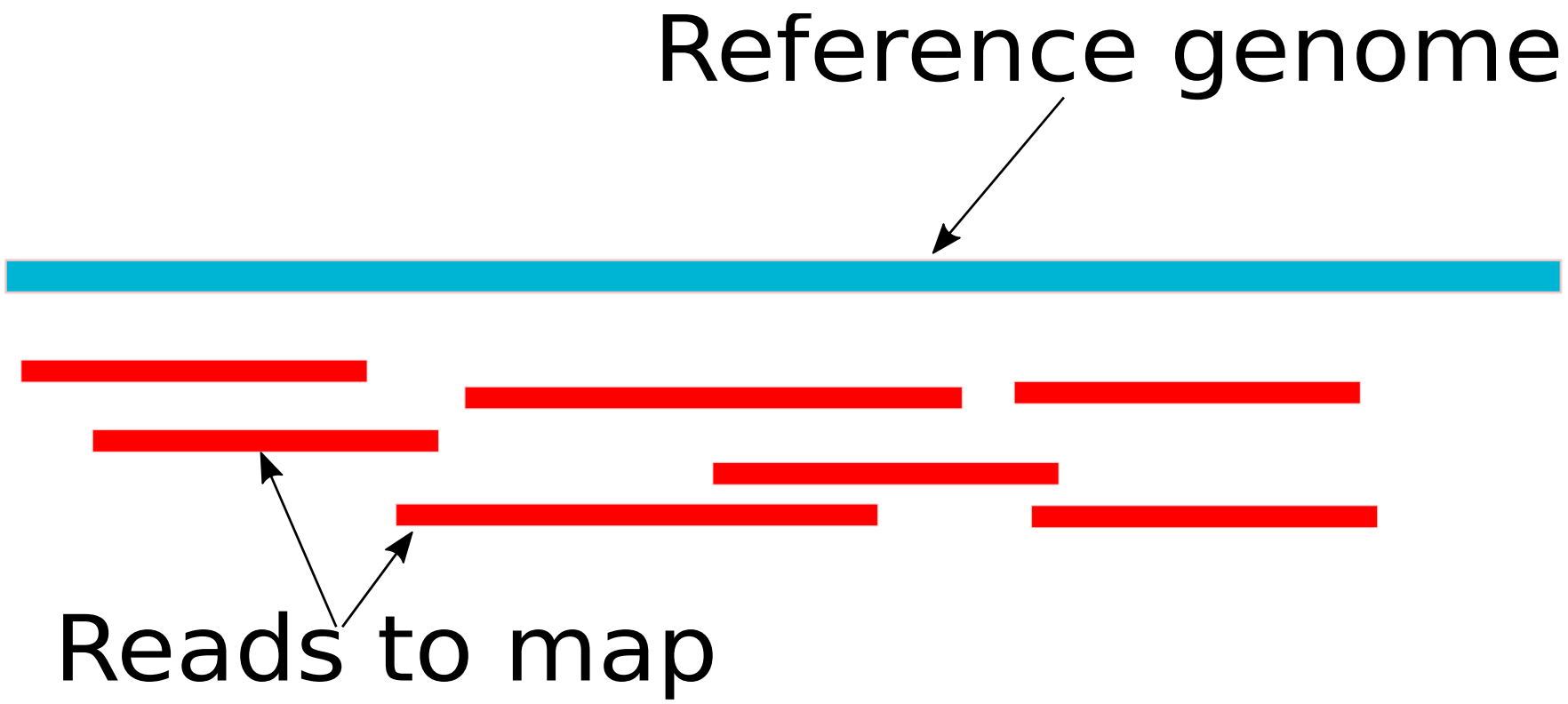
Individual to sequence

Sequencing machine

Unordered reads
(short DNA sequences)

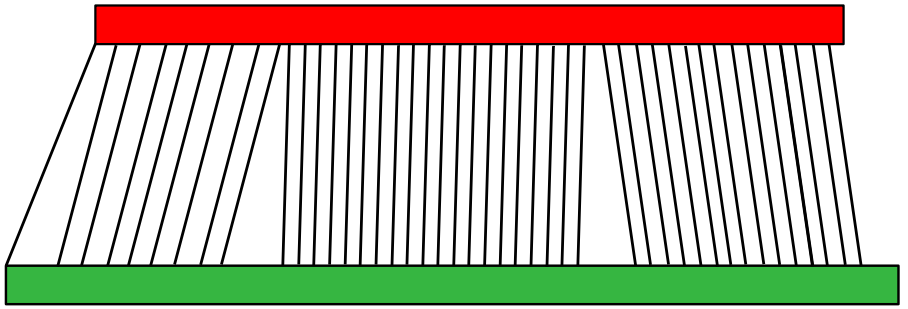
Assembled genome

Example 2: genome assembly

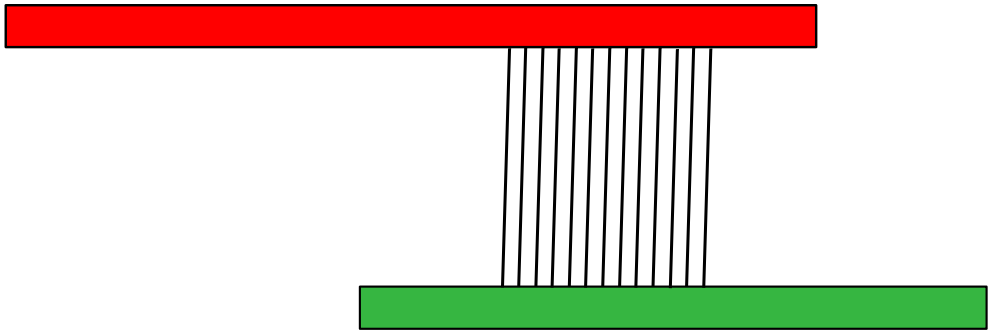


Global and local alignments

Global alignment: align the full strings



Local alignment: align similar substrings



Different approaches to alignment

Two approaches:

- **Online algorithms ← (today)**
- Offline algorithms

Four main approaches to online sequence alignment:

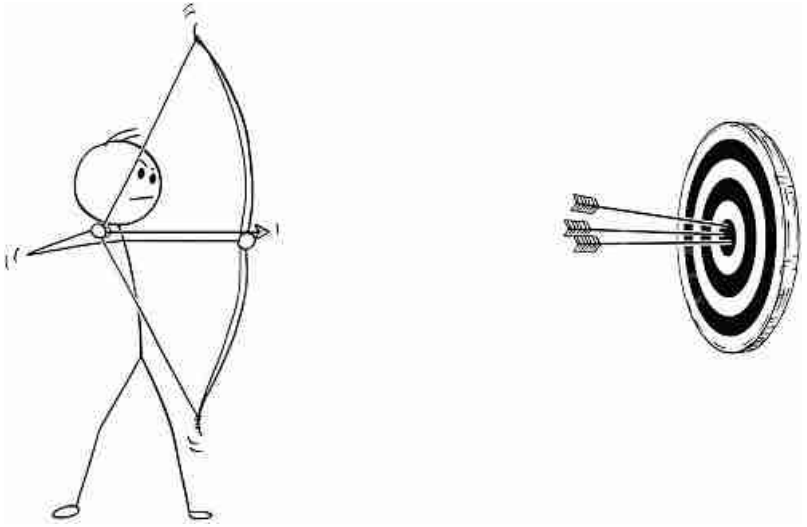
- **Dynamic programming ← (today)**
- Automata
- Word-level parallelism
- Filtering

Online VS offline

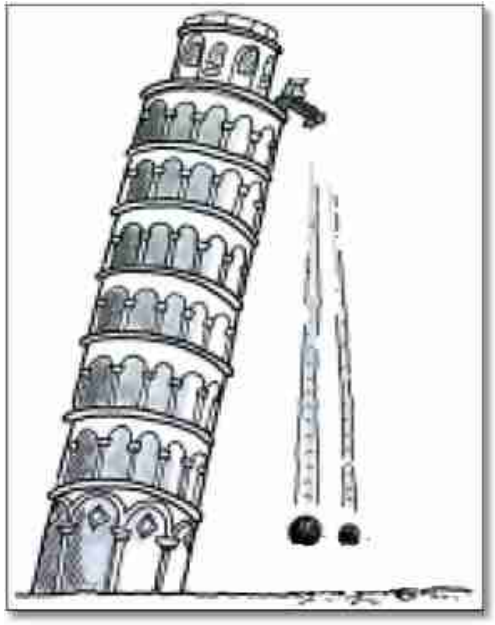
- **Online algorithm:** can process the input data piece by piece in a serial fashion

- **Offline algorithm:** needs the entire input data to start processing it

Online VS offline example



Online: we don’t need all arrows to start throwing them (assume they cannot collide)



Offline: we need both balls to start the experiment (check that they hit the ground at the same time)

Dynamic programming

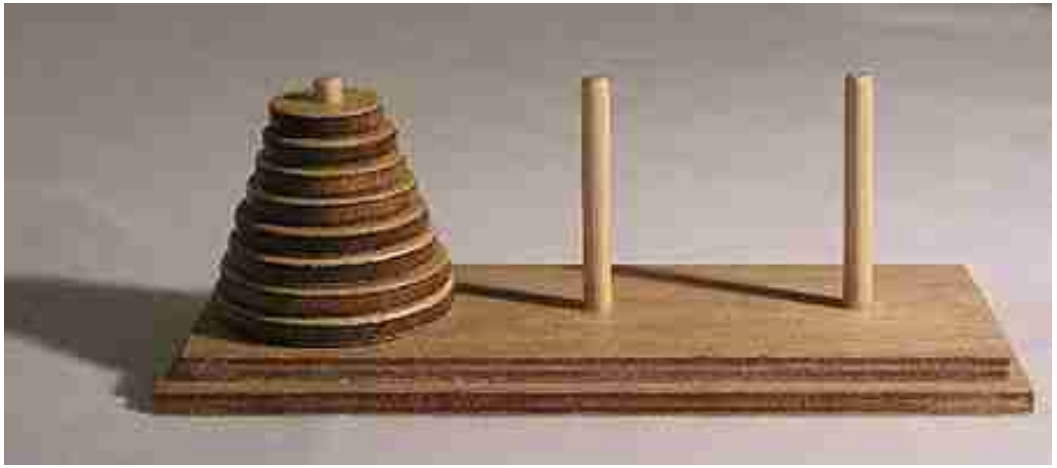
- Break down a complicated problem into simpler subproblems (typically with recursion)
- Cache the results of the recursive calls

Dynamic programming

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

Example:

Tower of Hanoi algorithm



Definitions

Alphabet

An **alphabet** Σ is a finite non empty set whose elements are called **letters**

Examples :

- DNA: Σ = {‘A’, ‘C’, ‘G’, ‘T’}
- Numerics: Σ = {‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’}
- German alphabet, Chinese alphabet…

String

- A **string** is a finite sequence of elements from an alphabet
- The empty string is denoted **ε**
- Notation: x[i] is the ith element of the string x

Examples:

- "ACCAGTGGTGTGGCA" (x[0] = 'A', x[1] = 'C' etc.)
- "31349756984"
- ε

String length

- The length of a string x is defined as the length of its sequence of letters.
- Notation: |x|

Examples:

- |“pizza”| = 5
- |ε| = 0

Identity between strings

x = y

if and only if

|x| = |y|

x[i] = y[x] for all i, 0 <= i < |x|:

Examples:

- “42” = “42”
- “Potato” != “Tomato”

String concatenation

- The concatenation of two strings x and y is the string of the letters of x followed by the letters of y.
- Notation: xy

Example:

x = “James_”

y = “Bond”

xy = “James_Bond”

Substrings

- A string x is a substring (or factor) of y if there exist u and v, such that y = uxv

Examples:

- ε is a substring of any string
- “logic” is a substring of “bio**logic**al” (u=“bio” v=“al”)

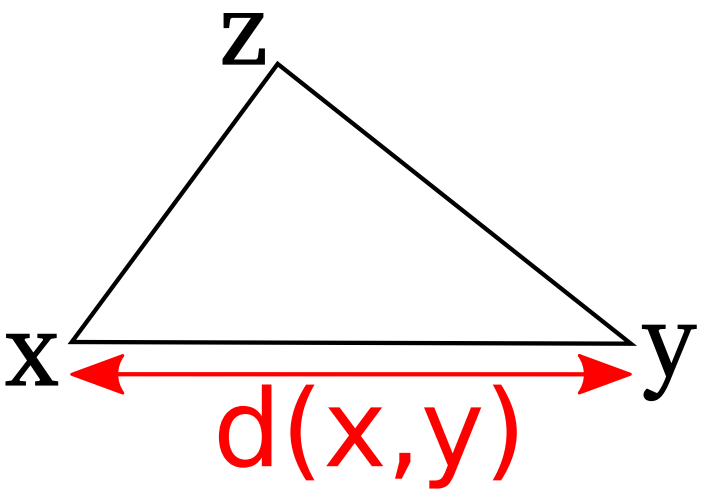
Occurrence of a string

X is an **occurrence** of y (X **occurs** in Y) if X is a substring of Y

"GA" occurs 3 times in "A**GA**TCT**GA**C**GA**"

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!

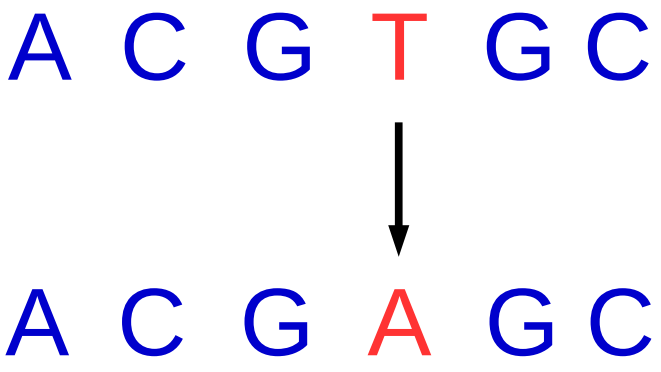
X = ACTATATATACG–

Y = –CTATATATACGT

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

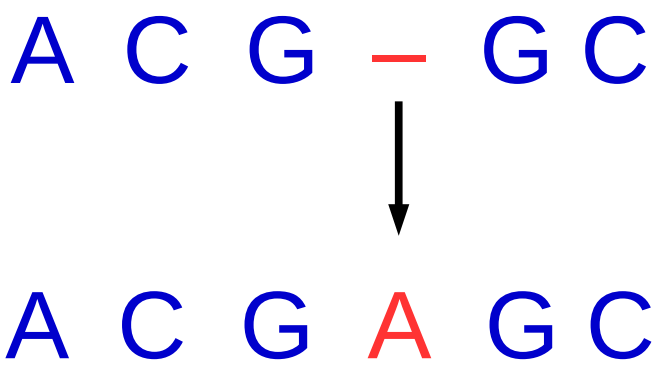
Edit operations: substitution, insertion and deletion

Substition of one letter x by another letter y



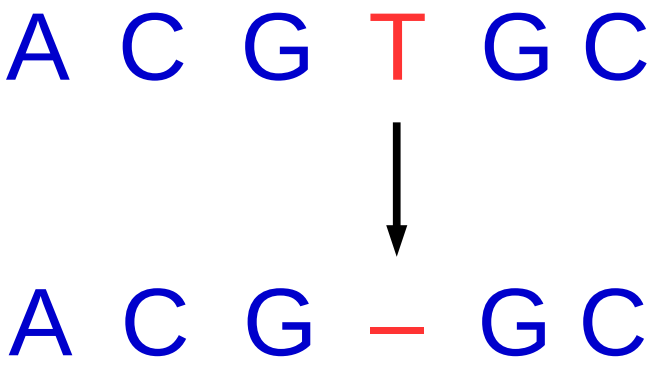
Edit operations: substitution, insertion and deletion

Insertion of one letter



Edit operations: substitution, insertion and deletion

Deletion of one letter



Edit distance

δe(x,y) = minimum 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.

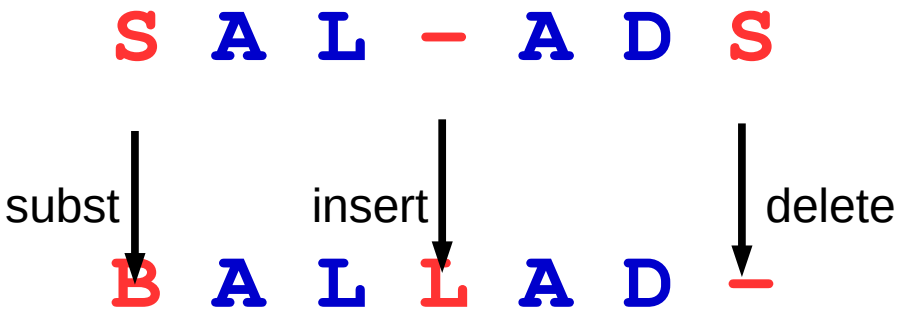
ACGA--	**A------CGA**	**A--CGA**
ATGCTA	**-ATGCTA---**	**ATGCTA**
d=4	**d=10**	**d=3**

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**?

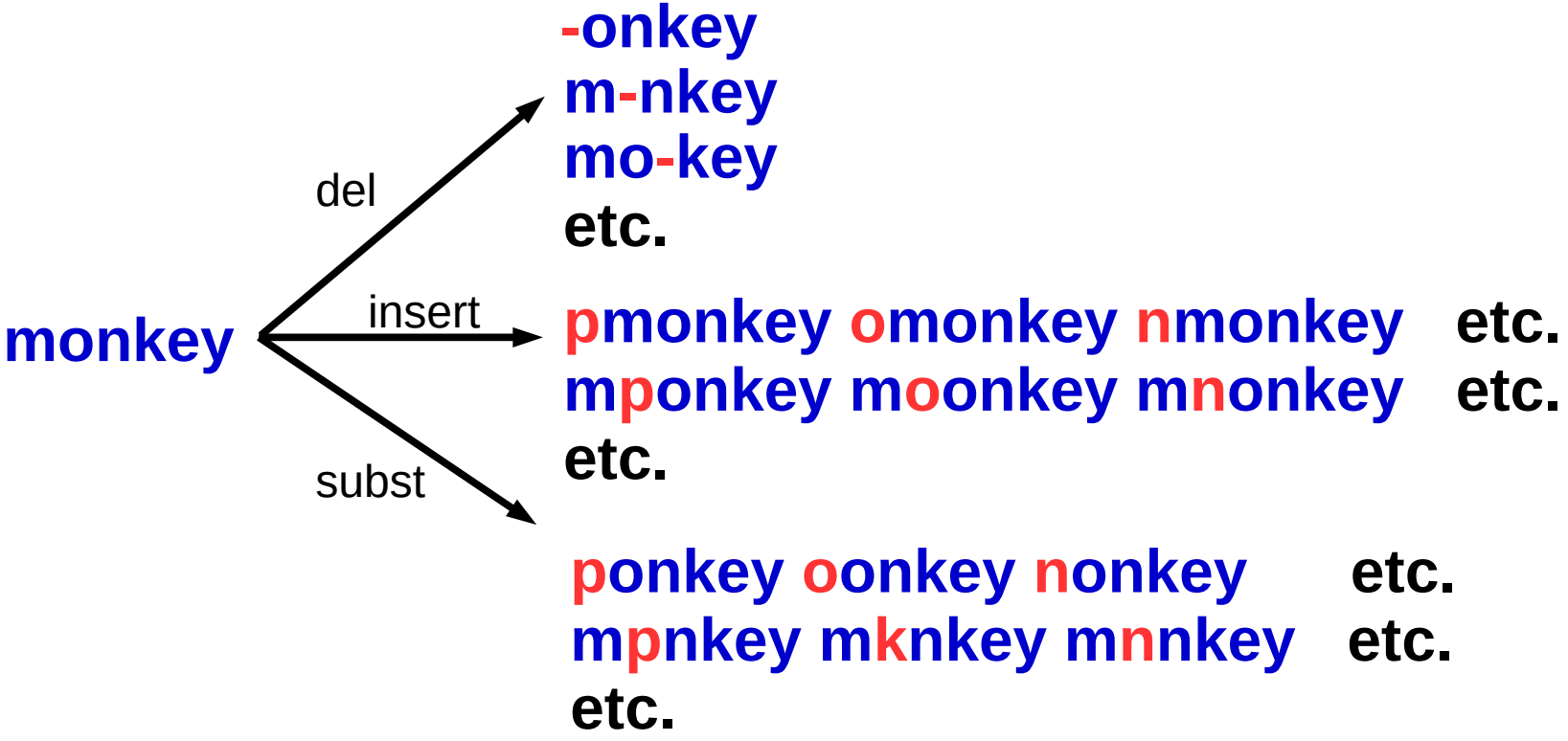
For instance, how to compute the distance between **poney** and **monkey**?

Naive bruteforce

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

Naive bruteforce

First iteration:



Naive bruteforce

<u>Second iteration:</u> 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

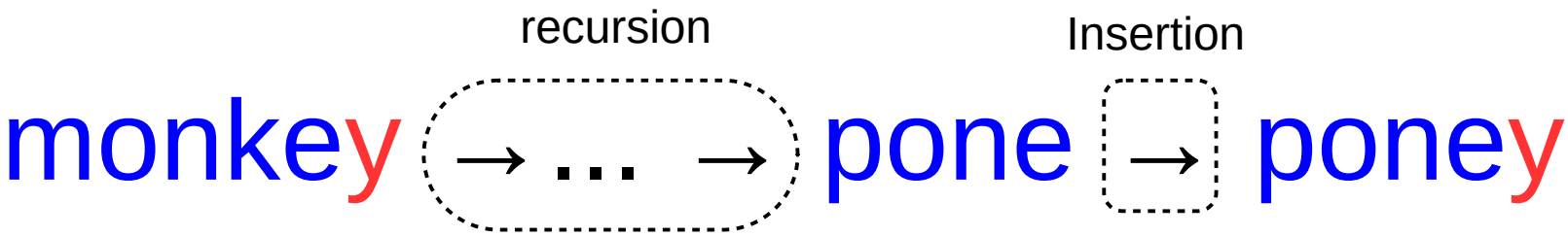
Let F(i,j) be the edit distance between the prefixes of X of size i and the prefix of Y of size j.

$$
\begin{aligned}
F(i,j) = \quad & \min (\\
& \quad F(i, j-1) + 1, \\
& \quad F(i-1, j) + 1, \\
& \quad F(i-1, j-1) + 1 * (X[i] \mathrel{!}= Y[j]) \\
&) \\
\end{aligned}
$$

Levenstein formula

$$
\begin{aligned}
&\delta e(monkey, poney) = \min (\\
&\quad \delta e(monkey, pone) + 1, \\
&\quad \delta e(monke, poney) + 1, \\
&\quad \delta e(monke, pone) + 1 * (y \mathrel{!}= y) \\
&)
\end{aligned}
$$

First term



δe(monkey,pone) + 1

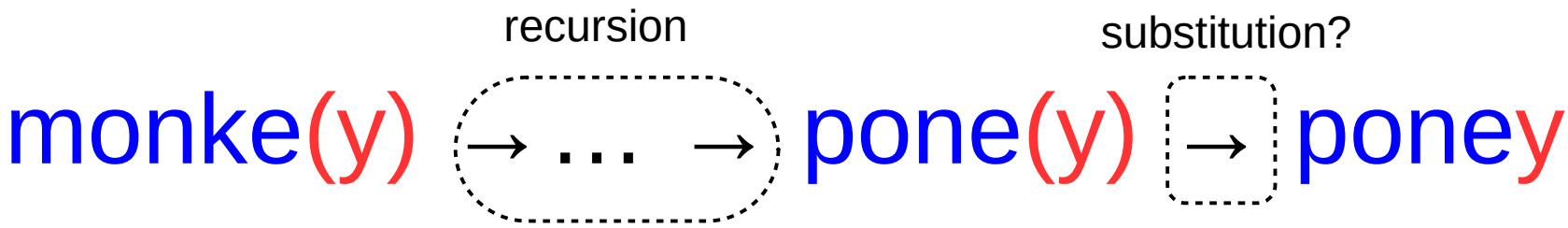
F(i, j – 1) + 1

Second term

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

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

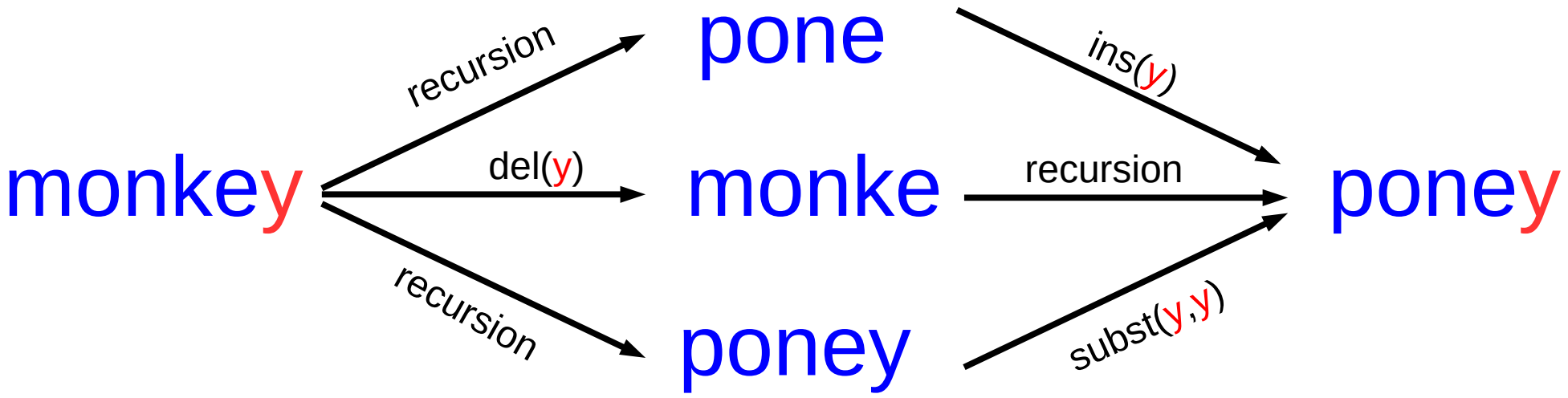
Third term



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:

$F(0, j) = j$

$F(i, 0) = i$

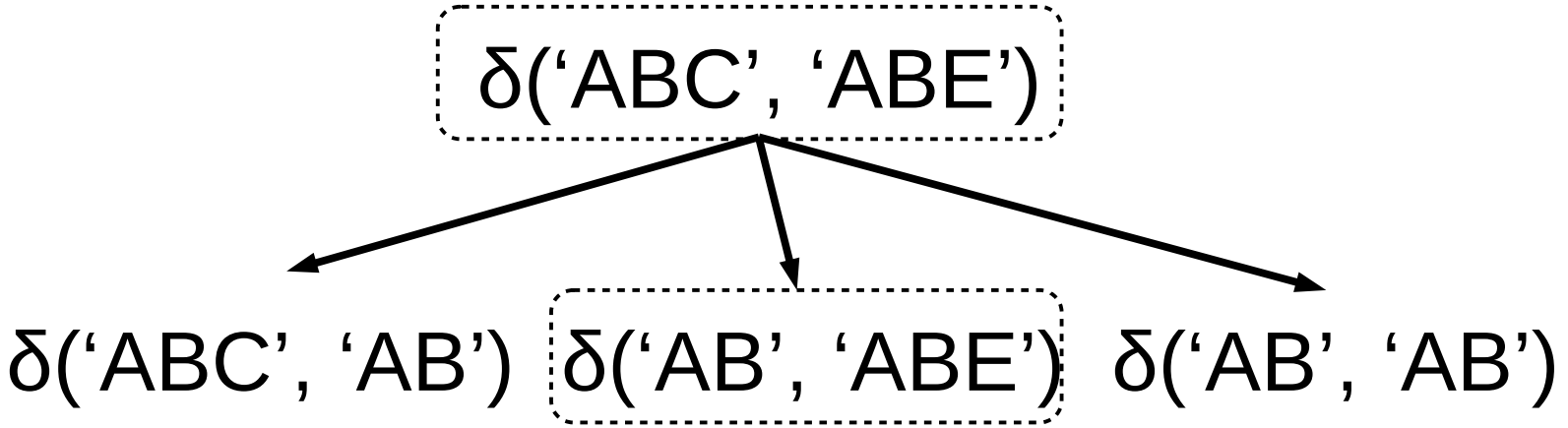
remember: $\delta e(\mathbf{X}, \varepsilon) = |\mathbf{X}| = \delta e(\varepsilon, \mathbf{X})$

Levenstein formula recursion convergence

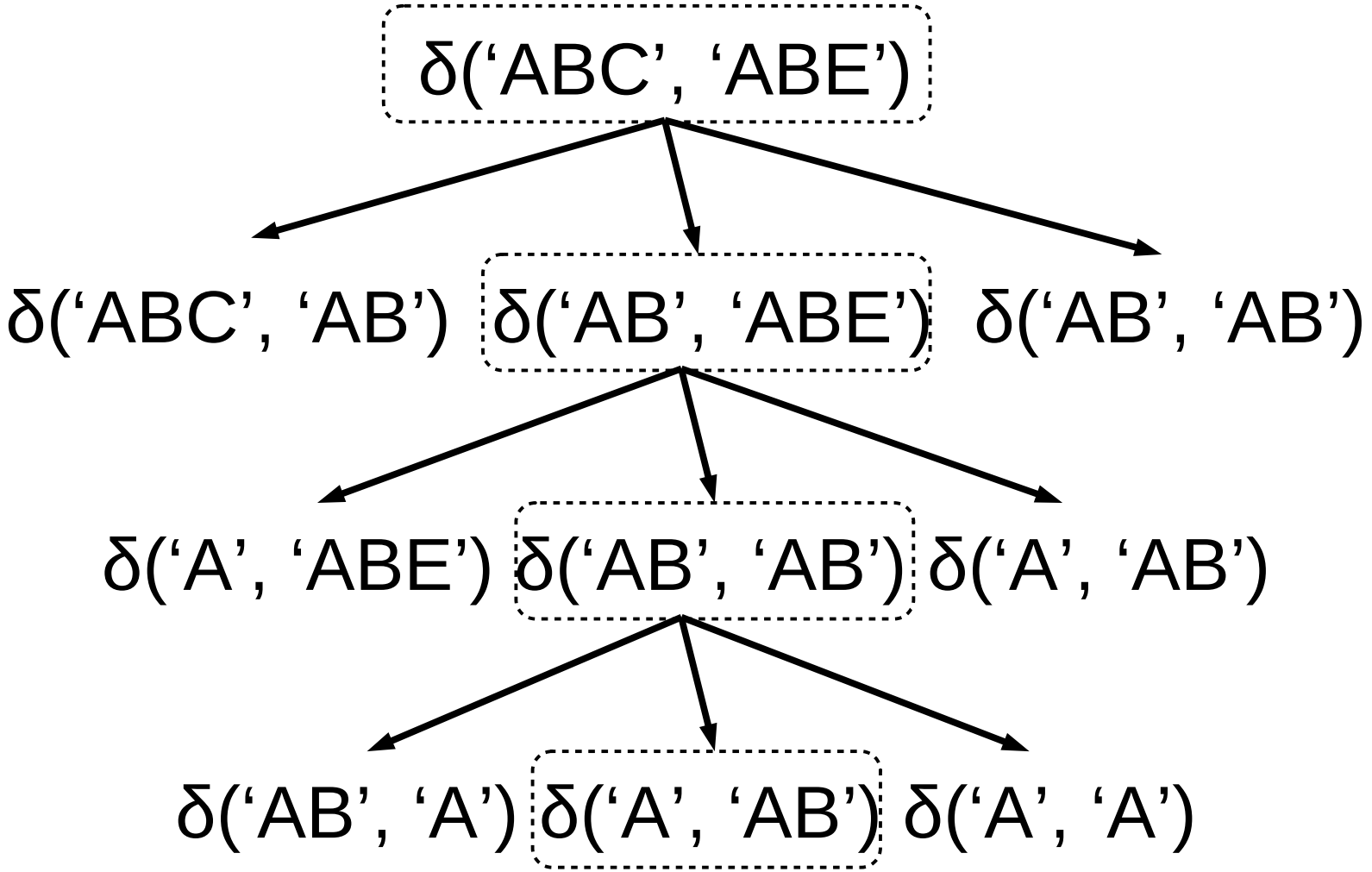
$$F(i,j) = \min \begin{cases} F(i, j-1) + 1, \\ F(i-1, j) + 1, \\ F(i-1, j-1) + 1 * (X[i] \neq Y[j]) \end{cases}$$

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

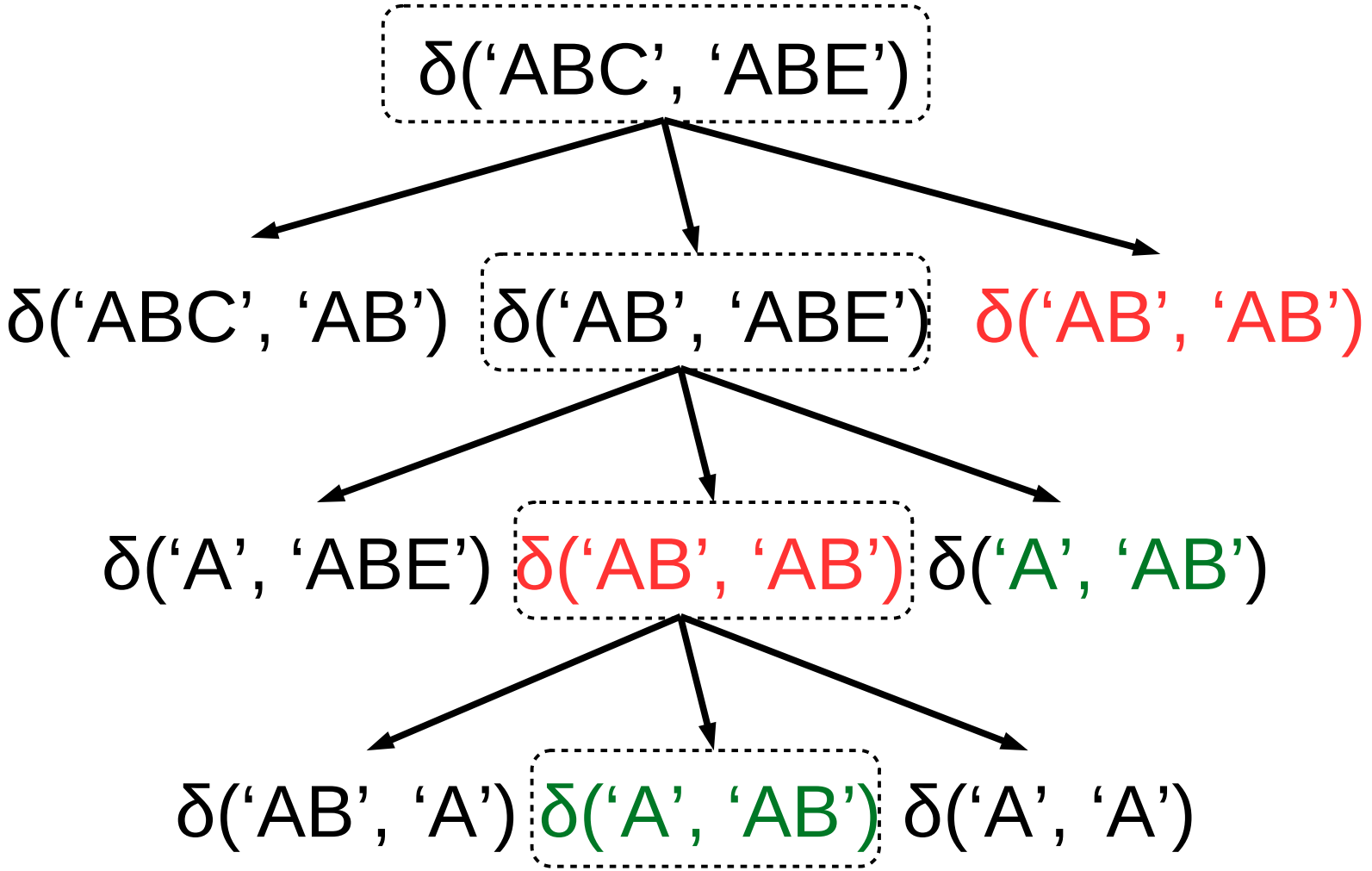
Let’s have a look at the functions calls



Naive implementation
→ exponential complexity



Redundant computations !



Dynamic programming

Be careful not to blindly implement the recursion!

Store intermediate results in a table to avoid redundant computations

Needleman-Wunsch algorithm

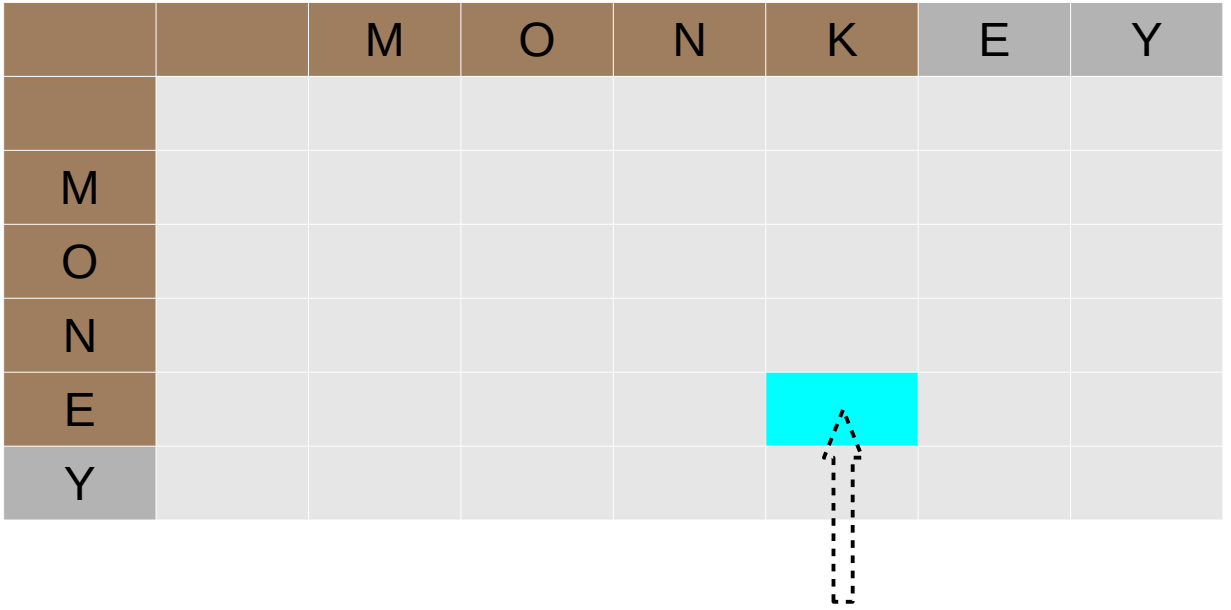
→ Computes the edit distance

→ 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.



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

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	??				
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						

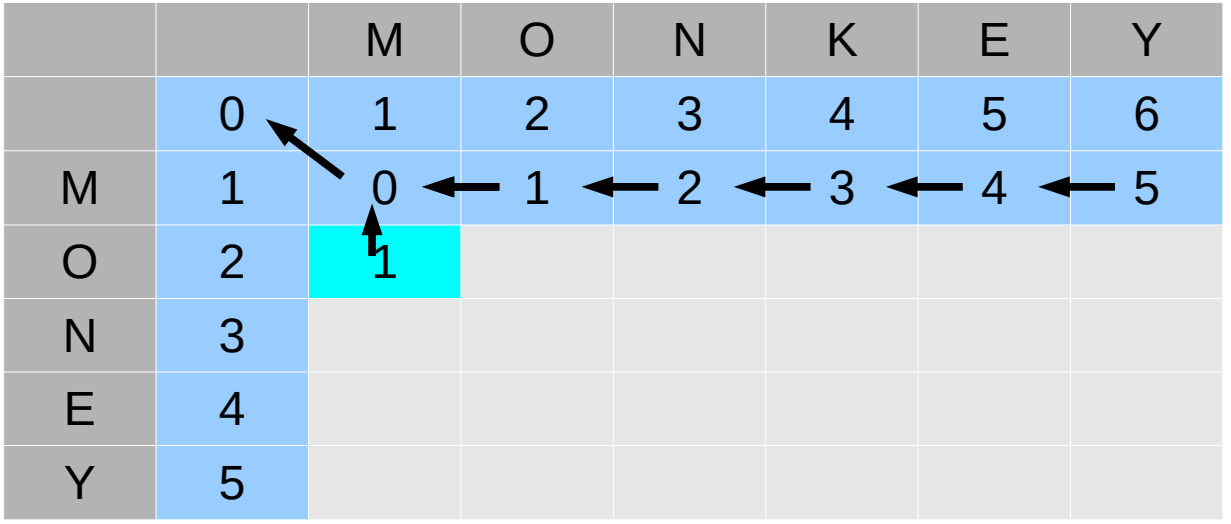
Needleman-Wunsch algorithm

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	1					
N	3						
E	4						
Y	5						

Needleman-Wunsch algorithm

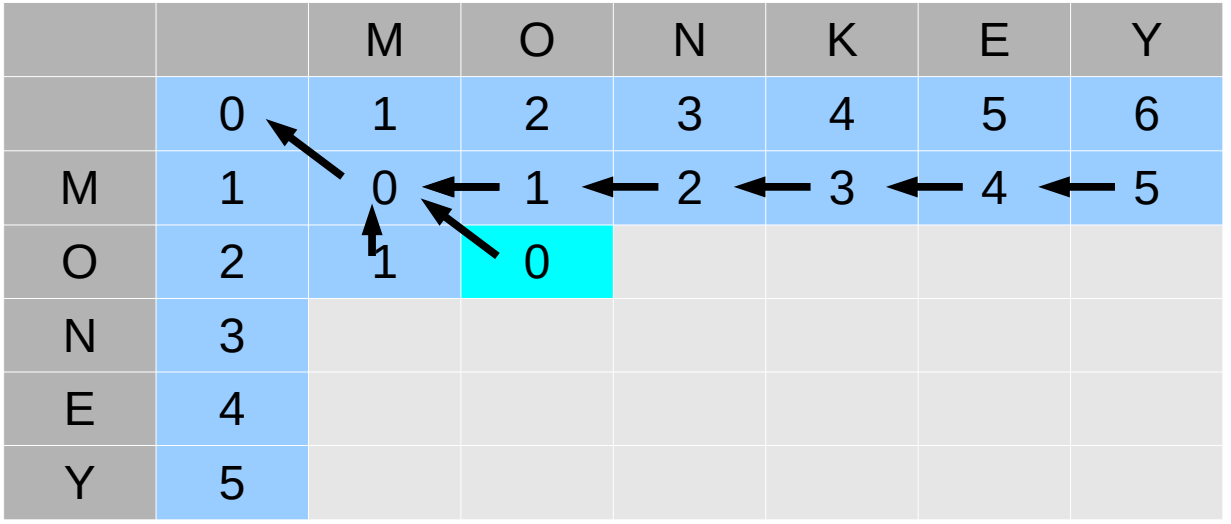
After filling a cell, keep track of the best path



Vertical path: deletion of ‘O’

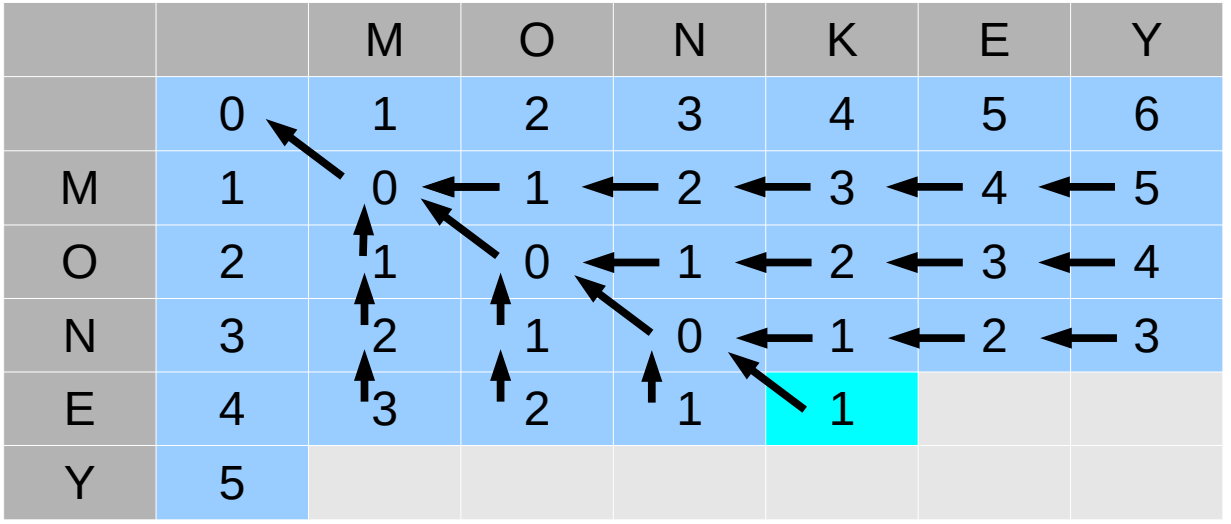
Needleman-Wunsch algorithm

Iterate...



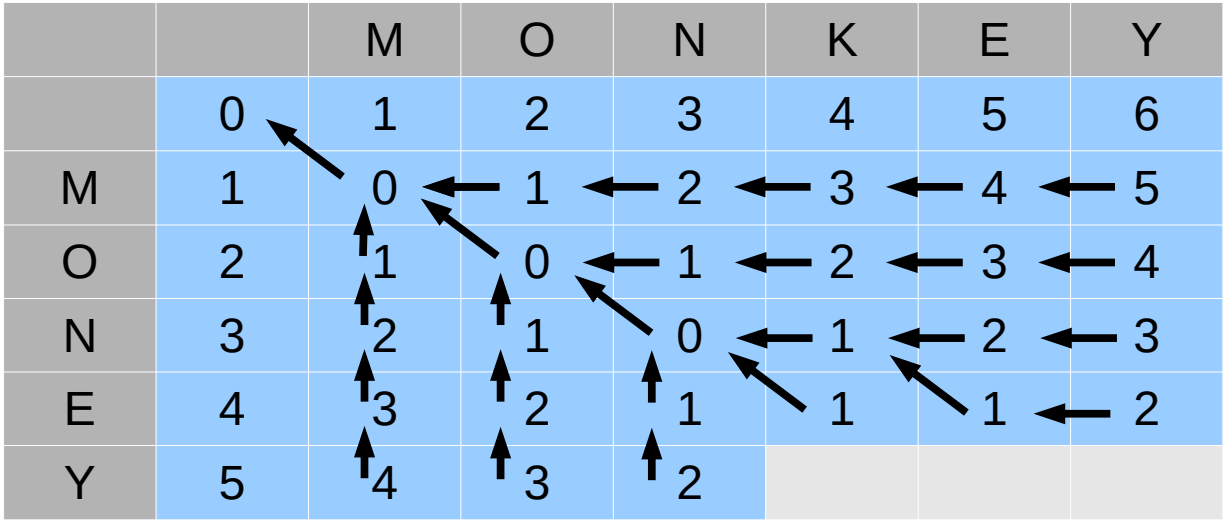
Needleman-Wunsch algorithm

Iterate...



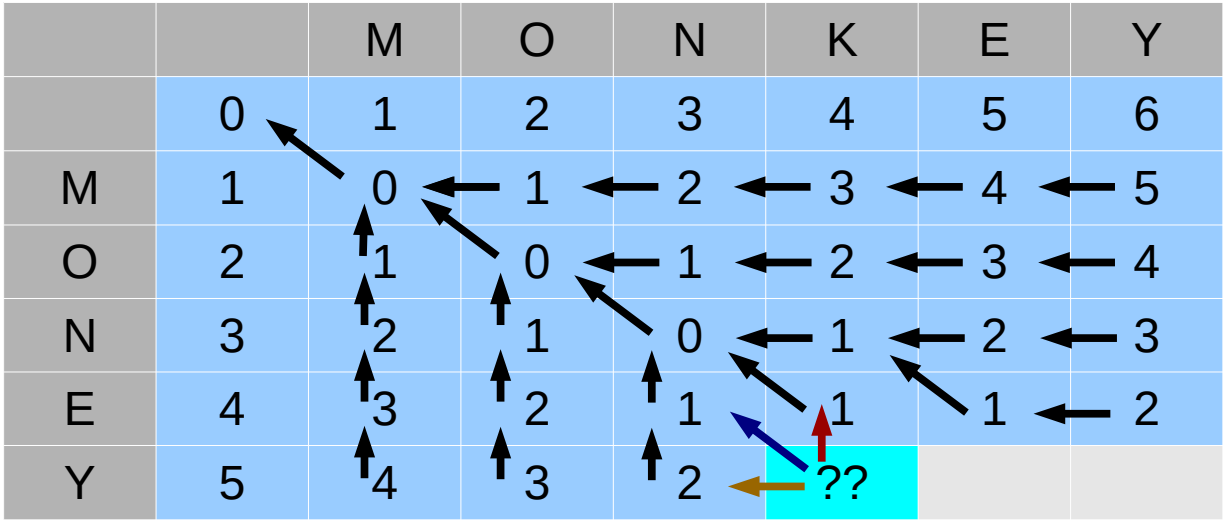
Needleman-Wunsch algorithm

Iterate...



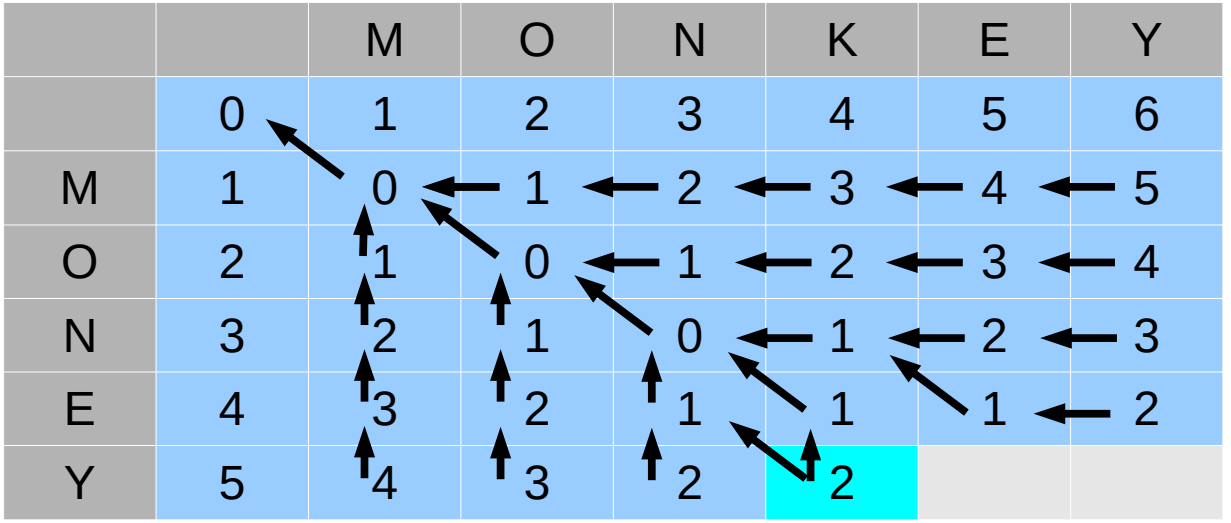
Needleman-Wunsch algorithm

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



Needleman-Wunsch algorithm

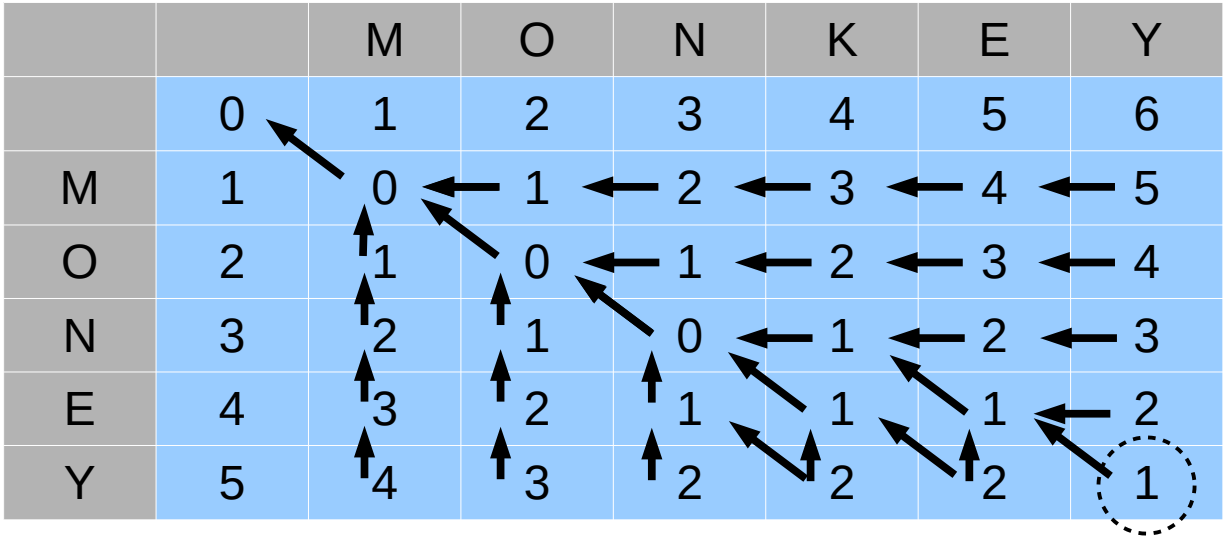
Sometimes, there are several best paths



Ambiguity!

Needleman-Wunsch algorithm

Edit distance between monkey and money = 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

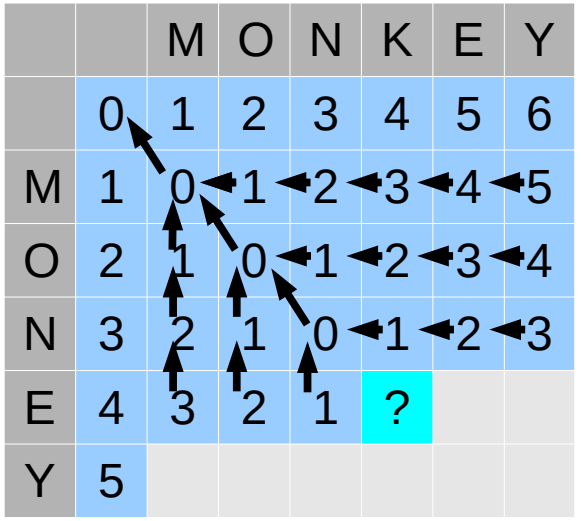
- Initialize first row and first column

		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						



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						

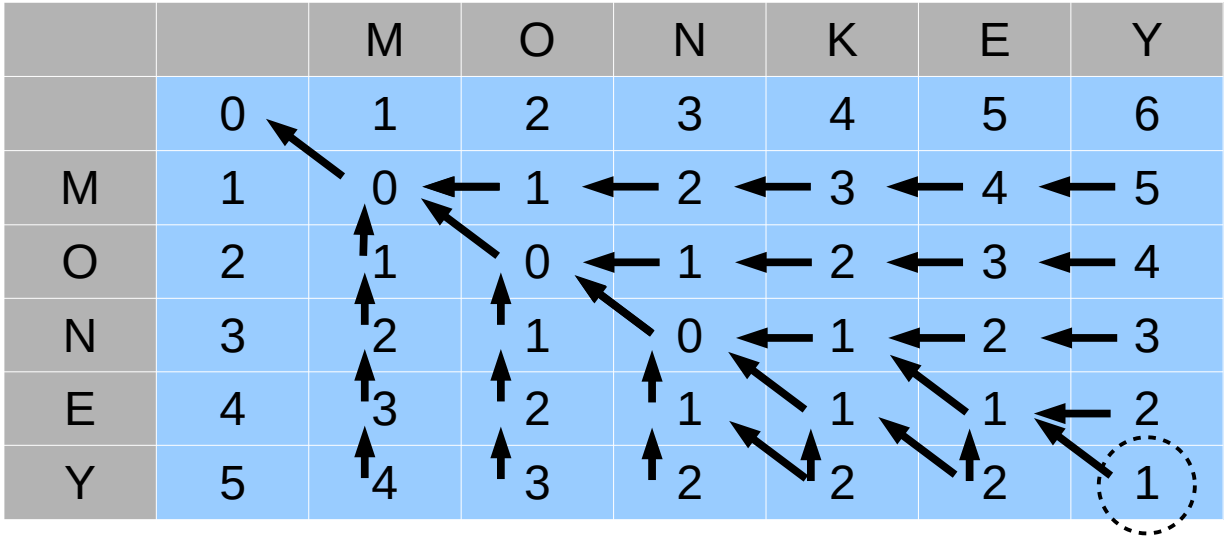
		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

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

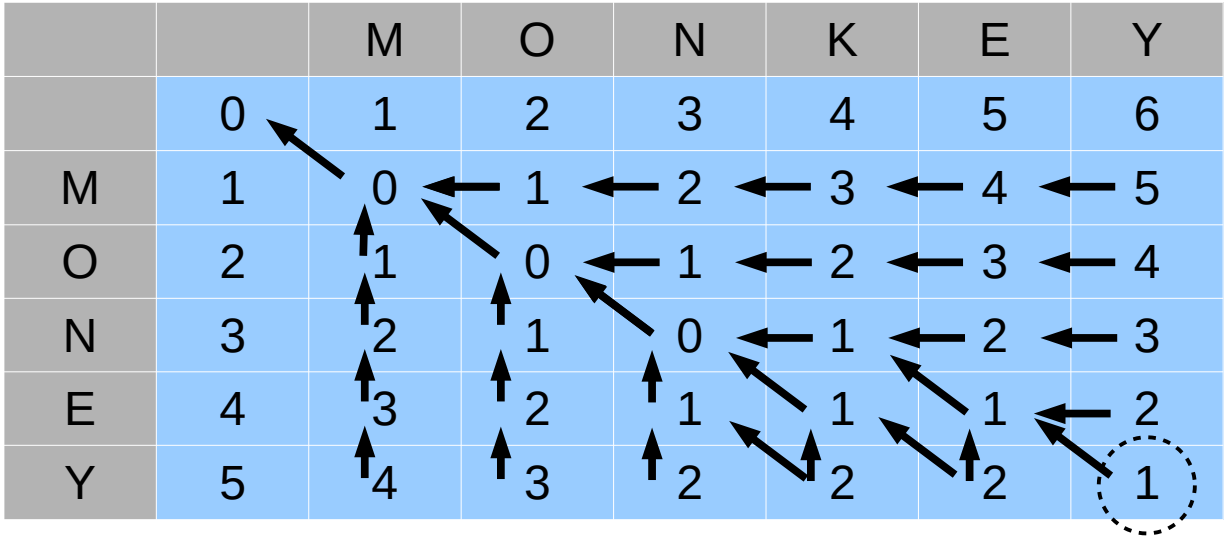
Complexity

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



Complexity

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



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?

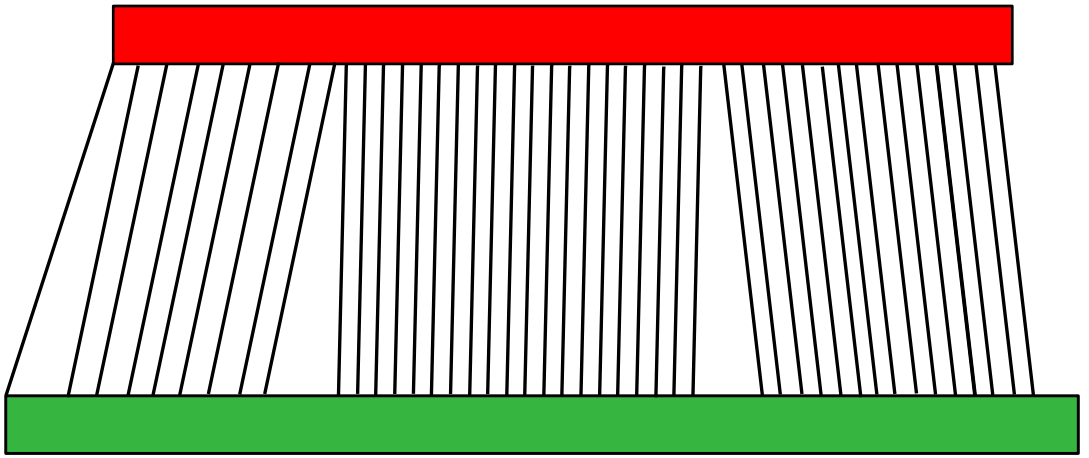
It depends!

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

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

Answer: global alignment

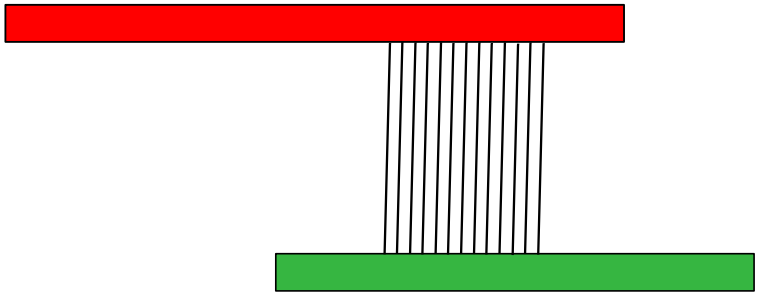
- Needleman-Wunsch gives us the best sequence of operations to get Y from X.
- This corresponds to the best global alignment.



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

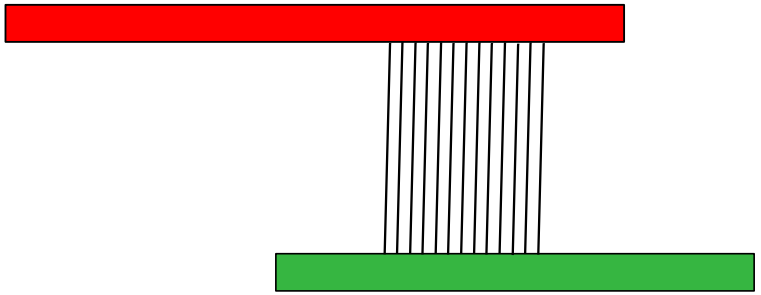
"*Find the substrings with maximum similarity*"

S(ACCACAAC, ACCATCAAC) = 8 – 1 = 7

S(ACCA, ACCA) = 4

Local alignment problem

Find the two substrings of X and Y with the maximal similarity.



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

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

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?

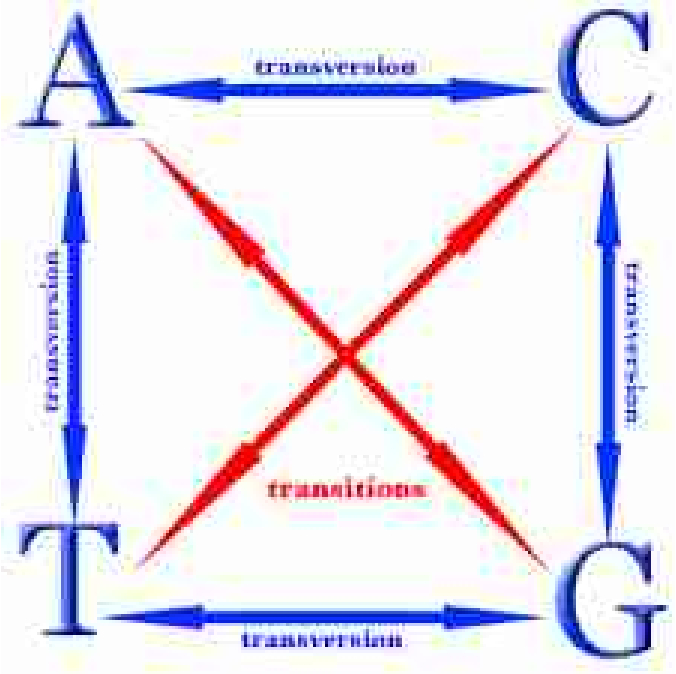
  No!

- 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 (F(i, j - 1) + ins(b),
  F(i – 1, j) + del(a),
  F(i – 1, j - 1) + subst(a,b))

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

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

BLOSUM matrices

- BLOSUM: BLOcks of Amino Acid SUbstitution Matrix
- The BLOSUM matrices are reference substitution matrices used for protein sequence alignment
- Computed from empirical local alignments

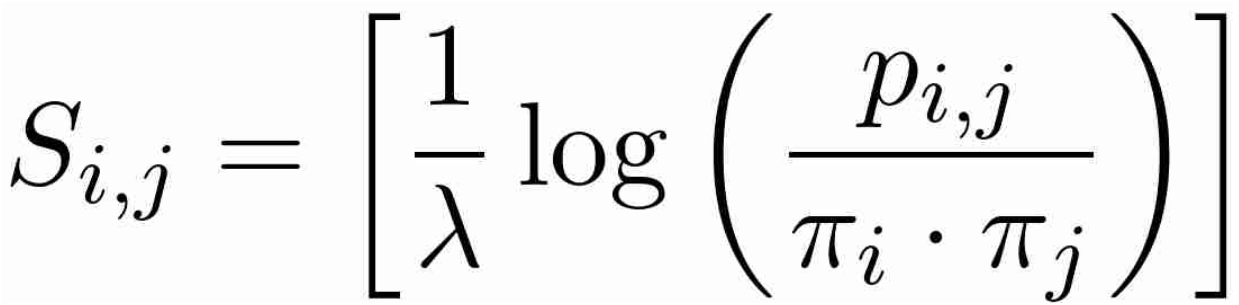
BLOSUM matrices

- There are several BLOSUM matrices named with numbers.

- High numbers (BLOSUM80) → close sequences

- Low numbers (BLOSUM45) → distant sequences

Log-odds scores



$$S_{i,j} = \left[\frac{1}{\lambda} \log \left(\frac{p_{i,j}}{\pi_i \cdot \pi_j} \right) \right]$$

Observed frequency of substitution

Expected frequency of substitution

scaling factor to obtain easily computable integer values.

(πi: frequency of character i in the sequences)

Log-odds scores

Observed frequency of substitution

$$S_{i,j} = \left[\frac{1}{\lambda} \log \left(\frac{p_{i,j}}{\pi_i \cdot \pi_j} \right) \right]$$

Expected frequency of substitution

scaling factor to obtain easily computable integer values.

Very likely substitution → high log-odds score
Unlikely substitution → low log-odds score

BLOSUM62

(values are rounded to obtain integers)

-	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	9	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	4	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	4	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	7	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	4	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	6	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	6	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	6	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	5	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	5	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	8	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	5	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11

Hamming distance to handle mismatches

- Let X be a long string and y be a short string.
- Find all the substrings x of X such that
  - |x| = |y|
  - δh(x, y) <= k (hamming distance between x and y does not exceed k)

(Example: quickly align a read to a reference sequence)

Example

- X = “ADCABCAABADBBCA”, Y = “ADBBCA”, k = 3

- 2 solutions:

```
D   C   A   B   C   A
.   .   .   |   |   |
A   D   B   B   C   A
```

```
A   D   B   B   C   A
|   |   |   |   |   |
A   D   B   B   C   A
```

Example

X = “ADCABCAABADBBCA”, Y = “ADBBCA”, k = 3

Two solutions:

A**DCABCA**ABADBBCA

ADBBCA

ADCABCAAB**ADBBCA**

ADBBCA

Recursion/DP formula

- Initialize first row with 0
- Initialize first column with k + 1
- T[i][j] = T[i-1][j-1] + subs(i,j)
- Look at the last line and keep all alignment with a score that does not exceed k

Hamming distance to handle mismatches

- X = “ADCABCAABADBBCA”, Y = “ADBBCA”, k = 3

D	-	A	D	C	A	B	C	A	A	B	A	D	B	B	C	A
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	4	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
D	4	5	0	2	2	1	2	2	1	1	2	0	2	2	2	2
B	4	5	6	1	3	2	2	3	3	1	2	3	0	2	3	3
B	4	5	6	7	2	3	3	3	4	3	2	3	3	0	3	4
C	4	5	6	6	8	3	3	4	4	5	4	3	4	4	0	4
A	4	4	6	7	6	9	4	3	4	5	5	5	4	5	5	0

```
D   C   A   B   C   A
.   .   .   |   |   |
A   D   B   B   C   A
```

```
A   D   B   B   C   A
|   |   |   |   |   |
A   D   B   B   C   A
```

Remarks

For this problem:

- we don’t need to keep the whole table in memory (we only use diagonal paths)
- We can implement early stopping

D	-	A	D	C	A	B	C	A	A	B	A	D	B	B	C	A
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	4	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
D	4	5	0	2	2	1	2	2	1	1	2	0	2	2	2	2
B	4	5	6	1	3	2	2	3	3	1	2	3	0	2	3	3
B	4	5	6	7	2	3	3	3	4	3	2	3	3	0	3	4
C	4	5	6	6	8	3	3	4	4	5	4	3	4	4	0	4
A	4	4	6	7	6	9	4	3	4	5	5	5	4	5	5	0

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, Blosum, gap penalties

Questions, feedback?

lukas.huebner@h-its.org