1 Alignment

We have seen two symmetrical approaches to the problem of "comparing sequences": the edit distance measures how different two strings are, and the longest common subsequence, that measures how similar they are. **What if we combine the two approaches into a unique one?**

Idea: given two strings S and T, define a similarity measure where

- *differences* have a negative effect: define **negative scores** for the weight of **edit operations** (insertions, deletions, substitutions).
- *conserved letters* have a positive effect: define **positive scores** for the weight of **matches** between letters of the two strings.

The goal is now to **find the alignment that maximises the score** resulting by the weights just defined.

1.1 Global alignment

The goal of **global alignment** is to **try to align every character in every sequence**. We can find a solution for this problem once again using dynamic programming, in a similar way to the previous algorithms.

The rule for calculating the cell values is a combination of what we have seen so far:

$$E(S(i),T(j)) = \mathbf{max} \begin{cases} E(S(i),T(j-1)) + \mathbf{w_d} & \\ E(S(i-1),T(j)) + \mathbf{w_d} & \\ E(S(i-1),T(j-1)) + \mathbf{w_m} & \text{if} \quad s_i = t_j \\ E(S(i-1),T(j-1)) + \mathbf{w_s} & \text{if} \quad s_i \neq t_j \end{cases}$$

We are still looking for the "max": we're trying to find the alignment with the maximum score. As before, we can use pointers to keep track of the choice made for each cell.

For the example we set $\mathbf{w_d = -2, w_s = -1, w_m = 1}$.

	-	**H**	**O**	**M**	**E**
-	0	-2	-4	-6	-8
H	-2	1	-1	-3	-5
O	-4	-1	2	0	-2
U	-6	-3	0	1	-1
S	-8	-5	-2	-1	0
E	-10	-7	-4	-3	0

Backtracking as before we can build the alignment:

```
H O M - E
H O U S E
```

1.1.1 Needleman-Wunsch algorithm

The Needleman-Wunsch is the most famous global alignment algorithm and is exactly the same as above but formalized in a different way.

$$M(i,j) = \mathbf{max} \begin{cases} M(i,j-1) + \mathbf{g} \\ M(i-1,j) + \mathbf{g} \\ M(i-1,j-1) + \boldsymbol{\sigma}(\mathbf{s_i}, \mathbf{t_j}) \end{cases}$$

We no longer distinguish matches/mismatches: the score of aligning s_i and t_j **is already included in the substitution matrix $\boldsymbol{\sigma(s_i, t_j)}$**

Given an alphabet **$\boldsymbol{\Sigma}$ the substitution matrix** is a $|\Sigma| \times |\Sigma|$ matrix that for every pair of characters $a_i, a_j \in \Sigma$ gives us the **"weight" in the alignment of substituting $\mathbf{a_i}$ with $\mathbf{a_j}$**. It has the following properties:

- the matrix is symmetrical, i.e. $\sigma(a_i, a_j) = \sigma(a_j, a_i)$.
- given a_i, the values $\sigma(a_i, a_j)$ for all the characters a_j can be different in the matrix.
- values for "matches" $\sigma(a_i, a_i)$ can be different for every letter a_i.

The simplest substitution matrices are the ones used for DNA/RNA sequences. The scores for matches (positive) and substitutions (negative) do not change with nucleotides. For example:

	A	**C**	**G**	**T**
A	+2	-1	-1	-1
C	-1	+2	-1	-1
G	-1	-1	+2	-1
T	-1	-1	-1	+2

1.2 Local alignment

DNA and the related molecules (RNA, proteins) evolve at different speeds, so in a genome, some regions tend to be more conserved by evolution than others. This means that, in several cases, an **evolutionary relationship can be found only for a part of the sequences** compared, but not for them globally.

This is the problem of local alignment: **find the two substrings, one from S and the other from T, that produce the alignment with the maximum score**.

Notice that: since **negative scores are possible**, if there is no pair of substrings with an alignment with score > 0, then the best solution is trivially given by two empty strings with score 0.

Notice also that if we had only positive or null scores, then the best solution would be the alignment of the two whole strings.

1.2.1 Smith Waterman algorithm

This algorithm was designed explicitely for the comparison of biological sequences and **guarantees to find the best (optimal) solution**, and is based again on the dynamic programming matrix.

We start by building a table with $(n+1) \times (m+1)$ cells as in the global alignment.

Cell (i, j) will contain the score of the alignment of a substring ending at i in S and a substring ending at j in T. From this we can already fill the first row and the first column with zeros since **the best possible score between an empty string and any other string is 0**.

The formula for filling each cell is similar to all the others.

$$M(i,j) = \max \begin{cases} \mathbf{0} \\ M(i, j-1) + g \\ M(i-1, j) + g \\ M(i-1, j-1) + \sigma(s_i, t_j) \end{cases}$$

The main different is that when computing the cell value, **if all the possible alignment extensions lead to a negative score, it's better to "reset" everything and start from 0 again**.

Also, **pointers are not saved for cells with a value of 0**.

In the following example we set $g = -2$ (insertions, deletions), $\sigma(a_i, a_j) = -1$ for every i, j (mismatch), $\sigma(a_k, a_k) = 1$ for every i (match).

	-	**H**	**O**	**M**	**E**
-	0	0	0	0	0
H	0	↖ 1	0	0	0
O	0	0	↖ 2	0	0
U	0	0	0	↖ 1	0
S	0	0	0	0	0
E	0	0	0	0	↖ 1

The **traceback** process **doesn't start from the bottom right cell**, but instead it **starts from the cell with the maximum score**, and follows the paths back until a cell with a score of 0, then it stops.

For the above example, the two substrings that produce the highest alignment score are

```
H O
H O
```

Once we have found, traced back, and aligned the best pair of strings, we can iterate the process finding the "second best" score in cells that haven't been used previously. And then the third, fourth, and so on (remember that the algorithm can be applied to sequences of hundreds or thousands of nucleotides/aminoacids).