**BioE 131/231**
**Fall 2018**

**Lab Assignment #10**

**Overview and references**

In previous labs, you used MUSCLE to generate a multiple alignment, aligned short sequencing reads to a reference with Bowtie2, and identified a sequence by aligning it against a large database with BLAST. Each of these aligners uses a different heuristic to increase speed, but all use either Smith-Waterman (SW) or Needleman-Wunsch (NW) algorithms at some point.

SW and NW are dynamic programming algorithms that produce the optimal local (SW) or global (NW) alignment between two sequences, given a scoring matrix. In this assignment, you will implement your own version of NW in Python by filling in parts of the provided code. There are a lot of implementations of this algorithm out there and you are encouraged to use them as a reference, but please write your own code following your own logic. NW will be covered on the final, so take this opportunity to understand it!

This assignment is a little different because we're asking you to complete a Python script. Feel free to debug in iPython, but please hand in a single .py file this time rather than a notebook or GitHub link.

**Smith-Waterman**
https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

**Needleman-Wunsch**
https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

**Dynamic Programming**
https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

## Getting Started

Find the Python script for the assignment on bCourses. Parts of the code that you must write are marked with “TODO” labels and a short description of what must be done.

Your script should accept two positional command-line arguments: (1) the path to the scoring matrix file, and (2) the path to the multi-FASTA to be aligned. It will be run like this:

```
script.py scoringmatrix inputfasta
```

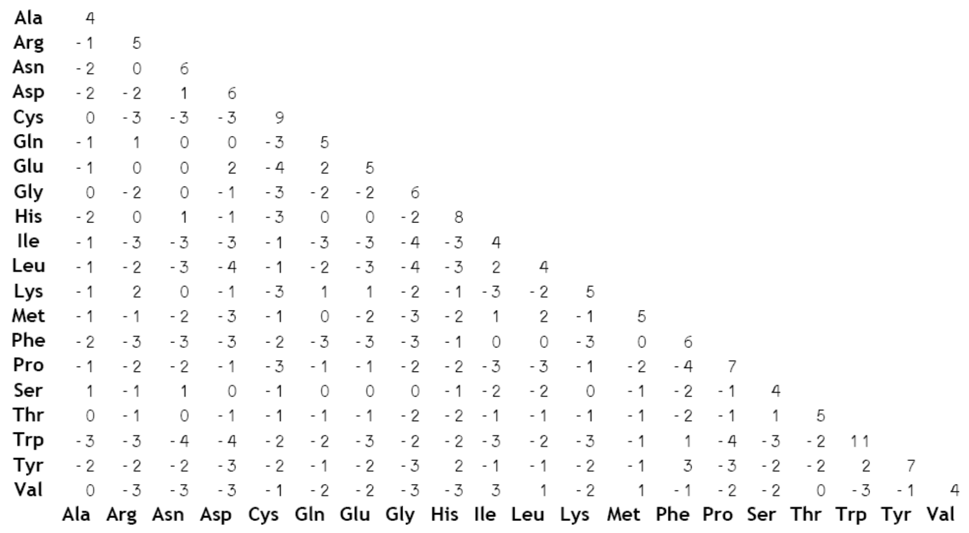
## Read in a scoring matrix

Scoring matrices specify the letters which are allowed in the input sequences, and the scores and penalties of aligning them to each other and to a gap. They need not be symmetrical, but in practice, most commonly are. Your script should be agnostic to the type of sequence being provided, i.e. it should be able to handle both proteins and nucleic acids by changing the scoring matrix provided. For example, here is a DNA scoring matrix:

```
A T G C
5 -4 -4 -4
-4 5 -4 -4
-4 -4 5 -4
-4 -4 -4 5
-10
```

The first line in the file contains the allowed letters, separated by a space. The next N rows contain the scores for each nucleotide being matched with each other nucleotide, where there are N such rows. The final line of the file contains the score for aligning a nucleotide with a gap--in this example, that's -10. A closer inspection of this matrix shows that we are assigning a -4 penalty for nucleotide mismatch (i.e. A aligning to T, or C aligning to A) and a +5 score for an exact match. This matrix is symmetrical.

This next example is a protein scoring matrix. Specifically, it's BLOSUM62 that is commonly used by the BLAST algorithm.



| | Ala | Arg | Asn | Asp | Cys | Gln | Glu | Gly | His | Ile | Leu | Lys | Met | Phe | Pro | Ser | Thr | Trp | Tyr | Val |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ala** | 4 | | | | | | | | | | | | | | | | | | | |
| **Arg** | -1 | 5 | | | | | | | | | | | | | | | | | | |
| **Asn** | -2 | 0 | 6 | | | | | | | | | | | | | | | | | |
| **Asp** | -2 | -2 | 1 | 6 | | | | | | | | | | | | | | | | |
| **Cys** | 0 | -3 | -3 | -3 | 9 | | | | | | | | | | | | | | | |
| **Gln** | -1 | 1 | 0 | 0 | -3 | 5 | | | | | | | | | | | | | | |
| **Glu** | -1 | 0 | 0 | 2 | -4 | 2 | 5 | | | | | | | | | | | | | |
| **Gly** | 0 | -2 | 0 | -1 | -3 | -2 | -2 | 6 | | | | | | | | | | | | |
| **His** | -2 | 0 | 1 | -1 | -3 | 0 | 0 | -2 | 8 | | | | | | | | | | | |
| **Ile** | -1 | -3 | -3 | -3 | -1 | -3 | -3 | -4 | -3 | 4 | | | | | | | | | | |
| **Leu** | -1 | -2 | -3 | -4 | -1 | -2 | -3 | -4 | -3 | 2 | 4 | | | | | | | | | |
| **Lys** | -1 | 2 | 0 | -1 | -3 | 1 | 1 | -2 | -1 | -3 | -2 | 5 | | | | | | | | |
| **Met** | -1 | -1 | -2 | -3 | -1 | 0 | -2 | -3 | -2 | 1 | 2 | -1 | 5 | | | | | | | |
| **Phe** | -2 | -3 | -3 | -3 | -2 | -3 | -3 | -3 | -1 | 0 | 0 | -3 | 0 | 6 | | | | | | |
| **Pro** | -1 | -2 | -2 | -1 | -3 | -1 | -1 | -2 | -2 | -3 | -3 | -1 | -2 | -4 | 7 | | | | | |
| **Ser** | 1 | -1 | 1 | 0 | -1 | 0 | 0 | 0 | -1 | -2 | -2 | 0 | -1 | -2 | -1 | 4 | | | | |
| **Thr** | 0 | -1 | 0 | -1 | -1 | -1 | -1 | -2 | -2 | -1 | -1 | -1 | -1 | -2 | -1 | 1 | 5 | | | |
| **Trp** | -3 | -3 | -4 | -4 | -2 | -2 | -3 | -2 | -2 | -3 | -2 | -3 | -1 | 1 | -4 | -3 | -2 | 11 | | |
| **Tyr** | -2 | -2 | -2 | -3 | -2 | -1 | -2 | -3 | 2 | -1 | -1 | -2 | -1 | 3 | -3 | -2 | -2 | 2 | 7 | |
| **Val** | 0 | -3 | -3 | -3 | -1 | -2 | -2 | -3 | -3 | 3 | 1 | -2 | 1 | -1 | -2 | -2 | 0 | -3 | -1 | 4 |

BLOSUM62 was computationally generated from hand-curated multiple alignments of sequences that we rationally believe to be related to each other. The idea is that these scores will pull out alignments from the database that reflect what we believe about certain amino acids playing similar roles in protein structure, i.e. negatively-charged residues replacing other negatively-charged residues, hydrophobic residues replacing hydrophobic residues, etc. See that "11" score in the middle there? That's for tryptophan exactly aligning with tryptophan. It is *so* favored to align tryptophan with itself, that the score is an 11!

You will have to parse this scoring matrix and store it in memory.

**Fill the dynamic programming matrix**

|   | – | A | T | A | G | G | C |
|---|---|---|---|---|---|---|---|
| – |   |   |   |   |   |   |   |
| A |   |   |   |   |   |   |   |
| A |   |   |   |   |   |   |   |
| G |   |   |   |   |   |   |   |
| G |   |   |   |   |   |   |   |
| C |   |   |   |   |   |   |   |

First, you will need to initialize an empty matrix of size N+1 by M+1 where N and M are the lengths of the two sequences you are aligning. They need not be the same length! The +1 is for gaps. In this below example, we are going to align sequence M, ATAGGC, to sequence N, AAGGC. Note that it looks like this matrix is bigger than N+1 by M+1, but that's just so I can label rows and columns.

|   | – | A | T | A | G | G | C |
|---|---|---|---|---|---|---|---|
| – | 0 | -10 | -20 | -30 | -40 | -50 | -60 |
| A | -10 |   |   |   |   |   |   |
| A | -20 |   |   |   |   |   |   |
| G | -30 |   |   |   |   |   |   |
| G | -40 |   |   |   |   |   |   |
| C | -50 |   |   |   |   |   |   |

Next, populate the first row and first column of this matrix with the "boundary conditions" that each sequence is aligned entirely to gaps. We know that we start with a score of zero, so the origin of the dynamic programming matrix is set to that. We also know that given our simple DNA scoring matrix, a gap is worth -10 points. Therefore, as we go down the first row and across the first column, we accumulate -10 for each position matched to a gap.

In the follow steps, beginning with the second row, fill each element in the matrix with the maximum of the following three options:

1. The value from the cell diagonally up and to the left of the element + the score of the two nucleotides exactly matching
2. The value from above + the gap penalty
3. The value from the left + the gap penalty

Beginning with the first empty element in the matrix, we can see that an exact match of A with A will yield a score of 0 + 5 = 5, whereas introducing a gap would produce a score of -10 + -10 = -20 from above or -10 + -10 = -20 from the left. Thus, we fill that element with a **value of 5**.

| | – | A | T | A | G | G | C |
|---|---|---|---|---|---|---|---|
| **–** | 0 | -10 | -20 | -30 | -40 | -50 | -60 |
| **A** | -10 | 5 | **-5** | -15 | -25 | -35 | -45 |
| **A** | -20 | | | | | | |
| **G** | -30 | | | | | | |
| **G** | -40 | | | | | | |
| **C** | -50 | | | | | | |

The next element is a choice between -10 + -4 = -14 for aligning A with T, or 5 + -10 = -5 for introducing a gap from the left, or -20 + -10 = -30 for introducing a gap from above. Thus, we choose to introduce a gap from the left and fill that element with a **value of -5**.

Next, we move on to the third row, then the fourth, and so on until the matrix is completed.

The matrix is completed in a "greedy" fashion--that is, each element in the matrix is the best-scoring move based on the options available immediately to that position. Despite this, however, the solution produced by the algorithm will be optimal.

| | – | A | T | A | G | G | C |
|---|---|---|---|---|---|---|---|
| **–** | 0 | -10 | -20 | -30 | -40 | -50 | -60 |
| **A** | -10 | 5 | -5 | -15 | -25 | -35 | -45 |
| **A** | -20 | -5 | 1 | 0 | -10 | -20 | -30 |
| **G** | -30 | -15 | -9 | -3 | 5 | -5 | -15 |
| **G** | -40 | -25 | -19 | -13 | 2 | 10 | 0 |
| **C** | -50 | -35 | -29 | -23 | -8 | 0 | 15 |

Here, the bottom-right element in the matrix is filled with the optimal score of aligning the two sequences, i.e. 15.

## Generate alignment by backtracing through the matrix and print it to STDOUT

After completing the matrix, an alignment can be obtained by backtracing through the matrix, starting at the bottom-right element and ending at the origin. At each element, we must ask ourselves which of the three options available to us were used to produce that value.

That is to say, is the element equal to...

1. The value of the cell up and to the left + the value from the scoring matrix for a match?
2. The value of the cell to the left + the gap penalty?
3. The value of the cell above + the gap penalty?

In our example from above, this translates to asking...

1. Does 15 = 10 + 5?
2. Does 15 = 0 - 10?
3. Does 15 = 0 - 10?

Depending on which is true, we will either add the matching character from each of the input sequences, a gap to sequence M, or a gap to sequence N.

1. Case 1 means both characters in M and N were aligned
2. Case 2 means the character in sequence M was aligned to a gap
3. Case 3 means the character in sequence N was aligned to a gap

**NOTE:** There can be multiple equally good choices at a given element. This means there are multiple optimal alignments.

Completing this example, let's work our way all the way back to the origin and produce the optimal alignment as worked out in the following chunk of pseudocode or whatever you would call this.

**NOTE:** Beginning at the bottom-right corner, we will be calculating the optimal alignment in reverse. That is, the first match happens at the *end* of the sequence, and the origin will be the *beginning*.

```
BEGIN

Set Position = (6,5)

Value = 15
We got here by aligning these two positions.
AlignmentM = "C"
AlignmentN = "C"
Move diagonally to Position = (5,4)

Value = 10
We got here by aligning these two positions.
AlignmentM = "GC"
AlignmentN = "GC"
Move diagonally to Position = (4,3)

Value = 5
We got here by aligning these two positions.
AlignmentM = "GGC"
AlignmentN = "GGC"
Move diagonally to Position = (3,2)

Value = 0
We got here by aligning these two positions.
AlignmentM = "AGGC"
AlignmentN = "AGGC"
Move diagonally to Position = (2,1)

Value = -5
We got here by introducing a gap in N.
AlignmentM = "TAGGC"
AlignmentN = "-AGGC"
Move left to Position = (1,1)

Value = 5
We got here by aligning these two positions.
AlignmentM = "ATAGGC"
AlignmentN = "A-AGGC"
Move diagonally to Position = (0,0)

We have reached the origin (0,0)

END
```