In this homework, we will write programs to compute the length of the Longest Common Subsequence (LCS), the Global Alignment, and the Local Alignment of two strings.

1 Longest Common Subsequence (LCS)

1.1 Compute the LCS length

Write a function that computes the length of the longest common subsequence between two strings. This is the same as the second part of Lab8; the code is also available in lecture.

`LCS()`: *Compute the length of the longest common subsequence between two strings.*
Inputs: Two strings, `s1` and `s2`
Returns: The length of the longest common subsequence between `s1` and `s2`.

To compute the length of the LCS, you fill a table that has `len(s1)+1` rows and `len(s2)+1` columns. The entry at $table[0][0]$ is 0. For all other values of i and j, compute the values using this recurrence.

$$table[i][j] = \max \begin{cases} table[i-1][j] & \text{if } i > 0 \text{ (deletion)} \\ table[i][j-1] & \text{if } j > 0 \text{ (insertion)} \\ table[i-1][j-1] + m(s1[i-1], s2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \text{ (match/mismatch)} \end{cases}$$

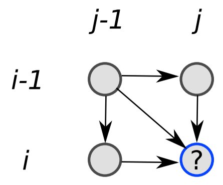
where $m(s1[i-1], s2[j-1])$ is 1 if the characters match and 0 otherwise.

Use very short strings (e.g., `TGT` and `GTAA`) as you work on this function. The strings need not be the same length. Test your function on the strings from lecture: `ATGTTATA` and `ATCGTCC`. Make sure it works when using either string as `string1` (this will fill a table with different dimensions).

1.2 Compute an alignment with the longest common subsequence

Modify `LCS` to also return an alignment. To compute the alignment, we need to keep track of the decision made at each node. At each node ("intersection") `table[i][j]`, we made one of three options:

Did we take an edge **south**? (Did we come from `table[i-1][j]`?)
Did we take an edge **east**? (Did we come from `table[i][j-1]`?)
Did we take an edge **diagonally**? (Did we come from `table[i-1][j-1]`?)



1. Create a new table, `backtrack`, that has exactly the same dimensions as `table`. Update each entry in `backtrack` with the decision you made (south/east/diagonal) as you fill `table`. If there are ties, pick any one.

2. Starting from the bottom right corner, build the alignment "backwards" by tracing the decisions made at each point. You will need to use a `while` loop for this, since the length of the alignment is unknown.

Note: Your alignment may differ from others depending on how you pick ties; however, you the columns with matching symbols should make up the longest common subsequence.

2 Global Sequence Alignment

The recurrence for global sequence alignment differs from LCS in only a few places.

`GlobalAlignment()`: *Compute the optimal global alignment between two strings.*
Inputs: Two strings, `s1` and `s2`, and three integers, *indel*, *matchscore*, *mismatchpenalty*.
Returns: (1) The score of the optimal global alignment, and (2) the alignment itself.

To compute this, you fill a table that has `len(s1)+1` rows and `len(s2)+1` columns. The entry at $table[0][0]$ is 0. For all other values of i and j, compute the values using this recurrence.

$$table[i][j] = \max \begin{cases} table[i-1][j] - indel & \text{if } i > 0 \text{ (deletion)} \\ table[i][j-1] - indel & \text{if } j > 0 \text{ (insertion)} \\ table[i-1][j-1] + score(s1[i-1], s2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \text{ (match/mismatch)} \end{cases}$$

where

$$score(s1[i-1], s2[j-1]) = \begin{cases} matchscore & \text{if the characters } s1[i-1] \text{ and } s2[j-1] \text{ match} \\ -mismatchpenalty & \text{if the characters } s1[i-1] \text{ and } s2[j-1] \text{ don't match} \end{cases}$$

and *indel* is a penalty for an insertion or a deletion. Note that the indel and mismatch penalties are given as **positive numbers**.

Run your function with the strings from lecture (`ATGTTATA` and `ATCGTCC`) using a match score of +1, a mismatch penalty of -1, and three different indel penalties of 0, 0.05, and 1. The three global alignments will be different from each other.

3 Local Sequence Alignment

The recurrence for global sequence alignment differs from global alignment **in only one place**:

$$table[i][j] = \max \begin{cases} table[i-1][j] - indel & \text{if } i > 0 \text{ (deletion)} \\ table[i][j-1] - indel & \text{if } j > 0 \text{ (insertion)} \\ table[i-1][j-1] + score(s1[i-1], s2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \text{ (match/mismatch)} \\ 0 & \text{``free taxi ride''} \end{cases}$$

All other inputs, outputs, and scoring functions remain the same.

Your backtracking procedure to find the optimal local alignment will change. When implementing backtracking, you must find the **largest value** in the dynamic programming table. You may also have to adjust the `while` loop in case your alignment doesn't begin at (0,0). Use the strings from lecture (`ATGTTATA` and `ATCGTCC`) and an indel penalty of 1, mismatch penalty of 1, and match score of 1. The optimal local alignment has a score of `3`, and may look like this:

```
AT-GT
ATCGT
```

Note there may be ties, so your program may produce a different local alignment with a score of `3`.

4 Protein Sequence Alignment

Your programs should now work for protein alignment! For this, we will copy the local alignment code to allow scores to come from a scoring dictionary. I have provided a variable called `blosum62` in `amino_acid_scoring.py`. This variable is a dictionary where keys are single-letter amino acids ('e.g., `'Y'`) and the values are **dictionaries**. Thus, we can get an integer score for two characters (e.g., matching a `'W'` with a `'Y'`) by using the following code:

```
blosum62['W']['Y']    ## score of a W/Y mismatch
blosum62['D']['Y']    ## score of a D/Y mismatch
blosum62['Y']['Y']    ## score of a Y/Y match
```

Copy the `blosum62` dictionary to your code. Add a new function to compute the local alignment of two peptides:

`LocalAlignmentWithScores()`: *Compute the optimal local alignment between two peptides.*
Inputs: Two peptides, `s1` and `s2`, an integer *indel*, and a scoring dictionary `blosum62`.
Returns: (1) The score of the optimal local alignment, and (2) the alignment itself.

To compute this, you fill a table that has `len(s1)+1` rows and `len(s2)+1` columns. The entry at $table[0][0]$ is 0. For all other values of i and j, compute the values using this recurrence.

$$table[i][j] = \max \begin{cases} table[i-1][j] - indel & \text{if } i > 0 \text{ (deletion)} \\ table[i][j-1] - indel & \text{if } j > 0 \text{ (insertion)} \\ table[i-1][j-1] + score(s1[i-1], s2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \text{ (match/mismatch)} \\ 0 & \text{“free taxi ride”} \end{cases}$$

where $score(aa1, aa2)$ = `blosum62[`$aa1$`][`$aa2$`]` for amino acids $aa1$ and $aa2$; and *indel* is a penalty for an insertion or a deletion.

You can check your function with the examples below:

	Example 1	**Example 2**
String1	`PLEASANTLY`	`PLEASANT`
String2	`MEANLY`	`MEANLY`
Indel Penalty	5	5
Matrix	BLOSUM62	BLOSUM62
Optimal Local Score	16	12
Possible Alignment	`SANTLY`	`LEAS`
	`EAN-LY`	`MEAN`
Another Alignment	`ANTLY`	
	`AN-LY`	

Handin Instructions

Before you submit your code to Moodle, make sure it runs the following examples:

1. Print the LCS score and the LCS alignment for strings `ATGTTATA` and `ATCGTCC`.
2. Print the global alignment (and score) of strings `ATGTTATA` and `ATCGTCC` with an *indel* penalty of 1, a *match* score of 1, and a *mismatch* penalty of 1.
3. Print the local alignment (and score) of strings `ATGTTATA` and `ATCGTCC` with an *indel* penalty of 1, a *match* score of 1, and a *mismatch* penalty of 1.
4. Print the local alignment (and score) of peptides `PLEASANTLY` and `MEANLY` with an *indel* penalty of 5 and the `blosum62` variable.
5. Print the local alignment (and score) of peptides `PLEASANT` and `MEANLY` with an *indel* penalty of 5 and the `blosum62` variable.

You can print additional examples if you wish. Use extra print statements to denote different aligners, e.g.,

```
print('GLOBAL ALIGNMENT (INDEL=1, MATCH=1, MISMATCH=1)')
```

Extra Exercises (Optional)

Solve these problems on Rosalind (log in with your original ID for access to the class).

- LCS Problem: `http://rosalind.info/problems/ba5c/?class=406`
  Note: The output here is the longest common subsequence, rather than an alignment.
- Global Alignment Problem: `http://rosalind.info/problems/ba5e/?class=406`
  Note: This problem uses the `blosum62` variable.
- Local Alignment Problem: `http://rosalind.info/problems/ba5f/?class=406`
  Note: This problem uses a different scoring matrix. You must write a function to read in the PAM250 file and make a dictionary similar to the `blosum62` variable.