Lab 5. Dynamic Programming. In this lab you will implement a program to align pairs of sequences.

A. Here is a warm-up example that introduces a Python feature called __name__, which will be a segue into object oriented programming. Run the following code called test.py in ipython: ">>run test.py AACC AGCT -1." What does the subst_score function do? What does the while loop do? Now try this interesting ipython run:
>>import test
>>test.subst_score('A','A')
The __name__ == "__main__" allows you to run the script as usual or import the python code as a module so you can use its functions. In the first use, the __name__ of the python interpreter is __main__, so if the module (test.py in this case) is called by the interpreter then the module's __name__ becomes __main__. If the module is imported, as in the second use, then the module keeps its name attribute.

```
import sys
from numpy import *

def usage():
      """Help screen"""
      print "Eventually: Dynamic programming sequence alignment"
      print "Usage:  dp.py top_seq side_seq gap-penalty"
      print "Top and side refer to the top and side of the alignment matrix."

def subst_score(a,b):
   # substitution score
   if a==b:
      return 1   # match
   else:
      return -1  # mismatch

### MAIN  ###
if __name__ == "__main__":
   if len(sys.argv) < 3:
      usage()
      sys.exit()
   else:
      top_seq = sys.argv[1]
      side_seq = sys.argv[2]
      gap_penalty = float(sys.argv[3])

   m=len(top_seq)-1
      while m>=0:
           if len(top_seq) != len(side_seq):
             print 'need equal lengths in this example.'
             sys.exit()
          print top_seq[m], ' ', side_seq[m]
          print subst_score(top_seq[m],side_seq[m])
          m-=1
```

B. Create Score and Traceback Matrices. Add the following function for calculating the score matrix to the function blocks in A. Remove the italicized code from main and the

function call to main so that you can print the score matrix for a few sequence pairs: WHAT&WHY, WHEY&WHY, AAAC&AGC, etc. Does it appear to be working right? Note the notation top_seq and side_seq refers to the sequences on the top and side of the alignment matrix.

```
def create_dp_matrices(top_seq,side_seq,gap_penalty):
    m=len(side_seq)
    n=len(top_seq)
    ###
    ### initialize matrices
    score_matrix=zeros([m+1,n+1])
    for i in range(1,m+1):  # intialize first column
        score_matrix[i,0]=score_matrix[i-1,0]+gap_penalty
    for j in range(1,n+1): #initialize first row
        score_matrix[0,j]=score_matrix[0,j-1]+gap_penalty
    # TRACEBACK MATRIX INITIALIZATION HERE
    #
    ### Now recursion
    ###
    for i in range (1, m + 1):
        for j in range (1, n + 1):
            local_scores=[score_matrix[i-1,j]+gap_penalty, \
                    score_matrix[i,j-1]+gap_penalty, \
                    score_matrix[i-1,j-1]+subst_score(side_seq[i-1],top_seq[j-1])]
            # ARGMAX USAGE HERE
            score_matrix[i,j]=max(local_scores)
     return score_matrix
```

Add code to the function above to create the Traceback Matrix as another return value, called trace_matrix. You will initialize the matrix with two for loops and you will use the numpy function argmax, recalling the following from class where A is the Traceback matrix. Now, print the traceback matrix for sample sequence pairs to test your code.

$$S_{m,n} = \max \begin{cases} S_{m-1,n} + gap & : f_{m,n}[0] \\ S_{m,n-1} + gap & : f_{m,n}[1] \\ S_{m-1,n-1} + B(x,y) & : f_{m,n}[2] \end{cases} \qquad A_{m,n} = \arg\max(f)$$

C. Tracing back to construct the alignment. Write a function called trace_back with inputs of top_seq, side_seq, and trace_matrix. The function will start with (m,n)=trace_matrix.shape, top_align=[], and side_align=[], and the following while loop. I added print statements to help you trace what is happening when you run the code.

```
print 'm,n:', m,n
    while m > 1 and n > 1:
        if trace_matrix[m-1,n-1] == 0:
            top_align.append('-')
            side_align.append(side_seq[m-2])
            print 'm,n:', m,n
            print top_align
```

```
            print side_align
            m = m - 1
        if trace_matrix[m-1,n-1] == 1:
            top_align.append(top_seq[n-2])
            side_align.append('-')
            print 'm,n:', m,n
            print top_align
            print side_align
            n = n - 1
        if trace_matrix[m-1,n-1] == 2:
            top_align.append(top_seq[n-2])
            side_align.append(side_seq[m-2])
            print 'm,n:', m,n
            print top_align
            print side_align
            m = m - 1
            n = n - 1
```

Often you also need these while loops to follow the previous one. Why?

```
    while m > 1:
        top_align += top_seq[m]
        side_align += '-'
        print 'm,n:', m,n
        print top_align
        print side_align
        m -= 1

    while n > 1:
        top_align += '-'
        side_align += side_seq[n]
        print 'm,n:', m,n
        print top_align
        print side_align
        n -= 1
```

Finally, you need to add top_align.reverse() and side_align.reverse() and return the aligned sequences. Test the entire sequence alignment process now.

What happens when you set the gap_penalty to -.5 instead of -2 for the pair AAAC, AGC? Which is a better gap penalty? Consider the same question for the following two peptides with penalty -.5 and -3.5.

IQIFSFIFRQEWNDA
QIFFIFRMSVEWND

How could we improve the scoring for a pair of amino acid sequences like the ones above?