Part I. From pairwise sequence alignment to multiple sequence alignment

To demonstrate mastery of sequence alignment, you must provide code screenshots, documentation, and test cases with results for a multiple sequence alignment algorithm that works with 2-5 sequences.

Code

```
from itertools import product
import copy
NUM_SEQ = 5

def subtract_tuples(a, b):
   if len(a) != len(b):
       raise ValueError("Tuples must have the same length for
subtraction.")
   result = tuple(ai - bi for ai, bi in zip(a, b))
   return result


def add_to_tuple(a):
   ones_tuple = (-1,) * len(a)
   return subtract_tuples(a, ones_tuple)


def get_score(*seqs, tup):
   if len(tup) != len(seqs[0]):
       raise ValueError("Tuples must have the same length for score.")
   for i in range(1, len(seqs[0])):
           if seqs[0][i][tup[i]-1] != seqs[0][i-1][tup[i-1]-1]:
               return 0
   return 1


def add_seq(*seqs, my_array, curr, this_tup):
   where_dash = subtract_tuples(curr, this_tup)
   pot_seq = copy.deepcopy(my_array[this_tup][1])
   for i in range(len(where_dash)):
       if where_dash[i] != 1:
           pot_seq[i]+="-"
       else:
           pot_seq[i]+=seqs[0][i][curr[i]-1]
   return pot_seq


def count_dashes(seq1, seq2):
   seq1_dashes = 0
   seq2_dashes = 0
   for i in seq1:
       if i == "-":
           seq1_dashes+=1

   for i in seq2:
       if i == "-":
           seq2_dashes+=1
   if seq1_dashes >= seq2_dashes:
       return 2
   else:
       return 1


def multiple_alignment(*sequences):
   my_array = {}
   this_round = []
   longest = -1
   for seq in sequences:
       if len(seq) > longest:
           longest = len(seq)
   indices = list(product(*[range(len(item) + 1) for item in
sequences]))


   to_fill = list(product(*[range(1, len(item) + 1) for item in
sequences]))
   for i in indices:
       my_array[i] = [0, [""] * len(i)]


   for i in indices:
       if 0 in i:
           localmax = max(i, key = lambda k: k)
           for v in range(len(i)):
               if i[v] == 0:
                   my_array[i][1][v] = "".join("-" for i in
range(localmax))
               else:
                   my_array[i][1][v] = sequences[v][:localmax]

   options = list(product(*[range(2) for _ in sequences]))


   remove = tuple(0 for _ in range(len(sequences)))
   options.remove(remove)


   for i in range(len(to_fill)):
        for opt in options:
            this_round.append(subtract_tuples(to_fill[i], opt))
        my_arr_index = add_to_tuple(this_round[-1])

        this_round_scores = {}
        this_round_scores[this_round[-1]] = [my_array[this_round[-1]]
[0] +
                                             get_score(sequences,
tup=my_arr_index),
                                             add_seq(sequences,
my_array=my_array, curr=my_arr_index, this_tup=this_round[-1])]
        for this_score in this_round[:-1]:
            this_round_scores[this_score] = [my_array[this_score][0],
add_seq(sequences, my_array=my_array, curr=my_arr_index,
this_tup=this_score)]

        max_score = -1
        max_key = None


        for key, value in this_round_scores.items():
            if value[0] > max_score:
                max_score = value[0]
                max_key = key
            elif value[0] == max_score:
                if longest > NUM_SEQ:
                   max_key = key
        my_array[my_arr_index] = this_round_scores[max_key]
        this_round = []
    return my_array[my_arr_index][0], my_array[my_arr_index][1]
```

Explanation

The code above is a multiple sequence alignment algorithm that, given 2-5 DNA sequences, it will return the alignment score and aligned sequences.

The algorithm leverages the itertools python library, as well as a number of helper functions, named: subtract_tuples, add_to_tuple, get_score, add_seq, and count_dashes.

But, how does it work?

The alignment algorithm begins by creating a 1D dictionary ("my_array") that represents an nD array (given that n is the number of sequences passed to the function). It does so by generating the indices for this nD array ("indices"), using the itertools library to create a cartesian product of the lengths of each inputted sequence + 1 (to account for the "-"). This "indices" list is the keys for the 1D dictionary. Then, the nD array values are a list pair of score and alignment sequences; where a sequence is aligned to dashes ("-") the alignment scores are initialized to 0 and the alignments for these cases are initialized as well.

The algorithm uses itertools to create a list of indices that need to be filled in the matrix ("to_fill"); this excludes the already initialized values that are aligned with dash sequences (for example, "-", "--", "---"...). Another list is also created that includes the list of options for the first empty index in the array ("options"), depending on the number of inputs given (for example, if the first empty index for a 2 sequence alignment is (1,1) then the list of options are (0,0), (1,0), (0,1)).

Next, the algorithm iterates through the list of indices that need to be filled ("to_fill"). It uses the "options" list and the subtract_tuples helper function to generate a list of options for that particular index that needs to be filled. It generates the scores for each option using the option, the get_score helper function, and the add_seq helper function, places them in a dictionary, with their respective potential new alignment. Lastly, it iterates through this dictionary of scores, and chooses the highest score for that particular index and places it into that index in the nD array ("my_array"). It continues preforming this algorithm until the entire array is filled with scores and alignments.

Finally, it returns the value for the last spot in the nD array, which includes the scores and the aligned sequences for the respective inputs.

Testing

My initial testing was simply running the algorithm on a variety of different inputs and seeing if the scores and alignments retrieved were expected. Some of the inputs I used are below:

Proof it works

Homologous Sequences. This should have a score of 3 since all 3 match.

```
multiple_alignment("ACG", "ACG", "ACG")

(3, ['ACG', 'ACG', 'ACG'])
```

Sequences with a mutation. This should be a score of 2, since the score above was a 3, and now one nucleotide is different.

```
multiple_alignment("ACG", "ATG", "AGG")

(2, ['ACG', 'ATG', 'AGG'])
```

Sequences with different lengths (should add a dash in the shorter sequences)

```
multiple_alignment("ACGG", "ATG", "AGG")

(2, ['ACGG', 'A-TG', 'A-GG'])
```

A long sequence and short ones. This should add dashes in the shorter sequences

```
multiple_alignment("ACGG", "A", "A")
```

```
(1, ['ACGG', 'A---', 'A---'])
```

3 long homologous sequences. Should be a score of 12 because the sequences are 12 nucleotides long.

```
multiple_alignment("ACTGACTGACTG", "ACTGACTGACTG", "ACTGACTGACTG")
```

```
(12, ['ACTGACTGACTG', 'ACTGACTGACTG', 'ACTGACTGACTG'])
```

Long differing sequences

```
multiple_alignment("ATCGATCGATCGATCGATCG", "ATGGGCTCGTAGGGCTCGTA",
"ATGTA")
```

```
(5,
 ['ATCG--AT-CGATCGATCGATCG',
  'AT-GGGCTCGTA--GGGCTCGTA',
  'AT-G---T---A-----------'])
```

5 inputs, same length with matching two first nucleotides. Should have a score of 2.

```
multiple_alignment("ATC", "ATG", "ATA", "ATG", "ATC")
```

```
(2, ['ATC', 'ATG', 'ATA', 'ATG', 'ATC'])
```

5 inputs, differing lengths. Should add dashes in the shorter sequences.

```
multiple_alignment("ATGGGG", "ATGGG", "AT", "ATGGG", "ATGGG")
```

```
(2, ['ATGGGG', 'AT-GGG', 'AT----', 'AT-GGG', 'AT-GGG'])
```

2 sequences, one nucleotide difference. Should have a score of 6 because 6 matching nucleotides and 1 differing one.

```
multiple_alignment("ATGATGA", "ATGATGC")
```

```
(6, ['ATGATG-A', 'ATGATGC-'])
```

Part II. From pairwise sequence alignment to multiple sequence alignment

Now extend your pairwise alignment algorithm to support local alignment. Modify your lab algorithm into the smith-waterman algorithm for local sequence alignment.

Local Alignment Code

```
import pandas as pd
import numpy as np

def local_align(s1,s2):
    scores = pd.DataFrame(index=["-"]+[s1[:i+1] for i in
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    aligned = pd.DataFrame(index=["-"]+[s1[:i+1] for i in
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    pointers = pd.DataFrame(index=["-"]+[s1[:i+1] for i in
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    for s2_part in scores.columns:
        scores.loc["-",s2_part] = 0
        pointers.loc["-",s2_part] = None
        if s2_part == "-":
            aligned.loc["-","-"] = ("","")
        else:
            aligned.loc["-",s2_part] = ("".join(["-" for i in
range(len(s2_part))]),s2_part)
    for s1_part in scores.index:
        scores.loc[s1_part,"-"] = 0
        pointers.loc[s1_part,"-"] = None
        if s1_part == "-":
            aligned.loc["-","-"] = ("","")
        else:
            aligned.loc[s1_part,"-"] = (s1_part,"".join(["-" for i in
range(len(s1_part))]))
    mismatch_score = -3
    match_score = 5
    gap_score = -4
    best_score = -1
    best_spot = (-1, -1)

    nrows,ncols = scores.shape
    for i in range(1,nrows):
        for j in range(1,ncols):

            opt1_s1 = scores.index[i-1]
            opt1_s2 = scores.columns[j-1]

            matching = int(s1[i-1] == s2[j-1])

            score_opt1 = scores.loc[opt1_s1, opt1_s2] +  (match_score
if matching else mismatch_score)

            s1_aligned_opt1 = aligned.loc[opt1_s1, opt1_s2][0]+s1[i-1]

            s2_aligned_opt1 = aligned.loc[opt1_s1, opt1_s2][1]+s2[j-1]
            opt2_s1 = scores.index[i-1]
            opt2_s2 = scores.columns[j]
            score_opt2 = scores.loc[opt2_s1, opt2_s2] + gap_score
            s1_aligned_opt2 = aligned.loc[opt2_s1, opt2_s2][0]+s1[i-1]
            s2_aligned_opt2 = aligned.loc[opt2_s1, opt2_s2][1]+"-"

            opt3_s1 = scores.index[i]
            opt3_s2 = scores.columns[j-1]
            score_opt3 = scores.loc[opt3_s1, opt3_s2] + gap_score
            s1_aligned_opt3 = aligned.loc[opt3_s1, opt3_s2][0]+"-"
            s2_aligned_opt3 = aligned.loc[opt3_s1, opt3_s2][1]+s2[j-1]

            max_score = max(score_opt1,score_opt2,score_opt3)

            if max_score > best_score:
                best_score = max_score
                best_spot = i, j
            scores.loc[scores.index[i],scores.columns[j]] = (max_score
if max_score > 0 else 0)
            if max(score_opt1,score_opt2,score_opt3) == score_opt1:
                aligned.loc[scores.index[i],scores.columns[j]] =
(s1_aligned_opt1,s2_aligned_opt1)
                pointers.loc[scores.index[i],scores.columns[j]] = (i-
1, j-1)
            elif max(score_opt1,score_opt2,score_opt3) == score_opt2:
                aligned.loc[scores.index[i],scores.columns[j]] =
(s1_aligned_opt2,s2_aligned_opt2)
                pointers.loc[scores.index[i],scores.columns[j]] = (i-
1, j)
            elif max(score_opt1,score_opt2,score_opt3) == score_opt3:
                aligned.loc[scores.index[i],scores.columns[j]] =
(s1_aligned_opt3,s2_aligned_opt3)
                pointers.loc[scores.index[i],scores.columns[j]] = (i,
j-1)

    curr = best_spot

    best_align1 = (aligned.loc[scores.index[curr[0]],
scores.columns[curr[1]]])[0]
    best_align2 = (aligned.loc[scores.index[curr[0]],
scores.columns[curr[1]]])[1]

    while(scores.loc[scores.index[curr[0]],scores.columns[curr[1]]] !=
0):
        curr =
pointers.loc[scores.index[curr[0]],scores.columns[curr[1]]]
    cutoff_1 = (aligned.loc[scores.index[curr[0]],
scores.columns[curr[1]]])[0]
    cutoff_2 = (aligned.loc[scores.index[curr[0]],
scores.columns[curr[1]]])[1]

    if best_align1.startswith(cutoff_1):
        best_align1 = best_align1[len(cutoff_1):]
    if best_align2.startswith(cutoff_2):
        best_align2 = best_align2[len(cutoff_2):]
    return best_score, best_align1, best_align2
```

Explanation

This code maintained the essence of the pairwise alignment algorithm with a few changes. As it fills up the scoring and aligned matrix, it keeps track of the index with the highest score ("best_spot"). In addition, I included the addition of another matrix ("pointers"), to track the path for each index (the index associated with the option chosen for each respective index in the aligned and scoring matrices). Once the entire 2D matrix is filled with scores and alignments for the two input sequences, the algorithm fetches the highest score alignment index ("best_spot"). Then, it backtracks the pointers matrix until the score equals 0. The algorithm fetches this alignment (the prefix of the local alignments), and removes this prefix. It then returns these alignments.

Testing

I used the textbook example to first see if my algorithm works.

```
local_align("CGTAGGCTTAAGGTTA", "ATAGATA")
```

```
(15, 'TAG', 'TAG')
```

I also used the example from the Smith-Waterman link, and received the correct alignment here as well.

```
local_align("CGTGAATTCAT", "GACTTAC")
```

```
(18, 'GAATT-C', 'GACTTAC')
```

Using these test cases I could deduce the algorithm was working as expected.

This is because once the scoring matrix is filled, I needed to backtrack from the best spot. This best spot in this case was 18 at location C,C. From there, my algorithm moved back to 13, which was T,A. It then chose to backtrack upwards, which had the best option, which was T,T. Then, T to T again at score 12. Backtracked to A to C at score 7, A to A at score 10, and lastly G to G at score 5. Once it reached 5, the only other scores were 0, which is the case for when the local alignment has been fully realized. At which case, my algorithm returns the alignments "GAATT-C" and "GACTTAC", with a total score of 18, the highest score in the matrix.

PAM Local Alignment Code

You must also perform both nucleotide sequence alignment and protein sequence alignment using the PAM scoring matrices.

```python
def local_align_PAM(s1, s2, pam_filename):

    pam = make_df(pam_filename)
    print(pam)
    scores = pd.DataFrame(index=["-"]+[s1[:i+1] for i in 
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    aligned = pd.DataFrame(index=["-"]+[s1[:i+1] for i in 
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    pointers = pd.DataFrame(index=["-"]+[s1[:i+1] for i in 
range(len(s1))],columns=["-"]+[s2[:i+1] for i in range(len(s2))])
    for s2_part in scores.columns:
        scores.loc["-",s2_part] = 0
        pointers.loc["-",s2_part] = None
        if s2_part == "-":
            aligned.loc["-","-"] = ("","")
        else:
            aligned.loc["-",s2_part] = ("".join(["-" for i in 
range(len(s2_part))]),s2_part)
    for s1_part in scores.index:
        scores.loc[s1_part,"-"] = 0
        pointers.loc[s1_part,"-"] = None
        if s1_part == "-":
            aligned.loc["-","-"] = ("","")
        else:
            aligned.loc[s1_part,"-"] = (s1_part,"".join(["-" for i in 
range(len(s1_part))]))

    best_score = -1
    best_spot = (-1, -1)

    nrows,ncols = scores.shape
    for i in range(1,nrows):
        for j in range(1,ncols):
            print(i, j)
            opt1_s1 = scores.index[i-1]
            opt1_s2 = scores.columns[j-1]
            score_opt1 = scores.loc[opt1_s1, opt1_s2] + 
int(pam.loc[s1[i-1], s2[j-1]])

            s1_aligned_opt1 = aligned.loc[opt1_s1, opt1_s2][0]+s1[i-1]

            s2_aligned_opt1 = aligned.loc[opt1_s1, opt1_s2][1]+s2[j-1]
            opt2_s1 = scores.index[i-1]
            opt2_s2 = scores.columns[j]
            score_opt2 = scores.loc[opt2_s1, opt2_s2] + 
int(pam.loc[s1[i-1], "-"])
            s1_aligned_opt2 = aligned.loc[opt2_s1, opt2_s2][0]+s1[i-1]
            s2_aligned_opt2 = aligned.loc[opt2_s1, opt2_s2][1]+"-"

            opt3_s1 = scores.index[i]
            opt3_s2 = scores.columns[j-1]
            score_opt3 = scores.loc[opt3_s1, opt3_s2] + 
int(pam.loc["-", s2[j-1]])
            s1_aligned_opt3 = aligned.loc[opt3_s1, opt3_s2][0]+"-"
            s2_aligned_opt3 = aligned.loc[opt3_s1, opt3_s2][1]+s2[j-1]

            max_score = max(score_opt1,score_opt2,score_opt3)

            if max_score > best_score:
                best_score = max_score
                best_spot = i, j
            scores.loc[scores.index[i],scores.columns[j]] = (max_score 
if max_score > 0 else 0)
            if max(score_opt1,score_opt2,score_opt3) == score_opt1:
                aligned.loc[scores.index[i],scores.columns[j]] = 
(s1_aligned_opt1,s2_aligned_opt1)
                pointers.loc[scores.index[i],scores.columns[j]] = (i-
1, j-1)
            elif max(score_opt1,score_opt2,score_opt3) == score_opt2:
                aligned.loc[scores.index[i],scores.columns[j]] = 
(s1_aligned_opt2,s2_aligned_opt2)
                pointers.loc[scores.index[i],scores.columns[j]] = (i-
1, j)
            else:
                aligned.loc[scores.index[i],scores.columns[j]] = 
(s1_aligned_opt3,s2_aligned_opt3)
                pointers.loc[scores.index[i],scores.columns[j]] = (i, 
j-1)
    curr = best_spot
    best_align1 = (aligned.loc[scores.index[curr[0]], 
scores.columns[curr[1]]])[0]
    best_align2 = (aligned.loc[scores.index[curr[0]], 
scores.columns[curr[1]]])[1]

    while(scores.loc[scores.index[curr[0]],scores.columns[curr[1]]] != 
0):
        curr = 
pointers.loc[scores.index[curr[0]],scores.columns[curr[1]]]
    cutoff_1 = (aligned.loc[scores.index[curr[0]], 
scores.columns[curr[1]]])[0]
    cutoff_2 = (aligned.loc[scores.index[curr[0]], 
scores.columns[curr[1]]])[1]
```

```
    if best_align1.startswith(cutoff_1):
        best_align1 = best_align1[len(cutoff_1):]
    if best_align2.startswith(cutoff_2):
        best_align2 = best_align2[len(cutoff_2):]
    return best_score, best_align1, best_align2


def make_df(filename):
    with open(filename, 'r') as file:
        data = [line.split() for line in file.readlines()]

    row_names = [row[0] for row in data]
    row_names = row_names[1:]
    print(row_names)

    data = data[1:]
    df = pd.DataFrame([row[1:] for row in data], columns=row_names)
    df.index = row_names
    return df
```

Explanation

The code above is similar to the implementation of local alignment with preassigned scores. The one distinct difference is the PAM local alignment algorithm takes in a text file argument that contains a PAM matrix of scores. I implemented a make_df function that takes this text fie of scores and creates a pandas matrix called pam to derive the scores from. An example of such a file is shown below:

```
 A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X
-
A  2 -2  0  0 -2  0  0  1 -1 -1 -2 -1 -1 -3  1  1  1 -6 -3  0  0  0  0
-8
R -2  6  0 -1 -4  1 -1 -3  2 -2 -3  3  0 -4  0  0 -1  2 -4 -2 -1  0 -1
-8
N  0  0  2  2 -4  1  1  0  2 -2 -3  1 -2 -3  0  1  0 -4 -2 -2  2  1  0
-8
D  0 -1  2  4 -5  2  3  1  1 -2 -4  0 -3 -6 -1  0  0 -7 -4 -2  3  3 -1
-8
C -2 -4 -4 -5 12 -5 -5 -3 -3 -2 -6 -5 -5 -4 -3  0 -2 -8  0 -2 -4 -5 -3
-8
Q  0  1  1  2 -5  4  2 -1  3 -2 -2  1 -1 -5  0 -1 -1 -5 -4 -2  1  3 -1
-8
E  0 -1  1  3 -5  2  4  0  1 -2 -3  0 -2 -5 -1  0  0 -7 -4 -2  3  3 -1
-8
G  1 -3  0  1 -3 -1  0  5 -2 -3 -4 -2 -3 -5  0  1  0 -7 -5 -1  0  0 -1
-8
H -1  2  2  1 -3  3  1 -2  6 -2 -2  0 -2 -2  0 -1 -1 -3  0 -2  1  2 -1
```

```
-8
I -1 -2 -2 -2 -2 -2 -2 -3 -2  5  2 -2  2  1 -2 -1  0 -5 -1  4 -2 -2 -1
-8
L -2 -3 -3 -4 -6 -2 -3 -4 -2  2  6 -3  4  2 -3 -3 -2 -2 -1  2 -3 -3 -1
-8
K -1  3  1  0 -5  1  0 -2  0 -2 -3  5  0 -5 -1  0  0 -3 -4 -2  1  0 -1
-8
M -1  0 -2 -3 -5 -1 -2 -3 -2  2  4  0  6  0 -2 -2 -1 -4 -2  2 -2 -2 -1
-8
F -3 -4 -3 -6 -4 -5 -5 -5 -2  1  2 -5  0  9 -5 -3 -3  0  7 -1 -4 -5 -2
-8
P  1  0  0 -1 -3  0 -1  0  0 -2 -3 -1 -2 -5  6  1  0 -6 -5 -1 -1  0 -1
-8
S  1  0  1  0  0 -1  0  1 -1 -1 -3  0 -2 -3  1  2  1 -2 -3 -1  0  0  0
-8
T  1 -1  0  0 -2 -1  0  0 -1  0 -2  0  -1 -3  0  1  3 -5 -3  0  0 -1
0 -8
W -6  2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4  0 -6 -2 -5 17  0 -6 -5 -6 -4
-8
Y -3 -4 -2 -4  0 -4 -4 -5  0 -1 -1 -4 -2  7 -5 -3 -3  0 10 -2 -3 -4 -2
-8
V  0 -2 -2 -2 -2 -2 -2 -1 -2  4  2 -2  2 -1 -1 -1  0 -6 -2  4 -2 -2 -1
-8
B  0 -1  2  3 -4  1  3  0  1 -2 -3  1 -2 -4 -1  0  0 -5 -3 -2  3  2 -1
-8
Z  0  0  1  3 -5  3  3  0  2 -2 -3  0 -2 -5  0  0 -1 -6 -4 -2  2  3 -1
-8
X  0 -1  0 -1 -3 -1 -1 -1 -1 -1 -1 -1 -1 -2 -1  0  0 -4 -2 -1 -1 -1 -1
-8
- -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8
1
```

Therefore, the text file is read in and the code generates a pandas dataframe containing the data from the file. When determining the score for each option, the score is calculated by finding the score for those letter combos using the matrix.

note: code cannot be run in colab with the text file so I am supplying the results when run on my local machine.

I know my algorithm is correct because when I passed in, for instance, local_align_PAM("AT", "AT", "PAM.txt") I get a score 5 for local align. This is correct because A to A returns a match score of 2 and T to T returns a match score of 3.

Then, when I passed in local_align_PAM("ATC", "ATG", "PAM.txt"), I still got a 5. This is because C and G returns a -3, so the algorithm prioritizes the alignment of "AT" to "AT".

When I make my test case even larger, for instance, local_align_PAM("CGTGAATTCAT", "GACTTAC", "PAM.txt") I receive a score of 17, with the alignments 'ATTC', 'TTAC'. I can validate this answer because as I move through the matrix I can calculate the matrix like the following:

- A to A aligns to 1

- T to T aligns to 3
- T to A aligns to 1
- C to C aligns to 12
- Therefore, I know the alignment is 17. This is greater than the global alignment of the entire DNA sequences.

Another example is when I ran "CCCR", and "CCCC". I know by hand calculation, the local alignment should be "CCC" to "CCC" with a score of 36 (C to C is a match score of 12). My algorithm returned this, and removed the last letters from the sequences, due to them lowering the overall score because C to R is a -4 mismatch score.