School of Biological Sciences

Bioinformatics Algorithms
ICA Report

Author: B236494

Tuesday 2nd April, 2024

Introduction

The Smith-Waterman algorithm is used to determine regions of sequence similarity when comparing protein sequences or nucleic acid sequences by performing a local alignment. A version of this algorithm has been coded into a Python script and provided. Instructions were given to add functionality to this code, keeping the EMBOSS suite's water program as the "gold-standard".

This report will be focused on the changes made in the SmithWaterman.py (Version 1.9 SRT) file that was provided to improve functionality and robustness. The code has been modified with the implementation of 4 key elements:

1. Allows the parsing of certain parameters that will be used to generate the alignment.
2. Checking input file is it is suitable for alignment.
3. Inclusion of print statements at multiple points in the code.
4. A test at the end which compares the output of the script to a "gold-standard".

The changes that were made to achieve these implementations will be described along with how it has aided in assessing if the code is being executed appropriately.

Please Note:

- The user is able to choose the pause (in seconds) between important steps to read what is being executed or input 0 and skip all pauses.
- The modified script file (B236494_SmithWaterman_Final_Script.py) has been provided in the Supplementary Material.
- A pdf has been appended with this report (Modifications_Made_To_SmithWaterman.py_V1.9.pdf) which shows the changes made in the program and a clear comparison can be seen with the base code that was provided. All line numbers mentioned moving forward will be quoted from the same.
- The -datafile parameter used (EDNAFULL_srt) in the EMBOSS water program execution was provided with the base script file.

Parameter Parsing for Alignment Generation

This was done using the argparse module. It enables the user to provide parameters for the match/mismatch/gap scores on the CLI (command line interface) to be used during the alignment. Default values have also been assigned so that the user does not have to provide the parameters if they do not wish to do so. These values will also be passed to EMBOSS water at the end so that meaningful comparison can be done between the two

This functionality has been implemented in lines 335 - 349 of the modified script file.

Verifying Input File Suitability for Alignment

Input files were vetted to ensure that only A/T/G/C characters were present in the files chosen for selection. Note that the user can provide the filename, provided it is in the current working directory, and the programme reads the filenames, stores the sequences and then checks for non-ATGC characters. Execution proceeds as normal if there are none or the script is terminated and the user is told to provide the right sequence files.

This functionality has been implemented in lines 361 - 422 of the modified script file.

Integrating Print Statements Throughout the Code

The script was developed using many print statements to gauge the output at various stages of execution. Most of these have been commented out as there is no benefit to the user in knowing these results. The informative print statements have been displayed with an optional pause functionality present before each, if the user wishes to use it.

Some examples of this functionality can be seen in lines 62, 70, 215 and 282.

Final Test Comparing Script Output to Gold Standard

It is good practice when testing a variation of a particular algorithm/function to have a “gold-standard” for comparison. As mentioned earlier, EMBOSS’s water is being used for that purpose here. The output of the script has been inspired from that of water and the output of water is used to validate that the script is working properly. After the script has given us the alignment that it has calculated, water is run using os.system() step and displayed to the user. The output of both is quite similar and this tells us that the script is working properly.

This functionality has been implemented in lines 445 - 460 of the modified script file.

Conclusion

All these implementations ensure that the code is being executed in the expected manner and also give the user room for flexibility in both altering the parameters and in reusing the code conveniently to include other functionalities or grasp the flow of execution. This script can be improved by additional test functions for comparison, not printing out certain outputs or including additional print statements and further error-trapping the inputs that it takes. The changes that have been made in the code can be visualised clearly in the next few pages.

Please view the supplementary Material for a full picture of the modifications made. In particular, do note the comparison view mode (after the code) as mentioned earlier.

Supplementary Material

The modified script file adapted from the base script file: B236494_SmithWaterman_Final_Script.py

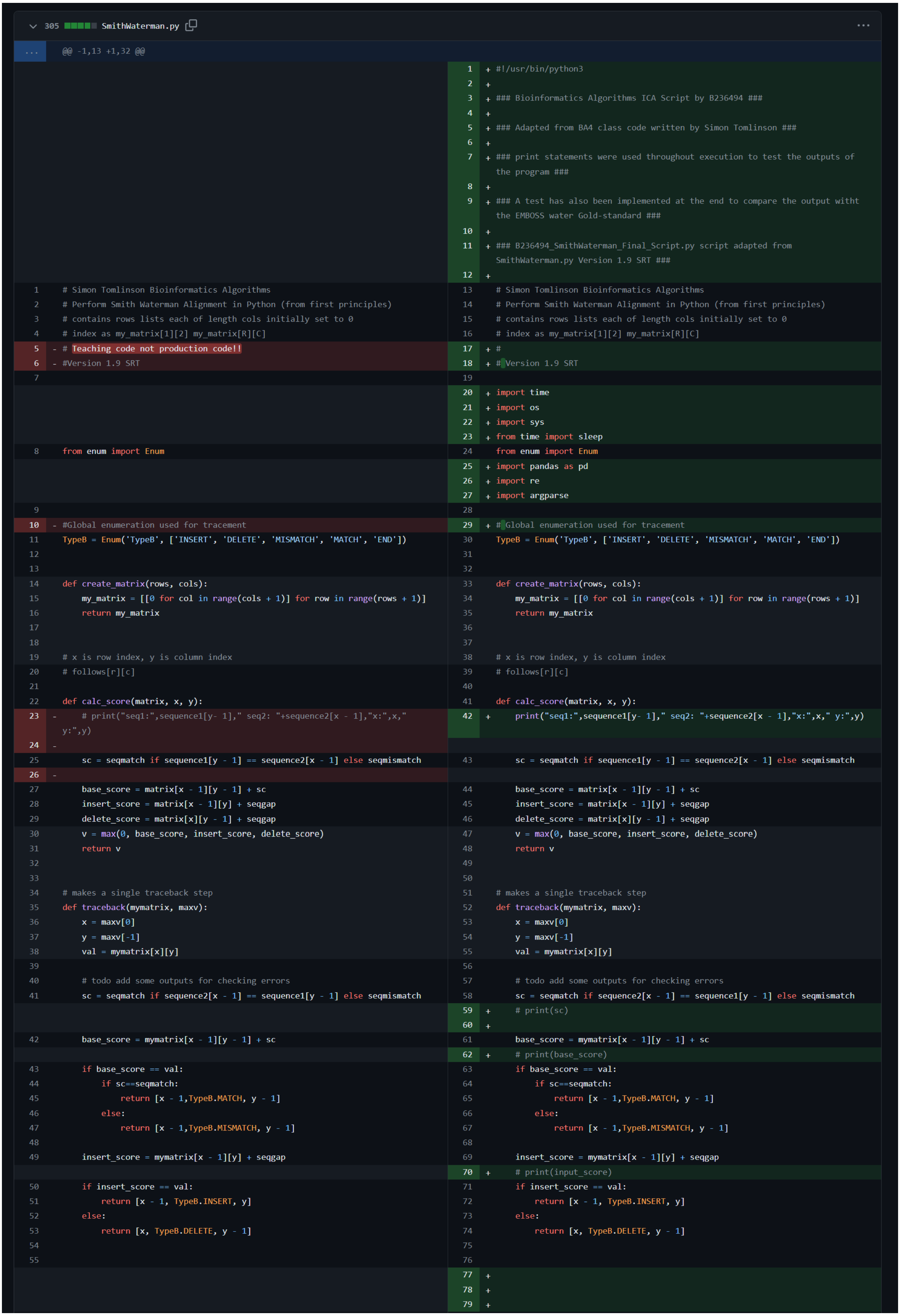
```
#!/usr/bin/python3

### Bioinformatics Algorithms ICA Script by B236494 ###

### Adapted from BA4 class code written by Simon Tomlinson ###

### print statements were used throughout execution to test the outputs of the program ###

### A test has also been implemented at the end to compare the output witht the EMBOSS water Gold-standard ###

### B236494_SmithWaterman_Final_Script.py script adapted from SmithWaterman.py Version 1.9 SRT ###

# Simon Tomlinson Bioinformatics Algorithms
# Perform Smith Waterman Alignment in Python (from first principles)
# contains rows lists each of length cols initially set to 0
# index as my_matrix[1][2] my_matrix[R][C]
#
# Version 1.9 SRT

import time
import os
import sys
from time import sleep
from enum import Enum
import pandas as pd
import re
import argparse

# Global enumeration used for tracement
TypeB = Enum('TypeB', ['INSERT', 'DELETE', 'MISMATCH', 'MATCH', 'END'])


def create_matrix(rows, cols):
    my_matrix = [[0 for col in range(cols + 1)] for row in range(rows + 1)]
    return my_matrix

# x is row index, y is column index
# follows[r][c]
def calc_score(matrix, x, y):
    print("seq1:",sequence1[y- 1]," seq2: "+sequence2[x - 1],"x:",x," y:",y)
    sc = seqmatch if sequence1[y - 1] == sequence2[x - 1] else seqmismatch
    base_score = matrix[x - 1][y - 1] + sc
    insert_score = matrix[x - 1][y] + seqgap
    delete_score = matrix[x][y - 1] + seqgap
    v = max(0, base_score, insert_score, delete_score)
    return v


# makes a single traceback step
def traceback(mymatrix, maxv):
    x = maxv[0]
    y = maxv[-1]
    val = mymatrix[x][y]

    # todo add some outputs for checking errors
    sc = seqmatch if sequence2[x - 1] == sequence1[y - 1] else seqmismatch
    # print(sc)

    base_score = mymatrix[x - 1][y - 1] + sc
    # print(base_score)
    if base_score == val:
        if sc==seqmatch:
            return [x - 1,TypeB.MATCH, y - 1]
        else:
            return [x - 1,TypeB.MISMATCH, y - 1]

    insert_score = mymatrix[x - 1][y] + seqgap
    # print(input_score)
    if insert_score == val:
        return [x - 1, TypeB.INSERT, y]
    else:
        return [x, TypeB.DELETE, y - 1]
```

```python
# builds the initial scoring matrix used for traceback
def build_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    row_number=0

    for i in range(1, rows):
        row_number = row_number + 1
        print("\nRow Number:", row_number)
        sleep(wait)
        for j in range(1, cols):
            mymatrix[i][j] = calc_score(mymatrix, i, j)

    return mymatrix


# gets the max value from the built matrix
# Max is the end of the traceback for SW
def get_max(mymatrix):
    max = mymatrix[0][0]
    mrow = 0
    mcol = 0

    rows = len(mymatrix)
    cols = len(mymatrix[0])

    for i in range(1, rows):
        for j in range(1, cols):
            if mymatrix[i][j] > max:
                max = mymatrix[i][j]
                mrow = i
                mcol = j
    print("The Maximum Score was: ", max,"\n")
    return [mrow, TypeB.END, mcol]


# print out the best scoring path from the SW matrix
def print_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    s1 = "  " + sequence1
    s2 = " " + sequence2

    sleep(wait)

    print("\nDimensions of The SmithWaterman Matrix: Rows= %2d , Columns= %2d\n" % (rows, cols))

    sleep(wait)

    for a in s1:
        print(a, end="")
        print(" \t", end="")
    print("\n", end="")

    for i in range(0, rows):
        print(s2[i], end="")
        print(" \t", end="")
        for j in range(0, cols):
            print("%02d\t" % (mymatrix[i][j]), end="")
        print("\n", end="")



# print out the traceback of the best scoring alignment
def print_traceback(mymatrix):
    # this will print as expected with internal gaps

    sleep(wait)

    print("\n### We Will Now Build The Traceback... ###\n")
    maxv = get_max(mymatrix)
    print(maxv)

    # stash the max score for later
```

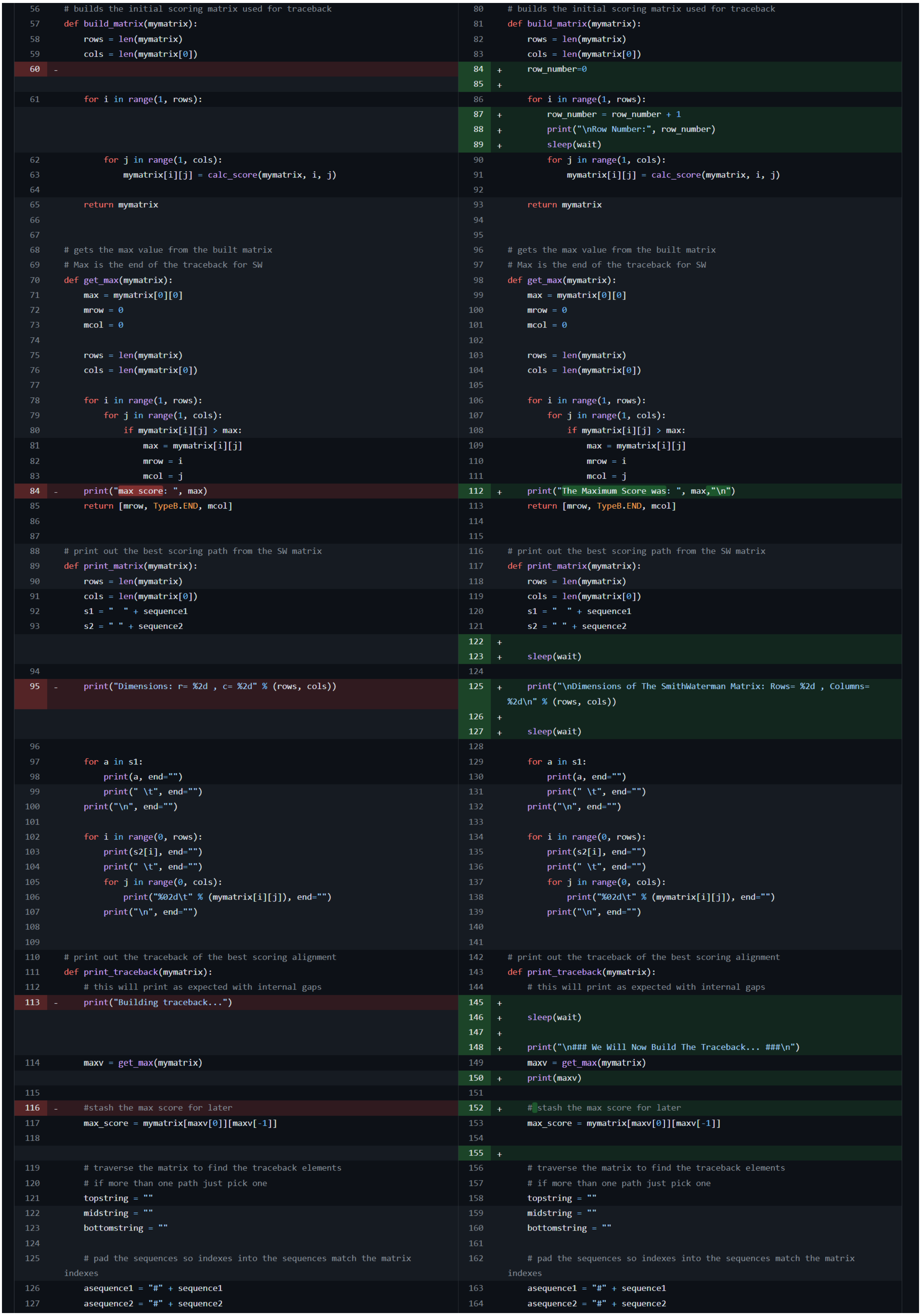
```
max_score = mymatrix[maxv[0]][maxv[-1]]


# traverse the matrix to find the traceback elements
# if more than one path just pick one
topstring = ""
midstring = ""
bottomstring = ""

# pad the sequences so indexes into the sequences match the matrix indexes
asequence1 = "#" + sequence1
asequence2 = "#" + sequence2

# this vector is used to store the traceback results

traversal_results = []

# add the rest of the elements
search = True
lastelement = False

# Stores the position so it can track if it is an insertion or deletion
# Check if it is a gap or not
if max_score <1:
    print ("There is no suitable alignment...Check your inputs please!")
    exit;
old_maxv = maxv


while (search):

    # print(" position:  %d, %d " % (maxv[0], maxv[-1]))

    # print("Testing execution, type is", current_type)

    # store the results
    traversal_results.append(maxv)

    # type_traversal = type(traversal_results)

    maxv = traceback(mymatrix, maxv)


    # catch the trivial case that we are at the end of one of the sequences
    if (maxv[-1] < 0 or maxv[0] < 0):
        traversal_results.append(maxv)
        search= False
        continue


    if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):
        lastelement = True
        continue

    if(lastelement==True) :
        search= False
        traversal_results.append(maxv)
        continue


for i in range(0, len(traversal_results)-2):

    # print("Testing execution")

    # The TypeB of the next element gives how the current element was reached
    # in the dynamic programming table
    # The current element gives the index of the two matching bases to be aligned


    curr_el=traversal_results[i]
    next_el=traversal_results[i+1]
```

```
        # Match
        if(next_el[1]==TypeB.MATCH):
            bottomstring += asequence2[curr_el[0]]
            topstring += asequence1[curr_el[-1]]
            midstring +="|"
            # print(" position: ", i,i )
            # print("MATCHHHHH")


        #Mismatch
        elif(next_el[1]==TypeB.MISMATCH):
            bottomstring += asequence2[curr_el[0]]
            topstring += asequence1[curr_el[-1]]
            midstring += "."
            # print(" position: ", i,i )
            # print("MISMATCH/START?")

        elif(next_el[1]==TypeB.INSERT):
            bottomstring += asequence2[curr_el[0]]
            topstring += "-"
            midstring += " "
            # print(" position: ", i,i )
            # print("Insertion")

        elif(next_el[1]==TypeB.DELETE):
            bottomstring += "-"
            topstring += asequence1[curr_el[-1]]
            midstring += " "
            # print(" position: ", i,i )
            # print("DELETED")


    print("")
    for element in traversal_results:
        print(element,"\n")

    sleep(wait)

    print("\nFinal Alignment, Score: %d\n" % max_score)

    sleep(wait)


    ### Printing the alignment with an effect ###

    # print(topstring[::-1])
    animated_print(topstring[::-1])

    # print(midstring[::-1])
    animated_print(midstring[::-1])

    # print(bottomstring[::-1])
    animated_print(bottomstring[::-1])


    sleep(wait)

    # print("Testing execution")


def time_to_pause():
    print("How long do you want the pauses in between important steps to be in seconds?")
    print("Please input a positive integer value <= 3 OR input 0 to execute without pauses")
    while True:
        wait_input = input("What is you choice? \n")
        if not wait_input.isdigit():
            print("TypeError: Please input a numerical value")
            continue;
        wait = int(wait_input)
        if wait < 0 or wait > 3:
            print("Invalid choice. Please choose a positive integer value <= 3")
        else :
            print("You have chosen a value of ", wait," seconds\n")
            return wait
            break;
```

```python
def animated_print(s):
    for c in s:
        sys.stdout.write(c)
        sys.stdout.flush()
        time.sleep(0.25)
    print("")


# build the SW alignment...
def perform_smith_waterman():
    # values for weights
    global seqmatch
    global seqmismatch
    global seqgap
    global sequence1
    global sequence2
    global wait

    print("\n\n\nWelcome to B236494's version of the SmithWaterman.py Script \n")
    print("\nThis is an adapted version of SmithWaterman.py V 1.9 by Simon Tomlinson\n")
    print("\nThis script takes command line inputs for the match/mismatch/gap scores and user-input during the start of execcution for the FASTA files to be aligned\n")
    print("\nEnsure that the files to be aligned are present in the current directory where the programme is being executed \n")
    print("\nExample CLI input could be ---> python3 SmithWaterman.py --seqmatch 1 --seqmismatch -1 --seqgap -1 \n")



    wait = time_to_pause()


    ### Taking command line input for the match, mismatch and gap penalties ###

    parser = argparse.ArgumentParser(description='Please provide the parameters to Perform Smith-Waterman Alignment.')
    sleep(wait)

    # Add arguments for sequence match, mismatch, and gap penalties
    # note these defaults are not the exact weights used in the original SW paper

    parser.add_argument('--seqmatch', type=int, default=1, help='Input the score for sequence matches. Default is 1.')
    parser.add_argument('--seqmismatch', type=int, default=-1, help='Input the Penalty for sequence mismatches. Default is -1.')
    parser.add_argument('--seqgap', type=int, default=-1, help='Input the penalty for a gap. Default is -1.')

    args = parser.parse_args()

    seqmatch = args.seqmatch
    seqmismatch = args.seqmismatch
    seqgap = args.seqgap

    sleep(wait)

    print(f"Using match score: {seqmatch}")
    print(f"Using mismatch penalty: {seqmismatch}")
    print(f"Using gap penalty: {seqgap}\n")

    sleep(wait)

    ### Function to check if there are any non ATGC characters ###

    def check_if_any_non_atgc_chars(sequence):
        pattern = re.compile(r'[^ATGC]')
        match = pattern.search(sequence.upper())
        return not bool(match)
    ### Function to store sequences that need to be aligned ###

    def read_fasta_filename(filename):
        seq = ""
        with open(filename, 'r') as filehandle:
            for line in filehandle:
                # Using regular expression search to ignore lines starting with ">"
                if re.search("^>", line.strip()):
                    continue
                seq += line.strip()
        return seq


    # Taking input for file name

    seq1_file = input("Enter the filename of the first sequence file: ")
    seq2_file = input("Enter the filename to the second sequence file: ")

    # Saving sequences to variables to be aligned

    sequence1 = read_fasta_filename(seq1_file)
    sequence2 = read_fasta_filename(seq2_file)



    ### Alternate options to test sequences ###

#    sequence1="AGTGATAAACTAGTAATTTTT"
#    sequence2="TTGGGGGTAAACAGGGG"

#    sequence1 ="AGTCGGTTAGTAAA"
#    sequence2 ="TTTTGGGTTTAGGCGC"

#    sequence1 = "GTGTATTTTTTT"
#    sequence2 = "AAAAGTGTTATT"

#    sequence1 = "TCGTTCTAG"
#    sequence2 = "TCGTTTTTG"

#    sequence1 = "SimonTomkinson"
#    sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"



    ### Checking sequence1 & sequence2 if they contain any characters other than A, T, G, C ###

    check_seq1 = check_if_any_non_atgc_chars(sequence1)
    check_seq2 = check_if_any_non_atgc_chars(sequence2)

    sleep(wait)

    if not check_seq1 or not check_seq2:
        print("Error in One or both input sequences. There are characters other than A, T, G, and C.")
        print("Please make sure that your input files contain the correct sequences and try again.")
        print("Exiting Program...Bye")
        sleep(wait)
        sys.exit()


    ### Printing the sequences ###

    print("The input sequences are\n")
    sleep(wait)

    print("Sequence1: " + sequence1)
    print("Sequence2: " + sequence2)
    print("\n") ### Empty line

    mymatrix = create_matrix(len(sequence2), len(sequence1))
    mymatrix = build_matrix(mymatrix)
    print_matrix(mymatrix)

    print_traceback(mymatrix)
    ### Taking the absolute value as EMBOSS water does not permit negative values for parameters ###

    # seqmismatch = abs(seqmismatch)
    seqgap = abs(seqgap)


    print("\n\nComparing output with EMBOSS water")

    sleep(wait)

    water_command = "water -asequence {} -bsequence {} -gapopen {} -gapextend {} -outfile water_{}_{}.water -datafile EDNAFULL_srt".format(seq1_file, seq2_file, seqgap, seqgap, seq1_file, seq2_file)
    os.system(water_command)

    print("\n\nThis is the output from EMBOSS water using the same files which is our Gold-Standard for alignment")

    sleep(wait)

    display_water_file = "cat water_{}_{}.water".format(seq1_file, seq2_file)
    os.system(display_water_file)

    sleep(wait)

    print("\nThanks for using this script.\n")



##this calls the SW algorithm when the script loads
perform_smith_waterman()
```

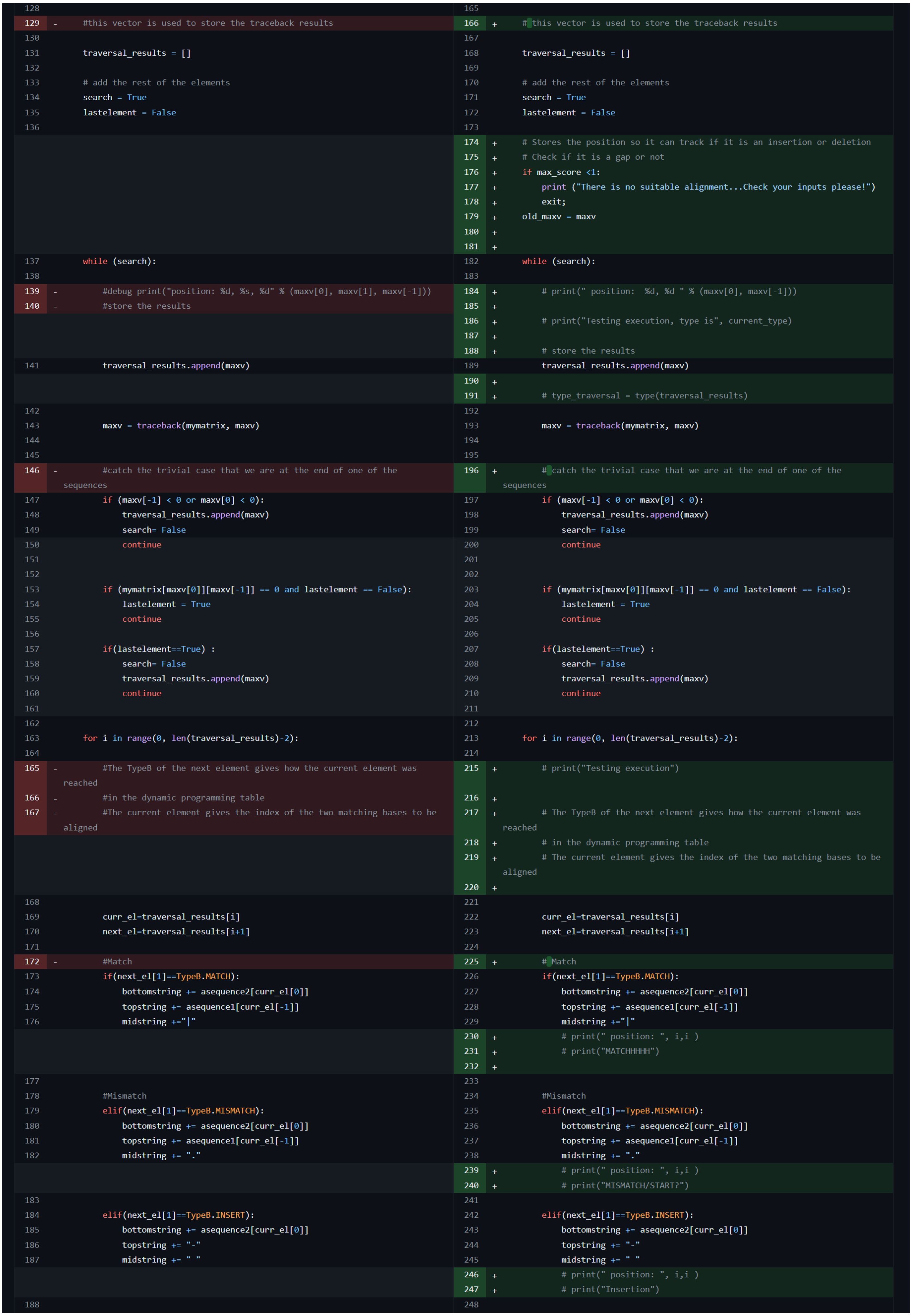


```
@@ -1,13 +1,32 @@
+#!/usr/bin/python3
+
+### Bioinformatics Algorithms ICA Script by B236494 ###
+
+### Adapted from BA4 class code written by Simon Tomlinson ###
+
+### print statements were used throughout execution to test the outputs of the program ###
+
+### A test has also been implemented at the end to compare the output witht the EMBOSS water Gold-standard ###
+
+### B236494_SmithWaterman_Final_Script.py script adapted from SmithWaterman.py Version 1.9 SRT ###
+
 # Simon Tomlinson Bioinformatics Algorithms
 # Perform Smith Waterman Alignment in Python (from first principles)
 # contains rows lists each of length cols initially set to 0
 # index as my_matrix[1][2] my_matrix[R][C]
-# Teaching code not production code!!
-#Version 1.9 SRT
+#
+# Version 1.9 SRT

+import time
+import os
+import sys
+from time import sleep
 from enum import Enum
+import pandas as pd
+import re
+import argparse

-#Global enumeration used for tracement
+# Global enumeration used for tracement
 TypeB = Enum('TypeB', ['INSERT', 'DELETE', 'MISMATCH', 'MATCH', 'END'])


 def create_matrix(rows, cols):
     my_matrix = [[0 for col in range(cols + 1)] for row in range(rows + 1)]
     return my_matrix


 # x is row index, y is column index
 # follows[r][c]

 def calc_score(matrix, x, y):
-    # print("seq1:",sequence1[y- 1]," seq2: "+sequence2[x - 1],"x:",x," y:",y)
-
+    print("seq1:",sequence1[y- 1]," seq2: "+sequence2[x - 1],"x:",x," y:",y)
     sc = seqmatch if sequence1[y - 1] == sequence2[x - 1] else seqmismatch
-
     base_score = matrix[x - 1][y - 1] + sc
     insert_score = matrix[x - 1][y] + seqgap
     delete_score = matrix[x][y - 1] + seqgap
     v = max(0, base_score, insert_score, delete_score)
     return v


 # makes a single traceback step
 def traceback(mymatrix, maxv):
     x = maxv[0]
     y = maxv[-1]
     val = mymatrix[x][y]

     # todo add some outputs for checking errors
     sc = seqmatch if sequence2[x - 1] == sequence1[y - 1] else seqmismatch
+    # print(sc)
+
     base_score = mymatrix[x - 1][y - 1] + sc
+    # print(base_score)
     if base_score == val:
         if sc==seqmatch:
             return [x - 1,TypeB.MATCH, y - 1]
         else:
             return [x - 1,TypeB.MISMATCH, y - 1]

     insert_score = mymatrix[x - 1][y] + seqgap
+    # print(input_score)
     if insert_score == val:
         return [x - 1, TypeB.INSERT, y]
     else:
         return [x, TypeB.DELETE, y - 1]


+
+
+
```



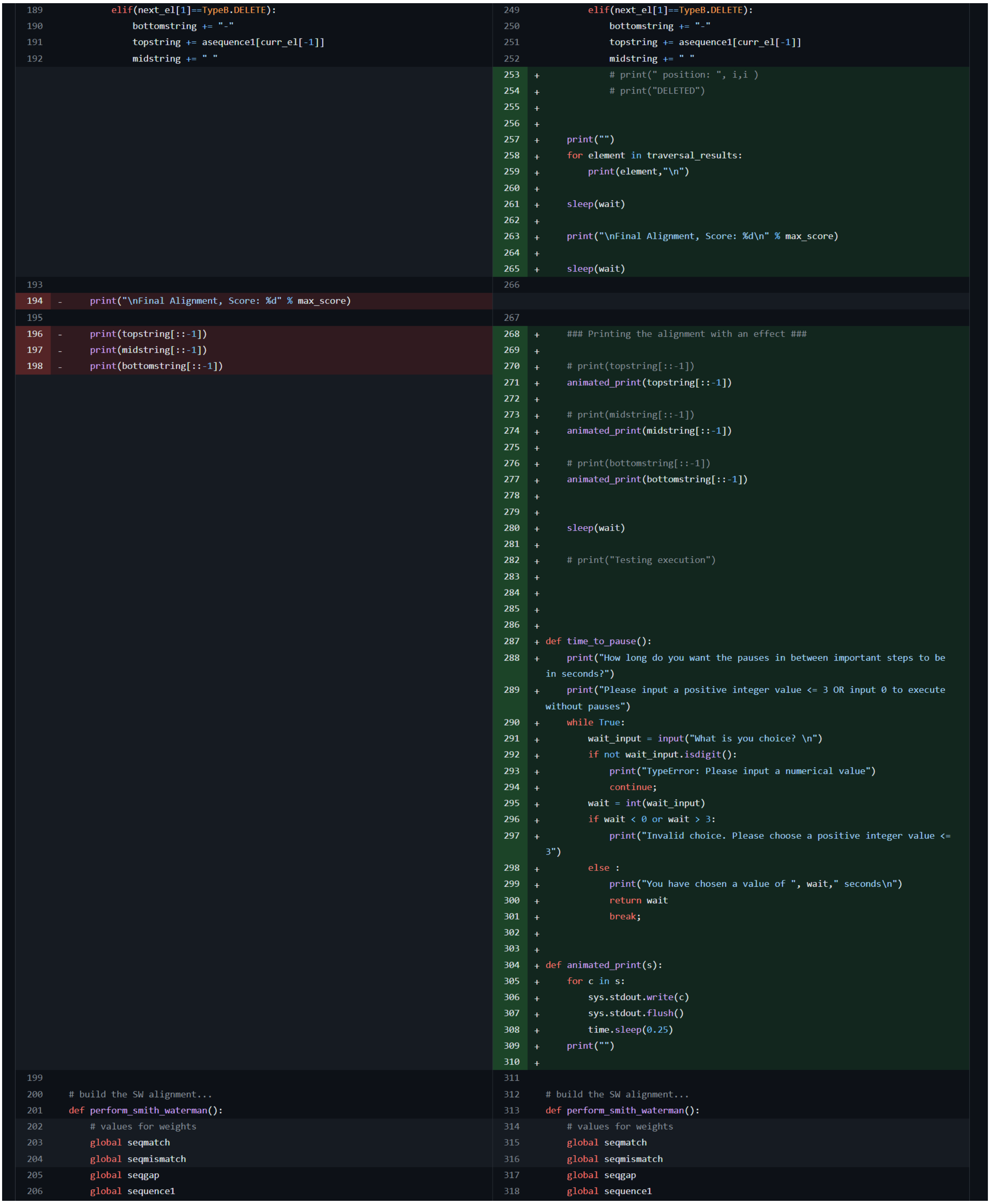
```
# builds the initial scoring matrix used for traceback
def build_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
-
    for i in range(1, rows):
        for j in range(1, cols):
            mymatrix[i][j] = calc_score(mymatrix, i, j)

    return mymatrix


# gets the max value from the built matrix
# Max is the end of the traceback for SW
def get_max(mymatrix):
    max = mymatrix[0][0]
    mrow = 0
    mcol = 0

    rows = len(mymatrix)
    cols = len(mymatrix[0])

    for i in range(1, rows):
        for j in range(1, cols):
            if mymatrix[i][j] > max:
                max = mymatrix[i][j]
                mrow = i
                mcol = j
-   print("max score: ", max)
    return [mrow, TypeB.END, mcol]


# print out the best scoring path from the SW matrix
def print_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    s1 = "  " + sequence1
    s2 = " " + sequence2

-   print("Dimensions: r= %2d , c= %2d" % (rows, cols))

    for a in s1:
        print(a, end="")
        print(" \t", end="")
    print("\n", end="")

    for i in range(0, rows):
        print(s2[i], end="")
        print(" \t", end="")
        for j in range(0, cols):
            print("%02d\t" % (mymatrix[i][j]), end="")
        print("\n", end="")


# print out the traceback of the best scoring alignment
def print_traceback(mymatrix):
    # this will print as expected with internal gaps
-   print("Building traceback...")
    maxv = get_max(mymatrix)

-   #stash the max score for later
    max_score = mymatrix[maxv[0]][maxv[-1]]

    # traverse the matrix to find the traceback elements
    # if more than one path just pick one
    topstring = ""
    midstring = ""
    bottomstring = ""

    # pad the sequences so indexes into the sequences match the matrix indexes
    asequence1 = "#" + sequence1
    asequence2 = "#" + sequence2
```

```
# builds the initial scoring matrix used for traceback
def build_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
+   row_number=0
+
    for i in range(1, rows):
+       row_number = row_number + 1
+       print("\nRow Number:", row_number)
+       sleep(wait)
        for j in range(1, cols):
            mymatrix[i][j] = calc_score(mymatrix, i, j)

    return mymatrix


# gets the max value from the built matrix
# Max is the end of the traceback for SW
def get_max(mymatrix):
    max = mymatrix[0][0]
    mrow = 0
    mcol = 0

    rows = len(mymatrix)
    cols = len(mymatrix[0])

    for i in range(1, rows):
        for j in range(1, cols):
            if mymatrix[i][j] > max:
                max = mymatrix[i][j]
                mrow = i
                mcol = j
+   print("The Maximum Score was: ", max,"\n")
    return [mrow, TypeB.END, mcol]


# print out the best scoring path from the SW matrix
def print_matrix(mymatrix):
    rows = len(mymatrix)
    cols = len(mymatrix[0])
    s1 = "  " + sequence1
    s2 = " " + sequence2
+
+   sleep(wait)

+   print("\nDimensions of The SmithWaterman Matrix: Rows= %2d , Columns=%2d\n" % (rows, cols))
+
+   sleep(wait)

    for a in s1:
        print(a, end="")
        print(" \t", end="")
    print("\n", end="")

    for i in range(0, rows):
        print(s2[i], end="")
        print(" \t", end="")
        for j in range(0, cols):
            print("%02d\t" % (mymatrix[i][j]), end="")
        print("\n", end="")


# print out the traceback of the best scoring alignment
def print_traceback(mymatrix):
    # this will print as expected with internal gaps
+
+   sleep(wait)
+
+   print("\n### We Will Now Build The Traceback... ###\n")
    maxv = get_max(mymatrix)
+   print(maxv)

+   # stash the max score for later
    max_score = mymatrix[maxv[0]][maxv[-1]]

+
    # traverse the matrix to find the traceback elements
    # if more than one path just pick one
    topstring = ""
    midstring = ""
    bottomstring = ""

    # pad the sequences so indexes into the sequences match the matrix indexes
    asequence1 = "#" + sequence1
    asequence2 = "#" + sequence2
```



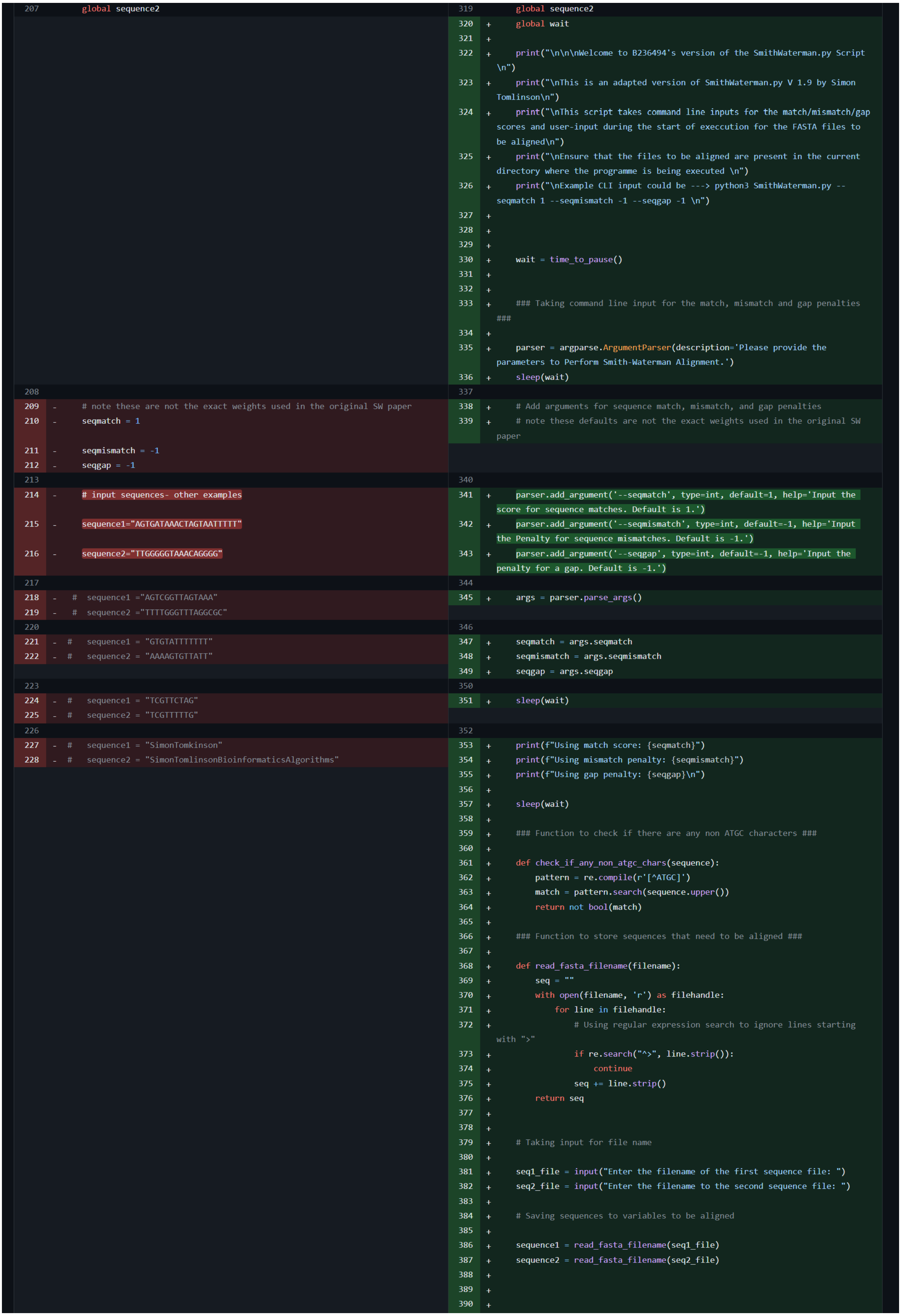
```
-    #this vector is used to store the traceback results

     traversal_results = []

     # add the rest of the elements
     search = True
     lastelement = False

     while (search):

-        #debug print("position: %d, %s, %d" % (maxv[0], maxv[1], maxv[-1]))
-        #store the results
         traversal_results.append(maxv)

         maxv = traceback(mymatrix, maxv)


-        #catch the trivial case that we are at the end of one of the sequences
         if (maxv[-1] < 0 or maxv[0] < 0):
             traversal_results.append(maxv)
             search= False
             continue


         if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):
             lastelement = True
             continue

         if(lastelement==True) :
             search= False
             traversal_results.append(maxv)
             continue

     for i in range(0, len(traversal_results)-2):

-        #The TypeB of the next element gives how the current element was reached
-        #in the dynamic programming table
-        #The current element gives the index of the two matching bases to be aligned

         curr_el=traversal_results[i]
         next_el=traversal_results[i+1]

-        #Match
         if(next_el[1]==TypeB.MATCH):
             bottomstring += asequence2[curr_el[0]]
             topstring += asequence1[curr_el[-1]]
             midstring +="|"

         #Mismatch
         elif(next_el[1]==TypeB.MISMATCH):
             bottomstring += asequence2[curr_el[0]]
             topstring += asequence1[curr_el[-1]]
             midstring += "."

         elif(next_el[1]==TypeB.INSERT):
             bottomstring += asequence2[curr_el[0]]
             topstring += "-"
             midstring += " "

```

```

+    # this vector is used to store the traceback results

     traversal_results = []

     # add the rest of the elements
     search = True
     lastelement = False

+    # Stores the position so it can track if it is an insertion or deletion
+    # Check if it is a gap or not
+    if max_score <1:
+        print ("There is no suitable alignment...Check your inputs please!")
+        exit;
+    old_maxv = maxv
+
+
     while (search):

+        # print(" position:  %d, %d " % (maxv[0], maxv[-1]))
+
+        # print("Testing execution, type is", current_type)
+
+        # store the results
         traversal_results.append(maxv)
+
+        # type_traversal = type(traversal_results)

         maxv = traceback(mymatrix, maxv)


+        # catch the trivial case that we are at the end of one of the sequences
         if (maxv[-1] < 0 or maxv[0] < 0):
             traversal_results.append(maxv)
             search= False
             continue


         if (mymatrix[maxv[0]][maxv[-1]] == 0 and lastelement == False):
             lastelement = True
             continue

         if(lastelement==True) :
             search= False
             traversal_results.append(maxv)
             continue

     for i in range(0, len(traversal_results)-2):

+        # print("Testing execution")
+
+        # The TypeB of the next element gives how the current element was reached
+        # in the dynamic programming table
+        # The current element gives the index of the two matching bases to be aligned
+

         curr_el=traversal_results[i]
         next_el=traversal_results[i+1]

+        # Match
         if(next_el[1]==TypeB.MATCH):
             bottomstring += asequence2[curr_el[0]]
             topstring += asequence1[curr_el[-1]]
             midstring +="|"
+            # print(" position: ", i,i )
+            # print("MATCHHHHH")
+

         #Mismatch
         elif(next_el[1]==TypeB.MISMATCH):
             bottomstring += asequence2[curr_el[0]]
             topstring += asequence1[curr_el[-1]]
             midstring += "."
+            # print(" position: ", i,i )
+            # print("MISMATCH/START?")

         elif(next_el[1]==TypeB.INSERT):
             bottomstring += asequence2[curr_el[0]]
             topstring += "-"
             midstring += " "
+            # print(" position: ", i,i )
+            # print("Insertion")

```



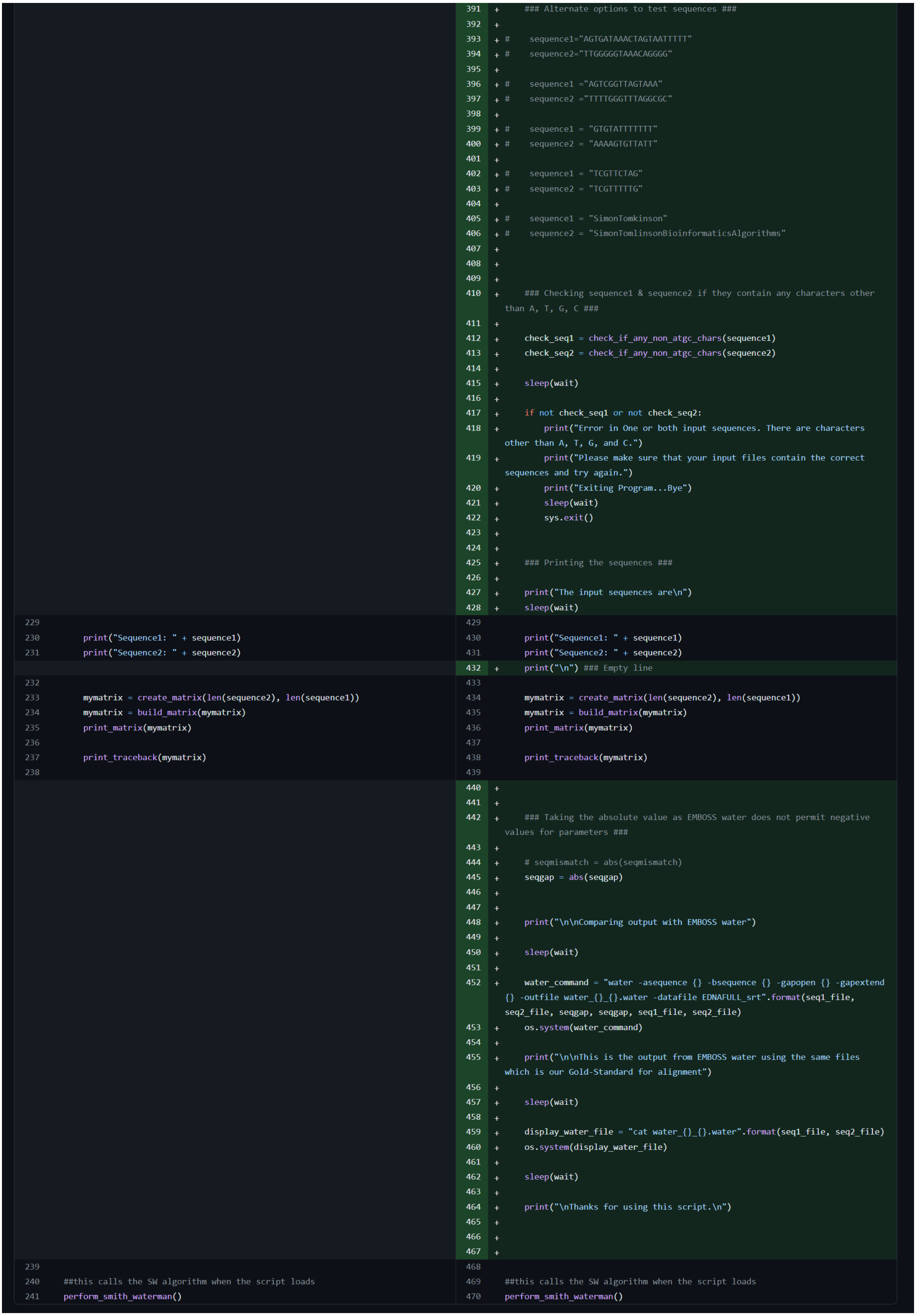
```
         elif(next_el[1]==TypeB.DELETE):
             bottomstring += "-"
             topstring += asequence1[curr_el[-1]]
             midstring += " "

-    print("\nFinal Alignment, Score: %d" % max_score)

-    print(topstring[::-1])
-    print(midstring[::-1])
-    print(bottomstring[::-1])

 # build the SW alignment...
 def perform_smith_waterman():
     # values for weights
     global seqmatch
     global seqmismatch
     global seqgap
     global sequence1
```

```
         elif(next_el[1]==TypeB.DELETE):
             bottomstring += "-"
             topstring += asequence1[curr_el[-1]]
             midstring += " "
+            # print(" position: ", i,i )
+            # print("DELETED")
+
+
+    print("")
+    for element in traversal_results:
+        print(element,"\n")
+
+    sleep(wait)
+
+    print("\nFinal Alignment, Score: %d\n" % max_score)
+
+    sleep(wait)


+    ### Printing the alignment with an effect ###
+
+    # print(topstring[::-1])
+    animated_print(topstring[::-1])
+
+    # print(midstring[::-1])
+    animated_print(midstring[::-1])
+
+    # print(bottomstring[::-1])
+    animated_print(bottomstring[::-1])
+
+
+    sleep(wait)
+
+    # print("Testing execution")
+
+
+
+
+def time_to_pause():
+    print("How long do you want the pauses in between important steps to be in seconds?")
+    print("Please input a positive integer value <= 3 OR input 0 to execute without pauses")
+    while True:
+        wait_input = input("What is you choice? \n")
+        if not wait_input.isdigit():
+            print("TypeError: Please input a numerical value")
+            continue;
+        wait = int(wait_input)
+        if wait < 0 or wait > 3:
+            print("Invalid choice. Please choose a positive integer value <= 3")
+        else :
+            print("You have chosen a value of ", wait," seconds\n")
+            return wait
+            break;
+
+
+def animated_print(s):
+    for c in s:
+        sys.stdout.write(c)
+        sys.stdout.flush()
+        time.sleep(0.25)
+    print("")
+

 # build the SW alignment...
 def perform_smith_waterman():
     # values for weights
     global seqmatch
     global seqmismatch
     global seqgap
     global sequence1
```



```
    global sequence2

-   # note these are not the exact weights used in the original SW paper
-   seqmatch = 1

-   seqmismatch = -1
-   seqgap = -1

-   # input sequences- other examples

-   sequence1="AGTGATAAACTAGTAATTTTT"

-   sequence2="TTGGGGGTAAACAGGGG"

-  #  sequence1 ="AGTCGGTTAGTAAA"
-  #  sequence2 ="TTTTGGGTTTAGGCGC"

- #   sequence1 = "GTGTATTTTTTTT"
- #   sequence2 = "AAAAGTGTTATT"

- #   sequence1 = "TCGTTCTAG"
- #   sequence2 = "TCGTTTTTG"

- #   sequence1 = "SimonTomkinson"
- #   sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"
```

```
    global sequence2
+   global wait
+
+   print("\n\n\nWelcome to B236494's version of the SmithWaterman.py Script \n")
+   print("\nThis is an adapted version of SmithWaterman.py V 1.9 by Simon Tomlinson\n")
+   print("\nThis script takes command line inputs for the match/mismatch/gap scores and user-input during the start of execcution for the FASTA files to be aligned\n")
+   print("\nEnsure that the files to be aligned are present in the current directory where the programme is being executed \n")
+   print("\nExample CLI input could be ---> python3 SmithWaterman.py --seqmatch 1 --seqmismatch -1 --seqgap -1 \n")
+
+
+
+   wait = time_to_pause()
+
+
+   ### Taking command line input for the match, mismatch and gap penalties ###
+
+   parser = argparse.ArgumentParser(description='Please provide the parameters to Perform Smith-Waterman Alignment.')
+   sleep(wait)

+   # Add arguments for sequence match, mismatch, and gap penalties
+   # note these defaults are not the exact weights used in the original SW paper

+   parser.add_argument('--seqmatch', type=int, default=1, help='Input the score for sequence matches. Default is 1.')
+   parser.add_argument('--seqmismatch', type=int, default=-1, help='Input the Penalty for sequence mismatches. Default is -1.')
+   parser.add_argument('--seqgap', type=int, default=-1, help='Input the penalty for a gap. Default is -1.')

+   args = parser.parse_args()

+   seqmatch = args.seqmatch
+   seqmismatch = args.seqmismatch
+   seqgap = args.seqgap

+   sleep(wait)

+   print(f"Using match score: {seqmatch}")
+   print(f"Using mismatch penalty: {seqmismatch}")
+   print(f"Using gap penalty: {seqgap}\n")
+
+   sleep(wait)
+
+   ### Function to check if there are any non ATGC characters ###
+
+   def check_if_any_non_atgc_chars(sequence):
+       pattern = re.compile(r'[^ATGC]')
+       match = pattern.search(sequence.upper())
+       return not bool(match)
+
+   ### Function to store sequences that need to be aligned ###
+
+   def read_fasta_filename(filename):
+       seq = ""
+       with open(filename, 'r') as filehandle:
+           for line in filehandle:
+               # Using regular expression search to ignore lines starting with ">"
+               if re.search("^>", line.strip()):
+                   continue
+               seq += line.strip()
+       return seq
+
+
+   # Taking input for file name
+
+   seq1_file = input("Enter the filename of the first sequence file: ")
+   seq2_file = input("Enter the filename to the second sequence file: ")
+
+   # Saving sequences to variables to be aligned
+
+   sequence1 = read_fasta_filename(seq1_file)
+   sequence2 = read_fasta_filename(seq2_file)
+
+
+
```



```
+    ### Alternate options to test sequences ###
+
+ #    sequence1="AGTGATAAACTAGTAATTTTT"
+ #    sequence2="TTGGGGGTAAACAGGGG"
+
+ #    sequence1 ="AGTCGGTTAGTAAA"
+ #    sequence2 ="TTTTGGGTTTAGGCGC"
+
+ #    sequence1 = "GTGTATTTTTTTT"
+ #    sequence2 = "AAAAGTGTTATT"
+
+ #    sequence1 = "TCGTTCTAG"
+ #    sequence2 = "TCGTTTTTG"
+
+ #    sequence1 = "SimonTomkinson"
+ #    sequence2 = "SimonTomlinsonBioinformaticsAlgorithms"
+
+
+
+    ### Checking sequence1 & sequence2 if they contain any characters other than A, T, G, C ###
+
+    check_seq1 = check_if_any_non_atgc_chars(sequence1)
+    check_seq2 = check_if_any_non_atgc_chars(sequence2)
+
+    sleep(wait)
+
+    if not check_seq1 or not check_seq2:
+        print("Error in One or both input sequences. There are characters other than A, T, G, and C.")
+        print("Please make sure that your input files contain the correct sequences and try again.")
+        print("Exiting Program...Bye")
+        sleep(wait)
+        sys.exit()
+
+
+    ### Printing the sequences ###
+
+    print("The input sequences are\n")
+    sleep(wait)

     print("Sequence1: " + sequence1)
     print("Sequence2: " + sequence2)
+    print("\n") ### Empty line

     mymatrix = create_matrix(len(sequence2), len(sequence1))
     mymatrix = build_matrix(mymatrix)
     print_matrix(mymatrix)

     print_traceback(mymatrix)

+
+
+    ### Taking the absolute value as EMBOSS water does not permit negative values for parameters ###
+
+    # seqmismatch = abs(seqmismatch)
+    seqgap = abs(seqgap)
+
+
+    print("\n\nComparing output with EMBOSS water")
+
+    sleep(wait)
+
+    water_command = "water -asequence {} -bsequence {} -gapopen {} -gapextend {} -outfile water_{}_{}.water -datafile EDNAFULL_srt".format(seq1_file, seq2_file, seqgap, seqgap, seq1_file, seq2_file)
+    os.system(water_command)
+
+    print("\n\nThis is the output from EMBOSS water using the same files which is our Gold-Standard for alignment")
+
+    sleep(wait)
+
+    display_water_file = "cat water_{}_{}.water".format(seq1_file, seq2_file)
+    os.system(display_water_file)
+
+    sleep(wait)
+
+    print("\nThanks for using this script.\n")
+
+
+

 ##this calls the SW algorithm when the script loads
 perform_smith_waterman()
```

```
     print("Sequence1: " + sequence1)
     print("Sequence2: " + sequence2)

     mymatrix = create_matrix(len(sequence2), len(sequence1))
     mymatrix = build_matrix(mymatrix)
     print_matrix(mymatrix)

     print_traceback(mymatrix)


 ##this calls the SW algorithm when the script loads
 perform_smith_waterman()
```