**CS423 Lab 5: String Alignment**

**Authors: Sara Perkins, Caleb Piekstra**

**Lab Exercises:**

***Timing experiment (record answers as you do these, so you have them for your write-up)***

1. Download the code (globalAlignmentSearch.py). This code includes the brute force search implementation of the global alignment algorithm. Read through the code to ensure your understanding of this version of the algorithm. Save the file and run it. It should print out some text, the best global alignment score, and the running time of the program. How long did it take (in sec)? \_\_\_**0.05**\_\_

2. Change the strings s and t in the file to something different, such that the strings have 10 characters each. Re-run the program. How long did it take (in sec)? \_\_\_\_\_**0.19**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Change the strings s and t in the file such that the strings have 11 characters each. Re-run the program. How long did it take (in sec)? \_\_\_\_\_\_\_\_**0.76**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Now run the program with strings of length 12. How long did it take (in sec)? \_\_\_\_\_\_\_**3.05**\_\_\_\_\_\_\_\_\_

5. Now use strings of length 13. How long did it take (in sec)? \_\_\_\_\_**12.35**\_\_\_\_\_\_

*At some point, you may not want to watch the screen, so it is recommended that you use the other person’s computer to work on the next part of the lab while this code is running.*

6. Now use strings of length 14. How long did it take (in sec)? \_\_\_\_**49.95**\_\_\_\_\_

7. Now use strings of length 15. How long did it take (in sec)? \_\_\_\_\_\_**203.04**\_\_\_\_\_\_\_\_\_

**Write-up (45 points):**

Create a Word or pdf file for your writeup of this lab and to show that you tested your code. Be sure to include your name(s) and lab number in your file. Unless otherwise specified below, use your implementation of global string alignment (not the search implementation) to answer the questions.

1. (6 pts) Let S = “AGCGTCTA” and T = “TGCATCTCGT”. Run your code on these sequences.
   1. What is the optimal **global** alignment score?

**-1**

* 1. What is the optimal **global** alignment?

**AGC-TCT--A**

**TGCATCTCGT**

* 1. What is the optimal **local** alignment score?

**21**

* 1. What is the optimal **local** alignment?

**GCCTCT**

**GCATCT**

* 1. What is the length of the LCS?

**5**

* 1. What is the LCS?

**GCTCT**

1. (6 pts) Load the FASTA files containing the coding region of PHO12 in yeast (yeastPHO12.txt) and the fly (flyPHO12.txt). You may copy/paste the convertFiletoSequence function from lab 1 to help you convert the FASTA file contents to a string. Answer the following questions based on these two DNA sequences.
   1. What is the optimal **global** alignment score?

**549**

* 1. Include the optimal **global** alignment of these two sequences as a text file in your zipped folder. Name the file PHO12\_global\_alignment.txt.
  2. What is the optimal **local** alignment score?

**660**

* 1. Include the optimal **local** alignment of these two sequences as a text file in your zipped folder. Name the file PHO12\_local\_alignment.txt.
  2. What is the length of the LCS?

**900**

* 1. Include the LCS as a text file in your zipped folder. Name the file PHO12\_LCS.txt.

1. (6 pts) Write a new python script called genRandom.py that defines the function genRandom that takes 4 parameters: length of string to generate, %AT, %CG, and output\_file\_name. It should generate a random DNA sequence of the given length that contains nucleotides produced at the AT and CG rates and writes the DNA sequence to a FASTA-formatted file (> random sequence in first line). Use the function to create two files. The first randomly generated DNA sequence should be the same length as the PHO12 yeast gene DNA sequence, with **38%** GC content. The second randomly generated DNA sequence should be the same length as the PHO12 fly gene, with **42%** GC content. Name the files yeast\_random.txt and fruit\_fly\_random.txt and includes these files in your zip file submission. You may find previous lab exercises useful for this task of generating a randomly composed DNA sequence and writing it to a file.
   1. Do you expect the global alignment score of the random DNA sequences to be less than, greater than, or about equal to the score you get for aligning the yeast and fruit fly genes in exercise 3? Why?

**We expect the global alignment score to be less than the score we get for aligning yeast and fruit fly. We are looking at a randomly generated sequence versus a naturally-selected mutated ortholog.**

* 1. Do you expect the local alignment score of the random DNA sequences to be less than, greater than, or about equal to the score you get for locally aligning the yeast and fruit fly genes in exercise 3? Why?

**We expect the global alignment score to be less than the score we get for aligning yeast and fruit fly. We are looking at a randomly generated sequence versus a naturally-selected mutated ortholog. Since it is an ortholog, they are fairly similar with a few insertions, deletions, or substitutions, whereas random has no similar heritage.**

* 1. Run your global alignment code on these two randomly generated DNA sequences. What is the optimal **global** alignment score of these random DNA sequences? Did this confirm your predication in (a)?

**Global—667; no**

* 1. Run your local alignment code on these randomly generated DNA sequences. What is the optimal **local** alignment score of these random DNA sequences? Did this confirm your prediction in (b)?

**Local—714, no**

1. (3 pts) Relationship between global alignment and local alignment
   1. Assume you are aligning two strings A and B and assume the same scoring model is used for global alignment and local alignment. Is the global score always smaller than or equal to the local score? Why or why not?

**Yes. Since the local alignment can skip bases at the beginning and/or end of the strings, it can skip a bunch of mismatches or insertions/deletions that the global alignment would have to account for, decreasing the global alignment score permanently. Also, the local alignment does not use negative values, simply replacing any negatives with 0 (a fresh start). In contrast, the global alignment saves negatives which can lower the score significantly compared to a 0.**

* 1. When are the two scores (local and global) equal?

**The two alignment values are equal when the strings align perfectly (all ordered bases in string one are identical to string two).**

1. (2 pts) Suppose you are aligning two short DNA sequences and, separately, aligning two long DNA sequences. Do you expect the optimal **local** alignment score to be larger for the short sequences or the long sequences? Why?

**Larger for the long sequences because you have more chances to get higher scoring alignments.**

1. (2 pts) Suppose you are aligning two short DNA sequences and, separately, aligning two long DNA sequences. Do you expect the optimal **global** alignment score to be bigger for the short sequences or the long sequences? Why?

**Bigger for the short sequences because the long ones have a higher chance of getting lots of mismatches and gaps, lowering the score substantially.**

1. (6 pts) Answer questions 4, 5, and 6 before doing this experiment. Previous lab exercises may provide guidance on this part. Download experiment.py and write the code to conduct the experiment described in the comments. Use the standard scoring system in local and global alignment (g = -6, match = 5, mismatch = -4). Run the experiment with strings of length 5 through 100, stepping by 5. Using the output file, create an excel line graph using the first column as the x data series and the second and third columns as the y data series. Paste the figure here.

**Figure 1: Excel graph of alignment scores across sequences of different lengths.**

1. (2 pts) Did the experiment confirm your predictions in questions 4, 5, and 6? If not, do you understand the trends in scores for local and global alignment?

**4. Confirmed—the global score is always less than the local for our results, however we never**

**found a case where they were exactly equal because with the random generation of sequences, we most likely never produced two sequences that were exactly the same**

**5. Confirmed—the general trend of the local alignment score is larger for the longer sequences**

**6. Somewhat confirmed—the general trend of the global scores is slightly higher for the larger**

**sequences (of lengths >= 50), yet there are also some big exceptions, for example**

**length 95. It appears that in some cases, the longer sequence had a negative effect,**

**causing the score to gather more negative penalties if the strings turned out to**

**differ greatly. While the larger sequences provide more chances to earn positive**

**scores, they simultaneously provide the same opportunity for the collection of**

**negative scores as well.**

1. (6 pts) Now, write a new python program called timingExperiment.py to run the following experiment:
   1. For N = 9 to 15 stepping by 1, generate 2 random DNA sequences (25% each for nucleotides), each of length N
      1. Time how long it takes your implementation of globalAlignment to calculate the optimal score (see globalAlignmentSearch.py to see how to use the time module) for strings of length N. Comment out the call to align in globalAlignmentScore, so the timing just includes how long it takes to get the score (and not the reconstructed alignment).
      2. Print the time results to a file with each line containing N, followed by tab, followed by the time.
   2. Open the resulting file in excel.
   3. Take the values you got for the search implementation version from the first part of this lab and insert those numbers into the excel file. Create a graph. Values for N are the x-axis with two line graphs (one for the dynamic programming) and one for the brute force search. Insert the graph here.

**Figure 2: Run times of brute force and dynamic programming algorithms on sequences of different lengths.**

* 1. Did the results of the graph match your expectations? Why or why not?

**Yes, the brute force technique generates a program with a run-time of O(2^(2N)), which has a very steep curve as N increases. The dynamic programing technique however is of magnitude O(N^2), providing a much gentler run time for all values N. While the brute force times increases sharply with each additional character, the dynamic solution remains relatively stable.**

* 1. If you have time, you can increase N past 15 and run the timings for the dynamic programming and brute search versions and re-draw the graph.

1. (6 pts) Using the code you wrote in 10, run the timing experiment for the dynamic programming algorithm implementation for global alignment for N from 50 to 5000, stepping by 50. Include the line graph with x-axis as N and y-axis as time in the report here.

**Figure 3: Dynamic programming run times for sequences from length 50-5000**

* 1. What is the big O running time of the dynamic programming algorithm for global alignment?

**O(N \* M)**

**Where N is the length of sequence 1 and M is the length of sequence 2**

**( If N = M then this is O(N2) )**

* 1. How does the big O function match the experimental results?

**The big O function seems to match the experimental results exactly, the curve looks like an N2 curve and since the length of both sequences are the same, this is a case of N\*N or O(N2).**

* 1. To align two DNA sequences, each of length N, how much memory is required using the dynamic programming approach? (in other words, how large is the scoring table?)

**The scoring table has N\*N entries, or N2. Thus, as the amount of memory used is the amount of memory for a single entry in the table times N2.**

* 1. To align two DNA sequences, each of length N, how much memory is required using the brute force search (in other words, how much data is stored while the algorithm runs)?

**Because the brute force is storing the best alignment substrings for S and T, the worst case scenario would be that both substrings are the entire strings S and T, which are each length N, meaning that there is 2N data being stored. However, based on analysis, the search is also storing all subsets of S and T, which would be** http://www4b.wolframalpha.com/Calculate/MSP/MSP8101eb9ca0ce2g6gc6100000i120d7335ca7d93?MSPStoreType=image/gif&s=15&w=125.&h=48.

**for each sequence S and T. So, the total amount of data being stored would be 2N + N(N+1) or 3N + N2. As such, the amount of memory used would be the cost of storing one letter in the sequence times (3N + N2) where N is the length of S or T (they both have the same length).**

**Appendix A: Authorship (please include statement in your write-up)**

The code and write-up submitted for this lab were authored by Caleb Piekstra and Sara Perkins. All external sources to BIO/CS423 are cited properly.

**Appendix B: Code (40 points, based on correctness and style)**

Copy and paste the code (all files – three dynamic programming files, genRandom.py, timingExperiment.py) you wrote for lab 5 here (use Courier 8pt font). Also, upload the code files to Moodle as part of you zip folder.

Put beautiful code here!

#!/usr/bin/python# Caleb Piekstra and Sara Perkins# CS 423, Lab 5# fall 2015##################################################################### convertFileToSequence - takes a FASTA file and returns the# sequence as a string#####################################################################def convertFileToSequence(filename): # read in file file = open(filename) # read in first line header = file.readline() if (header[0] == '>'): print("in FASTA format") else: print("invalid format") file.close() return # read in rest of file sequence = file.read() # close file file.close() # remove all return and newline characters sequence = sequence.replace("\r", "") sequence = sequence.replace("\n", "") return sequence####################################################################### Determines the score of the optimal global alignment of two strings######################################################################def globalAlignmentScore(s1, s2): # add a blank string as padding s1 = ' ' + s1 s2 = ' ' + s2 # Scoring system MATCH = 5 MISMATCH = -4 GAP = -6 # set table size NUM\_ROWS = len(s2) NUM\_COLS = len(s1) # Create table and fill it with zeros c = createTable(NUM\_ROWS, NUM\_COLS, 0) # Creates table for getting back the optimal alignment, fill table with "F" # uses "D", "L", and "T" for diagonal, left, and top d = createTable(NUM\_ROWS, NUM\_COLS, "F") # implements dynamic programming algorithm for global alignment # fills in entries in cost table and direction table # left column, align with empty string for i in range(1, NUM\_ROWS): c[i][0] = GAP\*i d[i][0] = "T" # top row, align with empty string for j in range(1, NUM\_COLS): c[0][j] = GAP\*j d[0][j] = "L" # rest of table for i in range(1, NUM\_ROWS): for j in range(1, NUM\_COLS): # set value of match/mismatch m = MATCH if s1[j] == s2[i] else MISMATCH # calculate costs based on direction left = c[i][j-1] + GAP top = c[i-1][j] + GAP diag = c[i-1][j-1] + m # set cost cost = max(left,top,diag) c[i][j] = cost # set direction based on cost if cost == diag: d[i][j] = "D" elif cost == top: d[i][j] = "T" else: d[i][j] = "L" # Prints out table (only useful for small tables - used for debugging) # Commented out, satisfied that the algorithm is working # printTable(c, "costs.txt") # printTable(d, "directions.txt") # find optimal alignment #align(d, s1, s2, "alignment.txt") # return optimal score (lower right-hand cell in table] return c[NUM\_ROWS-1][NUM\_COLS-1]##################################################################### createTable - Creates a 2D table with the given number of rows and# columns and fills all entries with value given as a parameter# (function completed by Tammy VanDeGrift)####################################################################def createTable(numRows, numCols, value): table = [] row = 0 # create 2D table initialized with value while (row < numRows): table.append([]) col = 0 while (col < numCols): table[row].append(value) col = col + 1 row = row + 1 return table################################################################### printTable- Prints 2D table to file (only useful for small # tables for short strings), can be used for integer as well as# char values tabs between the values on each row# Useful function for debugging purposes only##################################################################def printTable(table, filename): NUM\_ROWS = len(table) NUM\_COLS = len(table[0]) # write table to output file, one row at a time with open(filename, 'w') as out: for i in range(0, NUM\_ROWS): row = "" for j in range(0, NUM\_COLS): row += str(table[i][j]) + '\t' out.write(row+'\n') return################################################################# align - Reconstructs the optimal alignment and prints the # alignment to a file. Because the sequences can be long, prints # the alignment 50 characters on one line, the other string of # 50 characters on the next line, and then skips one line, as # follows:## AATT--GGCTATGCT--C-G-TTACGCA-TTACT-AA-TCCGGTC-AGGC# AAATATGG---TGCTGGCTGCTT---CAGTTA-TGAACTCC---CCAGGC## TATGGGTGCTATGCTCG--T--TACG-CA# TCAT--TGG---TGCTGGCTGCTT--ACA## direction is a 2D table, s1 and s2 are the original DNA# sequences to align, and filename is the name of the output file###############################################################def align(direction, s1, s2, filename): # initialize rows, columnss, and newly aligned sequences row = len(direction) -1 col = len(direction[0]) -1 alignS1 = "" alignS2 = "" # evalutate table for all valid indices while(row >= 0 and col >= 0): d = direction[row][col] # start with bottom right # diagnol if d == "D": alignS1 = s1[col] + alignS1 alignS2 = s2[row] + alignS2 row -= 1 col -= 1 # top elif d == "T": alignS1 = "-" + alignS1 alignS2 = s2[row] + alignS2 row -= 1 # left elif d == "L": alignS1 = s1[col] + alignS1 alignS2 = "-" + alignS2 col -= 1 # first, "F" else: break # print the strings to the output file, 50 characters at a time out = open(filename, 'w') for i in range(0, len(alignS1), 50): out.write("%s\n%s\n\n" % (alignS1[i:i+50], alignS2[i:i+50])) out.close() return### End of functions ######################################################################################### Testing ##########################################################################################if \_\_name\_\_ == "\_\_main\_\_": ## Calculate global alignment score of two sequences #s = "AGCGTCTA" #t = "TGCATCTCG" ## longer test, from lab sheet #s = "ATGTTGAAGTCAGCCGTTTATTCAATTTTAGCCGCTTCTTTGGTTAATGCAGGTACCATACCCCTCGGAAAGTTATCTGACATTGACAAAATCGGAACTCAAACGGAAATTTTCCCATTTTTGGGTGGTTCTGGGCC" #t = "ATGTTTTCCCGCAGTCGCTGTGGTTCACTTGTAACAAGTGTGGCTCGCAAAATGTGGAACCACCCAAGCCAGCGCTGGCTCATCTTGATCTGCGTTATATGTTTGCTGTCTTTTGCGCTGGCC" #s = "AATTGGCTATGCTCGTTACGCATTACTAATCCGGTCAGGCTATGGGTGCTATGCTCGTTACGCA" #t = "AAATATGGTGCTGGCTGCTTCAGTTATGAACTCCCCAGGCTCATTGGTGCTGGCTGCTTACA" #s = "TATGCT" #t = "TGACAGT" s = convertFileToSequence("yeast\_random.txt") t = convertFileToSequence("fruit\_fly\_random.txt") optimalScore = globalAlignmentScore(s, t) print(s) print(t) print("Global alignment score: " + str(optimalScore))

#!/usr/bin/python# Caleb Piekstra and Sara Perkins# CS 423, Lab 5# fall 2015##################################################################### convertFileToSequence - takes a FASTA file and returns the# sequence as a string#####################################################################def convertFileToSequence(filename): # read in file file = open(filename) # read in first line header = file.readline() if (header[0] == '>'): print("in FASTA format") else: print("invalid format") file.close() return # read in rest of file sequence = file.read() # close file file.close() # remove all return and newline characters sequence = sequence.replace("\r", "") sequence = sequence.replace("\n", "") return sequence####################################################################### localAlignmentScore - Determines the score of the optimal local # alignment of two strings######################################################################def localAlignmentScore(s1, s2, filename=None): # add a blank string as padding s1 = ' ' + s1 s2 = ' ' + s2 # Scoring system MATCH = 5 MISMATCH = -4 GAP = -6 # Max value of alignment found maxValue = 0 # To keep track of the location of the maximum alignment maxRowPos = 0 maxColPos = 0 # set table size NUM\_ROWS = len(s2) NUM\_COLS = len(s1) # Create table and fill it with zeros c = createTable(NUM\_ROWS, NUM\_COLS, 0) # Creates table for getting back the optimal alignment, fill table with "F" # uses "D", "L", and "T" for diagonal, left, and top d = createTable(NUM\_ROWS, NUM\_COLS, "F") # The above automatically sets the gap penalties in left column and top row # implements dynamic programming algorithm for local alignment # fills in entries in cost table and direction table # update table to show alignment penalties for i in range(1, NUM\_ROWS): for j in range(1, NUM\_COLS): # set value of match/mismatch m = MATCH if s1[j] == s2[i] else MISMATCH # calculate costs based on direction left = c[i][j-1] + GAP top = c[i-1][j] + GAP diag = c[i-1][j-1] + m # set cost cost = max(0, left,top,diag) c[i][j] = cost # set direction based on cost # Precedence: F > D > T > L if cost == 0: d[i][j] = "F" elif cost == diag: d[i][j] = "D" elif cost == top: d[i][j] = "T" else: d[i][j] = "L" # update max value and its position if cost > maxValue: maxValue = cost maxRowPos = i maxColPos = j ## # Prints out table (only useful for small tables - used for debugging)## # Commented out, satisfied that the algorithm is working## printTable(c, "costs.txt")## printTable(d, "directions.txt") # find optimal alignment # if filename: # align(d, s1, s2, maxRowPos, maxColPos, filename) # else: # align(d, s1, s2, maxRowPos, maxColPos, "alignment.txt") # return optimal score (lower right-hand cell in table] return c[maxRowPos][maxColPos]##################################################################### Creates a 2D table with the given number of rows and columns# and fills all entries with value given as a parameter# (function completed by Tammy VanDeGrift)####################################################################def createTable(numRows, numCols, value): table = [] row = 0 # create 2D table initialized with value while (row < numRows): table.append([]) col = 0 while (col < numCols): table[row].append(value) col = col + 1 row = row + 1 return table################################################################### Prints 2D table to file (only useful for small tables for short# strings), can be used for integer as well as char values# tabs between the values on each row# Useful function for debugging purposes only##################################################################def printTable(table, filename): NUM\_ROWS = len(table) NUM\_COLS = len(table[0]) # write table to output file, one row at a time with open(filename, 'w') as out: for i in range(0, NUM\_ROWS): row = "" for j in range(0, NUM\_COLS): row += str(table[i][j]) + '\t' out.write(row+'\n') return################################################################# Reconstructs the optimal alignment and prints the alignment# to a file. Because the sequences can be long, prints the# alignment 50 characters on one line, the other string of 50 characters# on the next line, and then skips one line, as follows:# AATT--GGCTATGCT--C-G-TTACGCA-TTACT-AA-TCCGGTC-AGGC# AAATATGG---TGCTGGCTGCTT---CAGTTA-TGAACTCC---CCAGGC## TATGGGTGCTATGCTCG--T--TACG-CA# TCAT--TGG---TGCTGGCTGCTT--ACA## direction is a 2D table, s1 and s2 are the original DNA# sequences to align, maxRow and maxCol are the position of the# max score in the local alignment table and filename is the# name of the output file###############################################################def align(direction, s1, s2, maxRow, maxCol, filename): # initialize rows, columnss, and newly aligned sequences alignS1 = "" alignS2 = "" # evalutate table for all valid indices while(maxRow >= 0 and maxCol >= 0): d = direction[maxRow][maxCol] # start with bottom right # diagonal if d == "D": alignS1 = s1[maxCol] + alignS1 alignS2 = s2[maxRow] + alignS2 maxRow -= 1 maxCol -= 1 # top elif d == "T": alignS1 = "-" + alignS1 alignS2 = s2[maxRow] + alignS2 maxRow -= 1 # left elif d == "L": alignS1 = s1[maxCol] + alignS1 alignS2 = "-" + alignS2 maxCol -= 1 # first, "F" else: break # print the strings to the output file, 50 characters at a time with open(filename, 'w') as out: for i in range(0, len(alignS1), 50): out.write("%s\n%s\n\n" % (alignS1[i:i+50], alignS2[i:i+50])) return### End of functions ######################################################################################### Testing ##########################################################################################if \_\_name\_\_ == "\_\_main\_\_": ## Calculate local alignment score of two sequences #s = "AGCGTCTA" #t = "TGCATCTCG" ## longer test, from lab sheet #s = "ATGTTGAAGTCAGCCGTTTATTCAATTTTAGCCGCTTCTTTGGTTAATGCAGGTACCATACCCCTCGGAAAGTTATCTGACATTGACAAAATCGGAACTCAAACGGAAATTTTCCCATTTTTGGGTGGTTCTGGGCC" #t = "ATGTTTTCCCGCAGTCGCTGTGGTTCACTTGTAACAAGTGTGGCTCGCAAAATGTGGAACCACCCAAGCCAGCGCTGGCTCATCTTGATCTGCGTTATATGTTTGCTGTCTTTTGCGCTGGCC" #s = "AATTGGCTATGCTCGTTACGCATTACTAATCCGGTCAGGCTATGGGTGCTATGCTCGTTACGCA" #t = "AAATATGGTGCTGGCTGCTTCAGTTATGAACTCCCCAGGCTCATTGGTGCTGGCTGCTTACA" #s = "TATGCT" #t = "TGACAGT" #s = "TGGTAGATTCCCACGAGATCTACCGAGTATGAGTAGGGGGACGTTCGCTCGG" #t = "GCCTCTAACACACTGCACGAGATCAACCGAGATATGAGTAATACAGCGGTACGGG" s = convertFileToSequence("yeast\_random.txt") t = convertFileToSequence("fruit\_fly\_random.txt") #t = convertFileToSequence("flyPHO12.txt") optimalScore = localAlignmentScore(s, t, "PHO12\_local\_alignment.txt") print(s) print(t) print("Local alignment score: " + str(optimalScore))

#!/usr/bin/python# Caleb Piekstra and Sara Perkins# CS 423, Lab 5# fall 2015##################################################################### convertFileToSequence - takes a FASTA file and returns the# sequence as a string#####################################################################def convertFileToSequence(filename): # read in file file = open(filename) # read in first line header = file.readline() if (header[0] == '>'): print("in FASTA format") else: print("invalid format") file.close() return # read in rest of file sequence = file.read() # close file file.close() # remove all return and newline characters sequence = sequence.replace("\r", "") sequence = sequence.replace("\n", "") return sequence ####################################################################### LCSScore - Determines the score of the optimal alignment of # two strings (s1 and s2)######################################################################def LCSScore(s1, s2, filename=None): # add a blank string as padding s1 = ' ' + s1 s2 = ' ' + s2 # set table size NUM\_ROWS = len(s2) NUM\_COLS = len(s1) # Create table and fill it with zeros c = createTable(NUM\_ROWS, NUM\_COLS, 0) # Creates table for getting back the optimal alignment, fill table with "F" # uses "D", "L", and "T" for diagonal, left, and top d = createTable(NUM\_ROWS, NUM\_COLS, "F") # The above automatically sets the gap penalties in left column and top row # implements dynamic programming algorithm for LCS # fills in entries in cost table and direction table # update table to show alignment penalties for i in range(1, NUM\_ROWS): for j in range(1, NUM\_COLS): # calculate costs based on direction left = c[i][j-1] top = c[i-1][j] diag = c[i-1][j-1] + 1 # set cost if s1[j] == s2[i]: cost = max(left, top, diag) else: cost = max(left, top) c[i][j] = cost # set direction based on cost # Precedence: L > T > D if cost == left: d[i][j] = "L" elif cost == top: d[i][j] = "T" else: d[i][j] = "D" # Prints out table (only useful for small tables - used for debugging) # Commented out, satisfied that the algorithm is working # printTable(c, "costs.txt") # printTable(d, "directions.txt") # find optimal alignment if filename: align(d, s1, s2, filename) else: align(d, s1, s2, "alignment.txt") # return optimal score (lower right-hand cell in table] return c[NUM\_ROWS-1][NUM\_COLS-1]##################################################################### Creates a 2D table with the given number of rows and columns# and fills all entries with value given as a parameter# (function completed by Tammy VanDeGrift)####################################################################def createTable(numRows, numCols, value): table = [] row = 0 # create 2D table initialized with value while (row < numRows): table.append([]) col = 0 while (col < numCols): table[row].append(value) col = col + 1 row = row + 1 return table################################################################### Prints 2D table to file (only useful for small tables for short# strings), can be used for integer as well as char values# tabs between the values on each row# Useful function for debugging purposes only##################################################################def printTable(table, filename): NUM\_ROWS = len(table) NUM\_COLS = len(table[0]) # write table to output file, one row at a time with open(filename, 'w') as out: for i in range(0, NUM\_ROWS): row = "" for j in range(0, NUM\_COLS): row += str(table[i][j]) + '\t' out.write(row+'\n') return################################################################# Reconstructs the LCS and prints it to a file.# Because the sequences can be long, prints the# alignment 50 characters on one line, the other string of 50 characters# on the next line, and then skips one line, as follows:# AATT--GGCTATGCT--C-G-TTACGCA-TTACT-AA-TCCGGTC-AGGC# AAATATGG---TGCTGGCTGCTT---CAGTTA-TGAACTCC---CCAGGC## TATGGGTGCTATGCTCG--T--TACG-CA# TCAT--TGG---TGCTGGCTGCTT--ACA## direction is a 2D table, s1 and s2 are the original DNA# sequences to align and filename is the name of the output file###############################################################def align(direction, s1, s2, filename): # initialize rows, columns, and newly aligned sequences row = len(direction) -1 col = len(direction[0]) -1 lcs = "" # evalutate table for all valid indices while(row >= 0 and col >= 0): d = direction[row][col] # start with bottom right # diagonal, only time add character if d == "D": lcs = s1[col] + lcs row -= 1 col -= 1 # top, update row elif d == "T": row -= 1 # left, update column elif d == "L": col -= 1 # first, "F" else: break # print the strings to the output file, 50 characters at a time out = open(filename, 'w') for i in range(0, len(lcs), 50): out.write("%s\n" % (lcs[i:i+50])) out.close() return### End of functions ######################################################################################### Testing ##########################################################################################if \_\_name\_\_ == "\_\_main\_\_": ## Calculate LCS score of two sequences #s = "AGCGTCTA" #t = "TGCATCTCG" ## longer test, from lab sheet #s = "ATGTTGAAGTCAGCCGTTTATTCAATTTTAGCCGCTTCTTTGGTTAATGCAGGTACCATACCCCTCGGAAAGTTATCTGACATTGACAAAATCGGAACTCAAACGGAAATTTTCCCATTTTTGGGTGGTTCTGGGCC" #t = "ATGTTTTCCCGCAGTCGCTGTGGTTCACTTGTAACAAGTGTGGCTCGCAAAATGTGGAACCACCCAAGCCAGCGCTGGCTCATCTTGATCTGCGTTATATGTTTGCTGTCTTTTGCGCTGGCC" #s = "AATTGGCTATGCTCGTTACGCATTACTAATCCGGTCAGGCTATGGGTGCTATGCTCGTTACGCA" #t = "AAATATGGTGCTGGCTGCTTCAGTTATGAACTCCCCAGGCTCATTGGTGCTGGCTGCTTACA" #s = "TATGCT" #t = "TGACAGT" #s = "TGGTAGATTCCCACGAGATCTACCGAGTATGAGTAGGGGGACGTTCGCTCGG" #t = "GCCTCTAACACACTGCACGAGATCAACCGAGATATGAGTAATACAGCGGTACGGG" s = convertFileToSequence("yeastPHO12.txt") t = convertFileToSequence("flyPHO12.txt") LCS = LCSScore(s, t, "PHO\_LCS.txt") print(s) print(t) print("Longest Common Subsequence: " + str(LCS))

# Caleb Piekstra and Sara Perkins

# CS423

# Lab 5: genRandom

#!/usr/bin/python

import random

#############

# genRandom -- generates a random sequence of input length

# with a choosing rate provided in the form of percent GC and AT

# content desired, writes the sequnece to a file

#############

def genRandom(lenStr, percentAT, percentCG, outputFileName):

# open file, set fasta header

with open(outputFileName, 'w') as out:

out.write("> random sequence\r\n")

# write the sequence to the file

for i in range(0, lenStr):

ran = random.random()

# choose the nucleotide based on the given desired content

# and random number produced

if ran < percentAT/2:

out.write("A")

elif ran < percentAT:

out.write("T")

elif ran < percentAT + (percentCG / 2):

out.write("C")

else:

out.write("G")

######## code for yeast and fruitfly needed for writeup

#genRandom(1404, 1 - 0.38, 0.38, "yeast\_random.txt")

#genRandom(1368, 1 - 0.42, 0.42, "fruit\_fly\_random.txt")

#!/usr/bin/python

###################################################################

# CS 423 lab 5 starter code for experiment

# This code should generate two randomly composed DNA sequences of various

# lengths and run the globalAlignment and localAlignment algorithms

# Authors: Sara Perkins, Caleb Piekstra

####################################################################

import globalAlignment, random

import localAlignment

# s = "AGAAAAAAAACTA"

# t = "TGCATCAAAG"

# optimalScore = globalAlignment.globalAlignmentScore(s, t)

# print ("Global alignment score: " + str(optimalScore))

######################################################################

# randomSeq - generates randomly composed DNA sequences

# (25%A, 25%C, 25%T, 25%G) of length N and returns it as a string

######################################################################

def randomSeq(N):

seq = ""

for i in range(0, N):

ran = random.random()

if ran < 0.25:

seq += "A"

elif ran < 0.50:

seq += "T"

elif ran < 0.75:

seq += "C"

else:

seq += "G"

return seq

######################################################################

# experiment - runs the following experiment:

# From size Start to Stop going by Step:

# Generate 2 random DNA strings of length size

# Calculate the global alignment score between the two strings

# Calculate the local alignment score between the two strings

# Write results to file per line

# such as:

# 5 -4 10

# where 5 is the length of the two strings, -4 is the optimal

# global alignment score, and 10 is the optimal local alignment

# score. Put a newline after each line and separate values per line

# by tabs. This output file will be used for graphing.

######################################################################

def experiment(Start, Stop, Step, Filename):

with open(Filename, 'w') as out:

for size in range(Start, Stop, Step):

ranSeq1 = randomSeq(size)

ranSeq2 = randomSeq(size)

g = globalAlignment.globalAlignmentScore(ranSeq1, ranSeq2)

l = localAlignment.localAlignmentScore(ranSeq1, ranSeq2)

out.write("%d\t%d\t%d\n" % (size, g, l))

return

### End of functions ###################################

###################################################

### Testing #######################################

###################################################

# run experiment

experiment(5, 100, 5, "expOutput.txt")

# Caleb Piekstra and Sara Perkins

# CS423

# Lab 5: Timing Experiment

import globalAlignment, random

#import localAlignment ## comment out once you have this module working

import time # to calculate running times

#######

# randomSeq

# generates randomly composed DNA sequences (25%A, 25%C, 25%T, 25%G)

# of length N and returns it as a string

# returns the sequence

#######

def randomSeq(N):

seq = ""

for i in range(0, N):

ran = random.random()

if ran < 0.25:

seq += "A"

elif ran < 0.50:

seq += "T"

elif ran < 0.75:

seq += "C"

else:

seq += "G"

return seq

########

# Timing experiment

# creates two random sequences of same length

# calculates time to find the optimal score

# writes the results to a file

# repeats for sequences of length 50 to 5000 stepping by 50

########

with open("longTimingExperiment.txt", 'w') as out:

for N in range(50, 5001, 50):

# generate two random sequences

ranSeq1 = randomSeq(N)

ranSeq2 = randomSeq(N)

# start clock and find global score

t0 = time.clock()

optimalScore = globalAlignment.globalAlignmentScore(ranSeq1, ranSeq2)

t1 = time.clock()

# print the time results to the file

totalTime = str(t1-t0)

print("Seq Len: %d\tTotal time: %s" % (N, totalTime))

out.write("%d\t%s\n" % (N, totalTime))