**CS423 Lab 5: String Alignment**

**Due:** Friday, Oct 9, 2015 [beginning of class]

**Note:** First midterm exam is Wednesday, Oct 14.

**Purpose:** This lab explores the string alignment dynamic programming algorithms (global alignment, local alignment) and longest common subsequence algorithm.

**Directions:** You should complete this lab in pairs, and you should take turns as the “driver” and “navigator” in pair programming. Choose a new partner so that one person has more experience in biology and one person has more experience in computer science. You should swap roles at least as often as every ten minutes. All submitted work should be typed and include your name(s). Submit your code files, your output files, and write-up (as a single .zip file) *electronically* to Moodle.

**Background Information:** Review your course notes and the textbook (chapter 2) regarding the string alignment problem.

For each of the problems below, let c[i,j] be the alignment cost for aligning the first j characters of string S and the first i characters of string T. Thus, i is the row index and j is the column index in the tables below. Let d[i,j] store the direction from which the best alignment score for each table entry c[i,j] was calculated.

Let g = penalty for insertion/deletion (more generally, a gap)

Let m = score for match/mismatch of aligning one character from S with one character from T

Standard alignment score values: g = -6, match = 5, mismatch = -4

***Variant #1:* Global string alignment**: Given two strings S and T, what is the optimal alignment and score for aligning the full content of both strings?

Algorithm for calculating global alignment score:

[S is along the top of the tables, T is on the left side of the tables. p = |T| and q = |S|.]

c[0][0] = 0 // cost to align two empty DNA sequences

d[0][0] = “F” // start location of alignment (“F” for first)

for i from 1 to p // fill in gap penalties in left column, p is length of string T

c[i][0] = g\*i // note that this index i is one larger than the actual string indexes in

// python

d[i][0] = “T” // “T” for top

for j from 1 to q // fill in gap penalties in top row, q is length of string S

c[0][j] = g\*j // note that this index j is one larger than the actual string indexes in

// python

d[0][j] = “L” // “L” for left

for i from 1 to p // fill in table left to right, top to bottom

for j from 1 to q

c[i][j] = max(c[i][j-1] + g, c[i-1][j] + g, c[i-1][j-1] + m)

d[i][j] = “L”, “T”, or “D” // wherever max value came from, “D” for diagonal

return c[p][q]

To reconstruct global alignment:

direction = d[p][q]

If direction == “T”, align corresponding character in T with -.

If direction == “L”, align corresponding character in S with -.

If direction == “D”, align corresponding character in S with corresponding character in T.

Continue retrieving values from table d (going up, left, or diagonally, as appropriate) until you reach position [0,0], which should contain “F”. At that point, the alignment should be constructed.

Note that the constructing the alignment processes the two strings from back to front (in other words, the constructed alignment is done in reverse order).

***Variant #2:* Local string alignment**: Given two strings S and T, what is the optimal alignment between **any substring** S’ of S and **any substring** T’ of T?

Algorithm for calculating local alignment score:

c[0][0] = 0 // cost to align two empty strings

d[0][0] = “F” // start location of alignment

maxValue = 0 // max value of alignment found

maxRowPos = 0 // to keep track of the location of the maximum alignment

maxColPos = 0

for i from 1 to p // fill in gap penalties in left column

c[i][0] = 0

d[i][0] = “F”

for j from 1 to q // fill in gap penalties in top row

c[0][j] = 0

d[0][j] = “F”

for i from 1 to p // fill in table left to right, top to bottom

for j from 1 to q

c[i][j] = max(0, c[i][j-1] + g, c[i-1][j] + g, c[i-1][j-1] + m)

d[i][j] = “F”, “L”, “T”, or “D” // wherever max value came from

// Use “F” if max is 0, starting new substring

if(c[i][j] > maxValue)

maxValue = c[i][j]

maxRowPos = i

maxColPos = j

return c[maxRowPos][maxColPos]

To reconstruct local alignment: Start at d[maxRowPos][maxColPos] and follow same reconstruction idea as in variant 1 until you reach a cell containing “F”. Note: this local alignment will align a *substring* of S with a *substring* of T in reverse order.

***Variant #3:* Longest common subsequence:** Given strings S and T, what is the longest common subsequence between them? A subsequence S\* of S is a sequence of characters from S where the order is maintained. For example, if S = “ATGCGTTAAT”, then “ATCGTAT” is a valid subsequence.

Algorithm for calculating the length of the longest common subsequence:

c[0][0] = 0 // no alignment

d[0][0] = “F” // start location of LCS

for i from 1 to p // fill in left column

c[i][0] = 0

d[i][0] = “F”

for j from 1 to q // fill top row

c[0][j] = 0

d[0][j] = “F”

for i from 1 to p // fill in table left to right, top to bottom

for j from 1 to q

if T[i] == S[j] // have a match

c[i][j] = max(c[i][j-1], c[i-1][j], c[i-1][j-1] + 1)

else

c[i][j] = max(c[i][j-1], c[i-1][j])

d[i][j] = “L”, “T”, or “D” – wherever max value came from

return c[p][q]

To reconstruct LCS:

Start in lower right corner in table d. Follow the “L”, “T”, and “D” directions. If alignment has “D”, the corresponding character is part of the LCS. Continue traversing table d until you reach a cell containing “F”.

Note that the LCS is just a single subsequence that is common to the two strings, so it is not an alignment between two strings. The reconstruction is done in reverse order, as in the two above algorithms.

**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)? \_\_\_\_\_\_\_\_

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)? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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)? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

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

*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)? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

*Switch driver and navigator*

***Global alignment via dynamic programming***

8. Download the starter code (globalAlignment.py). Complete the following functions:

* 1. globalAlignmentScore – takes two strings as parameters and returns the maximum global alignment score (according to variant 1 above). Some parts of the function are already completed for you. Note that the max function built into python can take multiple parameters.
  2. printTable – take a 2D table and filename as parameters and prints contents of the table to a file with the given filename. When printing, use tabs to separate entries (\t). Each row of the table should be printed on a separate line in the output file.
  3. align – takes a 2D table (of direction entries), two strings, and an output file name as parameters and reconstructs the optimal alignment and writes the alignment to the output file (50 characters per line). A gap should be written as a single hyphen “-“. Put an extra newline in the output file between the two lines showing the alignment. Below is an example of how the first few lines in the output file should look:

ATGTT-----GAAGTCAGCCGTTTATTCAATTTTAGCC-GCT-T--CTTT

ATGTTTTCCCGCAGTC-GCTGTGG-TTCACTTGTAACAAG-TGTGGCTC-

GGTTAATGCAGGTACCATACCCCTCGGAAAGTTATCTGACATTGACAAAAGCAAAATGT-GGAACCA--CCC-----AA-GCCAGC-G-C-TGG-CTCA-

TCG-GAACTCAAACGGAA-AT-TTTCCCAT-TTTTGGGTGGTTCTGGGCCTCTTGATCTG---CGTTATATGTTTGCTGTCTTTTGCG-----CTGG-CC

9. Test your dynamic programming code on small strings first. You can verify the optimal score matches that of globalAlignmentSearch.py’s output when run on the same two input strings (use short strings, though, since the search implementation is quite slow). When you are satisfied that the code is working, comment out the printTable function calls in the globalAlignmentScore function, so you can start running the code on longer strings.

*Switch driver and navigator*

***Local alignment via dynamic programming***

10. Copy your globalAlignment.py file to a new file called localAlignment.py. Modify the code so that it calculates the optimal **local** alignment (variant 2 above) between two strings and produces a file containing the best **local** alignment of the two strings. In the output file, print 50 characters per line, with an extra space between each set of aligned strings, similar to the above format. You may want to pass two extra parameters to the align function to indicate the maxRow and maxCol in the direction table to know where to start the alignment.

11. Rename and comment your functions, as appropriate. It should not take much code modification from the global alignment program.

*Switch driver and navigator*

***Longest common subsequence via dynamic programming***

12. Copy your globalAlignment.py file to a new file called LCS.py. Modify the code so that it calculates the length of the longest common subsequence of two strings and produces a file containing the longest common subsequence. In the output file, print 50 characters per line (note that you can print the characters directly (since there are no dashes) and no need for extra blank lines since there is only one common subsequence reconstructed when navigating the direction table). You may want to pass two extra parameters to the align function to indicate the maxRow and maxCol in the direction table to know where to start extracting the common subsequence.

13. Rename and comment your functions, as appropriate. It should not take much code modification from the global alignment program.

**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?
   2. What is the optimal **global** alignment?
   3. What is the optimal **local** alignment score?
   4. What is the optimal **local** alignment?
   5. What is the length of the LCS?
   6. What is the LCS?
2. (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?
   2. Include the optimal **global** alignment of these two sequences as a text file in your zipped folder. Name the file PHO12\_global\_alignment.txt.
   3. What is the optimal **local** alignment score?
   4. Include the optimal **local** alignment of these two sequences as a text file in your zipped folder. Name the file PHO12\_local\_alignment.txt.
   5. What is the length of the LCS?
   6. Include the LCS as a text file in your zipped folder. Name the file PHO12\_LCS.txt.
3. (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?
   2. 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?
   3. 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)?
   4. 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)?
4. (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?
   2. When are the two scores (local and global) equal?
5. (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?
6. (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?
7. (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.
8. (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?
9. (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.
   4. Did the results of the graph match your expectations? Why or why not?
   5. 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.
10. (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.
    1. What is the big O running time of the dynamic programming algorithm for global alignment?
    2. How does the big O function match the experimental results?
    3. 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?)
    4. 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)?

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

The code and write-up submitted for this lab were authored by the named person(s) on this lab report. 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.