DNA Alignment

A strand of DNA contains a sequence of nucleotides: Adenine, Thymine, Cytosine, and Guanine. These nucleotides are commonly abbreviated: A, T, C, and G. The problem of local DNA alignment consists of finding the regions of similarity between strands of DNA from different organisms. That is, one can use this process to try and identify genes.

Local Alignment Algorithm

The basic idea is that you want to try and match every substring of string1 with every substring of string2, compute the matching cost for each, and decide which set of substrings has the best matching cost. However, computing every possible pair of substrings could be extremely costly. Luckily, there is a Dynamic Programming algorithm that can be used to solve this problem.

The basis of any Dynamic Programming solution is a recurrence relation. In order to describe the recurrence relation, I will present a small example.

Given two strings “ACTGC” and “ACG” we want to find the best cost substring from the start of each string. In the following, I will refer to the first position of each string as position 1 (rather than 0 as CS people are used to – we will be using position 0 in our tables for the empty string). I will also refer to an index in string1 as i and an index in string2 as j. So, the potential substrings that start at the 1st location in string1 are: empty, A, AC, ACT, ACTG, and ACTGC. I will refer to these substrings as i=0, 1, 2, 3, 4, and 5. That is, when i=3, that means the string from position 1 up to, and including, position 3 (ACT). Similar is true of string2 (only j will be used).

The easiest way to explain this is to start with the general case. Let’s say i=3 and j=2. Thus, we are talking about substrings ACT and AC. There are 3 possible ways to match the last characters in the substrings.

1. We can match the T in string1 to the C in string2:
   
   ACT
   AC
2. We can match the T in string1 to a newly added gap in the string2:
   
   \[
   \begin{align*}
   &\text{ACT} \\
   &\text{AC} - \\
   \end{align*}
   \]

3. Or we can match the C in string2 to a newly added gap in the string1:

   \[
   \begin{align*}
   &\text{ACT} - \\
   &\text{AC} \\
   \end{align*}
   \]

   The first case costs us the MatchCost from T to C plus the optimal cost of matching AC to A (i=2, j=1)

   The second case costs us the MatchCost from T to -- plus the optimal cost of matching AC to AC (i=2, j=2)

   The third case costs us the MatchCost from -- to C plus the optimal cost of matching ACT to A (i=3, j=1)

   One of these cases will be the highest cost. This is the correct match for the substrings i=3, j=2.

   Put into mathematical terms (a recurrence relation):

   \[
   \text{OptimalCost}(i, j) = \max \left( \text{MatchCost}(i, j) + \text{OptimalCost}(i-1, j-1), \right. \\
   \left. \text{MatchCost}(i, -) + \text{OptimalCost}(i-1, j), \right. \\
   \left. \text{MatchCost}(-, j) + \text{OptimalCost}(i, j-1) \right) 
   \]

   Divide and Conquer could be used to solve this problem, but there would be lots of repeated computations. Thus, Dynamic Programming is a better choice. For the Dynamic Programming solution, you will be building up the 2D table of OptimalCosts. Note that the recurrence relation implies that each table entry relies on the entries to its left, top, and upper left.

   Then to find the best alignment, you need to find the highest score in the table and trace back to a starting point. Note that you can’t just start at the lower right entry in the table because this entry relates to the last characters in both strings and thus implies a full match of both strings – we want the best substring.

   There is one last problem. The algorithm as described above always finds the cost of the substrings that start at the first character of the strings. What we really want is for the starting position to be at any position in the strings. However, noticing that if some prefix has a negative weight, then it is better not to include the prefix in the substring match at all easily solves this problem. For example:

   \[
   \begin{align*}
   &\text{AC} \\
   &\text{TC} \\
   \end{align*}
   \]
The best match here is simply C to C and not AC to TC. To exclude the bogus prefix, we notice that the prefix match of A to T has a negative optimal cost. Thus, it is better to just ignore this prefix as it can’t possibly add to the best cost solution. This is accomplished by including a zero in the “max” computation above. Thus, our table will contain no negative costs.

As an example of something to test, given the strings and match costs in the GUI pictures below, the optimal cost table that is produced should look like (* means empty string):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>A</th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

The Code

You need to create 3 classes: **AlignmentGUI, DNAMatcher, and LocalAlignment.**

The **AlignmentGUI** should look like the following:

![AlignmentGUI](image)

It should contain places for the user to enter their test strings, enter their individual nucleotide matching costs, a button to perform alignment, and a place for the results of
the alignment. The following picture shows the result of pressing the Align button with the given inputs:

![Local Alignment](image)

There are actually several optimal alignments. That is, there are several alignments that all have the same best overall matching cost. You only need to produce one of these optimal alignments. What your final alignment will look like is dependent on how you handle ties. The alignment shown above was generated with a tie-breaker precedence of change, insertion, deletion. You might want to use the same precedence so your results match mine. It also makes good sense from a genetic standpoint to pick change over adding gaps if there are ties. Also your alignment is dependent on which starting maximum value you use (if there is more than one). My code uses the first one it finds if you want to mimic mine in this manner as well.

Note that the output fields are simply JTextFields with a particular region “selected.” Note that you can select a region from your code via the Caret class. That is, you can ask the JTextField for a Caret, then use the setDot and moveDot methods to mark the starting and ending places in the selection, and finally use the setSelectionVisible method to highlight the selection.

Also note that the Match Costs are symmetric. That is CG is the same as GC. To enforce this symmetry, only the upper right half of the matrix may be edited. The bottom left half should automatically change whenever the top half changes.

The Align button should be in charge of gathering the inputs, starting the matching process, and displaying the results.

To make this program easier for you, I have decided to include the entire AlignmentGUI class. This was created a while back using a visual editor so most of the methods are computer generated. You may use it as is – no modifications are needed. Or, if you wish, you may build your own GUI.
The **DNAMatcher** class should contain 2 public methods:

```java
public DNAMatcher(Map<String, Integer> matchCost)
and
public LocalAlignment findLocalAlignment(String string1, String string2)
```

The first is a constructor that takes in a Map of the individual matching costs. It is the GUI's job to gather this information from the GUI and create the Map that is sent to this constructor.

The second is the main access method for the class. It takes 2 strings and matches them, returning the result in a LocalAlignment object. To complete the alignment task, this method needs to perform two main tasks: compute the local alignment tables and generate the actual alignment from those tables. Each of these tasks should be done in a private helper method (computeOptimalAlignmentTable and generateAlignmentFromTable, respectively).

The **LocalAlignment** class is a class that glues the final alignment information together. An alignment can't be returned as a simple String, because an alignment involves two strings. The LocalAlignment class should have the following constructor:

```java
public LocalAlignment(String string1, String string2,
int string1MatchStart, int string2MatchStart, int matchLength)
```

Where string1 and string2 are the full strings, complete with matched subsections. That is, junk prefixes, matched sections (complete with gaps), and junk suffixes. The ints simply record where the actual matching substrings start and end.

And, since we don’t really want the GUI to reach into this class and grab its pieces to display the output, we will instead provide a method that the GUI will call to get the final alignment displayed in the JTextFields it has created:

```java
public void showAlignment(JTextField string1Field, JTextField string2Field)
```

**Concurrent Execution**

Lastly, I want you to extend this project to build the table in a concurrent manner to make better use of multicore processors. That is, each entry in the table only relies on 3 previous values. You had to pick a way of getting from the starting conditions in the table to the ending point so that each time you filled in a value, the previous values were already computed. This was done in a single threaded manner, one position at a time.
Submission

You are to turn in:

- A printed picture of your GUI results for a sample alignment run. The specific alignment run will be given before the due date.
- A printed single page (max) document describing any problems that your code might still have if it is not complete. Or if it is complete, possible places for improvement in your code.
- A jar file of your entire package submitted to the 335 folder on the W drive.

You may work with a partner.