Due date: Thursday, March 09, at the beginning of the lab.

1 About this Assignment

1.1 Objectives of the assignment

This is another big lab. The objectives for this week are for you to

- get better at implementing methods to solve practical problems;
- learn how to use efficiently while loops and if statements.

Recomfitting thought of the day: Until about 10 minutes ago, I was still thinking that it would be a good idea to throw in asking you to implement your own classes on top of that.

Read very carefully the text of this assignment before rushing to type code. Read it completely through once. Do not stop at the first unclear sentence you encounter; sometimes things are explained and detailed a bit later. Then you should start all over again. This time stop to ask questions when a point remains fuzzy, code your solution, and move on to the next section.

1.2 Handouts

This week again, the sole handout is the text of this assignment as a .pdf file.

1.3 DNA, genes, and bioinformatics

If you have been following the news on scientific issues, you must have heard about efforts to “map the human genome.” You probably read every now and then that researchers have identified (or rather, believe that they have identified) a gene or a group of genes that regulate something (e.g. “the gene(s) for skin cancer,” “the gene(s) for drunk driving,” etc.). You may also have heard that bioinformatics—the application of computer science tools, techniques, and methods to solve biological problems such as gene identification—is currently a very hot research domain, industry, and job market. In our lab today we will see how we can apply what we have learned so far in this course to solve a simple bioinformatics problem.

All our genetic information is encoded into DNA molecules, which are present in the kernel of every single healthy cell in our bodies. Although a typical DNA molecule contains tens of
thousands of atoms, it is in fact the assembly of only four different types of *nitrogenous bases*: adenine (A), guanine (G), thymine (T), and cytosine (C). These bases assemble to form long *polynucleotide chains*. The famous “double helix” you have certainly heard about is formed by two such polynucleotide chains winding around each other. Each base on one chain bonds with the base facing it on the other chain, with the requirement that an A base can only bond with a T base on the other chain (and vice versa), while a G base can only bond with a C base on the other chain (and vice versa).

This means that if one of the chains is


then the other chain must be


so that they can assemble to form the double helix once the following bonds are established:

\[
\begin{array}{cccccccccccccc}
|   |   |   |   |   |   |   |   |   |   |   |   |   |
C & G & A & G & T & T & C & G & G & A & T & C & G & A
\end{array}
\]

It is therefore sufficient to know the composition of one of the chains to deduce that of the other chain. That chain being stored as a sequence of characters, or string, gene matching can be performed as a series of manipulations of String object. This is what we will do in this lab.

1.4 Sequence matching

One very important problem faced by university and industry researchers is the comparison of two fragments of DNA sequences. For example, the researcher may try to identify a gene fragment obtained from diseased cells by matching it against some of the public gene databases available on the Web. Because of gene mutations and differences between organisms, it is not always possible to find perfect matches, so the objective is rather to provides “scores” evaluating the quality of a match, and report the matches with the best scores.

For example, imagine that our “gene database” contains the following reference “genes” (stored as strings):


and that we want to find the best match in our database for the following fragment:


\(^1\)All the examples I show in this documents have very short sequences. The ones stored in real gene databases are a lot longer. That does not change the nature of the computations, just their duration.
We can try to compare our target string against the reference strings by shifting them relative to one another, and counting how many matches we find between the two sequence for that particular relative position. Figure 1 shows us the best matches found for each of the reference genes.

14 consecutive matches

<table>
<thead>
<tr>
<th>Target</th>
<th>Reference 1</th>
</tr>
</thead>
</table>

7 consecutive matches

<table>
<thead>
<tr>
<th>Target</th>
<th>Reference 2</th>
</tr>
</thead>
</table>

12 consecutive matches

<table>
<thead>
<tr>
<th>Target</th>
<th>Reference 3</th>
</tr>
</thead>
</table>

4 consecutive matches

<table>
<thead>
<tr>
<th>Target</th>
<th>Reference 3</th>
</tr>
</thead>
</table>

Figure 1: Best matches found for each of our three reference strings.

When we try to match our target against the first reference string (top part of Figure 1), we find that shifting the target by two “characters” relative to the reference string gives us the longest sequence of consecutive matches: 14 consecutive matches. This is also the best consecutive matching sequence over our reference database. Reference gene 1 would therefore be a good match candidate for our target. At the opposite, we have reference string 3 (bottom part of Figure 1), which only provides at most 4 consecutive matches, a really poor score.

Reference gene 3 (middle part of Figure 1) poses an interesting problem. On the one hand, its best consecutive matching sequence only scores a score of 12 (pretty good, but still lower than reference gene 1). On the other hand, we see that it contains another—shorter—sequence of consecutive matches (of length 7). If it was not for this single “mutation” in the middle of our sequence, we would have an outstanding matching score of 20!

What we see from this is that there is not a single “best” criterion for scoring matches in the database. In this assignment, we will be using three different scoring methods, and report the best matches found (and their score) for each of these criteria. The different scoring criteria will be explained later in this assignment.
2 What to Do, Part I: Create a Project

2.1 Create a project file

By now you know the drill. Don’t forget to ask for separate source and binaries folders when you create your project.

2.2 Implement a rudimentary main class

You are going to add code to this class as you advance in this assignment, but for the time being, your main class only needs to contain a `main` method that does the following:

- Request from the user the input of three reference strings (reference genes for our small “gene database;”
- Request from the user the input of a target string for which matching scores will be computed.

The user should enter the strings without the dash signs, *i.e.* GCTACG rather than G–C–T–A–C–G. I only use the dash signs to make strings easier to read.

As we did last week, it is probably a good idea for you to have two versions of the code: One that actually requests user input from the user (the final version) and one that uses “hard-coded” string assignments, for easier and quicker testing and debugging. Believe me, you are going to tire pretty soon of typing a bunch of GCACAT…. Plus, by using your hard-coded strings for which you know the correct results, you eliminate the risk of user input errors. Do not forget to remove this “debugging” code from your program before you hand it in for evaluation.

2.3 Input data validity check

Now that we know about conditional statements, we can at long last check if our user entered valid data. In this simplified gene matching problem, the validity check is very simple: input string should be composed exclusively of the characters ‘A’, ‘G’, ‘C’, and ‘T’. If another character is found in an input string, then that string is not valid.

Project 1. Write a method that takes as parameter a `String` object, and returns `true` if the string is a valid gene fragment and `false` otherwise.

Thoroughly test your data validation method. Once you are convinced that it does what it is supposed to do, then you can use it to request from the user the input of a valid string.

Project 2. Modify the code of your `main` method so that if the user enters an invalid input string, your program rejects that string and keep requesting a new input string from the user until a valid string is entered.
2.4 Rewrite your data input as a separate method

As a general rule, you should try to avoid writing very long methods. If the code of your method covers multiple pages of your screen or printout, then it is more difficult to find mistakes in the code. It is therefore preferable to try to regroup together as methods all large blocks of code that implement a specific behavior. This makes it easier for you to debug your program because you can first test separately each of the methods.

If we regroup together all our data input code in a method named `inputData()`, or any equally meaningful identifier, then when you read the code of your `main` method, 20 to 30 lines of code will be replaced a single invocation of `inputData()`. This makes it much easier to read the code to understand what it is doing. Anybody looking at your code can understand at a superficial level what is going on there, and only needs to go look at the actual input code if changes need to be made there.

**Project 3.** Modify the code of your main class so that your input code ("final" and "debugging" versions) now appears in a method that can be invoked from the main method.

Note: To do this, you will need to define "global" variables in your class (since a Java method can return at most one value). Don’t forget that in an application’s main class these variables must be declared with the `static` modifier. The compiler won’t let you do otherwise anyway, but at least you should remember why you get compilation error messages.

3 What to Do, Part II: Computing Matching Scores

3.1 Our three scoring criteria

We will be using three different criteria for attributing a score to each reference string against which we want to match our target.

1. length of the longest consecutive matching sequence,
2. total number of character matches,
3. total number of matches that belong to a consecutive alignment of at least $n$ characters
Let us assume that our target string is


and that we want to compute a matching score for each of our three scoring criteria for the following reference gene:


Using the first criterion, longest consecutive matching sequence, we find that the best alignment is obtained for the relative position of the strings shown in Figure 2, where we obtain 6 consecutive matches. Note that there are other matches for this alignment, shown in the figure by the dashed box. However, since we are only interested in the longest consecutive matching sequence, these will not be counted in the score. The reference string therefore scores a 6 for Criterion 1.

$$\begin{array}{c}
\text{6 consecutive matches} \\
\end{array}$$

Figure 2: The reference gene scores a 6 for Criterion 1.

If we follow scoring criterion 2, overall number of matches, then the best alignment is obtained for the relative position of the strings shown in Figure 3, where we get a total of 10 matching characters. Our reference string therefore scores a 10 for Criterion 2.

$$\begin{array}{c}
\end{array}$$

Figure 3: The reference gene scores a 10 for Criterion 2.

Criterion 3 relies on a user-selected threshold length. If for example we decide to ignore all consecutive matches of length less than 3 (because we consider that such matches are likely to be accidental), then the best alignment is obtained for the relative position of the strings shown in Figure 4, where we get two consecutive matching sequences of length 4 (thus longer than our threshold of 3). Note that we ignore the alignments indicated by the dashed boxes because their length is less than the threshold. Our reference string therefore scores an 8 for Criterion 3.

$$\begin{array}{c}
\end{array}$$

Figure 4: The reference gene scores a 8 for Criterion 3.

### 3.2 Implement methods for computing matching scores

Obviously, since you are going to compute matching scores for each criterion and each reference string, it would not make any sense not to implement these criteria as Java methods.
Project 4. Add to your main class three methods that each implement one of the three scoring criteria we have just discussed. Each method should receive two `String` objects as parameters and return the matching score for these two string as an `int` number.

Important: It is not because the above task tells you to write one method for each scoring criterion that all the computations must be done inside that single method. Always look for identical operations that are done in different parts of your code: Rather than having slightly different versions of basically the same piece of code (with risks to add bugs whenever you make a change to one of them), implement the required behavior as a method that will be invoked whenever the behavior is required. This way, you only have one version of the code for that behavior to debug.

3.3 Find the best match over the “gene database”

Now that you can compute the score of any reference gene against our target, you can compare the scores of the three reference genes of your small “gene database” and decide which one provides the best match for each criterion.

Project 5. Determine for each of our three criteria the reference string in our gene database that scores the highest.

Do I need to remind you that you should be on the lookout for blocks of code that should be written as methods?

4 What to Do, Part III: Get the Right Output

4.1 Gene database input

You program should ask the user to enter 3 strings for the gene database. If the user enters an invalid string, he/she should be asked to enter it again until he/she gets it right. When the three strings have been entered, your program should print them to the console or to a message dialog window, with additional dashes added between the characters, to improve the readability.

For example, if the user entered

```
GCCTATCGCCT
AATCGATG
```
then your program should print to the console:

The three reference strings stored in the gene database are
G-C-C-T-A-T-C-G-C-C-T
C-C-G-A-C-T-G

4.2 Target string input

Your program should ask the user to input a target string. Again, if the string is not valid, your program should keep asking the user for a valid string.
then your program should print to the console:

Your target sequence is
A-T-C-G-C-T

After that, your program should ask the user to enter a threshold length for Criterion 3.

4.3 Result output

Having all the data it needs to compute matches, your program should now find the best match for each criterion, as well as the score achieved by that best match.

The best match for Criterion 1 is Gene 1, with a score of 6
The best match for Criterion 2 is Gene 1, with a score of 12
The best match for Criterion 3 is Gene 1, with a score of 10

4.4 Keep asking the user for a target string

After your program has computed the best matches for the target string entered by the user, your program should ask again for a target string and a threshold length for Criterion 3. It should do so until the user types !QUIT instead of a valid input string.

5 Extra Credit

5.1 Extra credit 1 [10 pts]

Once you know how to determine which reference gene is the best match for given criterion, it would be helpful to the user if you could display as well how this score was achieved. Let us assume that our target string was the same as in our previous examples,

and that we the best match for Criterion 2 is obtained for Reference Gene 1 (out of 3 genes in our database),

\[ A\rightarrow A\rightarrow G\rightarrow C\rightarrow T\rightarrow C\rightarrow T\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow T\rightarrow G\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow G\rightarrow T\rightarrow T\rightarrow G\rightarrow C\rightarrow G\rightarrow A\rightarrow T. \]

Then you should printout the following result for Criterion 2:

**Criterion 2 ():**

Best match obtained for reference gene 1: Score = 8

Target:  
\[ G\rightarrow G\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow T\rightarrow A\rightarrow C\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow G\rightarrow C\rightarrow G\rightarrow C\rightarrow G\rightarrow C\rightarrow T\rightarrow G \]

Reference:  
\[ A\rightarrow A\rightarrow G\rightarrow C\rightarrow T\rightarrow C\rightarrow T\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow T\rightarrow G\rightarrow T\rightarrow A\rightarrow G\rightarrow C\rightarrow G\rightarrow T\rightarrow T\rightarrow G\rightarrow C\rightarrow G\rightarrow A\rightarrow T \]

Please note the vertical bars between matching characters.

Similar output should be provided for the other two criteria.

5.2 **Extra credit 2 [2pts]**

We have already used one command in our assignment, with `!QUIT` as input sequence to stop the execution of the program (Quit). To get the extra credit points, you should add the following commands:

- `!TARGET` should get the application to print a reminder of the target string used.
- `!REFERENCE` should get the application to print a reminder of the contents of the gene database (our three reference strings)

5.3 **Extra credit 3 [5pts]**

We saw earlier that we can deduce the composition of one chain of the DNA molecule if we know that of the other chain. Now, we only store one chain of our reference DNA in our gene database. What if the target string we are trying to find a match for actually matches the dual chains of the ones stored in the gene database? Solve this problem.

5.4 **Extra credit 4 [5pts]**

Very similar to the above problem. The order in which we list the strings should not matter: `GCTATCAG` should be the same as `GACTATCG`. Modify your program so that it checks for this as well.
6 What to Hand in

6.1 End-of-session evaluation

You are not expected to complete the assignment by the end of the lab session, but you are definitely expected to have done some work during that session. Try to use the lab session to make sure you understand everything about the assignment. Ask questions; try things; ask more questions.

You should not leave the lab before your work has been evaluated. This first evaluation is worth 10 pts out of 100 for the complete assignment. If you leave before you have been evaluated, these points are lost with no chance of a later evaluation.

6.2 Your project

You should drop in your CSC211s1 folder a folder named Lab04.<your name>. That folder should contain the src (source) folder for your Eclipse project and your report as a pdf, doc, or rtf file. Do not post the documentation produced by javadoc. We can produce that ourselves from your source code.

6.3 Printed copy

You should hand in a printed copy of your report and of your source code at the beginning of the next lab session. If your report is not ready at the beginning of the session, a late penalty will be applied. There is no use typing the report during the lab since the penalty is the same whether you return the report at the end of the lab or the next day at the beginning of the class.

7 How You Will Be Evaluated

7.1 Point distribution

The maximum number of points is 100, but extra points could be awarded for excellent aspects of the project or report. The point distribution for this assignment is as follows:

<table>
<thead>
<tr>
<th>Execution evaluation</th>
<th>10 pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-of-session evaluation</td>
<td>10 pts</td>
</tr>
<tr>
<td>Execution of the project handed in</td>
<td>30 pts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier names</td>
<td>10 pts</td>
</tr>
<tr>
<td>Good indentation and general readability</td>
<td>10 pts</td>
</tr>
<tr>
<td>Judicious comments well positioned in the code</td>
<td>15 pts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Report</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>List of the program’s main variables and their use</td>
<td>5 pts</td>
</tr>
<tr>
<td>Discussion</td>
<td>10 pts</td>
</tr>
<tr>
<td>General quality of the writing and presentation</td>
<td>10 pts</td>
</tr>
</tbody>
</table>
7.2 Various point penalties

Hopefully we won’t have to apply many of these:

- Project left accessible on the workstation: -5 pts
- Project folder incomplete or not properly cleaned up: -5 pts
- Report file missing from the project folder: -5 pts

**Late penalties**
- Printed copy of the report, 1 day late: -5 pts
- Project folder (uploaded to EnVision server), per day late: -10%

If you submit a project late, then it is your responsibility to notify the TA (with CC. to the instructor) that the project is finally available for download on the EnVision server. If you fail to do so, then the “late penalty clock” will keep ticking until the TA gets around to checking your folder on the EnVision server and notices your project. Unless specifically asked to do so, do not mail your project folder as an attachment.