CSCI 1300 Introduction to Computer Programming
Instructor: Knox
Assignment 4
Due Sunday, October 1, by 6:00 pm
Submit by Saturday, September 30 at 6:00 PM for a +2% bonus
Submit by Friday, September 29 at 6:00 PM for a +5% bonus

This assignment requires you to write two files: *main.cpp* and *Assignment4.cpp*. There are three functions that you are required to write for this assignment. These should be written in a file called *Assignment4.cpp*. If you write more functions than the three that are required, these should be kept in Assignment4.cpp as well. Your main.cpp file should just implement your function calls to test the functionality of your solution. Linking the two files can be done with **#include "Assignment4.cpp"** in your main file.

Once you have your code running on your virtual machine (VM), you must zip your *Assignment4.cpp* file and main.cp file into a .zip file and submit that file to the autograder COG. You **must also** submit your code (both main.cpp and Assignment4.cpp in the same .zip file) to Moodle to get full credit for the assignment.

Submitting Your Code to the Autograder:
Before you submit your code to COG, make sure it runs on your computer. If it doesn't run on the VM, it won't run on COG. The computer science autograder, known as COG, can be found here: **https://web-cog-csci1300.cs.colorado.edu**

- Login to COG using your identikey and password.
- Select the CSCI1300 - Assignment 4 from the dropdown.
- Upload your .zip both files files and click Submit.

Your file within the .zip must be named Assignment4*.cpp* for the grading script to run. *Your main.cpp file is not needed in your submission to COG.* COG will run its tests and display the results in the window below the *Submit* button. If your code doesn't run correctly on COG, read the error messages carefully, correct the mistakes in your code, and upload a new file. You can modify your code and resubmit as many times as you need to, up until the assignment due date.

Submitting Your Code to Moodle:
You must also submit your code to Moodle to get full credit for the assignment, even if the computer science autograder gives you a perfect score.

Comments at the top of your source files should include your name, recitation TA, and the assignment number. **TA's may deduct 20 points if code is found to be not commented or formatted with proper indentation. See the explanation of the required comments in the *CodingStyle* document on Moodle.**

Upload one .zip file to Moodle containing two files: your own main.cpp *and* Assignment4.cpp *which was sent to COG.*

Homework Problem:

This problem will be comparing and measuring the differences between DNA sequences. This type of analysis is common in the Computational Biology field. There are three parts to this assignment which will ask you to implement respectively the following functions:

- *similarityScore(...)*
- *countMatches(...)*
- *findBestMatch(...), findBestGenome(...)*

You are free to choose the names of any other supporting functions. The names should be reflective of the function they perform.

In the writeup, when referring to a function we often remove the details of the parameters that the function accepts; i.e. in *similarityScore(...) shown above.* We do this for brevity. The parameters are described in detail in the problem descriptions listed below.

Measuring DNA Similarity

DNA is the hereditary material in humans and other species. Almost every cell in a person's body has the same DNA. All information in a DNA is stored as a code in four chemical bases: adenine (A), guanine (G), cytosine (C) and thymine (T). The differences in the order of these bases provide a means of specifying different information.

In the lectures there have been examples of string representation and use in C++. In this assignment we will create strings that represent the possible DNA sequences. DNA sequences use a limited alphabet (A,C,G,T) within a string. Here we will be implementing a number of functions that are used to search for substrings that may represent genes and protein sequences within the DNA.

One of the challenges in computational biology is determining where a DNA binding protein will bind in a genome. Each DNA binding protein has a preference for a specific sequence of nucleotides. This preference sequence is known as a motif. The locations of a motif within a genome are important to understanding the behavior of a cell's response to a changing environment.

To find each possible location along the DNA, we must consider how well the motif matches the DNA sequence at each possible position. The protein does not require an exact match to bind and can bind when the sequence is similar to the motif.

Another common DNA analysis function is the comparison of newly discovered DNA genomic sequences to the large databases of known DNA sequences and using the similarity of the sequences to help identify the origin of the new sequences.

Hamming distance and similarity between two strings

Hamming distance is one of the most common ways to measure the similarity between two strings of the same length. Hamming distance is a position-by-position comparison that counts the number of positions in which the corresponding characters in the string are different. Two strings with a small Hamming distance are more similar than two strings with a larger Hamming distance.

Example: first string = "ACCT" second string = "ACCG"

```
A C C T
|  |  |  *
A C C G
```

In this example, there are three matching characters and one mismatch, so the Hamming distance is one.

The similarity score for two sequences is then calculated as follows:

similarity_score = (string length - hamming distance) / string length

similarity_score = (4-1)/4 = 3/4 = 0.75

Two sequences with a high similarity score are more similar than two sequences with a lower similarity score. *The two strings are always the same length when calculating a Hamming distance.*

Assignment Details:

In this assignment, you will calculate the similarity of two sequences using Hamming distance between two strings, search a genomic string looking for matches to a subsequence, and calculate the similarity scores for a sample DNA sequences compared to known DNA sequences.

Here we have provided a small portion of a DNA sequence from a human, a mouse, and an unknown species. Smaller DNA sequences will be compared to each of these larger DNA sequences to determine which has the best match. Each of the DNA sequences can be copied from this write-up and stored as a constant or variable in your program.

Part 1 – Compare two sequences to each other

float *similarityScore* (string sequence1, string sequence2)

Define the ***similarityScore*** function. This function takes two string arguments, calculates their Hamming distance, and returns their **similarity score** as a floating-point number. **This function should only calculate the similarity if the two strings are the same length, otherwise return 0.**

You should implement your main function to repeatedly ask to the user for 2 strings (***sequence1*** and ***sequence2***) such that you can pass them as parameters to your ***similarityScore*** function. Writing your own main function will allow you to test and debug every function you write.

Example:

```
sequence1 = CCGCCGCCGA
            ||*||*||*|
sequence2 = CCTCCTCCTA

Matches = 7, Mismatches = 3
Hamming distance = 3, Length = 10
Similarity = (10 - 3) / 10 = 0.7

result = similarityScore("CCGCCGCCGA", "CCTCCTCCTA");
cout << result << endl;
The variable "result" should equal 0.7
```

COG will grade Part 1 by generating random sequences to be compared. The results of your function will be compared to our results for the same sequences.

Part 2 – Count the matches of a sequence to a genome

int *countMatches*(string genome, string sequence1 , float min_score)

The ***countMatches*** function takes three parameters, a string containing the genome to search, a string containing the sequence to find, and a floating point value containing the minimum similarity score that will be considered a match.

The function does the following:

- The function should find all the positions of the *genome* where the genome's substring matches *sequence1* with a similarity greater than or equal to the *min_score*.
- The function should return the count of the number of such positions it found.

Example:

```
n_matches = countMatches("CCGCCGCCGA", "CGC", 0.60);

Matching genome to CGC at 0.60 similarity.
The sequence CGC matches 100% at position 1 and 4.
Position 7 matches at 67%, still above the threshold.
Therefore, the value returned by the function would be 3.
```

COG will grade Part 2 by generating random sequences to be found in a given genome. The results of your function will be compared to our results for the same sequences.

Part 3 – Find best matching genome for a given sequence

We have a random DNA sequence, and we want to find the closest species to it.
Is the DNA sequence more similar to human, mouse, or unknown?

When could this kind of comparison be useful? Suppose that the emergency room of some hospital sees a sudden and drastic increase in patients presenting with a particular set of symptoms. Doctors determine the cause to be bacterial, but without knowing the specific species involved they are unable to treat patients effectively. One way of identifying the cause is to obtain a DNA sample and compare it against known bacterial genomes. With a set of similarity scores, doctors can then make more informed decisions regarding treatment, prevention, and tracking of the disease.

The goal of this part of the assignment is to write functions that can be useful to determine the identity of different species of bacteria, animals, etc … . By simply using the similarity score routine you implemented you can compare an unknown sequence to different genomes and figure out the identity of the unknown sample.

float *findBestMatch*(string genome, string seq)

The ***findBestMatch*** function should take two string arguments and return a floating point value of the highest similarity score found for the given sequence at any position within the genome. In other words, this function should traverse the entire genome and find the highest similarity score by using *similarityScore()* for the comparisons between *seq* and each sequential substring of genome.

<hint: this function is very similar in structure to the countMatches function>

int *findBestGenome*(string genome1, string genome2, string genome3, string seq)

- The ***findBestGenome*** function should take four string arguments(unknown sequence, mouse_genome, human_genome and unknown_genome).
- Return an integer indicating which genome string, out of the 3 given, had the highest similarity score with the given *sequence*.
- For each genome, the function will find the highest similarity score of the *sequence* (at any position) within that genome (call function ***findBestMatch*** described above).
- The return value from this function will indicate which genome had the best match, 1, 2, or 3. In the case that two or more of the sequences have the same best similarity score, return 0.

COG will grade Part 3 based on both the value returned from *findBestGenome* and *findBestMatch*.

Note: DNA sequences for human, mouse and unknown genomes will be uploaded as a file on Moodle with this assignment for testing purposes.

Getting started

Let's talk about implementation and testing your solutions. We should approach the design of algorithms or programs from the top down. Begin writing your algorithms as very abstract descriptions and then repeatedly writing more detailed descriptions of the complex abstractions. Each of your abstractions could be its own function. Once you have reached a level of detail in your descriptions that provides well-understood solutions to the problem, you can begin to implement (write the code) for those functions. Implementation is best done from the bottom up, meaning you implement the base functions (those functions that don't call other functions) first and test them until you are satisfied they behave correctly and therefore have the correct code for implementing that function.

Once you have your base functions implemented, you can implement the next layer of abstractions that use those base functions in their algorithms. Again, test the new functions until you are satisfied they perform as required. This method of bottom up will progressively add in functionality until the complete algorithm is implemented.

This assignment is written in a manner that will allow you to implement functions each part and use COG to verify that it works before implementing the next part.

Write your *similarityScore()* function and test it by calling it from *main()* and passing it two known strings. For example:

cout << "Test1: " << *similarityScore*("ACGT","ACGG") << endl;
cout << "Test2: " << *similarityScore*("ATAT","ACAC") << endl;
. . .

Once you know that your function works for the specific tests, modify your main to get the input from the user and pass the strings to the function. Once the user input and output are implemented as required, submit to COG to verify your compliance with all the requirements. Now you can implement the code for the next part of the assignment. Repeat the implementation, local testing, and COG testing stages for each of the required functions.

When testing the functions, it may be easier to use smaller genome sequences when debugging your code. Use simple examples to verify that the functions are working before testing it on longer strings. Once you're confident the function works, call your functions from *main()* using the following strings as the first input parameter to the function:

HumanDNA =
"CGCAAATTTGCCGGATTTCCTTTGCTGTTCCTGCATGTAGTTTAAACGAGATTGCC
AGCACCGGGTATCATTCACCATTTTTCTTTTCGTTAACTTGCCGTCAGCCTTTTCTT
TGACCTCTTCTTTCTGTTCATGTGTATTTGCTGTCTCTTAGCCCAGACTTCCCGTGT
CCTTTCCACCGGGCCTTTGAGAGGTCACAGGGTCTTGATGCTGTGGTCTTCATCTG
CAGGTGTCTGACTTCCAGCAACTGCTGGCCTGTGCCAGGGTGCAGCTGAGCACTG
GAGTGGAGTTTTCCTGTGGAGAGGAGCCATGCCTAGAGTGGGATGGGCCATTGTT
CATG"

mouseDNA =
"CGCAATTTTTACTTAATTCTTTTTCTTTTAATTCATATATTTTTAATATGTTTACTATT
AATGGTTATCATTCACCATTTAACTATTTGTTATTTTGACGTCATTTTTTTCTATTTCC
TCTTTTTTCAATTCATGTTTATTTTCTGTATTTTTGTTAAGTTTTCACAAGTCTAATAT
AATTGTCCTTTGAGAGGTTATTTGGTCTATATTTTTTTTTCTTCATCTGTATTTTTATG
ATTTCATTTAATTGATTTTCATTGACAGGGTTCTGCTGTGTTCTGGATTGTATTTTTC
TTGTGGAGAGGAACTATTTCTTGAGTGGGATGTACCTTTGTTCTTG"

unknownDNA =
"CGCATTTTTGCCGGTTTTCCTTTGCTGTTTATTCATTTATTTTAAACGATATTTATAT
CATCGGGTTTCATTCACTATTTTTCTTTTCGATAAATTTTTGTCAGCATTTTCTTTTAC
CTCTTCTTTCTGTTTATGTTAATTTTCTGTTTCTTAACCCAGTCTTCTCGATTCTTATC
TACCGGACCTATTATAGGTCACAGGGTCTTGATGCTTTGGTTTTCATCTGCAAGAG
TCTGACTTCCTGCTAATGCTGTTCTGTGTCAGGGTGCATCTGAGCACTGATGTGGA
GTTTTCTTGTGGATATGAGCCATTCATAGTGTGGGATGTGCCATAGTTCATG"

Challenge Problem (not graded by COG, do not submit):

DNA is double stranded even though we always write a single strand's nucleotide sequence. This is because we can infer the other strand's sequence from the given strand. Every A on one strand has a complementary T on the opposite strand (every C has a G). Therefore we can create the complement strand by swapping the nucleotides with the complementary nucleotide. *Create a function to take the sequence and produce the complement sequence.*

But, you are not done yet. You now have the complement strand, but the DNA is read in the forward direction for the given strand and in the reverse direction for the complementary strand. Therefore, we must reverse the strand (e.g. first character becomes the last character, second becomes second to last, …) to have the correctly specified reverse complement of the given DNA sequence. The challenge problem is to c*reate a function to produce the reverse complement of a given DNA sequence. Once you have a function to produce the reverse complement of a string, you can search both the DNA and reverse complement looking for the best match.*