CSCI 1300 Introduction to Computer Programming
Instructor: Knox
Assignment 4
Due Friday, Feb 24, by 12:30 pm

This assignment requires you to write two files: *main.cpp* and *Assignment4.cpp*. There are four 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 four that are required, these should be kept in Assignment4.cpp as well. Your main.cpp file should just implement your main() function. 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 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 # from the dropdown.
- Upload your .zip file 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 10% of the grade if code is found to be not commented or formatted with proper indentation.**

Upload one .zip file to Moodle containing <u>two files</u>: your own <u>main.cpp</u> *and* <u>Assignment4.cpp</u> *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. You must provide functions with the given names and parameters. The names of any other supporting functions are named at your discretion. The names should be reflective of the function they perform.

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

Your program will ask the user for two small sequences (no more than 10 characters) and calculate a similarity score between the sequences. Pass the two sequences to a function named *calcSimilarity()* that returns a floating-point result (similarity score described above). Have your main code repeat the similarity calculations until the value of sequence 1 is a single character '*'.

float *calcSimilarity* (string sequenceOne, string sequenceTwo)
The ***calcSimilarity****()* function will take two arguments that are both strings. The function calculates the Hamming distance and returns the 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.

The output should correspond to the following:

```
Enter sequence 1:
CCGCCGCCGA
Enter sequence 2:
CCTCCTCCTA
Similarity: 0.7
Enter sequence 1:
*
```

COG will grade Part 1 based on the value returned by the function.

Part 2 – Find locations of matches between genome and sequence

Your program will ask the user for a small DNA sequence and search each of the given genomes (human, mouse, unknown) to find exact matches. Your main program should prompt the user for the search sequence, and pass each of the predefined genomes, species names and the small search sequence to the function *listSequencePositions(),* which does not return any value. The function will print the name of the genome followed by all locations of the exact matches. Repeat the search across all genomes in your main code until the small sequence given is a single character '*'.

void *listSequencePositions*(string genomeSequence,
string genomeName, string seq)

The ***listSequencePositions****()* function will take three arguments that are all strings and print each of the positions where *genomeSequence* has an exact match to *seq*. The function will print the *genomeName* of the genome, followed by the locations of all exact matches on a single line and separated by spaces.

The output should correspond to the following:

```
Enter sequence:
CCG
Human match locations: 11 61 98 165 179
Mouse match locations:
Unknown match locations: 11 179
Enter sequence:
*
```

COG will grade Part 2 based on the print statement within the function.

Note: the DNA sequence for human, mouse and unknown genomes are given on page 8 in this assignment.

Part 3 – Find best match between genome and sequence

Your program will ask the user for a sequence that will be compared to each of the genomes (Human, Mouse, Unknown) to find which genome has the best match to the given sequence. Your program will provide a function *compareDNA()* to find the highest similarity score of the given sequence anywhere along the given genome. Your program will output the name of the genome with the best match. Have your main code repeat until the sequence given is a single character '*'.

float compareDNA(string genome, string seq)
The *compareDNA()* function should take two arguments that are both strings and return the *best* similarity score that can be found for that sequence in the genome. You should use the ***calcSimilarity()*** function described above to compare the sequence to all substrings of the genome.

void compare3Genomes(string genome1, string name1, string genome2, string name2, string genome3, string name3, string seq)
The *compare3Genomes()* function should take seven arguments that are all strings and print the name of the genome with the best similarity score that can be found for that sequence in the genome. In the case that multiple genomes have the same best similarity score, print the names of all of the genomes with the same score.

The output should correspond to the following:

```
Enter user sequence:
GTAGTTTAAA
Best match: Human
Enter user sequence:
GCGGGGTCG
Best match: Human
Best match: Unknown
Enter user sequence:
*
```

COG will grade Part 3 based on both the value returned from *compareDNA()* as well as the print statement output within the function *compare3Genomes()*.

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

```
cout << "Test1: " << calcSimilarity("ACGT","ACGG") << endl;
cout << "Test2: " << calcSimilarity("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 =
"CGCAAATTTGCCGGATTTCCTTTGCTGTTCCTGCATGTAGTTTAAACGAGATTGCCAG
CACCGGGTATCATTCACCATTTTTCTTTTCGTTAACTTGCCGTCAGCCTTTTCTTTGAC
CTCTTCTTTCTGTTCATGTGTATTTGCTGTCTCTTAGCCCAGACTTCCCGTGTCCTTTC
CACCGGGCCTTTGAGAGGTCACAGGGTCTTGATGCTGTGGTCTTCATCTGCAGGTGTCT
GACTTCCAGCAACTGCTGGCCTGTGCCAGGGTGCAGCTGAGCACTGGAGTGGAGTTTTC
CTGTGGAGAGGAGCCATGCCTAGAGTGGGATGGGCCATTGTTCATG"

mouseDNA =
"CGCAATTTTTACTTAATTCTTTTTCTTTTAATTCATATATTTTTAATATGTTTACTAT
TAATGGTTATCATTCACCATTTAACTATTTGTTATTTTGACGTCATTTTTTTCTATTTC
CTCTTTTTTCAATTCATGTTTATTTTCTGTATTTTTGTTAAGTTTTCACAAGTCTAATA
TAATTGTCCTTTGAGAGGTTATTTGGTCTATATTTTTTTTTCTTCATCTGTATTTTTAT
GATTTCATTTAATTGATTTTCATTGACAGGGTTCTGCTGTGTTCTGGATTGTATTTTTC
TTGTGGAGAGGAACTATTTCTTGAGTGGGATGTACCTTTGTTCTTG"

unknownDNA =
"CGCATTTTTGCCGGTTTTCCTTTGCTGTTTATTCATTTATTTTAAACGATATTTATAT
CATCGGGTTTCATTCACTATTTTTCTTTTCGATAAATTTTTGTCAGCATTTTCTTTTAC
CTCTTCTTTCTGTTTATGTTAATTTTCTGTTTCTTAACCCAGTCTTCTCGATTCTTATC
TACCGGACCTATTATAGGTCACAGGGTCTTGATGCTTTGGTTTTCATCTGCAAGAGTCT
GACTTCCTGCTAATGCTGTTCTGTGTCAGGGTGCATCTGAGCACTGATGTGGAGTTTTC
TTGTGGATATGAGCCATTCATAGTGTGGGATGTGCCATAGTTCATG"
```

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.*