CSCI 1300 - Intro to Computer Programming
Instructor: Fleming/Gupta
Homework 4

Due Sunday, February 18th, by 6 pm
+5% bonus if submitted by Friday February 16th 11:55 pm,
+2% bonus if submitted by Saturday February 17th 11:55 pm

This assignment is due **Sunday, February 18th 6pm**.

- ***All components* (Cloud9 workspace, moodle quiz attempts, and zip file) must be completed and submitted by Sunday, February 18th 6:00 pm for your homework to receive points.**
- Complete submissions (Cloud9 workspace, moodle quiz attempts, and zip file) before **Friday February 16th 11:55 pm** will receive a 5% bonus, and complete submissions before **Saturday February 17th 11:55 pm** will receive a 2% bonus.

This assignment is interview graded. If you do not schedule and complete an interview grading session with your TA, you will receive a 0 for the whole assignment, regardless of the score the autograder gives you.

If you need to reschedule an interview grading appointment and do not inform your TA at least 24 hours in advance, you will also receive a 0.

Develop in Cloud9: For this assignment, write and test your solution using Cloud9.

Submission: All three steps must be fully completed by the submission deadline for your homework to receive points. Partial submissions will not be graded.

1. ***Share your Cloud 9 workspace with your TA:*** Your recitation TA will review your code by going to your Cloud9 workspace. *TAs will check the last version that was saved before the submission deadline*.
    - Create a directory called **Hmwk4** and place all your file(s) for this assignment in this directory.
2. ***Submit to the Moodle Autograder:*** Head over to Moodle to the link **Hmwk 4**. You will find 3 programming quiz questions for each part of the assignment. You can modify your code and re-submit (press *Check* again) as many times as you need to, up until the assignment due date.
3. ***Submit a zip file to Moodle:*** After you have completed all the questions and checked them on Moodle, ***you must submit a zip file with the .cpp file(s) you wrote in Cloud9***. Submit this file going to **Hmwk 4 (File Submission)** on moodle.

Style and Comments (26 points)

Comments (10 points):

- Your code should be well commented. Use comments to explain what you are doing, especially if you have a complex section of code. These comments are intended to help other developers understand how your code works. These comments should begin with two backslashes (//).
- Please also include a comment at the top of your solution with the following format:

```
// CS1300 Spring 2018
// Author:
// Recitation: 123 – Favorite TA
// Cloud9 Workspace Editor Link: https://ide.c9.io/…
// Homework 4 - Problem # …
```

NEW: Indentation (6 points):

- Your code should use some form of indentation. Please look at examples posted on moodle as well as the example below (**checkMpg)** if you are unsure of how to use indents.

Algorithm (10 points):

- Remember to include in comments, before the function definition, a description of the algorithm you used for that function.
- This is an example C++ solution to problem 5 of Homework 1. Look at the code and the algorithm description for an example of what is expected.

```
/**
 * Algorithm: that checks what range a given MPG falls into.
 *      1. Take the mpg value passed to the function.
        2. Check if it is greater than 50.
              If yes, then print "Nice job"
        3. If not, then check if it is greater than 25.
              If yes, then print "Not great, but okay."
        4. If not, then print "So bad, so very, very bad"
 * Parameters: miles per gallon (float type)
 * Output: different string based on three categories of
 *            MPG: 50+, 25-49, and less than 25.
 * Returns: nothing
 */

void checkMPG(float mpg)
{
      if(mpg > 50) // check if the input value is greater than 50
      {
            cout << "Nice job" << endl; // output message
      }
      else if(mpg > 25) //if not, check if is greater than 25
      {
            cout << "Not great, but okay." << endl; // output message
      }
      else // for all other values
      {
            cout << "So bad, so very, very bad" << endl; // output message
      }
}
```

The following algorithm description does not mention in detail what the algorithm does and does not mention what value the function returns. Also, the solution is not commented. This would not receive full credit.

```
/**
 * Checks mpg
 */
void checkMPG(float mpg) {
      if(mpg > 50) {
            cout << "Nice job" << endl;
      }
      else if(mpg > 25) {
                  cout << "Not great, but okay." << endl;
      }
      else {
                  cout << "So bad, so very, very bad" << endl;
      }
}
```

Test Cases (4 points)

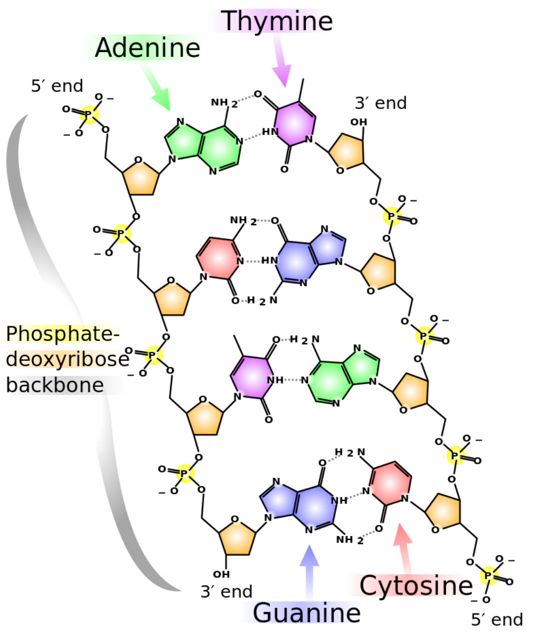
Code compiles and runs (4 points):

- The zip file you submit to moodle should contain .cpp file(s) that can be compiled and run on Cloud9 with no errors. The functions should match those submitted to the autograder.

No test cases required in main for this assignment.

Background

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 DNA is stored as code in four chemical bases: adenine (**A**), guanine (**G**), cytosine (**C**) and thymine (**T**). The differences in the order of these bases is a means of specifying different information.

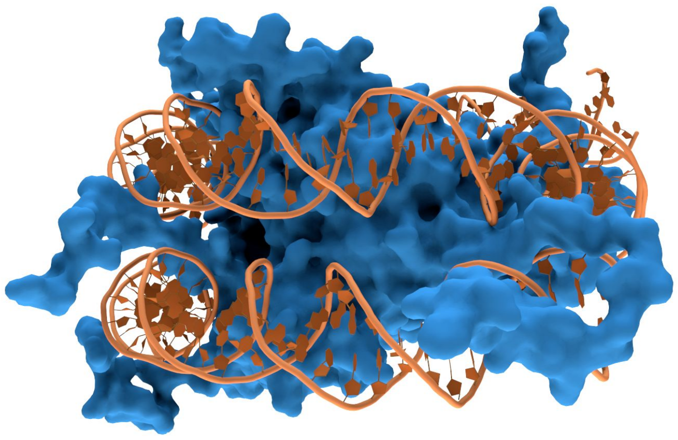
We refer to an organism's complete set of chemical bases as their *genome*. Every genome looks something like this:

```
GATCAATGAGGTGGACACCAGAGGCGGGGACTTGTAAATAACACTGGGCTGTAGGAGTGATGGGGTTCACCTCTAATTCTAAGATGGCTAGATAATG
CATCTTTCAGGGTTGTGCTTCTATCTAGAAGGTAGAGCTGTGGTCGTTCAATAAAAGTCCTCAAGAGGTTGGTTAATACGCATGTTTAATAGTACAG
TATGGTGACTATAGTCAACAATAATTTATTGTACATTTTTAAATAGCTAGAAGAAAAGCATTGGGAAGTTTCCAACATGAAGAAAAGATAAATGGTC
AAGGGAATGGATATCCTAATTACCCTGATTTGATCATTATGCATTATATACATGAATCAAAATATCACACATACCTTCAAACTATGTACAAATATTA
TATACCAATAAAAAATCATCATCATCATCTCCATCATCACCACCCTCCTCCTCATCACCACCAGCATCACCACCATCATCACCACCACCATCATCAC
CACCACCACTGCCATCATCATCACCACCACTGTGCCATCATCATCACCACCACTGTCATTATCACCACCACCATCATCACCAACACCACTGCCATCG
TCATCACCACCACTGTCATTATCACCACCACCATCACCAACATCACCACCACCATTATCACCACCATCAACACCACCACCCCCATCATCATCATCAC
TACTACCATCATTACCAGCACCACCACCACTATCACCACCACCACCACAATCACCATCACCACTATCATCAACATCATCACTACCACCATCACCAAC
ACCA
```

Human Genome: First 1000 lines of Chromosome 1 (http://www.sacred-texts.com/dna/hgp011k.htm)

The human genome has about 3 billion DNA base pairs. That means the entire human genome can be represented by a (very long) string of ~3 billion characters, where every character in the string is A, C, G, or T. You can find the first 1000 lines of Chromosome 1 of the Human Genome at this webpage.

In the lectures there have been examples of string representation and use in C++. In this assignment we will create strings that represent DNA sequences.   We will be implementing a number of functions that are used to search for substrings that represent sequences within the DNA.



Interaction of DNA (orange) with histones (blue), a DNA binding protein. These proteins' basic amino acids bind to the acidic phosphate groups on DNA.
Thomas Splettstoesser

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 3 matching characters and 1 mismatch, so:

hamming_distance = 1

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 must always be the same length when calculating a Hamming distance.

Problem Set:

In this assignment, you will be required to write a whole program (.cpp file) that can be interacted with via command-line user input. We have provided a skeleton **H4.cpp** file that has comments explaining the expected structure of the file on moodle.

Function 1: Find Similarity Score (20 points)
A method for calculating similarity score is described above in ***Hamming distance and similarity between two strings***.

Write a function that returns the similarity score for two sequences.

- Your function should take two parameters *in this order*:
  - a string parameter for the first sequence
  - a string parameter for the second sequence
- Your function should return the similarity score as a float.
- Your function should not print anything.
- Your function MUST be named **findSimilarityScore**

Edge Cases:
- lengths of sequences are different → return -1
- either sequence is an empty string → return -1

Examples:
- "**ATGC**", "**ATGC**" → 1.0
- "AT", "GC" → 0
- "**ATG**C, "**ATG**A" → 0.75

Function 2: Find Best Match (40 points)
We will use the term *genome* to refer to the string that represents the complete set of genes in an organism, and *sequence* to refer to some substring or sub-sequence in the genome.

Write a function called **findBestMatch** that takes a genome and a sequence and returns the similarity score of the best match found in the genome as a float.

HINT: Problem 3 from Recitation 5 is *very similar* to this function. We encourage you to write that function before attempting to write this one.

- Your function should take two parameters *in this order*:
  - a string parameter for the genome
  - a string parameter for the sequence
- Your function should return the highest similarity score as a float.
- Your function should not print anything.
- Your function MUST be named **findBestMatch**.

Edge Cases:

- sequence is empty → return -1
- genome is empty → return -1
- sequence is longer than genome → return -2

Examples:

- "ATACGC", "ACT" → 0.66

Our sequence is "ACT", which is a string of length 3. That means we need to compare our sequence with all the 3 character long sub-sequences (substrings) in the genome.

genome sub-sequence	sequence	similarity score
ATACGC	**ACT**	0.33
A**TAC**GC	**ACT**	0
AT**ACG**C	**ACT**	0.66
ATA**CGC**	**ACT**	0

← **findBestMatch** returns 0.66, since that is the highest similarity score found

<u>Main Function</u> (40 points)

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 make more informed decisions regarding treatment, prevention, and tracking of the disease.

Write a program that takes in three known genomes and a sequence from the unknown bacteria and determines the most probable match.

You will need to include the two functions you wrote above in your file. Here is what should occur in your main:

1. Prompt user to enter a genome, using the format provided below:
   - `"Please enter genome 1:"`
   - `"Please enter genome 2:"`
   - `"Please enter genome 3:"`
2. Prompt user for a sequence, using the format provided below:
   - `"Please enter a sequence:"`
3. Validate sequence length. If the sequence is longer than *any of the provided genomes*, provide the output below and go back to step 2. Repeat this step as many times as necessary until the user provides a valid sequence.
   - `"Sequence cannot be longer than genomes."`
4. Find the best match for genome 1 and the sequence.
5. Find the best match for genome 2 and the sequence.
6. Find the best match for genome 3 and the sequence.
7. Display the best matching genome using the format provided below if 1 genome gives the best match:
   - `"Genome 1 is the best match."`
   - `"Genome 2 is the best match."`
   - `"Genome 3 is the best match.”`
8. If 2 or more genomes are the best matches, then display all the best matches separated by ‘\n’:
   - E.g. If genome 2 and 3 are the best matches then display the following:
     i.
     ```
     "Genome 2 is the best match."
     "Genome 3 is the best match."
     ```
   - If all the genomes have the same score then display the following:
     i.
     ```
     "Genome 1 is the best match."
     "Genome 2 is the best match."
     "Genome 3 is the best match."
     ```
9. If either of the genomes or sequence is empty, display the following output:
   - `"Genome and sequence cannot be empty."`

Examples:

genome 1	AAT
genome 2	CGC
genome 3	ATG
sequence	C
Genome 2 is the best match.	

genome 1	AATGTTTCAC
genome 2	GACCGACTAA
genome 3	AAGGTGCTCC
sequence	TACTA
Genome 2 is the best match.	

genome 1	AAT
genome 2	AAT
genome 3	AAG
sequence	AAT
Genome 1 is the best match. Genome 2 is the best match.	

genome 1	CAAGAAGCAGGCTCGGTAACACACGGTCTAGCTGACTGTCTATCGCCTAG GTCATATAGGGACCTTTGATATCTGCATGTCCAGCCTTAGAATTCACTTCA GCGCGCAGGTTTGGGTCGAGATAAAATCACCAGTACCCAAGACCAGGGGG GCTCGGCGCGTTGGCTAATCCTGGTACATCTTGTTATGAATATTCAGTAG AAAATCTGTGTTAGAGGGACGAGTCACCATGTACCAAAAGCGATATTAAT CGGTGGGAGTATTCATCGTGGTGAAGACGCTGGGTTTACGTGGGAAAGGT GCTTGTGTCCCAACAGGCTAGGATATAA

genome 2	TAAGTTAATTCTTATGGAATATAATAACATGTGGATGGCCAGTGGTCGGT TGTTACACGCCTACCGCGATGCTGAATGACCCGGACTAGAGTGGCGAAAT TTATGGCGTGTGACCCGTTATGCTCCATTTCGGTCAGTGGGTCATTGCTAG TAGTCGATTGCATTGCCATTCTCCGAGTGATTTAGCGTGACAGCCGCAGG GAACCCATAAAATGCAATCGTAGTCCACCTGATCGTACTTAGAAATGAGG GTCCCCTTTTGCCCACGCACCTGTTCGCTCGTCGTTTGCTTTTAAGAACCG CACGAACCACAGAGCATAAAGAGAACCTCTAGCTCCTTTACAAAGTACTG GTTCCCTTTCCAGCGGGATGCCTTATCTAAACGCAATGACAGACGTATTCC TCAGGCCACATCGCTTCCTACTTCCGCTGGGATCCATCATTGGCGGCCGAA GCCGCCATTCCATAGTGAGTCCTTCGTCTGTGTCTTTCTGTGCCAG
genome 3	ATCGTCTAGCAAATTGCCGATCCAGTTTATCTCACGAAACTATAGTCGTAC AGACCGAAATCTTAAGTCAAATCACGCGACTAGGCTCAGCTCTATTTTAG TGGTCATGGGTTTTGGTCCGCCCGAGCGGTGCAACCGATTAGGACCATGT AAAACATTTGTTACAAGTCTTCTTTTAAACACAATCTTCCTGCTCAGTGGC GCATGATTATCGTTGTTGCTAGCCAGCGTGGTAAGTAACAGCACCACTGC GAGCCTAATGTGCCCTTTCCACGAACACAGGGCTGTCCGATCCTATATTAG GACTCCGCAATGGGGTTAGCAAGTCGCACCCTAAACGATGTTGAAGACTC GCGATGTACATGCTCTGGTACAATACATACGTGTTCCGGCTGTTATCCTGC ATCGGAACCTCAATCATGCATCGCACCAGCGTATTCGTGTCATCTAGGAGG GGCGCGTAGGATAAATAATTCAATTAAGATGTCGTTATGCTAGTA
sequence	GTGGTAAGTAACAGCACCACTGCGAGCCTAATGTGCCCTTTCCACGAACAC AGGGCTGTCCGATCCTATACTAGGACTCCGCAATGGGGTTAGCAAGTCGC ACCCTAAACGATGTTGAAGCCTCGCGATGTACATGCTCTGGTACAATACA TACGTG
Genome 3 is the best match.	

Edge Cases:

genome 1	AAG
genome 2	TGC
genome 3	GATTACA
sequence	GAAC
Sequence cannot be longer than genomes.	

genome 1	

genome 2	TGC
genome 3	GATTACA
sequence	GAAC
Genome and sequence cannot be empty.	