PA3: DNA (Due 10/4 @ 11:59PM)

The GitHub classroom link for this assignment is: https://classroom.github.com/a/4agZImhd

The focus of this assignment is on DNA and Bioinformatics. You will want to keep the corresponding lecture notes handy. The completed program has two main features: determining the amino acids produced by a DNA sequence and an implementation of the Needleman-Wunsch algorithm for global pairwise sequence alignment.

Creating New Files

For this project, you will need to create new sources files. Here's what you do:

- Only create new files in your repo's `src` directory. Don't put them anywhere else.
- So that CMake knows to build your new files, you need to edit the `src/CMakeLists.txt` file.
    - Header files go immediately after SrcMain.h in the file. Put your file name on a new line, before the closing parenthesis.
    - Likewise, source files go after SrcMain.cpp
- If you're on Mac, whenever you change the CMakeLists.txt file to add new files, you need to ***close*** Xcode and rerun the `cmake -G Xcode .` command. This will regenerate the Xcode project (you shouldn't have to set the scheme's working directory again, at least).

Command Line Arguments

The command line arguments are simple for this assignment.

If argc == 2, then argv[1] is the name of FASTA file which you should determine the amino acid count of.

If argc == 3, then argv[1] and argv[2] are the names of two FASTA files for pairwise sequence alignment.

Part 1: FASTA File Format

FASTA is a simple text-based file format for representing nucleotides and amino acid chains. There are several FASTA files we give you in the `input` directory. A good file for initial testing is `TAS2R16_Homo_sapiens.fasta`.

For this assignment, a FASTA file meets the following conditions:

- The first line must begin with a `>` and everything after the `>` is the "header" of the file
- The data for the nucleotides will only contain the letters T, C, A, and G.

You may assume we'll only give you valid input files.

You need to create a class to handle parsing of the FASTA files. You should make a new header/cpp file for this class (see the instructions on page 1). You'll need two member variables – a `std::string` to store the header and a string that stores the actual sequence.

The constructor should take in a file name. When you first open the file, it is recommended you use the `std::ios::ate` trick used in the RLE assignment to determine the size of the input file. However, you want to open the file in the default text mode, ***not*** in binary mode. Once you have the size of the file, you can then use the `std::string` member function `reserve` to preallocate that many bytes for the sequence member variable. This will save the unnecessary cost of the string resizing over time.

Then read in the file into the string. Keep in mind that you need to ignore whitespace characters like `'\n'`, `'\r'`, and `'\xff'` (EOF). You only care about the A, T, C, G letters.

You should probably write some test cases (in StudentTests.cpp) to verify your file's loading properly.

Part 2: Amino Acid Count

Given a FASTA file, you will run it through a DNA translator to determine which amino acids the sequence produces. Implement this in a separate class that performs the DNA translation of a given FASTA sequence. You ***must*** use the state machine approach outlined in lecture. ***Do not use a hash map*** to represent the state machine itself (you can use hash maps for other information, though, like tracking the count of specific amino acids).

To construct the amino acid histogram, you will want to consult a DNA codon table such as the one in the notes, or this page on Wikipedia.

Given an arbitrary sequence, there are three main parts to determining what amino acids it produces:

1. Keep reading in characters until you get a ATG start codon. This both produces Methionine (M) and signals that translation of amino acids should begin.
2. Reading in one character at a time, your state machine will then start producing different amino acids (just add to the count of that acid). This should continue until either the end of the sequence *or* you read a stop codon.
3. If you read a stop codon, you should not produce any more amino acids until the next ATG start codon (eg. go back to step 1)

Write some test cases for your translator class in StudentTests.cpp.

Then, hook it up in ProcessCommandArgs (this is the argc == 2 case) so that it loads the specified FASTA file, and then outputs the amino acid count into a file called amino.txt.

The sampleoutput directory has three samples, including TAS2R16_amino.txt, showing what the output should look like (the header, the total count, then the counts of each file).

Once you pass the first amino acid test in the given StudentTests.cpp file, you can uncomment the other two and make sure you pass.

If you push to Travis, you should get a graded test score of 30/75.

Part 3: Needleman-Wunsch Algorithm

In this final part of the lab, you will be implementing the Needlman-Wunsch algorithm for global pairwise sequence alignment. The purpose of global sequence alignment is to help visualize how closely related two DNA sequences are. I've provided a couple of sample output files in the output subdirectory. Look at the `TAS2R16_Comparison.result` file – this compares the same taste receptor gene in a human vs. a chimpanzee. As you can see, other than the beginning and end of the sequences, the two DNA sequences are nearly identical – thus, one can infer that this gene is very similar between the two species.

As discussed in lecture, this is an application of dynamic programming. You will want to keep the lecture notes handy for this part of the lab.

In the argc == 3 case in ProcessCommandArgs, you'll run the NW algorithm. Your output file should be a file called `out.result`.

Implementing Needleman-Wunsch

Now create a class to perform the Needleman-Wunsch algorithm. For testing/debugging purposes, I recommend splitting this into three functions:

1. The constructor will load in the two input FASTA files and make sure everything is okay with them.
2. A function that populates the score/direction grid, runs the algorithm, and produces the resultant sequence strings.
3. A function that writes the results of the match to the .result file.

The constructor is straightforward. The second function will be quite a bit of work, though.

To keep memory usage down, create two grids – one of shorts that stores the score, and one grid of characters that stores a direction to the prior cell. You can create an enum that stores the three possibilities (above left, left, and above).

To ensure that an enum uses up only 1 byte per instance, you can specify the size like such:

```
enum MyEnum : char
{
   // Values...
}
```

Using these two separate grids, a match between two 20KB input sequences (as the Ebola sequences approximately are) will use up 1.2 GB. This should be okay for our purposes – though as I'm sure you can see, we would run out of memory if the sequences got much larger.

There's a couple of different ways to store the grid. I'd recommend just having a `std::vector` of `std::vector`s, though if you really want to you could just use regular arrays (though you will have to manually shift indices since you cannot dynamically allocate a 2D array).

Recall that Needleman-Wunsch requires a match score, mismatch score, and gap score. For this lab, you can hard code them as follows:

- Match Score: 1
- Mismatch Score: -1
- Gap Score: -1

The implementation of the actual Needleman-Wunsch algorithm should be roughly as follows:

1. Initialize your grid of data (as in the notes)
2. For every square (going row by row), compute the three possible scores, select the best predecessor, and set both the score and direction pointer to this best predecessor. **One important note**: if there's a tie in scores, first select above left, then select left, then select above. If you don't do this your output will not match the reference output.
3. Once you've computed every row/column, the bottom right corner should contain the final score.
4. Create the resulting sequence strings by performing the traceback step – I'd recommend constructing the resultant strings backwards (since that's simplest), then using std::reverse to reverse the string. I'd also recommend preallocating the resultant strings so that you don't end up reallocating over and over.

Write some test cases to verify your functions.

Outputting the Result File

Once you're confident your Needlman-Wunsch algorithm works, you need to output the global pairwise match to the result file.

If you open up `TAS2R16_Comparison.result`, you'll see that the file format is not that complicated.

The first line contains an A: followed by the header from sequence A. The second line does the same for B. The third line outputs the final score of the sequence. There is then an extra newline and the matches start.

The matches look like this:

```
70 characters from result sequence A
Any characters that are identical at the same index in seq A/B have a |
70 characters from result sequence B
Extra newline
Repeat
```

Initially, only the Small test in StudentTests run. Once that passes, uncomment the TAS2R16 test and make sure that passes.

For the Ebola test, you need to first change your build to Release mode (otherwise it takes too long to complete). To run in Release mode:

- In Visual Studio, look for the dropdown on the toolbar that says "x64-Debug" and change it to "x64-Release". When you do this, give Visual Studio a moment. You then must set the startup executable to tests/tests.exe again.
- In Xcode, go to Product>Scheme>Edit Scheme… and under the "Info" tab change the "Configuration" to Release

Then uncomment the remaining pairwise match file tests. Note that one of the tests checks to see ***how long*** your Ebola comparison takes to complete. We've given you a time limit of 7 seconds, which is plenty of time. My implementation finishes in about 2 seconds on my PC, and 3.8 seconds on Travis.

Worst case, you only lose 5 points if your program runs too slowly.

Grading Rubric

Part	**Points**
Graded Unit Tests	75
Your Test Cass Quality/Variety	15
Code Quality (incl. following the style guide)	10
Total	**100**