CSC 211 Assignment 4

Quicksort

Due Friday, April 14th by 11PM

Background

Quicksort is an elegant, recursive sorting algorithm. Like any sorting algorithm, it can be applied to more than just numbers.

In this assignment, you'll implement quicksort such that it can sort instances of your DNA class. What makes this interesting is that there are multiple ways in which you might compare two DNA objects. For example, is one sequence shorter or longer than another? Or, if we want to arrange DNA objects in a catalog of some sort, we might want to order them alphabetically by header[1]. In this case, we might ask whether one header should appear before another header.

Some new features and concepts you'll need for this assignment are **reference types**, **templated functions**, and **recursion**.

Getting Started

- Get into your Docker development environment
- Download the assignment framework with `git clone https://github.com/csc211/a4`
- Look in the resulting directory `a4`. You'll see a working `dna.cpp` and its associated header file, as well as a `sort.cpp` (which isn't yet complete), and a `sort.h`.
- You'll need to write (or copy from your **lab 6**) `swap.h`.
- Note: `./compile` will not work yet.
  - You need to write a `main.cpp` containing a `main()`. Its design is up to you!
  - You need to fill in the function definitions in `sort.cpp`
- Once you can compile, you'll produce an executable binary called `sortdna`, so you can run it as `./sortdna` (with whatever arguments you implement.)

[1] more correctly, lexicographically. Alphabetic ordering implies only an ordering based on letters, but we have other characters such as punctuation and numbers. Since the ASCII table already gives us an ordering of these, we can generalize to *lexicographical* ordering.

Requirements

Ultimately, you must implement quicksort on DNA sequences.

To do this, you will need to write two comparator functions: `seqLenLessThan` and `headerLessThan`, as described in the comments in `sort.h`

You will also need to write `quicksort` (which is a recursive function) and `partition` (you may use either the Lomuto or Hoare algorithm).

I have **given** you an implementation of `sort()` itself. This function is not recursive; it calls your `quicksort()` function. Please make sure you understand what it's doing, and ask the course staff if you do not.

Helpful hints

1. Write a trivial `main()` and fill in enough of each function definition in `sort.cpp`that you can compile the project (you may temporarily want to `void` some argument variables).
2. Write `swap()`, and make sure it works!
    - In lab 6, you wrote a templated `swap()`. You can reuse it for this assignment.
3. Write `partition()` next. Modify your `main()` to test `partition()` on some small examples.
    - Once `partition()` is done, you are almost finished.
4. Write the recursive `quicksort()` and modify `main()` to test it.

For your own sake, write your own tests and run them before you try submitting to gradescope!

If you are iterating on Gradescope before you have passed some sensible tests of your own, you are **wasting time**. Gradescope takes **minutes** to give you an answer, and it's not always informative, and you can't debug it. Your own test cases take **milliseconds** to give you an answer, and you can debug it!

Now, this is the first time you have used function references, or passed a function as a parameter to another function. So, here is a hint for your `main.cpp`. When you call `sort()`, you need to pass it the function you wish to use for the sorting comparator. Suppose that function were called `sortByFitness()` and it magically gave you most fit DNA sequence (which would be very handy if you were running a simulation of Evolution). Then, if all your DNA objects were in a vector called `vec`, you might do: `sort(vec, &sortByFitness)`. Note that there is a `&` before the name of `sortByFitness`, and also that there are no function-call parentheses after the name `sortByFitness`. You are *passing* a reference to the function, rather than *calling* the function.

Compile early and often!

The compiler is your friend, not your enemy. Compile after you write a little bit of code; it's far easier to find a problem if you know it's in the last few lines of code you wrote.

Grading Rubric

For this assignment, correctly passing all tests on Gradescope is worth 70% of your grade. Another 10% is based on using a templated `swap()` inside your `partition()` function[2]. The remaining 20% is based on reasonable commenting habits, and the structure and organization of your code.

If you circumvent the purpose of this assignment by calling `std::sort()` you will receive zero credit.

Submitting

You will submit **both `sort.cpp` and `swap.h`** via Gradescope, where its functional correctness will be graded automatically. You can submit as many times as you like; only the last submission will count towards your grade.

[2]It is possible to pass all functional correctness tests without using a templated `swap()` function; I have no simple way to check for this automatically. So, it is part of the manual grading rubric.