Dept. of Computer Science, University of California, Davis
ECS122b Instructor: Rob Gysel 1/27/18

Programming Assignment #1*
Due date: 2/5/18 11:59pm

Programs are to be submitted using `handin` by the due date; be sure your code compiles and works on the CSIF. If you are unfamiliar with `handin` or the CSIF, see `http://csifdocs.cs.ucdavis.edu/`. Use the command:

```
handin rsgysel 122b-Program1 file1 file2 ... fileN
```

You may work in groups of up to 2 people. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty.

Overview

In this project, you will:

1. Write basic unit tests.
2. Add basic building instructions to `CMakeLists.txt` for `CMake`.
3. Write sanity checks.
4. Implement two exact matching algorithms: Z-algorithm based matching and Boyer-Moore.

Set up your development environment using the notes found in `Project1EnvironmentSetup.pdf`. Carefully follow part 3, "Setting up developing environment", which contains a step-by-step list of how to get your environment up and running.

You do not need to create any files from scratch for this project. Your programming will consist of writing tests and implementing the body of pre-defined functions. **Warning: to match the notation of the text, pattern strings, text strings, etc. are defined to start with index 1 (1-indexed). To simplify this, use the following convention: to represent a string "mystring", use** `std::string S = " mystring"`

*Last updated January 27, 2018

(note the leading space). Strings representing alphabets (e.g. Σ) are 0-indexed as normal.
Throughout this project, when you compile, do the following steps in your project's root directory[1]. In the following commands, I assume you are starting in the root directory.

1. Create a directory called `build` using `mkdir`, or change directory to `build` and delete its contents with `rm -r *`.
2. Run CMake in `build` with `CMake ..`
3. Run Make in `build` with `make` (any unit tests you have defined will execute during this stage.)

References

The following are helpful references for this project.

C++ Standard Library `http://www.cplusplus.com/reference/` (has tutorials) and `http://en.cppreference.com/w/` (terse)

Google Test Documentation `http://cheezyworld.com/wp-content/uploads/2010/12/PlainGoogleQuickTestReferenceGuide1.pdf`, especially page 3, which defines `EXPECT_*` and `ASSERT_*` macros that you must use for all of your tests.

CMake Documentation You should not need to deep-dive into CMake. Instead, you should be able to copy and paste CMake code and make the appropriate changes where necessary (e.g. change which source files are used, target names, etc.). However, if you feel that you need a reference, refer to `https://cmake.org/documentation/`.

Part1: Z-algorithm Exact Matching

Learning objectives: Understand and implement Z-algorithm exact matching as described in the text. Learn how to write unit tests and how unit tests inform your development process while you code.

Files to modify: `Zalgorithm.cpp`, `ZalgorithmTests.cpp`

Instructions: Complete the following steps, *in this order*.

1. Write missing code from `Zalgorithm.cpp` and check its results manually using small examples (keep your text at most 10 characters and pattern at most 3 characters). Be sure to address all `TODO`'s.
2. Write all of the unit tests in `ZalgorithmTests.cpp`. You must use either `EXPECT_*` or `ASSERT_*` for your tests. Use `EXPECT_*` for most of your tests, and use `ASSERT_*` if a failed test results in a segmentation fault (e.g. testing

[1]The root directory will have all of your .cpp, .h files etc.

the size of a container). Make sure all your unit tests pass when you use `make` in the `build` directory.

3. Run the sanity check `ZalgorithmSanityCheck`. If you see errors, you may need to debug code from the previous steps (in the `Zalgorithm.cpp`).

Part2: Boyer-Moore Preprocessing

Learning objectives: Boyer-Moore is another linear time string matching algorithm that we did not study in the class. It has been implemented for you. Your goals are to learn how to build libraries, executables, and sanity check code that you are not necessarily familiar with. The sanity check test your program in a more thorough manner than a set of unit tests.

Files to modify: `CMakeLists.txt`, `BoyerMooreSanityCheck.cpp`

Instructions: Complete the following steps, *in this order*.

1. Add `BoyerMooreSanityCheck` by modifying `CMakeLists.txt`. You should copy, paste, and slightly modify the current code in `CMakeLists.txt` to achieve this.
2. Implement `BoyerMooreSanityCheck.cpp`. See `ZalgorithmSanityCheck.cpp` and follow its example as you implement this program.
3. Add unit tests contained in `BoyerMoorePreprocessingTests.cpp` to the project by modifying `CMakeLists.txt`. You should copy, paste, and slightly modify the current code in `CMakeLists.txt` to achieve this.
4. Notice that when you build with `make`, the unit tests run and pass (are *green*). Experiment: comment out parts of `BoyerMoorePreprocessing.cpp` to purposefully introduce bugs into the code (you don't need to know *what* bugs they are, just break it), then build again and see the tests run and fail (are *red*). Once you are done, revert `BoyerMoorePreprocessing.cpp` back to its previous working condition.

Part3: Exact Matching

Learning objectives: Learn how to sanity check your results using two algorithms to verify results.

Files to modify: `TwoAlgorithmVerificationSanityCheck.cpp`, `CMakeLists.txt`

Instructions: Complete the following steps, *in this order*.

1. Add unit `TwoAlgorithmVerificationSanityCheck` to your project by modifying `CMakeLists.txt`. You should copy, paste, and slightly modify the current code in `CMakeLists.txt` to achieve this.

2. Write the sanity check in `TwoAlgorithmVerificationSanityCheck.cpp`. Once complete, build it, then run `TwoAlgorithmVerificationSanityCheck` in the `build` directory. If you see errors, you may need to debug your code from Part 1 and/or from `TwoAlgorithmVerificationSanityCheck.cpp`.