Dept. of Computer Science, University of California, Davis
ECS122b   Instructor: Rob Gysel   4/19/16

Programming Assignment #1*
Due date: 5/12/17 11:59pm

Programs are to be submitted to Gradescope by the due date. You may work in groups of up to 5 people. Only one submission needs to be submitted to Gradescope, but your entire group should be added to the submission. To check that you have correctly submitted and added your group members, your group members who have not submitted should be able to log in to Gradescope and see what their partner has submitted. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty (submit under ProjectPart*_FirstLateSubmission). Programs submitted between 24-48 hours late will still be accepted but incur a 25% grade penalty (submit under ProjectPart*_SecondLateSubmission). **Your last submission will be the score entered as your grade. If you have a previous submission with a higher grade than your last submission, you should re-submit that submission to ensure you get as many points as possible.**

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.

First, be sure your development environment is set up according to the instructions given in Discussion the first week. If you have not done this yet, notes are at `http://tinyurl.com/ecs122b-env-setup`. Carefully follow part 3, "Setting up developing environment", which contains a step-by-step list of how to get your environment up and running.

*Last updated April 19, 2017

Second, find the skeleton code for the project under Canvas→Files→ProgrammingProject. 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"` (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. Learn how to test your program in a more thorough manner by writing a sanity check.

Gradescope assignment to submit to: ProjectPart1

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

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

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

1. Write all of the unit tests in `ZalgorithmTests.cpp`. All of your tests should pass at this stage because they test code that has already been implemented. You must use either `EXPECT_*` or `ASSERT_*` for your tests.
2. Write the `ZalgorithmBasedMatching` function in `Zalgorithm.cpp` and check its results manually using small examples (keep your text at most 10 characters and pattern at most 3 characters).
3. Write the sanity check in `ZalgorithmSanityCheck.cpp`. Once complete, build it, then run `ZalgorithmSanityCheck` in the `build` directory. If you see errors, you may need to debug `ZalgorithmBasedMatching` from the previous step, or you may need to debug your sanity check in `ZalgorithmSanityCheck.cpp`. You may want to look at `TwoAlgorithmVerificationSanityCheck.cpp` as an example.
4. Submit to Gradescope under Assignments→ProjectPart1.

Part2: Boyer-Moore Preprocessing

Learning objectives: Understand and implement Boyer-Moore preprocessing rules as described in the text. Learn how to build libraries, build executables, and link libraries using CMake.

Gradescope assignment to submit to: ProjectPart2

Files to modify: `BoyerMoorePreprocessing.cpp`, `BoyerMoorePreprocessingTests.cpp`, `CMakeLists.txt`

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

1. Add unit tests to your project by modifying `CMakeLists.txt`. Under the line `"########## TODO: Add unit tests for BoyerMoorePreprocessing (call it BoyerMoorePreprocessingTests) here"`, add unit tests for `BoyerMoorePreprocessing` by following the example for `ZalgorithmTests` defined directly above it.
2. Implement all `CalculateRTests` in `BoyerMoorePreprocessingTests.cpp` and `CalculateR` in `BoyerMoorePreprocessing.cpp`. Be sure to check the contents of `R` and its size. Use `ASSERT` to check the size here in order to prevent a segmentation fault when checking the contents of `R`. All three of these tests should fail (red) at this stage because they test unwritten code.
3. Implement the `CalculateR` function in `BoyerMoorePreprocessing.cpp`. Once implemented, all three tests from the previous step should succeed (green).
4. Repeat steps 2-3 for `CalculateN`. The test defined should exactly match the textbook example in order to guide your programming.
5. Repeat steps 2-3 for `CalculateCapitalLprime`.

6. Implement `CalculateLowercaseLprimeTest` in `BoyerMoorePreprocessingTests.cpp`. The function `CalculateLowercaseLprime` is already implemented, so your test should pass. Once your test passes, check that it is valid by commenting out part of `CalculateLowercaseLprime` (or its entirety) and running the test to make sure it fails.
7. Submit to Gradescope under Assignments→ProjectPart2.

Part3: Boyer-Moore Exact Matching

Learning objectives: Understand and implement the remainder of the Boyer-Moore algorithm as described in the text (this is partially written for you). Learn how to sanity check your results using two algorithms to verify results.

Gradescope assignment to submit to: ProjectPart3

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

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

1. Add unit tests and sanity checks to your project by modifying `CMakeLists.txt`. You should copy, paste, and slightly modify the current code in `CMakeLists.txt` to achieve this.
2. Implement the rest of `BoyerMoore.cpp`.
3. Write the sanity check in `BoyerMooreSanityCheck.cpp`. Once complete, build it, then run `BoyerMooreSanityCheck.cpp` in the `build` directory. If you see errors, you may need to debug `BoyerMoore` from the previous step, or you may need to debug your sanity check in `BoyerMooreSanityCheck.cpp`.
4. Repeat step 3 for `TwoAlgorithmVerificationSanityCheck.cpp`.
5. Submit to Gradescope under Assignments→ProjectPart3.