Project 2: empirical analysis

CPSC 335 - Algorithm Engineering
Summer 2019
Instructor: Kevin Wortman (kwortman@fullerton.edu)

Abstract

In this project you will implement and analyze three algorithms. For each algorithm, you will analyze that algorithm mathematically to derive a big-O complexity class; implement the algorithm in C++; analyze the algorithm empirically, by running it for various input sizes and plotting the timing data; and conclude whether the two analyses agree with each other.

The Hypothesis

This experiment will test the following hypothesis:

For large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm.

The Problems

All three problems involve processing a string.

The problems are:

1. The vowel counting problem:

vowel counting
input: a string S of length n **output:** the number of vowels in S, i.e. the number of upper or lower case A, E, I, O, or U characters

Example: the string "programming" has 3 vowels.

There is a simple greedy algorithm that solves this problem in $O(n)$ time.

2. The longest oreo problem:

longest oreo problem
input: a string S of length $n > 0$ **output:** the longest non-empty substring u of S such that the first and last characters of u are identical

We call a string with the same first and last character an "oreo" because, like an Oreo™ cookie, it looks the same from either end.

Example: "angela" and "declared" are oreo strings. In the string "California" the longest oreo is "alifornia".

There is a greedy algorithm that solves this problem, that takes either $O(n^2)$ or $O(n^3)$ time depending on how careful you are about avoiding unnecessary copying.

3. The longest repeated substring problem:

longest repeated substring problem
input: a string S of length $n > 0$ **output:** the longest non-empty substring u of S such that u appears more than once in S; or an empty string if no such u exists

Examples:
"California" => "a"
"Fullerton College" => "lle"
"Code" => "" (empty string)
"" (empty string) => "" (empty string)

There is a greedy algorithm that solves this problem, that takes either $O(n^3)$ or $O(n^4)$ time depending on how careful you are about avoiding unnecessary copying.

Algorithm Design

Your first task is to design an algorithm for each of the three problems. Write clear pseudocode for each algorithm. This is not intended to be difficult; the algorithms I have in mind are all relatively simple, involving only familiar string operations and loops (nested loops in the case of oreos and substrings). Do not worry about making these algorithms exceptionally fast; the purpose of this experiment is to see whether observed timings correspond to big-O trends, not to design impressive algorithms.

Mathematical Analysis

Your next task is to analyze each of your three algorithms mathematically. You should prove a specific big-O efficiency class for each algorithm. These analyses should be routine, similar to the ones we have done in class and in the textbook. I expect each algorithm's efficiency class will be one of $O(n)$, $O(n^2)$, $O(n^3)$, or $O(n^4)$.

Implementation

You are provided with the following files.

1. `stringcount.hpp` is a C++ header that defines functions for the three algorithms described above. The function definitions are incomplete skeletons; you will need to rewrite them to actually work properly.
2. `unittest.cpp` is a Google Test-based unit test program that checks whether the code in `stringcount.hpp` works or not. You can run the corresponding `unittest` program to see whether your algorithm implementations are working correctly. The unit tests depend on the Google Test library, specifically `/usr/lib/libgtest.a`. If your Tuffix environment does not have this library, the `Makefile` (see below) will automatically install it; just give your password when prompted.
3. `rubricscore.cpp` is a program that cross-references the outcome of the unit tests against the grading point values defined in `rubric.json`. You can ignore this file. You can run the

corresponding `rubricscore` program to see how your code would be graded based on how many tests pass.

4. `timer.hpp` contains a small `Timer` class that implements a precise timer using the `std::chrono` library from C++11.
5. `experiment.cpp` is a C++ program with a `main()` function that measures one experimental data point for each of the algorithms. You can expand upon this code to obtain several data points for each of your algorithm implementations.
6. `README.md` contains a brief description of the project, and a place to write the names and CSUF-supplied email addresses of the group members. You need to modify this file to identify your group members.
7. `Makefile` is a GNU make configuration file that specifies how to compile all this code, run tests, and/or compute a grade score. You can run the command

   ```
   $ make
   ```

   inside your project directory, which will try to compile the unit test and `rubricscore` programs, and provided they build cleanly, run the unit tests and then print out your score. We recommend that you use this `make` command inside Tuffix to confirm that your code works, and preview how your submission will be graded.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education / Tuffix Instructions

Here is the invitation link for this project:

https://classroom.github.com/g/Un9jYVSi

What to Do

First, add your group member names to `README.md`. Implement all the skeleton functions in `stringcount.hpp`. Use the `unittest` program to test whether your code works.

Once you are confident that your algorithm implementations are correct, do the following for each of the three algorithms:

1. Analyze your pseudocode for the algorithm mathematically and prove its efficiency class.
2. Gather empirical timing data by running your implementation for various values of n. As discussed in class, you need enough data points to establish the shape of the best-fit curve (at least 5 data points, maybe more), and you should use n sizes that are large enough to produce large time values (ideally multiple seconds or even minutes) that minimize instrumental error.

3. Draw a scatter plot and fit line for your timing data. The instance size n should be on the horizontal axis and elapsed time should be on the vertical axis. Your plot should have a title; and each axis should have a label and units of measure.
4. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with your mathematically-derived big-O efficiency class.

Finally, produce a brief written project report ***in PDF format***. Submit your PDF by committing it to your GitHub repository along with your code. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 2.
2. Three scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
    a. Provide pseudocode for your three algorithms.
    b. What is the efficiency class of each of your algorithms, according to your own mathematical analysis? (You are not required to include all your math work, just state the classes you derived and proved.)
    c. Is there a noticeable difference in the running speed of the three algorithms? Which is faster, and by how much? Does this surprise you?
    d. Are the fit lines on your scatter plots consistent with these efficiency classes? Justify your answer.
    e. Is this evidence consistent or inconsistent with the hypothesis stated on the first page? Justify your answer.

Grading Rubric

Your grade will be comprised of three parts: *Form, Function,* and *Analysis.*

Function refers to whether your code works properly as defined by the test program. We will use the score reported by the test program, when run inside the Tuffix environment, as your Function grade.

Form refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

Analysis refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function = 9 points, scored by the unit test program
2. Form = 9 points, divided as follows:
    a. README.md completed clearly = 3 points
    b. Style (whitespace, variable names, comments, helper functions, etc.) = 3 points

    c. C++ Craftsmanship (appropriate handling of encapsulation, memory management, avoids gross inefficiency and taboo coding practices, etc.) = 3 points
3. Analysis = 15 points, divided as follows
    a. Report document presentation = 3 points
    b. Pseudocode an mathematical analysis = 3 points
    c. Scatter plots = 3 points
    d. Empirical analysis = 3 points
    e. Question answers = 3 points
4. Total = 33 points

Acceptability standard: As stated on the syllabus, a submission is unacceptable if either it cannot compile cleanly in the Tuffix environment; or if it is a copy of the skeleton code with only superficial changes. Unacceptable submissions will receive 18/33 points.

Deadline

The project deadline is Tuesday, July 23, 11 am.

At some point shortly after the deadline, I will download the main branch of your repository. You will be graded based on what you have pushed as of that moment. Commits made after that point will not be considered. Late submissions will not be accepted.

Be advised that we may use automated tools to detect plagiarised work. Do not plagiarize.