Project 1: empirical analysis

CPSC 335 - Algorithm Engineering

Fall 2017

Instructors: Kevin Wortman ([kwortman@fullerton.edu](mailto:kwortman@fullerton.edu)), Laurence Bernstein ([lebernstein@fullerton.edu](mailto:lebernstein@fullerton.edu))

# Abstract

In this project you will implement and analyze three algorithms. For each algorithm, you will analyze it 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 vector of strings, where each string is a word from the English language.

We will treat the length of words as constant, because words are commonly 12 characters or fewer, and the [longest word in any dictionary is only 45 characters long](https://en.wikipedia.org/wiki/Longest_word_in_English). Therefore, for our purposes the input size will refer to the number of words, not the number of characters, in a vector of strings. Likewise, an operation that loops through one string, such as copying a string or comparing a string, counts as time here.

The problems are:

1. The character mode problem:

|  |
| --- |
| *character mode problem* |
| **input:** a vector of strings that collectively contain at least one character  **output:** the character that appears the most times among all characters in ; when two characters are tied, the output is the tie with the least ASCII code |

We will assume that all strings use the ASCII encoding, so that every character is an integer char value between 0 and 255 inclusive. Under that assumption, the following algorithm solves the problem:

def character\_mode(V):

counts = Array(256, 0)

for string in V:

for character in string:

counts[character]++

c = <find the character c with the greatest value of counts[c] >

return c

1. The longest mirrored string problem. We say that string is a *mirror* of string when and is the reverse of So “cat” and “tac” are mirrored; but “cat” and “dog” are not mirrored, “aaa” and “aaa” are not mirrored, and “” and “” are not mirrored.

|  |
| --- |
| *longest mirrored string problem* |
| **input:** a vector of strings  **output:** a string from whose mirror is also in of maximum length; or the empty string |

The following algorithm solves the problem:

def longest\_mirrored\_string(V):

best = “”

for a in V:

for b in V:

if a is the mirror of b and len(a) > len(best):

best = a

return best

1. The longest subset trio problem. A *substring trio* is a set of three strings and such that all three strings are nonempty and distinct from each other, is a substring of and is a substring of c. We say that string is a substring of string if can be found within ; so “cat” and “tama” are both substrings of “catamaran”. The total length of a substring trio is

|  |
| --- |
| *longest subset trio problem* |
| **input:** a vector of strings  **output:** a vector containing exactly three strings comprising the substring trio of maximum total length; or a vector containing three empty strings when contains no substring trio |

The following algorithm solves the problem:

def longest\_subset\_trio(V):

best\_length = 0

trio = Vector(3, “”)

for a in V:

for b in V:

for c in V:

abc\_length = |a|+|b|+|c|

if <a, b, c are a subset trio> and

abc\_length > best\_length:

best\_length = abc\_length

trio[0] = a

trio[1] = b

trio[2] = c

return trio

# Mathematical Analysis

Your first task is to analyze each of the 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. *Hint:* the efficiency class of each of these algorithms is one of the Big Eight classes listed in section 3.4 of ADITA.

# Implementation

You are provided with the following files.

1. words.txt is an ASCII text file containing thousands of English words, courtesy of Ubuntu Linux. We will use this file as a source of words in the problem instances we use to gather experimental data.
2. project1.hh is a C++ header that defines functions for the three algorithms described above. In addition, it includes the helper functions load\_words, is\_mirrored, and is\_substring. The function definitions are incomplete skeletons; you will need to rewrite them to actually function properly.
3. project1\_test.cc is a C++ program with a main() function that performs unit tests on the functions defined in project1.hh to see whether they work, prints out the outcome, and calculates a score for the code. You can run this program to see whether your algorithm implementations are working correctly.
4. rubrictest.hh is the unit test library used for the test program; you can ignore this file.
5. timer.hh contains a small Timer class that implements a precise timer using the std::chrono library in C++11.
6. experiment.cc is a C++ program with a main() function that measures on experimental data point. Specifically, it loads words.txt into memory; forms a smaller vector containing the first words; runs the longest\_mirrored\_string algorithm implementation on those words; measures the elapsed time with the Timer class; and prints the results to stdout. You can expand upon this code to obtain several data points for each of your three algorithm implementations.
7. README.md contains a brief description of the project, and a place to write the names and CSUF email addresses of the group members. You need to modify this file to identify your group members.

# Obtaining and Submitting Code

This document explains how to obtain and submit your work:

[GitHub Education Instructions](https://docs.google.com/document/d/1UrmpTBOqI3-A6sfvrppEjsaYG1f2-Oa4gLYffiigcyY/edit?usp=sharing)

Here is the invitation link for this project:

<https://classroom.github.com/g/2QQWAGPQ>

# What to Do

First, add your group member names to README.md. implement all the skeleton functions in project1.hh. Use the project1\_test.cc 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 the pseudocode for the algorithm mathematically and prove its efficiency class.
2. Gather empirical timing data by running your implementation for various values of 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 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 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- efficiency class.

Finally, produce a brief written project report ***in PDF format***. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 1.
2. Three scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
   1. Is there a noticeable difference in the performance of the three algorithms? Which is faster, and by how much? Does this surprise you?
   2. 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.)
   3. Are the fit lines on your scatter plots consistent with these efficiency classes? Justify your answer.
   4. 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 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: 40% of project score, divided into 9 points:
   1. load\_words passes all tests: 1 point
   2. is\_mirrored passes all tests: 1 point
   3. is\_substring passes all tests: 1 point
   4. character\_mode passes all tests: 2 points
   5. longest\_mirrored\_string passes all tests: 2 points
   6. longest\_substring\_trio passes all tests: 2 points
2. Form: 30% of project score, divided equally among:
   1. README.md complete
   2. Style (whitespace, variable names, comments, etc.)
   3. Design (where appropriate, uses encapsulation, helper functions, data structures, etc.)
   4. Craftsmanship (no memory leaks, gross inefficiency, taboo coding practices, etc.)
3. Analysis: 30% of project score, divided equally among:
   1. Report document presentation
   2. Mathematical analysis
   3. Scatter plots
   4. Empirical analysis
   5. Question answers

# Deadline

The project deadline is Monday, October 9, 11 am.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.