Project 3: polynomial versus exponential time

CPSC 335 - Algorithm Engineering
Summer 2018
Instructor: Kevin Wortman (kwortman@fullerton.edu)

Abstract

In this project you will implement and compare two algorithms that solve similar but different problems. The first algorithms solves the *longest common substring* problem and runs in $O(n^3)$ time. The second algorithm solves the *longest common subsequence* problem and runs in $O(2^n n)$ time. (Note the difference between "substring" and "subsequence.") You will investigate the difference between polynomial and exponential time complexities.

The Hypotheses

This experiment will test the following hypotheses:

1. Exhaustive search algorithms are feasible to implement, and produce correct outputs.
2. Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

The Problems and Algorithms

The first problem we will study is the longest common substring problem.

longest common substring
input: a string a of length m and a string b of length n *output:* the longest string s such that s is a substring of both a and b; in the case of ties, use the substring that appears first in a

A substring is a contiguous sequence of characters that appears identically in both strings. For example, the longest common substring of "the rain" and "in spain" is "ain" because "ain" appears in both strings. The longest common substring of "cat" and "dog" is "" (the empty string).

The part about ties clarifies what should happen in the case of ties. For example, between a = "act" and b = "cake", both "a" and "c" are tied for longest common substring of length 1. But the correct output is "a" because "a" comes before "c" in a.

The empty string is a substring of all strings, so when a and b are very different, the output will be an empty string. Therefore there is no need for a "None" output.

Implement the following exhaustive search algorithm to solve the problem.

```
longest_common_substring(a, b):
      best = empty string
      for each start index i in [0, m):
            for each length k in [1, m-i-1]:
                  s = substring of a, starting at index i, of length k
                  if b contains s and length(s) > length(best):
                        best = s
      return best
```

The outer loop repeats m times, the inner loop repeats up to $m-1$ times, and the "b contains s" step takes $O(n)$ time, for a total of $O(m(m-1)n) = O(m^2n)$ time. When $m \approx n$, $O(m^2n) = O(n^3)$, so we say this algorithm takes $O(n^3)$ time.

Note that the outer for loop starts with low indices of a and works its way up. This handles the tie situation correctly.

The second problem we will study is the longest common subsequence problem.

longest common subsequence
input: a string a of length m and a string b of length n *output:* the longest string s such that s is a subsequence of both a and b; in the case of ties, use the substring that appears first in a

A subsequence is similar to subset; s is a subsequence of a when it is possible to obtain s by removing characters from a, but leaving the remaining characters in their original order. So "aceg" is a subsequence of "abcdefgh" because it is possible to remove b, d, f, and h from "abcdefgh" to produce "aceg".

The longest common subsequence of "banana" and "atana" is "aana". The longest common subsequence of "abc" and "def" is "" (empty string).

As with the longest common substring problem, the empty string is a valid subsequence of any other string, so there is no need for a "None" output in the longest common subsequence problem.

Implement the following exhaustive search algorithm to solve the problem.

```
longest_common_subsequence(a, b):
      shorter = the shorter of a and b
      longer = the other input string
      best = empty string
      for each candidate sub in subsets(shorter):
            if (sub is a subsequence of longer)
               and
               (length(sub) > length(best)):
                 best = sub
      return best
```

If each input string's length is approximately n, then there are 2^n candidates. The "sub is a subsequence of longer" step should take $O(n)$ time, and comparing lengths takes $O(1)$ time, so the entire algorithm takes $O(2^n n)$ time.

You will need to implement a verification algorithm to detect when sub is a subsequence of longer. Designing the details of that algorithm is part of your assignment.

You will also need to implement the code to generate subsets of `shorter`. This is discussed in detail in section 6.5.4. The pertinent pseudocode is on page 163.

Implementation

You are provided with the following files.

1. `exhaustive_subsequence.hpp` is a C++ header that defines skeleton functions for the two algorithms described above. The subsequence verification part is broken out as a helper function. You are responsible for implementing these three functions.
2. `exhaustive_subsequence_timing.cpp` is a unit test program, similar to the one in project 2.
3. `exhaustive_subsequence_timing.cpp` is a timing program for experimental results, similar to the one in project 2.
4. `README.md`, `rubrictest.hpp`, `timer.hpp`, and `Makefile` function essentially the same as in project 2.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education Instructions

Here is the invitation link for this project:

https://classroom.github.com/g/3Tnkawpw

What to Do

First, add your group member names to `README.md`.

Then design an algorithm for verifying a candidate subset. Write pseudocode, and then implement your algorithm in the `detect_subsequence` function.

Then implement the other two skeleton functions. Use the test program to check whether your code works.

Once you are confident that your algorithm implementations are correct, do the following:

1. Gather empirical timing data for each of the two algorithms by running them for various values of n.
2. Draw two scatter plots. Each plot should have the instance size n on the horizontal axis, elapsed time on the vertical axis, a fit line, title, and labels on both axes.
3. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with the mathematically-derived big-O efficiency classes.

Finally, produce a brief written project report ***in PDF format***. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 3.
2. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
    a. Write your pseudocode for subsequence detection.
    b. Analyze your subsequence detection algorithm; state and justify a time efficiency class.
    c. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
    d. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
    e. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.
    f. Is this evidence consistent or inconsistent with hypothesis 2? 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 = 11 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 = 12 points, divided as follows
    a. Report document presentation = 3 points
    b. Scatter plots = 3 points
    c. Algorithm pseudocode and analysis = 3 points
    d. Other question answers = 3 points

Legibility standard: As stated on the syllabus, submissions that cannot compile in the Tuffix environment are considered unacceptable and will be assigned an "F" (50%) grade.

Deadline

The project deadline is Friday, June 22, 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.