Project 2: empirical analysis

CPSC 335 - Algorithm Engineering
Fall 2018
Instructors: Doina Bein (dbein@fullerton.edu), Kevin Wortman (kwortman@fullerton.edu)

Abstract

In this project you will design, implement, and analyze two algorithms for the same problem. For this problem, you will design two separate algorithms, describe the algorithms using clear pseudocode, analyze them mathematically, implement your algorithms in C++, measure their performance in running time, compare your experimental results with the efficiency class of your algorithms, and draw conclusions. The first algorithm has a tractable (polynomial) running time, while the second algorithm has an intractable (exponential or factorial) running time.

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 or factorial running times are extremely slow, probably too slow to be of practical use.

The Longest Increasing Subsequence Problem

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique. The longest increasing subsequence problem can be formulated as follows.

longest increasing subsequence
input: a vector V of n comparable elements **output:** a vector R containing the longest increasing subsequence of V

There is a complicated algorithm that solves this problem in $O(n \log n)$ time[1], but we will be implementing two simpler algorithms with slower time complexities.

[1] Schensted, C. (1961), "Longest increasing and decreasing subsequences", Canadian Journal of Mathematics 13: 179–191, doi:10.4153/CJM-1961-015-3

The End-to-Beginning Algorithm

There is a straightforward algorithm that solves the problem and has $O(n^2)$ time complexity. The algorithm uses an additional array *H* of length *n*, of non-negative integers. The value *H[i]* will indicate how many elements, greater or equal to than *A[i],* are further in the sequence *A* and have some special property.

Initially, the array *H* is set to 0 (all the elements are 0). The algorithm proceeds by attempting to increase the values of *H* starting with the previous to last element and going down to the first element. Then a longest subsequence can be identified by selecting elements of *A* in decreasing order of the *H* values, starting with the element in *A* that has the largest *H* value.

Algorithm End_to_Beginning

Step 1. Set all the values in the array *H* to 0.
Step 2. Starting with *H[n-1]* and going down to *H[0]* try to increase the value of *H[i]* as follows:
Step 3. Starting with index *i+1* and going up to *n-1* (the last index in the array *A*) repeat Steps 4 and 5:
Step 4. See if any element is bigger than *A[i]* and has its *H* value also bigger to *H[i]*.
Step 5. If yes, then *A[i]* can be followed by that element in an increasing subsequence, thus set *H[i]* to be 1 plus the *H* value of that element.
Step 6. Calculate the largest (maximum) value in array *H*. By adding 1 to that value we have the length of a longest increasing subsequence.
Step 7. Identify a longest subsequence by identifying elements in array *A* that have decreasing *H* values, starting with the largest (maximum) value in array *H*.

An Exhaustive Algorithm

There is an exhaustive algorithm that solves the problem and has $O(n \cdot 2^n)$ time complexity. The algorithm generates all possible subsequences of the array *A* and tests each subsequence on whether it is in increasing order. The longest such subsequence is a solution to the problem.

We note that a subsequence can be uniquely identified by the set of indices in the array *A* that are part of the subsequence. For example, given the array
A = [1, 0, 2, 1, 5, 3, 13, 8, 34, 21, 89, 55, 233, 143]
the subsequence *R* = [1, 0, 3] is uniquely identified by the set of indices {0,1,5} of the elements in the array A. To generate all possible subsequences, one may consider generating the power set of {0,1,...n-1} and consider each subset of the power set as the set of indices in A of the subsequence.

There are several ways to generate the power set. One way is to implement an iterative algorithm that uses a stack to grow and shrink the set as needed. The benefit of this approach is that it prints the subsets in lexicographic order. Below find an implementation[2] in C++ that generates the power set of the set {1, 2, …, n} (where you can specify n in the program):

[2] http://www.programminglogic.com/powerset-algorithm-in-c/

```
void Powerset (int n)
// function to generate the power set of {1, .., n} and retrieve the best set
int *stack, k;
  stack = new int[n+1]; // allocate space for the set
  stack[0]=0; /* 0 is not considered as part of the set */
  k = 0;
  while(1) {
    if (stack[k] < n) {
          stack[k+1] = stack[k] + 1;
          k++;
    }
    else {
          stack[k-1]++;
          k--;
    }
    if (k==0) break;
  }
  delete [ ] stack; // deallocate space for the set
  return;
}
```

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/3mk_0Gx1

Implementation

You are provided with the following files.

1. `subsequence.hpp` is a C++ header that defines functions for the two algorithms described above. Some function definitions are incomplete skeletons; you will need to rewrite them to actually work properly.
1. `subsequence_timing.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.

2. `timer.hpp` contains a small `Timer` class that implements a precise timer using the `std::chrono` library from C++11. It is used by `subsequence_timing.cpp`.
2. `Makefile`, `subsequence_test.cpp`, `rubrictest.hpp`, and `README.md` work the same way as in project 1.

What to Do

Decide on who will be in your team, or decide to work alone; have one of your team members accept the GitHub assignment by following the invitation link; have any other team members join your team by following the invitation link; and add your group member names to `README.md`.

Then, implement each of the two algorithms in C++ using the provided skeleton code. Test your code using the provided unit tests.

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

1. Analyze your pseudocode mathematically and write its efficiency class using Big-Oh notation. (You need to compute the total number of steps of the algorithm.)
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. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
    a. Provide pseudocode for your **two** 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 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 = 6 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 = 20 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 = 8 points
    e. 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, November 2, 1 pm.

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.