CPSC 335
Fall 2014
Project #3 — sorting

Introduction

In this project you will implement and analyze three string sorting algorithms:

1. an $O(n^2)$ time algorithm implemented in Python;
2. the same algorithm, this time implemented in a fast programming language such as C++; and
3. an $O(n \log n)$ time algorithm implemented in Python.

The goal is to compare the performance impact of changing implementation choices, such as which programming language to use, against the impact of switching to a different algorithm with a theoretically-faster efficiency class.

The hypotheses

This experiment will test three hypotheses:

1. A mathematically-derived efficiency class for an algorithm will accurately predict the run time of the algorithm's implementation, regardless of which programming language is used.
2. Lower-level languages such as C++ are faster than higher-level languages such as Python, by some multiplicative constant.
3. $O(n \log n)$-time algorithms outperform $O(n^2)$-time algorithms for large n, regardless of the low-level implementation choices made.

Sorting strings

This project involves implementing three algorithms, each of which sorts a list of strings. To simplify matters I have provided you with an input file `beowulf.txt`. The file contains the words from the Project Gutenberg edition of *Beowulf*, limited to only ASCII letter characters, and newline characters separating the words. The first few lines look like this:

```
The
Project
Gutenberg
EBook
of
Beowulf
This
eBook
is
for
the
use
of
anyone
anywhere
at
no
cost
and
with
almost
no
restrictions
whatsoever
You
may
```

To form an input of size n, use first n words in the file. The file has 40,707 words so we are limited to $n \leq 40,707$.

Implementation 1: an $O(n^2)$-time sort in Python

First, implement an $O(n^2)$-time sorting algorithm in Python. You are welcome to use the out-of-place or in-place selection sort covered in class. If you prefer, you may implement a different $O(n^2)$-time sorting algorithm, such as insertion sort or bubble sort.

As with previous projects you need to also implement a timing harness that

1. loads the input file,
2. allows you to specify a value of n to use,
3. prints the first 10 words,
4. sorts the first n words,
5. measures the elapsed time of the sorting process,
6. prints out the first 10 words in the sorted sequence, and
7. prints the elapsed time.

We print out the first few words before and after the sorting process as a quick way of confirming that the loading and sorting processes appear to be working.

You may reuse the Python harness code that was provided with Project 1.

Implementation 2: the same $O(n^2)$-time sort in a faster language

Next, implement the same $O(n^2)$ sort algorithm in a "fast" lower-level language. Python has a reputation for being "slow," so we expect this implementation 2 to be faster than implementation 1. I expect most groups will use C++, but you are also authorized to use C, Java, or C#. If you would prefer to use a different fast language, you must get my permission first, and it must be a compiled and statically typed language.

You will also need to implement the same kind of timing harness so that we can collect empirical timing data and confirm this implementation's correctness. I have provided you with sample high resolution timing code in C++11. If you choose to use a different language, you are responsible for figuring out how to do high resolution timing in that language.

Implementation 3: $O(n \log n)$-time sort in Python

Finally, implement merge sort, out-of-place randomized quick sort, or in-place randomized quick sort in Python. As usual you will need a timing harness.

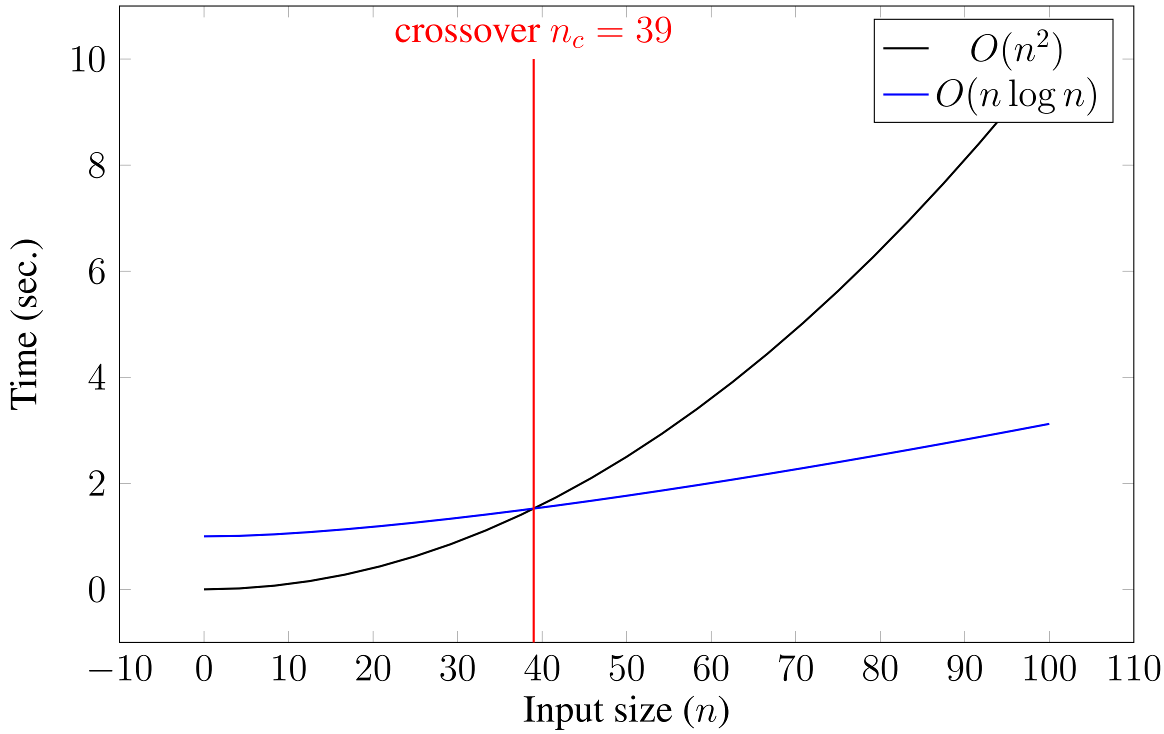
Sample output

The output of my C++ selection sort implementation:

```
requested n = 200
loaded 200 lines from 'beowulf.txt'
first 10 words: [The][Project][Gutenberg][EBook][of][Beowulf][This][eBook][is][for]
selection sort...
first 10 words: [AN][ANGLOSAXON][Act][An][AngloSaxon][Author][BEOWULF][BEOWULF][BOSTON][BY]
elapsed time: 0.00051 seconds
```

Run-time crossover point

According to our mathematical analyses, the $O(n^2)$-time algorithms will be faster for small values of n while the $O(n \log n)$-time algorithm will be faster for large values of n. We expect there to be a *crossover point* which is an input size n_c such that the $O(n^2)$-time algorithms are faster when $n < n_c$ and the $O(n \log n)$-time algorithm is faster when $n > n_c$.



Determining the crossover point between your implementation 2 and 3 is part of the project.

Deliverables

Produce a written project report. Your report should include the following:

1. Your name(s), CSUF-supplied email address(es), and an indication that the submission is for project 3.
2. Two scatter plots:
   (a) One showing the run time of all three implementations, zoomed out so that the quadratic curves are clear.
   (b) One zoomed in to show the crossover point.
3. An output for $n = 200$ for all three implementations.
4. Your complete Python source code for implementations 1 and 3.

5. Your complete source code (in C++ or similar language) for implementation 2.

6. Answers to the following questions, using complete sentences.

   (a) Which $O(n^2)$ algorithm did you choose to implement, and why?

   (b) Which $O(n \log n)$ algorithm did you choose to implement, and why?

   (c) Which of the three algorithms did you find most difficult to implement, and why?

   (d) Are your empirical results consistent or inconsistent with hypothesis 1? In other words, do the run times of both implementation 1 and 2 fit quadratic curves?

   (e) Are your empirical results consistent or inconsistent with hypothesis 2? In other words, are the run times of your implementation 1 greater than those of implementation 2 by a constant factor? If so, approximately what is that factor, as a percentage? Does this result surprise you?

   (f) Are your empirical results consistent or inconsistent with hypothesis 3? In other words, is there a crossover point n_c for which implementation 3 is faster than implementations 1 and 2? If so, what is the approximate value of n_c? How much faster is implementation 3 over implementation 2, as a percentage, for the full $n = 40,707$? Does this result surprise you?

   (g) Based on these results, which approach do you think is a better way of implementing algorithms efficiently: implementing a slow algorithm in a fast low level language (implementation 2), or implementing a fast algorithm in a slow high level language (implementation 3)? Why? What are the implications on software development in general?

Your document *must* be uploaded to TITANium as a single PDF file.

Due Date

The project deadline is Thursday, 10/30, 11:55 pm. Late submissions will not be accepted.