Advanced Topics in Comp Sci Project—Sorting Algorithm Analysis

ASSIGNMENT OVERVIEW

In this assignment, you'll implement a designated sorting algorithm, do a theoretical analysis of that algorithm's efficiency, collect actual performance data of your implementation, and compare your performance data with what you predicted.

BACKGROUND

Different algorithms have different efficiencies, often expressed using Big-O Notation. Common Big O-values for sort algorithms include

- $O(n^2)$ for a Bubble Sort
- $O(n^2)$ for a Selection Sort
- $O(n^2)$ for an Insertion Sort
- $O(n \log n)$ for a Merge Sort
- $O(n \log n)$ for a Quick Sort
- $O(n \log n)$ for a Shell Sort

These efficiencies don't reflect how quickly the algorithm works on a given machine with a given list of values. Big-O notation indicates how the execution time for an algorithm increases as a function of an increasing number of values.

For example, if a Bubble Sort algorithm on your computer takes 2 millisecond to sort 50 values, how much time would we predict it that it would take to sort 100 values? The number of values n that we're sorting has increased by a factor of 2, and Bubble Sort is an $O(n^2)$ algorithm, so we expect that the amount of time it will take increases by a factor of $(2)^2 = 4$, or about 4×2 milliseconds = 8 milliseconds.

A formal strategy for predicting an algorithm's efficiency is presented at the end of this document.

PROJECT SPECIFICATION

This project consists of four steps that you'll need to complete in preparation for submitting a final document, a formally written research report, submitted electronically as a PDF file. Once you have been assigned a sorting algorithm, you'll need to:

1. Implement the designated sorting algorithm using Python.
2. Develop a theoretical analysis of your sorting algorithm's efficiency using an analysis as outlined in class and at the end of this document.
3. Collect actual performance data of your implementation using Python's `time` module.
4. Compare your measured performance data from (3) with what you predicted in (2).

Your final report will include a formal presentation of the work that you've done, including:

0. **Abstact**
   A short one-paragraph summary of this project, what you did and what you figured out
1. **Introduction**
   An introduction to the project
2. **The Sorting Algorithm**
   A verbal description of your algorithm's sorting strategy
3. **Source Code**
   The source code of your Python script implementing that sorting algorithm, with a liberal amount of documentation/comments.
4. **Big-O Analysis**
   The theoretical predicted analysis of the sort's efficiency.

5. **Performance Data**
   Data tables from your actual performance data collected by running your program
6. **Graphical Results**
   A graph of that data, with a regression if possible
7. **Summary**
   A summary of your findings at the end of the report
8. **References**
   A series of references to any sources used in the development of your project. Include people you worked with, webpages you looked at, textbooks you read, etc.

The final report, including cover sheet, will probably be from 6-10 pages.

DELIVERABLES

`sorting_algorithm_analysis.pdf`

To submit your assignment for grading, copy `sorting_algorithm_analysis.pdf` to your directory in `/home/studentID/forInstructor/` at *crashwhite.polytechnic.org* before the deadline.

ASSIGNMENT NOTES

- Once you know which algorithm you'll be implementing, you'll need to identify a reference where you can learn about that algorithm. We'll be investigating the original versions of these algorithms, not versions that have been optimized by modifying them to run faster.
- *Write your own Python implementation of your sorting algorithm.* One of the intentions of this assignment is for you to get practice thinking about an algorithm and figuring it out how to implement it. It's easy to find Python versions of these algorithms online and/or in our book, but you shouldn't be using these programs as a reference.

Do not use other people's Python code as a reference in this assignment!

- Examining a pseudocode version of the algorithm may be useful if you get stuck, but your goal is to be able to write the program from scratch. Make sure you've given that a try first before looking at any pseudocode.
- You'll need to collect time data for a number of data sets of different sizes in order to determine the performance of your algorithm. A spreadsheet program may be useful in organizing and graphing this information.
- To empirically determine the performance of the algorithm, you can use the spreadsheet's regression tool, possibly, although an *n log n* performance isn't easily determine. If you suspect that your algorithm's performance is *n log n*, you might want to consider graphing *time* versus *n log n*. What shape would you expect that curve to have if the algorithm is *n log n*?
- You'll be submitting a formal written report for this assignment. Consider using one of the following media for writing up your results:
  - Microsoft Word / Microsoft Excel
  - LibreOffice Writer / LibreOffice Calc (an open source alternative to Microsoft's applications. This document itself was written using LibreOffice.)
  - GoogleDocs
  - Markdown (a system for including formatting markup in text documents. You'll still need to use a spreadsheet/graphing application to prepare your graphs.)

- HTML (You'll still need to use a spreadsheet/graphing application to prepare your graphs.)
- LaTeX (This is a markup language used primarily by math and physics people to present research. This option is beyond what most people will want to use for this assignment.)

Regardless of which medium you use for writing up your results, your final submission will be a PDF version of your report.

GETTING STARTED

1. Using an appropriate reference, identify the algorithm that you've been assigned to implement. Read and study the algorithm itself, written explanations of how the algorithm works, and possibly graphical presentations of how the algorithm operates.

2. Become familiar enough with the algorithm that you can take a list of values on paper and sort them using the given strategy. A good gauge of your understanding is whether or not you can sit down with someone else and explain the algorithm to them, and show them how it works on paper.

3. The implementation of your sorting algorithm probably won't take much code. Consider including your entire codebase into a single file. For example, a BubbleSort program might include:
   a. a `generate_random_numbers` function to fill an array
   b. a `bubblesort` function that sorts the array
   c. a `display` function that can be used to display the array at any point in time (useful for debugging)
   d. a `main` function that calls the functions and times the `bubblesort` function

EXTENSIONS

1. Create an animated visual (graphical) display of your sorting algorithm. Use Python's `turtle` module or Processing's Python module to implement your animation. (See https://www.youtube.com/watch?v=kPRA0W1kECg for examples of what this can look like.)

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

2. Why, exactly, is it useful for us to study sorting algorithms when Python already has a number of very good implementations of sorts available?

3. The instructor of this course wrote a similar assignment to this one during my Computer Science education. Sadly, copies of that assignment are no longer available. (Look up the word processor *Wordstar* for further info.) What are the advantages and disadvantages of writing this report in Microsoft Office? Google Docs? Plaintext and/or HTML?

4. Is a PDF file a stable format for preserving written documents? Why or why not?

SAMPLE ALGORITHM ANALYSIS

An algorithm's execution time T(*n*) typically changes as a function of the size of the problem *n*. A good way of approximating this time analytically is by counting the number of *assignment operations* that take place during the execution of the algorithm. As well, for our sorting analysis, we'll also be counting the number

of *comparisons* that are performed.

Examples of performing a T(*n*) analysis are included in our textbook.

We can get a convenient summary of a T(*n*) function by considering its dominant term, the value that increases faster than everything else in that function. This term controls by an "order of magnitude" the performance of the algorithm, and is written in "Big-O notation" as O(*n*) for the function.

Some algorithms have an execution time that changes only as a function of the size of the problem. Other algorithms have an execution time that changes depending on the actual values that are being manipulated. (Some sorting algorithms, for example, are especially efficient at sorting values that are already mostly in order.) In the case of these algorithms, the Big-O notation will be annotated with "best case," "worst case," and "average case" to indicate performance under those conditions.