CS2001 week 10 practical:
Complexity in action

Due 21:00 Wednesday Week 10
25% of practical mark for the module

Goal

To explore how a search algorithm's speed changes with problem size.

Background

When we compare the performance of algorithms, we typically speak in terms of the best-case, worst-case, and expected- or average-case asymptotic time complexities. The word *asymptotic* is important here, as it denotes the complexity as the problem size n tends to infinity – in other words, the complexities of "sufficiently large" problems. The issue here is that no real-world problem is infinite – and indeed many of them are not "sufficiently large" either. What happens to algorithms for "small" problems – problems of sizes that are actually more likely to be encountered in many circumstances? Exploring this issue is the goal of this practical.

The assignment

Begin by writing an implementation of merge sort, and an implementation of selection sort, in Java. (We discussed both of these in class, in more or less detail. You can also find descriptions of the algorithms on the web.) It's important that you code-up and understand your own versions of these algorithms, however, because....

Then instrument your algorithms to collect the amount of time they spend sorting a sequence: you might do this by grabbing the system time in milliseconds as you start sorting, grabbing the system time again when you finish, subtracting one from the other, and outputting the resulting elapsed sort time.

Then test the two algorithms on random sequences of different length – so for different values of problem size n. You should find that selection sort, whose time complexity is $O(n^2)$, is faster than merge sort, which has complexity $O(n \log n)$. At some point, however, for some n, the times taken will "cross over", making merge sort faster. Collect and present data to find where selection sort ceases to be the faster algorithm. You will probably find that you have to compare several tens of different sequence lengths to find the cross-over point accurately.

Requirements

You should submit two elements to MMS:

1. Your code, including any tests you implemented to check your work
2. A short (500—1000 word) report presenting your results (including a graph) and describing how you went about collecting the results and why they can be trusted as supporting your conclusion.

Marking

The practical will be graded according to the grade descriptors used for CS21001/CS2101, which can be found at:

https://studres.cs.st-andrews.ac.uk/CS2001/Assessment/descriptors.pdf

A good solution worthy of a top grade would consist of well-designed, -tested- and -explained instrumented algorithms, a clear experimental protocol for performing the data collection, and a detailed analysis of how the behaviour of the algorithms varies in terms of size.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice:

https://www.st-andrews.ac.uk/students/rules/academicpractice/

Hints for completing the practical

(Use or ignore as you see fit.)

This practical is *not* really about coding: it's about *assessment* of code. What matters is the way the code slows down (or speeds up), not how slow (or fast) it is. There's therefore no point in optimising anything: it's more important to have simple implementations that you understand and can instrument accurately.

Write your own code: don't use a pre-packaged solution, despite the temptation for such well-known and frequently-implemented algorithms. You have to instrument the code to collect data, and that can be a nightmare when editing some else's code. It's far easier to start from code you know yourself.

Drawing graphs of timings can be done with Java, with Excel, with Python, with R, with gnuplot, or with a host of other programs. Using Java would in my opinion be *the most complicated way possible* to generate such graphs – but don't let that stop you if you're determined.

Timing is often affected by external factors like other processes, so it is often worth conducting repeated runs against the same data. This can be used to generate error bars, if you like, or the results for a single sequence length could be averaged.

Make sure you spend time on your experimental design: how are you going to conduct the

experiments? Will they be repeatable, so that someone else could re-perform your experiments exactly to check your results? This is often made easier by automation. The hardest and least accurate way would be to run the program repeatedly by hand: that's just asking for mistakes to be made.