CS2001/CS2101 Week 10 Practical: Complexity in Action

Peter Macgregor <prm4@st-andrews.ac.uk>

Due date: Wednesday 19th November, 21:00
MMS shows practical weighting and definitive date and time for deadlines

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturers regarding any queries well in advance of the deadline.

Goal

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

Background

We have seen several algorithms for sorting an array of data. These algorithms vary in several respects:

- some sorting algorithms are in-place, while some require additional memory;
- the algorithms differ in worst-case time complexity;
- some algorithms have different worst-case and average-case time complexity;
- some algorithms are more difficult to implement than others.

In this practical, we will explore these differences in practice. We will implement several algorithms, measure their performance in practice, compare their performance to our expectations based on the theoretical analysis, and come to a conclusion about which algorithms to use in practice.

The Assignment

Begin by writing a Java implementation of

- selection sort,
- quicksort (with a fixed pivot), and
- merge sort.

Each algorithm should accept an array as input. You are free to choose the datatype stored in the array. Instrument your algorithms to record the amount of time they spend sorting a sequence. You might do this by storing the system time in milliseconds as you start sorting, storing the system time again when you finish, subtracting one from the other, and outputting the resulting elapsed sort time.

Design a set of experiments to measure the running time of each algorithm on a range of inputs of different lengths. Create two versions of the experiments - one with data in a random order, and another with data in a predictable order - in order to investigate the difference between the worst-case and average-case complexities.

Run your experiments, and collect the information about the running time of the algorithms. Write a report with the following content.

- A description of the design and implementation of the algorithms and experiments.

- A presentation of the results of your experiments. How does the running time of each algorithm scale with problem size? How does this compare to what is expected in theory? You might like to include figures.
- A recommendation of which algorithm to use in practice. Would you recommend different algorithms for different problem sizes?

The report has an *advisory* word limit of 1000 words.

Requirements

You should submit two elements to MMS (compressed into a single zip file):

- Your code, including any tests you implemented to check your work, with a README file describing how to run your experiments.
- Your report as a PDF file.

Marking

The practical will be graded according to the grade descriptors used for CS2001/CS2101, which can be found at:

https://studres.cs.st-andrews.ac.uk/CS2001/0-General/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 A: 1 mark per 24 hour period, or part thereof):

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

Good Academic Practice

As usual, you are reminded to follow the relevant guidelines on good academic practice as outlined at

https://www.st-andrews.ac.uk/education/handbook/good-academic-practice/.

See also the following page of the CS student handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html.

Hints for completing the practical

1. This practical is not about coding: it is about *assessment* of code. What matters is the way the code *scales*, not how fast it is.
    - There is therefore no point in optimising your sorting algorithms. It is more important to have simple implementations that you understand and can instrument accurately.
    - Since you are free to choose the datatype in the array, you might like to keep it simple by handling only integers.
2. Write your own code: do not 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: it is far easier to start from code you write yourself.

3. Drawing graphs of running times can be done with Java, with Excel, with Python, with R, with gnuplot, or any number of other programs. Using Java is one of the more difficult ways to draw graphs.

4. 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 could just be averaged.

5. 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 *easily* re-perform your experiments to check your results?

6. As with any argument about computational complexity, things often only work asymptotically, so you will need to generate suitably large sequences to showcase the asymptotic complexities.

7. Presentation of the report matters: make sure that figures are a sensible size; all text is readable; your key ideas, results and conclusions are easily identifiable; and the overall presentation of the report is tidy.