# 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 pathological cases affect its speed.

## Background

We've looked at various sorting algorithms, some of which we found to have pathological cases where a particular configuration of inputs led to radically different (usually worse) performance. For quicksort, for example, we showed that submitting an already-sorted list to the algorithm led to the worst possible execution time – even though in principle there was no work to be done.

Clearly there's a spectrum of pathological cases here. A fully-sorted list is worse than a pretty-much-sorted list, which itself is worse than a randomised list. But how does the sortedness of the list affect its pathological behaviour? How un-sorted does a list need to be to get good performance from the algorithm? – or, conversely, how does the degree of sortedness cause performance to degrade with respect to the randomised case? What does "degree of sortedness" even mean as a precise, scientific, statement?

## The assignment

Begin by writing an implementation of "ordinary" (not in-place) quicksort. Instrument your algorithm to collect the amount of time it spends sorting a sequence: you might do this by grabbing the system time in milliseconds as you start sorting, subtracting this start time from the end time when the sequence is sorted, and outputting the resulting sort time.

Then decide what it means for a list to be sorted, slightly sorted, or unsorted. (There are some hints later, but there are several possible definitions.) This number is often called a *metric*: it measures something about the data. Work out how to create sequences that correspond to your metric.

Then run your quicksort algorithm against different sequences with different metrics. Use this to generate a graph of how changing your metric changes the time taken to sort sequences with that metric. From this, conclude how sortedness affects sort time for quicksort.

## 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 conclusions, being careful to describe how you went about collecting the results and why they can be trusted as supporting your conclusions.

## Marking

The practical will be graded according to the guidelines in the student handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html

Credit will be allocated 20% for the code and 80% for showing the algorithms' performance profile using appropriate data and charts.

A 17 on this practical would be a report that shows robust about the behaviour of the algorithm as sortedness changes. Extra credit could come from additional characterisations of the algorithm, or from demonstrating and comparing the same approach against a different sorting algorithm, or convincingly showing the absence of pathological cases for some algorithm.

## 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, ensure you are following the relevant guidelines on good academic practice as outlined at

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

## Hints for completing the practical

(Use or ignore as you see fit.)

This is not 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.

Write your own code: don't use a pre-packaged implementation, You have to instrument the code to collect data, and that can be a nightmare when editing some else's code. You also need to be sure that the algorithm you're using is actually vanilla quicksort and doesn't do optimisations that interfere with its behaviour. It's far easier to start from code you know yourself.

Drawing charts of timings can be done with Java, with Excel, with R, with gnuplot, or with a host of other programs. Java would be *the most complicated possible way* to generate such charts, in my opinion – 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 of the same data. This can be used to generate error bars, if you like, or the results for a single sequence could be averaged.

As with any argument about computational complexity, things only work asymptotically, so you'll

need to generate suitably large sequences.

There are several different measures of sortedness. One of the easiest is what's referred to as the *edit distance* between two sequences A and B: how many elements need to move to turn A into B? (You can find full descriptions online: there are several variations of the basic idea.) You could use this by, for example, starting from a sorted sequence and then swapping pairs of values at random, where the more swaps you do the less sorted the list becomes.

How many repetitions will you need to do at each point? How will you decide?

Clearly doing the experiments is going to be quite repetitive, so it would be sensible to set things up to run automatically. You could do this with a single Java program that tests multiple sequences, or with a program that sorted one sequence run from a shell script that repeated multiple tests – or some other way, as long as you get the data. It's easiest if the data automatically ends up in a file, from which you automatically generate the graphs you need.