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 using the last element of the sequence as pivot. You'll recognise that this is setting the algorithm up for a fall, in that we *know* there are pathological cases for its behaviour: that's the point.

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 time spent actually sorting.

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 and lets you talk about a sequence that's "less sorted" than another. 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, and whether you can conclude anything about how sorted a sequence has to be to trigger a significant slowdown.

Requirements

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

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 CS2001/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 sortedness.

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.

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.

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.

As with any argument about computational complexity, things often only work asymptotically, so you'll need to generate suitably large sequences.

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.

You'll need to know how the algorithm behaves in non-pathological cases – perhaps with a random sequence?

Is there just one pathological situation? Or are there several?

Make sure you spend time on your experimental design: how are you going to conduct the experiments? How many repetitions will you need to do at each point? How will you decide? 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. The easiest way is to automate the process, using either Java or a shell script, and dump the results into files you can then process further.

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.