

Lab Activity: Sorting and the Big-O

Using *Order-of-Magnitude Algorithm Analysis* to Assess Software Efficiency

Introduction

Achieving lightening-fast performance in software solutions, is not a matter of luck. It involves a careful assessment of the performance of each algorithm.

In this lab activity, you'll experiment with a number of sorting algorithms by measuring the time taken to complete the sort of ever increasing data sets. Using this experimental evidence, you'll be able to identify the *Big-O* pattern, and predict the time required to sort even larger data sets.

That's the whole point of *Big-O* analysis: *as you increase the size of the data set being processed, how does the performance change.* In this case, you'll be systematically scaling to larger and larger data sets: *2-fold* increase; *3-fold* increase, *5-fold* increase, *10-fold* increase, etc.

Ultimately, you'll be experimenting with four different sort algorithms: *selection sort*, *insertion sort*, *merge sort*, and *qsort* (*qsort* is a C-implementation of the *quicksort* algorithm). But you'll start with *selection sort* and *insertion sort*.

The Details

Launch the Program

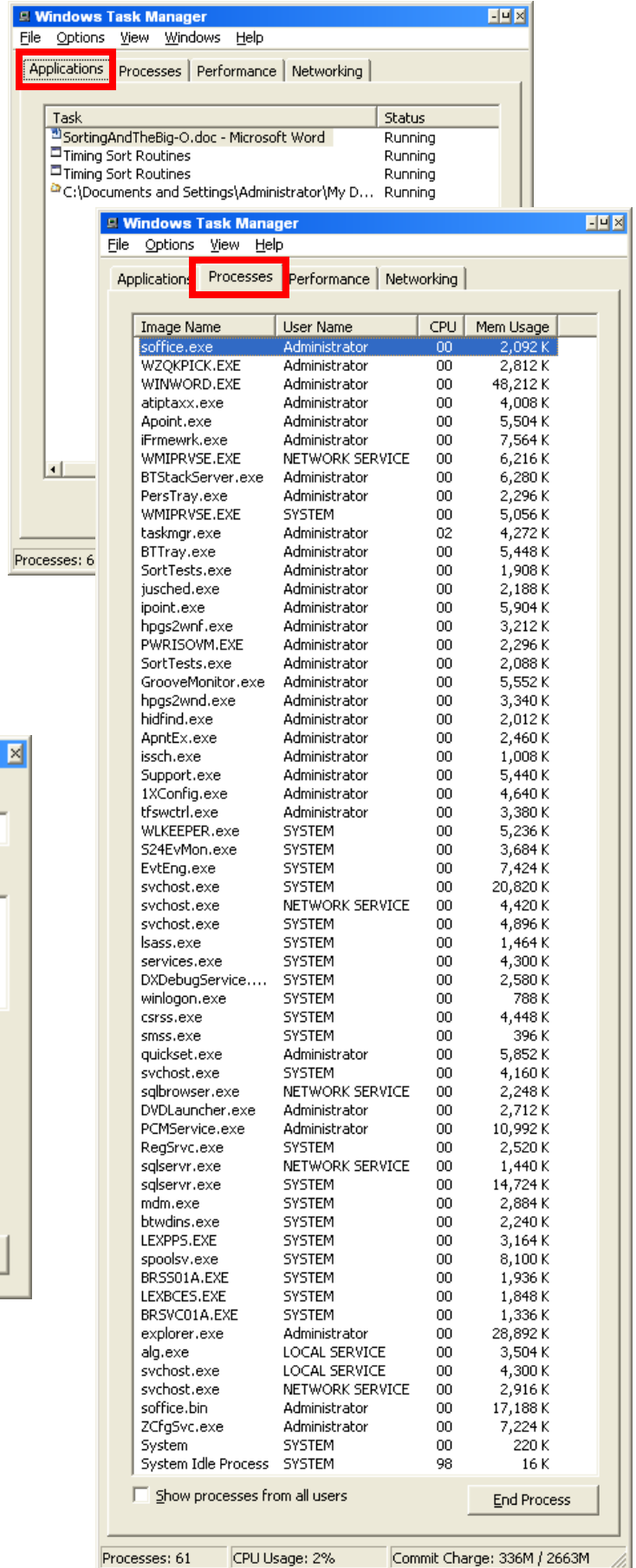
I wrote a simple Windows program to help perform this experimental analysis. You can find it on my website. I would suggest downloading it and storing it in your own workspace (it's only 11 K in size). Here's a screen capture of the freshly launched program.

Preparing to Sort

You'll have to choose a sort algorithm (only *Selection* and *Insertion* are currently available) and choose a size for your data set. The actual time taken to sort will vary from machine to machine. Primarily, the machine-to-machine time differences depend on the speed of the CPU, and the number of other programs currently running (and, of course, these other programs do demand CPU cycles). To get a sense of the configuration on your machine, launch *Task Manager*¹.

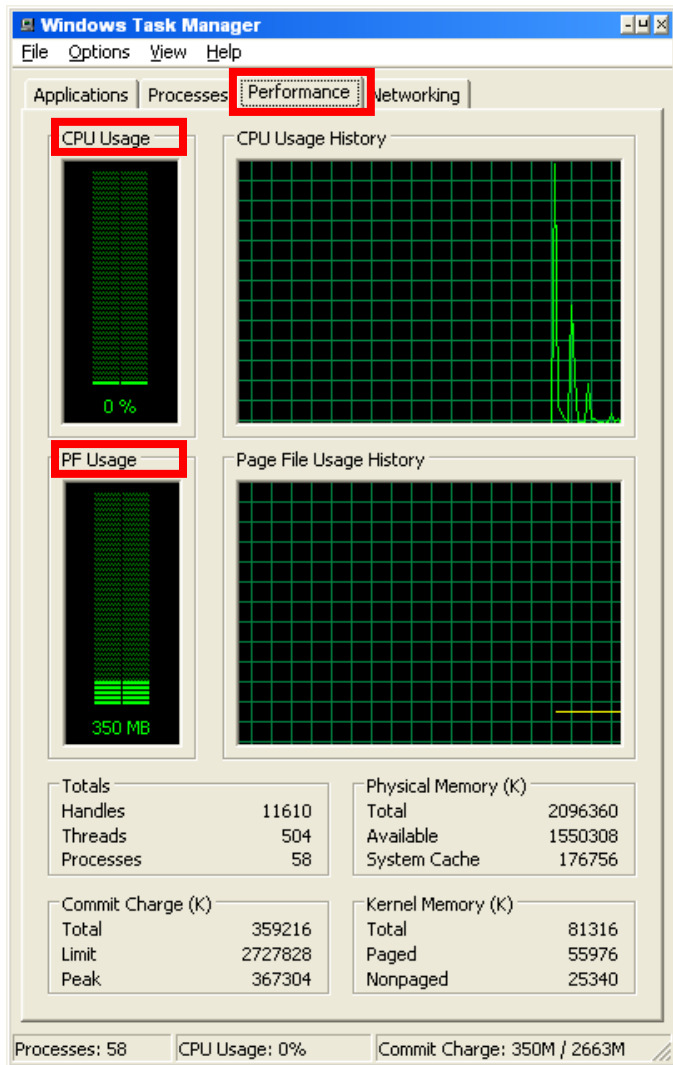
Based on my screen capture, you'll notice that my machine had four applications running, but this list is deceptive. At any given time, there could be dozens to hundreds of *processes* running. Click on the *Processes* tab to view all programs currently running.

Ultimately, you'll want to monitor the performance of the sorting algorithms by viewing the *Performance* tab of *Task Manager*.



¹ You access *Task Manager* by using <Ctrl-Alt-Del> to spawn the *Windows Security* dialog box, and then select *Task Manager*.

Here's a screen shot of my machine's performance. When I captured this image, very little was happening on my computer, hence *CPU Usage* shows up at 0%. When you perform your sorting experiments, you'll see this and *PF Usage* change. *PF Usage* tracks the memory use (both real RAM and virtual memory).



Experimenting with Selection Sort

Start by sorting a very small number of integers. Today's computers are so fast, that you're likely to record a time of 0.00 seconds for anything less than 1,000. I'll leave it for you to find a suitable starting size (since your computer will likely have different performance than mine), but look for a small size that takes at least 0.1 second. Choose a size that is easily multiplied (for example, 2,000 would be a far better choice than 2,433).

Fill out the chart below with your starting size and the time taken to sort.

In each of the subsequent rows, increase the data set size by some multiple (remember, you'll experiment with the following kinds of growth: 2-fold increase; 3-fold increase, 5-fold increase, 10-fold increase).

The column labeled *Proportional Increase* identifies the factor by which you chose to increase the data set. This will be one of 2, 3, 5 or 10. This item is grayed in the first row since it's the starting size and hasn't been scaled from a previous experiment.

In the column labeled *Proportional Change*, you'll need to identify the factor by which *Time to Complete* changed as you scaled to a larger data set. You don't want exact numbers with decimal places, you want an approximation. That's because this experimental evidence for *The Big-O*. And the *O* in *The Big-O* stands for *order of magnitude*. For the sake of argument, imagine that the first sort (with the smallest data set) took 0.24 second. Your second sort doubled the size of the data set and imagine that *Time to Complete* registered as 0.87. The *Proportional Change* is between 3 and 4.² It's closer to 4, so record 4.

Chart of Performance: Selection Sort

| Size | Proportional Increase | Time to Complete | Proportional Change |
|------|-----------------------|------------------|---------------------|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

If you tried to sort an enormous set of numbers, say 100,000,000, you would have waited a very long time. On my computer, based on *Big-O* analysis, sorting 100,000,000 numbers with *Selection Sort* would have taken well over 2 years to complete.

You might have to use *Task Manager* to kill a sort experiment that seems to be running endlessly.

Experimenting with Insertion Sort

Now that you've finished your experiments with *Selection Sort*, perform similar tests with *Insertion Sort*.

² An arithmetic calculation of the change shows that 0.87 is 3.625 times greater than 0.24. But because you are concerned with the *order-of-magnitude*, you would round that to 4 times greater.

