Week 09 Lab Exercise

Sort Detective

Objectives

- to familiarise yourself with practical aspects of computational complexity
- to practice a systematic approach to problem solving
- to apply sound scientific reasoning in reaching conclusions
- to hone your analysis skills
- to identify algorithms from their behaviour

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week09 Lab or Week 10 Lab
Submit	`give cs2521 lab09 report.pdf` or via WebCMS
Deadline	must be submitted by 11:59pm on **Sunday 9 August**

You can do this lab either as a pair or as an individual. The name and zID of all participants must appear at the top of the submitted document (`report.pdf`). No name, no marks.

You should **start early** and aim to briefly write up Phase 1 of the report before your Week 09 Lab. Discuss your Phase 1 write-up with your tutor. Your next aim should be to complete Phase 2 by your Week 10 Lab. Discuss your results with your tutor before submitting the report.

Background

A very long time ago, in a burst of enthusiasm, Richard Buckland wrote a bunch of sort programs. Sadly he forgot to give the programs meaningful names and when he passed them on to later COMP2521 lecturers to use for this lab, he couldn't tell us which program implemented which algorithm. All that he could remember** is that the algorithms he used came from the following list:

- Oblivious Bubble Sort
  - unstable and unoptimised bubble sort
- Bubble Sort With Early Exit
  - stable bubble sort that terminates when there have been no exchanges in one pass
- Insertion Sort
  - standard insertion sort

- Selection Sort
    - standard selection sort
- Merge Sort
    - normal merge sort
- Vanilla Quick Sort
    - normal quick sort (pivot is the last element)
- Quick Sort Median of Three
    - pivot is the median of first last and middle elements
- Randomised Quick Sort
    - list is shuffled then vanilla quick-sorted
- Shell Sort with Powers of Four
    - shell sort with intervals ..., 4096, 1024, 256, 64, 16, 4, 1
- Shell Sort similar to Sedgewick
    - shell sort with intervals ..., 4193, 1073, 281, 23, 8, 1
- Psuedo-Bogo Sort
    - choose two random array elements, if out-of-order then swap, repeat

(** which means that it's pointless trying to extract the algorithm identities by capturing and torturing him)

Despite not knowing what algorithm was in each program, Richard did remember a few things about the programs. One thing he remembered is that all of the programs read their input from `stdin` and write the sorted version of the data to `stdout`. Sorting happens line-by-line, like the Unix `sort` program does it. Another thing he remembered is that there is a limit on the size of the input each program can process (1,000,000 lines), because they read their input into a fixed size array and sort it there, before writing out the sorted result. The other thing Richard remembered is that **the programs all expect each line to start with a number, which acts as the sorting key**. The sorting is numeric, which means that the programs all behave something like the Unix `sort` program run with the `-n` option. If no number is present at the start of a line, it will be treated as a zero value. If the file has no numbers at all, the final ordering will depend on the stability of the sorting algorithm.

```
$ ./sortX < data > sorted_data
# behaves like
$ sort -n < data > sorted_data
```

One difference between Richard's sorting programs and the Unix one, is that many of his are stable, but the Unix sort is not (by default). Unless the algorithm itself is inherently unstable, or unless noted above, Richard's sorting programs are stable.

Your task, in two stages, is to help us identify the specific sort algorithms used in two of these programs. You will not be able to view the source code, instead you will have to try to identify the algorithms using only the programs' observable behaviour when sorting different data. Note that since the programs are only available in binary format, they will most likely only run on the CSE machines, so you'll have to do your testing work there.

In the setup phase, we will give you access to two different sort programs; each lab pair or individual gets a different (randomly chosen) pair of programs. The **first phase** of the task is to design and write up the experimental framework that you plan to use to solve the problem. In the **second phase**, you should gather data on the execution behaviour of the two programs according to your exerimental setup. You then add the data to your report and analyse it to reach a conclusion on which algorithm each of the supplied programs contains.

To make your task a little simpler, we've supplied a program to generate data in the correct format for the sorting programs to use.

```
$ /web/cs2521/20T2/labs/week09/gen  5  R
1 nwl
5 arz
4 hcd
2 rbb
3 mqb
$ /web/cs2521/20T2/labs/week09/gen
Not enough arguments
Usage: gen  N  A|D|R  [S]
       N = number of lines
       A|D|R = Ascending|Descending|Random
       S = seed for Random
$
```

Use the **gen** program however you like, or not at all. Note that the **gen** program always generates a unique set of keys (from 1..N). This won't enable you to test stability, so you'll need to come up with some more data of your own. Note that the *seed* argument allows you to generate the same sequence of "random" numbers; if you want to test both sort programs on the same random sequence, use the same seed.

Note that the `setup` script will put a copy of the **gen** executable into your lab directory, so you can run it as `./gen` rather than having to type the log file name.

Setting Up

Set up a directory for doing this lab and `cd` into it. To obtain your sorting programs for analysing, run the command:

```
$ /web/cs2521/20T2/labs/week09/setup
```

This must be run on the CSE machines, either via vlab or ssh. This command will set up two symbolic links in your directory called `sortA` and `sortB`. These reference the executable programs under the class account. Note that you do not have read permission on the executables, in order to remove the temptation to reverse engineer the algorithm from the binary code.

The `setup` script also makes a copy of the **gen** program and a copy of the `runtests` shell script (see below) into your lab directory.

You can check that the `sortA` and `sortB` programs actually do sorting by running something like the following:

```
$ ./gen  5  R
... unsorted output ...
```

```
$ ./gen 5 R | ./sortA
... sorted output ...
$ ./gen 5 R | ./sortB
... sorted output ...
```

If you're working in a pair, only one person in the lab pair needs to run `setup`. If you both do it, you'll end up with different pairs of programs. Each Lab Pair should work together on one pair of programs.

Phase 1: Designing your Experiment

Design, plan and document the set of tests which you will later use to deduce which sorting algorithm is used by each sort program. Once you have done this, show it to your tutor and explain to them how you think the tests will allow you to identify the algorithms.

This is to ensure that by the time you begin investigating and experimenting with the actual sort programs you have thoroughly thought about what kind of behaviour to look for and what further experimentation might be necessary when analysing your findings.

We expect this will involve coming up with numerous sequences of test data to use, and what differences (and why) you expect to be able to observe from different types of sorting algorithms. Typical properties to look for are execution time and output stability.

Of course, when designing tests you cannot anticipate all possible results which might occur during your experiment. This is the nature of scientific experimentation. But by formalising what you expect to occur and how you will respond, you can better account for unexpected behavior and sensibly revise your design or create new tests once the experiment is under way.

Your experimental design should detail the tests you have devised and explain, with clear reasons, how you will be able to distinguish each algorithm you might be given. You do not need to include all the unsorted input data you intend to use, only a description or small sample of it (you may put this in the appendix if you wish).

Write up the experimental design as Part 1 of your report. You can produce the report using whatever tools you want (e.g. OpenOffice, Google Docs, raw HTML, using your WebCMS blog, etc.), but it must eventually be submitted as PDF. Most document processing systems and Web browsers can produce PDF.

There is no size requirement for the report; it is *content* not length which counts, but as a guide we expect the average report will be 1-2 A4 pages for the experimental design and 2-3 A4 pages for the experimental results and analysis. If you want to include detailed reporting of timing results and/or output checking, then put these in an appendix. Your report should be clear, scientific/systematic in approach, and all reasoning and assumptions should be explicit. Make sure you ask your tutor questions if you are unclear about what is expected.

To help you get started, a template for the report is available. Note that a fault in many of the reports in COMP2521 in the past is that they simply report observations without attempting to analyze them or explain *why* these results occurred. For this lab try to get beyond saying just "This is what I saw" and including things like "These observations can be explained by ...".

Phase 2: Run Experiment and Analyse Results

The `setup` command has given you two sort programs to identify. As noted earlier, each sort program reads from its standard input and writes to its standard output, and assumes that each input line contains a numeric key (first field) and an arbitrary string following the key. The output should be in ascending order, based on the numeric ordering of keys. All programs should produce the same output for a given input when the keys are unique.

The following examples show some useful ways of running the sort programs, and auxiliary commands to help collect useful data on their behaviour:

```
# generate some data, put in a file called ""mydata"
$ ./gen 100 R > mydata
# count the number of lines in the data (should be 100)
$ wc -l mydata
# sort the data using sortA, put the result in "sortedA"
$ ./sortA < mydata > sortedA
# sort the data using sortB, put the result in "sortedB"
$ ./sortA < mydata > sortedB
# count the number of lines in "sortedA" (should also be 100)
$ wc -l sortedA
# sort the data using Unix sort
$ sort -n < mydata > sorted
# check that the sortA and sortB programs actually sorted
$ diff sorted sortedA   should show no diffs
$ diff sorted sortedB   should show no diffs
# run a large sort and throw away the result
$ ./gen 10000 R | ./sortA > /dev/null
# repeat the above, but get timing data on sortA
$ ./gen 10000 R | time ./sortA > /dev/null
# repeat the timing, but with better output format
$ ./gen 10000 R | /usr/bin/time --format="%U seconds" ./sortA > /dev/null
```

You should now carry out the experiment you designed in Phase 1. Collect and record all of the data, and then summarize it in your report. You can use whatever tools you like to produce useful summaries (e.g. plot graphs of time vs data size). Then *analyze* the data, draw conclusions, and explain them.

To help with the experiments, we have provided a shell script called `runtests` to (surprise!) run tests and collect timing data. As supplied, this script tests the built-in `sort` program, which is not helpful for you, so you'll need to modify it to use one of your `sortA` or `sortB` programs. You could use it as follows:

```
# check how to use the runtests script
$ sh runtests
Usage: sh runtests sortA|sortB
# run all tests using sortA, writes log file
$ sh runtests sortA
# save a copy of the test log for sortA
$ mv log logA
# run all tests using sortB, writes log file
$ sh runtests sortB
```

```
# save a copy of the test log for sortB
$ mv log logB
```

You can modify this script as much as you like to fit in with the tests that you devise. You don't even need to use the script at all if you're happy to run all of your test cases manually.

Note that some tests will take a l-o-n-g time to run with large data. You can remove the large data sizes from the outer `for` loop if you can't wait, but you should probably add more smaller sizes to get more data points to try to determine execution cost trends. Unfortunately, students cannot leave jobs running in background after logging out, so you'll need to stay logged in to a CSE machine while you're running tests.

Note also that some tests will be so fast that they register 0.00 user seconds for every run. The data for these tests is too small, so (a) you can ignore them, (b) you might want to tweak the `runtests` script to start from larger data.

Once you've run the tests, you might want to edit the `log` files to clean out the irrelevant stuff.

Tips for measuring: the Unix `time` command works by sampling and will likely produce different results for the same program run multiple times on the same data. The `runtests` script runs your program multiple times and records the timings, each of which will be a little bit different. Take an average over a number of timings to account for this. Also, beware of claiming too much accuracy. You can't really claim more than one or two significant digits on an average from the `time` command.

The precise format of your report is up to you, but it must include:

- a summary of the results for each program
- an argument, based on the observed behaviour, on what each program is

If you want to include detailed results, put them in an appendix.

Submission

Your submission for this lab is a report containing an experimental design and results/analysis from carrying out the experiment. It should be submitted as a PDF file called `report.pdf`.

Have fun, *jas*

Appendix: Tips on Writing the Report

Note that these tips come from an assignment that was used in this course several years ago. Since this is a lab and not an assignment, the scale will be smaller.

Bad things

- Not checking your spelling
- Not proofreading your report to check for other mistakes
    - e.g. writing "sortA" when you are actually talking about sortB

- Trying to perform meaningful analysis using very small timing data
  - e.g. 0.004s vs 0.005s. Too much noise! Aim for readings in seconds, not milliseconds.
- Focusing on absolute speed instead of on growth rates (O(n), O(n^2), etc)
  - e.g. "this ran fast", "this ran slow". What does that tell you? How do you account for constants?
- Trying to compare the absolute speed between your sorting algorithms
  - e.g. "sortB was faster than sortA". We could have implemented each sort in differing languages or with differing levels of optimisation, so it is meaningless to compare them this way. You may have had a "very slow" mergesort and a "very fast" bubble sort.
- Not being specific enough about how you decided on a particular sort algorithm
- Writing a table full of numbers and then leaving the units out (seconds, milliseconds, hours.. ?)
- Assuming a sort algorithm was stable for every input just because it was stable on a particular input you gave it
  - It is easy to show that a sort algorithm is unstable; just find an example. It is harder to prove that a sort algorithm is stable for every possible input that you could give it
- Being vague when you could be specific
  - e.g. "much less", "slightly better", "increase a lot", "fast"
- Not being specific enough about your testing data
- Not stating assumptions and reasons clearly
- Making incorrect assumptions about the way our sorts were implemented (based on what?)
- Not being skeptical enough
  - e.g. "obviously", "clearly", ...
- Putting filler in your report
  - e.g. putting filler in your report
- Not understanding how the sorting algorithms work
- Not taking report presentation seriously enough

Good things

- Trying to check whether an algorithm was not stable
- Clearly stating your expectations, testing a hypothesis
  - e.g. "i'm expecting..."
- Being skeptical
  - e.g. checking whether the sorts were actually sorting all your input, or dropping some?
- Trying to overcome timing accuracy errors
  - e.g. repeating tests and taking an average, using large inputs that gave times in seconds rather than milliseconds
- Giving reasons for claims you make
  - "Vanilla insertion sort is stable because..."