COMP26120
Academic Session: 2022-23

Lab Exercise 2: Spellchecking (Experimental Analysis)

Duration: 1 week

You should do all your work on the lab2 directory of the COMP26120_2022 repository - see Blackboard for further details.

This lab uses the results of Lab 1. If you did not complete Lab 1 then you can obtain the Lab 1 model solutions by refreshing your repository from upstream.

To refresh your repository from upstream, do the following:

```
git remote remove upstream
git remote add upstream https://gitlab.cs.man.ac.uk/t95229ld/comp26120_2022_base.git
```

This removes any previous `upstream` (it should be correct, but just in case) and then sets the correct one. If you are using cut-and-paste to transfer these commands to the command line, then watch out that the underscores (_) don't get lost!

Once you have done this run the following commands:

```
git fetch upstream
git merge upstream/master
```

The solutions are in a directory labelled *model_solutions* in the lab1 directory. You will need to copy your implementation (or the model solution) for `darray` (and, for C, also for `sorting`) into the relevant Lab 2 sub-directory.

Reminder: It is bad practice to include automatically generated files in source control (e.g. your git repositories). This applies to automatically generated files from LaTeX as well.

Learning Objectives

By the end of this lab you will be able to:

- Design an experiment to explore the relation between theoretical complexities and practical running times; present and analyse experimental results

Introduction

In this lab, you will design and carry out two experiments to explore the algorithms you implemented in Lab 1 and the relationship between their theoretical and practical complexity. The idea is to attempt to connect the theory that you have been learning about to the reality of real programs.

Questions. The experiments address the following questions:

1. What are the *best* and *worst* cases for the algorithm and can we predict the running time accurately?
2. Can we use experiments to find the complexity of the *average* case?
3. Under what conditions is it better to perform linear search rather than binary search?

The first two questions are closely related and are addressed by the first experiment. The third question concerns the idea of *amortisation* (the last topic in the complexity section of the course).

Write-Up. It is important to keep notes on your experimental design and results. These should be written up as a LaTeX report. We have provided a template for this report in *report.tex.* Since this is a formative lab, there is no requirement to write a report, however you will need to do something similar for our summative labs (Lab 3 and Lab 5) and this gives you an opportunity to attempt to write an appropriate report and get feedback from a GTA in a drop-in session.

Time. This lab is intended to take two hours of time to complete. *However*, actually generating data and running experiments is likely to take more time than that, but you can leave your computer running and do something else while this is happening – particularly if you write scripts to automate data generation and running experiments. You should plan your time with this in mind.

Generating Inputs

For your experiments you will need to generate various inputs and pass them to `speller_darray`. Here we provide some hints on how to do this. We don't provide sets of pre-made data for you too use as generating the appropriate data is part of the experimental design.

Please **DO NOT** include the data you generate for dictionaries and queries in your git lab repository. This can cause the repositories to become very large and interfere with the marking process for later labs.

Getting input with different structures. You can use the Unix `sort` command to get a sorted version of your dictionary. You should specify the `-d` flag to get the same dictionary order you're using in your sorting functions. You can also use the `-r` flag to do a reverse sort. The `sort` command is likely to use merge sort but may have other sort algorithms implemented (see the man page on your system).

Sometimes it might be interesting to look at *almost sorted* inputs and examine how sorting algorithms perform. This is a common *average* case in some applications. There are two ways to get an almost sorted input. Either permute an already sorted file or start sorting and give up before we're finished. We've provided the `AlmostSort.java` program that takes the second approach - see the comments in the file for how to use this.

Using one dictionary. You may need to run with different sizes of dictionaries or query files. Below is an example of how you can use a few bash commands to automate this process using a single `dict` file and the `tail` command.

```
echo "x" > x
a=( 10000 20000 30000 40000 50000 )
for i in "${a[@]}"; do tail -$i dict > t; time ./speller_darray -d t -m 1 x; done
```

You may need to edit this for your purposes. Note that here we use a dummy input file as the assumption is we're focussing only on the sorting algorithm (note that an empty input file won't trigger sorting).

Generating a large dictionary. If you want to generate a large dictionary you can use the provided `random_strings.py` file. Running `python random_strings.py > dict` will (by default) produce about a million random strings. It's 'about' as it might produce duplicates, but the probability is small. Edit the script to produce a different size dictionary.

Generating a dictionary and input file together. For the last experiment you are also interested in the query input file and its relation to the dictionary.

We have provided two scripts for generating dictionaries and input files with particular properties.

`generate.sh` was written by Giles Reger, a former unit lead. Read the help at the top of the file for full details but as an example running

```
 sh generate.sh random dict query 1000 500  reverse 10
```

will produce two files `dict` and `query` of length 1000 and 500 respectively with words taken from `random` such that `dict` is sorted in reverse and only 10% of the queries are in the dictionary. Running with a very large input dictionary will take a while!

`generate.py` was written by Nathan Taylor, a COMP26120 student in 2021-2022. It works much like `generate.sh` but was intended to work faster on large dictionaries (but obviously has not been as extensively used as `generate.sh` which is why we are providing both). You can get usage instructions by typing

```
 python3 generate.py
```

but as a working example running

```
 python3 generate.py random dict query 1000 500  reverse 10
```

will produce two files `dict` and `query` of length 1000 and 500 respectively with words taken from `random` such that `dict` is sorted in reverse and only 10% of the queries are in the dictionary. Running with a very large input dictionary will take a while!

Experiment One - Predicting Performance

In the description of this experiment we focus on *insertion sort*. If you have time (or interest) you may also like to duplicate the experiment for quick sort and perhaps other sorting algorithms you have explored.

Write Up

You can find a template report, `report.tex`, to fill in. We recommend that you fill this in as you go. So you start by writing the hypothesis section for each experiment. Then write the experimental design section. Once you have designed the experiment you can run it and generate results. Then fill in the results section.

Experimental Process

Best Case. Theoretically, the best case for insertion sort is $O(n)$ when the input is already sorted.

As a first step you should confirm that the behaviour of insertion sort on sorted input is, indeed $O(n)$. You should attempt to find a function $f(n)$ to predict the *running time* of the algorithm given the size of it's input. Given the asymptotic complexity we expect this function to be of the form $f(n) = c_1 + c_2 \times n$ e.g. some static cost of setting up the algorithm plus some cost taken for every element of the array. You should design some experiments to find a suitable value for c_2 and work out if c_1 is negligible or not.
Here are some hints:

- You should be able to do all of the analysis you need to do for this whole lab using a simple spreadsheet application. However the unix utility `gnuplot` and the python `matplotlib` library also both allow you to create plots from data and fit lines to those plots.
- As the numbers are all small you might want to count n in lots of 10k e.g. for an input of 10,000 say it's $n = 1$, for 20,000 say it's $n = 2$ etceteras. Alternatively, you may wish to measure time in milliseconds rather than seconds. This is just to make numbers look more sensible.
- For insertion sort we suggest running experiments with dictionary sizes up to 50k in length but for quick sort you will need to go bigger to get useful numbers as the algorithm is faster.
- If you're pre-sorting your dictionary make sure you sort with respect to the same order that your sorting algorithm(s) use.
- You can use the UNIX `time` utility to measure running time. This gives you times for `real`, `user` and `sys`. The most accurate way to judge the running time of your algorithm is to take the sum or `user` and `sys` (`real` includes the time when other resources might be using your CPU and can't account for computation time across multiple cores).

To be really thorough, once you have determined a values for c_1 and c_2, you would generate a much larger dictionary and validate your predicted running time.

Worst Case. Now we want you to do the same for the worst case e.g. confirm that it's behaviour on the theoretical worst case is as expected, and find a function of n to predict its running time (which will be based on the theoretical worst case with some constants that you need to find).

Average Case. Next we want you to experimentally explore the *average* case by producing some *average* (random) inputs and fitting a function $f(n)$ to them to estimate running times. For insertion sort, does this function use n or n^2? This should just involve replicating the above process for random data. Once you have an estimated $f(n)$, is this what you expect theoretically?

Experiment Two - Is it worth Sorting?

In this experiment we want you to explore when it is worth *amortising* the cost of sorting over many binary search calls. Let us recall the algorithmic problem being solved in this exercise:

> Given a dictionary D of k words and another file F of n query words, find all query words in F that are not in D.

The general algorithmic approach being taken is as follows

```
sort(D)
for w in F:
  if not find(D,w) then print w
```

where in the linear approach `sort(D)` does nothing and the cost of `find(D,w)` is $O(n)$ and in the sorting approaches the cost of `sort(D)` is greater but at a reduced cost for `find(D,w)`. Thus, when sorting we *amortise* the cost of sorting across all of the calls to `find`. However, when is it worth it? **Your job is to design an experiment to answer that question.**

To understand under what conditions sorting + binary search will perform better than linear search we need to first understand what conditions may vary. The main ones to consider are as follows (you may add your own):

- Size of dictionary D e.g. the value k
- Size of query file F e.g. the value n
- The structure of dictionary e.g. is it already sorted, sorted backwards, random
- The structure of queries e.g. are most words in the dictionary, or not
- The sorting algorithm used
- The initial size of the data structure

To design your experiment you should decide which of these you will vary, how you will vary them and to justify your decisions. Remember, you want to find out how these effect whether it is worthwhile sorting or not.

You should begin by considering how you expect the different conditions to impact the performance of different approaches given your knowledge about their *theoretical* complexities. This should suggest certain points of interest and you should design your experiment to examine these. For example, if we want to find the point where linear search and insertion sort + binary search should behave the same then we might consider solving

$$\underbrace{n \times k}_{\text{Linear search}} = \underbrace{k^2}_{\text{Sorting}} + \underbrace{n \times (log\ k)}_{\text{Binary search}}$$

for some fixed values of n or k. But recall, there will be some constants involved!

Hint: If you want to keep things simple then just vary k and n whilst using a reverse sorted dictionary, none of the queries being in the dictionary, and the insertion sort algorithm.

This experiment is a bit more open-ended than the first. The important thing to remember is that you need to make your hypothesis clear and back up your findings with data. You may want to consider graphing your data to see if it makes any trends or relationships more obvious.

Due to the variability of how you approach this question, how you have implemented Lab 1, and how you collect your data, **we expect different students to find different answers, backed up by different data**. We are not expecting a single answer but want you to produce well-designed experiment leading to a justified answer backed up by data.

Write Up

You can find a template report, `report.tex`, to fill in. We recommend that you fill this in as you go. So you start by writing the hypothesis section. Then write the experimental design section. Once you have designed the experiment you can run it and generate results. Then fill in the results section.