COMP26120
Academic Session: 2019-20

Lab Exercise 5: Spellchecking

Duration: 4 lab sessions.

The lab is made up of 3 parts but these are not equal in difficulty. You may wish to skip the extension activities in the earlier parts at first and come back to them if you have time. This is a big lab with lots of parts - don't leave it all to the last minute!

For this lab exercise you should do all your work in your *ex5* branch.

For this assignment, you can only get full credit if you do not use code from outside sources.

Note on extension activities: the marking scheme for this lab is organised so that you can get 80% without attempting any of the extension activities. These activities are directed at those wanting a challenge and may take considerable extra time or effort.

Reminder: it is bad practice to include automatically generated files in source control (e.g. your git repositories). This applies to object files and binaries and also to PDF and auxiliary files created by LaTeX. This is a general poor practice but for this course it can also cause issues with the online tests.

Learning Objectives

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

- Explain how and when to use linear and binary search
- Describe and compare different sorting algorithms, their implementation and complexity, and related concepts such as stability
- Design an experiment to explore the relation between theoretical complexities and practical running times; present and analyse experimental results
- Explain amortisation and when it can be useful
- Describe the basic structure of a binary search tree as well as the basic operations and their complexities
- Describe the role of the hash function in the implementation of hash tables and describe and compare various hash collision strategies
- Write C code to implement the above concepts

Introduction

The aim of this exercise is to use the context of a simple spell-checking program to explore some algorithmic concepts such as *search*, *sorting*, *amortisation*, and data structures such as *binary search trees* and *hash tables*.

The completed programs will consist of several ".c" and ".h" files, combined together using *make*. You are given a number of complete and incomplete components to start with:

- *global.h* and *global.c* - define some global variables and functions
- *speller.c* - the driver for each spell-checking program, which:
  1. reads strings from a dictionary file and *insert*s them in your data-structure
  2. reads strings from a second text file and *find*s them in your data-structure
     - if a string is not found then it is assumed to be a spelling mistake and is reported to the user, together with the line number on which it occurred
- *set.h* - defines the generic interface for the data-structure
- *darray.h* and *darray.c* - define a dynamic array. You will need to edit this in Part 1.
- *sorting.h* and *sorting.c* - include prototypes for sorting functions. You will need to edit this in Part 1.
- *bstree.h* and *bstree.c* - includes a partial implementation for binary search trees that you need to complete in Part 3a.
- *hashset.h* and *hashset.c* - includes a partial implementation for hash sets that you need to complete in Part 3b.

You are also given a makefile and a some data files to use for testing your code. You will probably want to create more data files of your own for testing and the experiments in part 2.

> **Note:** The code in *speller.c* that reads words treats any non-alphabetic character as a separator, so e.g. "non-alphabetic" would be read as the two words "non" and "alphabetic". This is intended to extract words from any text for checking (is that "-" a hyphen or a subtraction?) so we must also do it for the dictionary to be consistent. This means that your code has to be able to deal with duplicates i.e. recognise and ignore them. For example, on my PC */usr/share/dict/words* is intended to contain 479,829 words (1 per line) but is read as 526,065 words of which 418,666 are unique and 107,399 are duplicates.

Data structures

There are four data structures relevant to this lab. We briefly overview them here.

Set. This is an abstract datatype that is used to store the dictionary of words. The operations required for this application are:

- `initialize_set`. This creates the set
- `find` whether a given value (i.e. string, alphabetic word) appears in the set.
- `insert` a given value in the set. Note that there is no notion of multiplicity in sets - a value either appears or it does not. Therefore, if `insert` is called with a duplicate value (i.e. it is already in the set) it can be ignored.

There would usually also be a `remove` function but this is not required for this application. We also include two further utility functions that are useful for printing what is happening:

- `print_set` to list the contents of a Set.
- `print_stats` to output statistical information to show how well your code is working.

In this exercise we use three different data structures to implement this abstract datatype. We describe these briefly below but you may need to look at the recommended textbooks or online to complete your knowledge.

Dynamic Array. The majority of you will already have met this data structure as Java's `ArrayList` in COMP16121 or otherwise. The idea is to wrap-up an array so that it dynamically resizes on demand.

Binary Search Tree. You can think of a tree as a linked list where every node has two 'children'. This allows us to differentiate between what happens in each sub-tree. In a binary search tree the general idea is that the 'left' sub-tree holds values smaller than that found in the current node, and the 'right' sub-tree holds larger values.

Hash Table. The underlying memory structure for a hash table is an array. A hash function is then used to map from a large 'key' domain to the (much smaller) set of indices into the array, where a value can be stored. When two values in the input domain map to the same index in the array this is called a *collision* and there are multiple ways to resolve these collisions. To use a hash table to represent a set we make the key and value the same - the result is usually called a *hash set*.

Running your code

To test your implementation, you will need a sample dictionary and a sample text file with some text that contains the words from the sample dictionary (and perhaps also some spelling mistakes). You are given several such files, and you will probably need to create some more to help debug your code.

You should also test your program using a larger dictionary. One example is the Linux dictionary that can be found in */usr/share/dict/words*.

Compile and link your code using *make darray* (for part 1), *make bstree* (for part 3a), or *make hash* (for part 3b).

When you run your spell-checker program, you can:

- use the $-d$ flag to specify a dictionary.
- (for part 3b) use the $-s$ flag to specify a hash table size: try a prime number greater than twice the size of the dictionary (e.g. 1,000,003 for */usr/share/dict/words*).
- use the $-m$ flag to specify a particular mode of operation for your code (e.g. to use a particular sorting or hashing algorithm). You can access the mode setting by using the corresponding mode variable in the code you write. Note that the modes you should use have already been specified in existing header files.
- use the $-v$ flag to turn on diagnostic printing, or $-vv$ for more printing (or $-vvv$ etc. - the more "v"s, the higher the value of the corresponding variable *verbose*). We suggest using this verbose value to control your own debugging output.

e.g.:

speller_darray -d sample-dictionary -m 1 -vv sample-file

or:

speller_hash -d /usr/share/dict/words -s 1000003 -m 2 -v sample-file

Part 1: Searching and Sorting

In this first part we focus on using a *dynamic array* to store the dictionary. We have provided a partial implementation in *darray.h* and *darray.c*. This implements initialisation and insertion for you but you will need to complete the *find* function.

> **Aside:**
> We note that this point that we have opted to leave duplicates in the array. Why? Insertion into a dynamic array is generally $O(1)$ (although what happens if it is full?), whereas checking whether the value already exists is $O(n)$. How many duplicates would we need to remove during insertion to make it worth doing so when running find? Is there a more reasonable solution than making insertion $O(n)$ given the below discussion about binary search?

Your first job is to edit *darray.c* to complete the definitions of linear and binary search. As a reminder:

- Linear search iterates through each element in the array
- Binary search assumes a sorted array. It first picks the middle element and then either
  - Terminates if that element is the one being searched for,
  - Repeats on the left sub-list if that element is too big,
  - Repeats on the right sub-list if that element is too small.

The linear search should be quite straightforward but it is typical to make *out-by-one* style mistakes with binary search when first implementing it. To test your code try using a pre-sorted dictionary file.

But why implement binary search when our input array is not sorted? Because you're going to sort it. Your next job is to implement two or more sorting algorithms to sort the array.

> **Aside:**
> Is it worth it? Is it worth going to all this effort just so we can use binary search. Great question - this is actually something you're going to explore in Part 2. For now note that searching for 100 things in a list of $1,024$ items linearly will take in the order of $102,400$ time units, whereas using binary search will take in the order of $1,000$ time units. If we can spend less than $101,400$ time units sorting the array then we're winning!

As a first step you need to implement one iterative sorting algorithm and one recursive sorting algorithm. You should edit *sorting.c* to provide implementations of *insertion sort* and *quick sort* (see sorting document for hints). See below for some implementation hints. Once these are implemented you should test that your full setup works using modes 0 (for linear search), 1 (for binary search using insertion sort), and 2 (for binary search using quick sort). Do not change these modes as the online tests make use of them.

Extension Activities. If you still have time consider also implementing :

1. *Bucket sort*. This is interesting as it explores the trade-off between space and time complexity.
2. *Merge sort*. This is a popular sorting algorithm for coding interviews (along with quick sort). It also has interesting structure to analyse from a complexity viewpoint.
3. *Another sort*. For comparison, consider implementing a fifth sorting algorithm. You can look at the other sorting algorithms in the sorting document, do some online research, or try and invent your own.

Once you have implemented your sorting algorithms you should submit to COMPjudge to check that the tests pass. You will also be able to see some performance graphs comparing your solutions so far. Are they what you expect?

Implementation Hints. When implementing the above algorithms think about the following:

- Does stability matter for this application? (you should know what stability means)
- The prototypes in *sorting.h* assume that the sorting algorithms are *in-place* (make sure you know what that means) - is this the best approach for your chosen algorithm? What impact may it have on space or time complexity? You are allowed to edit *sorting.h* and the relevant place in *darray.c* to make your sorting algorithm call not-in-place.
- In quick sort you have a choice of where to select the pivot. Think carefully about the following different options and what impact they might have on the best/worst/average complexity of your algorithm:
  - The first element;
  - The last element;
  - The middle element;
  - A random element;
  - The median value from three randomly selected elements.

  You should also consider the (usually constant) *cost* of picking the pivot. You might want to try a few with different kinds of input to see what happens.
- In bucket sort you have a choice of (i) how many buckets to use, (ii) how to assign inputs to buckets, and (iii) which other algorithm to use to sort the buckets. For the number of buckets, you might want to run some experiments; this is where there is a trade-off between space and time complexity. For assigning inputs to buckets, you may assume the input type is always a `char*`. If you can place items into buckets such that the buckets remain in sorted order then your final step is much easier, is there a simple way of doing this? Alternatively, you will need to perform a $k-way$ merge on the final buckets. Something to think about - what is the difference between merge sort and bucket sort?

Part 2: Exploring the Trade-Offs

In this part you will write a report in a LaTeX document labelled *report.tex* (see your repository for a template report) that provides the following:

1. A detailed analysis of the complexity of *insertion sort* and *quick sort* using the techniques discussed in lectures. You should analyse best and worst cases and comment on what an average case might be.
2. An experimental analysis addressing the question

   Under what conditions is it better to perform linear search rather than binary search?

Hints on presenting complexity analysis

Firstly, what are the worst, best, or average cases for a particular sorting algorithm (this can vary). This will usually depend on the structure of the data. For example, in the case of linear search, if none of the queries are in the dictionary then we will always need to search the full dictionary for every query. In the case of sorting, consider what will happen if the input is already sorted, or sorted in reverse order, or the same entry multiple times. What is an *average* input (this can be subjective)? Note that you are only be asked to provide a formal analysis for the best and worst cases. Analysis of average complexity often involves probability functions and is generally more complicated.

Secondly, how do we present the algorithm for analysis? There are many examples in the suggested textbooks. For example, section 1.3 in Goodrich and Tamassia presents a case study analysing various algorithmic solutions to the maximum subarray problem.

For iterative algorithms one can apply simple operation counting to get an equation in terms of the input size n and then apply the big-Oh rules to this to get the appropriate complexity class. Alternatively, one can

reason directly over operations that impact the order of n in the above equation (e.g. loops). Note that it is important that you don't just count the number of (nested) loops but also check the number of iterations.

For recursive algorithms you must first define the recurrence equation for your algorithm and then solve this using one of the techniques in lectures:

- Substitution method,
- Iteration method,
- Master method

See the lecture notes and textbooks for further details and examples. Choosing the appropriate method is part of the problem.

Extension Activity. If you still have time consider also extending your complexity analysis to the other sorting algorithms you implemented (or if you did not implement other sorting algorithms, to other other sorting algorithms described in the sorting document).

Hints on designing the experiment

To understand under what conditions linear search will perform better than binary 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
- Size of queries
- Structure of dictionary e.g. sorted, random
- Structure of queries e.g. mostly in dictionary, or not
- Approach used e.g. linear/binary search, different sorting algorithms
- 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. 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.

To help we have provided a simple bash script called *generate.sh* that you can edit to automate the process of producing input of different kinds.

Hints on presenting experimental results

You need to present the answer to the posed question and back this answer up with data. You will want to make this data easy to understand. This may be through performance tables or graphs. Due to the variability of how you approach this question and how you collect your data, **we expect different students to find different answers, backed up by different data**. We are not looking for a single answer but a well-designed experiment leading to a justified answer backed up by data.

Other questions to consider

Other related sub-questions you may wish to consider whilst performing your analysis are:

- How do the theoretical complexities relate to the observed practical complexity;
- If two sorting algorithms have the same theoretical complexity why might they have different observed practical complexity;
- What impact does space complexity have on running times;
- Does the way the input is laid out in memory impact running times (think caches)

Part 3: Better Storage

In the above we achieved a faster *find* function by first sorting the input. In this part we explore two data structures that organise the data in a way that should make it faster to perform the find operation.

Part 3a: Binary Search Tree Lookup

In this part you need to complete the *bstree.c* implementation. You should:

1. Complete the insertion function by comparing the value to insert with the existing value. What should you do if they are equal? Hint: this is representing a set.
2. Complete the find function. Importantly, it should follow the same logic as insertion.
3. Extend the code to record statistics such as the height and the average number of comparisons per insertion or find. This will require some extra fields in the node struct.

Once you have done this, submit to COMPjudge and check the tests. Hint: if all the values in the tree are the same you probably forgot to make a copy of the char array holding the value (e.g. using *strdup*).

In this lab exercise we will stop there with trees. However, it is worth noting that this implementation is not optimal. What will happen if the input dictionary is already sorted? In the next lab we will be exploring self-balancing trees.

Part 3b: Hash Table Lookup

So far our solution to a fast find function has been either sorting the input or storing it in a sorted order. In this part you will take a different approach that relies on a *hash function* to distribute the values to store into a fixed size array. Unlike with binary search trees, we have only provided you with the prototypes you need to implement. You need to decide what data structure you need etc.

The hash-value(s) needed for inserting a string into your hash-table should be derived from the string. For example, you can consider a simple summation key based on ASCII values of the individual characters, or a more sophisticated polynomial hash code, in which different letter positions are associated with different weights. **(Warning: if your algorithm is too simple, you may find that your program is very slow when using a realistic dictionary.)** You should experiment with the various hash functions described in the lectures and implement at least two for comparison. One can be terrible. These can be accessed by the modes already given in *hashset.h*, please do not change these. Write your code so that you can use the $-m$ parameter, which sets the *mode* variable.

Initially, you need to use an open addressing hashing strategy so that collisions are dealt with by using a collision resolution function. As a first step you should use linear probing, the most straightforward strategy. To understand what is going on you should add code to count the number of collisions so you can calculate the average per access in print_stats.

An issue with open addressing is what to do when it is not possible to insert a value. To make things simple, to begin with you can simply fail in such cases. Your code should keep the *num_entries* field of the table up to date. Your *insert* function should check for a full table, and exit the program cleanly should this occur. Once this is working you should increase (double?) the hash table size when the table is getting full and then *rehash* into a larger table. Beware memory leaks!

Once you have done this, you can submit to COMPjudge to check that it works correctly.

Extension Activities. If you still have time consider also implementing alternative methods for dealing with collisions. But make sure you have completed part 3c without these extensions first as it is likely to be more work than it may initially appear. The alternative methods you may consider (and reasons for considering them) are:

1. *Quadratic Probing.* It is a good idea to get used to the general quadratic probing scheme. The observant among you will notice it's a common question in exam papers!

2. *Double Hashing.* You have two hash functions anyway, put them to work!

3. *Separate Chaining.* This is the most challenging as it requires making use of an additional data structure (which you've already written anyway) but it is also how many hash table implementations actually work so worth understanding.

During the marking of this part you will be asked to discuss the asymptotic algorithmic complexity of your function $find$, and the potential problems that linear and quadratic probing may cause with respect to clustering of the elements in a hash table. **You may be asked these questions even if you did not implement these collisions resolution methods.**

Part 3c: Making a Comparison

You should now extend your experimental evaluation from Part 2 to include the new modes of operation and add your conclusions to your report. The additional question being addressed is

> Under what conditions are different implementations of the dictionary data structure preferable?

Note that for hash set the initial size of the data structure will now play a larger role. You may find it useful to correlate your findings with statistics such as the number of collisions in the hash table.

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.