COMP26120
Academic Session: 2022-23

Lab Exercise 1: Spellchecking (Searching and Sorting)

Duration: 1 week

You should do all your work on the lab1 directory of the COMP26120_2022 repository - see Blackboard for further details. You will need to make use of the existing code in the branch as a starting point.

Important: This is the first of three related labs. Lab 3 is an assessed lab. It assumes that you have done the work in this lab. You have the choice to complete this lab in any programming language, but note that lab 3 must be completed in C, Java or Python and we advise using the same language for all three labs. Program stubs for this exercise exist C, Java and Python. ***Only one*** language solution will be marked for Lab 3. Since this lab is formative there is no submission requirement, but we recommend checking your work into gitlab regularly. This not only functions as a back up of your work but can also assist us in helping you with any queries you have. In online lab sessions staff and TAs will be happy to look at your work via a shared screen, but sometimes it really helps if we can check a piece of work out ourselves and run it locally to see what is happening. Lastly, getting into the habit of regularly checking in development code is useful.

Note on extension activities: The lab is organised so that you can cover the learning objectives to a reasonable standard 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 (C), class files (Java), and compiled bytecode files (Python). This is a general poor practice but for this course it can also cause issues with the online tests.

Other Languages: It has been mentioned that you are able to complete some labs in other languages. This is true for those labs that we do not assess. However, note that Lab 3 (assessed) builds on Labs 1 and 2.

Learning Objectives

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

- Explain how and when to use linear and binary search
- 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*, and *amortisation*, Lab 3 will investigate data structures such as *binary search trees* and *hash tables* in the same context.

The spell-checking program will take a *dictionary* of words and an *input file* and will check if all the words in the input file appear in the dictionary. Any words that do not appear in the dictionary will be printed out as spelling errors together with the line on which they appear. We have provided support files with functions for reading in files, printing out errors and so forth. The work in these initial labs is to implement the underlying data structure for storing the words in the dictionary and checking if a given word appears. We treat the stored dictionary as an abstract *Set* datatype. In this lab we ask you to implement this datatype using dynamic arrays. In lab 3 we will ask you to implement it using binary trees and hashsets.

Data structures

Initially we will use a *dynamic array* to implement an abstract *Set* datatype. We describe these briefly below but you may need to look online (e.g. the Wikipedia page) to complete your knowledge. Note that the course textbooks don't explicitly deal with a dynamic array as a data structure although *Introduction to Algorithms* uses it as an example of amortized complexity analysis in Section 17.4 (this section will be part of your directed reading when we look at amortized analysis).

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

- `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.

Dynamic Array. The majority of you will already have met this data structure as Java's `ArrayList` in COMP16121 or otherwise (it's also how Python's list is implemented). The idea is to use a dynamic array as the basis for an implementation of the *Set* abstract datatype. In the C implementation this means wrapping-up an array so that it dynamically resizes on demand, in Python and Java `ArrayList` and lists already have this resizing capability.

Sorting and Searching

We have provided a separate document (please read) giving an overview of the key concepts around sorting and searching, and the algorithms you should be aware of. As a brief summary:

- You should know whether a sorting algorithm is (i) stable, and/or (ii) in-place.
- Iterative sorting algorithms tend to have quadratic complexity as in the worst case they need to iterate over the list for each element.

- Recursive sorting algorithms do better by recursively sorting sublists. Splitting the list in two gives a logarithmic term in the complexity.
- Once you've sorted a list you can efficiently perform binary search by recursively splitting the list into the part 'smaller' and 'bigger' than the element you're looking for.

Lab 1: Searching and Sorting

In this lab we focus on using a *dynamic array* to store the dictionary. We have provided partial implementations in C, Java and Python that implement initialisation and insertion for you but you will need to complete the *find* function. If you wish to use a language other than C, Java or Python then you will need to create a spell-checking program that can utilise a dynamic array for its dictionary and can be used with similar command line flags to the program stubs we have provided.

> **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 the program stubs for dynamic arrays that we have provided 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 (we provide some, see data).

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 Lab 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 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,
- 2 for binary search using quick sort

Do not change these modes as the provided test program makes use of them. There are other modes that may be used for the extension activities (see next).

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.

Testing. Once you have implemented your searching and sorting algorithms you should test them. There are instructions in the individual language sections explaining how to test your algorithm on a single example and you should try that first. Once that is done you can test them on some test suites.

We have provided a test script, `test.py` in the top level directory of the COMP26120_2022 repository and some data in the `data` directory. `test.py` takes four arguments: the first is the language you are using; the second is the test suite (`simple` is provided but you can create more); the third is the data structure (`darray` in this lab) you are testing; and the fourth is the mode you want to test.

For example, to run the simple tests for mode 2 and your language is Java you should run

```
./python3 test.py java simple darray 2
```

You might want to edit the script to do different things. We have also provided some `large` tests (replace `simple` by `large` above) which will take *a lot* longer to run (see help box below). You will need to unzip the files by running `unzip henry.zip` in the `data/large` directory. These large tests should give you an insight into the relative performance of the different techniques but we'll revisit that further in Lab 2.

Failing Tests:
If the tests are failing there are two possible explanations. Either your implementation is wrong or the test script is being more picky than you thought.

For the first reason, make sure you understand what you expect to happen. Create a small dictionary and input file yourself and test using these or use one of the individual simple tests. There are nine of these and you will find them organised in the `data/simple` directory where each sub-directory contains an example dictionary, input file and the answer expected by the test. Most of these are small and easy to understand. Remember that the program should print out all of the words that are in the input file but not in the dictionary. For binary search make sure that you are searching using the same order that you sort with. Make sure that your code is connected up properly in `find` so that it returns true/false correctly.

For the second reason, make sure that you don't print anything extra at verbosity level 0. If you look at `test.py` you'll see that what it's doing is comparing the output of your program between "Spellchecking:" and "Usage statistics:" against some expected output. It runs the program at verbosity level 0 and expects that the only spelling errors are output in-between those two points.

Aside:
If you've been bold and implemented these in the three different languages you will already see a noticeable performance difference. On my laptop, the model solution for linear search takes twice as long in Java than in C, with the Python solution taking five times longer than the C solution. This is interesting for two reasons. Firstly, it is another motivation for ignoring constants when discussing computational complexity. Secondly, it makes us ask *why* and to get that answer we need to start looking at how these languages are implemented — C is compiled straight to machine code, Java compiles to bytecode, which is interpreted, and Python is interpreted (modulo JIT compilation and different implementations). But these aspects of languages are out of scope for this course.

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

- Does stability matter for this application? (you should know what stability means)
- The prototype implementations supplied with the lab 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 the stubs 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. 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?

Instructions for C Solutions

If you intend to provide a solution in C you should work in the `c` directory of the COMP26120_2022 repository. All program stubs and other support files for C programs can be found in this directory.

The completed programs will consist of several ".c" and ".h" files, combined together using *make* (the makefile covers Labs 1-3). 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.
- *sorting.h* and *sorting.c* - include prototypes for sorting functions. You will need to edit this.

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.

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 in the `data` directory of the COMP26120_2022 repository, and you will probably need to create some more to help debug your code. These files will need to be copied to the directory where you are working or you will need to set your *PATH* so that your program can be executed from anywhere.

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*. This will create an executable called *speller_darray*.

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

- use the $-d$ flag to specify a dictionary.
- 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`

So to run your program on the third of the simple test cases, in mode 1, with a medium level of verbosity, you would call:

```
speller_darray -d ../data/simple/3/dict -m 1 -vv ../data/simple/3/infile
```

Instructions for Java Solutions

If you intend to provide a solution in Java you should work in the `java` directory of the COMP26120_2022 repository. The completed programs will form a Java package called `comp26120`. You can find this as a sub-directory of the `java` directory. All program stubs and other support files for Java programs can be found in this directory.

You are given a number of complete and incomplete components to start with:

- *GetOpt.java*, *LongOpt.java* and *MessagesBundle.properties* - control command line options for your final program. You should not edit these.
- *speller_config.java* - defines configuration options for the program.
- *speller.java* - 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.java* - defines the generic interface for the data-structure
- *set_factory.java* - defines a factory class to return the appropriate data structure to the program. This will be used in later labs.
- *darray.java* - defines a dynamic array. You will need to edit this.
- *speller_darray.java* - sub-classes *speller* for use with *darray*.

Note: The code in *speller.java* 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.

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 in the `data` directory of the COMP26120_2022 repository, and you will probably need to create some more to help debug your code. These files will need to be copied to the `java` directory or you will need to set your $CLASSPATH$ so that your program can be executed from anywhere.

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 your code using *javac *.java*. This will create an executable called *speller_darray.class*. To run your program you should call `java comp26120.speller_darray sample-file` (where $sample - file$ is a sample input file for spell checking). Note that you will either need to set your $CLASSPATH$ to the `java` directory of the COMP26120_2022 repository or call `java comp26120.speller_darray sample-file` in that directory.

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

- use the $-d$ flag to specify a dictionary.
- 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 at the end of *darray.java*.
- 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.: `java comp26120.speller_darray -d sample-dictionary -m 1 -vv sample-file`

So to run your program on the third of the simple test cases, in mode 1, with a medium level of verbosity, you would call:

`java comp26120.speller_darray -d ../data/simple/3/dict -m 1 -vv ../data/simple/3/infile`

Instructions for Python Solutions

If you intend to provide a solution in Python you should work in the `python` directory of the COMP26120_2022 repository. All program stubs and other support files for Python programs can be found in this directory. You should use Python 3.

You are given a number of complete and incomplete components to start with:

- *config.py* - defines configuration options for the program.
- *speller.py* - 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_factory.py* - defines a factory class to return the appropriate data structure to the program. This will be used in later labs.
- *darray.py* - defines a dynamic array. You will need to edit this.
- *speller_darray.py* - front end for use with *darray* which then calls the functionality in *speller.py*.

> **Note:** The code in *speller.py* 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.

Running your code

Important: To run the provided program stubs in python2.7 (if you really want to do this, though we advise strongly using Python 3) you will need to install the enum34 package. You can do this from the UNIX command line

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 in the `data` directory of the COMP26120_2022 repository, and you will probably need to create some more to help debug your code. These files will need to be copied to the `python` directory or you will need to set your $PYTHONPATH$ so that your program can be executed from anywhere.

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

To run your program you should call `python3 speller_darray.py sample-file` (where $sample-file$ is a sample input file for spell checking). Note that you will either need to set your $PYTHONPATH$ to the `python` directory of the COMP26120_2022 repository or call `python3 speller_darray.py sample-file` in that directory.

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

- use the $-d$ flag to specify a dictionary.
- 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 at the end of *darray.py*.
- 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.: `python3 speller_darray.py -d sample-dictionary -m 1 -vv sample-file`

So to run your program on the third of the simple test cases, in mode 1, with a medium level of verbosity, you would call:

`python3 speller_darray.py -d ../data/simple/3/dict -m 1 -vv ../data/simple/3/infile`