COSC122 (2021) Lab 1

Introduction to Data Structures and Algorithms

Goals

This lab will give you some experience with different searching algorithms and a high-level overview of some of the data structures you will encounter in this course. This lab is designed to give you a taste of some of the material you will encounter — future labs will cover these topics (and others) in much greater detail.

Labs and Lab Quizzes

Welcome to the first lab for the course! These labs build upon the material presented in lectures and described in your textbook by giving you some experience with writing, using, and adapting various data structures and algorithms. The primary emphasis of these labs is on analysing and understanding, rather than writing code.

Material in these labs will often build upon examples shown in your textbook. As such, it is *strongly recommended* that you read the associated chapters/sections of the textbook before attempting these labs. Links to the online version of the textbook will usually be provided in footnotes. You might also want to bookmark the main textbook page (here).

For Lab 1, you are not expected to know how searching algorithms work. But if you are interested sections 6.2 (link) and 3.5 (link) in the online textbook cover the relevant details.

As with the introduction to programming course, each lab has a *Quiz Server* quiz associated with it. These quizzes must be completed and submitted by the due date given in each quiz (usually the Monday after the topic is covered in labs). You only get one attempt at each lab quiz, unlike COSC121/131, but a practice quiz will open after the due date for you to do more practice.

Searching Algorithms

In the archive for this lab is the `arrays.py` module, which contains a few classes that implement a data structure for an *array* that stores positive integers. The three classes in the module are:

1. `LinearArray`—uses a Python `list` for storing items.
2. `SortedArray`—uses a *sorted* Python `list` for storing items.
3. `BitVectorArray`—using a modified bit vector for storing items (see the *"Comparing Search Methods"* section).

However, some of the methods in these classes aren't finished yet—you will have to complete them before you can try them out.

Linear Searching

Please note, the linear search method is also known as sequential search and the textbook refers to it as sequential search. Given the simplicity of this method, it should be clear that linear and sequential can easily be interchanged. *Linear* refers more to the time complexity for the method and *sequential* refers more to the way we search through a list of items. If the list is laid out in a straight line then both ideas are very similar, ie, a search is just a walk along the line of items until we find the one we are looking for.

The `LinearArray` class contains the code for storing a simple Python `list` of `int`s, with a few methods to insert new values, delete existing values, and check to see if a particular value exists in the array. Although a Python `list` already comes with this functionality, we want to experiment with potentially more efficient ways of doing things.

You will need to complete the `find_index` method to search the `self.data` list for the provided value, returning the index of the item if it is found within the list. Use the comments in the method as a guide, and remember to count the number of comparisons with data in the array using the `self.comparisons` variable.

To test your code, you can create a new instance of the `LinearArray` class and manually test inserting, deleting, and checking for the existence of elements:

```
from arrays import LinearArray
x = LinearArray()
# Insert some items
x.insert(3)
x.insert(2)
x.insert(1)
# Look for an item; should return True
x.contains(2)
# Remove an item
x.remove(2)
# Look for an item; should return False
x.contains(2)
```

The archive also contains a few data files that you can use to test the class and the number of comparisons required for each operation. There first four files: `file0.txt`, `file1.txt`, `file2.txt`, and `file3.txt`; with 10, 100, 1000, and 10000 inserts into the array, respectively. The test files are basically lists of commands to run on the various arrays. These *trace* files allow us to test the arrays with exactly the same sequence of commands so we can see the difference in performance for the same workload. The data files from number 4 onward contain various sized batches of commands that can be used for testing and comparison—and lab quiz questions. The start of test file 0 looks like:

```
i 89
i 97
c 97
i 11
c 11
i 0
c 11
i 88
...
```

Where:

- **i** stands for insert
- **c** stands for contains? (or more specifically check whether a number is in an array)
- **d** stands for delete/remove number from array

Open the `array_tests.py` file and have a look at the `process_file` function. If you run the `array_tests.py` module then it will run the `main_tests` function that will run the following example:

```
    # for example
    filename = 'file0.txt'
    print('Processing', filename, 'with a linear array')
    test_array = LinearArray()  # initialise a LinearArray
    process_file(filename, test_array)
```

This will produce the following output for `file0.txt`, where the first number is the index of the trace line being run:

```
Processing file0.txt with a linear array
   0:     insert   89       0 comparisons
   1:     insert   97       0 comparisons
   2:   contains   97       2 comparisons (found)
```

```
  3:     insert   11       0 comparisons
  4:   contains   11       3 comparisons (found)
  5:     insert    0       0 comparisons
  6:   contains   11       3 comparisons (found)
  7:     insert   88       0 comparisons
  8:   contains    0       4 comparisons (found)
  9:     insert   55       0 comparisons
 10:     insert   76       0 comparisons
 11:     insert   27       0 comparisons
 12:   contains   97       2 comparisons (found)
 13:   contains   88       5 comparisons (found)
 14:   contains   89       1 comparisons (found)
 15:     insert    6       0 comparisons
 16:   contains   11       3 comparisons (found)
 17:   contains   27       8 comparisons (found)
 18:     insert   64       0 comparisons
 19:   contains    1      10 comparisons (not found)
```

Once `LinearArray` is working and giving the correct output, you can try the other input files as well. Note that inserting an element in a `LinearArray` takes no comparisons at all, while checking whether or not an array contains an element can take a long time for those inserted relatively recently (as they are near the end of the list). In the upcoming sections we will see that *Sorted* and *BitVector* arrays work differently, eg, the sorted array will use some comparisons when adding and the BitVector will only need one comparison when searching ...

> ***Now you can answer the* Linear search *questions in Quiz1.***

Binary Searching

The `SortedArray` class performs the same operations as `LinearArray`, but uses a binary search algorithm to insert and check if an array contains elements.

This time, all of the methods have been completed for you, *however*: you need to add the code to count the number of data comparisons (not index comparisons) in the appropriate places in both `insert` and `find_index`.

Once you have completed this, you can test your code by processing the data files. The output when processing `file0.txt` with a sorted array should look like:

```
Processing file0.txt with a sorted array
  0:     insert   89       0 comparisons
  1:     insert   97       1 comparisons
  2:   contains   97       1 comparisons (found)
  3:     insert   11       2 comparisons
  4:   contains   11       3 comparisons (found)
  5:     insert    0       2 comparisons
  6:   contains   11       3 comparisons (found)
  7:     insert   88       2 comparisons
  8:   contains    0       5 comparisons (found)
  9:     insert   55       2 comparisons
 10:     insert   76       3 comparisons
 11:     insert   27       3 comparisons
 12:   contains   97       5 comparisons (found)
 13:   contains   88       5 comparisons (found)
 14:   contains   89       3 comparisons (found)
 15:     insert    6       4 comparisons
 16:   contains   11       3 comparisons (found)
 17:   contains   27       5 comparisons (found)
 18:     insert   64       4 comparisons
 19:   contains    1       8 comparisons (not found)
```

NOTE: If Wing truncates (leaves out) some of the output with longer files then you should run your program in debug mode (click the red bug icon instead of the green ▷ play button.)

> ***Now you can answer the* Binary Search *questions in Lab Quiz 1.***

Once you have had a play and understand the search methods and data files you should consider the following questions:

- Which search is better for failed `contains` operations? Why?

- Why are there no comparisons needed for the `insert` operation in the linear search, but a few are needed for the binary search?
- For linear search, how is the number of comparisons needed to check that an item exists related to that item? Why does looking for 5635 in `file3` take fewer comparisons in linear search than in binary search?

Comparing Search Methods

Performing a comparison between two data values is one of the slowest operations and the most frequent operations in a searching algorithm, which is why algorithms seek to minimise the number of comparisons they make. In addition to printing out the number of comparisons, the `time.perf_counter()` function provided by the time module allows you to examine how long it actually takes code to execute.

The `array_tests.py` module imports `time` and provides a timing example in the `time_sorted_trial` function (as shown below):

```
def time_sorted_trial(filename):
    """ Times how long it takes to processes the
    given file with a SortedArray
    """
    test_array = SortedArray()
    print('\nRunning trial on sorted array with', filename)
    start_time = time.perf_counter()
    process_file(filename, test_array)
    end_time = time.perf_counter()
    time_taken = end_time - start_time
    print(f"Took {time_taken:.3f} seconds.")
    return time_taken
```

Try using `time` on a `SortedArray` and a `LinearArray` - note the difference in execution time. You will want to write a `time_linear_trial` function that is very similar to the provided `time_sorted_trial` function. Use files `file1`, `file2` and `file3` to look at how the relative performance of each implementation changes as the input size increases.

> ***Now you can answer the* Linear vs Binary *question(s) in Lab Quiz 1.***

The `arrays` module also contains a `BitVectorArray` class. This is an implementation that uses a variation of a *bit vector* or *bitmap* data structure to store its data—instead of storing each inserted value in the list, it simply records how many times a particular value is inserted. This data structure isn't covered in the textbook, so don't worry if you don't understand it—it's used here because it happens to work well for this particular kind of data (and is part of a good answer to Google's question to Barack Obama[1]).

You can use the `BitVectorArray` as you have for the `LinearArray` and `SortedArray` with one difference: when you're creating the `BitVectorArray`, you need to tell it what the largest possible value you will store is:

```
b1 = BitVectorArray(100)
process_file('file0.txt', b1)
b3 = BitVectorArray(10000)
process_file('file3.txt', b3)
```

Use `time` to examine how fast the `BitVectorArray` is, and compare its results to that of the other two implementations (especially when you use files `file2` and `file3`).

Although a bit vector is *extremely* fast and efficient (compared to linear and binary searching), this doesn't mean it's the panacea of searching algorithms; it has some major disadvantages. [2]

[1] http://www.youtube.com/watch?v=k4RRi_ntQc8&feature=youtu.be

[2] The fact that you need to tell it how big the largest item you want to store is might give you a hint about what some of those disadvantages are.

Testing Python's List and Set Implementations

Sets have been covered in the introduction to programming course. Please read section 3.5 to 3.8 of the online textbook. In this section, we will compare the behaviour of the Python 'in' operation (which checks whether an item is contained in the given list or set) using a list and a set implementation.

Note: In this section you will run tests for various list sizes (n's) *but* you will run each test a number of times and report the average time taken for the given list size (n).

- Run the appropriate code in `internal_trials.py` to test searching in a Python list. Plot the results in a graph. *See notes below for graphing ideas.*
- Change the number of trials to 5, 10, 100, etc. and use a graph to compare the behaviour of *'in'* in a `list`. Try 1000 trial runs if you don't get too bored waiting... See graphing tips below.
- Complete the code in `run_set_trials` so that it tests the *'in'* operation on a set. The code will be very similar to the `run_list_trials` function except you will need to execute a statement like `(found = value_to_find in test_set)` multiple times as specified by variable `num_trials`. Make sure that the program displays the average time taken by a single search, as the size of the set changes.
- Change the number of trials to 5, 10, 100, etc. and use a graph to compare the behaviour of *'in'* in a `set`. Try 1000 trial runs, hopefully it is a bearable wait... See graphing tips below.
- Running more trials shouldn't greatly affect the time per *'in'* operation, so why is it important to use multiple trials? *Think about the nature of modern multi-tasking, multi-processor computers.*
- Compare the two implementations by plotting, in a single graph, the values for both implementations for 100 trials. You should get a graph similar to the graph at the bottom of section 2.7 of the online text book.

Graphing Tips

Installing matplotlib

Lab computers have Python and `matplotlib` installed so don't try to install them there! If you are running Python on your own computer then you will need to make sure you install `matplotlib`. The easiest way to install `maplotlib` is to open the `install_numpy_and_matplotlib.py` in Wing101 and run it. This will ensure `maplotlib` is set-up for your user in the Wing environment that you will be running it. Or, check out the *matplotlib* install page directly, here. Either way, if you are using Windows, the important thing is to make sure you checked the ***Add Python 3.x to PATH*** option when installing Python.

Generating data and graphs

Tab delimiting your output (as shown in the `run_list_trials` function) will allow you to cut and paste output in to Excel (or Libre Office Calc). *But*, investing a little time in setting up some graphing code in Python will make experimenting a lot easier. Therefore, we recommend you try using `matplotlib` to get plots of the output. The following gives you an example of how to use `matplotlib` and is similar to the function `graph_one_series_example` in `internal_trials.py`.

```
import matplotlib.pyplot as plt
n_trials = 10
x1, y1 = run_list_trials(n_trials)

# We use the following instead of axes = plt.axes() as it opens new figures (figs)
# in new windows for example if you call graph_one_series_example then call
# graph_one_series_example you will get two graph windows :)
fig, axes = plt.subplots()

axes.plot(x1, y1, color='blue', marker='o', label='list')
axes.set_title(f'List Locate Testing, {n_trials} Trial runs')
axes.set_xlabel('n')
```

```
axes.set_ylabel('Average Time per locate')
axes.grid(True)
axes.xaxis.set_major_formatter('{x:.0f}')  # otherwise defaults to exp notation
legend = axes.legend(loc='upper left')
plt.show()
```

Check out the `graph_one_series_example` and `graph_two_series_example` functions for some graphing that we already had in the oven. The one series function currently plots the results of list tests but you can easily change it so that it runs set tests instead.

> ***Now you should be able to answer the Comparing Structures questions in Lab Quiz 1.***

Sorting Algorithms

Experiment with the sorting algorithm animations at http://www.sorting-algorithms.com/ and make notes about which method is the fastest and how they deteriorate as the problem size changes. You do not need to understand how the algorithms work, just examine the performance characteristics of each under various conditions.

> ***Now you can answer the* Comparing Sorts *questions in Lab Quiz 1.***

Complexity

To end the main part of this lab we have a quick review of asymptotic complexity.

Simple method to determine Big Oh for an algorithm

1. Find an operation that the algorithm does at least as much as any other.
2. Count how often it is done (in terms of the input size, n).
3. Take highest order term only
   (increasing order is 1, $\log n$, n, $n\log n$, n^2, n^3, $n^4, \ldots, 2^n$, 3^n, $n!$) and drop any constant that it is multiplied by.
4. Remove any constants.

For example, if an algorithm uses $6+5n+6n^2+3n^3$ operations we would say it is $O(n^3)$ because n^3 is the part that will grow the quickest. We don't worry about constant multiples so we *wouldn't* write it as $O(3n^3)$.

Note you can think of constants as $constant \times 1$, eg, an algorithm that takes 6 operations can be thought of as taking 6×1 operations and therefore we can treat it as O(1). Basically, any constant term is treated as $O(1)$.

> ***Now you can answer the* Complexity - Big Oh *questions in Lab Quiz 1.***

Extras

- Change the `contains` method in `SortedArray` to use fewer comparisons (down to roughly half in the best case). *Hints*: When searching for x, in initial comparisons, is x more likely to be less than the indexed value or equal to the indexed value? If x is less than (or greater than) the indexed item do you need to also check if it is equal to the indexed item?
- For `file3`, what percentage of entries are faster to check for existence in binary search compared with linear search?