CS917: Foundations of Computing
Lab 1: Search Algorithms and Complexity

1 Introduction

In this session you'll be exploring the use of search algorithms, and looking at how the selection of data structures and algorithms can impact the performance.

As always, any necessary resources are available to you on the website.

Have fun!

2 Benchmarking

As a means of highlighting how performance differs, let's introduce benchmarking (aka timing code). The code below is a small sample highlighting how to time a code block (as found in the example from lecture 1).

```
import timeit
...
timeStart = timeit.default_timer()
CodeBlock
timeEnd = timeit.default_timer()
time = timeEnd - timeStart
```

When measuring times, if something is too short to measure the time may come back as 0.0 seconds. In these cases, it may be beneficial to measure something repeated a number of times and instead generate an average time per operation (beware of misleading times if the time per operation is increasing with every repetition though!)

3 Sorting

(A) A theatre has had a sudden rush of demand for tickets for their latest performance. Rather than force people to queue, they have decided to implement a form of random lottery. Every person who has registered has had a random priority assigned to them. However currently the people in the data file are sorted alphabetically, not by their priority, meaning it is difficult for the theatre to determine who they should contact first. For your task you need to:

- Read in a data file, retrieving a name and a value per line.

- Store these values in an appropriate data structure.
- Implement and sort the data using a bubble sort (don't use the in-built sort for today's purposes)
- Print them back out to a file, sorted by their priority.

(B) Perform the same function as (A), but this time use a merge-sort. Use the timers as described in the benchmarking section to time how long it takes to sort the data using either method. Which is faster? Why?

Depending on whether you choose to do this iteratively or recursive the implementation can vary. Here are some hints for either approach:
For an iterative merge-sort, think on the properties of how the data is sorted:

- We know that we need to begin with groups of size 1, so we can take a 'bottom-up' approach, and just start with groups of size 1.
- We have a list of data, thus each index is equivalent to a block of size 1.
- The result of a merge-sort is two merged, sorted sub-lists. This suggests we need a method or in-place means to merge these groups.
- On one pass, we can pair up groups of 1 and apply our sorting algorithm. At the end of this pass, we will have a list of data where every pair of indexes is sorted within the pair.
- Now that we know every pair is sorted, we can apply another pass where we can merge two pairs together, such that every block of four indexes is sorted within those four indexes etc.
- We can apply this principle to keep going until the group size traverses the entire list - i.e. the entire list is sorted.

For a recursive merge-sort it helps to break it down into components:

- The result of a merge-sort is two merged, sorted sub-lists. This suggests we need a method that can merge these lists
- The two inputs to our merge function must be the result of a merge-sort of a smaller list (to guarantee it is sorted)
- We need a way to sub-divide our list - python lists have a concept known as slicing
- We need to know when to stop sub-dividing our lists - a termination condition. When should this occur?

- Hint: It may help to draw a diagram if you are have difficulty visualising the approach. At each method call, draw a concept of what the data should look like and see how you can write your code to achieve that.

Bonus Question: Would it have been cheaper to store them in a stored fashion as the priorities were originally assigned to them?

4 Linked-Lists

If you have some bonus-time left at the end (or in your free time), feel free to try out this exercise!

Python has some inbuilt data structures, in the form of tuples, lists and dicts. However, it doesn't have a linked-list. This doesn't stop us from creating one however! Objects encapsulate a collection of related data and functions to perform on that data. We can use these to create linked-lists. On the website, you should find a file that contains the skeleton code for a linked-list to give you some hints as to the necessary functionality needed. Fill out the methods that only contain 'pass' such that the data is stored in a way as described for linked-lists in the lectures.

In the skeleton file for the linked-list, there are also two completed functions that generate python lists (generateRandomList) or linked-lists (generateRandomLinkedList) of a specified length. Why not try generating and timing some lists of different sizes for both python lists and linked-lists? How does the performance differ for insertions/ deletions, gets etc? Is it as expected?

5 Final Notes

Today's exercises were intended to highlight the impact a few simple design choices can have on the performance of your programs! However, python already has a built-in capacity for sorting in the form of the sort or sorted methods. More often than not, you will likely use these - not only is code reuse a core tenant of programming in general, but often such in-built algorithms are highly tuned. I would recommend that more generally if you plan to sort in python you should look over the python documentation, in particular looking at how to specify the keys used to sort (as we saw today, some lists have matching data that must also be reordered).