This lab is part of the formal continuous assessment for CS2516. It focuses on experimental comparison of sorting algorithms. It will count for up to 10% of the total marks available for this module.

Note that you will not be able attempt Q1 or Q2 until after the lecture on 4th February, but you are able to do Q3 and Q4 now. After you have the code working for Q4, you can then go back to implementing Quicksort (Q2) and then add it to the calling methods in Q4.

1. **Revision**
    1. What is the difference between mergesort and quicksort?
    2. What is the worst case complexity of mergesort?
    3. What is the space complexity of standard mergesort on an array, and bottom-up mergesort on an array?
2. **Implementing Quicksort**
   Implement the in-place version of Quicksort from Lecture 6 to be applied to python lists (i.e. array-based lists and not linked lists). Do not use list slicing, as it will slow the algorithm down significantly on larger input lists. Your first version should not use any randomisation - instead, always pick the first element of the (sub-)list as the pivot, so that you can test and debug your code reliably. Once you are sure it is implemented properly, then introduce randomisation - the simplest way is to shuffle the list before you start.
3. **Implementing standard sorting algorithms for array-based lists**
   Make sure you have working versions of top-down Mergesort (from Lecture 5) and in-place Heapsort (from Lab 01). All of these should work on Python lists (i.e. array-based lists, and not linked lists). Test them to make sure they are working properly.
4. **Experimental comparison of sorting algorithms**
   The sorting algorithms discussed in lectures have all been analysed in terms of worst-case complexity. For many of them, we can identify the best-case input (usually an already-sorted list or a reverse sorted list), and the worst case input. But the typical runtime for an algorithm may not match these extreme cases. The aim of this practical is to implement some functions which will allow us to estimate the relative performance of the different sorting algorithms, highlighting changes in runtime as the size of the input list grows. We are not comparing to the basic sorting algorithms of bubblesort, selection sort or insertion sort - the lectures, plus lab01, showed that they are too inefficient on larger lists.
   For a given size *n* of input list, with *k* non-unique items (i.e. *k* of the items in the list are duplicates of other items), we should be able to generate 10 different randomised shuffles of the list. If we run each sorting algorithm on exactly the same lists, and take the average running time for each algorithm, we can get a sense of how algorithms perform relative to each other on typical cases.
   Implement a function evaluateall(n, k) which will generate input lists as above, and then evaluate each of the sorting algorithms
    1. heapsort

2. top-down mergesort
3. in-place quicksort
4. the built-in Python list sort

In your evaluateall function, you should first create a list of the first *n-k* integers, then fill up the list to size *n* by randomly selecting a position in [0,*n-k-1*] and copying that element to the end of the list. Then, 10 separate times, you should create a copy, and then call the shuffle method in the random library. This will give you the 10 lists for the input parameters.
To evaluate the four algorithms on these input lists, you need to create a new copy of each list for each algorithm, then for each algorithm, for each list, start a timer, call the algorithm, end the timer, run a test method that checks that the final list is indeed sorted correctly, and record the measured runtime. Once you have recorded the runtime for each of the 10 cases, compute the average, and print to standard output the average runtime, the name of the algorithm, and the input parameter values of *n* and *k*. E.g.
0.00532 heapsort 1000 0
0.00518 mergesort 1000 0
0.00481 quicksort 1000 0
0.00256 python 1000 0

5. (Do not pay attention to the numbers printed above - it is just an example to show you the format of the output). If you don't create a new copy of the input list for each algorithm, the first algorithm will do the work of sorting the list, and the remaining algorithms will be given that sorted list as input, and so their runtimes will appear to be much lower.
   Implement another function evaluatepartial(n,k), which generates a list of the first *n-k* integers augmented with *k* duplicates as before, but this time does not shuffle the list. Instead, sort the list using python's built-in sort, then *n/20* times randomly select a pair of indices between 0 and *n-1* and swap the elements. That will give a partially sorted list in which up to 10% of the elements have changed position. Run each of the algorithms on this list, and print the results as before, but also appending the character *p* to indicate the result is for a partially sorted list.
   Write a function evaluate() which calls the evaluateall(...) and evaluatepartial(...) functions for the inputs *(n,k)* =
   1. (100, 0)
   2. (1000, 0)
   3. (10000, 0)
   4. (100000, 0)
   5. (100, 20)
   6. (1000, 200)
   7. (10000, 2000)
   8. (100000, 20000)
   9. (100, 70)

10. (1000, 700)
11. (10000, 7000)
12. (100000, 70000)

6. Write a brief report as an opening comment in your .py file comparing the runtimes of the different algorithms. Don't give a blow-by-blow account of every line of output - just explain what you see in terms of how the algorithm perform relative to each other and relative to the size of the inputs.
   Note: statisticians tell us that we need to run more than 10 randomised copies in order to get a robust estimate of the runtime, but for this assignment that would take too long for the larger input sizes, so stick with 10.

Submission instructions

The submission deadline will be Friday 14th February, at 2pm. This gives 3 supervised lab sessions.

- You must submit a python file, which must be named sortingXYZ.py, where XYZ must be replaced with your student ID number. This file must contain the implementations of the different sorting algorithms and the test function, and the report (as a comment) on the experiments.
- To submit, you must send an email to k.brown@cs.ucc.ie, with the title 'CS2516 Assignment 1', remembering to attach your sortingXYZ.py file.
- Make sure that you compile and test your code before submitting, and that you submit exactly the version that you tested. After you have written the comment at the start comparing the results, you should test your code again to make sure it still works. You should also remove any debugging print statements from your code before you submit it. This is particularly important when we are testing algorithms for runtime performance - print() statements slow the code down considerably.
- Remember that this is part of the formal assessment for CS2516, and so the University's regulations on plagiarism apply. The code you submit must be your own code. Do not submit code written by someone else (or, at least, if you do submit someone else's code, you must declare it, and you will not get marks for that part of the assignment). Do not adapt code written by students for similar assignments in previous years. You are free to submit any code that I have posted this year in lectures or labs for CS2516 or CS2515, and you do not need to declare that.