Week 4 Assignment - Algorithms

Overview

In this week's assignment, we will be seeing how different search and sort algorithms in the reading compare to each other in their time efficiency. We'll be looking at the sequential and binary search algorithms, as well as the insertion and shell sorting algorithms. Make sure to create a github repository for this assignment named **IS211_Assignment4**. All development should be done in this repository.

Useful Reminders

1. Read the assignment over a few times. At least twice. It always helps to have a clear picture of the overall assignment when understanding how to build a solution.
2. Think about the problem for a while, and even try writing or drawing a solution using pencil and paper or a whiteboard.
3. Before submitting the assignment, review the "Functional Requirements" section and make sure you hit all the points. This will not guarantee a perfect score, however.

Part I - Search Algorithm Comparison

In this part, we will do a *worst-case scenario* comparison between the search algorithms in the reading. For searching, a worst-case scenario is searching for an element that does not exist. Using the sequential and binary search (both iterative and recursive) functions given in the text, generate a random list of positive integers and do a benchmark analysis for each one. Make sure to:

1. Create a new file in your repository called *search_compare.py*.
2. Create four functions, ***sequential_search, ordered_sequential_search***, ***binary_search_iterative*** and ***binary_search_recursive***, and use the code from the section on Sequential and Binary Search to implement them.
3. Modify each function to calculate how long the function takes and to return this along with the result of the search function.
    a. Do you remember how to return more than one value from a function?
4. The main function of the program should then print how long each function takes, on average. This should be done by generating 100 random input lists (of positive integers) of size 500, 1000, and 10000, and taking the average run time of the 100 lists.
    a. For clarity, you should generate 100 separate lists of size 500, and run each algorithm on these 100 lists.
    b. Since we are doing a worst-case analysis, we should search for an element we know will not be in the lists. Our lists should be made up of only positive integers, so for this part of the assignment, we'll search for the -1 element. This should never exist, and hence will be a worst-case scenario.
    c. Make sure to print out the result in this format: "Sequential Search took %10.7f seconds to run, on average"
    d. Remember, the binary search functions as well as the ordered sequential search function requires the list to be sorted. Make sure to sort the list (using Python's built in list sort() function; see Part III) before calling the function, so the time it takes to sort the list does not get included in your calculation for how long it took to search.

Food For Thought: How exactly do you plan on doing this? Should we generate all these lists upfront and then run algorithms on them? Or should we generate them one at a time when needed? How will you keep track of all the run time values and calculate the average?

Part II - Sorting Algorithm Comparison

Similar to above, in this part we will compare a few sorting functions we learned about in the readings. However, in this case, we will be looking only at the average-case scenario. Perform a benchmark analysis, just like Part I, with lists of random numbers of the same size, comparing insertion sort, shell sort and the built-in Python's sort algorithm. The reading material has example Python code that implements these algorithms. Make sure to:

1. Create a new file in your repository called *sort_compare.py*.
2. Create three functions, ***insertion_sort***, ***shell_sort*** and ***python_sort***. Use the code from the reading for the first two functions. The ***python_sort*** function simply a 'wrapper' function that calls ***sort()*** on the input list.
3. Modify these functions to calculate how long the function takes, similar to Part I.
4. The main function of the program should then call these functions for a list of random numbers of size, 500, 1000, and 10000.
    a. Make sure to print it out in the same format as Part I.

Functional Requirements

1. *sort_compare.py* should:
    a. define four functions, ***sequential_search, ordered_sequential_search***, ***binary_search_iterative*** and ***binary_search_recursive.***
    b. have these functions return appropriate values
    c. define a ***main()*** function that will print out the results as describe in Part I
2. *search_compare.py* should:
    a. define three functions, ***insertion_sort***, ***shell_sort*** and ***python_sort***.
    b. have these functions return appropriate values
    c. define a ***main()*** function that will print out the results as describe in Part II