Lab 6: Sort Algorithms

Sorting Algorithm Comparisons on Sorted and not Sorted List

This lab will compare four different sort algorithms. You are allow to use the given code in the class or code in the book or write your own code. (Before doing any comparison make sure, your code works!)

(Note: You are testing these sort algorithms on integer numbers)

Comparison Sorting Visualizations link can help you to have some perspective on sort comparison: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Create a python file as **comparison_sort.py**, write your functions for sorting sorted and unsorted list in this file. Your goal for this lab is to implement simple versions of Insertion Sort - insertion_sort(alist), Selection Sort - selection_sort(alist), Merge Sort - merge_sort(alist), and Quick Sort - quick_sort(alist), that will sort an array of integers and **count the number of comparisons**. Each function takes a list of integers as input, sorts the list, counts the number of comparisons at the same time, **and returns the number of comparisons**. After the function completes, the "alist" should be sorted.

Do not do any improvement in the sort algorithms, which can cause reducing the number of comparison. Use the original algorithms. Make sure you are using the same list for all sort algorithms.

The runtime complexity is O(n2) for selection and insertion sort; and O(n*log(n)) for merge and quick sort, Why? You will submit your answers and results of your code in a pdf file. Write out the summation that represents the number of comparisons. Note that you will need to run test cases of different sizes to show that Insertion and Selection sorts are O(n2) and Merge and Quick sorts are O(n*log(n)). To do this, plot the number of comparisons (y-axis) vs. the size of the list (x-axis). Since the plot is shaped like a parabola for selection and insertion sort this indicates that it is quadratic. In the pdf file, for each Insertion, Selection, Merge, and Quick sorts submit a table showing list size and number of comparisons from your test cases and a plot of that data.

Check the runtime complexity of Python sort( ) function for given list. Python uses **Timsort algorithm** (it is a hybrid sort that derived from merge and insertion sort algorithms with runtime complexity of O(n). (just use alist.sort() in your code and set the start and end time).

For generating the list you can create a list of random numbers.

```
import random
alist = random.sample(range(10000),10)
Which produce a list of 10 random number between 1 and 10000
[5445, 8692, 6915, 8637, 4848, 9408, 1744, 171, 4315, 2949]
```

For time you can use time class of Python

```
import time
start_time = time.time()
```

```
#Now call sort function
end_time = time.time()
sort_time = end_time - start_time
```

Submission:

1) comparison_sort.py
2) Comparison_analysis.pdf

The pdf file will include:

1) Two tables for each sort algorithm.
2) Answer to the questions in the last page
3) Plot the number of comparisons (y-axis) vs. the size of the list (x-axis). Since the plot is shaped like a parabola for selection and insertion sort this indicates that it is quadratic.

Selection Sort (Sorted List)		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)		
2,000 (observed)		
4,000 (observed)		
8,000 (observed)		
16,000 (observed)		
32,000 (observed)		
100,000 (estimated)		
500,000 (estimated)		

Selection Sort (Unsorted List)		
List Size	**Comparisons**	**Time (seconds)**
1,000 (observed)		
2,000 (observed)		
4,000 (observed)		
8,000 (observed)		
16,000 (observed)		
32,000 (observed)		
100,000 (estimated)		
500,000 (estimated)		

Answer the following questions:

1. Which sort do you think is better? Why?

2. Which sort is better when sorting a list that is already sorted (or mostly sorted)? Why?

3. You probably found that insertion sort had about half as many comparisons as selection sort. Why? Why are the times for insertion sort not half what they are for selection sort?