**CMPSC 412 – Lab-2** (25 points)

**Searching and Sorting**

**Due date: 2/03/2022**

**Note:** attach screenshots of your program and results under each programming exercises. Please make sure that the screenshot is readable. Don’t attach a very small screenshot image.

**Lab Exercises:**

1. Write and implement the algorithm for Linear search, Binary search, Insertion sort, Selection sort, Bubble sort and Merge sort. Calculate the time complexity for these searching and sorting algorithms.

Table-1: tabulate the time complexity for these algorithms with best and worst time complexities.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Linear | Binary | Insertion | Selection | Bubble | Merge |
| Best Case | 1 | 1 | N | N^2 | N | NlogN |
| Worst Case | N | Log N | N^2 | N^2 | N^2 | NlogN |

Linear Seach: In the best case, the target value is the first elem searched and returned. In the worst case, the target elem is not in the list, and the algorithm must search every value before declaring it does not exist and therefore is O(N).

Binary Search: In the best case, the target value is in the exact middle of the array and is returned in O(1) time. In the worst case, the target is not in the list and requires log N iterations through the algorithm. Therefore the time complexity is O(log N).

Insertions Sort: In the best case the array is already in sorted order, but the algorithm must still iterate through all elements at least once to verify, and is thus O(N). In the worst case, the algorithm is in reverse sorted order and for each iteration of the main loop, it can require up to n-1 iterations of the inner loop. Giving it an upper bounded time complexity of O(N^2).

Selection Sort: In the best case, the array is fully sorted, and no swaps are necessary. However, for each iteration of the outer loop, the inner loop still iterates trough the remaining values, giving it complexity O(N^2). In the worst case, the array is sorted in reverse order. However, the number of checks remains the same in the best and worst case. The only difference is in the worse case, we perform a swap every iteration. This still leads to an overall time complexity of O(N^2)

Bubble sort: In the best case, the array is fully sorted, and only requires n-1 iterations through the array. Thus it has a time complexity of O(N). In the worst case, the array is sorted in reverse order. The first element will require n-1 steps to get to the right place. The next will require n-2 and so on. Thus the time complexity will be O(N^2).

Merge sort: In both the best and worst case, the time complexity of merge sort will be O (NlogN). This is because regardless of whether the input is sorted or not, the algorithm first divides the entire array in half recursively. In each recursive call, we will perform comparisons on n/2 data points. And since there are log(n) recursive calls made, the total time complexity is O(N/2 \* log (n)) or O(NlogN).

1. Create a database with the following details for at least 20 students and store it as a text file:

* Student ID
* first name
* last name
* email id
* Major
* Write a program to read the data from the text file. Choose an appropriate data type and data structure (lists, lists of list, dictionary) for storing the information in your program.
* Write a function which takes a parameter and sorts the entire list of students and displays all the details of all students. Your function should sort the list using student id / first name / last name. Implement the sorting using selection sort, insertion sort, bubble sort and merge sort, and print out how much CPU time it took to sort the data. You can import a library to calculate the time.

Show an example for searching a value using linear search. Table-2: Tabulate your recorded time for the linear search and all the four sorting algorithms i.e., selection sort, insertion sort, bubble sort and merge sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linear Search | Insertion Sort | Selection Sort | Bubble Sort | Merge Sort |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

* Table-3: Now apply all the four sorting algorithms on the sorted data (i.e., sort the text file according to student id / first name / last name where the text file is already sorted) and tabulate your recorded time. Print out how much CPU time it took to sort the data.

|  |  |  |  |
| --- | --- | --- | --- |
| Insertion Sort | Selection Sort | Bubble Sort | Merge Sort |
| 0.0 | 0.0 | 0.0 | 0.0 |

* Table-4: Create a different text file and have 20 rows of same student details. Apply the sorting algorithm and tabulate your readings. Print out how much CPU time it took to sort the data.

|  |  |  |  |
| --- | --- | --- | --- |
| Insertion Sort | Selection Sort | Bubble Sort | Merge Sort |
| 0.0 | 0.0 | 0.0 | 0.0 |

* Write a conclusion paragraph about what you understood from this lab exercise. What did you understand from table-1, table-2 and table-3?

This lab attempted to demonstrate the execution time difference between the different commonly used sorting algorithms. Table 1 showed the general case time complexity in terms of Big O complexity. Tables 2 and 3 were meant to measure the time the sorting took on a manually generated text file. However, using only 20 data points in the text file allowed all the points to be sorted too fast to calculate (0.0 seconds). In order to attempt to gain more meaningful results, I increased the number of data points all the way to 120, but it still resulted in all sorting algorithms sorting within 0.0 seconds. Therefore, I conclude that in theory there is a difference in time complexity of the various sorting algorithms, however, in order to visualize it on a computer, it requires a significantly larger input size.

Update: In an effort to see more meaningful time readouts, I attempted to use a variety of other libraries and methods such as datetime and timeit. Unfortunately, all of these different techniques still resulted in every sorting algorithm completing in 0.0 seconds.