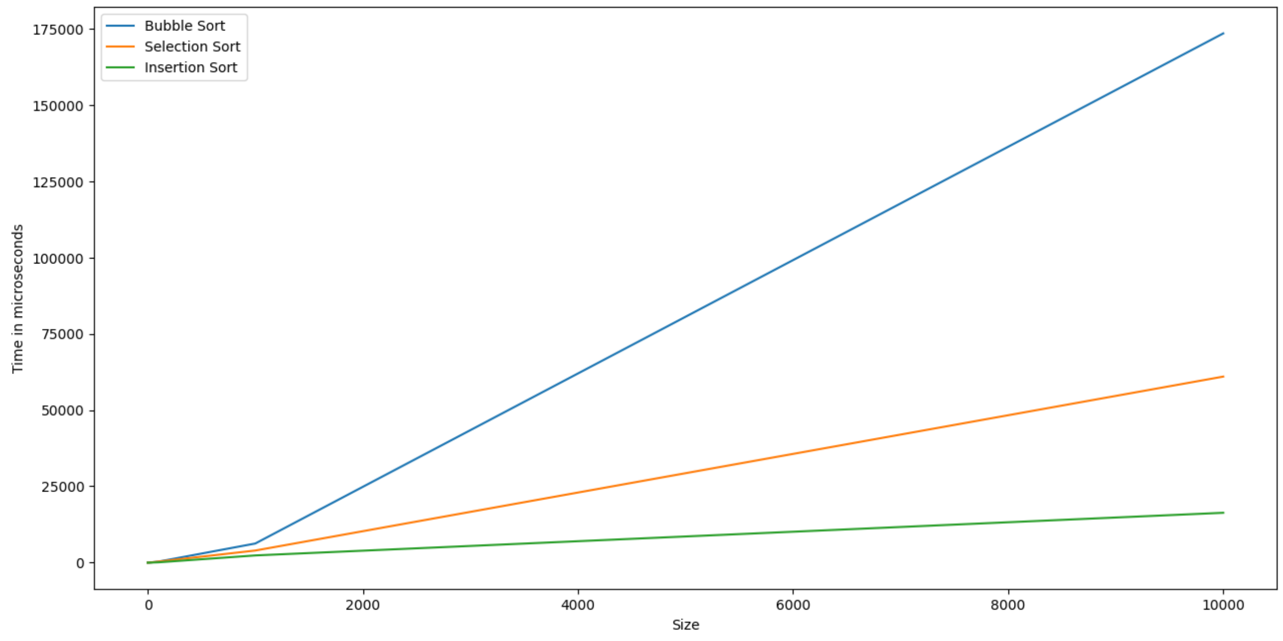
CO322 – Data Structures and Algorithms

Lab 01 – Sorting Algorithms

Ranage R.D.P.R. (E/19/310)

1). How does the performance vary with the input size

Sorting Algorithm	Runtime in microseconds(μs)			
	N = 10	N = 100	N = 1000	N = 10000
Bubble Sort	5.8	344.2	6264.4	173364.4
Selection Sort	5.6	349.4	3969.3	60945.9
Insertion Sort	5.6	106.9	2364.7	16333.9



This indicate that as the size of the array grows, the time required to sort it increases significantly with bubble sort. Consequently, when dealing with larger arrays, insertion sort emerges as the most efficient algorithm in terms of execution time.

Sorting Algorithm	Runtime in microseconds(µs) for N = 100		
	Best Case	**Average Case**	**Worst Case**
Bubble Sort	3.5	350.3	386.5
Selection Sort	3.8	394.6	403.7
Insertion Sort	3.4	121.6	219.4

In best case, where the input array is already sorted, the sorting process involves merely traversing the elements. Consequently, the execution time for the best case is less than both the average and worst cases across all three algorithms. Conversely, in the worst-case scenario, where the input array is in the least favorable order, the execution time is greater than both the average and best cases for all three algorithms. When comparing the three algorithms, insertion sort consistently exhibits lower execution times.

2). Does the empirical results you get agree with theoretical analysis?

	Best Case	**Average Case**	**Worst case**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Bubble Sort: When the array is already sorted (best case), Bubble Sort can go through it in a single pass without making any changes, leading to a time complexity of O(n). On the other hand, in average and worst cases where the array is partially or fully unsorted, Bubble Sort takes more time, specifically O(n^2), as it involves multiple passes and swaps.
The real-world observations match these theoretical expectations, confirming that Bubble Sort performs as anticipated for the given dataset.

Selection Sort: The actual outcomes for Selection Sort match what we expected based on theory. Looking at the data, it's clear that Selection Sort takes O(n^2) time for all situations—whether the array is best, average, or worst-case. This lines up with what we thought would happen; Selection Sort tends to be slower when dealing with arrays that are even partially unsorted. So, the data supports what we anticipated about how Selection Sort performs in different scenarios.

Insertion Sort: Looking at the table and graph, we see that for Insertion Sort, the actual results match what we expected in the average and worst cases, where it takes O(n^2) time. However, there's a difference in the best-case scenario. In theory, it should take O(n) time, but the real results might be different because the way it's implemented doesn't have a condition to stop early.

Even though there's a variation in how well Insertion Sort works in the best-case situation, when we consider how it, along with Bubble Sort and Selection Sort, performs overall, it matches up with what we expected according to theory.

3). How did/should you test your code. Explain the test cases you used and the rationale for use them.

We couldn't measure the execution time of the algorithm as the program's running time is influenced by factors like the complexity of the algorithm, input, and the specifics of the computer system, such as memory setup and the programming language/compiler/operating system used.

In this lab, we're focusing only on the algorithm's complexity and the input to the program. To do that, we calculate the average execution time, assuming that this average is not affected by the specific computer or programming language we're using.