Name: Aljun C. Cursiga
Subject: CS 26/L

NAME OF ALGORITHM	TIME COMPLEXITY	SPACE COMPLEXITY
1.) Bubble	The best case occurs when the list is already sorted. In this case, Bubble Sort only needs to check the list once, resulting in a time complexity of **O(n)**, where n is the number of elements. However, in the average and worst-case scenarios, where the list isn't sorted or is sorted in reverse order, Bubble Sort ends up making multiple passes through the list. This results in a time complexity of **$O(n^2)$**. Essentially, for every element, it compares it with every other element, which gets very inefficient as the list grows larger.	One of the main advantages of Bubble Sort is that it's an **in-place** sorting algorithm. This means it doesn't require any additional memory apart from what's already needed to store the input. Its space complexity is **O(1)**, making it memory efficient even for large lists.
2.) Insertion Sort	In the **best case**, where the list is already sorted, Insertion Sort will only need to check each element once and won't have to move anything around. This results in a time complexity of **O(n)**. In the **average case** and **worst case**, where the elements are not sorted or are in reverse order, Insertion Sort must compare each element with the ones already in the sorted list, resulting in a time complexity of **$O(n^2)$**.	Like Bubble Sort, Insertion Sort is an **in-place** algorithm, meaning it doesn't require extra memory aside from the input list itself. Therefore, its space complexity is also **O(1)**.
3.) Quick Sort	In the **best case**, when the pivot divides the list into two equal halves, Quick Sort can sort the list with a time complexity of **O(n log n)**. This is because, with each division, the size of the problem gets halved, and the number of comparisons grows logarithmically. The **average case** is also **O(n log n)**. In practice, Quick Sort performs very well when the	Quick Sort is a **recursive algorithm**, meaning it uses the call stack to keep track of its recursive calls. In the best and average cases, the depth of recursion is logarithmic, so the space complexity is **O(log n)**. In the worst case, where the recursion depth is proportional to the number of elements in the list, the

	pivots are chosen wisely, leading to balanced partitions and efficient sorting. However, in the **worst case**, if the pivot chosen is consistently the smallest or largest element (which can happen if the list is already sorted or nearly sorted), the algorithm becomes much less efficient. This leads to a time complexity of **$O(n^2)$**, because the list is divided into one large sublist and one small sublist, resulting in more comparisons and recursions than needed.	space complexity can grow to **O(n)**.