Team Megabyte
Project 2
CPSC 335-02

**Complexity Order of the Sort Race Algorithms**

**Insertion Sort**

We created the insertion sort algorithm to grab any values that are higher than the right-most element and place it in front of that element while moving the previous element one space to the left. Using a single while loop, we used a temporary variable to scan what elements are higher than element j, which is the initial right-most element. The algorithm will then pull the first element it determines to be higher than j and push it to the right, past j, by one index space. Because of using only one while loop, the time complexity can ultimately be converted to: T(N) = <u>O(N)</u>. If we had used a nested while loop, it would have worsened to O(N^2), its Big-O.

**Mergesort**

We implemented mergesort with a helper function. First, we divided the N-sized array into three subsets, left, mid, and right, creating N/3 arrays. The helper function (Merge) then pulls the elements from the input array, and sorts them accordingly using if-statements, to determine which subset it goes to, and while loops, to insert them in increasing order. Merging the left, right, and middle, which are all T(N/3), together we obtain O(N). Overall, this will be converted to: T(N) = 3 * T(N/3) + O(N) => N * T(1) + log(N) * O(N) => N * O(1) + O(N log(N)) => O(N) + O(N log(N)) => <u>O(N log(N))</u>. This is also its Big-O running time.

**Gold's Poresort**

We compared all the even-numbered cells to their adjacent cell, and then we did the same with the odd-numbered cells by using if statements. From there, the algorithm starts to swap the even-numbered cells; once finished, it swaps the odd-numbered cells and repeats the process until the entire array is sorted. We used a single for loop to read the entire array length, therefore our poresort algorithm reads as T(N) = <u>O(N)</u>. If we had used a nested for loop to read the array, we would've ended up with O(N^2), its Big-O.

**Quicksort**

We implemented the quicksort algorithm with partitioning. First, we pick an element for pivot and determine values that are smaller/larger than said pivot value. If an element is lower than the pivot, then we push the element to the left-most space by decrementing. If an element is higher than the pivot, we push it to the right-most space by incrementing. We also implemented a swap function to swap elements using a for loop if any element in the lower array is higher than a value in the higher array. This function repeats until the array is fully sorted.
In conclusion, T(N) = T(L) + T(H) + O(N), where L = elements less than or equal to pivot and H = elements more than or equal to pivot. Considering pivot divides array in half, then: T(N) = 2 * T(N/2) + O(N) => <u>O(N log(N))</u>. It avoids becoming Big-O (O(N^2)) since the partitioning sublists end up with an equal size.