Assignment Two

Anthony Rodriguez

Anthony.rodriguez2@Marist.edu

October 2, 2020

1 Comparisons

Selection Sort	Insertion Sort	Merge Sort	Quick Sort
428,367	426,637	428,367	428,367

2 Run Time

1. Selection Sort [$O(n^2)$]
   - Selection sort has a cubic order of growth. The act of swapping elements only is $O(n)$, a constant time. Where we get the most time is through the iterations. The cost for the first FOR loop is c1 and will executed n amount of times. The rest of the code besides the nested FOR loop executes n - 1 times. The time required for the nested FOR loop through every iteration is n - 1 (first iteration will be n), As code executes, the amount of time becomes n + (n - 1) + (n - 2)... etc. This is because on every iteration, an element is sorted and is removed from the unsorted array causing the total amount of unsorted elements to fall by 1. Eventually the total time for the loop will be n(n + 1) / 2. Since we don't care about constants, this translates to $O(n^2)$

2. Insertion Sort [$O(n)$ / $O(n^2)$]
   - Insertion sort somewhat varies running time depending on the input of the array. If the array is already mostly sorted, then the time required would be $O(n)$. As it compares elements and finds no swapping is required, the WHILE loop will not execute leaving the remaining code to have the above run-time. Worst case would be if the array requires the last element to be swapped resulting in a n - 1 and so on. Because the loop has to be iterated through the maximum number of times, we multiply those by (n - 1). At this point because of arithmetic progression, we can come to the conclusion that n(n - 1) / 2, which is $O(n^2)$.

3. Merge Sort [$O(n * log(n))$]

- Merge sort requires dividing the length of the array by half which can be reflected by (log n). Once we take into account the amount of actions to be executed, we can represent that by stating: (log n + 1). Running tasks to find and merge arrays within the sort only results in $O(n)$. The combination of steps above achieves a run-time of n(log n + 1) and once carried over we find the asymptotic run-time above.

4. Quick Sort [$O(n^2)$ / $O(n * log(n))$]
    - Because of the nature of partitioning and the chance of selecting an element that are the highest and/or lowest values, it would force code execution to happen on every instance of a swap. Starting from the beginning, we would have c * n (cost and time). Moving down the partition would result in c(n - 1) and so on. Similarly to Selection Sort and Insertion Sort, we represent the arithmetic progression causing $O(n^2)$. Concerning the best case, while the initial start of the code begins with c * n, the division of sub-arrays results in (2 * cn / 2). Each divide doubles the constant, since we don't care for them, we will eventually arrive to a similar run-time of Merge Sort $O(n * log(n))$.