**Assignment 2 – Megan Bender**

**Run Time’s**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **10** | **100** | **1 mil Almost Sorted** | **1 mil Unsorted** | **10 mil** |
| **Bubble Sort** | <1 sec | <1 sec | 24 mins and 48.972 | 48 mins and 52.450 secs | n/a |
| **Selection Sort** | <1 sec | <1 sec | 21 mins and 12.365 secs | 21 mins and 2.669 secs | n/a |
| **Insertion Sort** | <1 sec | <1 sec | 39.129 secs | 14 mins and 49.608 secs | n/a |
| **Shell Sort** | <1 sec | <1 sec | 0.151 sec | 1.352 secs | n/a |
| **Merge Sort** | <1 sec | <1 sec | 0.445 secs | .92 secs | 2.452 secs |
| **Quick Sort** | <1 sec | <1 sec | 0.157 secs | 0.178 secs | 2.035 secs |

**Algorithm Runtime Classification**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Best Case** | **Average Case** | **Worst Case** |
| **Bubble Sort** | O(n2) | O(n2) | O(n2) |
| **Selection Sort** | O(n2) | O(n2) | O(n2) |
| **Insertion Sort** | O(n) | O(n2) | O(n2) |
| **Shell Sort** | O(n) | O(nlogn) | O(nlogn) |
| **Merge Sort** | O(nlogn) | O(nlogn) | O(nlogn) |
| **Quick Sort** | O(nlogn) | O(nlogn) | O(n2) |

**Justification for Run Time Classification**

Merge Sort: Looking at the above tables, we can see that merge sort is significantly faster than any of our previous sorting algorithms. The reason for this comes from its underlying structure, unlike our previous algorithms this one uses a divide and conquer method which breaks the array up using a tree structure. Continuously, breaking it down until we are down to one item. Then it compares the values as it goes back up the tree, putting the lower value to the left and the higher value to the right. With that said, every time we leave a “level”, from bottom to top of the tree structure, the previous will be considered sorted. Also, one must keep in mind we are working our ways left to right for each level, so if we have 4 items, we will end up with 3 levels in total. Anyways, when we are breaking an array up using this technique, thus we are continuously dividing the algorithm by half each time would be considered a logarithmic algorithm because x = log n => n = 2x. The n is referring to the number of values in the array, thus if we have 8 values

=> n = 8

=> 8 = 2x, where x = log 8

Thus, x = 3.

So, what does this mean, x represents the amount of time it took us to structure the tree. When the previous algorithms had a runtime of n2, that would result in a runtime of 64. Which is significantly longer than a logarithmic run time! Lastly, when we look back to the explanation of this algorithm, I mentioned that we are traversing down the tree structure and back up, causing us to run through this n times, thus causing our big O notation to be, n logn.

Quick Sort: This explanation is relatively the same as merge sort, so keep in mind how we break items up using a tree structure. Again, like merge sort, this is another divide and conquer sorting algorithm. Instead of just breaking up our algorithm in half right away and continuously doing that throughout, we instead create partitions. These partitions are created by first running through the array and comparing two values, a low and a high. The low starts at index 1 and the high wherever the end of the array is, so if we have 10 elements it will be on index 10 (because we are using a 1 based array). Then we will compare these values back and forth if the element on the left is larger it will swap with the right and the right value will go into index 0, then the left side will increment and vice versa, do keep in mind the right side will decrement, until we have come to a central point at which we will create the partition. Then we drop down to the next level of the tree and repeat this process. This is a recursive process, continually calling the previous until we have reached the end and at which time, we will have a sorted array. Finally, the run time of this algorithm is also one is significantly faster than the first four sorting algorithms since it has a runtime of n logn, the same as merge sort. The preceding n here is not from going down and back up the tree, but because we have to run through the entire array to create our partition, this gives us our n. Overall, the runtime as compared to merge sort is not much different, as we can see from the first table, we have very similar times, quick sort has the upper hand by a fraction of a second, at least up to 10 million items.