Joe Romero && Brandon Hang

Project 2

CSCI-41

12-7-21

As shown in our project's code we ran the following sorting methods for each of the vector groups that we had created and the data is the following:

Size	Bubble	Insertion	Selection	Heap	Quick
2000	1.89383s	1.44201s	2.90646s	0.026781s	0.035609s
4000	7.65638s	5.8423s	11.5738s	0.049667s	0.063793s
6000	17.3535s	13.0161s	26.0941s	0.070358s	0.098405s
8000	30.7418s	23.381s	46.2476s	0.094001s	0.124206s
10000	48.4632s	36.5658s	72.1851s	0.11774s	0.163323s

As shown in the data above you can clearly state that the Heap Sort method was the quickest and that the Selection Sort was the slowest.

This is interesting because the way Heap sort works is quite similar to Selection Sort. They both take the largest value and put it in the last position of the vector but a key difference is that heap sort doesn't take the extra time to take an additional scan of the unsorted vector/array. Granted that selection sort has a time complexity of n^2 it clearly showed that behavior within the span of 2000 units and 10000 units.

As shown in the chart below we're given the time complexities, notice how Bubble, Insertion, and Selection all have n^2 time complexities and take the longest to finish. It's notably clear that due to the doubling of sizes within the vector itself these sorting methods double in time after every increase.

This also goes for both Heap Sort and Quick Sort have time complexities of nlogn which aren't as affected by the nature of this project; they are increasing in time but they're not doubling in time as the n^2 time complexity methods were.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Now these times are completely subjective to what device you're using. My computer has these times based on the components I have. Brandon ran the same code and had much quicker times than I did, but the change in times were the same regardless of components. That is because of the time complexity of the sorting methods. It's quite interesting to know that it still gives the same difference ratio throughout the difference in components.