Michael Nguyen
mnguy181@ucsc.edu
2/15/2021

CSE13S Winter 2021
Assignment 5: Sorting: Putting your affairs in order
Writeup Document

Analysis on Bubble Sort:

When experimenting with bubble sort, I found that in the best case scenario where items were already sorted, bubble sort's time complexity was found to be O(n). On average, however, I found that the Bubble Sort continued to do worse as more items were filled into the array. Thus, on average, Bubble Sort's time complexity is O(n^2). In the worst case scenario such as when elements in the array were sorted in descending order, bubble sort's time complexity was still O(n^2).

Overall, I feel that Bubble Sort was probably the most inefficient sorting algorithm when the array was not already sorted (which is most of the time). This is because it analyzes pairs of neighboring elements at a time, swapping them until all elements in the array are in their sorted positions in the array.

Analysis on Shell Sort:

When experimenting with shell sort, I found that in the best case scenario where items were already sorted, shell sort's time complexity was O(n) similar to Bubble Sort. On average, its time complexity was O(n^(5/3)) which was significantly better than Bubble Sort as I increased the size of the array to fill with more elements. However, because the sort was so dependent on its gap sequence, there were cases where Shell Sort performed poorly. One example is when a gap size was too big for the array size. Thus, it caused Shell Sort to swap elements inaccurately, making more iterations necessary. In this worst case scenario, Shell Sort's time complexity was O(n^2).

While Shell Sort, on average, performed much better than Bubble Sort, I learned that it is not considered to be a stable sort since it is highly dependent on the gap sequence that it is provided. Finding the optimal gap sequence takes a lot of time to find in order to optimize the sort further, and because of this, it is considered to be an unstable sort in comparison to bubble sort which you can expect to be a certain time complexity almost every time aside from its best case scenario.

Analysis on Quick Sort:

When experimenting with Quick Sort, I found that on average, it was one of the most efficient and fastest sorting algorithms that I implemented. In its best case scenario, Quick Sort's time complexity was O(n log(n)), and on average, its time complexity was also O(n log(n)). However, in its worst case scenario, for example, when elements were already sorted, Bubble Sort and Shell Sort performed much better. When elements were already sorted, Quick Sort's time complexity was O(n^2). However, in cases where there were many elements, it outperformed Bubble and Shell Sort.

I learned that Quick Sort, on average, was one of the most efficient sorting algorithms in comparison to Bubble Sort and Shell Sort. While there are cases that showcase that Quick Sort was worse than Bubble Sort, these cases can be deemed incredibly rare, when an array contains a list of the same elements.

Analysis on Heap Sort:

When experimenting with Heap Sort, on average, and in every case, Heap Sort's time complexity was O(n log(n)). Its worst case time complexity was also O(nlogn)). However, it was still slower than Quicksort when it came to the number of moves and comparisons required to sort, thus making it less efficient than Quick Sort. In heap sort, I found that even if all of the elements in the array were already ordered, there would still be swaps in order to "sort" the array again. Thus, it is still outshined by the other algorithms when the array is already sorted.

I learned that Heap Sort was the second most efficient sorting algorithm implemented on average, performing better than Bubble Sort and Shell Sort when there were a lot of elements in the array. When the elements are already sorted however, there were still more swaps than Quicksort which had a small amount of swaps, thus making it less efficient than Quick Sort.