Assignment 5 Writeup - Jack Vento

Bubble

Complexity

Bubble sort has a time complexity of $O(n^2)$ in all cases (best, average, and worst) due to it having two loops that will always carry out through the entirety of the array. It is incredibly inefficient, taking `n(n-1)` comparisons and half that many swaps. From a memory perspective, it is actually rather efficient with constant complexity as it sorts in place (no additional arrays or memory needed). Its constant is relatively mild compared to the logarithmic algorithms but significantly larger binary insertion sort's.

Testing

I found that bubble sort became completely inviable much quicker than every other algorithm. For n > 30,000 the entire program hung on account of bubble sort for upwards of 30 seconds, proving just how inefficient it is. When I removed bubble sort from my tests, the program often terminated for large data sets (n = 50,000) in ten or less seconds.

Conclusions

I believe bubble sort is somewhat sufficient for small data sets (though binary insertion sort is probably better for these applications) and woefully insufficient for anything larger than 30 or so elements. I believe adding a flag to the sort's inner loop could significant improve its effectiveness since it would be able to tell whether an array is sorting in O(n) time (better than every sort except binary insertion).

Binary

Complexity

Binary Insertion sort has a time complexity of $O(n^2)$ in average and worst cases due to it having two loops that will always carry out through the entirety of the array. In its best case, it will not execute the inner loop, leading to a complexity of O(n). Although it features the same amount of swaps as bubble sort (n^2), it performs the least amount of comparisons of any sort: O(logn). From a memory perspective, it is very efficient with constant complexity as it sorts in place (no additional arrays or memory needed). Its constant is among the lowest of sorting algorithms since it is great for small data sets (low overhead) and greatly improved from other quadratic sorts due to it utilizing a binary search.

Testing

I found comparisons for binary insertion were always smaller than every other sort, and it surprisingly was able to quickly sort an array of size 100,000. However, there were so many swaps that they overflowed the moves variable. This leads me to believe that in environments where swaps are expensive, this sort is a bad solution.

Conclusions

Even moreso than bubble sort, binary insertion is fantastic for small data sets from a swaps perspective and great from a comparisons perspective at any data range. Like bubble sort, by throwing a flag into the inner loop, the runtime could be significantly improved rather than forcibly looping through the array every inner iteration.

Quick

Complexity

Quick sort has a complexity of O(nlogn) in its average and a worst case of $O(n^2)$ when the pivot is the largest or smallest element. Fortunately, this worst case can be avoided by strategically choosing a median pivot. As for memory, the only additional memory comes from the recursive stack frame. Its constant is rather large (larger than the quadratic sorts) due to its logarithmic nature not shining until being tasked with large data sets.

Testing

Quick sort really shined when I threw large data sets at it. Unlike every other algorithm, it was able to quickly sort an array of 10,000,000 and even handle one of size 100,000,000 in reasonable time. It was efficient with small sets but really did not shine until tasked with these large sets.

Conclusions

I found my version of quick sort could be significantly improved by intelligently picking a pivot, rather than the first element than often leads to worst case complexity. Even with that said, it performed amazingly under a variety of circumstances: stressing the need for a logarithmic sort when tasked with large data sets.

Shell

Complexity

Although memory is always constant (unless gaps are stored in an array prior to execution), time varies with the gap method chosen. In my implementation, it has a time complexity of $O(n^{5/3})$. Its constant is rather low (seemingly on par with binary insertion) since it performs significantly less swaps than the quadratic sorts but also far more compares.

Testing

Shell sort did better than the quadratic sorts with medium and large sized data sets but far worse than quick sort with large sets. Shell started to hang around 100,000 elements: overflowing the comparison count incredibly quickly. The compares got unreasonably large rather quickly, even though swaps grew at a rate not that much larger than quick sort.

Conclusions

Shell sort seems effective for completely random arrays where continuous elements are few and far between. It does not seem particularly good for nearly sorted arrays because it performs so many comparisons, even though it is not frequently swapping. In environments where comparsions are expensive, shell sort is woefully inappropriate, though it is more efficient on them than the quadratic sorts for random arrays.