Introduction  
  
Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criterion (Java Foundations and Data Structures, pg. 659). In this assignment we will be specifically looking at the Bubble Sort and the Shell Sort sorting algorithms.

In a bubble sort, values are compared with the adjacent neighbor with the higher value being moved to the next index and the lower value being moved to the lower index effectively swapping. By this process the highest value in the list will ‘bubble’ to the end or top of the list and thus the name bubble sort. In the best case or the sorted array bubble sort will take O(n) comparisons as only one pass is required. Keep in mind this is for the optimized bubble sort. For the worst case bubble sort will take O(n2) comparisons as it needs to iterate through both for loops. For the unoptimized bubble sort this is always the case as it always iterates through both for loops. The worst case for bubble sort is when the smallest element is in the last index of the array or the array is in descending order. We will discuss this further below.   
  
In a shell sort, values are compared with elements separated by a gap value typically starting at the size of the array divided by 2. After this the gap is again decided by dividing by 2 again and values are compared with higher and lower being swapped into correct locations. In a shell sort the best case is when the array is already sorted and in this case it will require to go through each gap value and iterate. This leads to O(nlog(n)) comparisons. In the worst case we would have to go through every index for both loops leading to O(n2) comparisons.

Results  
  
Below I will be sharing the results of the assignment in the form of graphs. These are graphs showing the length of the arrays tested of size 10, 100 and 1000 against the various parameters such as comparisons needed for sorting, number of swaps required and time taken for execution. These tests are generally organized into two categories sorted and unsorted and all graphs presented together will either demonstrate results for sorted arrays or unsorted. For example, if the heading is Array Length vs Comparisons in the Sorted arrays; this indicates that each sorting algorithm (Shell Sort, Bubble Sort and Bubble Sort 2) were tested against the sorted arrays of size 10, 100 and 1000 and these are the results being viewed.

Sorted: Array Length vs Comparisons



Unsorted: Array Length vs Comparisons



Sorted: Array Length vs Swaps

No swaps occurred in this category as all the arrays were sorted.

Unsorted: Array Length vs Swaps

In this category both the bubble sort and optimized bubble sort had the same number of swaps as expected for the same number of out of place elements and therefore the graphs have been made into one graph to represent both.



Sorted: Array Length vs Execution time (microseconds)



Unsorted: Array Length vs Execution time (microseconds)



Discussion

For the discussion we can break the categories down into the three parameters being compared here which would be comparisons, swaps and execution time required for Shell sort versus Bubble sort versus the optimized Bubble sort.

For comparisons we can notice a few things right away for sorted vs unsorted lists. In the sorted case we can see that the bubble sort is extremely efficient with O(n). Of course this is the best case and we can see that for the sorted lists the optimized bubble sort is definitely the clear winner here. It’s also notable that the bubble sort without optimization does much poorer having nearly n^2/2 comparisons needed each time or O(n2) comparisons needed. For example array size 100 needs 100^2/2 ≈ 5000 or in this case 4950. When compared to the shell sort it fares better than the normal bubble sort but fails to get the efficiency of the optimized bubble sort. We can see that the shell sort in this case tends towards O(n(log(n)) which is not as good as O(n). So in the sorted case for comparison Optimized bubble sort is the clear winner with O(n) comparisons for comparison efficiency in the sorted case with shell sort being better than normal bubble sort, and bubble sort being the most inefficient. In the unsorted case for comparisons, we can see a different result. In this case we see that Shell sort tends towards O(n(log(n))2) whereas bubble sort and the optimized bubble sort both tend towards O(n2). It is also interesting to note that although optimized bubble sort was much better in the sorted case in the unsorted case comparisons for the bubble sort and optimized bubble sort were comparable at lower sizes of the array and very slightly better for optimized at larger sizes but at a negligible amount. From this we can see for the unsorted array shell sort is the clear winner for comparison efficiency tending towards O(n(log(n))2) comparisons. In the worst case for number of comparisons we should choose the shell sort as it has the lowest growth function for increasing values of n. In the best case we can choose the optimized bubble sort if our list is already sorted however we typically do not look at the best case as discussed below.

Moving on to swaps we will be looking at the unsorted cases because the sorted cases had no swaps occurring. Once again, the Shell sort is the clear winner in this category for swap efficiency. We can see that shell sort achieves nearly linear efficiency with O(n), on the other hand bubble sort and optimized bubble sort both require higher swaps. In terms of swap efficiency, a shell sort shows the lowest growth function with increasing list size for the number of swaps. In the worst case for number of swaps we should choose the shell sort.

Finally, we can discuss the execution time it takes and which algorithm efficiency in this category. For this category we can see that the algorithm with the lowest growth function is clearly the shell sort for large values of n. For array sizes of 1000 we see that it took the shell sort algorithm 2392.6 µs to execute whereas it took the bubble sort and optimized bubble sort algorithms 25362.9 µs and 12365.4 µs respectively. We can see that the bubble sort algorithm is larger on a scale of 10 times more than the shell sort in terms of time efficiency. We can see that time complexity for the shell sort is relatively linear as it goes up by a factor of 10 each time. This is not true for the bubble sort algorithms as they increase by an increasing factor. From this we can tell that the growth function for the shell sort is smaller than the growth function for the bubble sort and the optimized bubble sort. If we were to rank these on efficiency based on the tests done the most efficient would be the shell sort, then the optimized bubble sort and finally the normal bubble sort. In the worst case for execution time we should choose the Shell sort for maximum efficiency.

Conclusion

When comparing algorithms, we compare the worst case to have the most efficient programs possible. From comparing the data in this assignment, we can see that in the worst case the Shell sort performs the best, has the lowest growth function and is better than both the optimized bubble sort and the normal bubble sort. In terms of algorithm efficiency, the normal bubble sort is the worst as expected and the optimized bubble sort is slightly better than that. In the worst case the shell sort tends towards O(n(log(n))2) and in the worst case the bubble sort tends towards O(n2). The exponential growth function is much more inefficient at larger values of n than the logarithmic growth function. For these reasons we can conclude that when comparing these algorithms, in practical use the shell sort would be more efficient.