16 Elements

	Ordered		Reverse		Random	
	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges
Bubble	225	0	225	120	225	48
Selection	120	15	120	15	120	15
Insertion	15	0	120	15	63	48
Quick	120	135	56	71	35	45
Shell	0	0	32	32	28	28

2000 Elements

	Ordered		Reverse		Random	
	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges
Bubble	3996001	0	1999000	3996001	3996001	998658
Selection	1999000	1999	1999000	1999	1999000	1999
Insertion	1999	0	2000999	1999000	1000657	998658
Quick	1999000	2000999	999000	1000999	12063	13396
Shell	0	0	10400	10400	20462	20462

Quicksort Time Complexity

Best Case: O(Nlog(N))

Average Case: O(Nlog(N))

Worst Case: O(n^2)

Quicksort works by recursively partitioning an array into smaller arrays based on their values in relation to a chosen pivot value (typically first or last element of the array). Since it is a recursive algorithm, we can determine efficiency via tracing height of the created tree. To illustrate, a quick example of best case would be an array of {1,8,3}. Choosing 3 for our pivot, we partition the array into two single element arrays, 1, as the smaller, and 8, as the larger. We then merge them together and create the final array sorted, {1,3,8}. Since the example is so small, it is easy to visualize that the actual height of the tree is only N*log(n). However, the average case of this algorithm is the same time complexity, hence why it is important to understand that the pivot point of the algorithm is arbitrary. The worst-case time complexity of quicksort occurs when the list has already been sorted. Now utilizing this sorted array, {1,3,8}, choosing the final element as the pivot, the algorithm will operate as follows. Since 1 and 3 are smaller than 8, they will be put the “smaller half” partition, the larger partition would be empty. Choosing the new rightmost element, 3 is the new pivot, and 1 is sent to the “smaller than” partition. The algorithm has a height of n^2 in this scenario, which notably is very inefficient compared to average and base case. Quicksort is better than all the elementary sorts when it comes to time complexity, unless it is being utilized in its worst case. In its worst case we can observe in the above tables that quicksort is indeed up there with some of the most inefficient sorts.

Shellsort Time Complexity

The shell sort is a variation of the insertion sort where elements of an array are compared over a gap of space and continually swapped. The general idea is that you start with a large gap swapping largely contrasting numbers across a large gap, and progressively make the gap smaller as you go. Different gap sizes can shift the time complexity of the shell sort by changing the gap size, so there is not necessarily a static best- and worst-case scenario, like in other sorts. However, given increment = H/2 as the standard used in our example, best and average case would be Nlog(n), and the worst case would be n^2. In the best case the array is already sorted, where the inner statement cannot be true, so the time of the inner loop becomes O(1). Looking at the table above, even with the increment of h/2, which is considered poor, the shell sort has very impressive performance. It completely dominated the quicksort in every example except for the random 2000 element dataset. Needless to say, no elementary sort even comes close to the shell sort. This is likely in part due to the condition that the shell sort's worst case, where the array is required to be sorted in reverse order is never met. The shellsort is a very interesting algorithm, and some questions I have relate to the very worst imaginable case-scenario, as well as the best general increment for the algorithm.