Brandon Cohen

Input	HalfHeap Sort	HalfSelection Sort	Standard Sort	Merge Sort	InPlaceMerge Sort	QuickSelect	WorstCase QuickSelect
1000	0 ms	5.7 ms	0 ms	0.33 ms	0 ms	0 ms	341 ms
31,000	8.3 ms	4361 ms	8.6 ms	26 ms	16 ms	1 ms	
1,000,000	278 ms	N/A	281 ms	861 ms	539 ms	41 ms	
Average Performance	O(nlogn)	$O(n^2)$	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	

Table 1. Average Running Time and Time Complexity

Input File #	HalfHeap Sort	HalfSelection Sort	Standard Sort	Merge Sort	InPlaceMerge Sort	QuickSelect
1	0 ms	6	0 ms	1 ms	0 ms	0 ms
2	0 ms	6	0 ms	0 ms	0 ms	0 ms
3	0 ms	5	0 ms	0 ms	0 ms	0 ms
4	8 ms	4337 ms	9 ms	26 ms	16 ms	1 ms
5	8 ms	4313 ms	9 ms	27 ms	16 ms	1 ms
6	9 ms	4434 ms	8 ms	26 ms	16 ms	1 ms
7	281 ms	N/A	279 ms	862 ms	564 ms	41 ms
8	276 ms		276 ms	871 ms	515 ms	39 ms
9	276 ms		287 ms	851 ms	537 ms	42 ms

Table 2. Running Time for Each Input File

In this report, the running time and complexity of 6 common sorting algorithms are compared to find the median of an array. On average the best sorting algorithm was quickselect and half heap sort with C++ standard sort (std::sort) right behind. The worst sorting algorithm was by far half selection with merge sort and in-place merge sort performing faster than selection sort but significantly longer than the other sorting algorithms.

As shown in Table 1, half selection sort was incredibly inefficient to the point that an input array of 1,000,000 integers could not be sorted using this algorithm within an acceptable amount of time. This is because the number of comparisons required is n*(n-1)/4 which is indicative of the double for loop that can iterate over the whole array giving rise to $O(n^2)$.

Interestingly, four of the remaining five algorithms had a time complexity of O(nlogn) even though their performances varied significantly. The worst performing one out of them was merge sort even though it is known as an effective divide-and-conquer algorithm that runs in O(nlogn) at the worst case with the most optimal number of comparisons at N-1 comparisons. The number of comparisons is optimal due to the (logN)/2 temporary arrays being created during the recursive calls, but these temporary arrays are a problem because merging two sorted lists uses a linear amount of memory. Another problem is that copying the temporary arrays back slows the sorting down. The increased overhead of creating temporary arrays and copying them back was a major reason why it performed poorly. Another reason why copying the arrays slowed down merge sort was the fact that copying objects in C++ is more expensive than comparing objects. Even though there are many shortcomings to merge sort, an extension of merge sort called in-place sorting that uses the same array for sorting was significantly more effective at sorting. The improved running time of in-place sort also confirmed that the creation of temporary arrays and copying them back played a major role in the running time for merge sort.

While the C++ implementation of std::sort can vary for each compiler, the std::sort used in this report was introsort. Introsort is a hybrid sorting algorithm of O(nlogn) that begins with quicksort for its efficiency but uses heapsort when the recursion depth nears quicksort's worst-case scenario and insertion sort at small partition sizes to optimize performance. Currently, intro sort is one of the best overall sorting algorithms due to its optimized bounds as seen in this report with its superior running time compared to merge sort and in-place merge sort. The only O(nlogn) algorithm that was better than std::sort was half heap sort due to its best and worst-case scenario being O(nlogn). The algorithm first begins with building the heap which is guaranteed to be less than 2N comparisons. During deleteMin, the ith deleted item uses at most less than $2\log (N - i + 1)$ comparisons, for a total of at most $(2N \log N - O(N))/2$ comparisons when $N \geq 2$. In the worst case, there are at most $(2N \log N - O(N))/2$ comparisons used to sort, but on average, heapsort uses fewer comparisons than its worst case making it more powerful than other sorting algorithms that have the worst worst-case comparisons.

By far the best algorithm was quickSelect with an average running time of O(n). No other sorting algorithm came close to the efficiency of quickselect due to its one recursive call and partitioning method that reduced the size of the problem each time elements were partitioned. Unfortunately, one of the only problems of quickSelect is that the worst-case scenario is $O(N^2)$ because the depth of the recursive depth is at most N. To counteract this, median-of-three pivoting strategy was used because it forces the worst case to be negligible. To test the worst case, an array of 20,000 numbers was arranged to force the quickSelect algorithm to approach its worst case, but as can be seen in Table 1, it was only 2 ms. The basis of the worst-case scenario for quickSelect is that the first two smallest numbers must be at the first and last position respectively. The rest of the numbers must be put in pairs of i and i+1 up to halfway in the array {(3, 4), (5, 6), (7, 8), …, (9999, 10000)}, and the rest of the array can contain any numbers from

10001 to 20000. In all, the order of best to worst algorithms is (1) quickSelect, (2) standard sort (std::sort), (3) heapsort, (4) in-place merge sort, (5) merge sort, and (6) selection sort. My expectations were mostly correct because I knew quickSelect and std::sort are superior sorting algorithms. I thought heapsort would be right behind them but would not be able to compete with the very optimized introsort algorithm or O(n) average running time of quickSelect. I knew selection sort would perform poorly, but I did not realize how bad it was until I tested it. I also knew that in-place merge sort would be superior to merge sort because it is very expensive to make temporary arrays and copy them.