Emmanuel Olatunde

Nathaniel Hornik

Charles Baiden

**Code Analysis**

**Recursion & While-Loops**

From our experimentation on the various sorting algorithms: we realized that all Recursive implementations out-performed their iterative While-loop counterparts except Selection sort. For Insertion sort, its recursive implementation was 1469.19% faster. Recursive Heap sort was 5.65% faster than While-loop Heap sort. Recursive Quick sort was 99.35% faster than its While-loop variant. More so, Recursive Radix sort was 17.25% faster than While-loop Radix sort. However, we realized Selection sort was an outlier when its Recursive implementation was 23.9% slower than iterative version. Amongst the algorithms, the most consistent was Radix sort when taking into account the sorting time variations in each vector-size. In this experiment, we performed a more rigorous analysis on the Recursive variants of the algorithms to obtain data on the performance of each algorithm in comparison to one another.

**Analysis of each sorting Algorithm – Recursive Implementation**

**Insertion Sort**

As per our analysis, Insertion sort turned out to be one of the slowest of the algorithms especially for large vector sizes, typically those with vector-sizes exceeding a 1000. Insertion sort averaged a sorting time of 47ms for vector-sizes below a 1000. This is reasonable, perhaps even applicable, considering Insertion sort has a time complexity of O(n) in best case and O(n^2) in the average case. Our experiment began recording outrageously high sorting times for larger vectors accumulating an average of 6992.179ms for vectors with sizes in the range (1000-15000). Although this was not surprising considering it has a time complexity of O(n^2) in the worst case, our data suggests that one should avoid insertion sort when dealing with large vector sizes.

**Selection Sort**

Our experiment yielded the worst results during selection sort. Its average sorting-time for large vectors is staggering high. It produced results quite similar to Insertion sort except that it ran 16.68% slower on average, this is no small difference. For vectors less than 1000 and those above, it averaged a sorting time of 101.5ms and 8775.68ms respectively. Our analysis re-validates the fact that selection sort has a time complexity of O(n^2) in best, worst and averages cases. The repeated recursive comparisons to find the minimum value in the unsorted portion of the vector accounts for the many reasons’ selection sort is not admirable for large vectors. Like Insertion sort, Selection sort is only recommended for small vectors size, typically those below a 1000.

**Quick Sort**

Quick sort, as its name implies is indeed quick! With a time-complexity of O(n\*log(n)) in best and average cases, it returned desirable sorting times. It averages about 50.5ms for vectors within the range (500-15000). Despite its O(n^2) time-complexity in worst case, it only averaged at 109ms for a vector size of 15000 random numbers. This is indeed fast as it is 190.71% faster than the average recorded sorting-time of Insertion and Selection sort. In many cases, it appears that a recursive implementation of Quick sort appears to return one the fastest sorting-time, stooping only to Radix sort. From our analysis, Quick sort was faster than Radix sort in 10 of the 30 vector intervals especially in the 2500-11000 vector-size range.

**Heap Sort**

Heap sort was the only algorithm that utilizes the binary heap data structure and was which we were particularly excited to discover the results. With a time-complexity of O(n\*log(n)) in best, average and worst-case scenarios, it produced a sorting time quite comparable to fast algorithms like Radix and Quick sort. On average, it was 34.75% slower than Radix sort and 31.99% slower than Quick sort. With an average sorting-time of 66.7ms for vector sizes in range (500-15000), it is indeed a feasible sorting algorithm regardless of size.

**Radix Sort**

As per our experiment, Radix sort was by far the quickest and most consistent sorting algorithm. With a linear time-complexity, O(n), it returned results that satisfy expectations. It is indeed a unique sorting algorithm being the only algorithm that does not compute comparisons between elements in the vector. It utilizes ‘Buckets’, a data structure adapted from vectors. A Bucket holds vector elements whose nth digit corresponds with the currently processed place-value (units, tenths, hundredth, etc.). Elements are placed and dumped out of Buckets until the largest place-value has been processed. As a result of the fact that it utilizes no element comparison, it remains, on average, the fastest sorting algorithm. It returned the lowest average sorting time of 49.5ms for vector-size range (500-15000).

**Combination of Recursive Sorting Algorithms - Graph**

**Combination of While-Loop Sorting Algorithms - Graph**