K-WAY MERGE SORT ANALYSIS - 1A

CSCI 220 - DATA STRUCTURES AND ALGORITHMS

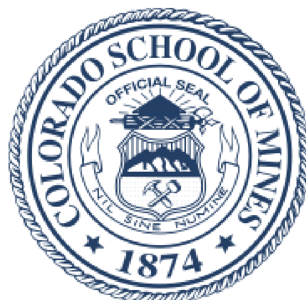
By

MR. DAVID VILLALOBOS

Professors

Keith Hellman

Christopher Painter-Wakefield



Department of Computer Science,

Colorado School of Mines, CO

March, 2023

ALGORITHM ANALYSIS

COMPLEXITY AT EACH MERGE STEP

To calculate the complexity of the Merge Step, we need to analyze the comparisons that are being made within each iteration and see which values are being compared how many times. For each of the n elements, we need to compare it to each last value in partition k. This means that worst case, the algorithm is going be making kn comparisons to find the largest value. This tells us that the worst-case complexity of each merge step part is: $\mathbf{O(kn)}$.

RECURRENCE RELATION CALCULATION

The two major parts of this algorithm include splitting and merging. Effectively we are dividing our problem into **k** sub-problems and each of those sub-problems is going to have a sub-problem size of $\mathbf{\frac{n}{k}}$. We previously calculated the worse case complexity of each merge step as $\mathbf{O(kn)}$ which will be used in our calculation. Below is our calculated Recurrence Relation Equation.

$$\text{Recurrence Relation Equation: } \mathbf{T(n) = kT(\frac{n}{k}) + O(kn)}$$

K-WAY MERGE SORT ALGORITHM TREE ANALYSIS

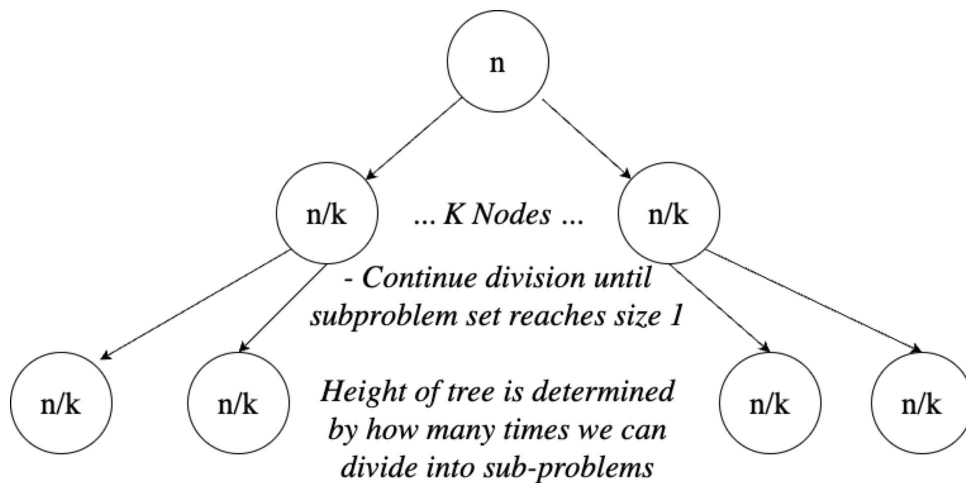


Figure 1: Binary tree demonstrates the complexity of K-Way Merge Sort Algorithm, this visual assists in calculating the complexity of the algorithm using Tree Method Analysis.

- Dividing into Subproblems: In the K-Way Merge Sort Algorithm Tree, we see that we are dividing the original array of size n recursively until we hit our base case of $size = 1$. The step of dividing the array is fairly straightforward since we're only calculating indices. This algorithm step has $\mathbf{O(n)}$ complexity as we are simply traversing the original data set and calculating the index where it'll be placed.

- Merging the Arrays: This is the more complex step of the algorithm which involves merging sorted arrays into a single sorted array. By using tree method analysis, we know that in each row of the binary tree, we will always have a total number of n elements at each level in the tree, to calculate the complexity of the tree at any given tree level, we multiply the complexity across, then multiply by the height/depth of the tree. Going across, we know that the complexity in any given row is going to be $\mathbf{O(nk)}$.

- Binary Tree Depth: As we break the original data set n into smaller data sets we are dividing by k at each level. This means that the K-Way algorithm tree analysis has a depth/height of: $\log_{\mathbf{k}}(\mathbf{n})$

- Cumulative Algorithm Complexity: Given that the complexity of work done at each level doesn't change, we can calculate the complexity of the algorithm by combining all components through the tree analysis method. Adding up each individual row, we will always have $\mathbf{O(nk)}$ then we can multiply by the depth to calculate the complexity.

  Asymptotic Algorithm Complexity: $\mathbf{f(n) = nk \log_k(n)}$

  $\lim_{\mathbf{k \to n}} \mathbf{nk} \log_{\mathbf{k}}(\mathbf{n}) = \mathbf{O(n^2)}$

- Analysis: As k approaches n, the depth of the tree approaches 1, however, the merge cost approaches $\mathbf{O(n^2)}$ and this becomes very inefficient for any large K. After plotting the complexity, the K value is most optimized at **3** regardless of the data input size n.

EXPERIMENTAL DATA ANALYSIS UN-OPTIMIZED ALGORITHM

- Following our theoretical analysis we will now implement a testing suite and compare our results to a multitude of various K and n values. The test suit will reveal the execution time for these different variable numbers, we'll plot the change in input sizes n and K values vs time to see what happens with different cases and if an optimal K value can be identified for this algorithm. This will assist us in verifying our theoretical analysis accuracy and identifying the realistic algorithm parameters.

  Caution: While algorithmic analysis is an excellent tool for evaluating algorithms, it does not always tell the entire story. Modern computers have such immense complexity that unrelated tasks running on hardware or exterior variables will significantly impact algorithm performance.

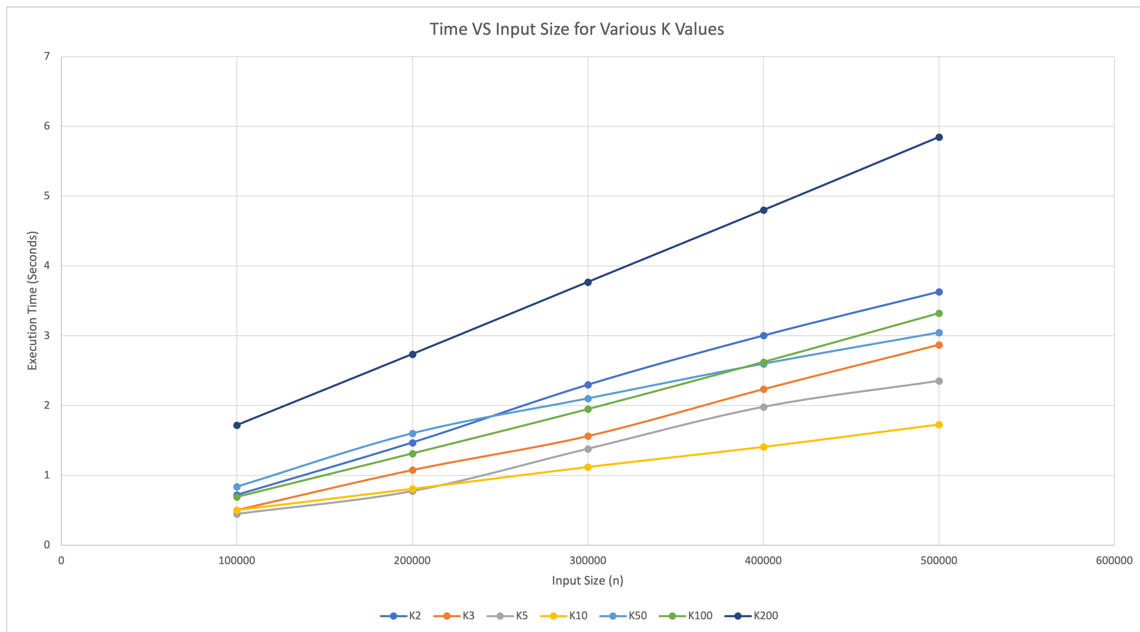


Figure 2: Time V.S. Size of Input for Various K Values

- Plotting the algorithm execution time for various input k values, there are no results that are consistent with our testing. Our theoretical analysis found that the most optimal value for **K** is **3**, this experimental data set shows that the optimal K value for all data sizes n is $K = 10$. Interpreting this, the execution time for each K10 value regardless of n size is smaller, these results can likely be traced to external variables, not the algorithm itself. It is very likely that during the execution of the testing suite, computer hardware

ran the algorithm in the most optimized manner to ensure other tasks were given CPU time thus resulting in inaccurate results.

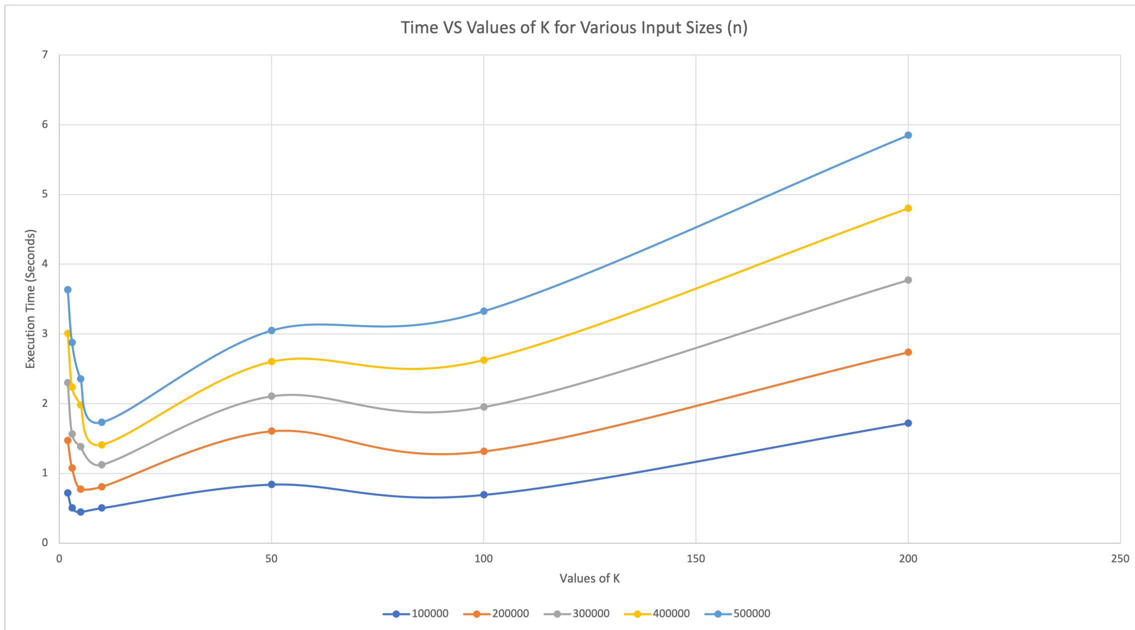


Figure 3: Time V.S. K for Various Sizes of Input n

- Plotting the algorithm execution time vs k values for different sizes of input n, we can see that an optimal range of k values hovers between 2-5 which perfectly supports the hypothesis. This data plot demonstrates that regardless of data input size n, the optimal k value is somewhere between the 2-5 range in which our prediction of $K = 3$ falls perfectly in line. This graph also shows that there's a direct correlation with the number of items n to the algorithm execution time which falls in line with our theoretical analysis where we're multiplying by nk multiple.

Conclusion: Our analysis, through a timed test suite, revealed some pretty helpful information. Figure 2 in which various inputs n were tested for K values, there were inconclusive results and these were attributed to external factors. However, figure 3 revealed some important statistics that perfectly support the theoretical hypothesis. The figure shows that for all input data sizes n, there's always an increase in performance (decrease in execution time) around K values of 2-5 which supports our theoretical analysis of the most optimal value $K = 3$.

Table 1: Execution times for different input sizes (in seconds) for un-optimized algorithm

Input Size (n)	K Values						
	2	3	5	10	50	100	200
100000	0.719 267	0.502 45	0.446 071	0.502 393	0.839 922	0.692 03	1.718 61
200000	1.470 12	1.076 32	0.773 708	0.807 235	1.604	1.314 56	2.737 93
300000	2.298 99	1.563 77	1.380 62	1.120 57	2.103 99	1.9499	3.772 05
400000	3.003 99	2.236 15	1.980 96	1.409 28	2.600 29	2.624 77	4.803 02
500000	3.631 95	2.873 13	2.353 58	1.730 73	3.047 38	3.324 24	5.850 39

Note: K values denote the number of partitions.

ALGORITHMIC VARIATIONS (IMPROVED ALGORITHM)

Optimized Algorithm Approach: In this analysis we will improve the current K-Way Merge Algorithm without changing the current data structure. This new algorithm will optimize the merge process, the algorithm will select the 2 smallest arrays and remove them from the collection, merge both arrays, and do this until only 1 array remains.

ALGORITHM ANALYSIS

- Complexity of Merging Operations: Every time 2 arrays are merged, the number of comparisons is always going to be proportional to the sum of the size of the arrays. Since only n number of elements are being moved per merge operation. Looking at a tree approach problem, we know that at each stage all elements are ultimately going to be merged which leads us to a simple solution that the complexity of this operation is simply the number of elements.

  The series merging operation is represented as: $\mathbf{f(n) = O(n)}$

- Complexity of Staging Process: At each stage, the number of arrays is halved since 2 arrays are being merged into 1. However, this also leads to the size of that final array doubling. For the first iteration, the algorithm will have $\frac{K}{2}$ or $\frac{K+1}{2}$ (if k is odd), the next iteration we'll have $\frac{k}{4}$ then $\frac{k}{8}$ and so on. This halving process continues until there's one sorted array. This exponential reduction can be expressed with a logarithmic relationship.

  The series relationship is represented as: $\mathbf{f(n) = \log_2(k)}$

- **Total Algorithm Asymptotic Complexity Calculation:** Evaluating the changes to the previous iteration of this algorithm we can put the steps of merging at each stage onto our binary tree analysis and see that the work per stage is $\mathbf{O(n)}$ and there are $\log_{\mathbf{2}}(\mathbf{k})$ stages so multiplying across at any level on the algorithm tree analysis we can get the asymptotic complexity of this improved algorithm. **By plotting this equation, we see that this improved algorithm will be more efficient with larger K values**. The graph below approximates the asymptotic complexity of the algorithm as K increases, and we see that as K increases the complexity levels out, supporting our theory of larger K optimizations. We can also conclude that larger input size n values can also contribute to a more optimized system. Further mathematical proofs demonstrate that algorithm performance gains will still be made even with larger input data sets n given logarithmic properties.

Asymptotic Algorithm Complexity: $\mathbf{f(n) = n \log_2(k)}$

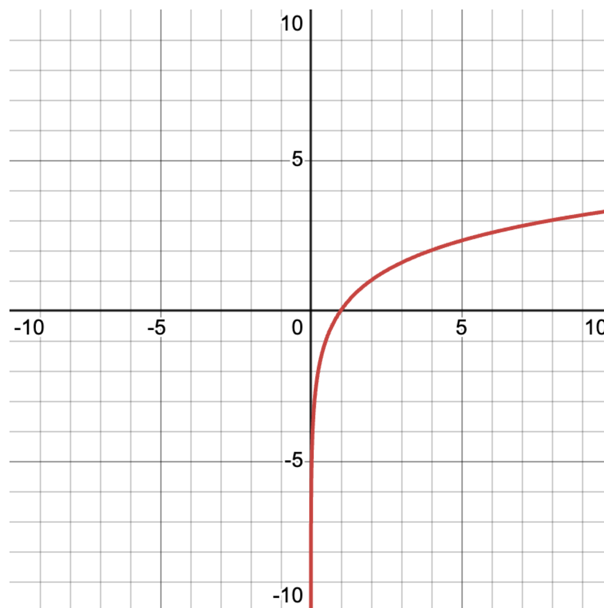


Figure 4: Mathematical Approximation of Optimized Algorithm Time Complexity

EXPERIMENTAL DATA ANALYSIS OPTIMIZED ALGORITHM

- Following our theoretical analysis, we conducted the exact same test suite that was applied to the original algorithm to compare the execution time to the optimized version. Several randomized data sets were created of various sizes from 100,000 to 500,000 and all results were plotted to see which variables had the greatest impacts on execution time.

  Caution: While algorithmic analysis is an excellent tool for evaluating algorithms, it does not always tell the entire story. Modern computers have such immense complexity that unrelated tasks running on hardware or exterior variables will significantly impact algorithm performance.

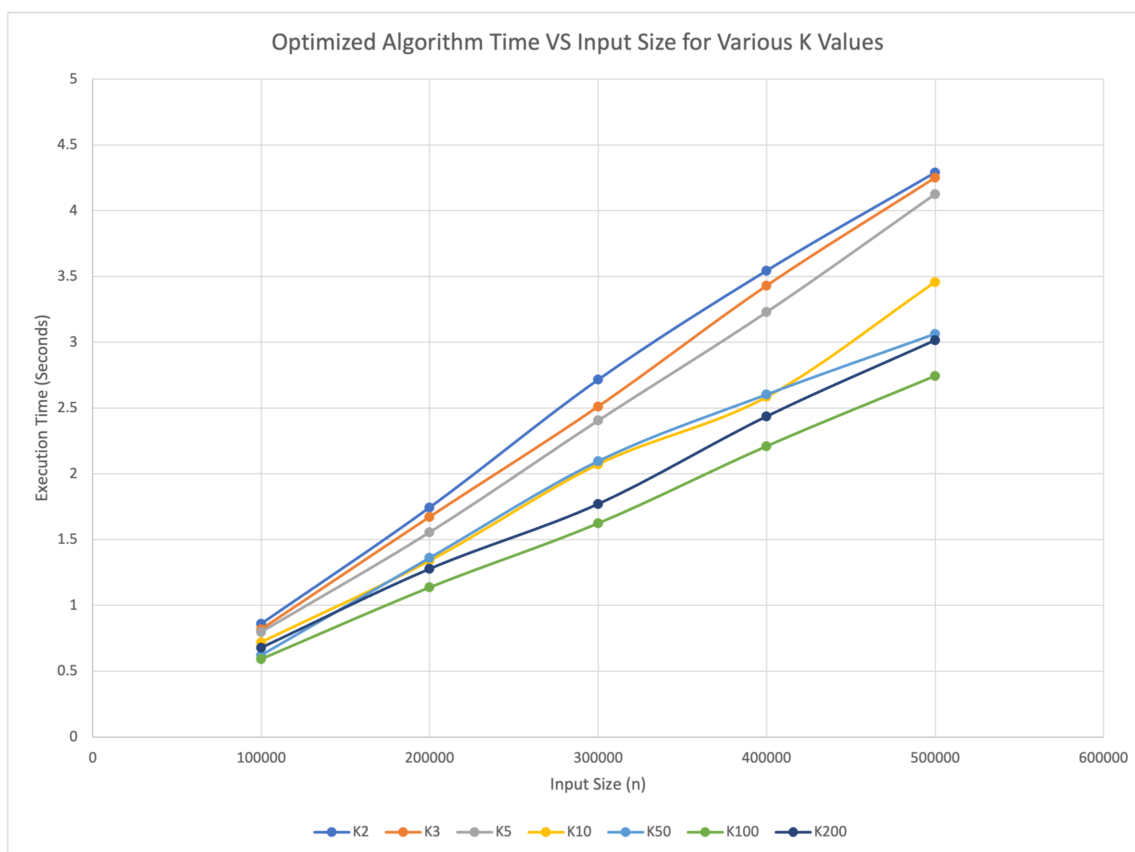


Figure 5: Time V.S. Input n for Various Values of K

- By plotting the algorithm's performance with respect to input size n, we identify algorithm is more optimized with larger values of K. This trend is consistent regardless of input size n as all lines effectively keep their respective position in the graph. This also demonstrates that the most optimal K value in every case regardless of input size n will be a larger K value. Both $K = 100$ and $K = 200$ have the most optimal execution time.

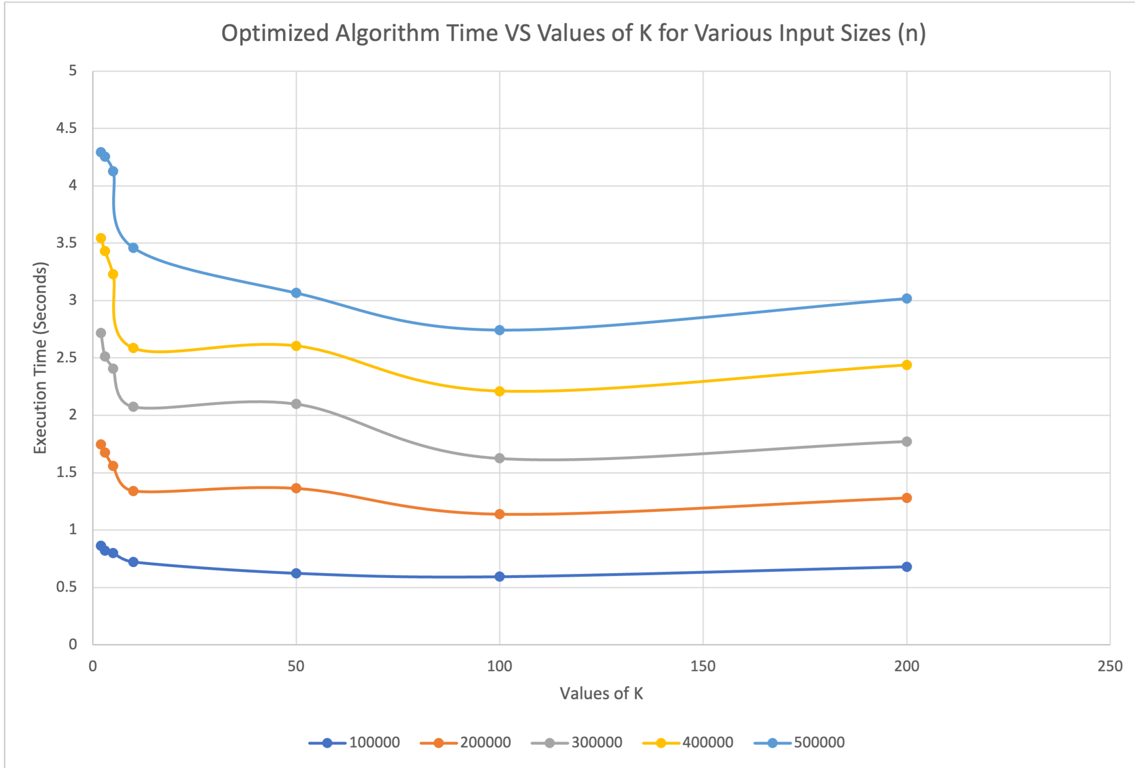


Figure 6: Time V.S. K Values for Various n Inputs

- Plotting the optimized algorithm execution time vs values of K reveals some very important information. In this improved algorithm, we see that larger values of K are more optimal for all data input n. The most optimal K value in this data set is $K = 100$ which has the lowest execution time for all input sizes n, however, all graphs level off likely indicating that the algorithm is most optimal with $K > 100$, we see a very small decrease in performance as we approach a $K = 200$ however we can assume that this trend is consistent for larger data sets.

Table 2: Execution times for the Optimized Algorithm (in seconds)

Input Size	K values						
	2	3	5	10	50	100	200
100000	0.861 734	0.819 442	0.799 205	0.721 404	0.623 608	0.593 887	0.680 163
200000	1.745 83	1.674 22	1.557 66	1.340 66	1.363 02	1.1381	1.279 59
300000	2.716 71	2.512 85	2.406 21	2.073 37	2.0979	1.624 41	1.772 54
400000	3.545 68	3.431 94	3.230 78	2.585 17	2.605 18	2.210 26	2.438 26
500000	4.291 05	4.252 79	4.127 91	3.458 15	3.065 57	2.743 42	3.0171

Note: K values denote the number of partitions.

Conclusion: After performing a full in-depth test suite on the optimized algorithm, our testing confirmed that there's a dramatic general increase in performance. One of the key features that make this optimized algorithm very robust is its efficiency as K increases to levels $K > 100$ which makes it suitable for extremely large data sets. Performance characterized by a logarithmic function allows the algorithm to achieve performance gains even as the input size n increases. Before performing an in-depth analysis, our mathematical equations represented the optimized algorithm as $f(n) = nlog_2(k)$ which is perfectly supported by our data collection. Our hypothesis stated that we expected the algorithm to be significantly optimized for larger K values as we could reach a horizontal asymptote where algorithm execution time would be at a maximum regardless of input size n. The logarithmic behavior of this algorithm will allow extremely large data sets to be processed offering minimal impact on cost-effectiveness deeming this optimized algorithm extremely efficient compared to the original algorithm.

SUMMARY

In conclusion, this paper analyzes the asymptotic complexity of the K Merge Sort Algorithm and compares results from a basic approach to a more advanced optimized algorithm involving an improved merge staged process. The basic algorithm was categorized by a linear time complexity while the advanced algorithm took advantage of logarithmic properties. The linearity proves the simplicity of this algorithm as time complexity is directly proportional to the size of the data input n. Our findings supported by our theoretical mathematical calculations concluded that the optimized algorithm performed significantly better given the logarithmic properties which allows performance gains even as the input size n continues increasing which is indicative of its increased complexity. Overall, this analysis highlights the importance of algorithm optimization when processing large data sets to ensure proper use of resource utilization. By understanding and leveraging the complexities of algorithmic design, we can develop more efficient systems capable of processing big data with greater speed, scalability, and stability.