We began our work on developing a new algorithm by first introducing a new algorithm called Merge Sort, which utilizes a Divide and Conquer approach to solve the sorting problem with a time complexity that is lower than quadratic time. We will show that Merge Sort has a logarithmic time complexity of $O(N\log(N))$. We will also analyze Merge Sort and its closest competitor to verify that Merge Sort performs fewer comparisons and has a lower time complexity than Insertion Sort.

Keywords
Divide and Conquer, Sorting, Merge Sort

1. INTRODUCTION
The task of sorting is a problem that commonly occurs in applications that retrieve information from a database or search through a file system. In both situations, the tables and file system can be very large, containing thousands of sources of data that need to be sorted quickly. For this reason, fast algorithms and solutions are desirable.

Insertion Sort, Selection Sort, and Bubble Sort are three existing algorithms that solve the task of sorting a list of comparable values. Of these three algorithms, Insertion Sort performs best, yet its performance diminishes rapidly as dataset sizes increase. This decrease in performance is due to its quadratic time complexity, which is an indicator that as a problem doubles in size, the time required to solve it will quadruple in size.

Given a list of comparable values, we desire an algorithm that can sort this list in polynomial time and has a time complexity lower than the quadratic time complexity of Insertion Sort.

We introduce a new algorithm called Merge Sort. The Merge Sort algorithm solves the problem of rearranging a list of linearly comparable values in less than quadratic time by using a Divide-and-Conquer approach to produce the output. This algorithm produces the rearranged list faster than other algorithms, such as Insertion Sort; Selection Sort; and Bubble Sort.

Section 2 describes the Selection, Insertion, and Bubble Sort algorithms. Section 3 details the Merge Sort algorithm being introduced. Section 4 proves that Merge Sort has a logarithmic time. Section 5 provides empirical evidence of Merge Sort’s superiority over Insertion Sort. Section 6 summarizes our results.

2. RELATED WORK
We began our work on developing a new algorithm by first looking at other solutions to the problem of reordering a list of comparable values that are given in an arbitrary order.

2.1 Selection Sort
Selection Sort is performed by selecting the smallest value of the list, then placing it at the beginning of the list [2]. This process is repeated once for every value of the list. Selection Sort is sound and easy to understand. It’s also very slow, and has a time complexity of $O(N^2)$ for both its worst and best case inputs due to the many comparisons it performs.

A single pass of Selection Sort can only guarantee that one element is sorted because only one element is placed into its correct position. In developing merge sort, we thought of an algorithm that would not only always sort every element in place correctly, but also sort faster on certain inputs.

2.2 Insertion Sort
Insertion Sort is another algorithm which solves the problem. Given a list of comparable values, search the list from the beginning. When a value is smaller than a previously viewed value, move the new value into its correct position and shift all of the previous values forward [1]. Like Selection Sort, Insertion Sort is also sound and has a time complexity of $O(N^2)$ for the worst case. Unlike Selection Sort, Insertion Sort has a time complexity of $O(N)$ for its best case input, and Insertion Sort will sort faster because it performs fewer comparisons [4].

Insertion Sort moves from the beginning of the list to the end of the list exactly once. During the process of sorting, if a value is out of place, it is moved backward into its correct position [1]. With a list that is mostly sorted, Insertion Sort will correctly reorder the list faster than Selection Sort can because it has fewer comparisons to perform and it only has to move a few elements a small number of times.

Insertion Sort is a faster solution than Selection Sort, but it’s still not fast enough for demanding applications. Our Merge Sort algorithm will perform faster than Insertion Sort.

2.3 Bubble Sort
Bubble Sort takes a different approach to solving our problem, which is why it is a good algorithm to include in our research. Bubble Sort’s approach to the problem is to make multiple passes through the algorithm, swapping any two consecutive values that are out of their proper order [3]. This process of full passes with order swapping has the effect of moving at least one value – the largest unsorted value – into its correct place.

The idea of sorting multiple items into their correct positions in each pass is a great idea, and it is used again in our merge sort algorithm. Also, due to this swapping process, it’s possible to stop sorting early; if any pass of Bubble Sort is completed without any swaps, then the list is fully sorted.

Due to the total number of swaps that Bubble Sort performs during each pass, this sort is actually slower than Insertion and Selection Sort. As you will see in the next section, one of the
improvements that our merge sort algorithm makes over Bubble Sort is that it will continue to sort quickly even with the worst possible inputs.

3. APPROACH
A list, L, is known to have a total of N linearly comparable elements in an arbitrary order. The task required is to rearrange the list’s elements in ascending order.

Merge Sort is known as a Divide and Conquer algorithm: given a task, divide it into parts; then solve the parts individually. There are two parts to the Merge Sort algorithm.

1. Divide the list into two separate lists
2. Merge the new lists together in order

First let’s look at Divide. The algorithm is simple: divide list L into left and right halves. This process is recursive, and does not stop until only one element remains in the list. A single-element list is already sorted.

Figure 1 (Divide)                             Figure 2 (Merge)

Merge does what its name implies: it merges two lists (left and right) into a single list (L) in ascending order. Merge also makes a claim: lists left and right are already sorted. Since left and right are already sorted in ascending order, the two lists can be merged by repeatedly removing the next lowest value in both lists and adding it to the result list L. This operation requires only a single comparison per value added to the result list, until one list is exhausted.

In Figure 1, the red arrows represent the left and right lists being compared, and the black arrows represent the result list L. Figure 2 shows the resulting sort that occurs when Merge is called on Line 5.

In the Merge algorithm, to merge the left and right into list L, we take the lowest value of either list that has not been placed in L, and place it at the end of L (Lines 8-10). Once either left or right is exhausted, L is filled with the remainder of the list that has not been merged (Lines 11-12).

![Figure 3](image)

Figure 3

The black arrows above the left and right lists represent i and j from the pseudo-code. In (a), these values are initially zero, so the first elements from each are compared. In (b) and (c), the lowest element from the previous step was appended to the end of L, and both i and j have increased. Finally, (d) represents Lines 11-12, where right is appended to L because left has been exhausted.

4. THEORETICAL EVALUATION

4.1 Criteria
The goal of both Merge Sort and Insertion Sort is to sort a list of comparable elements. The key operation in the execution of this goal is the comparison between list elements during the sorting process. This theoretical evaluation will measure the total number of comparisons between list elements in Merge Sort and in Bubble Sort according to each algorithm’s worst and best case input.

4.2 Merge Sort
In merge sort, the comparisons occur during the merging step, when two sorted lists are combined to output a single sorted list. During the merging step, the first available element of each list is compared and the lower value is appended to the output list. When either list runs out of values, the remaining elements of the opposing list are appended to the output list.

4.2.1 Worst Case
The worst case scenario for Merge Sort is when, during every merge step, exactly one value remains in the opposing list; in other words, no comparisons were skipped. This situation occurs when the two largest value in a merge step are contained in opposing lists. When this situation occurs, Merge Sort must continue comparing list elements in each of the opposing lists until the two largest values are compared.
The complexity of worst-case Merge Sort is:

$$T(N) = 2T(N/2) + N - 1 \quad (2.1)$$
$$T(N) = 2[2T(N/4) + N/2 - 1] + N - 1 \quad (2.2)$$
$$T(N) = 4[2T(N/8) + N/4 - 1] + 2N - 3 \quad (2.3)$$
$$T(N) = 8T(N/8) + N + N + N - 4 - 2 - 1 \quad (2.4)$$

Eq. 2.1 is the recurrence relation for Merge Sort. T(N) refers to the total number of comparisons between list elements in Merge Sort when we are sorting the entire list of N elements. The divide stage performs Merge Sort on two halves of the list, which is what $2^r T(N/2)$ refers to. The final part of the Eq., N-1, refers to the total comparisons in the merge step that returns the sorted list of N elements.

Eq. 2.1 describes the number of comparisons that occur in a merge sort, which is a recursive procedure. Since the method is recursive, we will not be able to count every comparison that occurs by only looking at a single call to Merge. Instead, we need to unroll the recursion and count the total number of comparisons.

Equations 2.2-2.3 perform this action of unrolling the recursion by performing substitution. We know what the value of T(N) is from 2.1, and by substitution we know the value of T(N/2). Eq. 2.4 is just a clearer view of Eq. 2.3. At this point we are in the third recursive call of Merge Sort, and a pattern has become clear enough to produce Eq. 2.5 below:

$$T(N) = 2^k T(N/2^k) + kN - (2^k - 1) \quad (2.5)$$

In Eq. 2.5, a new variable called “k” is introduced. This variable represents the depth of the recursion. When the sort is recursively dividing the input list, the recursion stops when the list contains a single element. A single element list is already sorted.

$$T(1) = 0 \quad (2.6)$$
$$2^k = N \quad (2.7)$$
$$k = \log_2 N \quad (2.8)$$
$$T(N) = N \log_2 N - N + 1 \quad (2.9)$$

By substituting Eq. 2.6 through Eq. 2.8 into Eq. 2.5, we eliminate the k term and reduce the recurrence relation to produce the complexity for merge sort, Eq. 2.9. Thus we’ve shown that Merge Sort has $O(N \log(N))$ complexity with the worst possible input.

### 4.2.2 Best Case

Merge sort’s best case is when the largest element of one sorted sub-list is smaller than the first element of its opposing sub-list, for every merge step that occurs. Only one element from the opposing list is compared, which reduces the number of comparisons in each merge step to N/2.
To get a better analysis of Merge Sort and Insertion Sort, the procedure, outlined below, was repeated ten times, and the time measurements were averaged to produce the results in this paper.

The procedure is as follows:
1. Store the first 20000 records in a list.
2. Choose any 500 consecutive records to sort.
3. Sort these 500 records separately using Merge Sort and Insertion Sort.
4. Record the completion time (the moment a sort is called until the moment that sort complete) of each sort.
5. Repeat steps 2-4, incrementing the number of consecutive records by 500.

5.3 Results

Figure 4 below shows average of the results of the experiment. In the figure, the completion time for Merge Sort is in red and the completion time for Insertion Sort is in blue.

![Figure 4](image-url)

From Figure 4 we clearly see that, as the number of sorted records increases, the time Insertion Sort takes to sort will also increase at a very rapid rate. In fact, performing a curve fitting on the graph produces a line that fits the data points very well. Merge sort, on the other hand, takes so much less time that it is unclear from Figure 4 whether the completion time is increasing. Linear regression was performed on the data to verify that the time was increasing – it is in fact linearly increasing as the size of the data increases.

Table 1 shows the t-test performed on the 10 experimental samples. Our critical value was 2.02, and we exceeded it, meaning our Merge Sort algorithm is clearly faster at sorting than Insertion Sort.

From the results we have shown that Merge Sort sorts faster than Insertion Sort for data sets of any significant size, and will continue to do so no matter how much the data size is increased. Also, from the same results, the difference between the two sorts for list sizes smaller than 2000 is so insignificantly small that using Insertion Sort would not give any short term speed increase.

<table>
<thead>
<tr>
<th>Variable 1</th>
<th>Variable 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>3379.42</td>
</tr>
<tr>
<td>Variance</td>
<td>9295934.83</td>
</tr>
<tr>
<td>t Stat</td>
<td>6.974</td>
</tr>
<tr>
<td>P(T&lt;=t) two-tail</td>
<td>1.97</td>
</tr>
<tr>
<td>t Critical two-tail</td>
<td>1.83</td>
</tr>
</tbody>
</table>

6. CONCLUSION

In this paper we introduced the Merge Sort algorithm which takes a Divide-and-Conquer approach to rearrange a list of linearly comparable values into ascending order. We have shown that Merge Sort is a faster sorting algorithm than Insertion Sort, both theoretically and empirically. We have also shown that Merge Sort has a best- and worst-case time complexity of O(Nlog₂(N)).

Despite its time complexity, Merge Sort’s performance degrades when the input size decreases during the Split phase. This is due to the overhead of creating, working with, and destroying many small lists. One possible improvement to the Merge Sort algorithm may include using a “faster” sorting algorithm when list sizes decrease below a certain size to prevent the speed decrease caused by the overhead of data storage. Such future work would also include determining whether such a size should be an input parameter, and whether it should be a discrete value or a percentage of the input size.

7. REFERENCES