In the modern day, computation has increased magnitudes in complexity from the 1970s and 1980s personal computers to current smartphones, tablets, desktops, and supercomputers. Nowadays, it seems like almost everybody in the U.S. has a smart device, and thus, quick and reliable access to information is a requirement in order to keep up with rapidly-evolving technology. Included in this domain is the problem of sorting data. Whether one searches for a new home, desires a funny video to watch on their favorite media site, decides to spend time on social media, or covets a synonym for a paper they are writing, they can sort by price, views, most recent post, and words from most common to least common respectively. Without a doubt, sorting plays a significant role in the average day of a human whether they know it not, which establishes this problem as more important than ever before. In order to effectively sort information nowadays, there are several characteristics one must consider, such as whether or not it will consume additional information to act as a temporary buffer, if it will be stable or unstable, and perhaps most importantly, its performance. For the algorithm proposed in this paper, it plans to solve the problem of performance by outperforming all other comparison-based sorting algorithms in the average and worst case runtimes, in addition to being stable, at the cost of requiring additional memory from the user. This contrasts with other comparison-based algorithms that may not need additional memory usage, but lack in performance. Or maybe they do not need additional memory usage and they have similar performance, but they are unstable. Either way, this algorithm aims to solve key problems that other sorting algorithms currently cannot keep up with, which in turn will revolutionize the speed and efficiency of how businesses and developers manage their data and present it to their consumers.

In order to achieve the level of performance and stability to outperform other comparison-based sorts, this algorithm will utilize a divide-and-conquer approach by splitting the data into chunks, and later **merging** them together, hence the algorithm will be named mergesort. An example scenario could occur with data contained in a homogenous, constant-access data structure, such as an array. In mergesort, the data will be recursively split in half into as many subarrays as necessary until only subarrays of length one exist. This will require ceil(log2N) levels of splitting, where N is the number of items to be sorted and ceil(x) rounds x up to the nearest integer value; for example, an array of length eight will be split into two arrays of length four, which will be split into four arrays of length two, and finally eight arrays of length one. As one can see, three levels of splits were performed in the process (log28 = 3). The reason the result must be rounded up is in the case of a dataset with length N that is not a power of two. For example, if there are twelve items to be sorted, they would be split into two arrays of length six, then four arrays of length three, eight arrays where four of them are of length one and four are length two (which must be split even further), and finally twelve subarrays of length one. In this case, log212 = 3.58 levels of splits occurred, which is falsified by the above scenario, so it must be rounded up using the ceiling function, resulting in ceil(log212) = 4 levels of splitting. Once the splitting is finished, there are N subarrays of length one. Make note that additional memory is not necessary at this point. However, the next step, merging, will require the use of the N supplementary memory. With these subarrays of length one, start at the first subarray, have a counter variable start at the beginning logical index of it and its sibling (not every subarray of length one will have a sibling, such as in the example with twelve items, because those length one subarrays will be merged back with the other two corresponding items in its sibling subarray to generate a subarray of length three), copying the lower value at each counter index into the next available slot of the additional memory, and incrementing the variable of that specific subarray until all items of the subarrays are copied into the temporary buffer. If the two values are equal, keep their order by always copying the value in the left subarray over first; this is where the stability of the algorithm originates. When two subarrays are merged together in the extra, reserved memory, the values are then copied back to their respective indices in the original array. Once this action is performed, they form a larger, sorted subarray of length K+M where K is the length of subarray one and M is the length of subarray two. This process is repeated until there are no more subarrays, and rather just one array of size N, which is the length of the original array, indicating that everything has been sorted. In short, this mergesort algorithm recursively splits arrays in half until subarrays of length one are formed, which is where they are sorted and sibling subarrays are then finally merged together.

As already mentioned, because equivalent values maintain their position when sorting data, mergesort is a stable sorting algorithm. However, the other significant issue that must be discussed is its performance. First, the splitting of the initial array into subarrays of length one requires ceil(log2N) levels, which turns out to visually look like a binary tree. These splits can be performed in O(1) time due to their recursive nature, and when dividing, one can logically just keep track of the indices of each subarray (utilizing parameters of the recursive function) instead of allocating new space for each divide. Then, when it comes to merging, the algorithm merges all N items (spread across numerous subarrays), thus spending N time merging each level of the “binary tree” together and pushing it back up to its parent state. But really, since mergesort is not in-place and the sorted subarrays are copied back into their original subarray locations, that requires another N copies, so each level really requires a runtime of 2N for the merge, which is O(N) asymptotically. Therefore, since a runtime of 2N occurs for each level of merging ceil(log2N) times, the product is an overall runtime of 2N\*ceil(log2N), or 2N\*log2N, which is O(Nlog2N) asymptotically.

An example of mergesort in the process would be on an array containing the values (8, 6, 3, 0, 3, 4, 7, 5). First, split the array into two subarrays containing (8, 6, 3, 0) and (3, 4, 7, 5) respectively. Then, split those into (8, 6), (3, 0), (3, 4), and (7, 5). Finally, split them into (8), (6), (3), (0), (3), (4), (7), and (5). For clarification, in the second split, (8, 6) and (3, 0) would be sibling subarrays and so would (3, 4) and (7, 5). Then for the final split, (8) and (6) are siblings, and the pattern is completed for the remaining elements. Now, for the split, sibling subarrays (8) and (6) are compared, 6 is less than 8, both are copied into the newly allocated array for merging, with the order (6, 8), which is the new subarray; the values of this new subarray are copied back into their original indices in the original array. This is repeated for the remaining one element subarrays, which leaves the algorithm with four sorted subarrays of length two, (6, 8), (0, 3), (3, 4), and (5, 7). In this step, (6, 8) and (0, 3) are sibling subarrays, so first 6 and 0 are compared; 0 is lower so it is placed into the merging subarray first and the counter variable keeping track of that subarray’s index is increased, then 6 and 3 are compared; 3 is lower so it is placed next to 0, and 6 and 8 are the only ones left, which are already sorted, so they are placed at the end. This is repeated for the other sibling subarrays, which converts into (0, 3, 6, 8) and (3, 4, 5, 7). The same steps as above are repeated, which finalizes the sorted array as (0, 3, 3, 4, 5, 6, 7, 8) where the first 3 is copied from the first subarray, and the second 3 follows it; this does not immediately show as a major problem, but when comparing objects or structures of data that contain more than integer values, maintaining their order matters in order to keep the sort stable.

As one can see, mergesort can prove effective when performance and stability are issues. For example, when searching for products to buy online, the option to sort by ascending price exists, but what if one wants to sort by rating out of five stars as well? Well, sorting by price and then grouping the items by one-star, two-star, three-star, four-star, and five-star maintains their respective order so one can check the cheapest prices for a four-star product. This not only comes in handy because the algorithm is the most efficient out of all comparison-based sorts based on runtime, but also because stability will maintain the respective order of items. Compare this to other sorting algorithms, such as insertion sort and quicksort. Insertion sort starts with a sorted subarray of length one, which is the first index in the array. Then, for the next index after the sorted subarray, insert that value into the sorted subarray by shifting the necessary values through a series of swaps. Repeat this for all values of the array to get a sorted array. This algorithm is stable by nature and does not require extra memory because it only performs shifting of values instead of temporarily storing values. For the best case, the array is already sorted and no shifting is required, so it is Ω(N), while in the worst case, the array is sorted in reverse order, so it is O(N2), and the average case ends up being Θ(N2). As for quicksort, a pivot, or random value in the array, is chosen. Then, all the values in the current array less than or equal to the pivot are placed to the left of it, while the values greater than the pivot are placed to the right; this process is referred to as partitioning and both sides of the pivot then become subarrays. This is recursively repeated for each subarray, as it is hopefully being split in half every time (differs by how well the pivot is chosen), similar to the splits of mergesort. Due to equivalent values being shifted around, disregarding their respective order to the pivot, quicksort is unstable. However, no additional memory is necessary to temporarily store any values. And the runtime is fairly comparable to mergesort as well: Ω(Nlog2N) when the pivot is the exact middle of the subarray every time, Θ(Nlog2N) in the average case, and O(N2) in the worst case when the pivot is the min or max of each subarray every time, so only one value is truly sorted per split and each subarray is of max size. So, in the best case, insertion sort wins in runtime, but overall, mergesort will win in the average and worst cases, which typically are more likely to occur. In addition, mergesort has stability, similar to insertion sort, which quicksort does not have. Lastly, mergesort is the only one of the three algorithms that requires additional memory; this may or may not be a problem depending on whether the system has space available for the sort and whether or not the tradeoffs between an increase in performance for an increase in space usage is worth it. Mergesort offers some of the best overall comparison-based sorting capabilities if space is not an issue.

This algorithm simply outperforms its competitors in almost every case. With its stability, it enjoys the comfort of maintain the order of necessary pieces of data, such as sorting by price and then by rating for a product, as well as the superior runtime performance compared to every other comparison-based sorting algorithm in the average and worst cases, only at the cost of using twice the memory of the initial data. In future work, more research will be focused on implementing an in-place mergesort algorithm, which will completely rid of the hassle of the memory overhead, further advancing its dominance in the domain of sorting. Another direction to take mergesort would be to look into an iterative approach rather than a recursive approach, which would remove the recursive calls placed on the stack during the splitting phase; insertion sort is iterative, which is a competitor. Lastly, improving the best case of the algorithm through preliminarily traversing the data and determining how unsorted it is, forming a mergesort-insertion sort hybrid may improve the runtime, thus allowing it to outperform all comparison-based sorting algorithms in the best, average, and worst cases instead of just the average and worst scenarios. In the end, mergesort is a quick, safe (stable), and easy-to-implement divide-and-conquer sorting algorithm that will make its mark in the realm of sorting. In the future, hopefully with the several improvements mentioned above, this sorting algorithm has the potential to best any others in space, performance, and stability.