In the modern day, computation has increased magnitudes in complexity from the 1970s and 1980s personal computers to 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 some part in the average day of a human whether they know it not, which establishes this problem as more significant 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 efficiency. For the solution proposed, it plans to solve the problem of efficiency by outperforming all other comparison-based sorting algorithms in the best, worst, and average case, 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 problems currently cannot keep up with, which in turn will revolutionize the speed 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 name 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 nothing but subarrays of length 1 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 splits were performed in the process (log28 = 3). The reason the result must be rounded up is in the case of 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 splits occurred, which is proven false by the above scenario, so it must be rounded up using the ceiling function, so ceil(log212) = 4 splits. At this point, 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 beginning, have a counter variable start at the logical index of each subarray (not every subarray of length one will have a neighbor, such as in the example with twelve items, because it 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 additional memory, and incrementing the variable of that specific subarray. 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.