Sorting algorithms occupy an important position in the field of computer science, and their performance directly affects the efficiency of many applications. Among many sorting algorithms, bubble sort and merge sort are the two most representative algorithms. This report will dive into the performance of these two sorting algorithms for different input sizes and analyze them in terms of code complexity. The first is the code complexity, which in the case of BubbleSort is a simple sorting algorithm based on comparison and swap. The rationale is to start at the beginning of the list, compare adjacent elements, swap them if they are out of order, and repeat the process until the entire list is sorted. In the code implementation, usually adopts a two-layer nested loop structure. The outer loop controls the number of times the list is traversed, and the inner loop compares and swaps adjacent elements. In terms of time complexity, BubbleSort has O (n^2) time complexity in the worst case and average case. This means that the execution time of the algorithm will grow quadratically as the input size n increases. In the best case, when the list is already sorted, BubbleSort only needs to traverse the list once to determine its order in O (n) time, but this is rarely the case in practice. From the perspective of space complexity, BubbleSort is a sort algorithm in place. It only needs a small amount of extra space to exchange elements, and its space complexity is O (1). MergeSort uses a divide and conquer strategy for sorting. It first splits the input list into two sublists, recursively sorts each sublist, and then merges the two sorted sublists into an ordered list. In code, it involves

recursive function calls and merging operations. The time complexity of MergeSort is O (n log n) in all cases. This time complexity makes MergeSort perform well when dealing with large-scale data. Although its code implementation is more complex than BubbleSort's because of the recursion and merge operations involved, its efficiency is significant for large data sets. In terms of space complexity, MergeSort is O (n) because it requires extra space to store temporary sublists and elements during merging. Both BubbleSort and MergeSort have relatively short execution times for small input sizes, such as the text sort10.txt given here, of 10 elements. This is because for small values of n, the actual execution time of even BubbleSort with high time complexity is not too long. However, while BubbleSort is able to sort faster in this case, it still requires multiple comparisons and swaps, while MergeSort may have more overhead for small data sizes due to its divide-and-conquer strategy and merge operations, but the overall difference is not very significant. For medium input sizes, such as the text sort100.txt, BubbleSort's performance starts to degrade significantly as the input size increases to 100 elements. Its O (n^2) time complexity leads to a rapid increase in execution time. MergeSort, on the other hand, is O (n log n), which has a more gradual increase in execution time. With this medium size of data, MergeSort starts to show the benefits of sorting the data more efficiently, reducing execution time. For large input sizes, such as the text sort10000.txt the execution time of BubbleSort rises sharply and becomes very long as the

input size reaches 10,000 elements. This is because n^2 grows much faster than n log n as n gets larger. While MergeSort can maintain a relatively reasonable execution time when dealing with such a large amount of data, which fully demonstrates the efficiency of mergesort in dealing with such a large amount of data.

In summary, the performance of BubbleSort and MergeSort differ significantly for different input sizes. Because of its simple code structure and high time complexity, BubbleSort can sort quickly when dealing with small-scale data, but its efficiency is extremely low when dealing with large-scale data. Although MergeSort is more complex to implement and requires extra space, its O (n log n) time complexity makes it significantly more efficient when dealing with large data sets. In practice, the selection of an appropriate sorting algorithm should consider factors such as data size, available system resources, and requirements for time and space efficiency.