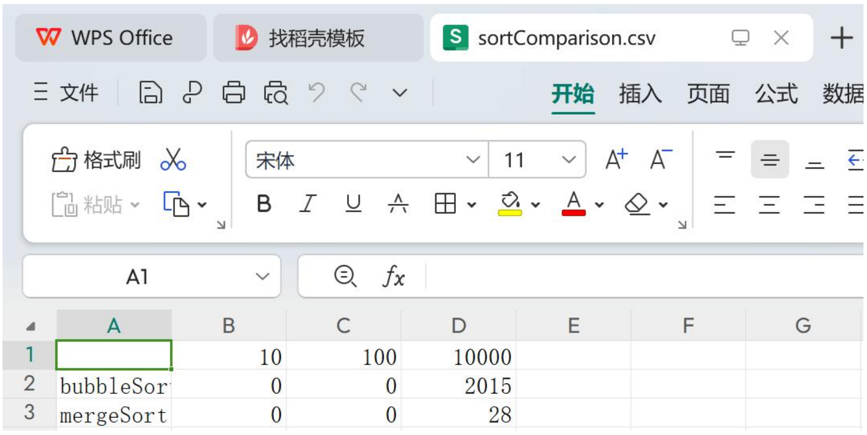
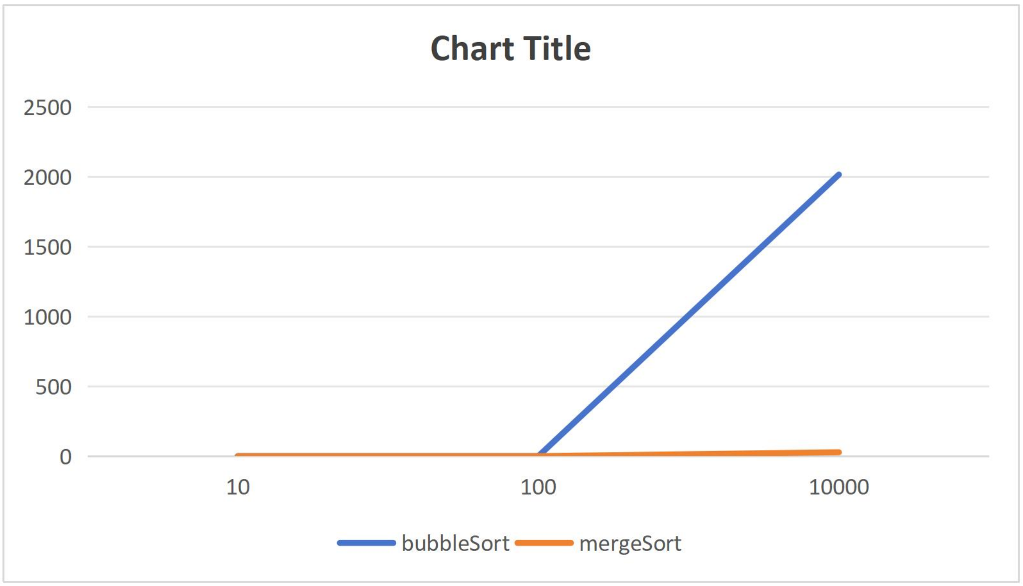
BubbleSort and MergeSort Complexity Reflection



	A	B	C	D
1		10	100	10000
2	bubbleSor	0	0	2015
3	mergeSort	0	0	28



The performance of sorting algorithms is often discussed in terms of their time complexity, which provides a theoretical upper bound on the number of operations required to sort a list of items. In the case of the sortComparison function, we compared the performance of two classic sorting algorithms: BubbleSort and MergeSort, using files sort10.txt, sort100.txt, and sort10000.txt to represent different input sizes.

First and foremost,BubbleSort has a worst-case and average time complexity of O(n^2), where n is the number of items to be sorted. This quadratic growth means that the algorithm's performance degrades rapidly as the input size increases. The comparison-based sorting works by repeatedly swapping adjacent elements if they are in the wrong order, requiring multiple passes through the list until no more swaps are needed. For the sort10.txt file, BubbleSort performs relatively well due to the small input size, but as the input size grows to sort100.txt and sort10000.txt, the number of comparisons and swaps increases significantly, leading to a substantial increase in execution time.On the other hand, MergeSort exhibits a more favorable time complexity of O(n log n) .This algorithm works by recursively splitting the list in half, sorting each half, and then merging the sorted halves. The result is a more consistent performance across different input sizes. As seen in the sortComparison results, MergeSort's execution time increases at a much slower rate compared to BubbleSort as the input size grows from sort10.txt to sort10000.txt. This is because the logarithmic factor in the complexity means that the number of operations grows much more gradually with each increase in input size.

In conclusion, from the complexity discussion, we can find that BubbleSort is less efficient for larger datasets due to its quadratic time complexity, while MergeSort's performance remains relatively stable for varying input sizes, demonstrating the importance of algorithm selection based on the context and expected input size. The results from the sortComparison function align with these theoretical complexities, highlighting the practical implications of algorithmic efficiency in real-world applications.