# VE281 Project1

Li Mingyu 518370910071

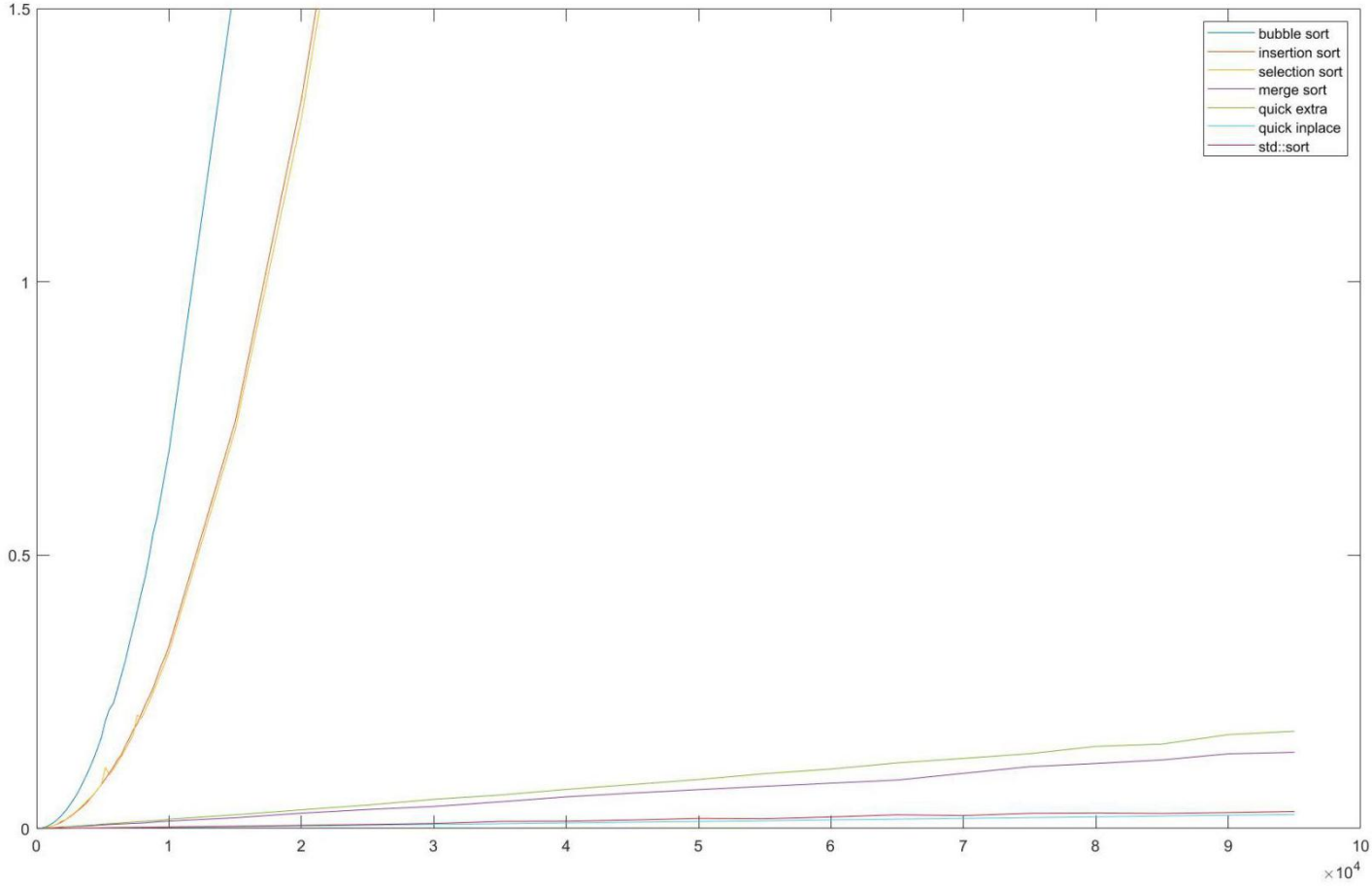


Figure 1

Set the threshold time to 1.5s, and label the 7 curves in the figure. The largest **n** is 95001.

By comparing the 7 curves, we find that bubble sort, insertion sort and selection sort all increase greatly and in a similar way as the vector size **n** grows. Among them, bubble sort has the largest increasing rate, while insertion sort and selection sort perform quite similarly in most cases.

This is because no matter how the array is ordered, bubble sort will always compare all the elements from the last one to the current i, and swap as long as the compare function is true. So it will always have n(n-1)/2 comparisons, almost same in best and worst cases. The time complexity is always $O(n^2)$. The insertion sort may have a better performance than bubble sort if the original array is well sorted and it has less comparisons needed, so it depends greatly on the array order. The selection sort has similar comparisons as bubble sort, but it only needs 1 swap after each scan through the array, so it's also better than bubble sort in large case.

The other 4 sort algorithms have much better performance. Comparing the 2 quick sort algorithms, the in-place quick sort is better than the other one. This is because the extra space to store the elements that are smaller and larger than the pivot is created and the elements will be stored into the extra space

then copied back. This process is time-consuming. But from the graph, the in-place quick sort is the best algorithm.

Merge sort is not in-place. It also need extra space to merge and copy back. So it performs similar to the not in-place quick sort.

The *std::sort()* algorithm also performs good, and it's very close to the in-place quick sort. The *sort()* algorithm uses optimized quick sort first. When the array length is smaller enough, change to insertions sort because insertion sort performs well in small arrays. So it has almost same time cost as quick sort and sort very fast, which is time complexity O(N·logN). *std::sort()* and in-place quick sort are the two most efficient algorithms when **n** is very large.

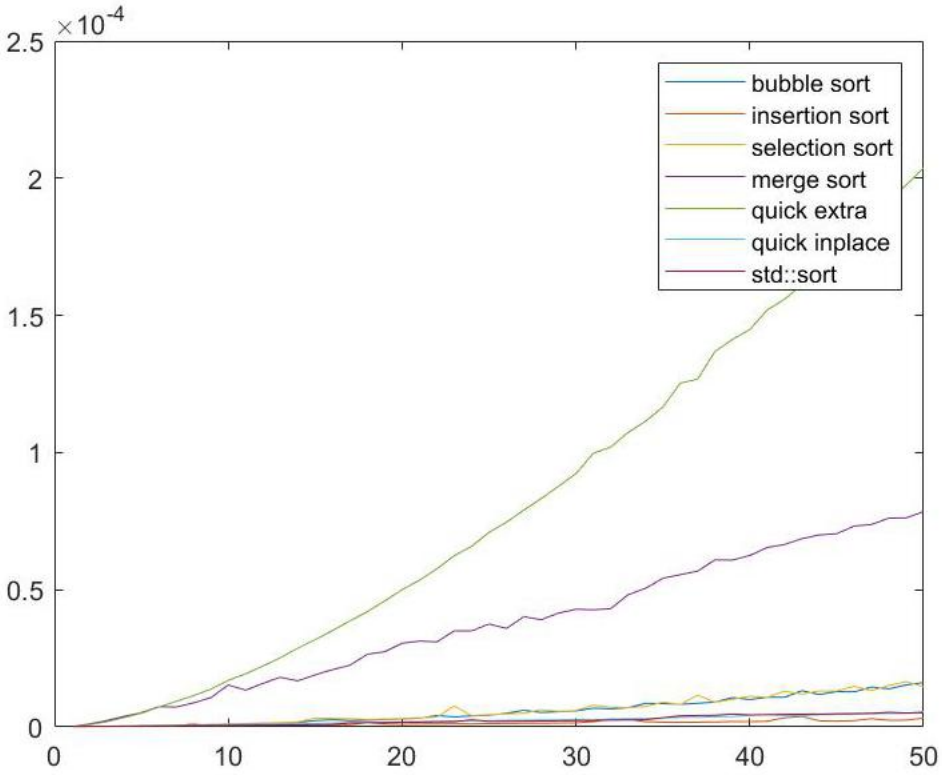


Figure 2

Figure 2 is the graph for small arrays with **n**=1-50. Each of the case is computed 30 times and get the average running time. From this graph, we find that quick sort with extra space takes longest time, followed by merge sort. But they used to sort quickly when dealing with larger size, this is because they always need to create new space in intermediate and copy back, which will cost a lot of time compared with small array sort. So their performance is stable and does not depend too much on the array size.

Insertion sort, in-place quick sort and *std::sort* perform best and similarly in small arrays. When the array size is small, std::sort use insertion sort algorithm so they are similar. Bubble sort and selection sort are relatively good in this condition, and since their principle are the same, they do not differ much in performance. To be more detailed, the most efficient algorithm is insertion sort when the array is small. The in-place quick sort and *std::sort()* are almost the same because *std::sort()* is a kind of quick sort.