VE281 Project 1

Comparison of Sorting Algorithms

In this project we test and compare six sorting algorithms, along with the std::sort() function. The six sorting algorithms are: Bubble sort, insertion sort, selection sort $\mathrm{O}(n^2)$; merge sort, in place quick sort, and extra space quick sort $\mathrm{O}(nlogn)$. We first complete the six function and then verify the time complexity. The test data are generated from pseudo-random number generation library, `*default_random_engine*`, and use `*steady_clock*` from chrono library to do the timing. The timing results (microsecond) are the average of 50 tests with different data sets. the average listed in the following table.

Data amount	**50**	**500**	**2000**	**5000**	**7500**	**10000**	**20000**
Bubble	16	1642	25600	161646	358378	468969	2551644
Selection	11	914	14244	94533	198156	350079	1570101
Insertion	7	726	11546	72085	160461	280369	1157445
Merge	14	195	895	2270	3422	4673	8995
Quick sort inplace	23	446	2654	9593	19193	31901	65601
Quick sort extra	34	404	1611	4637	6503	8885	18280
std::sort( )	6	92	449	1213	1944	2775	5718

Table 1. Time comsumption of 7 algorithms under different data size

For the first three algorithms, it's obvious that the time complexity is $\mathrm{O}(n^2)$, since the data size goes up 10 times, the time consumption goes up around a hundred times. For merge sort and quick sort, it's not that obvious. But if we try timing t_1 with $\frac{n_2 logn_2}{n_1 logn_1}$, we'll get a value close to t_2 (t_1, t_2 are two arbitrary data points in that three rows). Thus, it's safe to conclude that these three is very likely to follow the time complexity $\mathrm{O}(nlogn)$.

To visualize the results, we plot the performance of the 7 algorithms in a single graph as follows. It's obvious that the first three algorithms' time consumption growths much faster than the latter. But when the data size is small, algorithms with large time complexity may still runs faster.

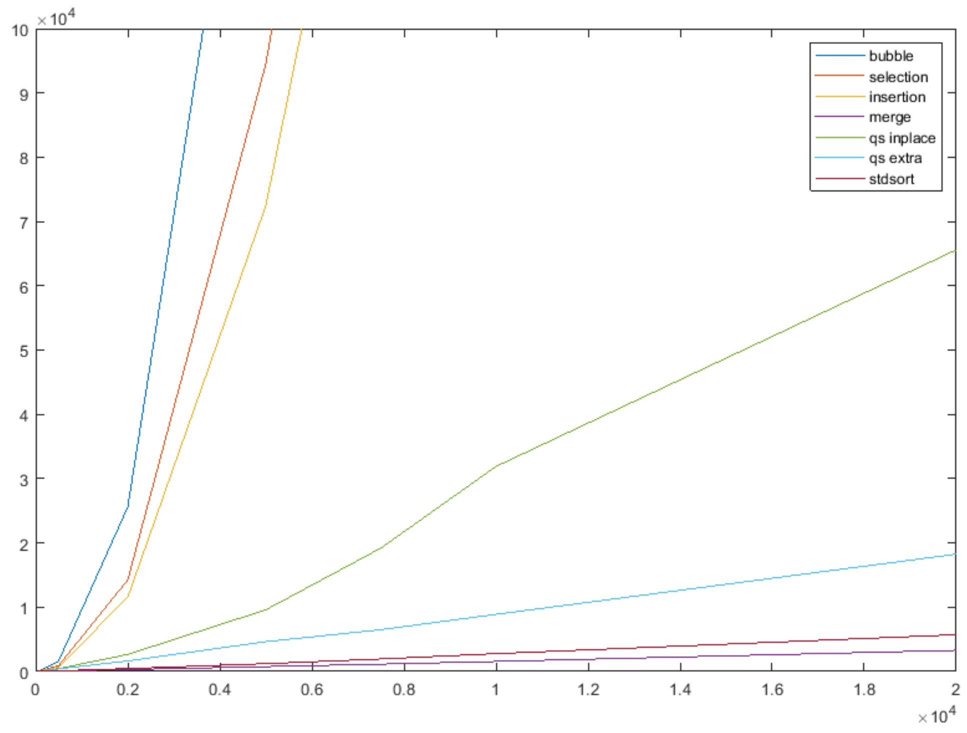

Fig 1. Time complexity of 7 algorithms

However, the six function written by myself is far more time-consuming although theoretically they have the same time complexity. There must be many details that can be optimized.

It's worth mentioning that std::sort( ) have good performance in both small data size and large data size. In comparison, quick sort may do worse than algorithms with $\mathrm{O}(n^2)$. Moreover, the worst case for quick sort is $\mathrm{O}(n^2)$, but for std::sort( ), it's always $\mathrm{O}(nlogn)$.

Let's see the code of std::sort( ) and try to understand the secret.

```
template <class RandomAccessIterator>
inline void sort(RandomAccessIterator first, RandomAccessIterator last) {
    if (first != last) {
        __introsort_loop(first, last, value_type(first), __lg(last - first) *
2);
        __final_insertion_sort(first, last);
    }
}
```

The first part `__introsort_loop` works like quick sort when the data size is large. But it uses special recursion that reduces the number of recursive calls, and special partition. And if the recursive depth is worsening, it will change to heap sort, whose time complexity is always $\mathrm{O}(nlogn)$ no matter best or worst. The second part `__final_insertion_sort` works when the unsorted part reaches a small enough value, then it will do insertion sort with sophisticated branch.

To conclude, we should tailor the sorting algorithms according to our requirement. Sorting seems simple, but it can also be a masterpiece, like std::sort( ).