Cheng Li

Project

Comparative performance evaluation of sorting algorithm

1. Project Overview

Sorting algorithm is a fundamental operation in different practical fields of computer sciences, database applications, networks and artificial intelligence. Most practical applications in computer programming will require the output to be arranged in a sequential order since handling the elements in a certain order is more efficient than handling randomize elements. Efficient sorting is important for optimizing the use of other algorithms, such as search and merge algorithms, that require sorted lists to work correctly. Information rapid growth in our world leads to the fact that the sorting problem has attracted a great deal of research. A lot of sorting algorithms has been developed to enhance the performance in terms of computational complexity, time complexity, memory space, and stability. Therefore, most programmers are faced with the challenges in chosen which algorithm would be best used in developing their software. We make a comparison research with the aim to come up with the most efficient sorting algorithm. When comparing various sorting algorithms, the factors mentioned above must be taken in consideration.

The sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. In this project, a comparative performance evaluation of four comparison based sorting algorithms: Insertion-Sort, Merge-Sort, combined Merge-Insertion-Sort and Quick-Sort, based on vectors of integers of increasing sizes, will be presented by the performance factors. Each of them has a different mechanism to reorder. All algorithms were implemented in C++ programming languages and tested for the random sequence input of length 100 through 50000.

The ultimate goal of this study is to proposed a methodology for the users to select an efficient sorting algorithm. Finally, the reader with a particular problem in mind can choose the best sorting algorithm.

2. Methodology

Insertion-Sort

Insertion sort algorithm considers the elements once at a time, inserting each in its suitable place among those already considered while keeping them sorted. It is an example of an incremental algorithm, because it builds the sorted sequence one number at a time. We build up a complicated structure on *n* items by first building it on *n* - 1 items and then making the necessary changes to fix things in adding the last item. It has $O(n^2)$ time complexity, It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort. However, insertion sort provides several advantages: simple implementation, save memory and efficient for small data sets and mostly sorted list.

Merge-Sort

Merge sort is a divide and conquer algorithm. Merging is the process of combining two or more sorted arrays into a third sorted array. This is an ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets, then sort the sub lists recursively by re-applying merge sort. At last, the combining operation merges two sorted sub-lists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime. First, a small list will take fewer steps to sort than a large list. Second, fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. You only traverse each list once if they're already sorted.

As a stable and fast recursive sorting, this algorithm parallelizes better and works well for very large list. Merge sort is often the best choice for sorting a linked list. It has an $O(n \log n)$ time complexity.

Merge-Insertion Sort

Since for small number of values insertion sort runs faster than merge sort, insertion sort can be used to optimize merge sort. The basic idea is applying insertion sort on sub-lists obtained in merge sort and merge the sorted lists. Suppose there are n elements. We divide it into n/x sublists, each contains x elements.

Complexity of Merge Sort

- Total work done by merge sort in copying n elements from each level to upper level is $O(n)$.
- Number of levels = $O\left(\log \frac{n}{x}\right)$
- Complexity of Merge sort: $O\left(n \log \frac{n}{x}\right)$

Complexity of Insertion Sort

Since we are sorting n/x lists of x elements each using insertion sort, complexity contributed by insertion sort is n/x multiple of complexity of Insertion Sort.

Total Time Complexity

Total Time Complexity = Complexity of Insertion Sort + Complexity of Merge Sort.
When Insertion sort performs in best case with $O(n)$, for example x, the Total Complexity for Best Case = $\theta\left(n + n\log\frac{n}{x}\right)$.

When Insertion sort performs in worst case with $\theta\left(n^2\right)$, for example x^2, the Total Complexity for Worst Case = $\theta\left(nx + n\log\frac{n}{x}\right)$.

Quick Sort

Quick Sort, known as partition exchange sort, is a sorting algorithm which relies on a partition operation based on the fact that it is faster and easier to sort two small lists than one larger one. Like Merge Sort, it is a recursive algorithm and developed with the divide and conquer approach. The method recursively calls itself for sorting once a partition has been created. It is also the fastest generic sorting algorithm in practice. In this sort, an element called pivot is identified and that element is fixed in its place by moving all the elements less than or equal to that to its left and all the elements greater than that to its right. This operation is called partitioning, and is recursively called until all the elements are sorted. There are many different Quick Sorts that pick pivot in different ways: always pick first element as pivot; always pick last element as pivot; pick a random element as pivot; pick median as pivot.

Quicksort is not stable, that means Quicksort may change the relative order of records with equal keys. Quicksort is in-place algorithm. It don't need an auxiliary array. The complexity of the quick-sort algorithm is essentially $O(n\log n)$ when the input is a random permutation. Best case is $O(n\log n)$, average case is $O(n\log n)$ and the worst case is $O(n^2)$. This algorithm is sorts a list of data elements significantly faster than any of the common simple sorts in practice, though there remains the chance of worst case performance.

3. Calculation

In order to sort a list of elements, first of all, we need to analyze the given problem, i.e. whether the given problem has small numbers, large values. Here the values lie in between -100 and 100. The number of the values can be as small as 100 and as large as 50000.

To obtain the performance curves with those of the algorithm asymptotic complexity, we choose challenging sizes. The metrics to be analyzed include the number of comparisons performed by the sorting algorithm and the time it takes for sorting. What’s more, I run my sorting algorithms on a number of arrays with the same size sufficient to obtain the average for a given metric with 95% confidence within a 5% precision.

We set the number of samples you want to sort and run the simulation program a certain time N for each case accordingly, collecting the value for the metric x_i at each run. Based on these values we compute the sample mean $\overline{x}$ and standard deviation s. Then the number n we should run the program to obtain the selected statistical confidence is:

$$n = \left(\frac{100zs}{r\overline{x}} \right)^2$$

where $z = 1.96$, $r = 5$, $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}\left(x_i - \overline{x}\right)^2}$.

The generalized performance and running time determined of the four algorithms on integer arrays for different sizes and running times is summarized in the tables below:

Running times = 50, Input Size = 100

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	2605	181.1076	7.4262
	Time	0.0041	0.0017	259.851
Merge Sort	Comparison	541.3	6.2833	0.2070
	Time	0.0013	0.00053605	269.5
Merge-Insertion Sort	Comparison	586.9	10.6201	0.5032
	Time	0.0015	0.002	2889.7
Quick Sort	Comparison	955.92	88.7771	13.2535
	Time	0.0017	0.00092162	431.1016

According to the result, we need to use 500 running times for this input size.

Running times = 50, Input Size = 1000

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	248750	5577.6	0.7726
	Time	0.2497	0.0324	25.9187
Merge Sort	Comparison	8706.6	18.4067	0.0069
	Time	0.0134	0.0027	62.7178
Insertion-Merge Sort	Comparison	9324.5	43.044	0.0327
	Time	0.0124	0.002	101.6855
Quick Sort	Comparison	16319	0.0032	6.3785
	Time	0.0156	0.0042	112.9881

According to the result, we need to use 150 running times for this input size.

Running times = 50, Input Size = 10000

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	2.491e7	1.3915e5	0.048
	Time	24.8253	1.3716	4.6907
Merge Sort	Comparison	1.2036e5	69.1184	5.067e-4
	Time	0.1798	0.0309	45.2618
Insertion-Merge Sort	Comparison	1.2395e5	102.8346	0.0011
	Time	0.1641	0.021	25.1223
Quick Sort	Comparison	3.9266e5	1.1478e4	1.3131
	Time	0.2351	0.0204	11.5422

According to the result, we need to run 50 times for this input size.

Running times = 30, Input Size = 20000

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	9.9637e7	6.0255e5	0.0562
	Time	96.6477	6.8782	7.7828
Merge Sort	Comparison	2.6069e5	76.0911	1.3091e-4
	Time	0.3684	0.0295	9.8689
Insertion-Merge Sort	Comparison	2.6787e5	147.9303	4.6864e-4
	Time	0.3461	0.0399	20.4282
Quick Sort	Comparison	1.2903e6	2.0882e4	0.4025
	Time	0.6456	0.1092	43.9618

According to the results, we need to run 45 times for this input size.

Running times = 20, Size = 30000

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	2.2356e8	8.2079e5	0.0207
	Time	209.0036	10.2153	3.6709
Merge Sort	Comparison	4.0827e5	99.9738	9.2139e-5
	Time	0.5435	0.0088	0.4006
Insertion-Merge Sort	Comparison	4.2536e5	195.7735	3.2552e-4
	Time	0.5092	0.0076	0.3458
Quick Sort	Comparison	2.68e6	2.8069e4	0.1686
	Time	1.1243	0.0165	0.3303

According to the results, we can choose running times as 5 for this input size.

Running times = 10, Size = 50000

Algorithm	Metric	Mean	Standard Deviation	n
Insertion Sort	Comparison	6.21668e8	1.6919e6	0.0114
	Time	576.9163	4.7354	0.1035
Merge Sort	Comparison	7.1759e5	154.7444	7.1459e-5
	Time	0.9589	0.0135	0.3028
Insertion-Merge Sort	Comparison	7.3996e5	349.8505	3.435e-4
	Time	0.8958	0.0098	0.1854
Quick Sort	Comparison	6.9484e6	6.2601e4	0.1247
	Time	2.6669	0.0627	0.8481

According to the results, we can choose running times as 1 for this input size.

After we obtain the number we should run the program, we can guarantee to obtain the selected statistical confidence, if we run our sorting algorithms sufficient enough.

4. Performance Comparison

This section presents the comparison and discussion of the results. The performance of these algorithms was measured and analyzed in number of comparison and time consumed with respect to the size of the input data. Results are graphically presented.

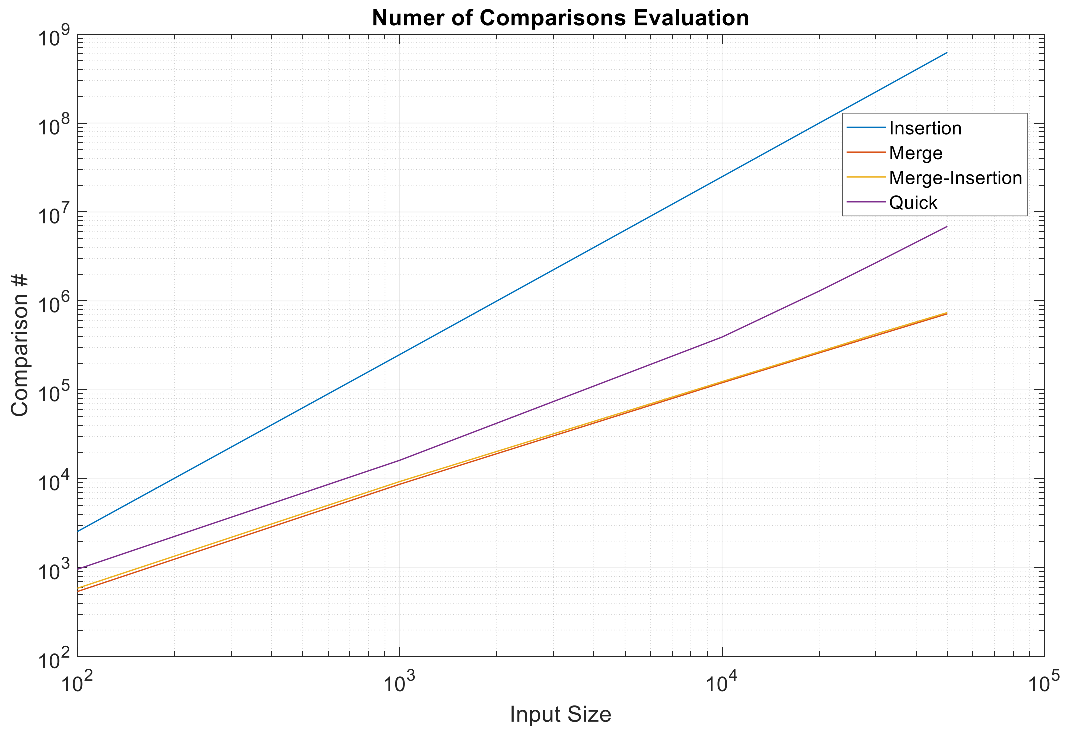

Fig.1 Number of Comparisons Evaluation

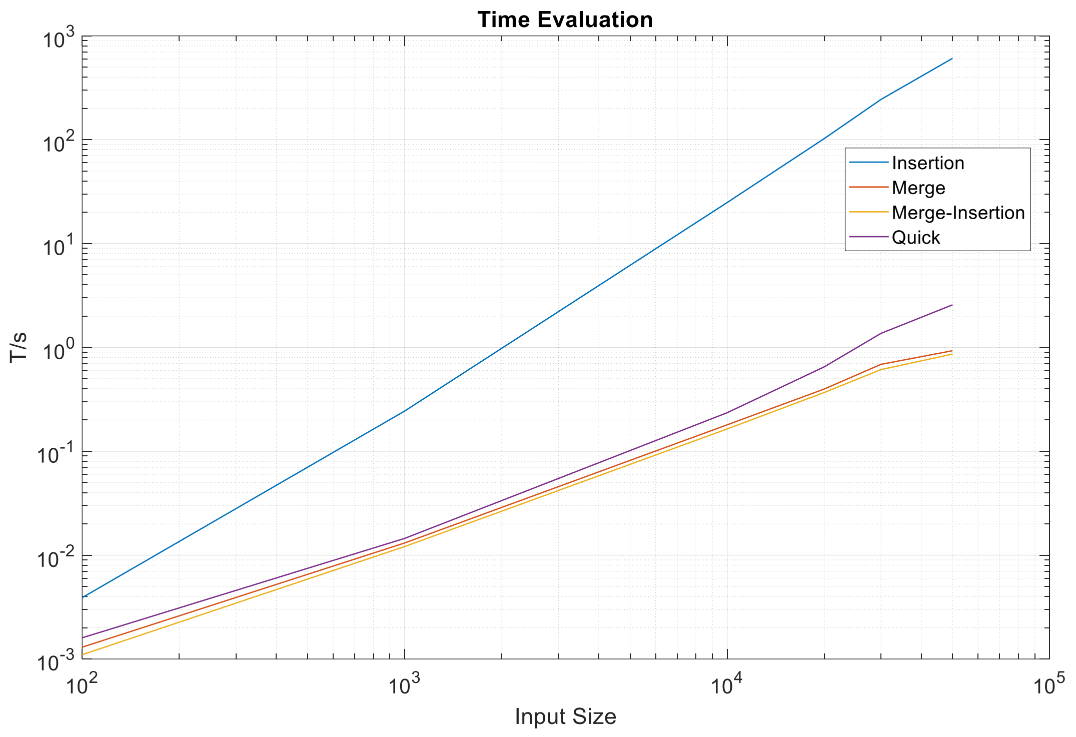

Fig.2 Number of Comparison Evaluation

Graph 1 and 2 illustrate results of complexities of the sorting techniques in terms of input data size against number of comparisons and time consumed respectively. Through analyzing the metrics, we can discover that the metrics increase as the array size increases, more specifically they are directly proportional to the input data size.

The results indicate that in majority of the cases considered, merge-insertion sort technique is faster than insertion sort and quick sort in sorting data of any input data size. Similarly, we observed that the slowest technique of the four is insertion sort. It uses the longest time, the largest comparisons and assignments. And these metrics increases tremendously for the larger arrays. While quick sort technique is faster and requires less comparisons than insertion sort, but slower and requires more comparisons than merge sort and merge-insertion sort. Merge sort and merge-insertion sort algorithms compete in almost all the cases of input data size.

The finding obtained also shows that for small input, the performance for those algorithms except insertion sort is very close to each other, but for the large input, merge-insertion sort is the fastest and the insertion sort is the slowest. Even though merge sort has slight less number of comparison and merge-insertion sort has slight less time consumed compared with each other at the beginning, as size increases, the difference between merge-insertion sort and merge sort becomes smaller.

From the above analysis, it is clear performance of merge-insertion sort and merge sort perform better than other sorts. Insertion sort is a fairly simple algorithm and can be easily implemented, which can be used for small amount of data but not for large amount of data. The algorithm of quick sort is rather complex and it will give poor running time in case of large sets of data.

4. Conclusion

Every sorting algorithm has some advantages and disadvantages. This project deals and analyzes four most commonly used comparison based sorting algorithms and evaluate their performance. We can conclude that merge and merge-insertion sort show better performance than quick sort, and much better than insertion sort.

The difference between them helps us have a better understanding of the algorithm, so as to make the best usage of them. Furthermore, improving such sorting algorithm is needed to facilitate its applications in order to reduce time and resources.

5. Future work

This research is an initial step for future work. We realized that in the solution space, numerous implementation solutions of sorting techniques exist. However, time constraint has limited this study to only four sort techniques. We can investigate complexities of other sorting techniques or take into account for other factors like memory space, stability.

REFERENCES

[1] P.Adhikari, Review on Sorting Algorithms, "A comparative study on two sorting algorithm", Mississppi state university, 2007.

[2] T. Cormen, C.Leiserson, R. Rivest and C.Stein, *Introduction To Algorithms*, McGraw-Hill, Third Edition, 2009, pp.15-17.

[3] M. Goodrich and R. Tamassia, Data *Structures and Algorithms in Ja*va,John wiley & sons 4th edition, 2010, pp.241-243.

[4] R. Sedgewick and K. Wayne, *Algorithms*,Pearson Education, 4th Edition, 2011,pp.248-249.

[5] T. Cormen, C.Leiserson, R. Rivest and C.Stein,*Introduction to Algorithms*, McGraw Hill, 2001, pp.320-330.

[6] D. Cantone and G. Cincotti, "QuickHeapsort, an Efficient Mix of Classical Sorting Algorithms," presented at the Proceedings of the 4th Italian Conference on Algorithms and Complexity.