Understanding Merge Sort and Its Performance

Steven Kang

Horace Greeley High School

March 2024 (1490 words)

1 Introduction

When you search "HGHS" on Google, it is reasonable to expect that the most relevant websites are at the top of the search results. The process under the hood is like this: a program called Googlebot "crawls" billions of pages on the web related to the search words [Google, 2024]. Amazingly, Google can return the ordered search results in less than a second, thanks to the sorting algorithm, a topic computer scientists have studied for decades.

Researchers and engineers have developed many sorting algorithms; the main differences are in the complexity and the speed of processing data. Bubble sort is one of the most basic sorting algorithms—it is intuitive but slow in sorting large amounts of data. Merge sort, on the other hand, is a more sophisticated sorting algorithm invented by John Von Neumann in 1945 [Knuth, 1973]. Using the divide-and-conquer approach, merge sort breaks an array into several sub-arrays, and then compares and merges every pair of sub-arrays until it creates a single sorted list. This article explains how merge sort works and assess its performance compared to bubble sort.

There have been many works on merge sort. However, most reported experimental results were on Linux or Unix operating systems. It is still interesting to observe how merge sort performs on a MacOS with a modern Apple M2 chip. In addition, as people like me—a high school student—navigates Google to find explanations of the algorithm, they often become perplexed by the articles that assume the reader knows algorithm analysis, which typically requires at least one year of study on the subject. My goal is to explain merge sort from the perspective of a high school student after taking a 4-month coding class and provide comprehensible data that will leave a lasting impression on the efficiency of merge sort. Last but not least, despite knowing that merge sort is more performant than bubble sort, it is fascinating to prove a theory through empirical studies, from understanding the algorithm to writing the code, and from designing experiments to collecting data. This article reveals all the details for reproducing the results and shares them with my curious fellow students.

2 Research On Merge Sort

Over the last decades researchers have proposed various methods to enhance the performance of the traditional merge sort. I examine two variants of merge sort—natural merge sort and external merge sort.

Instead of sorting the raw data, natural merge sort finds already-sorted sequences or “runs” of elements to reduce the number of comparisons needed during the merging phase. One popular implementation of natural merge sort is Timsort that was invented by Tim Peters [Peters, 2002]. Timsort is widely used in industrial software such as Python, Java library, and the Android operating system. Buss and Knop proved the upper and lower bounds of several natural merge sort algorithms, including Timsort and proposed a new 2-merge and a-merge sort, which outperformed Timsort and was simpler to implement [Buss and Knop, 2019].

Sorting large datasets could be problematic if datasets cannot fit entirely into a computer’s memory (RAM). To address this issue, external merge sort splits the dataset into smaller chunks that are read, sorted, and written to a temporary file in external memory (usually a hard drive). Subsequently, these sub-files are sorted and merged into one bigger file. This process operates on portions of the dataset one at a time, thus minimizing memory requirements. This technique is commonly employed in external storage systems such as databases and file-processing programs that sort large volumes of data. Chronis proposed eliminating parts of the input before sorting or writing them to secondary storage to reduce the high cost when data significantly exceeds memory capacity during external merge sort [Chronis et al., 2020].

3 Merge Sort Algorithm

There are two distinctive ways to implement merge sort—top-down and bottom-up. I examine the bottom-up approach since the top-down approach requires understanding recursive functions, a concept often beyond the scope of an introductory coding class.

Given an input array of integers, the algorithm splits the array into sub-arrays, starting from single-element sub-arrays. Sub-arrays are continuous arrays in the original array. At each step, a pair of adjacent sub-arrays is merged to create a new larger sub-array with elements sorted in ascending order. For example, Figure 1 shows an array of seven integers with four sub-arrays of size one or two. Merging the first two sub-arrays produces a larger sub-array of four integers. This process is repeated until the size of the merged array is greater than or equal to the size of the input array.

Let's trace the steps using the example in Figure 2. At Step 1, individual elements are sub-arrays, so the sub-array size is one. There are four pairs of adjacent sub-arrays. Merging each pair creates new sub-arrays of two integers. These new sub-arrays are processed at step 2, where the number of pairs is two. Similarly, merging these sub-arrays produces new sub-arrays of four integers. At step 3, given the sub-array size of four, there is only one

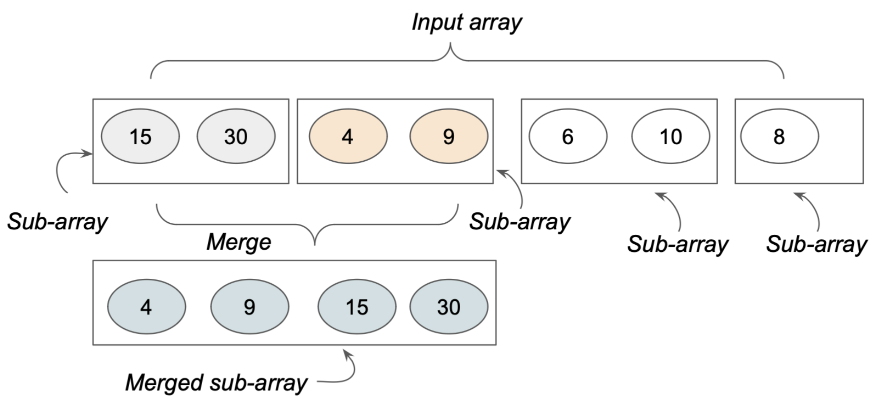

Figure 1: Array and sub-array in merge sort. Two adjacent sub-arrays are merged into a larger sub-array.

pair of adjacent sub-arrays. Finally, at step 4, the merged sub-array size is seven, which equals the input size, so the algorithm stops and returns the sorted array.

To implement the above algorithm, I create an outer `for-loop` that iterates as long as the sub-array size is less than the input array size, corresponding to steps in Figure 2. The sub-array size doubles after each iteration. At each step, the number of sub-array pairs is given by $InputArraySize \div SubArraySize \div 2$, rounding up to the nearest integer. Then, I use an inner `for-loop` to merge every pair of sub-arrays, i.e., Left and Right in Figure 2. I use *leftStart*, *leftEnd*, *rightStart*, and *rightEnd* to represent the left and

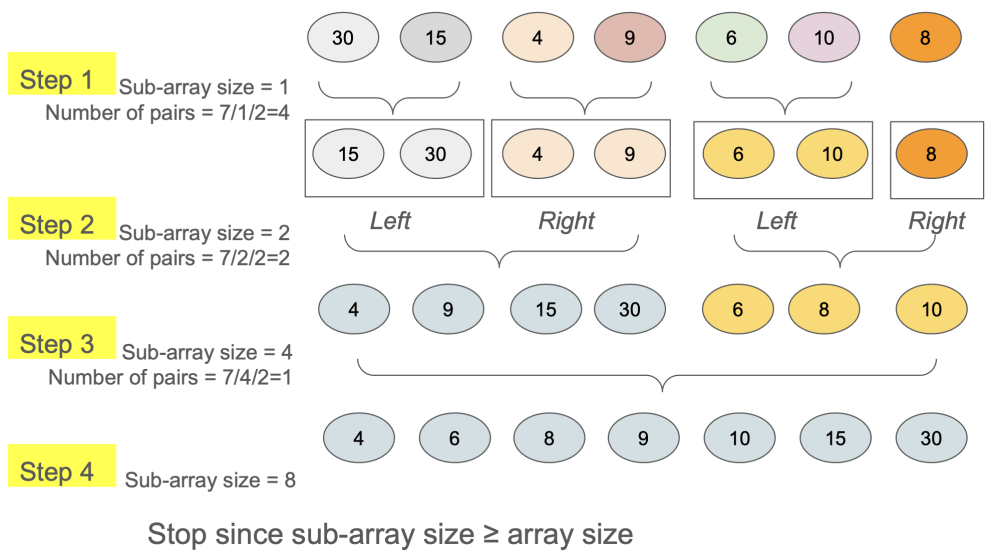

Figure 2: Illustration of running merge sort on an array of 7 integers.

right sub-arrays. They can be calculated as follows:

$$leftStart = pairIndex \times subArraySize \times 2$$

$$leftEnd = leftStart + subArraySize - 1$$

$$rightStart = leftStart + subArraySize$$

$$rightEnd = leftStart + 2 \times subArraySize - 1$$

In effect, once we find the leftStart, the remaining positions can be derived by adding SubArraySize to the leftStart. When the algorithm merges two sub-arrays, there are two special cases. If the rightStart exceeds the input array, the algorithm reaches the end of the array, so it breaks the inner `for-loop`. If the rightEnd exceeds the input array, the algorithm sets rightEnd to the last element of the array.

4 Evaluation

4.1 Implementation

I implemented both merge sort and bubble sort in Java. For bubble sort, I used a nested `for-loop`: the outer `for-loop` controls the number of times to go through the array, and the inner `for-loop` passes over the array to compare each pair of numbers. For merge sort, I implemented the bottom-up approach as described in Section 3. The two main functions I made were `mergeSort` and `mergeSortedArray`. The total line of code for bubble sort is 18, while merge sort takes 86 lines of code. The code is available at `https://github.com/stevenhanwen/learn-sorting-algorithms`.

4.2 Experimental Setup

Testbed: The experiments are conducted on a MacBook Air with an Apple M2 chip of 8-core CPU and 8 GB memory. The operating system used to run the sorting algorithms was MacOS Sonoma 14.2.1. The MacBook was also installed with the Java Virtual Machine (JVM) 21.0.2.

Dataset: I measured the run time of each algorithm with large data sets. I used the Random Number Generator by Text Tools [Tools, 2024] to create arrays of integer numbers with sizes increasing from 2000 to 10000 in steps of 2000 in different files named `2k.data`, `4k.data`, etc.

4.3 Results

Figure 3 shows the results. The bubble sort algorithm displays a run time that increases linearly as the size of the array goes up by steps of 2000 integers. For example, when the size of the array is 2000, the run time is around 5 milliseconds; when the size of the array is 10,000, the run time is around 50 milliseconds. Therefore, run time increases by about 9 milliseconds per 2000 integers.

Compared to bubble sort, merge sort clearly produces much better results—as array size increases, its run time stays relatively stable as shown by the horizontal line in Figure 3. For 2000 integers, merge sort is 5 times faster than bubble sort., and for 10,000 integers 50 times faster. I noticed a small spike at the array size of 4000, which needs further investigation.

These results show merge sort outperforms bubble sort, and as the array size increases, the benefit of using merge sort increases due to the divide-and-conquer strategy.

5 Conclusion

This article covered recent research on merge sort and dived deep into the algorithm. Results show that it is an efficient sorting method especially compared to bubble sort. Sorting is almost used in every software. If a

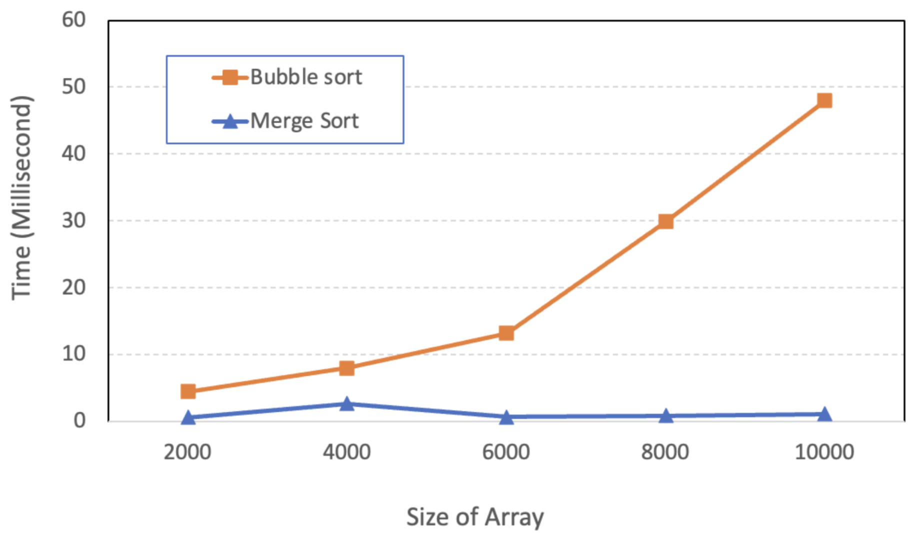

Figure 3: Comparison between bubble sort and merge sort. Each data point represents the average over three runs (Lower is better).

program does not choose the right method, then users will experience delayed responses such as when searching on Google. Of course, even if we implement efficient sorting methods, it is still only one part of a software. Engineers also have to consider the other components to build a successful product.

References

[Buss and Knop, 2019] Buss, S. and Knop, A. (2019). Strategies for stable merge sorting. SODA '19, page 1272–1290, USA. Society for Industrial and Applied Mathematics.

[Chronis et al., 2020] Chronis, Y., Do, T., Graefe, G., and Peters, K. (2020). External merge sort for top-k queries: Eager input filtering guided by histograms. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2423–2437, New York, NY, USA. Association for Computing Machinery.

[Google, 2024] Google (2024). How Google search organizes information. Accessed on March 14, 2024.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley Professional.

[Peters, 2002] Peters, T. (2002). [python-dev] sorting. Accessed on March 14, 2024.

[Tools, 2024] Tools, T. (2024). Random number generator. Accessed on March 14, 2024.