Sorting Algorithms Theoretical and Experimental Comparison

Mihyar Al Hariri
Department of Computer Science,
Faculty of Computer Science and Mathematics,
West University of Timisoara,
Email: mihyar.la10@e-uvt.ro

May 2023

Abstract

One of the biggest and most important problems and one of the essential tools in the Computer Science field is sorting algorithms. the problem we are going to discuss is there are several factors that must be taken into consideration; time complexity stability, and memory space in using the sorting algorithms. In this paper, we are going to compare some of the most famous sorting algorithms and the comparison will be a theoretical and experimental comparison

Contents

1 Introduction

Motivation of the problem

The difficulty of sorting algorithms is equivalent to managing a messy room full of stuff. Assume you have a slew of unsorted goods all over the place, and your duty is to arrange them in a specified order, such as smallest to largest or alphabetically. The objective is to build an effective mechanism for automatically sorting these things and placing them in their proper locations. Sorting algorithms are similar to sets of instructions that help you through the process of selecting how to compare and rearrange objects until they are in the desired order. It's similar to organizing a cluttered space, but instead of the real stuff, we're working with data and figures.

Sorting algorithms are significant because they assist us in organizing data, improving search speed, optimizing operations, effectively analyzing data, and serving as building blocks for other algorithms.

Existing sorting algorithms differ in terms of time complexity, space complexity, stability, adaptability, and applicability for specific data types. Choosing the best algorithm is determined by the problem's specific requirements and constraints.

Existing solutions for sorting algorithms are similar to a toolbox containing various tools. Each tool has advantages and disadvantages, and the decision is dictated by the nature of the problem. Simple tools are available for small datasets, efficient tools are available for large datasets, specialized tools are available for certain data types, and adaptive techniques are available for partially sorted data.

Informal Description of Solution

To compare the sorting algorithms, we will follow the following steps:

1. Selection of Sorting Algorithms: We have chosen to compare six commonly used sorting algorithms - Bubble Sort, Insertion Sort, Quick Sort, Selection Sort, Merge Sort, and Heap Sort.

2. Code Implementation: We obtained the code for the sorting algorithms from trusted sources. In this paper, the code used can be found on my GitHub repository at the following link: `https://github.com/MihyarH/MPI-Sorting-Algorithms.git`. The code was implemented in Python.

3. Experimental Setup: We conducted experiments to measure the performance of each sorting algorithm. The details of the experiments are provided at the end of this paper.

By following these steps, we aim to provide a comprehensive comparison of the selected sorting algorithms. The results and analysis of the experiments can be found at the end of this paper.

Informal example Let us consider the performance of Bubble Sort, Insertion Sort, Quick Sort, Selection Sort, Merge Sort, and Heap Sort.

We look to seek solid code implementations of these sorting methods, ideally from trusted sources or repositories.

We set up experiments in Python with random test cases of varying sizes. We measure execution time and memory usage for each algorithm.

We collect data and analyze it to discover performance differences between sorting algorithms. We take into account issues such as time complexity, space complexity, and stability.

To properly compare the performance of the algorithms, we build visualizations such as charts or graphs to explain our findings.

Based on the analysis, we draw judgments about each algorithm's strengths and errors, making recommendations for certain scenarios or data types. We obtain vital insights into the efficiency of sorting algorithms as a result of this process, and we can make educated decisions when selecting the best algorithm for a given task.

Reading instructions 1. Introduction: Read through the introduction carefully in order to gain an understanding of its objective and compare different sorting algorithms being tested.

2. Code Implementations: You can find the code at the following link: `https://github.com/MihyarH/MPI-Sorting-Algorithms.git`.

4. Data Analysis: Review the section on data analysis to learn about how the collected information has been assessed and considered when making analysis decisions.

5. Visualizations: Visit this section on visualizations to view charts or graphs depicting the relative performance of various sorting algorithms.

6. Conclusions and Recommendations: Read this section in order to gain an understanding of any judgments drawn and specific suggestions made in regard to conclusions and recommendations.

7. Final Remarks: As your research comes to an end, read over the final remarks section for an overview and highlight of all its significance and key takeaways.

These concise reading instructions offer a quick introduction to each section, making for easier understanding of key points and findings regarding sorting algorithms comparison solutions.

2 Analysis of Algorithm

The analysis of an algorithm defines the estimation of resources required for an algorithm to solve a given problem. Sometimes the resources include memory and time. Running time and memory required is of primary concern for the reason that algorithms that needed a month or year to solve a problem are not useful. Besides that, it also requires gigabytes of main memory to solve a problem and is not efficient. In general, to find a suitable algorithm for a specific problem we have to analyze several possible algorithms. By doing so, we might locate more than one useful algorithm. One way to recognize the best suitable algorithm for a given problem is to implement both algorithms and find out their efficiency, most importantly the running time of a program. If the observed running time matches the predicted running time of the analysis and also outperforms the other algorithm that would be the best suitable algorithm for the given problem. Generally, various factors affect the running time of a program in which the size of the input is the primary concern. Moreover, most of the basic algorithms perform very well in small arrays and take a longer time for a bigger size. Typically, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. [Karunanithi et al., 2014]

2.1 Running Time Analysis

The running time analysis is a theoretical process to categorize the algorithm into a relative order among functions by predicting and calculating approximately the increase in running time of an algorithm as its input size increases. For instance, a program can take seconds, hours, or even years to complete the execution, usually, this depends upon the particular algorithm used to construct the program. Besides the run time of a program describes the number of primary operations executed during implementation. To ensure the execution time of an algorithm we should anticipate the worst-case, average-case, and best-case performance of an algorithm. These analyze, assist the understanding of algorithm complexity.[Karunanithi et al., 2014]

2.1.1 Worst-Case Analysis

The worst-case analysis anticipates the greatest amount of running time that an algorithm needed to solve a problem for any input of size n. The worst-case running time of an algorithm gives us an upper bound on the computational complexity. Significantly it also guarantees that the performance of an algorithm will not get worse. In general, we consider the worst-case performance of an algorithm very often.[Karunanithi et al., 2014]

2.1.2 Best-Case Analysis

The best case analysis anticipates the least amount of running time that an algorithm needs to solve a problem for any input of size n. In this, the running time of an algorithm gives us a lower bound on the computational complexity. Most analysts do not consider the best-case performance of an algorithm for the reason that it is not useful.[Karunanithi et al., 2014]

2.1.3 Average-Case Analysis

The average case analysis anticipates the average amount of running time that an algorithm needed to solve a problem for any input of size n. Generally, the average case running time is considered approximately as bad as the worst case time. However, from a practical point of view, it is frequently useful to review the performance of an algorithm if we average its behavior over all possible sets of input data. One of the obstacles in the average case analysis is that it is much more difficult to carry out the process and typically requires considerable mathematical refinement, for that reason worst-case analysis became prevalent.[Karunanithi et al., 2014]

3 Running Time Calculations

One can evaluate the running time of a program in different ways. In general, there is more than one algorithm anticipated to take a similar time to complete because we have to program both algorithms and decide which is faster in practice. In the following, we describe how to calculate the running time of a simple program.[Karunanithi et al., 2014]

sum-of-list(A, n)	Cost	No. of times	
{			
int $i,\ \text{total} = 0;$	$1(C_1)$	1	/*1*/
for$(i = 0; i < n; i++)$	$2(C_2)$	$n + 1$	/*2*/
total=total+A[i];	$2(C_2)$	n	/*3*/
return total;	$1(C_1)$	1	/*4*/
}			

As per the rule, the cost for each statement is described in the above program. Lines 1 and 4 compute only one time, so they each cost one. Similarly, line 4 takes 1 unit of cost because it executes once. In line 2, the for loop has 1 unit of cost for initialization and $(n + 1)$ units of cost for testing the condition, and n units of cost for incrementing the total $(2n + 2)$.[Karunanithi et al., 2014]

$$T_{\text{sum-of-list}} = 1 + 2(n + 1) + 2n + 1 = 4n + 4$$

$$T(n) = Cn + C_0, \text{ where } Cn \text{ and } C' \text{ are constants}$$

Likewise, for different types of problems, if we calculate the running time for instance,

$$T_{\text{sum}} = K \Rightarrow T(n) = O(1)$$

$$T_{\text{sum-of-list}} = Cn + C_0 \Rightarrow T(n) = O(n)$$

$$T_{\text{sum-of-matrix}} = a.n^2 + b.n + c \Rightarrow T(n) = O(n^2)$$

4 Classification of Sorting Algorithms

In general, sorting algorithms can be classified with various parameters. Some of them are detailed below.[Karunanithi et al., 2014]

4.1 Space Complexity

The space complexity of an algorithm is a factor that considers seriously when selecting an algorithm. There are two kinds of memory usage patterns such as "in-place" sorting in which the algorithm needs constant memory size (i.e.) O(1) beyond the items being sorted. While the other sorting methods use additional memory space according to their relative input size may be called "out-place" sorting. Due to this, the in-place sorting algorithms are slower than the algorithms that use additional memory. Sometimes escalation can be achieved by considering O(log n) additional memory in in-place sorting algorithms.[Karunanithi et al., 2014]

4.2 Stability

A stable sorting algorithm preserves the relative order of elements with equal values. For instance, a sorting algorithm is stable means whenever there are two elements a[0] and a[1] with the same value and with a[0] show up before a[1] in the unsorted list, a[0] will also show up before a[1] in the sorted list.[Karunanithi et al., 2014]

5 Sorting Algorithms

5.1 Bubble Sort Algorithm

Bubble Sort is a fundamental algorithm taught in computer science courses that serve as a building block for other sorting algorithms. It is an intuitive comparison-based algorithm that iteratively moves through an array, swapping adjacent elements if they are out of order and "bubbling up to the top" similar to how liquid surface bubbles rise to the surface. Bubble Sort derives its name from how smaller elements tend to rise toward the surface - similar to how bubbles rise with rising liquid surfaces.

5.1.1 Explanation

The Bubble Sort algorithm begins by comparing two elements in an array. If one element exceeds another, a swap operation takes place; otherwise, this process repeats until all adjacent pairs of elements have been processed and reached an endpoint of an array.

Once complete, the largest element will appear at the end of an array. Subsequent passes reduce the number of elements needing sorting by one since its appropriate place has already been achieved by this first pass; and so forth until no swappings are required any longer and thus an array is fully organized.

Bubble Sort has an ordered complexity of O(n2) for worst and average case scenarios; when using already sorted arrays it becomes O(n).

5.2 Insertion Sort Algorithm

Insertion sort is a simple and efficient sorting algorithm useful for small lists and mostly sorted lists. It works by inserting each element into its appropriate position in the final sorted list. For each insertion, it takes one element and finds the appropriate position in the sorted list by comparing it with neighboring elements and inserts it in that position. This operation is repeated until the list becomes sorted in the desired order. Insertion sort is an in-place algorithm and needed only a constant amount of additional memory space. It becomes more inefficient for the greater size of input data when compared to other algorithms. However, in general, insertion sort is frequently used as a part of more sophisticated algorithms. The algorithm that explains insertion sort is as follows.[Karunanithi et al., 2014]

5.2.1 Explanation

The Insertion Sort Algorithm (ISA) is an efficient sorting algorithm that works by splitting input data into two sections - an ordered subarray containing only its first element, while an unsorted

subarray contains the remaining ones. At first, only its first element will be placed into this sorted subarray before returning back into the unsorted subarray for processing.

Sorting an array requires iteratively traversing each unsorted subarray from its second element to the last; at each iteration, one element from iteration one will be randomly selected and compared against those present in its sorted subarray for comparison purposes.

The algorithm employs a while loop to shift elements greater than the current element rightward, creating space for it to fit back in its proper order in the subarray.

The process continues until either the current element is less than or equal to compared element or the beginning of the sorted subarray is reached; once found, the current element will be placed within that subarray for sorting purposes.

This process repeats for each element in an unsorted subarray until all elements have been processed; ultimately, this leads to the entire array being sorted in ascending order.

5.3 Quick-Sort Algorithm

Quick Sort is a well-known sorting algorithm renowned for its efficiency and simplicity, using a divide-and-conquer approach that divides an input array into smaller subarrays that are then sorted recursively before being combined together into the final sorted array.

5.3.1 Explanation

Quick Sort works by first selecting a pivot element from an input array as a reference for partitioning it into two subarrays based on their distance from this pivot, small being below and large above it respectively. Subarrays are then independently sorted before being combined together using pivot element as part of the final array sort.

Quick Sort's efficiency lies in its ability to divide data efficiently into subarrays, significantly decreasing comparison requirements compared with traditional sorting algorithms. On average, Quick Sort typically requires O(n log n) time complexity making it suitable for large datasets.

Note: Selecting an effective pivot can dramatically impact an algorithm's performance. While the pseudocode here selects either the first or last element as its pivot, other techniques exist such as randomly selecting elements or using median-of-three to improve it in certain instances.

5.4 Selection Sort Algorithm

Selection sort is another simple sorting method that works better than bubble sort and worse than insertion sort. It works by finding the smallest or highest element (most probably desired element) from the unsorted list and swapping it with the first element in the sorted list and then finding the next smallest element from the unsorted list and swapping it with the second element in the sorted list. Consequently, sorted elements are increasing at the top of an array and the rest will remain unsorted. The algorithm continues this operation until the list is sorted. Selection sort is also an in-place algorithm since it requires a constant amount of memory space. Like some other simple sorting methods, selection sort is also inefficient for large arrays. The algorithm for selection sort is as follows.[Karunanithi et al., 2014]

5.4.1 Explanation

Selection Sort is an algorithmic sorting strategy that works by splitting an input array into two separate parts - both are then processed independently to sort. When sorting, Selection Sort randomly chooses elements from both parts that should be swapped - the smaller/larger element from the unsorted part being swapped out with its equivalent in the sorted section until finally all elements in the unsorted are sorted accordingly. This process continues until all array elements have been properly organized into distinct groups.

The algorithm can be summarized in the following steps:

1. Start with an unsorted array of elements.
2. Set the first element as the current minimum (or maximum) value.
3. Scan the remaining unsorted part of the array to find the smallest (or largest) element.
4. If a smaller (or larger) element is found, update the current minimum (or maximum) value.

5. Swap the current minimum (or maximum) value with the first element of the unsorted part.
6. Move the boundary between the sorted and unsorted parts one element to the right.
7. Repeat steps 2–6 until the entire array is sorted.

The selection sort algorithm has a time complexity of $O(n^2)$, where n is the number of elements in the array. It is an in-place sorting algorithm, meaning it operates directly on the input array without requiring additional space. However, the selection sort is not suitable for large arrays or datasets due to its quadratic time complexity.

5.5 Merge Sorting Algorithm

Merge sort uses the divide and conquer approach to solve a given problem. It works by splitting the unsorted array into n sub-array recursively until each sub-array has 1 element. In general, an array with one element is considered to be sorted. Consequently, it merges each sub-array to generate a final sorted array. The divide and conquer approach works by dividing the array into two halves such as sub-array and follows the same step for each sub-array recursively until each sub-array has 1 element. Later it combines each sub-array into a sorted array until there is only 1 sub-array with desired order. This can also be done non-recursively however, most consider only recursive approaches for the reason that non-recursive is not efficient. Merge sort is a stable sort meaning that it preserves the relative order of elements with equal keys. The algorithm for merge sort is as follows.[Elkahlout and Maghari, 2017]

5.5.1 Explanation

The **Merge Sort** algorithm follows a divide-and-conquer approach to sort an array A from index p to r. The algorithm can be described as follows:

1. **MergeSort**: This function recursively divides the array into two halves until the base case is reached, where $p < r$. It then calls itself for each half and finally merges the two sorted halves using the **Merge** function.
2. **Merge**: Given an array A, indices p, q, and r, this function merges the two subarrays $A[p..q]$ and $A[q+1..r]$ into a single sorted array. It first creates two temporary arrays L and R to store the values of the subarrays. Then, it merges the elements of L and R back into A by comparing the elements at corresponding indices and placing the smaller element in A. This process is repeated until all the elements have been merged.

The merge sort algorithm has a time complexity of $\mathcal{O}(n \log n)$, where n is the number of elements in the array. It is a stable sorting algorithm, meaning that the relative order of equal elements is preserved. Merge

5.6 Heap-Sort Algorithm

Heap Sort is based on the heap data structure and in-place sorting algorithm. It is quite slower than merge sort in real applications even though it has the same theoretical complexity. Unlike merge sort and quick sort, it does not work recursively. In general, the heap is a specialized tree-based data structure that satisfies the heap property, mostly we use binary trees. The tree structure is well-balanced, space efficient, and fast. Heap sort works by building a heap from the input array

and then removing the maximum element from the heap and placing it at the end of the final sorted array i.e. n 1st position. Every time when it removes the maximum element from the heap it restores the heap property until the heap is empty. Thus it removes the second largest element from a heap and puts it on the n 2nd position and so on. The algorithm repeats this operation until the array is sorted. Heap sort does not preserve the relative order of elements with equal keys; hence it is not a stable sort. The algorithm for Heap sort is as follows.[Karunanithi et al., 2014]

5.6.1 Explanation

Heap Sort leverages the properties of binary heap data structures to efficiently sort an array in place. It consists of two primary procedures.

- The textscHeapify procedure is responsible for creating a max heap from any given array, using three parameters as input: arr, the size of heap n and index i. When implemented it compares each element at index i with their left and right children and swaps out as necessary in order to maintain max heap property; further recursively calling itself to ensure all elements in an affected subtree have been converted to max heap properties before proceeding further with processing array as whole.

- Heap Sort (textscHeapSort) is the main function for sorting arrays using Heap Sort. It takes as a parameter arr and begins by building a maximum heap from it using the heapify procedure; this involves iterating over all non-leaf nodes of the heap in reverse order until all nodes have been reached; once completed, one element located at its root (i.e. at its roots) will be swapped out and removed before calling Heapify once more so as to restore max heap properties and thus completing its task of sorting ascendant order until everything in an array has been completed and all elements sorted ascendant order by Heap sort!

Overall, Heap Sort's time complexity is $O(nlogn)$; where n represents the number of elements in an array. As it uses comparison-based sorting rather than direct sorting methods to sort array elements efficiently with constant space usage requirements. The Heap Sort algorithm begins by creating a max heap from its input array, by calling the textscHeapify procedure on each non-leaf node of the heap in reverse order and performing comparisons against left and right children of element i to keep max heap property intact - guaranteeing that its largest element always sits at its root of heap construction.

Once we've constructed our max heap, the largest element (at the root) will be swapped out with its opposite and removed from consideration - this ensures it resides correctly within its respective array and then reduces by one. Finally, we reduce its size.

To restore the maximum heap property, we employ the textscHeapify procedure on the new root element, repeating this step for all remaining elements (from $n - 1$ down to 1). Each iteration involves swapping out one root element with each of its counterparts before calling textscHeapify to keep up the maximum heap property.

After the second loop is complete, the array will be sorted in ascending order as a result of repeatedly extracting maximum elements from the heap and placing them at the end of the array.

Heap Sort is an efficient sorting algorithm with an O(n log n) worst-case time complexity, being in-place and only needing constant additional space to complete. Cache performance is excellent due to local memory access patterns allowing access quickly when necessary; however, it might not

be suitable for sorting small arrays with mostly-sorted elements as its overhead for building its heap may be relatively higher when compared with other algorithms.

5.7 Counting Sort Algorithm

Counting sort is a simple, stable, and efficient sorting algorithm with linear running time, which is a fundamental building block for many applications. The counting sort algorithm has been widely used in data processing systems, because of its high efficiency, fast speed, and stable nature. Therefore, a thorough study of its time complexity is required. This paper presents a modified version of counting sort E-Counting Sort with some efficiency improvements. An analysis of the ECounting sort algorithm in comparison with the original counting sort algorithm clearly shows that the E-Counting sort algorithm has a reduced time complexity of approximately half of the original one. The new version can be applied to many real-world applications providing the required result as efficiently and as effectively as the original counting sort without affecting its real nature and improving the efficiency of the application program.[Bajpai and Kots, 2014]

5.7.1 Explanation

When the range of input values is minimal, counting sort is an efficient algorithm for sorting. It operates by counting the occurrences of each individual element and then using that data to generate the sorted output.

The algorithm starts by identifying the value range in the input list. It generates a counting array of the range's length and counts the occurrences of each element. The counting array is then modified to store cumulative counts.

By iterating through the input list, the method generates the sorted output list using the cumulative counts. It locates the corresponding value in the cumulative count array and positions the element in the output list.

The temporal complexity of counting sort is O(n + k), where n is the number of elements and k is the range of values. Because it is not a comparison-based technique, it is only appropriate for non-negative integers or elements with defined keys.

Overall, counting sort is a basic and efficient sorting algorithm that works well when sorting a large number of objects within a narrow value range.

6 Conclusion

On the basis of our analysis and the charts created using these data sets, we can reach some interesting conclusions:

Time Complexity: Out of all of the sorting algorithms tested, Quick Sort was found to perform best when measured against other algorithms in terms of execution time required to sort arrays containing 1 million elements. Insertion Sort and Selection Sort showed comparable performance while Bubble Sort's execution times were considerably slower; Merge Sort and Heap Sort both also performed adequately but their performance could be slightly delayed when compared with Quick Sort's results.

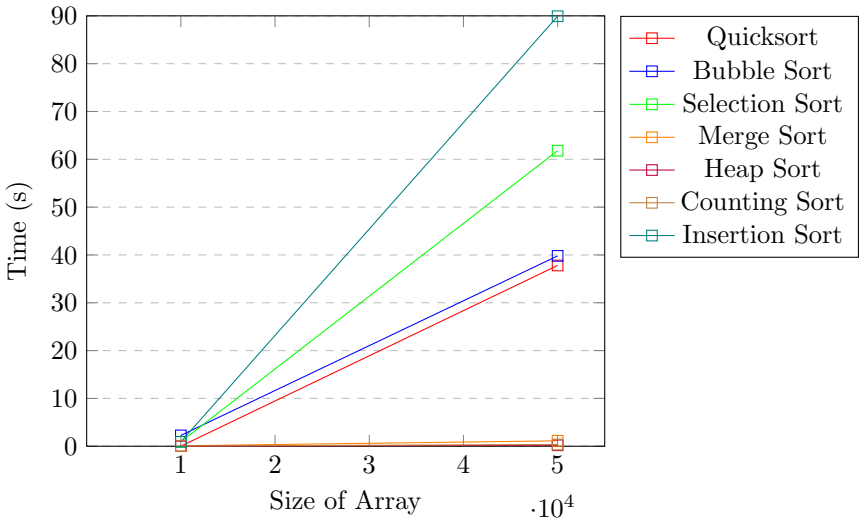

Figure 1: Time taken by various sorting algorithms.

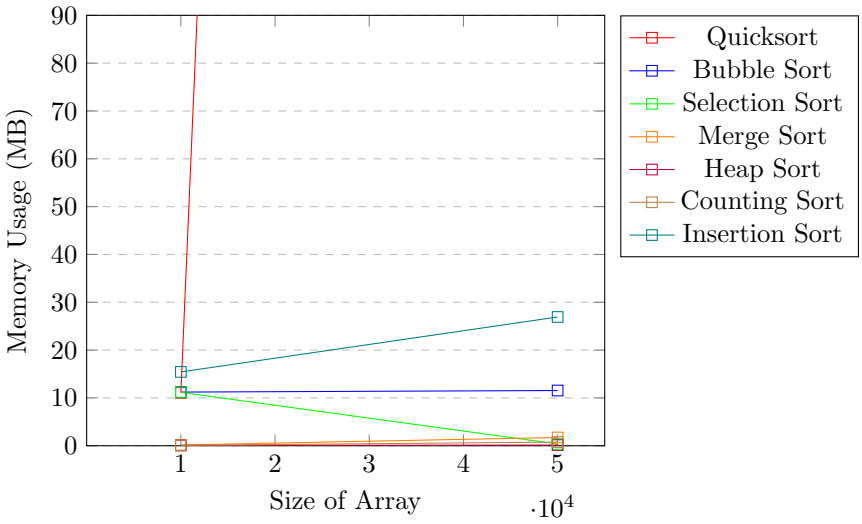

Figure 2: Memory usage of various sorting algorithms.

Memory Usage: Of all of the algorithms considered here, Counting Sort had the lowest memory footprint; consistently using less space than any of its rivals. Meanwhile, Merge Sort and Quick Sort used moderate amounts, while Bubble and Heap Sort consumed considerably more memory; in comparison, Insertion Sort and Selection Sort had reasonable consumption levels.

Algorithm Suitability: Based on your application requirements, different sorting algorithms may be more suitable than others for sorting purposes. Quick Sort is ideal when optimizing for time

complexity in large arrays while Counting Sort provides excellent memory usage control whereas Merge Sort strikes a balance between both time complexity and memory usage for Merge Sort users.

Impact of Input Range: Input value range has an impactful influence on algorithm performance. When dealing with an input array consisting of only small value ranges like between 0 and 1, the performance of all algorithms is affected in particular with Bubble Sort and Quick Sort experiencing slower execution times when compared with other algorithms.

Conclusion: In general, selecting an optimal sorting algorithm depends upon your application's individual requirements for time complexity, memory usage, and input range. It's crucial that these factors be carefully taken into account in order to select an algorithm suitable for your situation.

Table 1: Sorting Algorithms Performance

Sorting Algorithm	**Array Size**	**Time (Seconds)**	**Memory Usage (MB)**
Quicksort	10,000	0.014	11.046875
	50,000	0.123	1,840.78125
	100,000	0.288	26.09375
Bubble Sort	10,000	2.265	11.21875
	50,000	189.908	11.546875
	100,000	470.481	25.8125
Selection Sort	10,000	0.937	11.156875
	50,000	93.088	0.381
	100,000	192.838	98.304
Merge Sort	10,000	0.097	0.170072
	50,000	1.131	1.728328
	100,000	1.812	2.458888
Heap Sort	10,000	0.030	0.060
	50,000	0.196	0.140
	100,000	2.645	7.160
Counting Sort	10,000	0.041	0.0762939453125
	50,000	0.365	0.762939453125
	100,000	1.352	1.0825729370117188
Insertion Sort	10,000	0.918	15.4375
	50,000	89.919	26.921875
	100,000	181.444	31.078125

References

[Bajpai and Kots, 2014] Bajpai, K. and Kots, A. (2014). Implementing and analyzing an efficient version of counting sort (e-counting sort). *International Journal of Computer Applications*, 98(9).

[Elkahlout and Maghari, 2017] Elkahlout, A. H. and Maghari, A. Y. (2017). A comparative study of sorting algorithms comb, cocktail and counting sorting. *International Research Journal of Engineering and Technology (IRJET)*, 4(01).

[Karunanithi et al., 2014] Karunanithi, A. K. et al. (2014). A survey, discussion and comparison of sorting algorithms. *Department of Computing Science, Umea University.*