A Comparative Analysis of Different Sorting Algorithms

Chethana Muppalam

Abstract

This paper presents a comprehensive and comparative study of six widely-used sorting algorithms (Quick Sort, Heap Sort, Merge Sort, Radix Sort, Bucket Sort, and Tim Sort) across different kinds of input sequences. The analysis empirically measures time usage and stability across a variety of input sizes. The algorithms are compared from both theoretical and practical standpoints, as well as their performance across six different use cases using randomized sequences of numbers. This paper also shows how optimizations to some of the slower algorithms results in a drastic performance difference. The measurements provide insights into their effectiveness in various situations and help determine the ideal algorithm to choose.

1. INTRODUCTION

Sorting algorithms play a critical role in data management and processing, particularly when dealing with large datasets. Efficient sorting simplifies data processing by organizing information in a systematic order, facilitating swift retrieval of specific data points. Selecting the most suitable sorting algorithm involves considering factors such as time usage, memory consumption, and ease of implementation. Comprehensive empirical data on sorting algorithms is essential to make informed decisions and optimize performance for specific use cases. This paper aims to compare the most commonly employed sorting algorithms, including Quick Sort, Heap Sort, Merge Sort, Radix Sort, Bucket Sort, and Tim Sort.

2. SORTING ALGORITHMS

2.1. Quick Sort

Quick Sort is like organizing a messy shelf of books. We pick a book as a pivot, then rearrange the shelf so that all the books smaller than the pivot are on one side, and all the books larger are on the other. You repeat this process for each side until the shelf is sorted. The main challenge with the Quick Sort is finding the right pivot and this can be done by a variety of techniques and median of medians is one among them that proved suitable and a more balanced partitioning of the array and improving overall efficiency. Known to work great with big, disordered datasets (average case: O(n log n)). However, it can struggle when data is nearly sorted (worst case: O(n^2)). On average, it's space-efficient, needing just a bit more memory than the size of the list itself (space complexity: O(log n)).

2.2. Heap Sort

Heap Sort is a comparison based sorting algorithm. It is similar to organizing people by height in a line. It creates a tree-like structure called heap where each parent node is larger (for a max heap) or smaller (for a min heap) than its children. After heapifying the array, we repeatedly extract the maximum (or minimum) element and place it at the end, sorting the array. Known to well with large, disorganized lists, an average

time complexity of O(n log n). It stays steady even in the worst-case scenarios, maintaining O(n log n) efficiency. However, it operates in place, making it space-efficient with a constant space complexity of O(1).

2.3. Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that recursively divides an unsorted list into smaller sublists until they consist of only one element or are nearly sorted and then merges these sublists back together in sorted order. This process continues until the entire list is sorted. During the merging phase, elements from the sublists are compared and combined to create increasingly larger sorted sublists until the entire list is sorted. Known to work well for big, messy lists—it's consistently fast with an average time complexity of O(n log n). Even if the data is nearly sorted or in its worst case, it keeps its cool (still O(n log n)). However, it needs extra space to create temporary merge lists, around O(n) in total.

2.4. Radix Sort

Radix Sort is a non-comparison based sort in which elements are not compared to each other. Simple example we can think about is how the mail is grouped and processed by the postal services based on the zip code of a city. Elements are sorted from least significant bits to the most significant bits. After that, all cards are sorted using the second largest digit and so on. While it may not be as versatile as comparison-based sorts like Quick Sort or Merge Sort, Radix Sort can be efficient for sorting integers and strings with fixed-length keys. However, its space complexity can be a concern, particularly when dealing with large ranges of values.

2.5. Bucket Sort

Bucket Sort operates by creating a number of buckets. Each integer is placed into a bucket based on the integer value. A common way is to divide the input element by the integer range, thus defining an integer range for each of the buckets. All buckets are then sorted. After that, the buckets are concatenated into a single list, which is the output of the sorted algorithm. This sort organizes elements into buckets (O(n)), sorting each bucket (O(K)), summing up to O(n+k) in the best scenario. However, if elements are close or reverse sorted, it may degrade as insertion sort performs worst leading to O(n^2). On average, with random or evenly distributed data, it maintains linear time (O(n)). Its space complexity remains O(n+k) due to extra storage needs. Quick Sort is a type of Bucket Sort with 2 buckets at each recursion and elements distributed into the bucket based on the pivot.

2.6. Tim Sort

Algorithm originally developed for use in Python and is now the standard sorting algorithm used in Python and Java libraries. It works as a hybrid between Merge Sort and Insertion Sort. Tim Sort algorithm consists of three phases – we calculate a run length, sort each run of elements using insertion sort, and then recursively sort adjacent runs using Merge Sort. From best to worst, the O(n log n) time complexity of Tim Sort arises from its hybrid approach, which combines elements of Merge Sort and Insertion Sort.

2.7. Theoretical Complexity

Sorting Algorithm	Best Case TC	Average Case TC	Worst Case TC	Storage	Stability
Quick	Ω(n log n)	θ(n log n)	O(n^2)	O(log n)	Unstable
Heap	Ω(n log n)	θ(n log n)	O(n log n)	O(1)	Unstable
Merge	Ω(n log n)	θ(n log n)	O(n log n)	O(n)	Stable
Radix	Ω(nk)	θ(nk)	O(nk)	O(n+k)	Stable
Bucket	Ω(n+k)	θ(n+k)	O(n^2)	O(n+k)	Stable
Tim	Ω(n log n)	θ(n log n)	O(n log n)	O(n)	Stable

3. COMPARISON OF ALGORITHMS

Algorithms Implementation Repository: https://github.com/chethana613/analysis-sorting-algos

3.1. Test Cases

The Sorting algorithms performance is evaluated using different sizes of inputs on the following 6 cases.

A. n randomly chosen integers in the range [0…n]
B. n randomly chosen integers in the range [0…k], k<1000
C. n randomly chosen integers in the range [0..n^3]
D. n randomly chosen integers in the range [0..log n]
E. n randomly chosen integers that are multiples of 1000 in the range [0..n]
F. The inorder integers [0…n] where log(n)/2 randomly chosen values have been swapped with another value

3.2. Picking improved variations to compare for slower algorithms

While executing the vanilla implementation of all the algorithms in Python, it was observed that the performance drastically degraded for the algorithms with the worst case time complexity O(n^2).

1) Quick Sort
2) Bucket Sort

The traditional implementation of Quick Sort uses the Lomuto partition scheme: selecting a pivot (in this implementation, the rightmost element), rearranging the elements so that those less than or equal to the pivot come before it, and those greater than the pivot come after it. While this approach is more straightforward, it didn't perform optimally in cases where the array size was huge, as it led to highly unbalanced partitions. A variation of this approach, where emphasis is placed on the middle element as the pivot and the array is divided instead of partitioned, showed a significant improvement. Acknowledging that the original algorithm is rather underperforming, this paper uses the latter variation to enable a fair comparison with other sorting algorithms.

The native Bucket Sort algorithm employs fixed-sized buckets determined by dividing the maximum value by the length of the input list, distributing elements based on this fixed size, and then applying insertion sort within each bucket. This implementation performed badly for certain distributions (Test Case C in our case). A variation that dynamically adjusts bucket sizes according to the range of values present in the input array has resulted in better performance and has been considered in this paper.

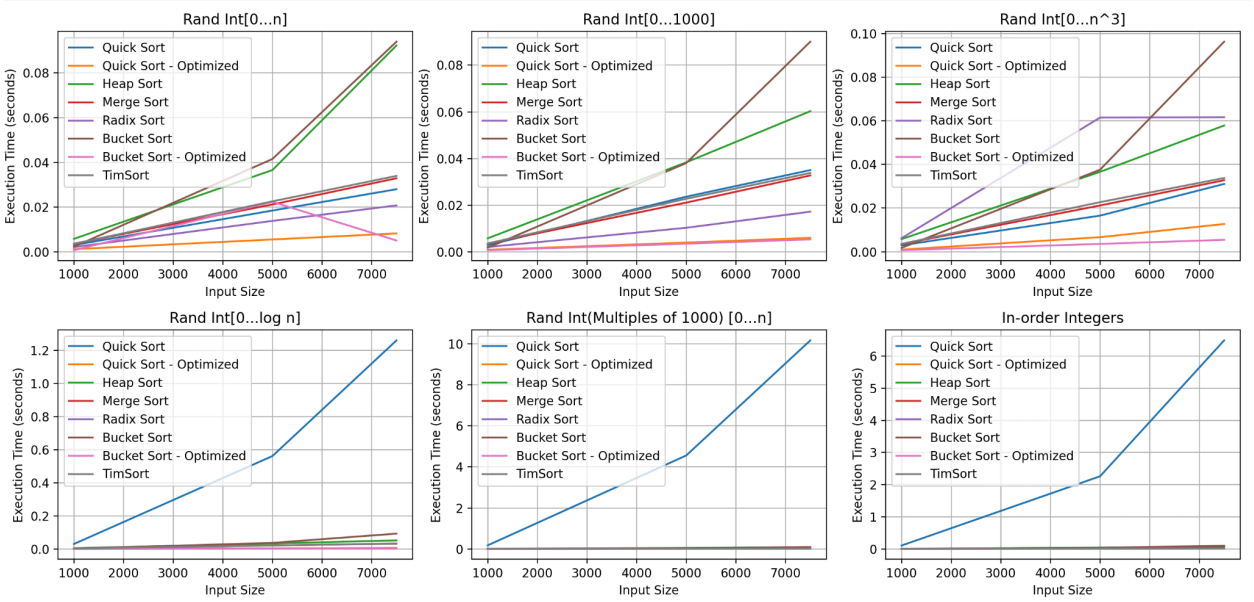

Figure 1: Comparison of execution times for all the sorting algorithms in six different distributions. The vanilla variations of Quick and Bucket Sorts are observed to be the slowest.

3.3. Observations

For input sizes ranging from 250k to 1M, the performance of the algorithms can be seen in the Figure 2 below.

The Bucket Sort variation we picked outperformed all other algorithms (~2 secs) in the majority of the Test Cases. One notable observation is that in Test Case C, where there were more unique values in the widely distributed dataset, the algorithm exhibited slower performance. This is due to the necessity for a greater number of buckets when handling a larger input range.

The Quick Sort variation we selected showed comparable performance to Bucket Sort. Its partitioning step often groups duplicate elements together, reducing the size of the subproblems. As a result, Quick Sort tends to perform better in scenarios with many duplicates compared to other sorting algorithms. Test Case D, where elements are distributed in a smaller range, further supports this observation, that Quick Sort performs well when the data is already somewhat sorted or when the range of elements is limited.

Tim Sort demonstrated steady performance (~4-6 secs) across all input sizes and Test Cases. This is mainly due to its hybrid nature, combining the benefits of Insertion sort and Merge Sort to optimize performance for different types of data distributions. This also explains why the time complexities of

Merge Sort and Tim Sort overlap for Test Cases A - D. For Test Cases E & F, the Tim Sort performs slightly better.

While Merge Sort often performs comparably to Tim Sort in various scenarios, it is not the optimal choice for handling nearly sorted data. This is evident in Test Case F, where the numbers are almost sorted, leading to poorer performance by Merge Sort.

Radix Sort performed well (~2 secs) in the cases where the largest value is small (Test Case B, D). In contrast it performed poorly (~7-13 secs) when the number of digits in the maximum value was significantly large (Test Case E) or when the range of values in the dataset was large (Test Case C). This is likely due to the increased memory consumption and longer processing times, as Radix Sort's efficiency relies on the number of digits in the largest number rather than the number of elements in the dataset.

Among all the algorithms, Heap Sort was observed to be the slowest in terms of time complexity (~12-15 secs). The reason for this can be attributed to the process of heapify that occurs at each iteration of the algorithm, which repeatedly swaps elements and adjusts the heap structure. Heap Sort is more suitable for scenarios where data structures are limited, such as embedded systems or environments with strict memory constraints.

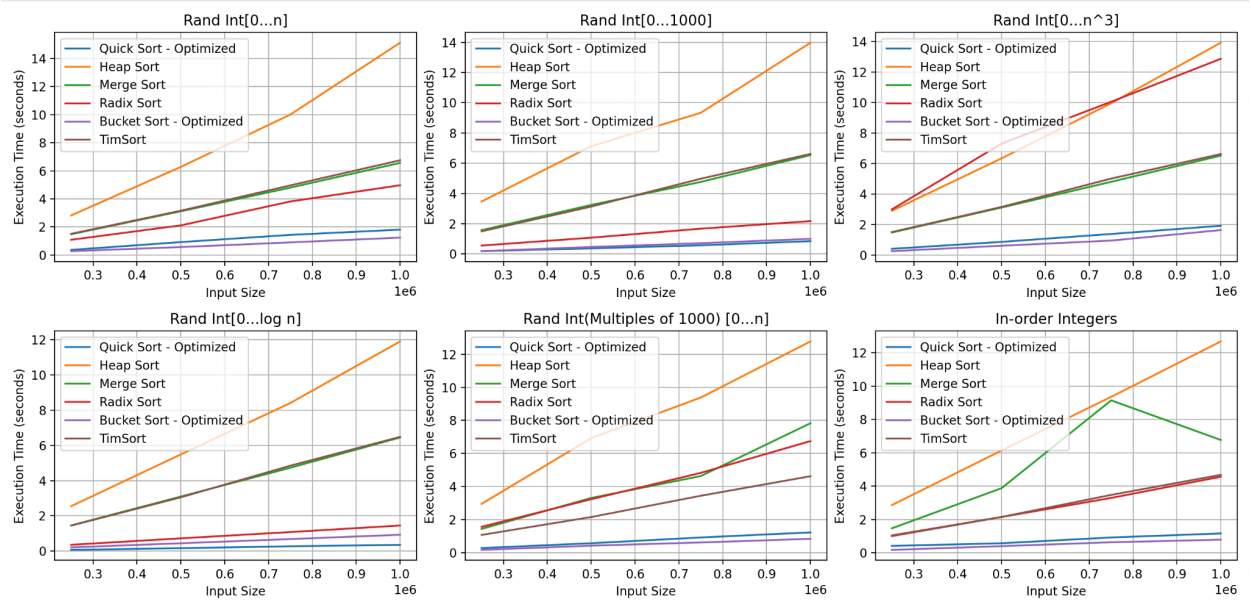

Figure 2: Comparison of execution times for all the sorting algorithms in six different distributions. The real-world variations of Quick and Bucket Sorts are observed to be the fastest among the other candidates.

3.4. Results

Case (n=250000)	Quick Sort	Heap Sort	Merge Sort	Radix Sort	Bucket Sort	Tim Sort
Rand Int[0...n]	0.364	2.823	1.481	1.085	0.273	1.511
Rand Int[0...1000]	0.186	3.466	1.571	0.551	0.18	1.492

Rand Int[0...n^3]	0.407	2.903	1.479	2.996	0.253	1.506
Rand Int[0...log n]	0.067	2.551	1.459	0.357	0.209	1.446
Rand Int(Multiples of 1000) [0...n]	0.27	2.945	1.428	1.556	0.158	1.064
In-order Integers	0.417	2.866	1.472	1.045	0.177	1.004

Case(n=500000)	Quick Sort	Heap Sort	Merge Sort	Radix Sort	Bucket Sort	Tim Sort
Rand Int[0...n]	0.92	6.277	3.115	2.111	0.566	3.158
Rand Int[0...1000]	0.363	7.123	3.24	1.072	0.459	3.13
Rand Int[0...n^3]	0.863	6.341	3.116	7.311	0.609	3.155
Rand Int[0...log n]	0.165	5.49	3.098	0.723	0.444	3.054
Rand Int(Multiples of 1000) [0...n]	0.562	6.925	3.304	3.227	0.414	2.147
In-order Integers	0.576	6.141	3.881	2.151	0.403	2.153

Case(n=750000)	Quick Sort	Heap Sort	Merge Sort	Radix Sort	Bucket Sort	Tim Sort
Rand Int[0...n]	1.434	10.01	4.803	3.816	0.899	4.962
Rand Int[0...1000]	0.568	9.337	4.761	1.669	0.709	4.974
Rand Int[0...n^3]	1.382	9.962	4.805	10.059	0.949	5.015
Rand Int[0...log n]	0.273	8.415	4.733	1.08	0.682	4.853
Rand Int(Multiples of 1000) [0...n]	0.911	9.385	4.63	4.825	0.608	3.438
In-order Integers	0.923	9.361	9.151	3.297	0.635	3.477

Case(n=1000000)	Quick Sort	Heap Sort	Merge Sort	Radix Sort	Bucket Sort	Tim Sort
Rand Int[0...n]	1.806	15.126	6.561	4.971	1.238	6.751
Rand Int[0...1000]	0.841	13.974	6.538	2.169	0.995	6.614
Rand Int[0...n^3]	1.922	13.934	6.524	12.878	1.637	6.629
Rand Int[0...log n]	0.353	11.91	6.453	1.45	0.926	6.486
Rand Int(Multiples of 1000) [0...n]	1.218	12.784	7.83	6.748	0.829	4.62
In-order Integers	1.163	12.688	6.774	4.568	0.792	4.681

4. SUMMARY

The paper thoroughly compares six common sorting algorithms: Quick Sort, Heap Sort, Merge Sort, Radix Sort, Bucket Sort, and Tim Sort. It dives into different input sizes and ranges to understand how these algorithms perform. It finds that each algorithm has its unique strengths and weaknesses, depending on the type and size of the data. To improve slower algorithms like Quick Sort and Bucket Sort, the study explores tweaked versions that show promising enhancements.

Interestingly, Bucket Sort shines when dealing with evenly spread-out data, while Quick Sort stands out in situations with many duplicate values. Merge Sort demonstrated consistent performance but showed limitations in efficiently handling nearly sorted data. On the other hand, Radix Sort performed well with small maximum values but struggled with large ranges, while Heap Sort, maintaining efficiency even in worst-case scenarios, proved slower in certain cases, making it less favorable for time-sensitive tasks. The paper also sheds light on the delicate balance between the time an algorithm takes and the memory it uses. The evaluation also delves into space and time complexity trade-offs, highlighting algorithms like Heap Sort, which offer space-efficient sorting solutions suitable for constrained environments.

References

[1] Sabah, A. S., Abu-Naser, S. S., Helles, Y. E., Abdallatif, R. F., Abu Samra, F. Y. A., Abu Taha, A. H., Massa, N. M., & Hamouda, A. A. (Department of Information Technology, Faculty of Engineering & Information Technology, Al-Azhar University - Gaza, Palestine). "Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics."

[2] Horsmalahti, P. (2012). Comparison of Bucket Sort and Radix Sort

[3] Sabah, A. S., Abu-Naser, S. S., Helles, Y. E., Abdallatif, R. F., Abu Samra, F. Y. A., Abu Taha, A. H., Massa, N. M., & Hamouda, A. A. (2020). Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics. International Journal of Computer Science and Information Security, 18(5)

[4] https://www.geeksforgeeks.org/sorting-algorithms/

[5] A Comparative Study of Various Sorting Algorithms International Journal of Advanced Studies of Scientific Research, Vol. 4, No. 1, 2019