International Organization of Research & Development (IORD)
ISSN: 2348-0831
Vol 13 Special Issue 02 | 2026

A Systematic Comparative Analysis of Classical and Linear Sorting Algorithms

Pallavi Kailas Ghanokar[1], Gayatri Ramdas Wanare[2], Prof. Shraddha Ratnaparkhi[3]
[1,2]*Student, Department of Computer Science and Engineering, Siddhivinayak Technical Campus, Shegaon, Maharashtra, India.*
[3]*Professor, Department of Computer Science and Engineering, Siddhivinayak Technical Campus, Shegaon, Maharashtra, India.*

DOI: 10.5281/zenodo.19539776

ABSTRACT

Sorting is one of the most fundamental computational operations and serves as a building block for numerous advanced algorithms in computer science. The performance of applications such as database indexing, searching, data mining, and analytics significantly depends on the efficiency of sorting mechanisms. Over the years, various sorting techniques have been proposed, each offering distinct advantages under specific input conditions.
This paper presents a detailed comparative and analytical study of major comparison-based and non-comparison-based sorting algorithms, including Insertion Sort, Merge Sort, Quick Sort, Heap Sort (tree-based implementation), Counting Sort, Radix Sort, and Bucket Sort. The algorithms are examined on the basis of theoretical time complexity, auxiliary space requirements, stability, adaptability, and practical applicability.
The study highlights that divide-and-conquer techniques provide reliable performance for large datasets, while linear-time sorting algorithms demonstrate superior efficiency when constraints on input range are satisfied. The objective of this review is to assist researchers and students in selecting appropriate sorting strategies based on problem requirements and computational limitations.

1.INTRODUCTION

Efficient data organization is essential in computing systems, and sorting is one of the most frequently executed operations in algorithm design. Sorting refers to arranging elements of a dataset into a specific order, typically ascending or descending. This ordered structure enhances the efficiency of searching and processing tasks.
In modern computing environments where large volumes of structured and unstructured data are processed, selecting an appropriate sorting technique is critical. The performance of a sorting algorithm depends on multiple factors including dataset size, memory availability, stability requirements, and data distribution characteristics.
Sorting algorithms are broadly categorized into:
Comparison-based algorithms
Non-comparison (distribution-based) algorithms
Internal sorting methods
External sorting methods
This research focuses on internal sorting techniques that operate entirely within main memory. The key objectives of this study are:

- To provide a structured overview of major sorting techniques.
- To compare theoretical computational complexities.
- To analyze memory consumption and stability characteristics.
- To identify optimal use cases for each algorithm.

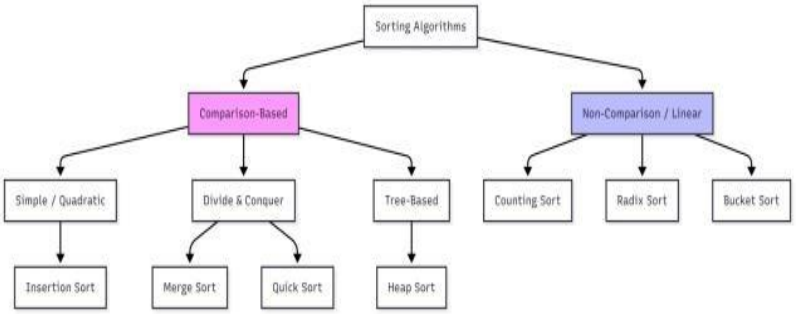

Fig 1: Hierarchy of Sorting Algorithms (Comparison vs non-comparison)"

2. LITERATURE REVIEW / RELATED WORK

Foundational research in algorithm theory establishes that comparison-based sorting algorithms cannot achieve better than O(nlog⁡n)O(n \log n)O(nlogn) complexity in the worst case. Classical textbooks and algorithmic studies have extensively analyzed divide-and-conquer methods such as Merge Sort and Quick Sort.

Recent academic comparisons emphasize that although Quick Sort may exhibit quadratic behavior in worst-case scenarios, its practical efficiency and cache-friendly nature make it one of the most widely used sorting algorithms. On the other hand, algorithms such as Counting Sort and Radix Sort surpass comparison-based lower bounds by utilizing distribution properties of data.

This paper consolidates theoretical insights and presents a comprehensive structured comparison.

3. ANALYSIS OF SORTING ALGORITHMS

3.1 Insertion Sort

Insertion Sort constructs the final sorted array incrementally by placing each element into its appropriate position relative to previously sorted elements. Repeat the following process: Begin with a prefix of the array containing just one element as a sorted array. From the remaining elements, choose the next one and find its position in the sorted array. Insert the element by moving the higher elements upward by one. The algorithm is implemented as C program named insert.c.

It can be seen that the insertion sort is $O(n^2)$ by analyzing the program using step counting. Intuitively, bubble sort and insertion sort get their $O(n^2)$ due to sequential nature of their attack on the problem: we have an outerloop that runs n steps and the cost of that loop ranges from 1 to n. If we are to break through $O(n^2)$ to a lower value for the upper bound, we must find an approach that is not sequential. Here is where the divide-and-conquer principle comes into use.

Characteristics:

- Adaptive behavior
- Stable sorting technique
- Efficient for small or nearly sorted inputs

Time Complexity:

- Best Case : O(n)O(n)O(n)
- Average Case: O(n2)O(n^2)O(n2)
- Worst Case : O(n2)O(n^2)O(n2)

3.2 Merge Sort

Merge Sort applies a recursive divide-and-conquer strategy. The input array is divided into smaller subarrays, sorted independently, and then merged systematically.

Our analysis of mergesort is as follows:

1. The time to merge a pair of lists to get an n element list is O(*n*) since each recur-sive call "retires" an element to the result list and each element of the result gets retired only once.
2. Thus, the time to merge pairs on a list of lists, the summed total length of which is n, is O(*n*), since from (1) the time to merge a single pair is proportional to the number of elements in the result.
3. The time to use repeat on a list of *n* 1-element lists is the number of times repeat is called recursively times O(*n*), from (2).
4. The number of times repeat is called is O(log *n*), since each call encounters a list which is about half as long as the previous call.
5. The time to mergesort an N element list is O(*n*) + time to repeat an N element list, since mapping an in element list is O(*n*). From (3) and (4), the time to repeat is O(*n* log *n*).

Therefore the overall time to mergesort an *n* element list is O(*n* log *n*).

Characteristics:

- Predictable performance
- Stable
- Requires auxiliary memory

TimeComplexity:

O(nlogn)O(n \log n)O(nlogn) in all cases

3.3 Quick Sort

Quick Sort selects a pivot element and partitions the dataset into two subsets around the pivot. Recursive application of this partitioning produces a sorted array.

The most popular divide-and-conquer sorting algorithm is quicksort. QuickSort is easier to state recursively:

QuickSort: Sorting by divide-and-conquer Basis: If the sequence consists of atmost one element, it is sorted. Recursion: Break a sequence of more than one element into two, as follows

1. Choose an element from the sequence as a pivot value. All elements less than the pivot value are selected as

sub-sequence L and all elements greater than or equal to the pivot value are selected as sub-sequence *R*.

2. Sort the sub-sequences *L* and R recursively. Then form the sequence consisting of *L* (sorted) followed by the pivot, followed by *R* (sorted).

A C program implementing this algorithm is given as ***qsort.c*** Under ideal circumstances, the dividing phase of quicksort will split the elements into two equal-length sub-sequences. In this case, there will be log n levels of recursive calls to quicksort. At each level, O(*n*) steps must be done to split the array. So the running time is of the order of O(*n* log *n*). Unfortunately, this is only in ideal circumstances, although by a probabilistic argument it also represents an average case performance under reasonable assumptions. The worst case performance, however, causes the array algorithm. the split unevenly, resulting in a worst case running time of O(n^2), which is the worst case for other sorting algorithms. In a worst-case sense, quicksort is not better than any other sorting algorithm.

Characteristics:

- In-place sorting
- High practical efficiency
- Not stable

Time Complexity:

- Average Case: O(nlogn)O(n \log n)O(nlogn)
- Worst Case: O(n2)O(n^2)O(n2)

3.4 Heap Sort (Tree-Based Approach)

Heap Sort utilizes a binary heap data structure, which is represented as a complete binary tree. The largest element is repeatedly extracted from the heap and placed in its correct position.

Tree Structuring Principle:

Rather than dealing with the sequence linearly, try to employ a tree structure to cut sequence traversal needs from in to log *n*.

How about doing insertions within a tree rather than in a linear array, as is done by the simple insertion sort? If there are in elements and we can do each insertion in O(log *n*) time, we might be able to achieve our goal. We have to work out the details concerning how the insertions can be done so as to maintain the balance of the tree.

Characteristics:

- Guaranteed worst-case performance
- In-place implementation
- Not stable

TimeComplexity:

O(nlogn)O(n\logn)O(nlogn)

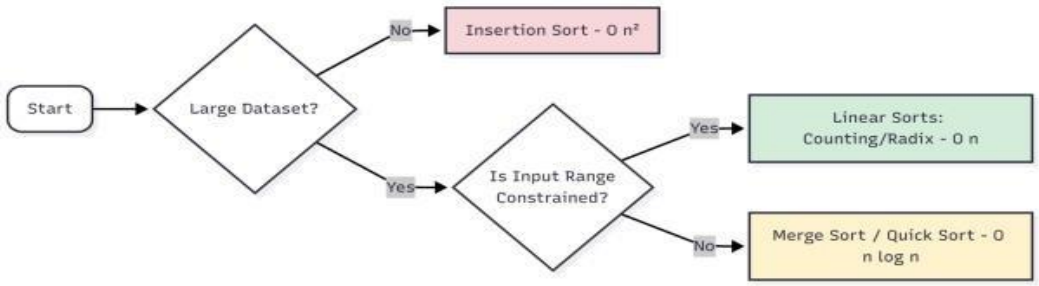

Fig., " Flowchart for Selecting an Appropriate Sorting Algorithm"

3.5 Counting Sort

Counting Sort determines the frequency of each distinct element within a specified range and uses this frequency to determine sorted positions.

This sorting method takes advantage of knowing the range of the numbers in the array to be sorted. Counting sort assumes that each of the items to be sorted is an integer in the ranges to s + k-1, of size k for some integer k. When k = O(*n*), the Counting-sort runs in O(*n*) time.

The basic idea of Counting sort is to determine, for each input item x, the number of items less than x. This information can be used to place x directly into its correct posi-tion. For example, if there are 13 items less than x, than x belongs to position 14 in the sorted array.

In a typical code for Counting sort, we are given an array A = a, ay..., a, of length, B = b[1] to b[n] which having items $s \leq a \leq s + k - 1$. We require two more arrays, an array holds the sorted output and an array C = c[1], c[2], ..., c[k] provides temporary working storage.

The algorithm is simple enough to be illustrated directly by a C code, as given Counting Sort

```
1.  int min, max, range;
2.  void counting_sort (int array[], int size){
3.  int i, j, z = 0, *count;
4.  min=max=array[0];
5.  for( i = 1; i < size; i++) {
6.  if (array[i] < min) min=array[i];
7.  else if (array[i] > max) max=array[i];
8.  }
9.  range = max-min+1;
10. count = (int*) malloc (range*sizeof ( int ) );
11. for (i = 0 ; i <  range; i++) count [i] = 0;
12. for (i =0; i < size ; i++) count[array[i] - min]++;
13. for (i = min ; i <=max; i++)
14. for (j = 0; j < count [i - min] ; j++)
15. array [ z ++] = i;
16. free (count);
17. }
18. int main(){
19. int i;
20. int a[10]= {23, 12, 31, 14, 5, 16, 9, 18, 27, 20};
21. counting sort (a, 10);
22. printf("min= %d, max = %d, range =  %d \n" , min, max, range );
23. for ( i = 0; i<10;i++) printf("%d", a[i]);
24. printf("\n");
25. }
```

With the input array A = (23, 12, 31, 14, 5, 16, 9, 18, 27, 20) we obtained the following output:

min = 5, max = 31, range = 27

5 9 12 14 16 18 20 23 27 31

We may dispense with lines 5, 6, 7 and 8 in the listing, which calculate the minimum maximum and the range of values in the input array, if these values are known otherwise. These lines take time less than O*(n)*, where *n=range* of the numbers. The line 12 take O*(n)* time .The time taken by the nested loops at lines 13,14,15 and 16 is O(*n*).Thus,this is a O*(n)*algorithm

Characteristics:

- Linear time performance
- Stable
- Requires known limited range

TimeComplexity:

O(n+k)O(n + k)O(n+k) where kkk is the range

3.6 Radix Sort

Radix Sort processes elements digit by digit, starting from the least significant digit to the most significant digit, using a stable intermediate sorting method.

Characteristics:

- Efficient for fixed-length integers
- Stable
- Requires additional memory

TimeComplexity:

O(nk)O(nk)O(nk)

3.7 Bucket Sort

Bucket Sort distributes elements into multiple buckets based on value intervals. Each bucket is sorted individually and then concatenated.

Characteristics:

- Highly efficient for uniformly distributed data

- Performance depends on distribution

Time Complexity:

- Average: O(n+k)O(n + k)O(n+k)
- Worst: O(n2)O(n^2)O(n2)

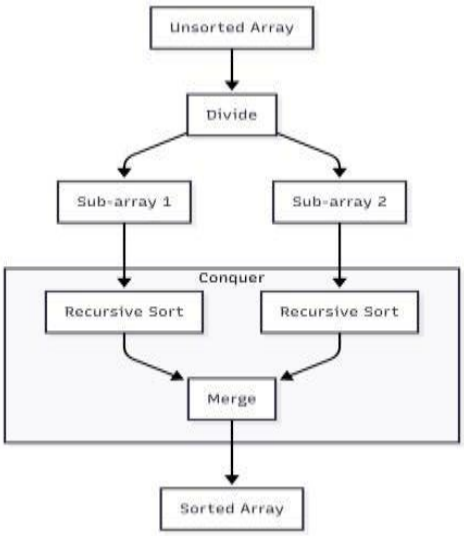

Fig., "Divide and Conquer Approach in Merge Sort"

4. COMPARATIVE EVALUATION

Table no.1: Comparative Evaluation

Algorithm	Best Case	Average Case	Worst Case	Space	Stable
Insertion	O(n)	O(n²)	O(n²)	Low	Yes
Merge	O(n log n)	O(n log n)	O(n log n)	High	Yes
Quick	O(n log n)	O(n log n)	O(n²)	Low	No
Heap	O(n log n)	O(n log n)	O(n log n)	Low	No
Counting	O(n+k)	O(n+k)	O(n+k)	High	Yes
Radix	O(nk)	O(nk)	O(nk)	High	Yes
Bucket	O(n+k)	O(n+k)	O(n²)	High	Depends

5. DISCUSSION

The comparative study indicates:

- Quadratic algorithms are suitable only for limited data volumes.
- Divide-and-conquer strategies ensure scalability.
- Distribution-based algorithms outperform comparison sorts when constraints are satisfied.
- Practical implementation factors such as cache performance and recursion overhead significantly influence real-world efficiency.
- Algorithm selection must therefore be problem-specific rather than universal.

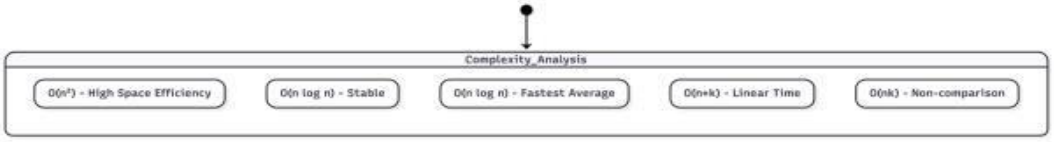

Fig. Complexity Analysis

6. CONCLUSION

This paper presented a systematic comparative evaluation of classical and linear sorting algorithms. The analysis confirms that no single algorithm is universally optimal. For large datasets requiring stable output, Merge Sort is recommended. Quick Sort is preferable when average performance is prioritized. Heap Sort ensures guaranteed time bounds. Counting, Radix, and Bucket Sort provide superior performance under constrained input conditions.

Future work may involve experimental benchmarking, hybrid algorithm development, and parallelized implementations for high-performance systems.

7. REFERENCES

[1] Cormen, T. H., et al., *Introduction to Algorithms*, MIT Press.
[2] Knuth, D. E., *The Art of Computer Programming, Vol. 3.*
[3] Aho, A. V., et al., *Data Structures and Algorithms*.
[4] Weiss, M. A., *Data Structures and Algorithm Analysis*.
[5] Sedgewick, R., *Algorithms*.
[6] IEEE Research Articles on Linear Sorting Techniques.
[7] Parag Himanshu Dave ,Himanshu Bhalchandra Dave, *Design And Analysis Of Algorithm.*