# Comparative Analysis and Implementation of Radix Sort and Tim Sort Algorithms


Yafiz Ibn Rahman
Roll: 2003003
Dept. Of CSE, RUET
2003003@student.ruet.ac.bd

Shihab Sarar Islam Rafid
Roll: 2003014
Dept. Of CSE, RUET
2003014@student.ruet.ac.bd

Tonmoy
Roll: 2003027
Dept. Of CSE, RUET
2003027@student.ruet.ac.bd

Md Basaratul Ferdaus
Roll: 2003021
Dept. Of CSE, RUET
2003021@student.ruet.ac.bd

Md Touhidul Islam
Roll: 2003051
Dept. Of CSE, RUET
2003051@student.ruet.ac.bd

Md. Fazle Rabbi Rithik
Roll: 2003057
Dept. Of CSE, RUET
2003057@student.ruet.ac.bd

Paperwork

LaTeX PDF

LaTeX Slide



***Abstract*—The goal of the comparative analysis and sorting algorithm implementation is to present two distinct algorithms that sort an array's items in place. Sorting is a powerful algorithm that has been used in numerous applications. Comparative analysis aids in our ability to arrange sequences and search for data processing, sequence merging, and search. An analogy has been drawn between this specific concept and its other well-known applications. During our program, we present two distinct methods that provide the optimal temporal complexity for sorting a single data set. It arranges the list of components so that, starting from any point and taking into account every nth component, the list is ordered. After doing the suggested tasks, we at last get in touch with the draw of a judgment, identify the circumstances in which this performs better, examine the limitations that are likely to arise, and outline the potential areas for future development.**



***Index Terms*—Radix Sort, Tim Sort, Algorithm Analysis, Sorting Algorithms, C++**


## I. Introduction

Sorting is a technique that falls into two categories: comparison sorting and non-comparison sorting technique. Sorting is the process of arranging information in a collection of sequences to lower the achievable cost of obtaining the data. The comparison sorting approach, which we will employ, just reads the list of components through one abstract comparison operation to assist us in determining that which, when the list is finally sorted, will be arranged in preference order first. Then, there is bitonic sorting, gnome sorting, shell sorting, radix sorting, and Tim sorting. The sorting algorithm's performance is measured in terms of existence complexities. A sorting algorithm is frequently categorized using terminology like internal sorting, which refers to a method where data is sorted directly in main memory. When the specified type of data is sorted in auxiliary memory, such as a hard drive, floppy disk, etc., the process is known as external sorting.The last type of complexity is computational complexity, which refers to the number of swaps each algorithm must perform in order to sort and obtain the desired result. System complexity also includes sorting techniques, such as when the algorithm we use can be classified based on performance in accordance with the time, such as worst cast, best case, and best scenario. Among the multitude of sorting algorithms, radix sort and Timsort stand out as notable approaches with distinct methodologies and performance characteristics.

## II. Background Study

The need for efficient data organization and retrieval has been a longstanding challenge in computer science. Sorting algorithms aim to arrange data in a specified order, such as numerical or lexicographical, optimizing search and retrieval operations. Over time, diverse sorting algorithms have been developed, each tailored to specific data structures, sizes, and computational constraints.

Radix sort, a non-comparative integer sorting algorithm, and Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, have gained prominence due to their efficiency and adaptability in various scenarios. These algorithms address different sorting challenges, offering unique advantages in terms of time complexity, space utilization, and stability.

### A. Objectives of the Study

[1] This research aims to conduct an in-depth analysis and comparative study of radix sort and Timsort, exploring their methodologies, performance metrics, strengths, weaknesses, and practical applications. The primary objectives include:

1) **Comprehensive Understanding:** Provide a detailed overview and explanation of radix sort and Timsort algorithms, delving into their inner workings, core principles, and operational procedures.

2) **Performance Evaluation:** Conduct a comparative analysis of these algorithms, evaluating their time and space complexities, stability, adaptability to different data types, and real-world performance.
3) **Practical Implications:** Examine the practical applications and scenarios where radix sort and Timsort excel, offering insights into their optimal utilization based on specific data characteristics and computational requirements.

### *B. Radix Sort*

[2]

1) **Algorithmic Overview :** Radix sort is a non-comparative integer sorting algorithm that operates on the principle of sorting elements based on individual digits or radix positions within the keys. It categorizes elements into buckets according to each significant digit, sorting them progressively from the least significant digit (rightmost) to the most significant digit (leftmost). The process continues until all digits are considered, resulting in a fully sorted sequence.
2) **Implementation and Workflow :**
   a) Initialization: Determine the maximum number of digits among the elements to be sorted.
   b) Bucket Creation: Create buckets corresponding to each digit (0-9) for integer-based data.
   c) Sorting Passes: Perform iterations, sorting elements into the respective buckets based on the current digit's value.
   d) Merge or Concatenate Buckets: Combine the buckets into a single sequence after each pass.
   e) Repeat: Continue the sorting process for each subsequent digit until the entire sequence is sorted.
   f) Here is the algorithm :

**Algorithm 1** Radix Sort Algorithm

```
function GetMax(arr, n)
    max ← arr[0]
    for i ← 1 to n − 1 do
        if arr[i] > max then
            max ← arr[i]
        end if
    end for
    return max
end function
function CountSort(arr, n, exp)
    output[n] , count[10] ← {0}
    for i ← 0 to n − 1 do
        count[(arr[i]/exp)mod10] + +
    end for
    for i ← 1 to 9 do
        count[i] ← count[i] + count[i − 1]
    end for
    for i ← n − 1 downto 0 do
        output[count[(arr[i]/exp)mod10] − 1] ← arr[i]
        count[(arr[i]/exp)mod10] − −
    end for
    for i ← 0 to n − 1 do
        arr[i] ← output[i]
    end for
end function
function RadixSort(arr, n)
    m ← GetMax(arr, n)
    for exp ← 1 while m/exp > 0 do exp ← exp · 10 do
        CountSort(arr, n, exp)
    end for
end function
```

3) **Time and Space Complexity Analysis :**
   a) **Time Complexity:** Radix sort exhibits linear time complexity of O(n * k), where n represents the number of elements and k is the average number of digits in the elements. It operates effectively on fixed-length integer keys, making it particularly efficient for sorting large datasets with limited key sizes.
   b) **Space Complexity:** The space complexity of radix sort is O(n + k), where n is the number of elements and k represents the number of buckets used. The algorithm requires additional space for creating and managing buckets.
4) **Strength and Limitations :**
   a) **Strengths:**
      i) Linear time complexity, making it efficient for certain datasets with fixed key sizes.
      ii) Well-suited for integer-based data or keys with multiple radix positions.
      iii) Stable sorting algorithm, preserving the order of equal elements.
   b) **Limitations:**
      i) Limited applicability to non-integer or variable-length keys.
      ii) Requires additional space proportional to the number of buckets.
      iii) Slower for small key sizes or when the number of digits is significantly large.

### *C. Tim Sort*

[3]

1) **Algorithmic Overview :** Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort, specifically designed to optimize performance for various data distributions. It efficiently handles real-world data sets by exploiting patterns like runs or partially ordered subsequences within the input sequence.
2) **Implementation and Workflow :**
   a) Identifying Runs: Timsort starts by identifying and categorizing runs - ascending or descending sequences within the input data.

b) Merge of Runs: It then merges these runs using a modified merge sort approach, combining them into larger sorted subarrays.
c) Insertion Sort Optimization: For smaller subarrays or runs, it switches to insertion sort due to its efficiency on smaller data sets.
d) Merging Runs: Continuously merges adjacent runs until the entire sequence is sorted.
e) Here is the algorithm :

**Algorithm 2** Tim Sort Algorithm

```
function InsertionSort(arr, left, right)
    for i ← left + 1 to right do
        temp ← arr[i]
        j ← i − 1
        while j ≥ left and arr[j] > temp do
            arr[j + 1] ← arr[j]
            j ← j − 1
        end while
        arr[j + 1] ← temp
    end for
end function
function Merge(arr, l, m, r)
    len1 ← m − l + 1
    len2 ← r − m
    left[len1], right[len2]
    for i ← 0 to len1 − 1 do
        left[i] ← arr[l + i]
    end for
    for i ← 0 to len2 − 1 do
        right[i] ← arr[m + 1 + i]
    end for
    i ← 0, j ← 0, k ← l
    while i < len1 and j < len2 do
        if left[i] ≤ right[j] then
            arr[k] ← left[i]
            i ← i + 1
        else
            arr[k] ← right[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i < len1 do
        arr[k] ← left[i], k ← k+1, i ← i+1
    end while
    while j < len2 do
        arr[k] ← right[j], k ← k + 1, j ← j + 1
    end while
end function
function TimSort(arr, n)
    RUN ← 32
    for i ← 0 to n by RUN do
        InsertionSort(arr, i, min(i + RUN − 1, n − 1))
    end for
    for size ← RUN to n by 2 · size do
        for left ← 0 to n by 2 · size do
            mid ← left + size − 1
            right ← min(left+2·size−1, n−1)
            if mid < right then
                Merge(arr, left, mid, right)
            end if
        end for
    end for
end function
```

3) **Time and Space Complexity Analysis :**
   a) **Time Complexity:** Timsort typically exhibits time complexity close to O(n log n) for average cases. However, its adaptive nature allows it to perform exceptionally well on partially ordered or already sorted data, achieving linear time complexity in best-case scenarios.
   b) **Space Complexity:** Timsort has a space complexity of O(n), requiring additional memory for the merging process and auxiliary arrays used during sorting.
4) **Strength and Limitations :**
   a) **Strengths:**
      i) Highly adaptive and efficient on various data distributions, including partially ordered sequences
      ii) Combines the strengths of merge sort and insertion sort, optimizing performance for diverse scenarios.
      iii) Offers stable sorting, preserving the original order of equal elements.
   b) **Limitations:**
      i) Requires additional memory for auxiliary arrays, impacting space complexity.
      ii) Although generally efficient, it might not always outperform specialized algorithms tailored to specific data distributions.
      iii) Implementation complexity due to its hybrid nature and intricate algorithms for run detection and merging.

## III. Comparative Analysis: Radix Sort vs. Timsort

### A. Time Complexity Comparison

1) **Radix Sort :** The time complexity of radix sort is O(n * k), where n represents the number of elements and k denotes the average number of digits or radix positions. It remains linear but is sensitive to the number of digits, which might impact performance on large datasets with varying digit lengths.
2) **Tim Sort :** Timsort typically exhibits a time complexity of O(n log n) for average cases. However, its adaptability allows it to achieve linear time complexity (O(n)) in

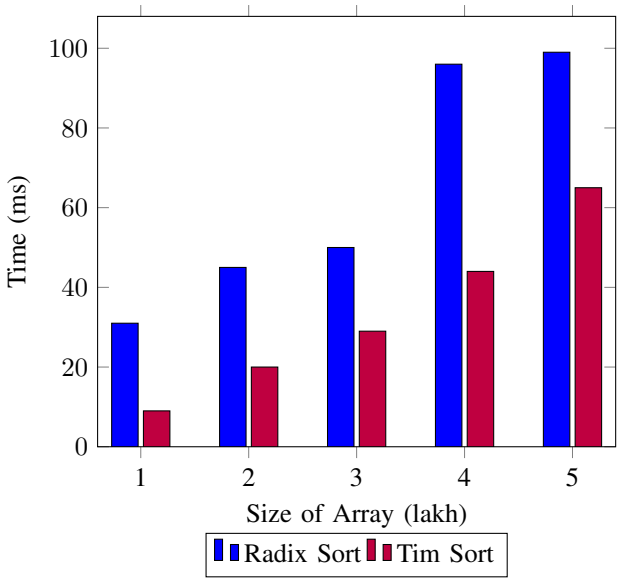


Fig. 1: Comparison of Radix Sort and Tim Sort

best-case scenarios, especially on partially ordered or already sorted data.

### B. Space Complexity Comparison

1) **Radix Sort :** The space complexity [3] of radix sort is O(n + k), where n is the number of elements and k represents the number of buckets used. It necessitates additional space for creating and managing buckets, which might impact its efficiency in memory-constrained environments.
2) **Tim Sort :** Timsort has a space complexity of O(n), requiring additional memory for the merging process and auxiliary arrays. While it also consumes extra space, the amount required is generally proportional to the input size and less than radix sort in typical scenarios.

### C. Stability and Adaptability

1) **Radix Sort :** Radix sort is inherently stable, meaning it preserves the relative order of equal elements. However, its adaptability is limited to scenarios involving fixed-length integer keys or elements with known radix positions.
2) **Tim Sort :** Timsort is stable, ensuring the preservation of the original order of equal elements. Its adaptability is notable, especially when handling diverse data distributions or partially ordered sequences, where its adaptive nature shines.

### D. Performance on Different Data Distributions

[2]

1) **Radix Sort :** Radix sort performs optimally on fixed-length integer keys, demonstrating efficiency when the number of digits is limited and relatively uniform. However, its performance might degrade with variable-length keys or non-integer data.
2) **Tim Sort :** Timsort excels in handling various data distributions, performing well on partially ordered, reverse-ordered, or already sorted data. Its adaptive nature allows it to adjust its performance based on the characteristics of the input sequence.

## IV. Conclusion

Both radix sort and Timsort offer unique advantages and are suitable for specific use cases. Radix sort excels in scenarios with fixed-length integer keys, demonstrating linear time complexity but is limited in adaptability to variable-length keys. In contrast, Timsort's adaptability and efficiency across diverse data distributions make it a popular choice for general-purpose sorting tasks, especially when the nature of the data is unpredictable or partially ordered. Choosing between the two algorithms depends on the specific characteristics of the data and the performance requirements of the task at hand.

## V. References

## References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
[2] C. Thomas H, E. Charles, R. Ronald L, S. Clifford, *et al.*, "Introduction to algorithms third edition," 2009.
[3] R. Sedgewick and K. Wayne, *Algorithms*. Addison-wesley professional, 2011.