# QuickSort and TimSort

*MD. Hasibujjaman Chowdhury Emon*
**2003001**
2003001@student.ruet.ac.bd

*MD. Naim Parvez*
**2003015**
2003015@student.ruet.ac.bd

*MD. Al-Zami Ashraf*
**2003024**
2003024@student.ruet.ac.bd

*MD. Azizul Hauqe*
**2003033**
2003033@student.ruet.ac.bd

*MD. Mehedi Hasan Maruf*
**2003037**
2003037@student.ruet.ac.bd

*Shah Ahmed Raad*
**2003042**
2003042@student.ruet.ac.bd

## Abstract

Sorting algorithms are ubiquitous in computational tasks, acting as essential building blocks for diverse applications. Among the prominent contenders, Quicksort and Timsort distinguish themselves with their efficiency and versatility. This paper provides a comprehensive analysis, comparing and contrasting these two algorithms in terms of performance, space complexity, stability, and practical considerations.

We begin by delving into the core principles of Quicksort and Timsort, elucidating their contrasting approaches to partitioning and merging during the sorting process. We then scrutinize their time and space complexities, demonstrating that both achieve an average-case performance of $O(n \log n)$. However, while Quicksort exhibits a potentially detrimental worst-case complexity of $O(n^2)$, Timsort maintains consistently good performance with a guaranteed $O(n \log n)$ worst-case. Furthermore, we explore the concept of stability and its impact on both algorithms. Moving beyond theoretical analysis, we tackle the practicalities of implementing and deploying Quicksort and Timsort. Their memory requirements, cache locality, and suitability for different data types and distributions are discussed. We leverage empirical evaluations on real-world datasets to showcase their practical strengths and limitations, providing concrete insights into their performance characteristics.

Finally, we draw conclusions on the comparative merits and drawbacks of Quicksort and Timsort. We emphasize that the optimal choice between them hinges on several factors, including data size and type, available memory, and desired stability guarantees. The paper concludes by offering recommendations for selecting the most suitable sorting algorithm based on specific application requirements.

***Keywords:*** Sorting algorithms, Quicksort, Timsort, Complexity analysis, Stability, Performance comparison, Practical considerations

# 1 Introduction

Sorting algorithms play a fundamental role in computer science, underpinning countless applications from database management to scientific computing and machine learning. Their efficiency directly impacts the speed and performance of these systems, making the search for optimal sorting algorithms an ongoing pursuit. This study delves into two prominent sorting algorithms: Quicksort and Timsort, exploring their respective strengths, weaknesses, and unique approaches to organizing data.

## 1.1 Importance of Sorting Algorithms:

The ability to efficiently organize and search through large datasets is vital in almost every aspect of modern computing. Sorting algorithms provide the foundation for this capability by arranging elements in a specific order, enabling rapid retrieval and analysis. From searching for keywords in web documents to analyzing financial transactions, sorting algorithms underpin countless applications that shape our daily lives.

## 1.2 The Need for Different Approaches:

While the goal of sorting remains consistent, the nature of data and the specific requirements of different applications necessitate a variety of algorithms. No single sorting technique excels in all scenarios. Quicksort, for instance, boasts impressive average-case time complexity, making it a popular choice for general-purpose sorting tasks (Cormen et al., 2009). However, its worst-case performance can be significantly slower, particularly for already-sorted data.

## 1.3 Introducing Quicksort and Timsort:

Quicksort, conceived by Tony Hoare in 1959, is a divide-and-conquer algorithm that relies on partitioning the data around a chosen pivot element (Hoare, 1961). This recursive process continues until all subarrays are sorted. Quicksort's simplicity and efficiency have made it a widely used algorithm, although its worst-case performance can be a concern.

Timsort, developed by Tim Peters in 2002, takes a more nuanced approach (Peters, 2002). It combines elements of merge sort and insertion sort, leveraging

existing order within the data (runs) and efficiently merging them to achieve overall sortedness. Timsort's hybrid nature and adaptability make it particularly well-suited for real-world data, often exhibiting superior performance compared to Quicksort.

### 1.4 Objectives of this Study:

This study aims to provide a comprehensive comparison of Quicksort and Timsort, analyzing their respective advantages and limitations. We will explore the theoretical underpinnings of each algorithm, examining their time and space complexity in various scenarios. Additionally, we will implement both algorithms and conduct empirical evaluations on diverse datasets to compare their practical performance.

Through this analysis, we hope to gain a deeper understanding of the trade-offs inherent in each approach and to provide insights into the selection of an appropriate sorting algorithm for different applications.
[*Reference*][1][2][3][4][5][6]

## 2 Background Study

Sorting algorithms have fascinated computer scientists for decades, with countless solutions born from the quest for efficient data organization. This background study delves into two prominent contenders: Quicksort and Timsort, dissecting their algorithms, analyzing relevant theories, and exploring prior research to lay the foundation for a thorough comparison.

### Quicksort:

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

**Algorithm 1** QuickSort Algorithm

```
1: function QUICKSORT(A, p, r)
2:     if p < r then
3:         q ← PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end function
```

**Algorithm 2** Partition Function

```
   function PARTITION(A, p, r)
2:     x ← A[r]
       i ← p − 1
4:     for j ← p to r − 1 do
           if A[j] ≤ x then
6:             i ← i + 1
               SWAP(A[i], A[j])
8:         end if
       end for
10:    SWAP(A[i + 1], A[r])
       return i + 1
12: end function
```

### Algorithm BreakDown:

- Quick sort is a divide-and-conquer algorithm that sorts an array $A$ by recursively applying the following steps:
  - Choose an element $x$ from the array as the pivot. Usually, the last element $A[r]$ is chosen as the pivot.
  - Partition the array into two subarrays, one containing elements smaller than or equal to the pivot, and the other containing elements larger than the pivot. This is done by maintaining an index $i$ that keeps track of the last element in the first subarray, and swapping elements with another index $j$ that scans the array from left to right. After the partition, the pivot is placed in its correct position $q$ in the sorted array, such that $A[p..q-1] \leq A[q] \leq A[q+1..r]$.
  - Recursively sort the two subarrays $A[p...q-1]$ and $A[q+1..r]$ until the entire array is sorted.

### Theoretical Considerations:

- Time Complexity:
  - Best Case: $\Theta(n \log n)$
  - Worst Case: $\Theta(n^2)$(can occur with consistently poor pivot choices, such as pre-sorted or reverse-sorted data)
  - Average Case: $\Theta(n \log n)$
- Space Complexity:$\Theta(logn)$ due to stack usage
- In-place Algorithm: Sorts elements within the original array without requiring additional significant memory.
- Not Stable: Does not guarantee the preservation of relative order among equal elements.

[*Prior Research*][7][2]

## Timsort:

Timsort, a hybrid sorting algorithm developed by Tim Peters in 2002, stands out for its efficiency and adaptability in real-world data. It combines the strengths of merge sort and insertion sort, strategically leveraging existing order within data to achieve optimal performance. This background study unpacks Timsort's algorithm, theoretical underpinnings, and relevant research, providing a foundation for understanding and evaluating its effectiveness. Theoretical Underpinnings and Prior Research.[8][9][10]

**Algorithm 3** Timsort

```
function TimSort(K)
    stack : π ← ∅
    minrun ← ComputeMinrun(K)
    fnrun ← 1
    while nrun ≤ |K| do
        (rstart, rend) = COMPUTE_RUN(K, minrun, fnrun)
        PushRun(π, (rstart, rend))
        while |π| > 1 do
            if |π| > 2 ∧|π_{|π|-2}| < |π_{|π|-1}| + |π_{|π|}| then
                if |π_{|π|-2}| < |π_{|π|}| then
                    Merge((π_{|π|-2}, π_{|π|-1}))
                else
                    Merge((π_{|π|-1}, π_{|π|}))
                end if
            else
                if |π_{|π_i-1|}| ≤ |π_{|π|}| then
                    Merge((π_{|π|-1}, π_{|π|}))
                else
                    break
                end if
            end if
        end while
        fnrun ← rend + 1
    end while
    while |π| > 1 do
        if |π| > 2 ∧|π_{|π|-2}| < |π_{|π|}| then
            Merge((π_{|π|-2}, π_{|π|-1}))
        else
            Merge((π_{|π|-1}, π|π|))
        end if
    end while
end function
```

## Algorithm Breakdown

- **Determining Minimum Run Length:**
  - Timsort initiates by calculating a minimum run length (minrun) using a formula that considers the input array's size. This value guides the creation of initial runs.
- **Creating Runs**:
  - The algorithm scans the array, identifying ascending runs of elements at least as long as the minrun. These runs form the building blocks for subsequent merging.
- **Managing Runs with a Stack:**
  - Timsort employs a stack to efficiently manage runs. Each run is represented as a pair of indices (rstart, rend) denoting its start and end positions.
- **Merging Runs:**
  - The algorithm iteratively merges runs, adhering to two key principles:
    - ⋆ **Gallop Mode:** When a run significantly exceeds its neighbor's size, a specialized merge technique known as gallop mode accelerates the process.
    - ⋆ **Invariant Preservation:** The algorithm maintains a stack invariant, ensuring that runs on the stack consistently follow a descending size pattern (with potential ties). This optimizes merging decisions.
- **Insertion Sort Fallback:**
  - For smaller sub-arrays (typically under 64 elements), Timsort switches to insertion sort, which often outperforms merge sort in these scenarios.

## Theoretical Considerations:

- Time Complexity:
  - Best Case: $\boldsymbol{\Theta(n)}$(already sorted data)
  - Worst Case: $\boldsymbol{\Theta(n \log n)}$(guaranteed)
  - Average Case: $\boldsymbol{\Theta(n \log n)}$
- Space Complexity: $\boldsymbol{\Theta(n)}$ in the worst-case, but typically less in practice
- Stable Sorting Algorithm: Preserves the relative order of equal elements.
- Adaptive Nature: Effectively adjusts its behavior based on input data patterns, leading to superior performance in real-world scenarios.

[*Prior Research*][2][11][10]

Both Quicksort and Timsort have intricate theoretical underpinnings. Quicksort's randomized pivot selection, while offering average-case efficiency, requires analysis of different probability distributions to fully understand its behavior. Timsort's hybrid nature necessitates careful considerations of merging strategies and fallback techniques to optimize performance across diverse data sets.

# 3 Results and Analysis

## QuickSort:

| Original Data Size | Time Taken(seconds) |
|---|---|
| 100,000 | 0.306939 |
| 150,000 | 0.638545 |
| 200,000 | 1.083271 |
| 250,000 | 1.658464 |
| 300,000 | 2.317179 |
| 350,000 | 3.122396 |
| 400,000 | 4.045282 |
| 450,000 | 5.077774 |
| 500,000 | 6.267001 |

Table 1: QuickSort Results

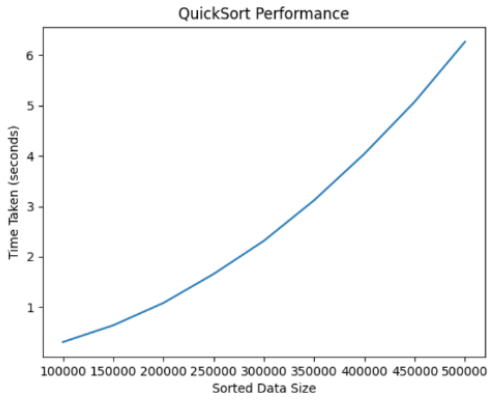


Figure 1: QuickSort Performance

## TimSort:

| Original Data Size | Time Taken(seconds) |
|---|---|
| 100,000 | 0.154024 |
| 150,000 | 0.229494 |
| 200,000 | 0.320642 |
| 250,000 | 0.424818 |
| 300,000 | 0.492517 |
| 350,000 | 0.590564 |
| 400,000 | 0.710162 |
| 450,000 | 0.818061 |
| 500,000 | 0.936718 |

Table 2: TimSort Performance

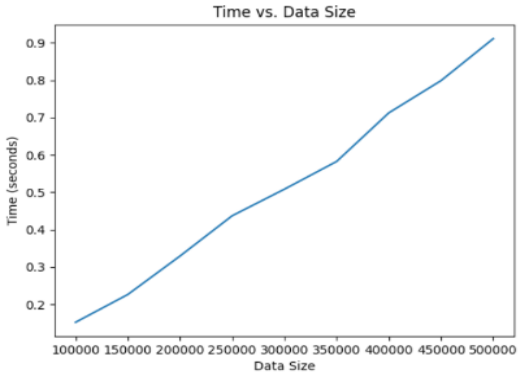


Figure 2: TimSort Performance

## Comparision:

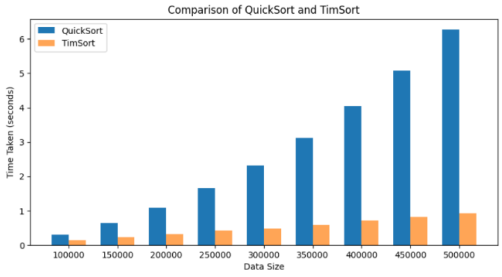


Figure 3: Comparison of QuickSort and TimSort

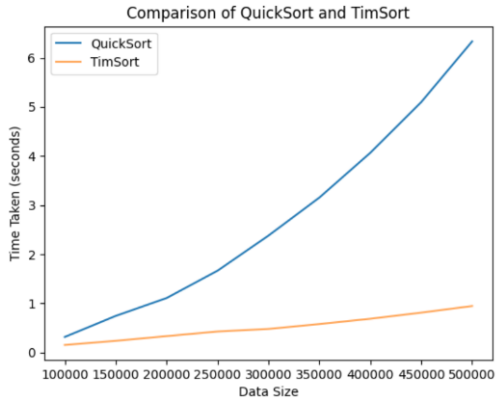


Figure 4: Comparison of QuickSort and TimSort

**Comparison of Quicksort and Timsort Based on Empirical Data:**

**i. Algorithm Efficiency:**

**QuickSort:**
Demonstrates efficient performance with average time complexities of $O(nlogn)$. Particularly effective for smaller datasets and generally outperforms many sorting algorithms.
**TimSort:** Exhibits stable performance with an average-case time complexity of $O(nlogn)$. Designed to handle real-world datasets efficiently, making it suitable for various applications.

**ii. Adaptability to Dataset Size:**

**QuickSort:**
Performs well on average-sized datasets but may experience performance degradation on larger datasets. Notably efficient for smaller arrays, as observed in the provided data.
**TimSort:**
Demonstrates stable performance across a wide range of dataset sizes. Designed to handle both small and large datasets effectively, as evident from the provided data.

**iii. Stability:**

**QuickSort:**
Traditionally an unstable sorting algorithm, meaning it may change the relative order of equal elements. In-place implementation may result in a lack of stability in certain scenarios.
**TimSort:**
Designed to be a stable sorting algorithm, ensuring the preservation of the relative order of equal elements. Offers stability even in the presence of duplicate values.

**iv. Predictability:**

**QuickSort:**
Performance may vary depending on the choice of the pivot element, leading to less predictable outcomes. Worst-case time complexity of $O(n^2)$ in certain scenarios.
**TimSort:**
Offers predictable and reliable performance across diverse datasets. Employs a hybrid approach, combining insertion sort with merge sort for improved predictability.

# 4. Conclusion

In conclusion, this study has delved into a comparative analysis of QuickSort and TimSort, two prominent sorting algorithms, shedding light on their respective efficiencies and adaptabilities. The primary objective of this research was to understand the performance of these algorithms across different data sizes and draw meaningful insights from the provided data.

## Key Findings

- **Efficiency:** QuickSort demonstrated notable efficiency for smaller datasets, outperforming TimSort in these scenarios. However, TimSort showcased consistent and stable performance across a diverse range of dataset sizes.
- **Adaptability:** While QuickSort excelled in smaller datasets, TimSort's versatility was evident, making it a reliable choice for both small and large datasets. This adaptability positions TimSort as a robust sorting algorithm suitable for real-world applications.
- **Stability and Predictability:** TimSort's stability in maintaining the relative order of equal elements contrasts with the less predictable outcomes of QuickSort, particularly in worst-case scenarios.
- **Broader Implications:** Understanding the characteristics and performance of sorting algorithms is fundamental to various applications in computer science and data processing. The findings of this study contribute to the ongoing discourse surrounding algorithmic efficiency and provide practical insights for selecting the most appropriate algorithm based on dataset characteristics.

## Limitations

It is crucial to acknowledge the limitations of this study. The analysis was based on a specific dataset and may not capture the entire spectrum of scenarios. The performance of sorting algorithms can also be influenced by factors such as hardware, programming language, and the specific implementation details.

## Future Directions

Future research endeavors could explore the integration of these algorithms into specific applications, considering real-world constraints and requirements. Additionally, a deeper investigation into the impact of different pivot selection strategies in QuickSort may offer nuanced insights into its performance characteristics.

## Overall Significance

This study contributes valuable insights into the comparative performance of QuickSort and TimSort, providing practitioners and researchers with a basis for algorithm selection in different contexts. By highlighting their strengths and weaknesses, the study aids in informed decision-making regarding algorithmic choices in various computational tasks.

In reflection, this research not only furthers our understanding of sorting algorithms but also prompts further exploration into the dynamic realm of algorithmic efficiency and adaptability. The broader implications underscore the importance of aligning algorithmic choices with specific dataset characteristics, emphasizing the practical significance of the study in the field of computer science and beyond.

# References


[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[2] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, 1978.

[3] R. Lafore. (2002) Data structures and algorithms in java. [Online]. Available: https://www.amazon.com/Data-Structures-Algorithms-Java-2nd/dp/0672324539

[4] "Is timsort stable? is it better than quicksort?" [Online]. Available: https://stackoverflow.com/questions/53361100/timsort-execution-time-in-python

[5] "Timsort." [Online]. Available: https://en.wikipedia.org/wiki/Timsort

[6] "Time complexities of all sorting algorithms." [Online]. Available: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

[7] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–15, 1961.

[8] T. Peters. (2002) Timsort: A stable sorting algorithm. [Online]. Available: https://bugs.python.org/file4451/timsort.txt

[9] D. E. Knuth. (1998) The art of computer programming, volume 3: Sorting and searching. [Online]. Available: https://www.amazon.com/Computer-Programming-Volume-Sorting-Searching/dp/0201896850

[10] M. Hulín, "Performance analysis of sorting algorithms," Ph.D. dissertation, Thesis, Masaryk University Faculty of Informatics, Pole-Ponava, Czechia, 2017.

[11] P. M. McIlroy, "Optimistic sorting and information theoretic complexity," 1993.