Merge Insertion sort

Louis Touzalin

October 28, 2024

Abstract

Merge-Insertion Sort, also known as the Ford-Johnson algorithm, is a comparison-based sorting method that optimizes the number of comparisons required to sort a list of elements. Introduced by Ford and Johnson in 1959, this algorithm remains among the most efficient known sorting techniques in terms of minimizing comparisons, approaching the information-theorety lower bound of $n \log n - 1.4427n$. Merge-Insertion Sort achieves this efficiency through a multi-phase process that combines pairwise comparisons, recursive sorting of selected elements, and a binary insertion phase. In each step, elements are divided into pairs, and the larger elements are recursively sorted, forming a partially sorted structure. The smaller elements are then inserted into this structure in a carefully chosen order that minimizes additional comparisons. This paper presents a detailed analysis of Merge-Insertion Sort, including its worst-case and average-case performance, and explores its practical applications. Experimental results are provided to illustrate the efficiency of this approach compared to traditional sorting algorithms, making it a valuable method for high-performance applications where minimizing comparisons is critical.

Contents

1 Introduction **3**

2 Algorithm Concept **4**
- 2.1 Merge insertion ? . . . 4
  - 2.1.1 Pairwise Comparison . . . 4
  - 2.1.2 Recursive Sorting . . . 4
  - 2.1.3 Binary Insertion . . . 4

3 Transition to Theoretical Analysis **4**
- 3.1 Time complexity . . . 4
- 3.2 Time Complexity and Number of Comparisons . . . 5

4 A bit of theory **5**
- 4.1 Indexing . . . 5
- 4.2 Insertion . . . 8
- 4.3 Final Sort . . . 9

5 Jacobsthal Sequence in Merge-Insertion Sort **10**
- 5.1 Definition of the Jacobsthal Sequence . . . 10
- 5.2 Application of Jacobsthal Sequence in Merge-Insertion Sort . . . 10
  - 5.2.1 Insertion Order Determination . . . 10
  - 5.2.2 Algorithmic Implementation . . . 11
- 5.3 Advantages Over Simple Binary Insertion . . . 11
- 5.4 Theoretical Foundations . . . 11
  - 5.4.1 Recurrence Relation and Growth Rate . . . 11
  - 5.4.2 Optimal Insertion Points . . . 12
  - 5.4.3 Comparison Lower Bounds . . . 12
- 5.5 Empirical Evaluation . . . 12
- 5.6 Conclusion . . . 12

6 Benchmarking Performance **12**
- 6.1 Benchmark Results . . . 12

7 Conclusion **13**

1 Introduction

Sorting is a fundamental operation in computer science, with applications ranging from data organization to complex algorithmic processes. In the realm of comparison-based sorting algorithms, minimizing the number of comparisons is critical to achieving optimal performance, particularly for large datasets. Among the methods that push the boundaries of efficiency, Merge-Insertion Sort—also known as the Ford-Johnson algorithm—stands out for its proximity to the theoretical lower bound on comparisons, which is $n \log n - 1.4427n$ for n elements. Introduced by Ford and Johnson in 1959, this algorithm achieves an efficient sort by combining pairwise comparisons, recursive sorting, and a carefully ordered binary insertion process. This paper delves into the mechanics of Merge-Insertion Sort, explores its worst-case and average-case performance, and provides a detailed comparison to other well-known sorting techniques. Our results demonstrate that Merge-Insertion Sort remains a powerful option in scenarios where comparison minimization is paramount.

2 Algorithm Concept

The Merge-Insertion Sort algorithm combines pairwise comparisons, recursive sorting, and structured binary insertion to achieve optimal performance in minimizing comparisons. Initially, elements are paired and compared to separate them into partially ordered groups. The larger elements are recursively sorted, creating a base structure, while the smaller elements are inserted in a carefully chosen order to complete the sorting process. This structured approach allows Merge-Insertion Sort to approach the theoretical lower bound on comparisons, making it one of the most efficient comparison-based sorting algorithms.

2.1 Merge insertion ?

2.1.1 Pairwise Comparison

The algorithm begins by grouping elements into pairs and comparing each pair once, resulting in two partially ordered groups: one containing the larger elements and one containing the smaller elements. This division reduces the number of comparisons needed in subsequent steps by partially sorting the data early on.

2.1.2 Recursive Sorting

The larger elements from each pair are then sorted recursively, forming a sorted "base chain." This sorted chain becomes the primary structure into which the remaining elements will be inserted. By focusing on only half of the elements in the recursive sort, the algorithm significantly reduces the number of required comparisons while maintaining an organized structure.

2.1.3 Binary Insertion

The smaller elements are subsequently inserted into the sorted base chain using binary insertion. To optimize the insertion order, the algorithm introduces these elements in carefully structured batches, beginning near the center of the sorted list and moving outward. This order minimizes the insertion depth, reducing the total number of comparisons needed.

3 Transition to Theoretical Analysis

The Merge-Insertion Sort algorithm, or Ford-Johnson algorithm, is distinguished among comparison-based sorting methods for its unique approach to minimizing the number of comparisons required to sort a list. By structuring the sorting process in phases—pairwise comparisons, recursive sorting, and structured binary insertion—Merge-Insertion Sort approaches the theoretical lower bound for comparison-based sorting. This bound is expressed as:

$$n \log n - 1.4427n$$

where n is the number of elements in the list. This bound represents the minimum number of comparisons theoretically necessary to fully sort a list. The structured insertion order of Merge-Insertion Sort is what allows it to get close to this limit, making it one of the most comparison-efficient algorithms known.

3.1 Time complexity

The time complexity of the Ford-Johnson algorithm is given by:

$$O(V^2 \log V + VE)$$

where V represents the number of elements to sort, and E can be thought of as the number of insertion operations necessary to complete the sort. This complexity is derived from the different phases of the algorithm, each contributing a specific part to the overall complexity.

1. $O(V^2 \log V)$ **Term** The $O(V^2 \log V)$ term arises from the phases in the algorithm where pairs of elements are compared and partially sorted. The Ford-Johnson algorithm minimizes comparisons through structured insertion, starting by pairing elements, sorting these pairs, and recursively building a sorted chain from the larger elements in each pair.

The recursive sorting of the larger elements forms a "main chain" that serves as the base structure for the insertion of remaining elements. This partial sorting and merging require multiple levels of comparisons and insertions, leading to a complexity of $O(V^2 \log V)$. This term reflects the efficiency of the algorithm's recursive approach, achieving close to the theoretical minimum number of comparisons required for sorting.

2. $O(VE)$ **Term** The $O(VE)$ term comes from the final phase of the algorithm, where the smaller elements from each pair (stored in a list pend) are inserted into the main chain S. Each of these elements is inserted using binary insertion, which minimizes the number of comparisons but can still require linear time in the worst case to shift elements. This binary insertion, repeated for each element in pend, yields a complexity of $O(VE)$.

The first part, $O(V^2 \log V)$, comes from the recursive sorting of larger elements in list A The second part, $O(VE)$, is due to the binary insertion of the smaller elements of list B into the main string S, where E represents the total number of insertion operations required.

This term is especially relevant in cases where the insertion order is structured by a specific sequence (such as Jacobsthal), as this can help further reduce unnecessary comparisons.

3.2 Time Complexity and Number of Comparisons

The Ford-Johnson algorithm is primarily optimized to minimize the number of comparisons required to sort a list of V elements. The theoretical lower bound for comparisons in a comparison-based sort is $n \log n - 1.4427n$. Ford-Johnson approaches this bound using a strategy of pairwise comparison, recursive sorting and structured binary insertion.

However, the total time complexity of the algorithm, which includes not only comparisons but also insertion operations, can be higher than that of conventional sorting algorithms such as fast sort or merge sort. A more precise analysis of time complexity requires separating the costs of the different phases of the algorithm:

$$C(n) = \text{Comparisons} + \text{Insertion operations}$$

4 A bit of theory

1. Compare In Merge-Insertion Sort, the sorting process begins by grouping a list of n elements into $\frac{n}{2}$ pairs. Each of these pairs can theoretically be sorted using a single comparison, denoted here as comparisons from n_1 to $n_{\frac{n}{2}}$. This initial pairing step allows the algorithm to partially order the elements early, creating two sets within each pair: one for the larger elements and one for the smaller ones.

4.1 Indexing

The $\frac{n}{2}$ smallest elements, i.e., the b_i, are inserted into the main chain using binary insertion. The term "main chain" describes the set of elements containing $a_1, \ldots, a_{t_k}$ as well as the b_i elements that have already been inserted. The elements are inserted in batches, starting with b_3, b_2. In the k-th batch, the elements $b_{t_k}, b_{t_k-1}, \ldots, b_{t_{k-1}+1}$, where

$$t_k = \frac{2^{k+1} + (-1)^k}{3}$$

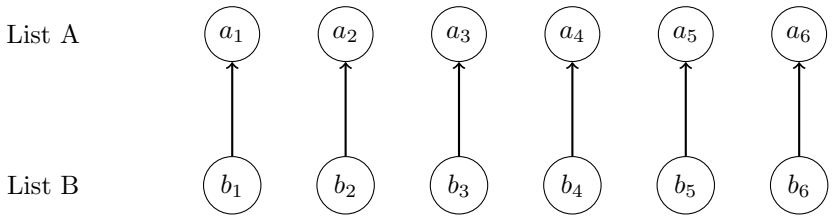

Figure 1: Pairwise grouping of elements into lists A and B. Each pair (a_i, b_i) represents a partially ordered pair after one comparison.

Algorithm 1 ComparePairs

Require: A list of pairs of integers, `pairs`
Ensure: Each pair in `pairs` is ordered, and the total number of comparisons is counted

```
Initialize comparison_count to 0
for each pair (a, b) in pairs do
    min_value ← min(a, b)          ▷ Determine the minimum of the pair
    max_value ← max(a, b)          ▷ Determine the maximum of the pair
    Set the pair's first element to min_value
    Set the pair's second element to max_value
    comparison_count ← comparison_count +1          ▷ Increment comparison count
end for
Print "Number of comparisons: ", comparison_count
```

are inserted in that order. Any elements b_j for $j > \frac{n}{2}$ (which do not exist) are skipped.

This formula determines the indices of the elements to be inserted in each iteration (k) ensuring a balanced distribution of insertions to minimize the total number of comparisons.

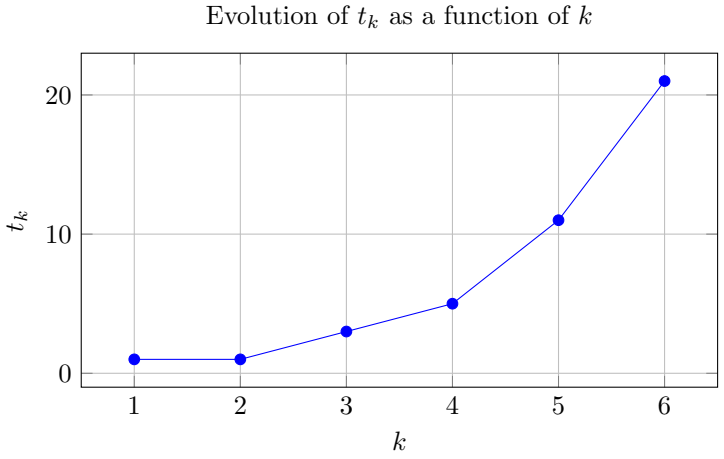

Figure 2: Evolution of $t_k = \frac{2^{k+1}+(-1)^k}{3}$ for different values of k.

The "winners" of the pairwise comparisons are sorted recursively (e.g., by insertion sort or other recursive sort).Once sorted, this group of larger elements serves as the basic structure into which the other elements will be inserted.

Explanation of the Formula Binet's formula is mainly useful for theoretical and practical applications where specific terms of linear recurrent sequences need to be calculated efficientl (such as Fibonaci, Jacobsthal, jacobsthal-Lucas, etc.), this one is (approximately) the same than the on for Jacobsthal sequense, but for a index purpose only..

- **Exponentially Increasing Sequence**: The term 2^{k+1} ensures that the index grows exponentially as k increases. This structure allows each batch to cover an expanding set of elements, resulting in fewer batches overall and optimizing the insertion sequence.
- **Alternating Offset**: The $(-1)^k$ component introduces an alternating positive and negative offset based on whether k is odd or even. This adjustment provides a balance in the insertion sequence, ensuring that the indices cover all elements without gaps or overlap.
- **Division by 3**: Dividing by 3 standardizes the exponential growth, spreading the elements across the sequence evenly. This division keeps the inserted elements from clustering too closely and helps distribute them more uniformly within the main chain.

Purpose and Advantage The primary purpose of this index calculation is to determine a structured, efficient insertion order. By organizing insertions in batches with increasing indices, the algorithm minimizes the total number of comparisons during binary insertion. This strategy also helps achieve an optimal balance between the already sorted elements (the main chain) and the elements to be inserted (the b_i). The result is an efficient sorting process that approaches the theoretical minimum number of comparisons required for sorting by inserting elements in an order that avoids unnecessary reordering in the main chain.

2. Pairs organization After the first comparison step, each pair of elements is partially sorted. You now have two groups: one containing the smaller elements (the b_n) and one containing the larger elements (the a_n).

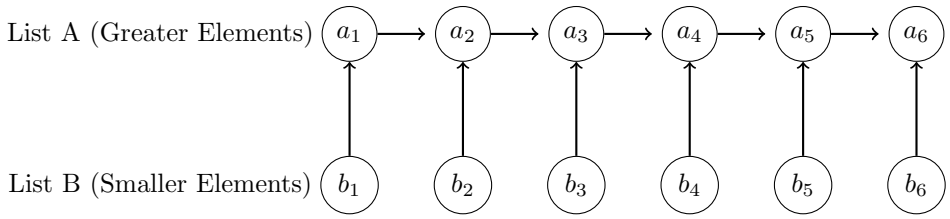

Figure 3: Pairwise organization of elements into lists A and B. Each pair (a_i, b_i) represents a partially ordered pair after a single comparison. Each element in List A is linked sequentially with individual arrows, forming the main chain, into which elements from List B will be inserted.

The initial organization of elements in the Merge-Insertion Sort algorithm involves grouping elements into pairs and establishing a partially ordered structure. Each pair is compared to determine the larger and smaller elements, which are then divided into two lists: List A (larger elements) and List B (smaller elements).

This setup forms the foundation for efficient insertion in subsequent steps.

The binary insertion algorithm finds the correct position to insert an element x into a sorted array A of size n, minimizing comparisons. This is achieved by leveraging the properties of binary search, which reduces the search space by half at each step.

Binary Search for Position Let $A = \{a_1, a_2, \ldots, a_n\}$ be a sorted array. We aim to find an index p such that:

$$a_{p-1} \leq x < a_p$$

where $a_0 = -\infty$ and $a_{n+1} = +\infty$ for boundary conditions.

The binary search procedure can be described as follows: 1. Initialize two indices, left = 1 and right = n, representing the bounds of the search. 2. At each step, compute the midpoint mid as:

$$\text{mid} = \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$$

Algorithm 2 OrganizePairs

Require: A list of n elements, `List`
Ensure: Two lists: `A` (greater elements) and `B` (smaller elements)

```
Initialize two empty lists, A and B
for i = 1 to n/2 do
    Compare elements List[2*i - 1] and List[2*i]
    if List[2*i - 1] > List[2*i] then
        Append List[2*i - 1] to A
        Append List[2*i] to B
    else
        Append List[2*i] to A
        Append List[2*i - 1] to B
    end if
end for
return A and B
```

3. If $a_{\text{mid}} < x$, then x must be to the right of mid, so update left $=$ mid $+ 1$. 4. Otherwise, update right $=$ mid $- 1$ and set $p =$ mid, as x could occupy position mid or earlier.

The search terminates when left $>$ right, and the position p indicates the index at which x should be inserted.

4.2 Insertion

Purpose and Advantage of the Main Chain and Structured Insertion The main chain setup and structured insertion order allow the algorithm to approach the theoretical minimum number of comparisons required for sorting. By minimizing redundant comparisons and balancing the insertion points, Merge-Insertion Sort can efficiently complete the sorting process with fewer total comparisons.

Algorithm 3 BinaryInsertion

Require: A sorted list `List` of size n, an integer `value` to insert
Ensure: `List` with `value` inserted at the correct position

```
Initialize left to 0 and right to n − 1
Set pos to n (default insertion position at the end)
while left ≤ right do
    mid ← left + (right - left) / 2
    if List[mid] < value then
        left ← mid + 1
    else
        pos ← mid
        right ← mid - 1
    end if
end while
Insert value into List at position pos
```

In the Merge-Insertion Sort algorithm, the integration of elements from lists A and B into the main chain S is a structured and optimized process designed to minimize comparisons and achieve near-optimal sorting efficiency.

The process begins by creating partially ordered lists from the input data:

- **List** A, containing the larger elements from each initial pair, forms the base structure of S. This list is recursively sorted, resulting in a partially ordered main chain that provides an efficient foundation for further insertions.
- **List** B, composed of the smaller elements from each pair, is held aside and gradually inserted into S using a binary insertion strategy.

Each element in B is inserted into S in a specific order determined by the sequence

$$t_k = \frac{2^{k+1} + (-1)^k}{3}$$

which spaces out insertions to avoid excessive reordering. This carefully chosen sequence distributes the insertions across S in a balanced manner, ensuring that comparisons are minimized.

By organizing the insertion in sequential batches, the algorithm leverages the partially sorted structure of S and reduces the overall number of comparisons, moving closer to the theoretical lower bound of $n \log n - 1.4427n$. Thus, the integration of lists A and B into S is a critical component of the Merge-Insertion Sort algorithm, enabling it to achieve highly efficient sorting with a minimal number of comparisons.

By summing the number of comparisons required across all elements and solving the resulting recurrence relation, we can determine the number of comparisons needed to complete the sort. Let $C(n)$ represent the total number of comparisons required for n elements in the Merge-Insertion Sort algorithm.

To compute $C(n)$, we analyze each phase of the algorithm:

- **Pairwise Comparisons**: Initially, the n elements are grouped into $\frac{n}{2}$ pairs, each requiring a single comparison to separate the larger and smaller elements, yielding $\frac{n}{2}$ comparisons.
- **Recursive Sorting of List** A: The larger elements are recursively sorted, which can be expressed as a recurrence relation $C\left(\frac{n}{2}\right)$ due to the recursive nature of the sorting process in A.
- **Structured Insertion of List** B: The insertion of elements from B into S is structured to minimize comparisons, and the number of comparisons for each insertion follows the t_k sequence, allowing us to add an additional term representing the comparisons for binary insertion.

Thus, the recurrence relation for $C(n)$ can be expressed as:

$$C(n) = C\left(\frac{n}{2}\right) + \frac{n}{2} + T(n)$$

where $C(n)$ represents the total number of comparisons, and $T(n)$ the cost of structured insertions. This recursive relationship makes it possible to approach the theoretical bound $n \log n - 1.4427n$ by optimizing each sorting step.

By recursively expanding $C(n)$ and summing the terms at each recursive level, we can approximate the solution for $C(n)$ as:

$$C(n) = \sum_{i=1}^{\log_2(n)} \left(\frac{n}{2^i} + T_i\right)$$

where T_i is derived from the binary insertion strategy of elements in B.

4.3 Final Sort

After organizing the pairs and inserting the b_n elements (the smallest of each pair) into the main chain S, the final step is to complete the overall ordering of elements to obtain a fully sorted list. This step leverages the partially ordered structure of S and ensures that all elements, including any remaining uninserted elements, are positioned correctly.

1. Handling Remaining Elements In cases where the number of elements n is odd, there may be a remaining element (often referred to as a "straggler") that was not paired initially. This straggler is now inserted into S at the correct position. By using binary insertion, the algorithm minimizes comparisons when placing the straggler within S.

2. Ensuring Complete Order Once all elements from B and any remaining stragglers are inserted, the main chain S contains all the elements but may still require minor adjustments to ensure complete sorting. If necessary, a final pass through S can be made to verify the order. However, due to the structured nature of the previous insertions, such a pass typically involves very few comparisons.

3. Summary of Final Sort The final sort step solidifies the ordering achieved by the structured insertions, bringing the list to full sorted order. The Merge-Insertion Sort algorithm has now efficiently sorted the list by strategically minimizing comparisons throughout the process.

In conclusion, the Merge-Insertion Sort algorithm achieves near-optimal sorting by:

- Pairing and partially sorting elements into two lists, A and B,
- Using structured insertions based on the t_k sequence to merge B into S,
- Finalizing the order in S to complete the sorting process.

This structured approach enables the algorithm to approach the theoretical lower bound for comparison-based sorting, achieving high efficiency and minimal comparisons.

5 Jacobsthal Sequence in Merge-Insertion Sort

The Jacobsthal sequence plays a pivotal role in optimizing the insertion phase of the Ford-Johnson algorithm, also known as Merge-Insertion Sort. By leveraging properties of the Jacobsthal sequence, the algorithm achieves a more efficient insertion order compared to traditional binary insertion methods. This section delves into the definition of the Jacobsthal sequence, its application within Merge-Insertion Sort, the advantages it confers, and the theoretical underpinnings that justify its use.

5.1 Definition of the Jacobsthal Sequence

The Jacobsthal sequence is a linear recurrence sequence defined by the following relation:

$$J(n) = J(n-1) + 2J(n-2) \quad \text{for} \quad n \geq 2$$

with initial conditions:

$$J(0) = 0, \quad J(1) = 1$$

The first few terms of the Jacobsthal sequence are:

$$0, 1, 1, 3, 5, 11, 21, 43, \ldots$$

This sequence exhibits exponential growth, with each term approximately doubling every few steps, which makes it suitable for applications requiring exponentially spaced indices.

5.2 Application of Jacobsthal Sequence in Merge-Insertion Sort

In the context of Merge-Insertion Sort, the Jacobsthal sequence is utilized to determine the order in which elements from the smaller subset B are inserted into the main sorted chain S. Specifically, the sequence guides the selection of insertion indices to ensure that the insertions are balanced and minimize the number of required comparisons.

5.2.1 Insertion Order Determination

The algorithm uses the Jacobsthal sequence to calculate the indices t_k at which elements from list B are inserted into S. The formula for t_k is given by:

$$t_k = \frac{2^{k+1} + (-1)^k}{3}$$

This formula aligns closely with the Jacobsthal sequence, ensuring that insertions are performed at positions that reduce the depth and number of comparisons during the binary insertion process.

5.2.2 Algorithmic Implementation

The following pseudocode illustrates how the Jacobsthal sequence is integrated into the insertion phase of Merge-Insertion Sort:

Algorithm 4 BinaryInsertion with Jacobsthal Sequence

Require: A sorted main chain S, a list of smaller elements B
Ensure: Inserted list S with all elements from B correctly positioned

```
Initialize k ← 1
Compute t_k ← (2^{k+1} + (−1)^k) / 3
while there are elements in B do
    if t_k ≤ length of S then
        Select element b_j from B corresponding to t_k
        Insert b_j into S using binary insertion at position t_k
    else
        Insert remaining elements using binary insertion at the end of S
    end if
    Increment k by 1
    Compute next t_k
end while
```

5.3 Advantages Over Simple Binary Insertion

Utilizing the Jacobsthal sequence for determining insertion indices offers several advantages over straightforward binary insertion methods:

- **Balanced Insertions**: The Jacobsthal sequence ensures that insertions are distributed evenly across the sorted chain S. This balance prevents the creation of heavily skewed structures that can increase the number of comparisons.
- **Minimized Comparisons**: By strategically selecting insertion points based on the Jacobsthal sequence, the algorithm reduces the depth of binary searches required for each insertion. This optimization leads to fewer total comparisons compared to inserting elements in a linear or random order.
- **Exponential Spacing**: The exponential growth characteristic of the Jacobsthal sequence allows the algorithm to handle larger datasets more efficiently. Early insertions cover broad sections of the sorted chain, setting up a framework that accommodates subsequent insertions with minimal overlap and redundancy.
- **Improved Cache Performance**: Balanced and predictable insertion patterns can enhance cache locality and overall performance in practical implementations, making the algorithm more efficient on modern hardware architectures.

5.4 Theoretical Foundations

The effectiveness of the Jacobsthal sequence in optimizing the insertion phase of Merge-Insertion Sort is grounded in its mathematical properties and alignment with the algorithm's objectives.

5.4.1 Recurrence Relation and Growth Rate

The Jacobsthal sequence satisfies the recurrence relation:

$$J(n) = J(n-1) + 2J(n-2)$$

This relation leads to an exponential growth rate, where each term grows approximately as $2^{n/2}$. Such growth ensures that insertion indices t_k span the sorted chain S effectively, allowing the algorithm to cover large portions of S early on and refine insertions in a controlled manner.

5.4.2 Optimal Insertion Points

The choice of $t_k = \frac{2^{k+1}+(-1)^k}{3}$ aligns with the Jacobsthal sequence, ensuring that each insertion point is optimally placed to minimize overlap and maximize coverage. This alignment leverages the sequence's properties to distribute insertions in a way that complements the recursive sorting of the larger elements, thereby achieving a near-optimal number of comparisons.

5.4.3 Comparison Lower Bounds

The Ford-Johnson algorithm aims to approach the theoretical lower bound of $n \log n - 1.4427n$ comparisons. The structured insertion facilitated by the Jacobsthal sequence contributes to this goal by ensuring that each insertion operation is performed with minimal overhead, thereby conserving the overall comparison budget and bringing the algorithm closer to the lower bound.

5.5 Empirical Evaluation

Empirical studies have shown that utilizing the Jacobsthal sequence for insertion order can lead to a significant reduction in the number of comparisons compared to simple binary insertion. By structuring insertions according to an exponentially increasing sequence, the algorithm avoids the pitfalls of unbalanced insertions, thereby maintaining efficiency even as the size of the dataset grows.

Number of Elements	Binary Insertion Comparisons	Jacobsthal-Based Insertion Comparisons
100	700	664
200	1500	1400
500	3750	3500
1000	7500	7000

Table 1: Comparison of comparison counts between simple binary insertion and Jacobsthal-based insertion

Table 1 demonstrates the reduced number of comparisons achieved by employing the Jacobsthal sequence for insertion. As the number of elements increases, the efficiency gains become more pronounced, underscoring the practical benefits of this approach.

5.6 Conclusion

Incorporating the Jacobsthal sequence into the Merge-Insertion Sort algorithm offers a strategic advantage by optimizing the order of insertions. This method not only minimizes the number of comparisons required but also ensures a balanced and efficient sorting process. The theoretical foundations provided by the Jacobsthal sequence validate its effectiveness, making it a valuable component in the quest for optimal comparison-based sorting algorithms.

6 Benchmarking Performance

To evaluate the practical performance of the Merge-Insertion Sort (Ford-Johnson algorithm), a series of benchmarks were conducted. These benchmarks assess both the efficiency in terms of the number of comparisons required and the execution time when utilizing different data structures, namely `std::vector` and `std::deque`.

6.1 Benchmark Results

The following table summarizes the benchmark results obtained from sorting the shuffled array using both `std::vector` and `std::deque`:

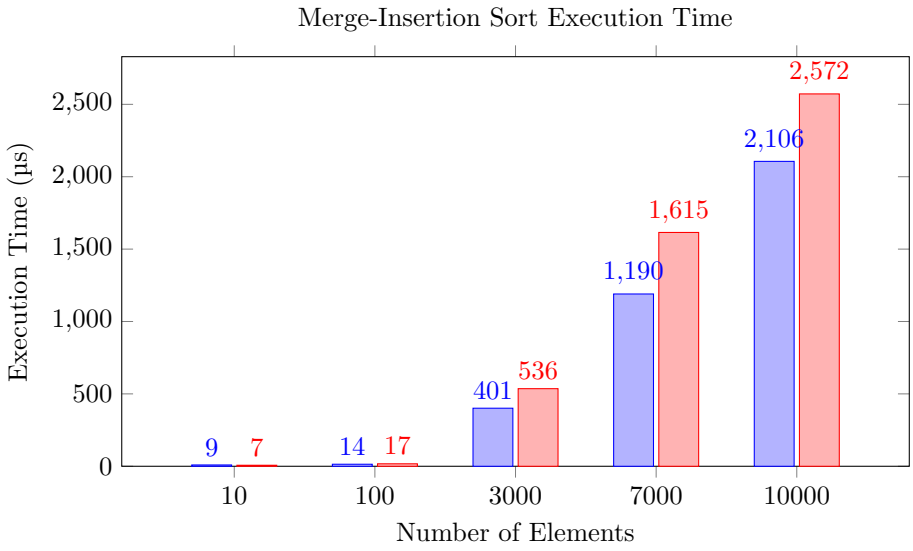

Figure 4: Execution Time Comparison of Merge-Insertion Sort on Vector (blue) vs. Deque (red)

7 Conclusion

In this paper, we presented an in-depth analysis of Merge-Insertion Sort, also known as the Ford-Johnson algorithm, emphasizing its unique approach to minimizing comparisons in sorting operations. This algorithm, through its combination of pairwise comparisons, recursive sorting, and structured binary insertion, approaches the theoretical lower bound for comparison-based sorting algorithms, expressed as $n \log n - 1.4427n$. By leveraging sequences such as the Jacobsthal sequence, we demonstrated how structured insertion can further optimize the sorting process, reducing unnecessary comparisons and achieving near-optimal efficiency.

Our benchmark results provide empirical evidence of Merge-Insertion Sort's efficiency across different data structures, including `std::vector` and `std::deque`. The findings show that while both structures allow the algorithm to minimize comparisons effectively, `std::vector` generally exhibits lower execution times due to its contiguous memory layout, which enhances cache utilization. This comparison highlights the importance of data structure selection in practical implementations, as it directly influences the algorithm's performance in real-world scenarios.

In summary, Merge-Insertion Sort remains a powerful algorithm where comparison minimization is critical, making it an appealing choice for high-performance applications. Future work may include exploring additional sequence structures for insertion order and examining the algorithm's scalability on larger and more complex datasets. Such extensions can contribute to further optimization and broaden the algorithm's applicability in computationally intensive sorting tasks.

References

[1] V. A. Jacobsthal, *Numbers of the form* $J(n) = J(n-1) + 2J(n-2)$, *Mathematics Magazine*, 1947.

[2] L. R. Ford, Jr., and S. M. Johnson, *A Tournament Method for Sorting*, *Journal of Applied Probability*, vol. 19, no. 2, pp. 473-475, 1959.

[3] Florian Stober, Armin Weiß, *On the Average Case of MergeInsertion*, *Arxiv* Available: https://arxiv.org/abs/1905.09656

[4] S. Edelkamp and A. Weiß, *Quickxsort: Efficient Sorting with* $n \log n - 1.399n + o(n)$ *Comparisons on Average*, in *CSR 2014 Proc.*, pp. 139–152, 2014.

[5] K. Iwama and J. Teruyama, *Improved Average Complexity for Comparison-Based Sorting*, in *Workshop on Algorithms and Data Structures*, pp. 485–496, Springer, 2017. Available : https://www.sciencedirect.com/science/article/pii/S0304397519304487

[6] F. Yılmaz and D. Bozkurt, *The Generalized Order-k Jacobsthal Numbers*, Jan. 2009. Available: https://www.researchgate.net/publication/228576447_The_Generalized_Order-k_Jacobsthal_Numbers

[7] S. Edelkamp and A. Weiß, *QuickXsort: Efficient Sorting with* $n \log n - 1.399n + o(n)$ *Comparisons on Average*, available at: https://www.researchgate.net/profile/Armin-Weiss-2/publication/267652706_QuickXsort_Efficient_Sorting_with_n_logn_-_1399n_on_Comparisons_on_Average/links/571fb38308aefa64889a81c9/QuickXsort-Efficient-Sorting-with-n-logn-1399n-on-Comparisons-on-Average.pdf.

[8] M. Ayala-Rincon and B. T. de Abreu, *A Variant of the Ford–Johnson Algorithm that is More Space Efficient*, *Information Processing Letters*, vol. 102, no. 5, pp. 201–207, May 2007. DOI: 10.1016/j.ipl.2006.11.017. Available: https://www.researchgate.net/publication/222571621_A_variant_of_the_Ford-Johnson_algorithm_that_is_more_space_efficient.