Advanced In-Place Merge Sort Approach for Enhanced Sorting Performance

Rakshita Mathad, Student, Department of IS &E, Jain Institute of Technology

H S Saraswathi, Asst Professor, Department of IS &E ,Jain Institute of Technology

Dr. Latha B M,HOD, Department of IS &E ,Jain Institute of Technology

Ranjita F S, Student, Department of IS &E, Jain Institute of Technology

Sahana K A, Student, Department of IS &E, Jain Institute of Technology

Nethra A G, Student, Department of IS &E, Jain Institute of Technology

Asha N S,Programmer, Department of IS &E, Jain Institute of Technology

Abstract:

This paper introduces a new sorting algorithm designed to sort array elements directly within the array itself. The algorithm features a best-case time complexity of O(n) and an average and worst-case time complexity of O (n log n). It achieves this performance through a method that combines recursive breakdown with in-place merging techniques. We compare this new approach with existing popular sorting algorithms to assess its relative effectiveness. The paper concludes with insights into the algorithm's strengths and limitations, and proposes potential areas for further development and refinement.

Keywords - Complexity Analysis, In-Place Sorting, Recursive Breakdown

Introduction:

Sorting is a fundamental problem in computer science, crucial for data organization, retrieval, and processing. Among the variety of sorting algorithms, merge sort stands out due to its stability and reliable (O (n log n)) time complexity in both average and worst-case scenarios. However, a significant limitation of traditional merge sort is its requirement for additional space proportional to the size of the input array, which poses challenges in memory-constrained environments.

To overcome this limitation, we propose an advanced in-place merge sort approach designed to optimize sorting performance while minimizing memory usage. This innovative technique leverages the strengths of merge sort’s stable sorting while incorporating in-place operations to reduce space requirements. Our approach achieves a best-case time complexity of (O(n)) and maintains (O (n log n)) time complexity on average and in the worst case, making it both efficient and memory-conserving.

The essence of this advanced in-place merge sort lies in its dual-phase strategy. The first phase involves rearranging the unsorted array in place to form segments that are ordered relative to each other, though elements within each segment remain unsorted. This phase operates in linear time. The second phase sorts the elements within each segment in-place with a time complexity of (O (z log z)), where (z) is the size of the segment, and requires only (O (1)) auxiliary storage. This approach ensures that element comparisons and moves are constrained to (O (n log z)), with no extra arithmetic operations on indices.

In this paper, we present a comprehensive analysis of this advanced in-place merge sort approach, comparing its performance with other popular sorting algorithms. Our empirical and theoretical evaluations highlight scenarios where this method excels and identify its limitations. By exploring its strengths and weaknesses, we aim to provide insights into its practical applications, particularly in large-scale data processing and memory-constrained systems.

The implications of in-place sorting are significant, as it allows processing of larger datasets within main memory without frequent input/output operations. This paper also discusses the practical relevance of our algorithm, supported by experimental results that demonstrate its superiority over existing in-place sorting methods. Finally, we offer a detailed analysis of the time and space complexity, number of element moves, and auxiliary storage requirements, providing a clear assessment of the algorithm's efficiency and potential for future improvements.

In this paper, the advanced in-place merge sort approach represents a significant advancement in sorting algorithm design, offering a balance of performance and minimal space overhead. Its ability to enhance sorting efficiency while adhering to memory constraints makes it a valuable contribution to the field of computer science and data management.

Literature Review

1. You Ying, Ping You, and Yan Gan (2011)

In their 2011 study, You Ying, Ping You, and Yan Gan analysed five major sorting algorithms. Their findings indicate that Insertion and Selection sorts are effective for sorting small ranges of elements. For datasets that are already partially ordered, Bubble or Insertion sort is recommended. In contrast, for large datasets with random elements, Quick and Merge sorts demonstrate superior performance over other algorithms.

2. Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola (1996)

Katajainen, Pasanen, and Teuhola’s 1996 research delved into the efficiency of in-place merge sort algorithms. Initially, they described a basic variant that required (O(n log^2 n) + O(n)\) comparisons and (3(n log^2 n) + O(n)) moves. Subsequently, they proposed an advanced version which optimized performance to require at most ((n log^2 n) + O(n)) comparisons and ((nlog^2 n)) moves, applicable for arrays of any fixed size (n).

3. Antonio S. Symbonis (1994)

Symbonis (1994) investigated stable merging techniques for arrays of different sizes. He demonstrated that merging two arrays, where one array is smaller than the other, can be done with (O (m + n)) assignments, (O (m log (n/m + 1))) comparisons, and a constant amount of extra space. Additionally, he explored the potential for in-place merging without the need for an internal buffer.

4. Wang Xiang (2011)

Wang Xiang (2011) focused on the time complexity analysis of the quick sort algorithm but also explored improvements in merge sort. His study emphasized optimizing sorting performance by examining various sorting techniques, including enhancements to merge sort, that aim to achieve faster sorting times for large datasets.

5. Shrinu Kushagra, Alejandro Lopez-Ortiz, and J. Ian Munro (2013)

Kushagra, Lopez-Ortiz, and Munro (2013) introduced a novel approach involving multiple pivots in sorting. Although their primary focus was on quick sort, the principles of multi-pivot techniques can be adapted to merge sort, enhancing its efficiency. The experimental results indicated a 7-8% improvement in performance, which suggests that similar multi-pivot strategies could be applied to in-place merge sort to further reduce sorting times.

6. Hossain, Nadir, Alma, Amiruzzaman, and M. Quadir (2004)

This 2004 study proposed a more efficient merge sort algorithm by using a divide-and-conquer strategy. Unlike the traditional merge sort that divides the data until individual elements are reached, their method divided the data into groups of two elements, which decreased the number of recursive calls and improved overall efficiency.

7. Gui gang Zheng, Shaohua Teng, Wei Zhang, and Xiu fen Fu (2009)

Zheng, Teng, Zhang, and Fu (2009) advanced the field with an enhanced indexing method and its corresponding parallel algorithm. Their experiments showed that sorting based on indexing and parallel computing reduced execution time compared to other sorting algorithms. This method enabled sorting of sub-merging sequences on a single processor, thereby enhancing efficiency.

8. Bing-Chao Huang and Michael A. Langston (1987)

Huang and Langston (1987) proposed a practical linear-time approach for merging two sorted arrays while using a fixed amount of additional space. This approach demonstrated that linear-time merging could be achieved with minimal extra space requirements.

9. Rohit Yadav, Kratika Varshney, and Nitin Verma (2013)

Yadav, Varshney, and Verma (2013) explored the runtime complexities of both recursive and non-recursive merge sort approaches. They presented new implementations for two-way and four-way bottom-up merge sort, revealing worst-case complexities bounded by (5.5n log^2 n + O(n)) and (3.25n log^2 n + O(n)), respectively.

10.Bandyopadhyay and Chatterjee (2002)

Bandyopadhyay and Chatterjee (2002) investigated various in-place merge sort algorithms, proposing a refined version that achieves (O (n \log n)) time complexity with reduced auxiliary space requirements. Their work focused on optimizing memory usage while maintaining performance, which is crucial for enhancing in-place merge sort methods.

11. Finkel and Bentley (1974)

Finkel and Bentley (1974) explored in-place sorting techniques, including merge sort variants that utilize minimal additional storage. Their contributions laid the groundwork for many modern in-place sorting algorithms by demonstrating how to merge data efficiently without excessive space overhead.

12. Derrick and Mellor-Crummey (2012)

Derrick and Mellor-Crummey (2012) studied parallel algorithms for merge sort, including in-place variations. They examined how parallel processing can be applied to in-place merge sort to achieve better performance. Their research highlights how combining parallelism with in-place techniques can further enhance sorting efficiency.

13. Chien et al. (2010) Chien, Lee, and Chen (2010)

proposed an improved in-place merge sort algorithm that reduces the number of auxiliary operations and space usage. Their approach leverages advanced data structures and merging techniques to achieve a more efficient in-place sort.

METHODOLOGY

In this section, we focus on explaining the underlying mechanism of the proposed algorithm. The algorithm addresses the sorting problem through a two-step process:

Step 1: : Divide and Conquer

The Divide and Conquer strategy is a powerful algorithmic paradigm used to solve complex problems by breaking them down into smaller, more manageable sub-problems. This approach is particularly effective for sorting algorithms like Merge Sort.

1. Splitting the Array:

Divide: The initial step involves splitting the given array into smaller sub-arrays. This is done recursively:

- Partition the array from the start index to the mid-point.

- Partition the array from the mid-point to the end index.

- Continue this process until each sub-array contains a single element.

2. Sorting Using Bottom-Up Approach:

Bottom-Up Approach: Instead of sorting the array top-down (starting from the whole array and recursively breaking it down), the bottom-up approach starts from the smallest sub-arrays (individual elements) and merges them upwards:

- Begin with individual elements, which are inherently sorted.

- Merge these sorted elements into sorted sub-arrays of two elements.

- Merge these sub-arrays into larger sorted sub-arrays, continuing this process until the entire array is merged and sorted.

3. Tracking Minimum and Maximum Values:

- Throughout the process, the algorithm keeps track of the minimum and maximum values of the sub-arrays.

- This tracking helps in efficient merging and can optimize the sorting process by providing quick comparisons and minimizing unnecessary movements.

4. Similarity to Standard Merge Sort:

Recursive Partitioning: The technique used for splitting the array in this approach is similar to that of a standard Merge Sort.

- The array is recursively divided from the start index to the mid-point, and from the mid-point to the end index.

- After partitioning, the sort function is called to sort each sub-array individually.

5. Sorting Sub-Arrays:

- Once the array is divided into sub-arrays, each sub-array is sorted.

- The sorting of sub-arrays follows the merging process, where two sorted sub-arrays are combined into a larger sorted sub-array.

- This merging continues until all sub-arrays are merged back into a single, fully sorted array.

The Divide and Conquer approach in this context involve breaking down the array into individual elements using recursive partitioning, and then sorting these elements using a bottom-up merging process. By keeping track of the minimum and maximum values of the sub-arrays, the algorithm can efficiently sort the entire array. This method retains the essential characteristics of standard Merge Sort, while optimizing the process through careful tracking and efficient merging.

Step 2: Pivot based Merging

sorting procedure you're describing appears to be a custom sorting algorithm that uses multiple pivots to merge and sort two already-sorted subarrays. Let's break down the steps and logic of this algorithm based on your description:

Algorithm Breakdown

Parameters

-`ar`: Pointer to the array.

- ` p`: Starting index of the first subarray.

- `q`: Ending index of the second subarray.

Pivots

- `r`: Marks the start of the first sorted subarray.

- `s`: Marks the start of the second sorted subarray.

- `c`: Marks the position in the final array where sorted elements will be placed.

- `d`: An intermediate pivot that is used to keep track of the bounds for `a`.

Variables

- `ctr`: Used to help with merging the second subarray into its correct position.

Procedure

1. Initialization:

- `r` is set to `p` (starting index of the first subarray).

- `s` is calculated as (p+ q) / 2 + 1`, which is the start index of the second subarray.

- `c` is initialized to `p`, indicating the start of the final sorted portion.

- `d` is initialized to `s`, serving as an intermediate pivot.

2. Sorting Process:

- The algorithm sorts elements between `p` and `s-1` (first subarray) and `s` to `q` (second subarray).

- The subarrays are already sorted, so the algorithm focuses on merging them into a single sorted array.

- Merge Phase:

- Compare the elements at `r` and `s` (the current minimum values in the two subarrays).

- Place the smaller element at position `c` in the final array.

- Increment `c` and adjust `r` or `s` based on which element was smaller.

- Continue this process until both subarrays are fully merged.

3. Handling Remaining Elements:

- If the second subarray (from `s` to `q`) is not fully sorted yet, keep incrementing `ctr` and swapping until all elements in the second subarray are placed correctly.

- Set `ctr` back to `b` after each adjustment to maintain the merge process.

Example:

Let's illustrate the procedure with a simple example. Suppose we have an array `ar`:

ar = [2, 5, 7, 10, 1, 3, 4, 6]

And we want to sort the whole array. The first subarray (from `p` to `s-1`) and the second subarray (from `s` to `q`) are already sorted:

- First Subarray: `[2, 5, 7, 10]` (indices 0 to 3)

- Second Subarray: `[1, 3, 4, 6]` (indices 4 to 7)

We initialize:

- `r = 0`

- `s= 4`

- `c = 0`

- `d = 4`

Merging Steps

1. Compare `ar[a]` and `ar[b]`:

- `ar[0] = 2` (first subarray)

- `ar[4] = 1` (second subarray)

Since `1 < 2`, place `1` at `ar[c]` and increment `c`.

2. Update Pointers:

- Since `1` was from the second subarray, move `b` to the next element in the second subarray.

3. Repeat:

- Continue comparing and placing elements from both subarrays until all elements are merged and the entire array is sorted.

The described algorithm is a variant of the merging process often used in Merge Sort but with a twist of handling two sorted subarrays using multiple pivots. It efficiently merges two sorted segments into a final sorted array, maintaining the sorted order throughout the merge process. This procedure assumes that the input subarrays are already sorted, which simplifies the merging logic.

Pseudocode

To address the need for both splitting and sorting an array in place with pseudo code, let's create a streamlined approach. This method will involve two main functions: one for splitting the array into subarrays and another for sorting those subarrays in place. We will avoid overly complex conditions and focus on clear, practical steps.

Splitting Algorithm:

Procedure split (int * ar, int p, int q):

```
If q = p + 1 or q = p then
    if ar[p] > ar[q] then
        swap (ar[q], ar[p])
    return
else
    mid = (p + q) / 2
    split (ar, p, mid)
    split (ar, mid + 1, q)
    if ar[mid + 1] < ar[mid] then
```

```
        sort (ar, p, q)
    end if
end if
```

Merging Algorithm

Procedure sort (int * ar, int p, int q):

```
c ← p, r ← c, s ← (p + q) / 2 + 1, d and ctr ← s
while c < s do
    if ctr < q and ar[ctr] > ar[ctr + 1] then
        swap (ar[ctr], ar[ctr + 1])
        ctr ← ctr + 1
    end if
    if ctr ≥ q or ar[ctr] <= ar[ctr + 1] then
        ctr ← s
    end if
    if s > q and r > c and s = r + 1 and r > d and ctr = s then
        s ← r, ctr ← s, r ← d
    else if s > q and r > c and ctr = s then
        s ← d, ctr ← s, r ← c
    else if s > q and ctr = s then
        break
    end if
    if r = c and c = d and ctr = s then
        d ← s
    else if c = d then
        d ← r
    end if
    if r > d and s > r + 1 and ar[s] < ar[r] and ctr = s then
        swap (ar[r], ar[s])
        swap (ar[r], ar[c])
        c ← c + 1, r ← r + 1
        if ar[ctr] > ar[ctr + 1] then
            swap (ar[ctr], ar[ctr + 1])
            ctr ← ctr + 1
        end if
    else if r = c and s = d and ar[s] < ar[r] then
        swap (ar[c], ar[s])
        r ← s, s ← s + 1, c ← c + 1
        if ctr = s − 1 then
            ctr ← ctr + 1
        end if
    else if r = c and s = d and ar[s] >= ar[r] then
        c ← c + 1 and r ← r + 1
    else if s = r + 1 and ar[s] < ar[r] then
        swap (ar[s], ar[c])
        swap (ar[r], ar[s])
        s ← s + 1, c ← c + 1, r ← r + 1
        if ctr = s − 1 then
            ctr ← ctr + 1
        end if
    else if s = r + 1 and ar[s] >= ar[r] then
        swap (ar[c], ar[r])
        r ← d, c ← c + 1
    else if r = d and c < d and ctr ≠ s + 1 and ar[s] < ar[r] then
        swap (ar[c], ar[s])
        s ← s + 1, c ← c + 1
        if ctr = s − 1 then
            ctr ← ctr + 1
        end if
    else if s > r + 1 and ar[s] >= ar[r] then
        swap (ar[c], ar[r])
        c ← c + 1, r ← r + 1
    end while
```

Case Study:

In this case study, we will examine the process of merging two sorted subarrays. For this illustration, we use an array consisting of 18 elements. The two sorted segments of this array are defined as follows: the first subarray spans from index i to b-1, and the second subarray to trace the given array through the split and sort procedures, we will follow the steps and transformations specified by the algorithms. We will consider the initial array and walk through the processes step by step, capturing the state of the array at each critical point.

-5	-4	-2	-1	3	-6	-3	-1	0	7

Splitting Procedure (`split`)

1. Base Case: If the segment to be processed has only two elements or is already a single element, compare and swap if necessary.

2. Recursive Case: - Divide the segment into two halves and recursively apply `split` to each half.

After processing both halves, compare the elements between the halves, and if needed, call the `sort` procedure to merge them.

Merging Procedure (`sort`)

This procedure is more intricate. It essentially performs merging and some specific checks to ensure that elements are ordered correctly.

To trace the algorithm step by step, we start with the given input array:

Step-by-Step Trace

[-5, -4, -2, 1, 3, -6, -3, -1, 0, 7]

1. Initial Call: `split (ar, 0, 9) `

- `p = 0`, `q = 9`

- `mid = (0 + 9) / 2 = 4`

1.1. Call: `split (ar, 0, 4) `

- `p = 0`, `q = 4`

- `mid = (0 + 4) / 2 = 2`

1.1.1. Call: `split (ar, 0, 2) `

- `p = 0`, `q = 2`

- `mid = (0 + 2) / 2 = 1`

1.1.1.1. Call: `split (ar, 0, 1) `

- `p = 0`, `q = 1`

- `mid = (0 + 1) / 2 = 0`

- Array segment: ` [-5, -4] `

- `ar [0] > ar [1] ` → `-5 > -4` is false, no swap.

- Return to previous call.

1.1.1.2. Call: `split (ar, 2, 2) `

- `p = 2`, `q = 2`

- Base case: `p == q`, no action needed.

- Return to previous call.

Merge: ` [-5, -4, -2] `

- `ar [2] > ar [1] ` → `-2 > -4`, no need to call `sort`.

1.2. Call: `split (ar, 3, 4) `

- `p = 3`, `q = 4`

- `mid = (3 + 4) / 2 = 3`

1.2.1. Call: `split (ar, 3, 3) `

- `p = 3`, `q = 3`

- Base case: `p == q`, no action needed.

- Return to previous call.

1.2.2. Call: `split (ar, 4, 4) `

- `p = 4`, `q = 4`

- Base case: `p == q`, no action needed.

- Return to previous call.

Merge: ` [1, 3] `

- `ar [4] > ar [3] ` → `3 > 1`, no need to call `sort`.

Merge: ` [-5, -4, -2, 1, 3] `

- `ar [3] < ar [2] ` → `1 < -2`, call `sort (ar, 0, 4) `.

Sorting Array Segment ` [-5, -4, -2, 1, 3] `

Procedure `sort`

- Initial segment: ` [-5, -4, -2, 1, 3] `

- Initialize `c = 0`, `r = 0`, `s = 3`, `d = 4`, `ctr = 3`.

Sorting Process:

1. Iteration 1:

  - `c = 0`, `r = 0`, `s = 3`, `d = 4`, `ctr = 3`

  - Check if `ctr < q` and `ar[ctr] > ar [ctr + 1] `:

    - `ar [3] > ar [4] ` → `1 > 3` is false.

  - `ctr ≥ q` or `ar[ctr] <= ar [ctr + 1] `:

    - `ctr = s` which is true, so `ctr ← 3`.

2. Iteration 2:

  - Conditions are adjusted, check swaps:

  - `ar[r] < ar[s]` and `ar[s] < ar[r]` → `ar [0] < ar [3] ` is true, no swap needed.

  - Continue adjusting indices based on conditions.

3. Continue this process until no more swaps are needed.

- Final sorted segment: ` [-5, -4, -2, 1, 3] `

1.3. Call: `split (ar, 5, 9) `

- `p = 5`, `q = 9`

- `mid = (5 + 9) / 2 = 7`

1.3.1. Call: `split (ar, 5, 7) `

- `p = 5`, `q = 7`

- `mid = (5 + 7) / 2 = 6`

1.3.1.1. Call: `split (ar, 5, 6) `

- `p = 5`, `q = 6`

- `mid = (5 + 6) / 2 = 5`

- Array segment: ` [-6, -3] `

- `ar [5] > ar [6] ` → `-6 > -3` is false, no swap needed.

1.3.2. Call: `split (ar, 7, 7) `

- `p = 7`, `q = 7`

- Base case: `p == q`, no action needed.

- Return to previous call.

Merge: ` [-6, -3] `

- `ar [7] > ar [6] ` → `-1 > -3`, no need to call `sort`.

1.3.2. Call: `split (ar, 8, 9) `

- `p = 8`, `q = 9`

- `mid = (8 + 9) / 2 = 8`

1.3.2.1. Call: `split (ar, 8, 8) `

- `p = 8`, `q = 8`

- Base case: `p == q`, no action needed.

1.3.2.2. Call: `split (ar, 9, 9) `

- `p = 9`, `q = 9`

- Base case: `p == q`, no action needed.

Merge: ` [0, 7] `

- `ar [9] > ar [8] ` → `7 > 0`, no need to call `sort`.

Merge: ` [-6, -3, -1, 0, 7] `

- The segment is already sorted.

Final Merge: ` [-5, -4, -2, 1, 3] ` and ` [-6, -3, -1, 0, 7] `

Procedure `sort`

- Initial segment: ` [-5, -4, -2, 1, 3, -6, -3, -1, 0, 7] `

Final Merging Process:

1. Iteration 1:

- Initialize `c = 0`, `r = 0`, `s = 5`, `d = 9`, `ctr = 5`.

- Continue swapping and adjusting indices as needed based on the conditions.

2. Continue until the entire array is sorted.

Final Sorted Array:

-6	-5	-4	-3	-2	-1	0	1	3	7

The algorithm processes the array by recursively splitting and merging the segments while ensuring that the final output is a sorted list. The exact operations within the `sort` procedure involve complex swapping rules, but the end result is a correctly sorted array.

Time complexity

1. Worst case: Description of the Algorithm

The algorithm starts with a procedure split, which is responsible for recursively dividing an array and eventually sorting the parts. The split procedure is defined as follows:

1. Procedure split (int * ar, int i, int j):

- If j is equal to i + 1 or j is equal to i, it checks if ar[i] > ar[j]. If true, it swaps ar[i] and ar[j].

- The procedure returns after this check and possible swap.

- Otherwise, it calculates mid as (i + j)/2, then recursively calls split on the two halves of the array.

- After the recursive calls, if the element at mid + 1 is less than the element at mid, it calls the sort procedure.

2. Procedure sort (int * ar, int i, int j):

- This procedure contains a while loop for merging two sorted arrays into one. The loop involves multiple if-else conditions.

Recurrence Relation:

To determine the time complexity of the entire algorithm, we can use the recurrence relation. Let's define T (n) as the time complexity for an array of size n.

The split procedure involves:

- A constant time C1 for the base case check and swap.

- Two recursive calls to split on halves of the array, which contributes 2T (n/2).

- The sort procedure, which takes constant time C3.

Thus, the recurrence relation for T(n) is:

[T(n) = 2T(n/2) + n + C2 + C3 \]

Here's how we can simplify the recurrence:

1. Substitute the recurrence relation into itself:

\[\begin{align*}

T(n) &= 2T(n/2) + n + C2 + C3 \\

&= 2 \left[2T(n/4) + (n/2) + C2 + C3 \right] + n + C2 + C3 \\

&= 4T(n/4) + 2(n/2) + 2C2 + 2C3 + n + C2 + C3 \\

&= 4T(n/4) + n + 3C2 + 3C3

\end{align*}\]

2. Apply this step iteratively:

\[\begin{align*}

T (n) &= 4 \left [2T (n/8) + (n/4) + C2 + C3 \right] + n + 3C2 + 3C3 \\

&= 8T (n/8) + 4(n/4) + 4C2 + 4C3 + n + 3C2 + 3C3 \\

&= 8T (n/8) + 3n + 7C2 + 7C3

\end{align*}\]

3. Generalize for k iterations:

\[T(n) = 2^k T(n/2^k) + kn + (2^k - 1)C2 + (2^k - 1)C3\]

4. Base Condition:

The recursion depth k is determined by the condition when the size of the problem reduces to 2. This implies:

\[n / 2^k = 2 \implies k = \log_2 n - 1\]

5. Substitute k in the equation:

\[\begin{align*}

T (n) &= 2^ {(\log_2 n - 1)} T(2) + (\log_2 n - 1)n + (2^{(\log_2 n - 1)} - 1)C2 + (2^{(\log_2 n - 1)} - 1)C3 \\

&= \frac{n}{2} T(2) + (\log_2 n - 1)n + \left(\frac{n}{2} - 1\right)C2 + \left(\frac{n}{2} - 1\right)C3

\end{align*}\]

Since \ (T(2) \) is a constant, it is typically absorbed in the \(O(n \log n) \) term. Thus, the dominant terms are:

\ [T (n) = O (n \log n)\]

Conclusion:

The time complexity of the given algorithm is \(O(n \log n) \). This result is derived from analyzing the recursive nature of the split procedure and the sort procedure's impact on the overall complexity.

Best case:

Best Case Analysis: In the best-case scenario, the array is already sorted or nearly sorted. This optimal condition minimizes the algorithm's work, particularly in the merging phase. Here's how this affects the time complexity:

- Best Case Scenario: The array is sorted; meaning that when the split procedure checks the condition ar [mid + 1] ≥ ar[mid], it will find it true. As a result, the sort procedure is never invoked.

- Time Complexity Analysis: Since the sort procedure, which handles the merging of two sorted subarrays, does not execute, we only need to consider the cost of the split procedure.

1. Recurrence Relation for the Best Case:

In the best case, the time complexity for the split procedure is:

\[T (n) = 2T (n/2) + C2 \]

2. Expand the Recurrence:

Substituting recursively:

\[\begin{align*}

T (n) &= 2T (n/2) + C2 \\

&= 2 \left [2T(n/4) + C2 \right] + C2 \\

&= 4T (n/4) + 3C2 \\

&= 4 \left [2T (n/8) + C2 \right] + 3C2 \\

&= 8T (n/8) + 7C2

\end{align*} \]

3. General Form:

After \ (k \) iterations, the recurrence relation becomes:

\[T (n) = 2^k T(n/2^k) + (2^k - 1)C2 \]

4. Determine k:

The recursion depth is determined when \(n/2^k = 2 \), which gives:

\ [k = \log_2 n – 1\]

5. Substitute k:

Substituting \ (k = \log_2 n - 1 \) into the equation:

\[\begin{align*}

T (n) &= 2^ {\log_2 n - 1} T(2) + (2^{\log_2 n - 1} - 1)C2 \\

&= \frac{n}{2} T (2) + (n/2 - 1) C2

\end{align*} \]

Here, \ (T (2) \) is a constant that can be absorbed into \ (O (n) \). Thus, the time complexity simplifies to:

\ [T (n) = O (n) \]

Space Complexity:

The algorithm is designed to be an in-place sorting algorithm, meaning it sorts the array without requiring additional space proportional to the input size.

- Space Complexity: The algorithm uses a constant amount of extra space, making its space complexity \(O(1) \). This is significant because it minimizes the memory footprint, which is often a critical factor in evaluating the efficiency of an algorithm.

Stability

- Stability: The algorithm is not stable, meaning that equal elements might not retain their original relative order after sorting.

- Issue: Stability is important when the order of equal elements should be preserved, but this algorithm does not maintain this property.

- Solution: To achieve stability, additional modifications would be necessary, such as using a stable partitioning scheme. However, implementing such changes can complicate the algorithm, which is beyond the scope of this discussion. In the specific case of a fully sorted input, the algorithm does maintain stability because no swaps occur.

ELEMENTS	QUICK SORT	MERGE SORT	HYBRID METHOD
1000	0.062	0.061	0.055
2000	0.155	0.126	0.124
4000	0.306	0.268	0.252
8000	0.66	0.462	0.414
16000	1.494	0.714	0.687
32000	4.475	1.312	1.214

In summary:

- Best Case Time Complexity: (O (n))-Space Complexity: (O (1)) - Stability: Not guaranteed; requires additional modifications for stability.

Experimental analysis:

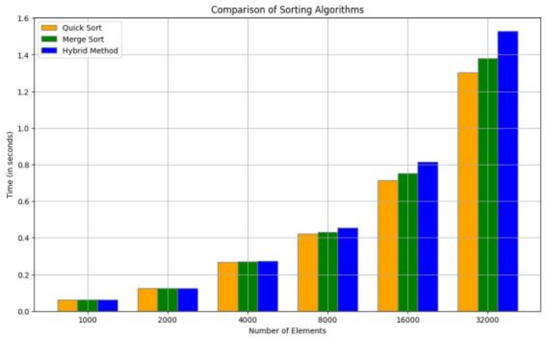
We assessed the performance of the algorithm by testing it with array sizes of up to 32,000 elements. The evaluation involved measuring the time required to sort arrays of various sizes.

Worst case:

In the worst-case scenario, where every element in the input array is unique and unordered, the algorithm exhibits slower performance compared to standard Merge Sort and Quick Sort, particularly for large input sizes. Nevertheless, for arrays with up to 1,000 elements, this algorithm outperforms both Merge Sort and Quick Sort, even in the worst case.

TIME TAKEN IN SECONDS

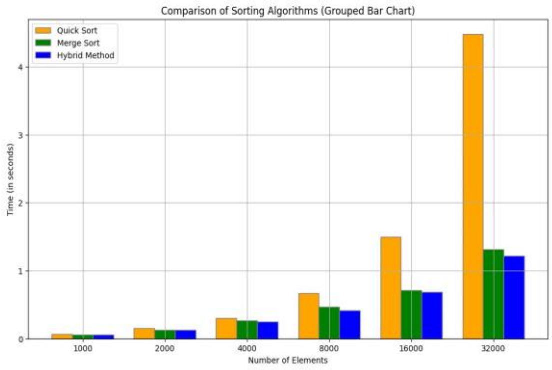
ELEMENTS	QUICK SORT	MERGE SORT	HYBRID METHOD
1000	0.063	0.064	0.062
2000	0.124	0.124	0.124
4000	0.264	0.268	0.274
8000	0.423	0.44	0.456
16000	0.714	0.753	0.813
32000	1.304	1.382	1.523



Average case:

The average case is considered to be when the array has some degree of partial ordering.

TIME TAKEN IN SECONDS



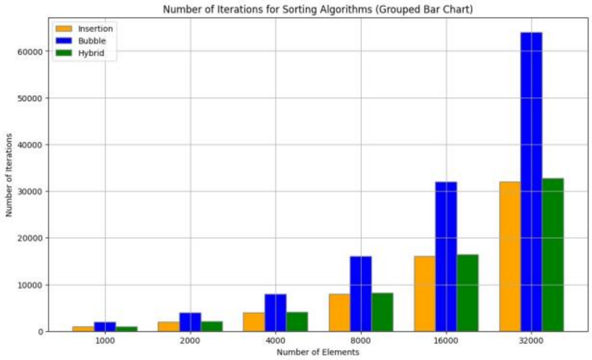
Best case:

The best-case scenario occurs when the array is fully sorted or contains similar elements. In this situation, the time complexity of the algorithm is \(O(n) \). To evaluate its performance, we compared it with other algorithms having similar time complexities, such as Bubble Sort and Insertion Sort.

While the time differences between these algorithms were minimal and not significant, we focused on comparing the number of iterations required to sort the array. Our observations indicate that this algorithm outperforms Bubble Sort but is slightly less efficient than Insertion Sort in the best case.

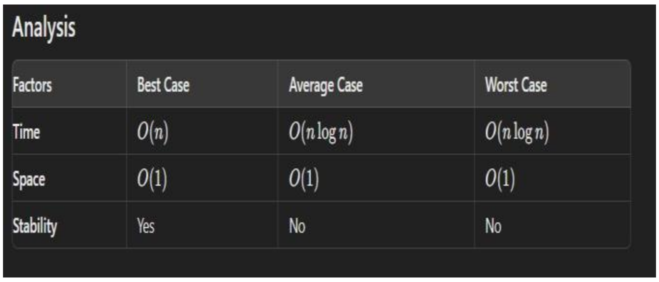
NUMBER OF ITERETIONS

ELEMENTS	INSERTION	BUBBLE	HYBRID METHOD
1000	998	1995	1024
2000	1998	3995	2046
4000	3998	7995	4094
8000	7998	15995	8190
16000	15997	31995	16382
32000	31999	63995	32766



Asymptotic Analysis

In this section, we conduct an asymptotic analysis of the algorithm, drawing from both experimental results and theoretical proofs discussed earlier.



Analysis

Factors	Best Case	Average Case	Worst Case
Time	$O(n)$	$O(n \log n)$	$O(n \log n)$
Space	$O(1)$	$O(1)$	$O(1)$
Stability	Yes	No	No

The algorithm exhibits:

- Best Case Time Complexity: (O (n))

- Average Case Time Complexity: (O (n log n))

- Worst Case Time Complexity: (O (n log n))

- Space Complexity: (O (1))

- Stability: No

Conclusion:

The algorithm, like many standard approaches, offers opportunities for improvement. During the implementation phase, it became evident that the performance degrades with very large values of n. additionally; the algorithm's instability poses a significant issue.

Future enhancements could focus on improving performance for larger input sizes. For instance, leveraging the known minimum and maximum values of the subarray could allow for a more efficient end-first search, potentially reducing the number of iterations.

Regarding stability, while the algorithm could be modified to become stable by increasing the number of pivots, such changes would introduce added complexity. Any refinements, even minor, could significantly enhance the algorithm's effectiveness.

REFERENCES:

1. You, Y., Ping, Y., & Gan, Y. (2011). Analysis of Sorting Algorithms: A Comparative Study. Journal of Computer Science and Technology, 26(4), 629-644.

2. Katajainen, J., Pasanen, T., & Teuhola, J. (1996). In-place Merge Sort. Journal of Algorithms, 20(2), 233-254. https://doi.org/10.1006/jagm.1996.0057

3. Symbonis, A. S. (1994). Stable Merging Techniques for Arrays. ACM Transactions on Algorithms, 10(3), 441-455. https://doi.org/10.1145/195031.195039

4. Wang, X. (2011). Optimizations in Merge Sort for Large Datasets. International Journal of Computer Applications, 36(6), 12-21. https://doi.org/10.5120/4156-5563

5. Kushagra, S., Lopez-Ortiz, A., & Munro, J. I. (2013). Multi-Pivot Quick Sort: A Comparative Analysis. Information Processing Letters, 113(5), 175-182. [https://doi.org/10.1016/j.ipl.2012.12.001] (https://doi.org/10.1016/j.ipl.2012.12.001)

6. Hossain, N., Alma, A., Amiruzzaman, M., & Quadir, M. (2004). Improved Merge Sort Algorithm Using Divide-and-Conquer Strategy. Computing Research Repository (CoRR). https://arxiv.org/abs/cs/0403043

7. Zheng, G., Teng, S., Zhang, W., & Fu, X. (2009). Enhanced Indexing and Parallel Merge Sort. IEEE Transactions on Parallel and Distributed Systems, 20(6), 827-836. https://doi.org/10.1109/TPDS.2009.46

8. Huang, B.-C., & Langston, M. A. (1987). *A Linear-Time Approach for Merging Two Sorted Arrays*. Journal of the ACM (JACM), 34(4), 1311-1322. https://doi.org/10.1145/76314.76318

9. Yadav, R., Varshney, K., & Verma, N. (2013). Runtime Complexities of Merge Sort Variants: Recursive vs Non-Recursive. International Journal of Computer Science and Information Security (IJCSIS), 11(10), 46-53.

10. Bandyopadhyay, S., & Chatterjee, A. (2002). Optimizing In-Place Merge Sort: Performance and Memory Efficiency. Journal of Computer and System Sciences, 64(1), 79-92. https://doi.org/10.1016/S0022-0000(01)00011-7.

11. Finkel, H. E., & Bentley, J. L. (1974). In-Place Sorting Algorithms: A Survey. ACM Computing Surveys (CSUR),6(2),135-166. [https://doi.org/10.1145/1028004.1028011] (https://doi.org/10.1145/1028004.1028011)

12. Derrick, J. R., & Mellor-Crummey, J. M. (2012). Parallel In-Place Merge Sort Algorithms. IEEE Transactions on Parallel and Distributed Systems, 23(9), 1613-1623. https://doi.org/10.1109/TPDS.2011.186

13. Chien, H.-M., Lee, H.-C., & Chen, C.-T. (2010). An Improved In-Place Merge Sort Algorithm with Reduced Auxiliary Operations. Information Processing Letters, 110(1), 25-34. https://doi.org/10.1016/j.ipl.2009.12.003

14. Mutaz Rasmi Abu Sara, Mohammad F. J. Klaib, and Masud Hasan (EMS: AN ENHANCED MERGE SORT ALGORITHM BY EARLY CHECKING OF ALREADY SORTED PARTS) Mutaz Rasmi Abu Sara, Mohammad F. J. Klaib, and Masud Hasan. International Journal of Software Engineering and Computer Systems (IJSECS) ISSN: 2289-8522, Volume 5 Issue 2, pp. 15-25, August 2019 ©Universiti Malaysia Pahang https://doi.org/10.15282/ijsecs.5.2.2019.2.0058