Enhancing IntroSort with Modified Hoare Partitioning*

Dominguez, Carl Vince (*Author*), Yecyec, Marian Ivy (*Author*)

[1]Department of Computer Science, College of Information Technology and Computing, University of Science and Technology of Southern Philippines; [2]Department of Computer Science, College of Information Technology and Computing, University of Science and Technology of Southern Philippines.
Email: dominguez.carlvince25@gmail.com, yecyec.marianivykate7@gmail.com

ABSTRACT

This electronic document is a "live" template. The various components of your paper [title, text, heads, etc.] are already defined on the style sheet, as illustrated by the portions given in this document. (Abstract)

Keywords : Component; Formatting; Style; Styling; insert (keywords)

1 INTRODUCTION

Sorting algorithms emerged and can be defined as algorithms focused on sorting items with the help of computer processes. However, to improve computational speed and accuracy of an algorithm, it is combined to leverage the strengths of each individual algorithm and is called Hybrid Algorithm. An example of a hybrid algorithm is Introsort, proposed by David Musser in 1997, a combination of quick sort, heap sort, and insertion sort to optimize performance and prevent worst-case scenarios typical in quick sort. While Introsort is efficient, Musser identified a disadvantage: it requires extra code space, nearly double that of using quick sort or heap sort alone David Musser, (1997). Musser recommended exploring alternative partitioning methods to enhance Introsort's efficiency further, such as using the first element as the pivot.

In response to this recommendation, the researchers propose the Introsort with Modified Hoare Partitioning Method. The primary goal is to reduce the code space requirement and potentially improve partitioning efficiency and stability. The researchers chose the Modified Hoare Partitioning Method because it uses the first element as the pivot, which Musser recommended in his study.

The goal of this paper is to propose Introsort with the Modified Hoare Partitioning Method, focusing on the code space reduction and better partitioning performance than the original Introsort. We conduct theoretical assessments of the algorithm efficiency, comparisons of the two and performance measurements across different data distributions. Through this study, we aim to address the limitations identified by David Musser and enhance the overall performance of Introsort.

2 REVIEW OF RELATED LITERATURE

2.1 Hybrid Algorithm

Sharma, B., & Kumar, A. (2022) said that a hybrid algorithm is a combination of two or more algorithms that work together to solve a specific problem. There are a lot of advantages offered by hybrid sorting algorithms, such as improved computational speed and accuracy while improving the strengths of each individual algorithm. Examples of these hybrid algorithms are Introsort, Hybrid Mergesort, Tim sort, Insertion-Merge hybrid. These algorithms can be more adaptive and robust than single sorting algorithms and they can handle different data types and scenarios.

2.2 Introsort

Li, S., et.al. (2019) said that Introsort is a hybrid sorting algorithm that combines the best features of quicksort, heapsort and insertion sort to optimize sorting performance. It is used in various applications and libraries, such as the GNU Standard C++ Library. It aims to achieve the best average performance of quicksort while avoiding the poor performance of quicksort in the worst-case scenario.

According to Ferrada (2022), Introsort is considered one of the fastest algorithms used for sorting any kind of inputs and offers a balance between the theoretical worst-case time complexity and empirical efficiency.

In the study of Lammich, P. (2020), it was written that introsort uses the simpler median-of-three partition

Musser (1997) noted the partitioning method used and its advantages which includes achieving an *O(n log n)* worst-case time, maintaining efficiency in most cases through quicksort-like behavior, and improving performance on median-of-killer sequences. However, one possible disadvantage of this is that it requires an extra code space. This can lead to increased space utilization and possibly impact performance in compile-time generic libraries.

2.3 Modified Hoare

According to Abhyankar, D., & Ingle, M. (2011), the Modified Hoare partitioning algorithm is an optimized version of the original Hoare partitioning algorithm used in Quicksort. It improves the efficiency of the partitioning process by reducing unnecessary index manipulations and swap operations. It uses the first element as the pivot, then swaps the first element with the last element if the first element is greater than the last element. This make sures that the pivot element is the first element before beginning the partitioning process, which improves the efficiency of the partitioning algorithm.

In a different study of Abhyankar, D., & Ingle, M. (2011), based on various performance indicators like elapsed time, CPU cache misses, and branch mispredictions the Modified Hoare algorithm has superior performance compared to other partitioning algorithms, such as Lomuto and Modified Lomuto.

3 METHODOLOGY

As noted by David Musser, one disadvantage of Introsort is the extra code space it requires. To address this, the researchers propose using a different partitioning method to reduce the code space. Instead of the median-of-three pivot selection, the researchers proposed to use the Modified Hoare Algorithm partitioning method, an optimized version of the original Hoare partitioning algorithm.

3.1 Pseudocode

```
procedure swap(a, b):
    temp ← a
    a ← b
    b ← temp

procedure insertionSort(arr, left, right):
    for i from left + 1 to right:
        key ← arr[i]
        j ← i - 1
        while j ≥ left and arr[j] > key:
            arr[j + 1] ← arr[j]
            j ← j - 1
        arr[j + 1] ← key

function MHoare(data, first, last):
    if data[first] > data[last]:
        swap(data[first], data[last])
    pivot ← data[first]
    left ← first
    right ← last
    while true:
        while data[--right] > pivot:
            continue
        data[left] ← data[right]


        while data[++left] < pivot:
            continue
        if left < right:
            data[right] ← data[left]
        else:
            if data[right + 1] ≤ pivot:
                right ← right + 1
            data[right] ← pivot
            return right

procedure partition(arr, low, high):
    pivot ← arr[high]
    i ← low - 1
    for j from low to high - 1:
        if arr[j] ≤ pivot:
            i ← i + 1
            swap(arr[i], arr[j])
    swap(arr[i + 1], arr[high])
    return i + 1

procedure IntrosortUtil(arr, begin, end, depthLimit):
    size ← end - begin
    if size ≤ 16:
        insertionSort(arr, begin, end)
        return
    if depthLimit = 0:
        heapsort(arr, begin, end) // assuming heapsort is implemented
        return
    pivot ← MHoare(arr, begin, end)  // Using Modified Hoare partitioning
    swap(arr[pivot], arr[end])  // Swap pivot with the last element
    partitionPoint ← partition(arr, begin, end)
    IntrosortUtil(arr, begin, partitionPoint - 1, depthLimit - 1)
    IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1)

procedure Introsort(arr, begin, end):
    depthLimit ← 2 × log(end - begin)
    IntrosortUtil(arr, begin, end, depthLimit)
```

3.2 Prototype: C++ Implementation

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm> // for swap

// Swap function
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Insertion sort function
void insertionSort(std::vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}

// Modified Hoare partitioning
int MHoare(std::vector<int>& data, int first, int last) {


  if (data[first] > data[last]) {
        swap(data[first], data[last]);
    }
    int pivot = data[first];
    int left = first;
    int right = last;
    while (true) {
        while (data[--right] > pivot) continue;
        data[left] = data[right];



        while (data[++left] < pivot) continue;
        if (left < right) {
            data[right] = data[left];
        } else {
            if (data[right + 1] <= pivot) {
                right = right + 1;
            }
            data[right] = pivot;
            return right;
        }
    }
}

// Partition function
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Heapsort function (placeholder, needs proper implementation)
void heapsort(std::vector<int>& arr, int begin, int end) {
    std::make_heap(arr.begin() + begin, arr.begin() + end + 1);
    std::sort_heap(arr.begin() + begin, arr.begin() + end + 1);
}

// Utility function for Introsort
void IntrosortUtil(std::vector<int>& arr, int begin, int end, int
depthLimit) {
    int size = end - begin;
    if (size <= 16) {
        insertionSort(arr, begin, end);
        return;
    }
    if (depthLimit == 0) {
        heapsort(arr, begin, end);
        return;


    }
    int pivot = MHoare(arr, begin, end);
    swap(arr[pivot], arr[end]);
    int partitionPoint = partition(arr, begin, end);
    IntrosortUtil(arr, begin, partitionPoint - 1, depthLimit - 1);
    IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1);
}

// Introsort function
void Introsort(std::vector<int>& arr, int begin, int end) {
    int depthLimit = 2 * log(end - begin);
    IntrosortUtil(arr, begin, end, depthLimit);
}

int main() {
    std::vector<int> arr = {3, 7, 2, 1, 9, 5, 8, 6, 4};
    int n = arr.size();
    Introsort(arr, 0, n - 1);
    for (int i : arr) {
        std::cout << i << " ";
    }
    return 0;
}
```

3.3 Introsort with Modified Hoare Partitioning

Introsort is a hybrid sorting algorithm by David Musser. It uses three algorithms; it begins with the quicksort and if it's near worst case $O(n^2)$ it switches to Heapsort. Lastly, it uses insertion when the number of elements is less than the threshold. However, even though Introsort is known for its efficiency, it still has a disadvantage. David Musser noted a disadvantage of introsort, which is the extra code space required. He recommended to to use another partitioning method wherein it uses the first element as the pivot.

With that, the researchers propose an Introsort with Modified Hoare Partition. Modified Hoare Partition is an optimized version of the original Hoare partitioning. This works by using the first element as the pivot, then swaps the first element with the last element if the first element is greater than the last element, making sure that the pivot element is the first element before beginning the partitioning process, which improves the efficiency of the partitioning algorithm.

How the proposed Introsort with Modified Hoare Partitioning works:

First Layer - Initial Partitioning

Pivot Selection:

- Comparing the first and last elements of the array. If the first element is greater, swap them. Then, set the pivot to the first element.

Partitioning:

- Perform partitioning using the Modified Hoare algorithm.
- Two pointers, left and right, are employed to rearrange the elements.
- The right pointer moves left until it finds an element less than or equal to the pivot, while the left pointer moves right until it finds an element greater than or equal to the pivot.
- If left is less than right, the elements at left and right are swapped.
- This process continues until left is no longer less than right. At this point, the pivot is placed in its correct position, and its index is returned.

Second Layer - Sorting

Recursive Sorting:

- After partitioning, the array is divided into two partitions based on the pivot's index.
- The IntrosortUtil function is recursively called on each partition, with a depth limit parameter to prevent excessive recursion (which could lead to worst-case behavior).
- The depth limit is typically set as a logarithmic function of the array size.

Base Case - Insertion Sort:

- If the size of the partition falls below a certain threshold, the algorithm switches to insertion sort to efficiently handle small subarrays.

Fallback to Heap Sort:

- If the depth limit is reached before the partition size becomes small enough for insertion sort, the algorithm switches to a heap sort to guarantee a *O(n log n)* worst-case time complexity.

Combining Sorted Subarrays:

- After recursively sorting the subarrays, they are combined to produce the final sorted array

3.4 Theoretical Analysis

Time Complexity

Modified Hoare Partitioning:

- The partitioning step involves scanning the array with two pointers, which is *O(n)* in the worst case.

Introsort:

- Quicksort Phase: In the best and average cases, the partitioning results in two balanced subarrays, leading to a recurrence relation similar to quicksort: *T(n) = 2T(n2) + O(n), which solves to O(n log n)*.
- Heap Sort Phase: If the recursion depth exceeds to worst case complexity, it switches to heap sort, ensuring O(n log n) complexity in the worst case.
- Insertion Sort Phase: $O(n^2)$ in the worst case, used for small subarrays (size ≤ 16).

Overall Complexity

- Best Case: *O(n log n)* when the partitions are well balanced.
- Average Case: *O(n log n)* as the probability of balanced partitions is high.
- Worst Case: *O(n log n)* because the fallback to heap sort when recursion depth exceeds the limit.

Space Complexity

The space complexity of Introsort with Modified Hoare Partitioning includes:

Auxiliary Space:

- Recursive Call Stack: The depth of the recursive call stack in the worst case is *O(log n)* due to the depth limit mechanism, ensuring it does not exceed this.
- Heap Sort Phase: Uses *O(1)* additional space since it is performed in-place.
- Insertion Sort Phase: Also in-place, adding *O(1)* space complexity.

The overall space complexity remains *O(log n)* due to the recursive stack, which is efficient and manageable even for large datasets.

3.5 Experimental Setup and Procedure

- Compare the code performance of Introsort and Introsort with Modified Hoare Partition Method with a **sample array {3, 7, 2, 1, 9, 5, 8, 6, 4}**
- Measure the execution time using chrono library of c++ and the code memory usage using <unistd.h> to access POSIX operating system APIs and <sys/resource.h> to access system resource usage information.
- For the sample array;

  Hardware: Desktop with Intel Core i3-10110U CPU @ 2.10GHz 2.59 GHz, 8 GB RAM
  Software: Implementation in C++ using Visual Studio Code.

- For each dataset size and type, conduct multiple runs with;

  Hardware: Desktop with Intel Core i3-10110U CPU @ 2.10GHz 2.59 GHz, 8 GB RAM
  Software: Implementation in C++ using Visual Studio Code.

3.6 Datasets:

1st Test: Small Datasets

Description: Assess algorithm performance with small datasets in different sizes and ranges.

Datasets:

Sizes: 10, 100, 1,000, 10,000
Ranges: 1-10, 1-100, 1-1,000, 1-10,000

2nd Test: Partially Sorted Data

Description: Evaluate algorithm performance with partially sorted data in different sizes and ranges.

Datasets:

Sizes: 100, 1,000, 10,000, 100,000
Ranges: Sorted first half, random second half

3rd Test: Large Datasets

Description: Assess algorithm performance with large datasets and varying ranges.

Datasets:

Size: 10, 000, 100, 000, 1,000,000, 10, 000, 000
Ranges: 1-100, 1-1,000, 1-10,000, 1-100,000

---

4 RESULTS AND DISCUSSION

The researchers executed both Introsort by Musser and the proposed Introsort with Modified Hoarew Partition codes to compare their execution time and memory usage. The researchers conducted four different tests; 1st test is sorting a sample array and three datasets with different sizes and range. The results will be visually presented using tables and line graphs. These visualizations aim to reveal whether the proposed Introsort with Modified Hoare partition performs better and if it uses less code space than Introsort.

1st Test: Sample Array

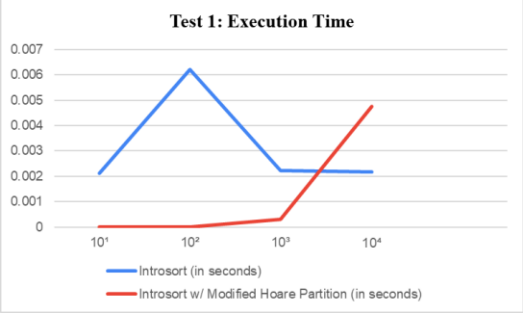
	Execution Time	**Code Space**
Introsort	0.000015 s	2,027,520 bytes
Introsort: Modified Hoare Partition	0.000013 s	1,957,888 bytes

Table 4.1 Sample Array

2nd Test: Small Datasets

Data Size	**10**	**100**	**1,000**	**10,000**
Range	**1 - 10**	**1 - 100**	**1 - 1,000**	**1 10,000**
Introsort	0.002116 s	0.006210 s	0.002220 s	0.002181 s
Introsort: Modified Hoare Partition	0.000013 s	0.000021 s	0.000302 s	0.004758 s

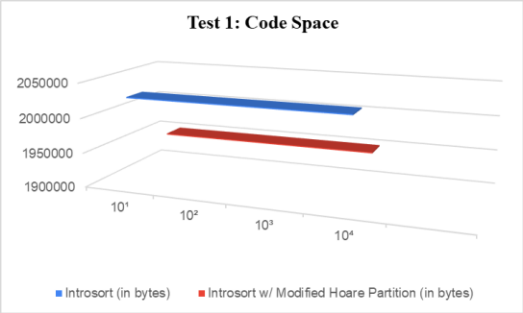
Table 4.2 Small Datasets Execution Time



The results for small datasets show that both Introsort and the proposed Introsort with Modified Hoare partition had comparable execution times, except for larger dataset sizes where the proposed algorithm showed a slight increase in execution time. Overall, both algorithms performed efficiently for random small datasets.

Data Size	**10**	**100**	**1,000**	**10,000**
Range	**1 - 10**	**1 - 100**	**1 - 1,000**	**1 - 10,000**
Introsort	2027520 bytes	2027520 bytes	2027520 bytes	2027520 bytes
Introsort: Modified Hoare Partition	1949696 byte	1949696 byte	1949696 byte	1949696 byte

Table 4.3 Small Datasets Code Space

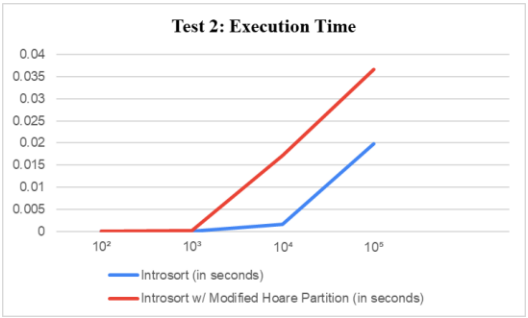


In the results for the code space used, Introsort by David Musser used significantly more space compared to the proposed Introsort with Modified Hoare partition, which utilized 0 byte code space for small datasets.

3rd Test: Partially Sorted Data

Data Size	**100**	**1,000**	**10,000**	**100,000**
Range	**Sorted first half, random second half**			
Introsort	0.000021 s	0.000118 s	0.001672 s	0.019867 s
Introsort: Modified Hoare Partition	0.000029 s	0.000162 s	0.017189 s	0.036655 s

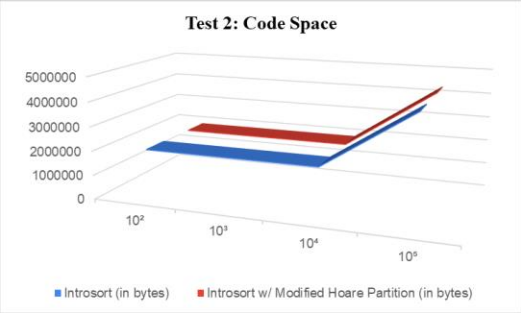
Table 4.4 Partially Sorted Datasets Execution Time



Both algorithms performed well execution time for smaller dataset sizes execution time for partially sorted data. However, the proposed algorithm showed an increase in execution time as the dataset size increased compared to Introsort.

Data Size	**100**	**1,000**	**10,000**	**100,000**
Range	**Sorted first half, random second half**			
Introsort	1961984 bytes	1961984 bytes	1961984 bytes	4325376 bytes
Introsort: Modified Hoare Partition	1937408 bytes	1937408 bytes	1937408 bytes	4341760 bytes

Table 4.5 Partially Sorted Datasets Code Space

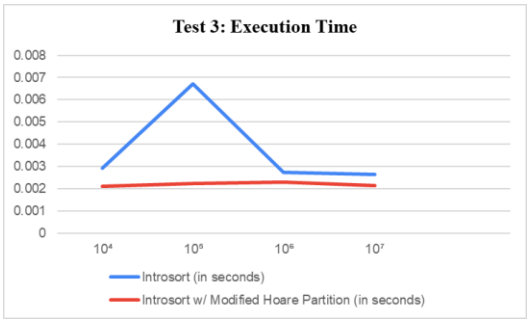


Introsort consumed more space as the dataset size increased, while the proposed Introsort with Modified Hoare partition consistently used 0 byte code space across all dataset sizes.

4th Test: Large Datasets

Data Size	**10,000**	**100,000**	**1,000,000**	**10,000,000**
Range	**1 - 1,000**	**1 - 10,000**	**1 - 100,000**	**1 - 1,000,000**
Introsort	0.002935	0.006711	0.002718	0.002651
Introsort: Modified Hoare Partition	0.002098	0.002226	0.002282	0.002134

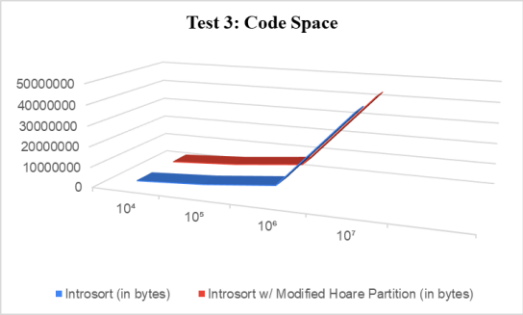
Table 4.6 Large Datasets Execution Time



There is a huge difference in performance between the two algorithms. Introsort has consistent execution times as the dataset size increases, while the proposed algorithm has higher execution times as the dataset size increases.

Data Size	**10,000**	**100,000**	**1,000,000**	**10,000,000**
Range	**1 - 1,000**	**1 - 10,000**	**1 - 100,000**	**1 - 1,000,000**
Introsort	2048000 bytes	4055040 bytes	7495680 bytes	43450368 bytes
Introsort: Modified Hoare Partition	1978368 bytes	4055040 bytes	7499776 bytes	43454464 bytes

Table 4.7 Large Datasets Code Space



Introsort used more space as the dataset size increased, while the proposed Introsort with Modified Hoare partition used 0 byte code space, showing its efficiency in managing memory resources for larger datasets.

7 CONCLUSION

The proposed Introsort with Modified Hoare Partitioning Method demonstrates superior performance by reducing code space and maintaining efficiency in sorting algorithms. This algorithm performed well for small datasets, showcasing execution times comparable to Introsort, and efficiently managing memory resources for larger datasets. However, as the dataset size increased, a slight increase in execution time was observed compared to Introsort. This study holds significant importance as it addresses the limitations identified by David Musser, thereby enhancing sorting algorithm efficiency, speed, and stability. Introsort with Modified Hoare Partitioning Method presents an alternative to traditional partitioning methods, highlighting its potential benefits for efficiently sorting large datasets.

8 LIMITATIONS AND FUTURE WORK

Limitations of this study include the need for further evaluation and comparison with a wider range of datasets and scenarios to fully assess the algorithm's performance. Additionally, the study focused on theoretical analyses and experimental measurements, but real-world applications and usability considerations were not extensively explored.

Although, the proposed algorithm shows improvements in code space utilization and performance, there may be specific dataset types or sizes where Introsort outperforms the modified version. Additionally, further empirical testing and real-world application of the algorithm are needed to validate its effectiveness across a wider range of scenarios.

9 ACKNOWLEDGMENT

The authors wish to thank A, B, C. This work was supported in part by a grant from XYZ.

REFERENCES

[1] Musser, D. (1997). Introspective Sorting and Selection Algorithms. *Journal of Software: Practice and Experience*, 27(8), 983–993.
[2] Lammich, P. (2020). Efficient Verified Implementation of Introsort and Pdqsort. *Springer Science+Business Media*, 307–323. https://doi.org/10.1007/978-3-030-51054-1_18
[3] Sharma, B., & Kumar, A. (2022). Introduction to Hybrid Algorithm. *International Journal of Advance Research and Innovative Ideas in Education*, 8(3), 4495–4501

[4] Li, S., Li, H., Liang, X., Chen, J., Giem, E., Ouyang, K., Zhao, K., Di, S., Cappello, F., & Chen, Z. (2019). FT-isort. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3295500.3356195

[5] Ferrada, H. (2022). A sorting algorithm based on ordered block insertions. *Journal of Computational Science*, *64*, 101866. https://doi.org/10.1016/j.jocs.2022.101866

[6] Abhyankar, D., & Ingle, M. (2011). Engineering of a Quicksort Partitioning Algorithm. Journal of Global Research in Computer Science, 2(2), 17–23.

[7] Abhyankar, D., & M.Ingle. (2011). A Performance Study of Some Sophisticated Partitioning Algorithms. *International Journal of Advanced Computer Science and Applications*, *2*(5). https://doi.org/10.14569/ijacsa.2011.020523

[8] https://fastercapital.com/topics/comparison-of-hybrid-sorting-algorithms.html

[9] https://aquarchitect.github.io/swift-algorithm-club/Introsort/

[10] https://stackoverflow.com/questions/10324830/how-to-get-onlogn-from-tn-2tn-2-on

[11] https://www.geeksforgeeks.org/introsort-cs-sorting-weapon/