Runtime Optimization of Modified Quicksort Using Insertion Sort

Javier Raut, Krystal Heart Bacalso, Junar Landicho

***Abstract*—Quicksort is one of the popular divide-and-conquer sorting algorithms because of its average-case complexity of $O(nlogn)$. However, it can approach $O(n^2)$ on data that is pre-sorted, reversed, or nearly sorted. A new approach by Hossain in his paper contained a new pivot selection method, along with a "Manual Sort" function. It reduced the worst-case complexity down to $O(nlogn)$. To further enhance its runtime performance, this paper integrated Insertion Sort, an algorithm known for its efficiency in sorting small datasets, as a subroutine in sorting subarrays. Three versions of the Proposed Quicksort were implemented based on the threshold set for the insertion sort function in the algorithm which are the Proposed 10, 50, and 100 Quicksort. These are then tested together with the Classical, and Hossain's Quicksort in the Uniform, Normal, Exponential, Bimodal, and Reversed data distributions of sizes 10, 100, 1000, 10000, and 100000. The results show that the Proposed 10 Quicksort is the overall fastest ranking 1st to sort in 16 datasets, with a 27.9% improved performance from Hossain's Quicksort. Further increasing the threshold for the Insertion Sort can decrease performance making the Proposed 50 and 100 quicksort improve only by 22.31% and 0.62% from Hossain's Quicksort respectively.**

***Index Terms*—Quicksort, Insertion Sort, Optimization, Sorting Algorithms, Time Complexity**

I. Introduction

QUICKSORT, is a popular sorting algorithm that uses the divide-and-conquer approach. It selects a pivot and partitions the array into two separate subarrays recursively until the data is sorted. It is highly regarded for having an $O(nlogn)$average-case time complexity. Unbalanced partitions, however, can cause its worst-case complexity to drop to $O(n^2)$ for data that has already been sorted or almost sorted. Multiple approaches has been made to try to address this issue, like based pivot selection and bidirectional partitioning [6], dynamic pivot selection [7], multi-pivot quicksort [5], however these approaches did not reach the ideal $O(nlogn)$ worst case complexity of divide-and-conquer algorithms. A recent approach made by Hossain [1] suggests a new pivot selection process by averaging the maximum and minimum values from two sub-arrays within the main array, and adding a "Manual Sort" function as a subroutine to reduce redundant comparisons when the sub-array size is less than or equal to three. This approach has dropped its worst-case complexity from $O(n^2)$ to $O(nlogn)$. The manual sort function, however, is limited by its size, which can only accommodate up to 3 elements. This leaves the algorithm to sort all subarrays with elements above 3 using quicksort. The problem with quicksort is that it performs slower in small datasets, a common occurrence to divide-and-conquer algorithms which are running in $O(nlogn)$ in contrast to other sorting algorithms like insertion sort which is a known algorithm that performs well in smaller datasets.

Insertion Sort, also called the in-place comparison sorting algorithm, is a simple sorting algorithm that works by inserting each element to its correct position in a sorted sub-list. It does this by comparing each element with the previous elements and then moving the element to its correct position by shifting the larger elements to the right. The worst-case complexity of this algorithm is $O(n^2)$. Despite the quadratic complexity, it is very efficient when sorting small datasets.

Quicksort has been slower to perform in small datasets because of its recursive overhead. There have been multiple attempts to reduce the runtime of Quicksort in smaller datasets. Saraswat [3] improved the performance of Classical Quicksort by integrating Insertion Sort as a subroutine. This approach has made a performance improvement of around 60% in smaller datasets. In that study, they only tested it on small datasets with a uniform random distribution. Considering the drawback of quicksort in small datasets, this paper aims to integrate the Insertion Sort algorithm alongside Hossain's Quicksort, in addition to the "manual sort" function utilized by the previous optimization [1] to handle the sorting on small subarrays. In theory, leveraging this algorithm will improve its overall performance. This algorithm can also make quicksort a viable option not just for large datasets, but also the small ones.

Through the proposed algorithm, the researchers will evaluate the efficacy of the combined approach, including its time complexity, runtime, and aim to determine the optimal threshold for the insertion sort and compare the proposed algorithm to Hossain's Modified Quicksort and the Classical Quicksort. With further analysis of the implications of this proposed approach, the researchers aim to gain insights into the potential performance gains in this algorithm, and a better understanding of this approach and its versatility for varying types of data distribution and sizes.

II. Review of Related Literature

Multiple studies have been conducted on optimizing the Quicksort algorithm. Different techniques like based pivot selection and bidirectional partitioning [6], dynamic pivot selection [7], multi-pivot quicksort [5]. Hossain [1] proposed a new approach to the pivot selection of the Quicksort algorithm, and introduced the manual sort method. For the pivot selection method, it divided the array into two-subarray of equal size, then find the mean of each subarray with the minimum and maximum value of each array. These values were then calculated to find the pivot element. The manual sort method

meanwhile, was only used when the sub-array size was equal or less than three. It is done by swapping the positions of each element based on its size. These optimizations improved the worst-case complexity from $O(n^2)$ to $O(nlogn)$. However, a fully detailed explanation of the methods of testing the algorithm's performance was not discussed.

The problem of maximizing the performance on small datasets of divide-and-conquer algorithms has been long explored. One of the common approaches researchers did was implementation of hybrid approach. Hybrid algorithms can utilize a simpler approach to a complex problem, thus improving the overall performance [10]. One of the popular algorithm used as a subroutine in hybrid algorithms is Insertion Sort, a very efficient algorithm in small arrays which makes it effective to use as a subroutine of Timsort /cite11, a hybrid of Merge Sort and Insertion sort. It is also good at sorting presorted data, thereby was used as part of the touch-up phase in Cache-Optimized Learned Sort /citeref11, where it sorts the remaining misplacements due to non-monotonicity. Existing approaches like Introsort [9] which used a median of three randomly chosen elements for the pivot selection, Insertion Sort and Heapsort. Another approach by Saraswat [3], which employed a hybrid of Insertion Sort and Classical Quicksort. Quinsertion solves the issue of Quicksort by using Insertion Sort to handle small-sized data. Insertion Sort takes less time to sort small and average-sized data [2], making it reasonable to be integrated with the Quicksort algorithm. Another thing of note is that these multiple algorithms have set different thresholds on when to utilize Insertion Sort to sort small-lengthed data.

When these algorithms were tested for the performance of these algorithms, both Quinsertion [3], and Hossain [1] used a randomly generated distribution. Their approach to random number distribution was not disclosed and can be challenging to replicate. Detailed testing methods of random generation with different types of number distribution were implemented by Sabah [4] where they made a comparative analysis of popular sorting algorithms which was tested using different open-source datasets like the Iris Dataset, Student Dataset, Wine Dataset, and computer-generated Uniform, Normal, Exponential, and Bimodal distribution. These datasets were sorted 10 times to minimize outliers, showcasing the strengths of each algorithm based on the variation and size of data being sorted.

Another issue with multiple papers mentioned in this study was the lack of mention and use of statistical tools in comparing the performance of their algorithms. For starters, when conducting tests in multiple datasets, Data itself is inconsistent and is not expected to be distributed in a normal manner, as mentioned in the Mostly Harmless Statistics [8], statistical methods like Paired T-Test were just not suited to compare performance across multiple algorithms with wide variations in their performance. To simplify the methods in determining the improvement of performance across multiple datasets, one of the promising statistical tools that can be used is the Wilcoxon Rank-Sum Test. Wilcoxon is non-parametric, making it robust against outliers and non-normal distributions. Another concern was the increased chance of Type-I errors when doing multiple hypothesis tests, which can happen in cases like comparing the performance of algorithms. Another tool to minimize this error is the Bonferroni Correction which scales based on the number of comparisons being made, making it ideal when trying to scale the datasets and algorithms to test.

III. Methodology

Numerous researchers have tried to reduce the worst-case complexity of Quicksort. In this study, the researchers was solely be focused on improving the runtime performance of the Quicksort algorithm, we combined Hossain's proposed Quicksort algorithm, and the classical Insertion Sort to improve the runtime when sorting smaller-sized partitions. Hossain's algorithm modified the pivot selection by dividing the array into two sub-arrays. The maximum and minimum elements of the sub-arrays were determined and their mean was calculated. The pivot element was the mean of the two means. The data is partitioned by comparing each element of the subarray with the pivot element. The loop variable increased if the element in the right subarray was smaller than the pivot. Similarly, the loop variable decreased if the element in the left subarray was larger than the pivot. In their proposed algorithm, a Manual Sort was also introduced. When the sub-array size was equal to or less than 3, the Manual Sort function was called to compare the three elements. Due to the function having no loop, there are no recursions. However, this implementation of the Manual Sort made it limited up to an sub-array of size 3. As stated in their paper, this was done to minimize the worst-case complexity of the function up to $O(n)$. To further expand the capacity of sorting the subarrays efficiently, the researchers used Insertion Sort when sorting subarrays up to a certain threshold.

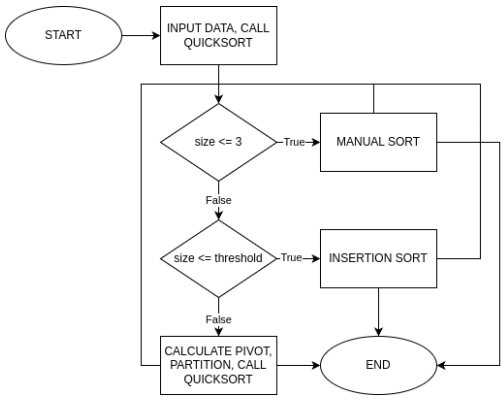

Fig. 1. Flowchart of the Proposed Algorithm

Integrating Insertion Sort when the dataset is up to a certain threshold, quickens the time to sort smaller subarrays. So when the Quick Sort function was called, it will check if the size of the current array or subarray was less than or equal to three, if so, Manual Sort was called. If not, it will check if the size was less than or equal to the threshold to check whether to call the Insertion Sort function. If the size was larger, then the pivot was calculated and the partition function was called where values were compared with the pivot. After completing all the functions, we got the sorted array.

A. *Optimal Insertion Sort Threshold*

A study conducted by Saraswat and Yadav, [3] has noted that the performance of Insertion Sort in small datasets was efficient which led them to propose the Quinsertion algorithm where they have combined the Classical Quicksort and the Classical Insertion Sort. In their paper, they used a threshold of less than 9, however, the reason behind it was not discussed thoroughly. With the lack of clarity, the researchers decided to test it at thresholds of 10, 50, 100, 500, and 1000. Through this, they determined the trends, and changes in the behavior of the algorithm in different types of thresholds, and multiple types of datasets.

B. *Proposed Algorithm*

The proposed algorithm has five portions, the quicksort, manual sort, insertion sort, calculate pivot, and partition function.

*1) **The Quicksort Algorithm**:* The quicksort function showed the process on how the algorithm deals with certain sizes of data. When the size of the data was less than or equal to three, Manual Sort was called. At sizes less than or equal to the threshold, Insertion Sort was called, and if the size was greater than the threshold, the algorithm calculated the pivot, partitioned it, and recursively called Quicksort. Manual Sort, Calculate Pivot, and Partition are based on Hossain's proposed algorithm, aimed to improve the performance, this proposed algorithm was integrated with insertion sort.

Algorithm 1 QUICKSORT

```
procedure QUICKSORT(arr, low, high)
    N ← high − low + 1
    if N ≤ 3 then
        MANUAL_SORT(arr, low, high)
    else if N ≤ Threshold then
        INSERTION_SORT(arr, low, high)
    else
        a ← CALCULATE_PIVOT(arr, low, high)
        q ← PARTITION(arr, low, high, a)
        QUICKSORT(arr, low, q)
        QUICKSORT(arr, q + 1, high)
    end if
end procedure=0
```

*2) **Manual Sort and Insertion Sort**:* When smaller-size subarrays are sorted, either of these two functions were used. For data with sizes 0, 1, or 2, Manual Sort was used. This was done by simply comparing each element to one another, and performed the necessary swaps until the data was sorted. This function was added in the previous study to sort data of length three and lower faster while running at constant time $O(n)$. In the case that the data was less than or equal to the threshold, insertion sort gets the job done. With the nature of insertion sort which iteratively sorts all data, it would approach a worst-case complexity of $O(n^2)$. This however is being limited by the threshold implemented in the algorithm, thereby reducing it to constant time $O(n)$ when interpreted with asymptotic analysis.

Algorithm 2 MANUAL_SORT

```
procedure MANUAL_SORT(arr, low, high)
    N ← high − low + 1
    if N ≤ 1 then
        return
    end if
    if N = 2 then
        if arr[low] > arr[high] then
            Exchange arr[low] ↔ arr[high]
        end if
    end if
    if arr[low] > arr[high] then
        Exchange arr[low] ↔ arr[high]
    end if
    if arr[high − 1] > arr[high] then
        Exchange arr[high − 1] ↔ arr[high]
    end if
end procedure=0
```

Algorithm 3 INSERTION_SORT

```
procedure INSERTION_SORT(arr, low, high)
    for i = low + 1 to high do
        key ← arr[i]
        j ← i
        while j ≥ low and arr[j] > key do
            arr[j + 1] ← arr[j]
            j ← j − 1
        end while
        arr[j + 1] ← key
    end for
end procedure=0
```

*3) **Pivot Calculation and Partitioning**:* Hossain's [1] approach to calculating the pivot was done by getting the mean of the minimum and maximum values of the two subarrays and then getting the mean of the means. This pivot would then be used by the Partition function to rearrange the elements if they were greater than or lesser than the pivot. This method of calculating the pivot has made a significant impact in the overall performance of the algorithm.

Algorithm 4 CALCULATE_PIVOT

```
procedure CALCULATE_PIVOT(arr, low, high)
    mid ← (low + high)/2
    leftMin ← min(arr[low], arr[mid])
    leftMax ← max(arr[low], arr[mid])
    rightMin ← min(arr[mid + 1], arr[high])
    rightMax ← max(arr[mid + 1], arr[high])
    leftMean ← (leftMin + leftMax)/2
    rightMean ← (rightMin + rightMax)/2
    pivot ← (leftMean + rightMean)/2
    return pivot
end procedure=0
```

Algorithm 5 PARTITION

```
procedure PARTITION(arr, low, high, pivot)
    i ← low − 1, j ← high + 1
    repeat
        repeat
            i ← i + 1
        until arr[i] ≥ pivot
        repeat
            j ← j − 1
        until arr[j] ≤ pivot
        if i ≥ j then
            break
        end if
    until False
    Exchange arr[i] ↔ arr[j]
end procedure=0
```

C. *Experimental Setup*

*1) **Data Gathering**:* To test the runtime performance of the algorithm, we will be compared the classical QuickSort and Hossain's Quicksort Algorithm as the control, with the new Proposed Modified Quicksort with Insertion Sort, with thresholds of 10, 50, and 100. The data used would be similar to the approach used by Sabah [4] in testing multiple types of sorting algorithms. Here, the dataset would be randomly generated with the following distribution types and sizes:

- Uniform (10, 100, 1K, 10K, 100K),
- Normal (10, 100, 1K, 10K, 100K),
- Exponential (10, 100, 1K, 10K, 100K),
- Bimodal (10, 100, 1K, 10K, 100K), and
- Reversed (10, 100, 1K, 10K, 100K)

Using the mentioned algorithms, the datasets above were be sorted ten times and record their average. This was done to reduce data outliers. The algorithms are written using C++, and the runtime will be measured with high precision using the Chrono library. The experiments were performed on a computer with the following specifications:

- Processor: AMD Ryzen 5 5600H (6 cores, 12 threads @ 4.28GHz)
- RAM: 16GB DDR4 RAM (3200 MHz)
- Operating System: Ubuntu 24.04 LTS x86_64
- Kernel: Linux 6.8.0-35-generic
- Compiler: GCC 13.2.0

With this experimental setup, the researchers were able to determine the performance of all tested algorithms across different datasets and distributions, and were able to determine the most optimal threshold in the proposed algorithm.

To determine if the Proposed Quicksort Algorithms presented in this study had a statistical significance in the overall performance of the algorithm. All 25 datasets tested in each algorithm were pooled and compared to other algorithms. In consideration to the wide variation of datasets and algorithms tested, a wide variation in the runtime was expected. The researchers utilized the Wilcoxon signed-rank test, a non-parametric two-paired test. In addition to the Wilcoxon signed-rank test, we also utilized the Bonferroni Correction to prevent the likelihood of Type I errors in this test. The test to be conducted will evaluate the null hypothesis:

H_o: There is no difference in mean runtimes between the two algorithms.

The algorithms that were compared here are the Classical Quicksort and the Proposed 10 Quicksort, Proposed 50 Quicksort, and Proposed 100 Quicksort, and Hossain's Quicksort and Proposed 10 Quicksort, Proposed 50 Quicksort, and the Proposed 100 Quicksort. Alongside the statistical test, the rate of the difference in performance in the mentioned comparisons will also be shown.

IV. Results and Discussions

In this chapter, the researchers discussed the gathered data along with its analysis. In this chapter, the researchers presented and analyze the data gathered during the course of their study. The data will be presented in a variety of formats, including tables and charts. The researchers used statistical methods to analyze the data and identify any significant trends or patterns. The results of the analysis will be discussed in detail, and the researchers has drawn conclusions based on their findings.

A. *Complexity Analysis*

1) Best Case Analysis: The Manual Sort sorts data of size of less than or equal to three. It does the sorting at $O(1)$ since it deals with a very small and fixed number of elements. The Insertion Sort on the other hand performs at $O(n)$, this takes place when the given subarray was already sorted. Lastly, the proposed quicksort algorithm takes place when the pivot splits the array into two nearly equal halves every iteration, thereby the two subarrays would be of size roughly $\frac{n}{2}$ and $\frac{n}{2} - 1$. The recurrence relation would look like:

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

The needed level of recursion to reduce the size of subarrays would be $log_2(n)$. And at each level there would be $O(n)$ work to partition the array. The time complexity of this algorithm would be $T(n) = O(nlogn)$.

2) Average Case Analysis: The time complexity of the Manual Sort would remain the same $O(1)$ in the average case, the Insertion Sort have an average case of $O(n^2)$. However, in context to the proposed algorithm where the Insertion Sort only runs when below the cap, the overall complexity in context of large data is effectively $O(1)$. The proposed quicksort algorithm may take place when the partitions are split in a reasonably balanced manner like $\frac{n}{4}$ and $\frac{3n}{4}$. The recurrence relation would look like:

TABLE I
AVERAGE SORTING TIME OF SORTING ALGORITHMS ACROSS DIFFERENT DATASETS

Dataset	Classical Quicksort	Hossain Quicksort	Proposed 10	Proposed 50	Proposed 100
Uniform 10	555.9	861	386.8	**347.7**	515.7
Uniform 100	11752.6	12667.1	7896.3	**7795.1**	27418.6
Uniform 1K	141433	145234.4	**114811.1**	134045.6	211276.4
Uniform 10K	2024231.5	2103344.4	**1403447.1**	1510219	2006211.1
Uniform 100K	23167390	21448187.8	**16477111.1**	17605396	21944659.5
Normal 10	350.6	556.2	**216.3**	255.5	300.7
Normal 100	9396	11929	**7463.2**	8269.9	24862.5
Normal 1K	137351.9	140974.2	**113309.9**	133228.2	204257
Normal 10K	1981414.1	2013027.1	**1364422.6**	1613569.8	1939238.6
Normal 100K	24406272.3	20546184	**16473071.5**	17987606.9	22941355.6
Exponential 10	450.6	805.6	**343.6**	348.9	419.8
Exponential 100	9464	12139.3	8139.5	**7255.1**	23254.6
Exponential 1K	145927.8	144009.8	**115608.4**	145163.7	209331.6
Exponential 10K	1909979.6	2063445.9	**1404884.1**	1717619.7	1891905.4
Exponential 100K	24100410.1	21060088.8	**17238851.5**	18682530.2	22202110.2
Bimodal 10	467.7	843.4	**318.6**	369.9	404.9
Bimodal 100	9680.6	12189.1	8081.4	**7649.8**	23301.5
Bimodal 1K	138825.8	145938.9	**116350.6**	139708.6	202046.7
Bimodal 10K	1926556.9	2069302.2	**1511994.1**	1709323.1	1995303.1
Bimodal 100K	24288399.6	21626058.6	**17529451.1**	18550716.9	22310811.5
Reversed 10	562.2	537	417	**392.9**	570.1
Reversed 100	45501.6	7854.8	3678.1	**1776.1**	41665.8
Reversed 1K	3698615.4	64027.8	44625.4	**32451.1**	33468.1
Reversed 10K	341142933.5	946759.7	459358.4	393705.4	**297932.2**
Reversed 100K	35451076660	8208667.2	6219785.7	3645192.7	**3613224.3**

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$$

Even if it was not balanced, the way it was split was still enough to keep the recursion depth around $log_2(n)$. Each recursion also required $O(n)$ time for partitioning. Therefore, the time complexity would be at $T(n) = O(nlogn)$.

3) Worst Case Analysis: For the worst case analysis, the Manual Sort would remain as $O(1)$, the Insertion Sort would have the same behavior the average case, where its complexity would be $O(n^2)$ but does not have significant impact to the overall runtime, thus it would run in $O(1)$ in the context of the asymptotic analysis for large sized data. For the overall algorithm, the pivot calculation consistently divides the data into an 80/20 split. The recurrence would be shown as:

$$T(n) = T(\frac{8n}{10}) + T(\frac{2n}{10}) + cn$$

This can be calculated as $log_{\frac{10}{8}}(n)$. This split performs as fast as a 50-50 split in the data, making the worst-case complexity of the algorithm as $T(n) = O(nlogn)$.

B. *Experimental Results*

We have tested five sorting algorithms in Uniform, Normal, Exponential, Bimodal, and Reversed distributions. Table I reflects the average runtime in nanoseconds spent after 10 runs of the sorting algorithms in each respective dataset.

*1) **Performance in Classical Quicksort**:* A quick review of the Classical Quicksort is that it performs on average at $O(nlogn)$, and $O(n^2)$ at its worst. The worst-case complexity of the Classical Quicksort occurs when the data was already sorted or reversed. This was shown in the test in the Reversed Distribution where it sorted it very slowly reaching 35.451 seconds in the Reversed 100K Dataset. On average, it also performed slower in the Uniform, Normal, Exponential, and Bimodal datasets. Overall, the Classical Quicksort performs closer and at times slightly faster than Hossain's Quicksort with exception in the reversed dataset as visualized in Figure 2. This weakness of the Classical Quicksort, including uneven splits can cause the overall performance of Classical Quicksort to worsen which made it lack in versatility.

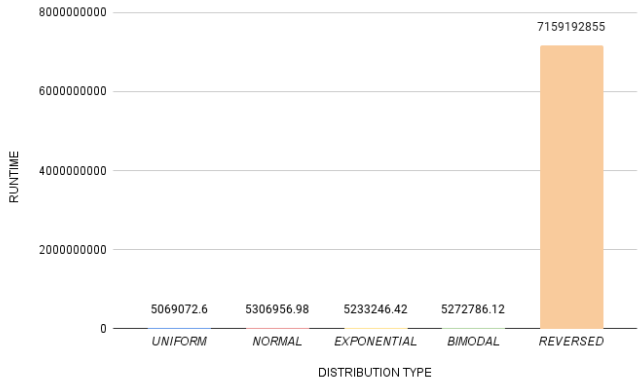

Fig. 2. Average Runtime of Classical Quicksort in Five Data Distribution Types

*2) **Performance of Hossain's Quicksort**:* Hossain has aimed to reduce the worst case complexity of Quicksort down to $O(nlogn)$. In contrast to the Classical Quicksort, it has shown a massive improvement from the Classical Quicksort specifically in the reversed dataset, this tests shown the impact of optimizing the algorithm down to $O(nlogn)$ from the Classical Quicksort's $O(n^2)$. As shown in in Figure 3, it also performed consistently close runtime in the Uniform, Normal, Exponential, and Bimodal Datasets. Overall, this made Hossain's Quicksort more reliable than the Classical Quicksort, in which the issues of uneven splits and quadratic performance in reversed data was addressed.

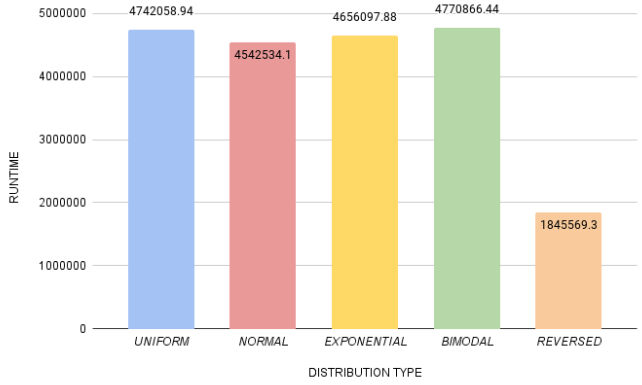

Fig. 3. Average Runtime of Hossain's Quicksort in Five Data Distribution Types

*3) **Performance of the Proposed Quicksort Algorithms**:* The utilization of Insertion Sort as a subroutine in the Proposed Quicksort Algorithms provided a significant impact to the overall performance of Hossain's Algorithm. The most promising among the three propositions was the Proposed 10, followed by Proposed 50, and lastly the Proposed 100. The Proposed 10 had consistently performed the fastest in 16 out of 25 datasets in the setup, outpacing all other algorithms. It dominated in the Normal Distribution, and performed second and third in some cases. The Proposed 50 Quicksort performed slightly slower than the Proposed 10 Quicksort. In most cases, it came in as the second fastest performer in this experiment with a very close gap between them. The slowest among the three is the Proposed 100 Quicksort. On average, it comes in as the third fastest algorithm in the test, but the impact of the large threshold of the insertion sort starts to take effect in its performance. One obvious example is its performance in all datasets of size 100. In this case, the algorithm had to rely solely on Insertion Sort, which in turn made its performance slower than other algorithms in this dataset.

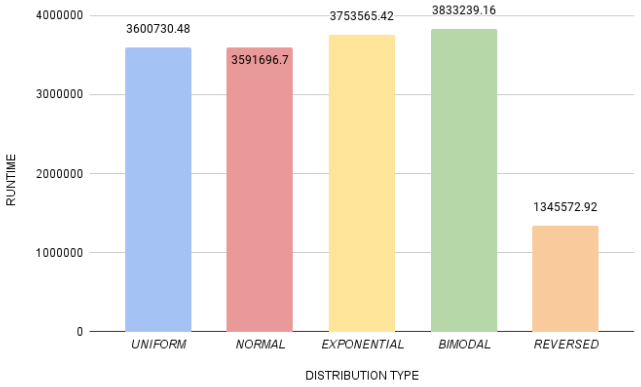

Fig. 4. Average Runtime of Proposed 10 Quicksort in Five Data Distribution Types

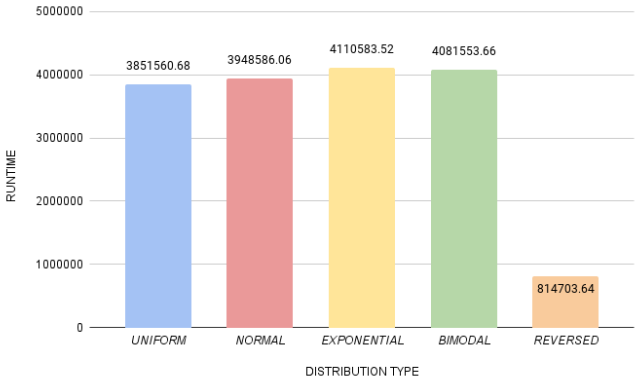

Fig. 5. Average Runtime of Proposed 50 Quicksort in Five Data Distribution Types

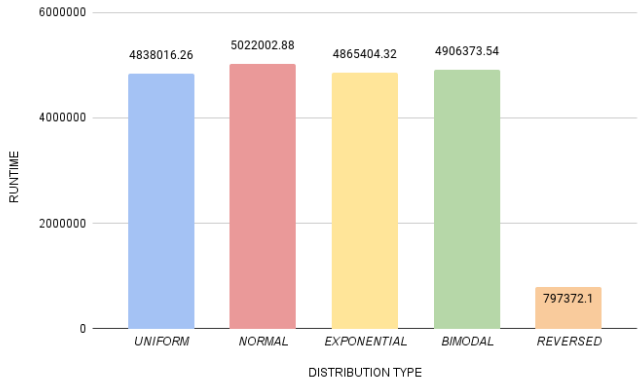

Fig. 6. Average Runtime of Proposed 100 Quicksort in Five Data Distribution Types

*4) **Statistical Analysis**:* Table II shows us the rate of improvement of the Proposed Quicksort and its p-value when compared with Classical Quicksort. For the Proposed 10 Quicksort, there is a 44428.14% backed with a corrected p-value of 1.788e-07 which means that the null hypothesis is rejected and there is a significant difference in the mean execution times between the Classical Quicksort and the Proposed 10 Quicksort. The Proposed 50 Quicksort has 42620.77% difference and a corrected p-value of 3.397e-06 that implies that the null hypothesis is rejected and there is a significant difference in the mean execution times between the Classical Quicksort and the Proposed 50 Quicksort. Meanwhile, the Proposed 100 Quicksort has 35046.19% and a corrected p-value of 0.6298 which is above the significance value of 0.05, meaning that it failed to reject the null hypothesis. The rates has a high difference because of the very slow runtime of the Classical Quicksort in the Reversed Dataset.

TABLE II
RATE OF IMPROVEMENT OF PROPOSED ALGORITHMS FROM CLASSICAL QUICKSORT ACROSS ALL DATASETS

Algorithm	Rate of Improvement	p-value
Proposed 10 Quicksort	44428.14%	1.788e-07
Proposed 50 Quicksort	42620.77%	3.397e-06
Proposed 100 Quicksort	32046.19%	0.6298

Table III shows the difference of the Proposed Algorithms from Hossain's Algorithm, the greatest improvement is at 27.9% from the Proposed 10 Quicksort which is backed by a corrected p-value 1.788e-07 and implies that there is a significant difference between Hossain's Quicksort and the Proposed 10 Quicksort. This is followed by 22.31% of the Proposed 50 Quicksort also rejecting the null hypothesis with a corrected p-value of 2.503e-06, and lastly, the little to no difference of the Proposed 100 Quicksort at only 0.62% with a corrected p-value of 1.5231, therefore there is no significant difference in the mean execution times between Hossain's Quicksort and the Proposed 100 Quicksort .

TABLE III
RATE OF IMPROVEMENT OF PROPOSED ALGORITHMS FROM HOSSAIN'S QUICKSORT ACROSS ALL DATASETS

Algorithm	Rate of Improvement	p-value
Proposed 10 Quicksort	27.49%	1.788e-07
Proposed 50 Quicksort	22.31%	2.503e-06
Proposed 100 Quicksort	0.62%	1.5231

V. LIMITATIONS OF THE STUDY

A. Dataset Generation

The data used in this study is not an accurate representation of data in real life. Despite considering distribution types in the dataset generation, realistic datasets like the one used by Sabbah [4], can be utilized to test real world application of the algorithms tested in the study.

B. Hardware and Software Environment

The data is tested specifically in a specific hardware and software configuration. Factors like, operating system, hardware specifications, and programming languages may affect the performance of the algorithms tested in this study.

C. Focus on Threshold

The study solely focused on the effects of the threshold of the insertion sort function. Other factors might also have an influence in the overall performance of the algorithm that have been overlooked.

D. Parallel Performance

Modern hardware contains multiple cores which can be utilized by any program. In this study, there is a lack of consideration with regards to parallel performance, thereby the results in this test might be different if implemented in parallel.

VI. CONCLUSION

This study reflects the influence of insertion sort as a subroutine of Hossain's Quicksort Algorithm which is being used to sort small subarrays and assess its performance across various dataset distribution types and sizes and to determine the ideal threshold of sorting using insertion sort.

The most promising among the proposed algorithms is the Proposed 10 Quicksort, which utilizes a threshold of less than or equal to 10. It has performed the fastest in 16 out of 25 datasets and is significantly faster than Classical Quicksort by 44428.14%, and 27.49% when compared to Hossain's Quicksort. Signs of deterioration can also be observed once the threshold for Insertion Sort is increased, this made the Proposed 50 Quicksort perform only at 22.31% when compared to Hossain's Quicksort, around 5% slower than the Proposed 10 Quicksort. The obvious proof of performance degradation is clearly observed on the Proposed 100 Quicksort, where it statistically failed to have a significant difference not just with Hossain's Quicksort, but also with the Classical Quicksort with a p-value of 1.5231, and 0.6298 respectively.

VII. Future Works

A. Real Life Datasets

The current approach is not fully indicative of its real life performance considering the data used is randomly generated. Testing the algorithms using datasets like MNIST dataset, Yelp dataset, used by Sabah [4] could give more accurate assessment of the algorithm's performance.

B. Different Hardware and Software Configurations

Factors like hardware, and software implementations can affect the performance of algorithms. TA more detailed testing in different hardware, and software implementations like parallelization [12] can show significant improvements in the algorithm.

C. Dynamic Threshold

As observed in the study, different thresholds can affect how fast data is sorted. One note observed is the fast performance of Proposed 100 in the Reversed 100K Dataset. Leveraging dynamic threshold can bolster the performance of the algorithm across different types of data.

References

[1] M. S. Hossain, S. Mondal, R. S. Ali, and M. Hasan. "Optimizing complexity of quicksort," In *Communications in computer and information science*, vol. 1235, pp. 329-339. Jul. 2020.

[2] A. Soomro, H. Ali, H. N. Lashari, and A. Maitlo, "Performance analysis of heap sort and insertion sort algorithm." *International Journal of Emerging Trends in Engineering Research*, vol. 9, no. 5, pp. 580-586, May 2021

[3] P. Saraswat, and A. Yadav, "Quicksertion: a hybrid sorting technique," *International Journal of Engineering Applied Sciences and Technology*, pp.45-49, Jan. 2016

[4] A. Sabah, S. Abu-Naser, Y. E. Helles, R. F., Abdallatif, F. Abu Samra, A. H. Abu Taha, N. M. Massa, and A. Hamouda, "Comparative analysis of the performance of popular sorting algorithms on datasets of different sizes and characteristics, *International Journal of Academic Engineering Research*, vol. 7, no. 6, pp. 76-84, Jun. 2023

[5] S. Kushagara, A. Lopez-Ortiz, J. I. Munro, and A. Qiao, "Multi-pivot quicksort: theory and experiments," *Society for Industrial and Applied Mathematics*, 2014

[6] R. Devi, and V. Khemchandani, "An efficient quicksort using value based pivot selection an bidirectional partitioning," *International Journal of Information Sciences and Application*, vol. 3, no. 1, pp. 25-30, 2011

[7] A. Latif, T. Kobbay, M. Alfonseca, and A. Ortega, "Enhancing quicksort algorithm using a dynamic pivot selection technique", *Wulfenia*, vol. 19, no. 10, 2012

[8] R. L. Webb, Mostly harmless statistics. LibreTexts, 2023. [Online].

[9] S. Edelkamp and A. Weiß, "QuickMergeSort: Practically efficient Constant-Factor optimal sorting," *arXiv (Cornell University)*, Jan. 2018, doi: 10.48550/arxiv.1804.10062.

[10] Y. Cheng, F. Sun, Y. Zhang, F. Tao, "Task allocation in manufacturing: a review," *Journal of Industrial Information Integration*, vol 15, Pages 207-218, 2019

[11] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," *SIGMOD/PODS '20: International Conference on Management of Data*, May 2020, doi: 10.1145/3318464.3389752.

[12] D. Langr and K. Schovánková, "CPP11sort: A parallel quicksort based on C++ threading," Concurrency and Computation, vol. 34, no. 4, Sep. 2021, doi: 10.1002/cpe.6606.

Javier Martinez Raut Born on November 29, 2003. Currently an undergrduate of Bachelor of Science in Computer Science at the University of Science and Technology of Southern Philippines - Cagayan de Oro City.

Krystal Heart Monteza Bacalso Born on June 10, 2003. Currently an undergraduate of Bachelor of Science in Computer Science at the University of Science and Technology of Southern Philippines - Cagayan de Oro City.

Junar Arciete Landicho received PhD in Information Management from the Asian Institute of Technology, Thailand. His research interest is in the field of database system, e-commerce, mobile application and healthcare informatics.