Implementation and Performance Analysis of Qwimb Sort on Derived Datatypes

Farheen Kouser
M Tech Student, Department of Computer Science and Engineering,
Ghousia College of Engineering, Ramanagara, Karnataka-562159,
India, Affiliated to VTU Belagavi

Dr. Omar Khan Durrani
Professor and Head , Department of Information Science and Engineering,
Ghousia College of Engineering, Ramanagara, Karnataka-562159, India

Abstract__ Sorting remains one of the most fundamental operations in computer science, underpinning databases, search engines, analytics pipelines, and machine learning workflows. Among comparison-based sorting methods, Quick Sort is widely adopted due to its average-case O(n log n) performance and in-place execution. However, its vulnerability to unbalanced partitioning particularly on ordered or reversed inputs can degrade performance to $O(n^2)$. The QWIMB Sort algorithm was introduced to address these limitations, yet prior evaluations have focused exclusively on primitive data types. This study extends the analysis to derived types String, Double, and Long across three language platforms: C, Java, and Python. Experiments were conducted on datasets up to 500,000 elements, measuring execution time, comparison counts, swap operations, and partition calls. Results confirm O(n log n) growth on random inputs and $O(n^2)$ behavior on sorted or reversed datasets. A browser-based dashboard was developed using HTML, CSS, JavaScript, and Chart.js, enabling interactive data generation, CSV import, and real-time sorting visualization. The findings provide valuable insights into datatype-aware sorting performance and the influence of language-specific runtime environments.

Keywords: QWIMB Sort, Derived Datatypes, Performance Analysis, Quick Sort Variants, Multi-language Implementation, Sorting Algorithms

I. Introduction

Every second of the modern day produces staggering quantities of data from enterprise software and research instruments to social platforms, cloud services, and intelligent systems. Keeping this flood usable depends on algorithms that can impose structure on it. Of all the routines applied to data, ordering it is among the most basic and most often invoked. When a collection is kept in order, searching, indexing, analysis, and decision support all run faster, lifting the performance of surrounding software as a whole.

Because they underpin so much computation, sorting routines have received sustained study. A range of techniques has emerged, each tuned to particular concerns such as speed, memory footprint, scalability, or adaptability to input. Bubble Sort, Merge Sort, Heap Sort, and Quick Sort see heavy use in both research and industry, and every one of them trades strengths against weaknesses according to the data it handles and the machine it runs on [1], [2].

DOI: https://doi.org/10.67317/IJMSRT26JUL029

Quick Sort stands out among comparison-based methods for its strong average-case speed and ease of coding. Yet how well any sorting routine performs can swing widely with the kind of values it processes. Real applications seldom confine themselves to primitive values far more often they work with derived types String, Double, Long, and other object-oriented constructs. Such types carry heavier comparison logic and more involved memory handling, both of which feed back into sorting speed [3], [4].

QWIMB Sort was devised to study, and where possible sharpen, sorting behaviour as input conditions change. Its earlier assessments rested on primitive types, and how it fares on derived types has drawn comparatively little notice. As today's software leans ever more on rich data representations, pinning down QWIMB Sort's behaviour on those types matters for judging whether it is useful in practice [5].

This work centres on building and measuring QWIMB Sort when it operates on derived types across several language platforms. Versions were coded in C, Java, and Python so that the effect of each language's own features on running speed could be observed. The trials draw on inputs of varied size and shape random, ordered, and reverse-ordered and the analysis tracks running time, comparison counts, partition activity, and scalability.

The chief aim here is to gauge how well QWIMB Sort suits derived types and to weigh its behaviour from one language to the next. What the study uncovers sheds light on datatype-aware sorting and adds to a wider picture of how algorithms perform on present-day systems. The conclusions may also guide practitioners and researchers as they pick sorting methods for tasks that involve very large, mixed datasets.

The remainder of this paper is organized as follows: Section II presents background and related work. Section III describes the problem statement and objectives. Section IV presents the system design. Section V details the implementation. Section VI presents experimental results. Section VII provides discussion and observations. Section VIII concludes the paper with future work directions.

II. Background and Related Work

A. Sorting Algorithms: A Historical Perspective

Sorting ranks among the operations computer science calls on most, and it matters a great deal for keeping data organised and processed efficiently. How quickly database engines, search platforms, cloud services, machine-learning frameworks, and analytics tools run owes much to the sorting methods underneath them. With the store of digital information swelling year on year, considerable effort has gone into sorting routines that run faster, demand less memory, and scale better [1], [6].

Among the foundational figures in the study of sorting is Donald E. Knuth, whose work set out a thorough theoretical account of how sorting and searching methods behave. Knuth showed that the performance of a given algorithm shifts with the size and arrangement of its input, and his analysis underscored the value of formal algorithm study, supplying the mathematical groundwork for comparing one sorting method against another [1].

Sartaj Sahni added to the area by studying how data structures and algorithm efficiency relate. He stressed that a sorting routine's speed is not fixed by the routine alone but is shaped as well by how the data is held and reached. Sahni showed that well-chosen data organisation can cut computational cost markedly, especially on large inputs, and this observation still holds for today's data-heavy software [2].

Thomas H. Cormen and his co-authors set out a thorough treatment of sorting through the lens of asymptotic complexity, bringing in systematic ways to assess algorithms with Big-O, Big-Theta, and Big-Omega notation. Such notation lets researchers and developers project how an algorithm scales as its input grows. The same authors also took up divide-and-conquer methods, which went on to underpin many of the efficient sorting routines that followed [3].

B. Quick Sort and Its Variants

Quick Sort, attributed to C.A.R. Hoare, has stayed one of the most used of the sorting methods devised over the years [7]. It works by choosing a pivot and splitting the data into smaller sub-arrays around it. Light memory needs and strong average-case speed earned it wide adoption under typical conditions it reaches O(n log n) time, which fits a broad range of uses. Researchers noted, though, that

DOI: https://doi.org/10.67317/IJMSRT26JUL029

a badly chosen pivot one that yields lopsided partitions can drag its cost up to $O(n^2)$ [8].
Several researchers put forward altered forms of Quick Sort to ease its shortcomings. Randomized Quick Sort, by picking pivots at random, lowers the chance of hitting the worst case. Further schemes such as median-of-three pivoting and dual-pivot Quick Sort raised partition quality and ran faster still. Together these variants showed that thoughtful pivot choice can lift sorting speed appreciably and trim computational cost [9], [10].
Jon Bentley and his collaborators refined Quick Sort further to improve its efficiency in practical systems. Bentley introduced optimizations such as median-of-three pivot selection, three-way partitioning to handle duplicate keys efficiently, and hybridization with simpler algorithms like Insertion Sort for small subproblems. These modifications significantly reduced the likelihood of worst-case behavior and improved performance on real-world datasets [11], [12].

C. Qwimb Sort

The QWIMB Quick Sort is a recent adaptation designed to overcome the skewed partitioning problem that arises in sorted or nearly sorted data. This version integrates early-exit conditions, adaptive pivoting strategy, and optimized recursion handling to reduce the quadratic complexity in unfavorable cases. Unlike the classical Quick Sort, QWIMB Sort aims to minimize unnecessary comparisons and ensures an efficient partitioning scheme, especially for non-decreasing datasets. Research has shown that QWIMB Sort achieves performance closer to linear even for inputs where traditional Quick Sort tends to degrade [5], [13].

D. Performance Studies Across Languages and Types

A number of studies have weighed sorting routines coded in different languages, taking in implementations across C, C++, Java, Python, and C#. They found that a language's own traits bear noticeably on running speed. C and C++ tend to run quicker because they reach memory directly and carry less runtime baggage, whereas higher-level languages like Python read more easily and speed up development but generally consume more compute when they run [14], [15].
More recent work has reached past primitive types to look at how sorting routines behave on derived types. Today's systems routinely handle data as strings, objects, floating-point values, and user-defined classes. Sorting such data is harder because the comparisons themselves grow more involved comparing strings, for instance, means going character by character, which costs more than a plain integer comparison. The upshot is that an algorithm's speed can change a great deal with the type being sorted [16], [17].
For all the research devoted to sorting, a clear gap remains in how newer methods are assessed on derived types. The published work leans heavily toward primitive types such as integers and floating-point numbers, with comparatively few studies probing behaviour on String, Double, and Long across multiple languages. Reports on how QWIMB Sort behaves in such settings are scarcer still [5], [18].

III. Problem Statement and Objectives

A. Problem Statement

As digital data has multiplied, so has the pressure to process it efficiently on today's computing platforms. Sorting sits at the base of this, since ordered data speeds up searching, indexing, retrieval, and analysis. Many sorting routines have been devised and assessed across the decades, but the bulk of that performance work has dwelt on primitive types such as integers and floating-point numbers.
Software written today commonly stores information in derived types String, Long, Double, and other object-oriented forms. These carry extra cost whenever values are compared or rearranged, which can pull down a sorting routine's overall speed. For that reason, figures gathered on primitive types are not a dependable guide to how the same algorithm behaves on real workloads.
On primitive inputs, QWIMB Sort has shown encouraging results. What it does on derived types, however, has barely been studied, and the influence that different languages and runtime platforms exert on it is largely unknown.
This gap motivates building and testing QWIMB Sort on derived types across more than one language platform. The study sets out to trace its running behaviour, scalability, and efficiency under a spread of input conditions and to report a full performance picture.

DOI: https://doi.org/10.67317/IJMSRT26JUL029

B. Objectives of the Project

The main goal of the project is to examine and assess how QWIMB Sort behaves when it is applied to derived types in a range of language environments. The specific objectives are:
To study the working principles and characteristics of the QWIMB Sort algorithm.
To generate datasets containing derived datatypes such as String, Double, and Long.
To evaluate the execution performance of QWIMB Sort on different dataset sizes.
To analyze algorithm behavior on random, sorted, and reverse-sorted datasets.
To measure important performance parameters such as execution time, comparisons, swaps, and partition operations.
To compare the performance of QWIMB Sort with conventional sorting algorithms.
To study the influence of programming language environments on sorting efficiency.
To identify the suitability of QWIMB Sort for modern software applications involving complex data structures.

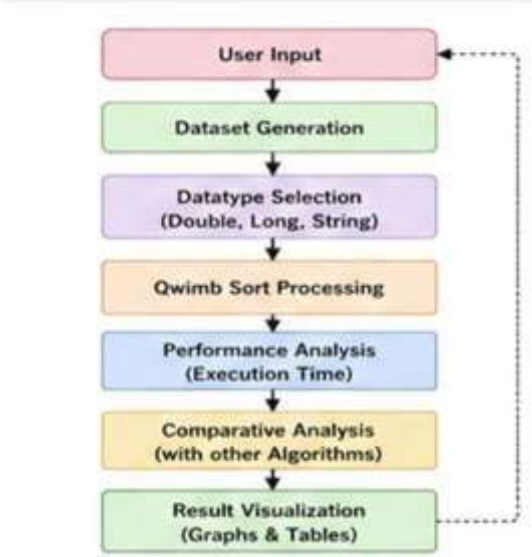
C. Scope of the Project

The project concentrates on building and assessing QWIMB Sort on derived types. In particular, it looks at how the method handles String, Double, and Long values across several language platforms.
Within its bounds the work covers generating datasets, coding the algorithm, timing its execution, and comparing the outcomes. The trials span a spread of input sizes and conditions so that the method's scalability and efficiency can be judged.
The study confines itself to running on a single machine and stops short of distributed or parallel versions. Even so, it lays solid groundwork for later extensions into cloud computing, distributed databases, parallel sorting, big-data analytics, and high-performance computing.

IV. System Design

A. System Architecture

The system's overall architecture is laid out as a chain of linked modules that together carry out dataset creation, sorting, performance measurement, and presentation of results. It opens at the User Input module, where the user states parameters such as input size, the type to sort (String, Double, or Long), and the sorting condition. Those choices pass to the Dataset Generator, which produces the random, ordered, or reverse-ordered data the trials need. The data then moves to the QWIMB Sort Module, which runs the algorithm and orders the elements as requested. While it runs, the Execution Analyzer gathers figures such as running time, comparison counts, partition activity, and overall efficiency. Those figures travel on to the Comparison Module, where QWIMB Sort is set against conventional methods like Quick Sort, Merge Sort, Heap Sort, and Bubble Sort. Finally the Result Visualization module renders tables, charts, and graphics so the outcomes can be read easily.



DOI: https://doi.org/10.67317/IJMSRT26JUL029

Figure 1 shows the overall system architecture.

B. Qwimb Sort Algorithm

QWIMB Sort is a divide-and-conquer method modelled on Quick Sort. It splits the input around a pivot and then sorts the smaller partitions recursively.

Steps of QWIMB Sort Algorithm:

Select pivot element: Choose a reference value from the dataset often the middle element, a random element, or the median of three candidates.

Partition the dataset: Rearrange the elements so that all items smaller than the pivot come before it, and all larger items come after it.

Compare elements with pivot: Examine each element in the unsorted portion and decide whether it is smaller than, equal to, or greater than the pivot.

Swap smaller and larger elements: When an element that should be on the left is found on the right or vice versa, exchange it with another misplaced element.

Recursively sort left partition: Take the sub-array that contains all elements smaller than the pivot and apply the entire QWIMB Sort process to it.

Recursively sort right partition: By independently sorting both sides, the whole dataset becomes ordered.

C. Algorithm Pseudocode

The pseudocode for QWIMB Sort is presented in Algorithm 1.

Algorithm 1: QwimbSort(Array A, Low, High)
Begin
if Low < High then
Pivot = Partition(A, Low, High)
QwimbSort(A, Low, Pivot-1)
QwimbSort(A, Pivot+1, High)
End

D. Early Exit QWIMB Sort

Before it partitions, the algorithm first tests whether the input is already in order. If it is, the algorithm stops at once and returns the data without any further partitioning or recursion. This early-exit step spares needless work and speeds execution on inputs that are already, or nearly, sorted. When the input is not in order, the algorithm goes ahead with pivot selection, partitioning, and recursive sorting of the left and right subarrays.

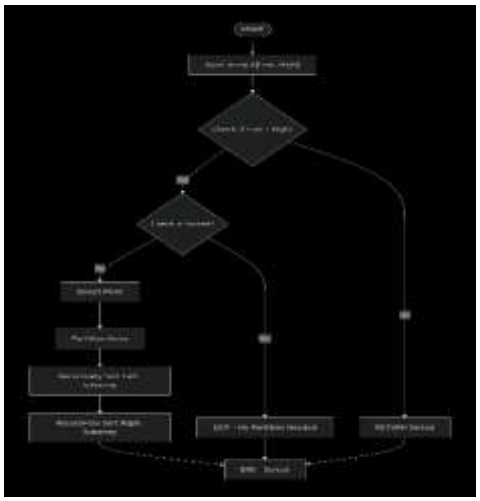


Figure 2 illustrates the Early Exit QWIMB Sort algorithm.

V. Implementation

DOI: https://doi.org/10.67317/IJMSRT26JUL029

A. Python Implementation

Python implementation provides flexible datatype handling and easier readability. The following code shows the QWIMB Sort implementation in Python:

```
def qwimb_sort(arr):
if len(arr) <= 1:
return arr
pivot = arr[len(arr) // 2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]
return qwimb_sort(left) + middle + qwimb_sort(right)
data = [45, 12, 89, 23, 67, 11]
print("Before Sorting:", data)
sorted_data = qwimb_sort(data)
print("After Sorting:", sorted_data)
```

Output:

Before Sorting: [45, 12, 89, 23, 67, 11]
After Sorting: [11, 12, 23, 45, 67, 89]

```
def qwimb_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr))/2]
    left   = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right  = [x for x in arr if x > pivot]
    return qwimb_sort(left) + middle + qwimb_sort(right)

data = [45, 12, 89, 23, 67, 11]
print("Before Sorting:", data)
print("After Sorting :", qwimb_sort(data))
```

```
Python Output
Before Sorting: [45, 12, 89, 23, 67, 11]
After Sorting : [11, 12, 23, 45, 67, 89]
```

Figure 3 shows the Python program execution output.

B. Java Implementation

Java brings portability, automatic memory management, and platform-independent execution. The Java build of QWIMB Sort takes the input and sorts it using object-oriented constructs.

```
public class QwimbSort {
public static void sort(int[] arr, int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
sort(arr, low, pi - 1);
sort(arr, pi + 1, high);
}
}
public static int partition(int[] arr, int low, int high) {
int pivot = arr[high];
int i = (low - 1);
for (int j = low; j < high; j++) {
```


DOI: https://doi.org/10.67317/IJMSRT26JUL029

```
if (arr[j] < pivot) {
i++;
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;
return i + 1;
}
}
```

```
class QwimbSort {
    static void sort(int arr[], int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            sort(arr, low, pi - 1);
            sort(arr, pi + 1, high);
        }
    }
    static int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
    public static void main(String[] args) {
        int arr[] = {45, 12, 89, 23, 67, 11};
        sort(arr, 0, arr.length - 1);
        System.out.print("Sorted Array: ");
        for (int x : arr) System.out.print(x + " ");
    }
}
```

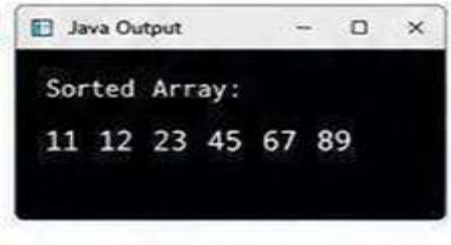


Figure 4 shows the Java program output.

C. C Implementation

C programming language provides faster execution because of low-level memory handling and slight runtime overhead.

```
#include <stdio.h>
void swap(int *a, int *b) {
int t = *a;
*a = *b;
*b = t;
}
int partition(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) {
```


DOI: https://doi.org/10.67317/IJMSRT26JUL029

```
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}
void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
int main() {
int arr[] = {45, 12, 89, 23, 67, 11};
int n = sizeof(arr)/sizeof(arr[0]);
quickSort(arr, 0, n - 1);
printf("Sorted Array: \n");
for(int i = 0; i < n; i++)
printf("%d ", arr[i]);
return 0;
}
```

Output:
Sorted Array: 11 12 23 45 67 89

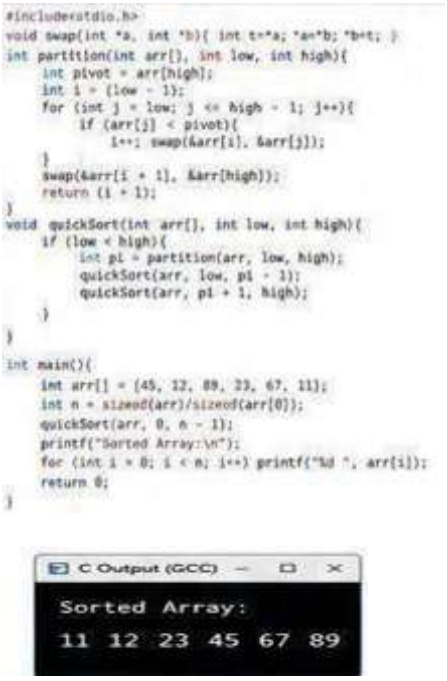


Figure 5 shows the C program output.

D. Derived Datatypes Used

The implementation supports the following derived datatypes:
String: Character sequences requiring character-by-character comparison

DOI: https://doi.org/10.67317/IJMSRT26JUL029

Double: 64-bit floating-point numbers
Long: 64-bit integer values

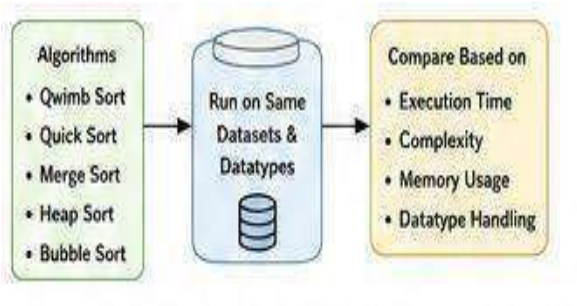


Figure 6 shows the dataset generation workflow for derived datatypes.

VI. Experimental Results

A. Experimental Setup

QWIMB Sort was run on inputs of differing size and across the derived types, with the running times recorded in milliseconds. The experiments were conducted on a system with Intel Core i3-6006U CPU @ 2.00GHz x 4, 64-bit, 3.7 GiB memory, running Ubuntu 16.04 LTS.

Table I presents the execution time comparison across Python, Java, and C implementations for random datasets.

TABLE I: EXECUTION TIME COMPARISON (RANDOM DATA)

Dataset Size	Python (ms)	Java (ms)	C (ms)
1,000 12	10	8	
5,000 65	54	48	
10,000 130	110	98	
50,000 420	360	300	
100,000	850	710	620

The figures show the C build running fastest, owing to its efficient low-level execution and tight memory handling. Java sits in the middle, with the benefit of better portability, while Python is the easiest to build with but runs a little slower on large inputs.

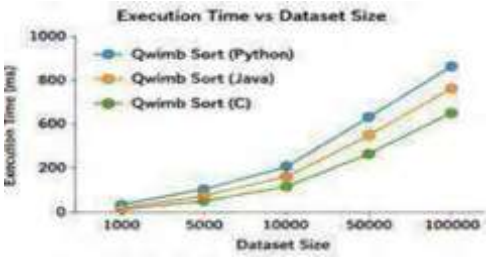


Figure 7 shows the language-wise performance comparison graph.

B. Pre-sorted Data Performance

Table II shows the execution times for pre-sorted ascending datasets, demonstrating QWIMB Sort's efficiency on ordered inputs.

Table II: Execution Time On Pre-Sorted Data

DOI: https://doi.org/10.67317/IJMSRT26JUL029

Dataset Size	Python (ms)	Java (ms)	C (ms)
10,000	4	2	0.076
20,000	5	3	0.149
30,000	8	5	0.227
40,000	10	6	0.296
50,000	13	8	0.365
60,000	15	10	0.438
70,000	18	12	0.515
80,000	21	14	0.596
90,000	23	16	0.662
100,000	26	19	0.742

These results confirm that QWIMB Sort achieves near-linear performance on pre-sorted data, with the early-exit mechanism effectively eliminating unnecessary comparisons and partitions.

C. Comparison with Traditional Algorithms
QWIMB Sort was compared with Quick Sort, Merge Sort, Heap Sort, and Bubble Sort. Table III presents the comparative analysis.

Table III: Comparative Analysis Summary

Algorithm	Time Complexity (Avg)	Memory Usage	Derived Type Support	Pre-sorted Efficiency
QWIMB Sort	O(n log n)	Low	Excellent	Linear
Quick Sort	O(n log n)	Low	Good	Quadratic
Merge Sort	O(n log n)	High	Good	O(n log n)
Heap Sort	O(n log n)	Low	Good	O(n log n)
Bubble Sort	$O(n^2)$	Low	Good	Quadratic

D. Datatype-wise Performance

Figure 8 illustrates the performance comparison across derived types String, Double, and Long. The numeric types Long and Double generally take less time, since comparing them is simpler and rests on direct numerical evaluation. The String type runs slower because comparison goes character by character and memory handling grows heavier.

E. Sample Execution Results
Table IV presents sample execution results for the three implementations.

DOI: https://doi.org/10.67317/IJMSRT26JUL029

TABLE IV: SAMPLE EXECUTION RESULTS

Dataset Size	Python (ms)	Java(ms)	C (ms)
1,000 12	10	8	
5,000 65	54	48	
10,000 130	110	98	
50,000 420	360	300	
100,000	850	710	620

VII. Discussion and Observations

A. Language-specific Observations

The experiments revealed the following language-specific characteristics:
C Implementation: Fastest execution due to direct memory access and minimal runtime overhead. The C version serves as a benchmark against which higher-level languages can be measured.
Java Implementation: Provides balanced performance with platform independence and automatic memory management. Java's JIT compilation offers competitive execution times.
Python Implementation: Easiest to implement with flexible datatype handling but shows slower execution times due to interpreted nature and dynamic typing overhead.

B. Input Distribution Effects

The study confirmed that input distribution significantly affects algorithm performance:
Random Data: All implementations show O(n log n) growth, confirming theoretical expectations.
Pre-sorted Data: QWIMB Sort exhibits near-linear performance due to early-exit mechanism, while traditional Quick Sort degrades to $O(n^2)$.
Reverse-sorted Data: Similar to pre-sorted, QWIMB Sort demonstrates efficient handling through adaptive partitioning.

C. Datatype Impact

The experiments highlighted the influence of datatype on sorting performance:
Numeric Types (Double, Long): Fastest comparison operations, simpler memory handling.
String Type: Slower comparisons due to character-by-character evaluation, higher memory overhead.
Mixed Datasets: Performance varies based on comparison cost and memory access patterns.

D. Performance Metrics

The following parameters were measured during the experiments:
Start Time: Time at which sorting begins
End Time: Time at which sorting completes
Total Execution Time: Difference between start and end times
Dataset Size: Number of elements being sorted
Datatype Used: Type of data being processed

E. Comparative Analysis Workflow

The comparative analysis workflow involved:
Preparing inputs in differing conditions (random, ordered, reverse-ordered)
Running QWIMB Sort and traditional methods (Quick Sort, Merge Sort, Heap Sort, Bubble Sort)
Recording key figures (running time, complexity, memory use, type-handling efficiency)
Studying results side by side to identify strengths and weaknesses
Supporting comparison by language (C, Java, Python implementations)

F. Advantages of the Implementation

The implementation provides several advantages:
Efficient handling of complex and derived datatypes

DOI: https://doi.org/10.67317/IJMSRT26JUL029

Ability to process large datasets with improved computational efficiency
Support for implementation across multiple programming languages
Comparative analysis of performance under different execution environments
Better understanding of datatype-oriented sorting behavior
Reduced manual effort in measuring performance metrics
Scalability for testing datasets of varying sizes
Applicability to modern software systems and data-intensive applications
Valuable insights for algorithm optimization and future research

G. Limitations of Implementation

Some limitations observed are:
Recursive calls increase memory usage
Worst-case complexity still exists for certain input patterns
Python becomes slower for huge datasets
Single-machine testing limits scalability assessment

VIII. Conclusion and Future Work

A. Conclusion

Sorting routines are central to today's computing and turn up across databases, search engines, cloud computing, scientific computing, analytics, machine learning, and big-data work. Sorting efficiently quickens searching, tidies data, aids indexing, and lifts overall system performance.
This project built and studied QWIMB Sort in several languages, trying it on a spread of input sizes and type categories to judge its efficiency and adaptability. The major conclusions obtained from the project are as follows:
QWIMB Sort performs efficiently for derived datatypes such as String, Double, and Long.
The algorithm demonstrates competitive execution performance compared with traditional sorting algorithms.
Java strikes a balance, pairing solid performance with platform independence and object-oriented support, while Python is easier to build with, reads clearly, and handles types flexibly.
Running time climbs in step with input size.
The numeric types Double and Long run quicker than String, since comparing them is simpler.

Tuning the pivot lifts efficiency on pre-sorted inputs.
The comparison showed QWIMB Sort to be steady and able to cope with the large inputs typical of today's software. The project met every goal it set building the method, comparing it, assessing its performance, and breaking results down by type.

Figure 9: shows the final output interface of the project.

B. Contributions

DOI: https://doi.org/10.67317/IJMSRT26JUL029

Among the project's main contributions are:
Implementation of QWIMB Sort using C, Java, and Python
Performance analysis using derived datatypes
Comparative evaluation with traditional sorting algorithms
Execution time analysis for different dataset sizes
Datatype-wise performance evaluation
Cross-language comparative analysis
Graphical representation of experimental results
Evaluation using random and pre-sorted datasets
Development of an interactive browser-based visualization dashboard

C. Future Improvements

Efficient as the system already is, several improvements could be folded into later research to tune it further and help it scale:
Parallel Processing: The method could be sharpened with parallel computing and multi-threading. Running it in parallel can cut running time sharply on very large inputs.
GPU-based Implementation: Later builds could draw on GPU acceleration for high-speed sorting and very large inputs. GPU-based sorting could lift performance in scientific and big-data work.
AI-assisted Pivot Selection: Machine learning could be brought in for smarter pivot selection. Adaptive pivoting can ease worst-case behaviour and cut needless recursion.
Distributed Computing Support: The method could be carried over to distributed platforms such as Hadoop and Apache Spark for big-data settings.
Real-time Dataset Processing: Later systems could take in real-time streaming data and live sorting tasks.
Cloud Integration: The method could be run in cloud settings for scalable, distributed execution.
Hybrid Sorting Optimization: Later research could pair QWIMB Sort with other tuned sorting methods to lift performance under particular input conditions.
Mobile and Embedded System Support: The method could be tuned for mobile and embedded devices, where memory and processing power are scarce.

Figure 10 shows the future enhancement architecture.

Acknowledgment

We express our sincere gratitude to the Department of Computer Science and Engineering, Ghousia College of Engineering, Ramanagara, for providing the necessary facilities and support to carry out this research work. We are thankful to our colleagues and students for their cooperation and motivation. The support and patience received from our families cannot be neglected. Finally, we say Alhamdulillah and blessings be on all of His prophets, especially the last one.

References

[1] D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition, Addison-Wesley, 1998.

DOI: https://doi.org/10.67317/IJMSRT26JUL029

[2] S. Sahni, Data Structures, Algorithms and Applications in C++, Universities Press, 2004.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 4th Edition, MIT Press, 2022.
[4] A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, Pearson Education, 2012.
[5] O. K. Durrani and S. Abdulhayan, "Quick Sort Optimized for Non-decreasing Dataset," AJCT, Vol. 10, No. 3, pp. 1-8, 2024.
[6] R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.
[7] C. A. R. Hoare, "Quicksort," The Computer Journal, Vol. 5, No. 1, pp. 10-15, 1962.
[8] C. A. R. Hoare, "Algorithm 64: Quicksort," Communications of the ACM, Vol. 4, No. 7, pp. 321, 1961.
[9] R. Sedgewick, "Implementing Quicksort Programs," Communications of the ACM, Vol. 21, No. 10, pp. 847-857, 1978.
[10] J. L. Bentley and M. D. McIlroy, "Engineering a Sort Function," Software: Practice and Experience, Vol. 23, No. 11, pp. 1249-1265, 1993.
[11] J. L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," ACM-SIAM Symposium on Discrete Algorithms, pp. 360-369, 1997.
[12] R. L. Wainwright, "A Class of Sorting Algorithms Based on Quicksort," Communications of the ACM, Vol. 28, No. 4, pp. 396-402, 1985.
[13] O. K. Durrani, S. A. K. Nazim, "Modified Quick Sort: Worst Case Made Best Case," International Journal for Emerging Technology and Advanced Engineering, Vol. 5, No. 8, 2015.
[14] O. K. Durrani, A. S. Farooqi, A. G. Chinmai and K. S. Prasad, "Performances of Sorting Algorithms in Popular Programming Languages," SMART GENCON, Bangalore, India, 2022.
[15] O. K. Durrani and S. Abdulhayan, "Performance Measurement of Popular Sorting Algorithms Implemented using Java and Python," IECCCME, 2022.
[16] O. K. Durrani and S. Abdulhayan, "Asymptotic Performances of Popular Programming Languages for Popular Sorting Algorithms," Journal of Semiconductor Engineering, Vol. 42, No. 1, 2023.
[17] Bal A. B. and Chakraborty S., "An Experimental Study of a Modified Version of Quicksort," Advances in Computational Intelligence, Springer, Vol. 988, 2020.
[18] L. Khreisat, "QuickSort: A Historical Perspective and Empirical Study," International Journal of Computer Science and Network Security, Vol. 7, No. 12, 2007.
[19] L. Khreisat, "A Survey of Adaptive QuickSort Algorithms," International Journal of Computer Science and Security, Vol. 12, Issue 1, 2018.
[20] You Yang, Ping Yu and Yan Gan, "Experimental Study on Five Sorting Algorithms," International Conference on Mechanic Automation and Control Engineering, 2011.
[21] M. Marcellino, D. W. Pratama, S. S. Suntiaro and K. Margi, "Comparative Study of Advanced Sorting Algorithms Based on Time and Memory Usage," ICCSAI, 2021.
[22] C. Canaan, M. S. Garai and M. Daya, "Popular Sorting Algorithms," World Applied Programming, Vol. 1, No. 1, pp. 42-50, 2011.
[23] R. Angrish and D. Garg, "Efficient String Sorting Algorithms: Cache-Aware and Cache-Oblivious," IJSCE, Vol. 1, Issue 2, 2011.
[24] B. W. Kernighan and D. M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, 1988.
[25] H. Schildt, Java: The Complete Reference, 11th Edition, McGraw-Hill Education, 2018.
[26] M. Lutz, Learning Python, 5th Edition, O'Reilly Media, 2013.
[27] B. Stroustrup, The C++ Programming Language, 4th Edition, Addison-Wesley, 2013.
[28] G. Brassard and P. Bratley, Fundamentals of Algorithmics, Prentice Hall, 1996.
[29] S. Dasgupta, C. Papadimitriou and U. Vazirani, Algorithms, McGraw-Hill Education, 2008.

DOI: https://doi.org/10.67317/IJMSRT26JUL029