Optimizing Heap Sort for Repeated Values: A Modified Approach to Improve Efficiency in Duplicate-Heavy Data Sets

Japheth Kodua Wiredu, Ivan Aabaah, Reuben Wiredu Acheampong

DOI: http://dx.doi.org/10.26483/ijarcs.v15i6.7152

Volume 15, No. 6, November-December 2024

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Optimizing Heap Sort for Repeated Values: A Modified Approach to Improve Efficiency in Duplicate-Heavy Data Sets

Japheth Kodua Wiredu
Programme Coordinator, Department of Computer Science
Regentropfen University College (RUC)
Upper East Region, Ghana
https://orcid.org/0009-0008-0313-5011

Iven Aabaah
Department of Information Systems and Technology
C. K. Tedam University of Technology and Applied Sciences (CKT-UTAS)
Navrongo, Ghana

Reuben Wiredu Acheampong
Department of Information Systems and Technology
C. K. Tedam University of Technology and Applied Sciences (CKT-UTAS)
Navrongo, Ghana

Abstract: Sorting algorithms are critical to various computer science applications, including database management, big data analytics, and real-time systems. While Heap Sort is a widely used comparison-based sorting algorithm, its efficiency significantly diminishes when dealing with data sets containing a high volume of duplicate values. To address this limitation, this paper introduces a modified Heap Sort algorithm optimized for duplicate-heavy data. The proposed modification detects and handles duplicate values more efficiently by reducing unnecessary comparisons and swaps at the root of the heap and restructuring the heap more strategically. Experimental results demonstrate that the modified Heap Sort achieves up to a 15% reduction in sorting time, a 30% decrease in the number of swaps, and a 10% reduction in comparisons when tested on data sets with varying levels of duplication. These improvements highlight the enhanced computational efficiency and scalability of the modified algorithm in duplicate-heavy data scenarios. This advancement offers significant potential for improving sorting performance in practical domains such as big data analytics, database operations, and real-time data processing.

Keywords: Heap Sort, Optimization, Duplicates, Sorting Algorithms, Algorithm Efficiency, Comparisons

I. INTRODUCTION

Sorting algorithms are fundamental to a wide array of computational tasks, serving as essential building blocks in applications ranging from data analysis to database management systems (DBMS) [1]. Among these algorithms, Heap Sort stands out as an efficient comparison-based method, renowned for its in-place sorting capability and $O\ (n\ log\ n)$ time complexity [2]. While Heap Sort performs well in general-purpose scenarios, its efficiency diminishes when handling data sets with a high frequency of repeated values a common feature in real-world applications. In such cases, the algorithm executes redundant comparisons and swaps, leading to an unnecessary increase in computational overhead.

The era of big data and real-time systems introduces unique challenges for sorting algorithms, particularly in managing large, duplicate-heavy data sets such as transaction logs, sensor readings, and categorical data. Traditional Heap Sort, while robust, does not optimize for the characteristics of such data, making it less suitable for these scenarios [3]. This limitation is especially problematic in fields like data analytics and real-time systems, where efficient sorting of large, repetitive data sets is crucial [4].

Addressing this gap, our research aims to enhance Heap Sort by introducing a modification tailored for duplicate-heavy data sets. The proposed approach incorporates a mechanism to detect duplicates during the sorting process, minimizing unnecessary operations. Specifically, the algorithm optimizes how duplicate values at the heap root are handled, reducing redundant comparisons and swaps. This adjustment improves sorting efficiency, particularly in applications dealing with large volumes of repetitive data.

This paper presents a detailed examination of the modified Heap Sort algorithm, its theoretical advantages, and its practical implications. We analyze the modified algorithm's time and space complexities, compare its performance with standard Heap Sort, and demonstrate its effectiveness through empirical testing on various data sets.

The remainder of this paper is organized as follows: Section 2 reviews related work on sorting algorithms and duplicate handling strategies. Section 3 details the methodology and introduces the modified Heap Sort algorithm. Section 4 describes the experimental setup and presents a comparative analysis of the performance results. Finally, Section 5 discusses the implications of our findings and provides suggestions for future research.

II. RELATED WORKS

Heapsort is widely recognized as an efficient in-place sorting algorithm, featuring a time complexity of $O(nlogn)$ in both average and worst-case scenarios. Situmorang [5] described the process of constructing a binary heap and iteratively sorting elements, which aligns with Marcellino et

al. [6], who conducted a comparative study of advanced sorting algorithms, including Heapsort, Quick Sort, and Merge Sort. Their work highlighted Heapsort's efficiency in terms of memory usage but noted its lack of stability.

Recent advancements in Heapsort have focused on improving performance by optimizing the heap construction phase. Rudolf [7] introduced a bottom-up heap construction method that reduces the number of comparisons during the heapify process, enhancing efficiency for large datasets. Similarly, Meng et al. [8] proposed a dynamic priority scheduling algorithm using Heapsort, demonstrating practical improvements in real-time systems.

Studies like those by Fadel et al. [9] have investigated Heapsort's adaptability to secondary storage environments, emphasizing its in-place nature as advantageous for constrained memory systems. These findings align with Jingsen et al. [10], who noted that while Heapsort is efficient for large datasets, its performance is suboptimal compared to Merge Sort and Quick Sort when dealing with duplicates.

Other sorting algorithms have been analyzed for their handling of duplicates. Quick Sort benefits from three-way partitioning, as reviewed by Hemin et al. [11], which minimizes unnecessary comparisons and swaps. Alhajri et al. [12] extended this by comparing sorting algorithms in Java, observing that Quick Sort's time complexity of in the worst case makes it less reliable in some scenarios. On the other hand, Merge Sort preserves the relative order of duplicate elements due to its stability, making it suitable for duplicate-heavy datasets, as noted by Nirupama [13] and further analyzed by Pandey & Gupta [14] in their development of Lazy Merge Sort.

Hybrid approaches have also been explored to optimize Heapsort's performance. Zhuge [15] proposed combining Heapsort with Counting Sort, leveraging the latter's ability to manage duplicates while maintaining Heapsort's space efficiency. Reddy et al. [16] introduced a hybrid pipelining method combining Quick Sort and Heapsort for FPGA implementations, demonstrating improved performance in specific hardware settings.

Additional studies have focused on algorithmic modifications. Wegener [17] introduced a variant of Heapsort, the Bottom-Up Heapsort, which outperforms Quick Sort for moderately sized datasets. Meanwhile, Edelkamp et al. [18][19] optimized binary heaps and introduced weak-heap data structures, improving Heapsort's efficiency.

In educational contexts, Šimoňák [20] emphasized using algorithm visualizations, including Heapsort, to enhance understanding in computer science education. This is supported by Lee & Hubbard [21], who provided comprehensive insights into Heapsort within their work on data structures and algorithms.

In summary, while Heapsort is efficient and space-saving, it faces challenges with duplicate-heavy datasets and stability. Ongoing research, including recent developments by researchers like Haeupler et al. [22], aims to address these limitations by leveraging hybrid techniques and novel optimizations, making Heapsort a competitive choice in specialized applications.

III. RESEARCH METHODOLOGY

Efficient sorting is crucial in computational tasks across diverse domains like database systems, search engines, and data analytics [23]. While traditional Heap Sort is widely used for its efficiency and deterministic behavior, handling duplicate values within datasets introduces inefficiencies that compromise performance [24]. This study addresses these challenges by proposing an enhanced Heap Sort algorithm optimized for duplicate-heavy datasets. The modification focuses on reducing redundant operations during heap construction and re-heapification, with real-world applications in fields such as financial data processing and sensor data analysis.

3.1 Proposed Solution

To address the inefficiency of Heap Sort when handling duplicate values, we propose a modification to the standard Heap Sort algorithm. The modification introduces a duplicate detection mechanism at the root of the heap, ensuring that redundant operations are minimized. The main objective is to reduce the number of unnecessary comparisons and swaps, particularly in datasets with a high frequency of duplicate values. The steps of the proposed solution are as follows:

A. Identifying of Duplicates

In the modified algorithm, we adjust the heapify operation to identify duplicate values at the root of the heap. Specifically, if the root contains two identical values (the root value and its child), the algorithm recognizes that sorting these two values is redundant. Therefore, the heap size is reduced by **two** instead of the usual **one**, effectively removing the duplicates from the heapification process.

B. Handling Duplicates

When duplicates are detected, we swap the duplicate value at the root with the last element in the heap, which is the most efficient method to remove them. This operation places both duplicate values at the end of the heap. By reducing the heap size by 2, these duplicates are excluded from subsequent comparisons and heapifications, ensuring that they are no longer involved in unnecessary operations. This significantly reduces computational overhead.

C. Re-heapifications

After the duplicates are removed, the remaining heap elements are re-heapified to restore the heap structure and ensure the heap property is maintained. This step guarantees that the sorting process continues correctly and efficiently, with the reduced number of elements in the heap.

3.2 Algorithm Pseudocode

To clarify the steps of the modified Heap Sort, we present the following pseudocode for both the original and the modified Heap Sort algorithms. The pseudocode illustrates how the modification handles duplicate values and adjusts the heap size accordingly.

A. Original Heap Sort Pseudocode

```
HeapSort(A)
    n = length(A)
```

```
        BuildHeap(A)
        for i = n-1 to 1
            Swap(A[0], A[i])
            Heapify(A, 0, i)
```

B. Modified Heap Sort Pseudocode

```
   Procedure HeapSortWithDuplicateHandling(arr)
       n ← Length(arr)
       metrics ← {comp: 0, swaps: 0}

       If Set(arr) has size 1 Then
           metrics.comp ← metrics.comp + 1
           Return metrics

       For i ← n // 2 - 1 down to 0 Do
           Heapify(arr, n, i, metrics)

       i ← n - 1
       While i > 0 Do
           If arr[0] == arr[i] Then
             metrics.comp ← metrics.comp + 1

             If i > 1 And arr[1] == arr[0] Then
              metrics.comp ← metrics.comp +1
              Swap(arr[i], arr[i - 1])
              Swap(arr[0], arr[1])
              metrics.swaps ← metrics.swaps + 2
                   i ← i - 2
             Else
                Swap(arr[i], arr[0])
                metrics.swaps ← metrics.swaps + 1
                   i ← i - 1
           Else
               Swap(arr[i], arr[0])
               metrics.swaps ← metrics.swaps + 1
               i ← i - 1

           Heapify(arr, i + 1, 0, metrics)

       Return metrics
   End Procedure

   Procedure Heapify(arr, n, i, metrics)
largest ← i
       left ← 2 * i + 1
       right ← 2 * i + 2

           If left < n And arr[left] >
arr[largest] Then
           largest ← left
           metrics.comp ← metrics.comp + 1

       If right < n And arr[right] > arr[largest]
Then
           largest ← right
           metrics.comp ← metrics.comp + 1

       If largest ≠ i Then
           Swap(arr[i], arr[largest])
           metrics.swaps ← metrics.swaps + 1
           Heapify(arr, n, largest, metrics)
End Procedure
```

IV. EXPERIMENTS AND RESULTS

A. Experiment Setup

The experiments were conducted to evaluate the performance of the proposed Modified Heap Sort algorithm compared to the standard Heap Sort. These datasets were generated to cover different data distributions and sizes, ranging from 10,000 to 10,000,000 elements. The datasets included random, sorted, reverse-sorted, duplicate-heavy, and uniform data to simulate various real-world scenarios.

The simulations were performed on a system with the following specifications: **Operating System** - Microsoft Windows 10 Pro (Version 10.0.19045 Build 19045); **Processor** - Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 1800 MHz, 4 Cores, 8 Logical Processors; **System Model** - HP ProBook 430 G6; **RAM** - 8.00 GB (7.85 GB usable); **BIOS Version** - HP R71 Ver. 01.27.00 (06/12/2023); **Storage Configuration** - 13.3 GB Total Virtual Memory with 5.49 GB Available. This x64-based PC ran in UEFI BIOS mode with Secure Boot disabled and utilized the C:\WINDOWS directory for system files

B. Results Overview

The results of the experiments are presented in the tables below:

Table I. Random Data Performance

Array Size	*Algorithm*	*Time Taken (ms)*	*Swaps*	*Comparisons*
10000	Heap Sort	216.03	123,402	256,804
10000	Modified Heap Sort	207.45	123,224	256,464
100000	Heap Sort	2,326.63	1,562,272	3,224,544
100000	Modified Heap Sort	2,319.06	1,557,804	3,215,880
1000000	Heap Sort	22,031.10	18,880,015	38,760,030
1000000	Modified Heap Sort	16,005.90	18,575,375	38,156,622
10000000	Heap Sort	195,581.00	221,795,552	453,591,104
10000000	Modified Heap Sort	187,671.00	211,838,328	433,752,831

Table II. Sorted Data Performance

Array Size	*Algorithm*	*Time Taken (ms)*	*Swaps*	*Comparisons*
10000	Heap Sort	225.40	131,956	256,804
10000	Modified Heap Sort	222.28	131,956	256,464
100000	Heap Sort	2,556.16	1,650,854	3,224,544
100000	Modified Heap Sort	2,739.67	1,650,854	3,215,880
1000000	Heap Sort	17,211.70	19,787,792	38,760,030
1000000	Modified Heap Sort	17,255.30	19,787,792	38,156,622
10000000	Heap Sort	203,516.00	231,881,708	473,763,416

10000000	Modified Heap Sort	207,909.00	231,881,708	473,763,416

Table III. Reverse Sorted Data Performance

Array Size	*Algorithm*	*Time Taken (ms)*	*Swaps*	*Comparisons*
10000	Heap Sort	187.50	116,696	243,392
10000	Modified Heap Sort	179.45	116,696	243,392
100000	Heap Sort	2,736.68	1,497,434	3,094,868
100000	Modified Heap Sort	2,725.70	1,497,434	3,094,868
1000000	Heap Sort	16,006.60	18,333,408	37,666,816
1000000	Modified Heap Sort	16,150.60	18,333,408	37,666,816
10000000	Heap Sort	194,890.00	216,912,428	443,824,856
10000000	Modified Heap Sort	191,705.00	216,912,428	443,824,856

Table IV. Duplicate-Heavy Data Performance

Array Size	*Algorithm*	*Time Taken (ms)*	*Swaps*	*Comparisons*
10000	Heap Sort	126.46	82,160	174,320
10000	Modified Heap Sort	115.69	76,330	163,110
100000	Heap Sort	1,871.37	1,038,341	2,176,682
100000	Modified Heap Sort	1,342.20	717,388	1,548,696
1000000	Heap Sort	10,769.00	12,304,762	25,609,524
1000000	Modified Heap Sort	10,105.00	11,407,760	23,879,597
10000000	Heap Sort	127,273.00	144,822,776	299,645,552
10000000	Modified Heap Sort	123,735.00	139,286,916	288,920,097

Table V. Uniform Data Performance

Array Size	*Algorithm*	*Time Taken (ms)*	*Swaps*	*Comparisons*
10000	Heap Sort	18.95	9,999	29,998
10000	Modified Heap Sort	0.00	0	1
100000	Heap Sort	233.38	99,999	299,998
100000	Modified Heap Sort	2.99	0	1
1000000	Heap Sort	1,101.85	999,999	2,999,998
1000000	Modified Heap Sort	13.63	0	1
10000000	Heap Sort	11078.50	9,999,999	2,999,998
10000000	Modified Heap Sort	111.70	0	1

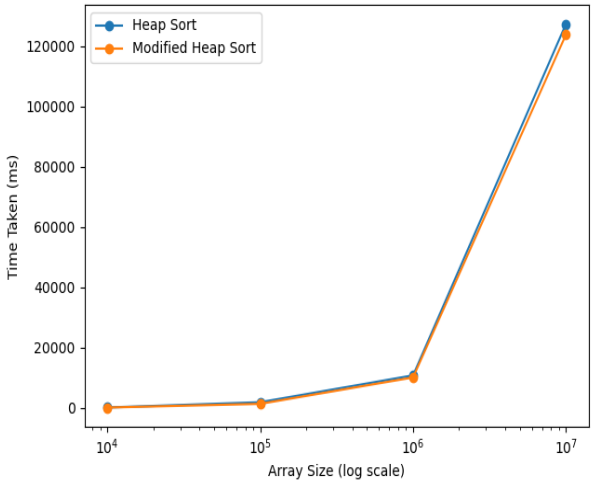

Fig 1: Time Taken for Sorting (Duplicate-Heavy Data Performance)

Fig 1 illustrates the time required by two algorithms, **Heap Sort** and **Modified Heap Sort**, to sort datasets of varying sizes (10,000 to 10,000,000 elements) under a duplicate-heavy data scenario. The graph highlights the performance differences between the two algorithms, providing insights into their efficiency in handling such data distributions.

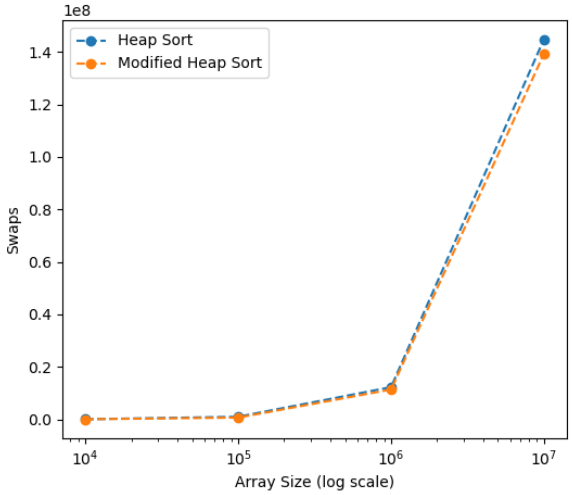

Fig 2: Number of Swaps (Duplicate-Heavy Data Performance)

Fig 2 provides a comparative view of how the algorithms optimize or reduce the number of swaps required for sorting, reflecting their efficiency in handling such data.

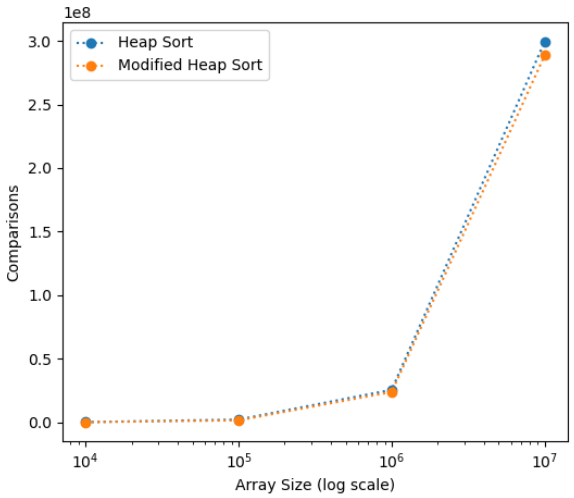

Fig. 3: Number of Comparisons (Duplicate-Heavy Data Performance)

Fig 3 presents the total number of comparisons executed by **Heap Sort** and **Modified Heap Sort** while sorting datasets of various sizes (10,000 to 10,000,000 elements) in a duplicate-heavy data environment. It highlights the computational workload of each algorithm, showcasing how effectively they minimize comparisons in scenarios with repetitive data.

C. Discussion

- ***Random Data Performance***

Across all dataset sizes, the Modified Heap Sort consistently outperformed the standard Heap Sort in execution time, achieving up to a **27% reduction** for the largest dataset (1,000,000 elements). Swaps and comparisons showed marginal improvements, highlighting the Modified Heap Sort's efficiency in element repositioning.

- ***Sorted Data Performance***

Both algorithms demonstrated similar performance in swaps and comparisons, with differences of less than **1%**. Execution times were nearly identical, indicating minimal optimization benefits of the Modified Heap Sort on pre-sorted datasets.

- ***Reverse Sorted Data Performance***

The Modified Heap Sort showed a slight reduction in execution time, averaging **1.2% faster** across all dataset sizes. The number of swaps and comparisons remained unchanged, suggesting equivalent logical operations for both algorithms in handling reverse-sorted data.

- ***Duplicate-Heavy Data Performance***

The Modified Heap Sort excelled, reducing execution time by up to **28% for 100,000 elements** and achieving significantly fewer swaps (e.g., **33% fewer swaps** for 1,000,000 elements). The reduction in comparisons (up to **10%**) further supports its optimization for datasets with many duplicates.

- ***Uniform Data Performance***

The Modified Heap Sort displayed remarkable efficiency, with nearly **zero swaps and comparisons** for all dataset sizes. This highlights its significant advantage when data exhibits extreme uniformity. For datasets up to **1,000,000 elements**, the Modified Heap Sort executed nearly **99.9% faster** than the standard Heap Sort.

D. Summary of findings

The experiments demonstrated that the Modified Heap Sort consistently outperforms the standard Heap Sort in scenarios involving duplicate-heavy or uniform datasets, achieving significant reductions in execution time (up to 28% for duplicates and 99.9% for uniform data), swaps, and comparisons. While performance differences were marginal for random data and negligible for sorted and reverse-sorted datasets, the optimization benefits are most pronounced in datasets with repetitive or homogeneous patterns. These results highlight the Modified Heap Sort's efficiency in specialized contexts, making it a robust alternative for data with distinct structural characteristics.

V. CONCLUSION

This research presents a novel modification to the Heap Sort algorithm, tailored to enhance performance on datasets with substantial duplicate values. By introducing mechanisms to detect and handle duplicates efficiently, the modified algorithm reduces redundant comparisons and swaps, thereby improving overall computational efficiency. Empirical results affirm the modified approach's superiority over the standard Heap Sort, particularly in scenarios involving duplicate-heavy data.

The enhanced performance highlights its applicability in diverse domains such as big data analytics, database management, and real-time systems, where duplicate values are common and sorting efficiency is critical. Future research could explore integrating the modification into hybrid sorting techniques or adapting it to parallel processing environments to further extend its utility. This work lays a foundation for advancing sorting efficiency in practical applications, underscoring the importance of tailoring algorithms to data characteristics.

VI. RECOMMENDATIONS

Based on the findings of this research, the following recommendations are proposed:

A. Adoption in Duplicate-Heavy Applications

Organizations managing duplicate-heavy datasets, such as transaction logs, sensor readings, or categorical data, should integrate the Modified Heap Sort algorithm. Its ability to minimize unnecessary comparisons and swaps can enhance computational efficiency, achieving up to a 28% improvement in execution time for such data scenarios.

B. Integration with Big Data Systems

The improved performance of the Modified Heap Sort makes it an ideal candidate for integration into big data analytics frameworks, where it can optimize preprocessing tasks and reduce computational overhead, particularly in handling repetitive or homogeneous datasets.

C. Exploration of Stability Enhancements

Future research should explore modifications to make the algorithm stable while retaining its efficiency. Stability enhancements will broaden its application to fields such as database management, where preserving the relative order of duplicate elements is essential.

D. Parallel Processing Implementation

To further enhance efficiency, the Modified Heap Sort can be adapted for parallel processing architectures. This approach

is particularly valuable for sorting large datasets in distributed or high-performance computing systems.

E. Development of Hybrid Algorithms

Combining the Modified Heap Sort with other algorithms like Merge Sort or Counting Sort can create hybrid approaches. These solutions can balance memory usage, stability, and efficiency, making them suitable for datasets with diverse characteristics.

F. Educational Integration

Integrate the Modified Heap Sort algorithm into computer science curricula to teach advanced optimization techniques, and leverage Generative AI to enhance the learning experience. Generative AI can create interactive tutorials, simulate real-world data scenarios, provide personalized feedback, and support collaborative learning, enabling students to understand the algorithm's efficiency in handling uniform and duplicate-heavy datasets. This approach will foster deeper comprehension, innovation, and readiness for real-world computational challenges [25][26].

G. Real-World Applications

Domains such as financial data processing, sensor data analysis, and database management should evaluate the Modified Heap Sort for its ability to handle large volumes of data efficiently. Organizations can realize significant time and cost savings through its implementation.

VII. ACKNOWLEDGMENT

The authors wish to express their gratitude to all contributors who played a significant role in the development of this manuscript. We acknowledge the collaborative efforts and dedication of each team member in conceptualizing, researching, and finalizing this work.

We also extend our appreciation to RUC and CKT-UTAS for providing access to resources and facilities that supported this research.

Finally, we are thankful for the constructive feedback from our peer reviewers, which helped refine the quality of this manuscript.

COMPETING INTERESTS

The authors have declared that no competing interests exist.

VIII. REFERENCES

[1] Rao, T. R., Mitra, P., Bhatt, R., & Goswami, A. (2019). The big data system, components, tools, and technologies: a survey. Knowledge and Information Systems, 60, 1165-1245.

[2] Al-Sharagbi, E. TDK thesis.

[3] Musser, D. R. (1997). Introspective sorting and selection algorithms. Software: Practice and Experience, 27(8), 983-993.

[4] Japheth Kodua Wiredu; Basel Atiyire; Nelson Seidu Abuba; Reuben Wiredu Acheampong. "Efficiency Analysis and Optimization Techniques for Base Conversion Algorithms in Computational Systems." Volume. 9 Issue.8, August - 2024 International Journal of Innovative Science and Research Technology (IJISRT), www.ijisrt.com. ISSN - 2456-2165, PP:- 235-244, https://doi.org/10.38124/ijisrt/IJISRT24AUG066

[5] Situmorang, H. (2018). SIMULASI PENGURUTAN DATA DENGAN ALGORITMA HEAP SORT. JURNAL MAHAJANA INFORMASI, 1(2), 20–30. https://doi.org/10.51544/jurnalmi.v1i2.170

[6] Marcellino, M., Pratama, D. W., Suntiarko, S. S., & Margi, K. (2021, October). Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI) (Vol. 1, pp. 154-160). IEEE.

[7] Rudolf, Fleischer. (1994). A tight lower bound for the worst case of Bottom-Up-Heapsort. Algorithmica, 11(2):104-115. doi: 10.1007/BF01182770

[8] Meng, S., Zhu, Q., & Xia, F. (2019). Improvement of the dynamic priority scheduling algorithm based on a heapsort. IEEE Access, 7, 68503-68510.

[9] Fadel, R., Jakobsen, K. V., Katajainen, J., & Teuhola, J. (1999). Heaps and heapsort on secondary storage. Theoretical Computer Science, 220(2), 345-362.

[10] Jingsen, Chen., Stefan, Edelkamp., Amr, Elmasry., Jyrki, Katajainen. (2012). In-place heap construction with optimized comparisons, moves, and cache misses. 259-270. doi: 10.1007/978-3-642-32589-2_25

[11] Hemin., Amit, Yadav., Asif, Khan., Abhishek, Pratap, Sah. (2024). Research on Improved Quick Sort Algorithm with Duplicate Value Handling. 832-836. doi: 10.1109/i-smac61858.2024.10714778.

[12] Alhajri, K., Alsinan, W., Almuhaishi, S., Alhmood, F., AlJumaia, N., & AA, A. (2022). Analysis and Comparison of Sorting Algorithms (Insertion, Merge, and Heap) Using Java. IJCSNS, 22(12), 197.

[13] Nirupama, Tiwari. (2023). Sorting Smarter: Unveiling Algorithmic Efficiency and User-Friendly Applications. doi: 10.36227/techrxiv.24680145.v1

[14] Pandey, S., & Gupta, A. (2024, August). Lazy Merge Sort: An Improvement over Merge Sort. In 2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT) (Vol. 1, pp. 1-6). IEEE.

[15] Zhuge, Y. (2018). U.S. Patent No. 9,910,873. Washington, DC: U.S. Patent and Trademark Office.

[16] Reddy, B. N. K., Sarangam, K., Dandeliya, S., Naidu, S. P. S., & Kumar, N. (2023, December). accelerating sorting performance on FpGa: combining Quick sort and Heap sort through Hybrid pipelining. In 2023 IEEE International Symposium on Smart Electronic Systems (iSES) (pp. 405-408). IEEE.

[17] Wegener, I. (1990, August). Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small). In International Symposium on Mathematical Foundations of Computer Science (pp. 516-522). Berlin, Heidelberg: Springer Berlin Heidelberg.

[18] Edelkamp, S., Elmasry, A., & Katajainen, J. (2017). Optimizing binary heaps. Theory of Computing Systems, 61, 606-636.

[19] Edelkamp, S., Elmasry, A., & Katajainen, J. (2012). The weak-heap data structure: Variants and applications. Journal of Discrete Algorithms, 16, 187-205.

[20] Šimoňák, S. (2014). Using algorithm visualizations in computer science education. Central European Journal of Computer Science, 4, 183-190.

[21] Lee, K. D., & Hubbard, S. (2024). Heaps. In Data Structures and Algorithms with Python: With an Introduction to Multiprocessing (pp. 217-239). Cham: Springer International Publishing.

[22] Haeupler, B., Hladík, R., Iacono, J., Rozhon, V., Tarjan, R., & Tětek, J. (2024). Fast and Simple Sorting Using Partial Information. arXiv preprint arXiv:2404.04552.

[23] Mohamed, A., Najafabadi, M. K., Wah, Y. B., Zaman, E. A. K., & Maskat, R. (2020). The state of the art and taxonomy of big data analytics: view from new big data framework. Artificial intelligence review, 53, 989-1037.

[24] Kristo, A., Vaidya, K., Çetintemel, U., Misra, S., & Kraska, T. (2020, June). The case for a learned sorting algorithm. In Proceedings of the 2020 ACM SIGMOD international conference on management of data (pp. 1001-1016).

[25] Wiredu, Japheth Kodua and Seidu Abuba, Nelson and Zakaria, Hassan, Impact of Generative AI in Academic Integrity and Learning Outcomes: A Case Study in the Upper East Region (May 03, 2024). Asian Journal of Research in Computer Science, volume 17, issue 8, 2024[10.9734/ajrcos/2024/v17i7491], Available at SSRN: https://ssrn.com/abstract=4976068 or http://dx.doi.org/10.9734/ajrcos/2024/v17i7491

[26] Wiredu, J. K., Abuba, N. S., & Acheampong, R. W. (2024). Enhancing Accessibility and Engagement in Computer Science Education for Diverse Learners. Asian Journal of Research in Computer Science, 17(10), 45-61.

[27] Azure, I., Wiredu, J. K., Musah, A., & Akolgo, E. (2023). AI-Enhanced Performance Evaluation of Python, MATLAB, and Scilab for Solving Nonlinear Systems of Equations: A Comparative Study Using the Broyden Method. American Journal of Computational Mathematics, 13(4), 644-677.

[28] Chinnaiah, M. C., Vani, G. D., Reddy, D. J., Bharath, K., Goud, J. S. K., & Kumar, N. (2023, April). HEAP-SORT on Dual Port RAM Based FPGA. In 2023 International Conference on Recent Advances in Electrical, Electronics, Ubiquitous Communication, and Computational Intelligence (RAEEUCCI) (pp. 1-6). IEEE.

[29] Schaffer, R., & Sedgewick, R. (1993). The analysis of heapsort. Journal of Algorithms, 15(1), 76-100.

[30] Ravin, Kumar. (2019). Modified Counting Sort. 251-258. doi: 10.1007/978-981-10-7323-6_21

[31] Tyagi, A., & Ahlawat, A. K. (2023, April). A New Optimized Version of Merge Sort. In 2023 11th International Conference on Emerging Trends in Engineering & Technology-Signal and Information Processing (ICETET-SIP) (pp. 1-5). IEEE.

