electronics

MDPI

Article

TwinArray Sort: An Ultrarapid Conditional Non-Comparison Integer Sorting Algorithm

Amin Amini

School of Computer Science and Mathematics, Liverpool John Moores University, James Parsons Building, Byrom Street, Liverpool L3 3AF, UK; a.amini@ljmu.ac.uk; Tel.: +44-(0)-151-231-5464

Abstract

TwinArray Sort is a non-comparison integer sorting algorithm designed for non-negative integers with relatively dense key ranges, offering competitive runtime performance and reduced memory usage relative to other counting-based methods. The algorithm introduces a conditional distinct-array verification mechanism that adapts the reconstruction strategy based on data characteristics while maintaining worst-case time and space complexity of $O(n + k)$. Comprehensive experimental evaluations were conducted on datasets containing up to 10^8 elements across multiple data distributions, including random, reverse-sorted, nearly sorted, and their unique variants. The results demonstrate consistent performance improvements compared with established algorithms such as Counting Sort, Pigeonhole Sort, MSD Radix Sort, Spreadsort, Flash Sort, Bucket Sort, and Quicksort. TwinArray Sort achieved execution times up to 2.7 times faster and reduced memory usage by up to 50%, with particularly strong performance observed for unique and reverse-sorted datasets. The algorithm exhibits good scalability for large datasets and key ranges, with performance degradation occurring primarily in extreme cases where the key range significantly exceeds the input size due to auxiliary array requirements. These findings indicate that TwinArray Sort is a competitive solution for in-memory sorting in high-performance and distributed computing environments. Future work will focus on optimizing performance for wide key ranges and developing parallel implementations for multi-core and GPU architectures.

Keywords: non-comparison integer sorting; linear-time sorting; dense key ranges; duplicate-aware algorithms; memory–time trade-offs

1. Introduction and Related Work

Sorting algorithms are essential to computer science and are used in many different applications. Through rigorous optimization and study, classic comparison-based sorting algorithms such as Quicksort, Merge Sort, and Heapsort have achieved average-case time complexities of $O(n\log n)$ [1–4]. Nonetheless, these algorithms have inherent limitations, particularly when dealing with large datasets or under technological constraints where time and space efficiency is crucial.

Non-comparison-based sorting algorithms, including Bucket Sort, Radix Sort, and Counting Sort, offer an alternative approach by leveraging data attributes instead of direct comparisons [5,6]. These algorithms are suitable for specific types of data and can achieve linear time complexity under certain conditions. For example, Counting Sort, which counts the frequency of each element, can run in $O(n + k)$ time, where n is the number of elements and k is the range of input values [7].

Academic Editor: Ping-Feng Pai

Received: 28 December 2025
Revised: 24 January 2026
Accepted: 27 January 2026
Published: 30 January 2026

Despite their efficiency, non-comparison-based algorithms also have drawbacks. Counting Sort, for instance, consumes additional memory proportional to the range of input values, which poses challenges for datasets with large value ranges [8,9]. It may also struggle with handling duplicate elements efficiently [10]. Similarly, Bucket Sort and Radix Sort often involve high overhead due to initialization demands and multiple passes over the dataset. For instance, the setup and management of buckets in Bucket Sort introduces substantial overhead [11,12]. Moreover, the performance of Bucket Sort depends heavily on the number of buckets chosen: too few leads to under-utilization and inefficiency, while too many significantly increase space complexity [13].

To address these issues and enhance performance, the proposed TwinArray Sort introduces techniques such as dual auxiliary arrays and a conditional distinct array verifier. Its design avoids tunable heuristic parameters but explicitly relies on the maximum key value k, making it most suitable for datasets where the key range is relatively dense. TwinArray Sort is especially advantageous for datasets with particular distribution patterns such as nearly sorted, reversed, or randomized sequences. It offers consistent performance across various data types, making it a versatile solution for modern sorting requirements. Applications include machine learning, where efficient data pre-processing can significantly reduce model training time, and large-scale data processing pipelines, where fast sorting is critical. Unlike traditional sorting algorithms that may degrade under certain conditions, TwinArray Sort maintains stable performance in most scenarios. TwinArray Sort applies exclusively to arrays of non-negative integers and is not intended for floating-point or general object sorting.

2. Materials and Methods

To define the size for two auxiliary arrays, TwinArray Sort first identifies the maximum value within the input array. The values from the input array and their associated frequencies are stored in these arrays. Once the arrays are populated according to their indices, the algorithm checks for duplicate elements. It generates the sorted output by either directly extracting nonzero components from the value array or by reconstructing elements based on their frequencies, depending on whether duplicates are detected. While both reconstruction paths require a linear scan over the auxiliary domain, the no-duplicate case avoids frequency-based replication and repeated writes, reducing constant-factor overhead without altering asymptotic complexity. The procedure may prepend a zero if the input array's index 0 holds the value 0, ensuring that the output array remains the same size as the input array. This technique, which is particularly effective with datasets that have a narrow range of integer values, successfully combines the concepts of Counting Sort with direct element insertion. Algorithm 1 shows the pseudocode of the TwinArray Sort algorithm.

By utilizing the built-in indices of array elements, the TwinArray Sort method is designed to sort an array of integers efficiently. This approach ensures that it can handle both unique and repeated numbers effectively by sorting the data using dual auxiliary arrays. The TwinArray Sort algorithm is implemented and analyzed in the steps that follow.

Formally, let the maximum value in the input be $k = \max(A)$. TwinArray Sort allocates two auxiliary arrays of length $k + 1$:

$$\begin{aligned} value\_store[i] &= i \cdot 1_{\{\exists\, j\,:\, a_j = i\}}, \\ count\_store[i] &= \textstyle\sum_{j=1}^{n} 1_{a_j = i}, i = 0, \ldots, k. \end{aligned} \tag{1}$$

Here, $1\{\cdot\}$ denotes the indicator function, which equals 1 if the condition is true and 0 otherwise. The first array mirrors the domain by storing each encountered value at its corresponding index, while the second records the frequency of each value. This

formalization captures the mapping phase of the algorithm and provides the mathematical basis for its duplicate-handling mechanism.

Algorithm 1 Pseudocode of the TwinArray Sort algorithm

Require: Array *arr*
Ensure: Sorted array *sorted_arr*
max val ← maximum value in *arr*
Create array *value_store* of size *max_val* + 1 initialized to 0
Create array *count_store* of size *max_val* + 1 initialized to 0
has duplicates ← false
for each *num* **in** *arr* **do**
    value_store[*num*] ← *num*
    count_store[*num*] ← *count_store*[*num*] + 1
    if *count_store*[*num*] > 1 **then**
    has_duplicates ← true
    end if
end for
Create empty array *sorted_arr*
if not *has_duplicates* **then**
    for each *value* **in** *value_store* **do**
    if *value* ≠ 0 **then**
    Append *value* to *sorted_arr*
    end if
    end for
    if *count_store*[0] == 1 **then**
    Insert 0 at the beginning of *sorted_arr*
    end if
else
    for *i* ← 0 **to** length of *count_store* − 1 **do**
    if *count_store*[*i*] > 0 **then**
Append *count_store*[*i*] copies of *value_store*[*i*] to *sorted_arr*
    end if
    end for
end if
return *sorted_arr*

The presence of duplicates is detected using the following conditional trigger:

$$has\_duplicates = (\exists i \in \{0, \ldots, k\} : count\_store[i] > 1) \tag{2}$$

Based on this condition, the sorted output array is constructed as follows. If no duplicates are present:

$$sorted_{arr} = [\, i \mid i \in \{0, \ldots, k\} \wedge count\_store[i] > 0 \,] \tag{3}$$

If duplicates are present:

$$sorted_{arr} = [i \text{ repeated } count\_store[i] \text{ times} \mid i \in \{0, \ldots, k\}] \tag{4}$$

Let A be an input array with n elements, $A = \{a_1, a_2, a_3, \ldots, a_n\}$, where $\forall a_i, a_j \in A, a_i \geq 0$. The TwinArray Sort algorithm involves the following steps (Figure 1):

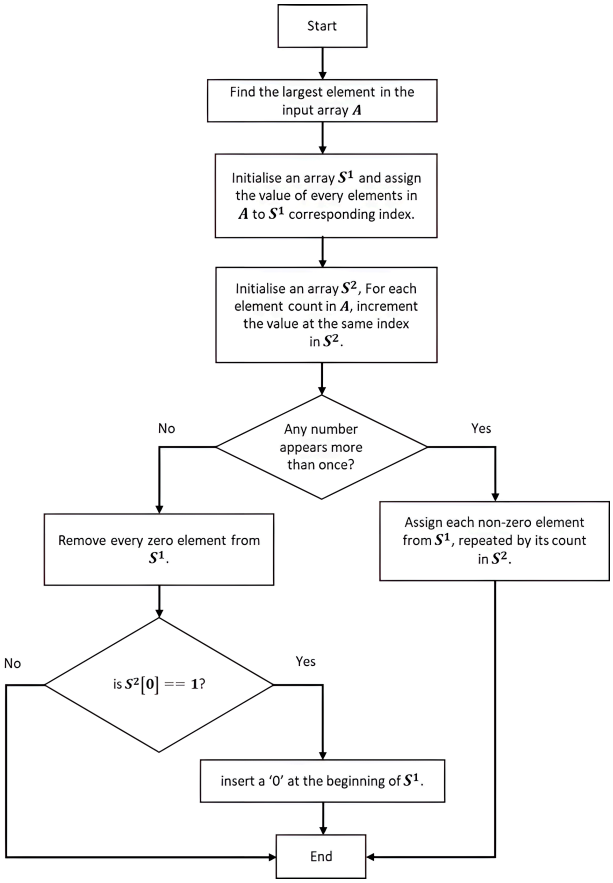

Figure 1. Flowchart of the TwinArray Sort algorithm.

The TwinArray Sort algorithm has a time complexity of O($n + k$), where n is the number of elements in the input array.

This is achieved by performing a single pass over the input array and another pass over the auxiliary arrays during the mapping and reconstruction phases, respectively. The space complexity is also O($n + k$), where k is the highest value in the input array, primarily due to the use of two auxiliary arrays.

Although TwinArray Sort shares some similarities with Counting Sort, the two differ significantly in their methodology, advantages, limitations, and unique features. TwinArray Sort offers several advantages over Counting Sort, particularly in terms of memory optimization and efficient handling of duplicate values. Its conditional distinct array verification mechanism enables the algorithm to determine whether a simpler reconstruction path can be used when no duplicates are present—reducing constant-factor overhead (e.g., fewer frequency-based checks and writes) without changing asymptotic complexity.

Moreover, by separating value storage and frequency counting into two auxiliary arrays, TwinArray Sort allows for greater flexibility in certain scenarios. One of its defining features is the use of a conditional trigger that detects the presence of duplicates and adapts the sorting strategy accordingly. Unlike Counting Sort, which processes all elements uniformly regardless of duplication, TwinArray Sort dynamically modifies its approach based on data characteristics.

3. Experimental Setup

For the purposes of this investigation, the TwinArray Sort algorithm was developed in Python 3.11. The effectiveness of the TwinArray Sort algorithm was assessed by comparing it to many other widely used sorting algorithms. These included Counting Sort, Pigeonhole

Sort, MSD Radix Sort, Flash Sort, Tim Sort, Heap Sort, Shell Sort, Comb Sort, Bucket Sort, Block Sort, Spreadsort, Quicksort, and Merge Sort. Following a preliminary comparative investigation, the seven fastest algorithms were chosen. A more thorough comparative study and analysis were then conducted on these selected algorithms. This method enabled a comprehensive understanding of TwinArray Sort's performance characteristics in relation to other established sorting techniques.

TwinArray Sort is contrasted in this study with Counting Sort, Pigeonhole Sort, MSD Radix Sort, Spreadsort, Flash Sort, Bucket Sort, and Quicksort. Various data distributions based on 64-bit unsigned integers were used, including nearly sorted, reversed, and random arrays with different sizes: 10^5, 10^6, 10^7, and 10^8. The sorting algorithms were evaluated on an 8-core AMD Ryzen 7 5700X CPU with 32 GB of RAM, running Ubuntu Linux on a virtual machine (WSL 2).

Selecting the middle element as the pivot in Quicksort is a calculated decision that balances performance and stability. In the best-case scenario, the middle element tends to split the array into two nearly equal parts, maintaining the optimal O($n$log$n$) time complexity and reducing recursion depth [14]. However, in the worst-case scenario, poor splits can lead to O(n^2) time complexity [15]. Nonetheless, using the middle element often yields better average-case performance compared to choosing the first or last element, which are more susceptible to producing unbalanced partitions [14]. Therefore, the middle element is selected in this study to support overall Quicksort performance and stability.

For Bucket Sort, the number of buckets was set equal to the length of the array. This approach provides a balanced strategy suitable for a wide range of data distributions. It ensures that each bucket receives a manageable number of items, which facilitates efficient intra-bucket sorting and keeps the overall time complexity close to O(n + k), where k is a small constant. This method works particularly well when the input is uniformly distributed, as it promotes evenly sized buckets and minimizes performance bottlenecks [12]. Moreover, scaling the number of buckets proportionally to the array size allows the algorithm to adapt to both small and large datasets without complex heuristics. The findings of Burnetas et al. (1997) support this approach, noting that bucket count significantly influences partitioning balance and intra-bucket sorting efficiency [12]. This implementation aligns with those insights, offering a robust and adaptable solution.

Six different random number generators were used to produce test arrays: Random, Reversed, Nsorted (nearly sorted), U Random (unique random), U Reversed (unique reversed), and U Nsorted (unique nearly sorted). In the Nsorted and U Nsorted arrays, 5% of the elements were displaced to simulate near ordering. To ensure consistency and fairness, the same generated arrays were used to benchmark all sorting algorithms under identical conditions.

To ensure consistency and reproducibility, we employed the memory profiler module in Python for tracking memory consumption during each sorting run. This profiler periodically samples the Python process's memory footprint (resident set size) at specified intervals. Specifically, we instrumented each sorting function call with a decorated context manager that monitors peak memory usage until the sorting routine completes. By capturing the highest recorded value, we obtain a reliable indication of peak resident memory consumption for each algorithm under test. All experiments were run in the same Python environment and on the same system configuration to minimize variability.

All evaluated algorithms were implemented in pure Python without reliance on NumPy vectorization, C extensions, or external optimized libraries. Each algorithm operates directly on Python lists and returns a Python list as output. Input generation, output construction, and any required data handling are included in the reported execution times for all methods to ensure consistent and fair comparisons.

4. Results

A comparison of several sorting algorithms under varied input distributions and dataset sizes is presented in Table 1. The algorithms evaluated include TwinArray Sort, Counting Sort, Pigeonhole Sort, MSD Radix Sort, Spreadsort, Flashsort, Bucket Sort, and Quicksort. Arrays with distributions including Random, Reversed, nearly Sorted (Nsorted), Unique Random (U Random), Unique Reversed (U Reversed), and Unique Sorted (U Nsorted) were used to assess each technique. The dataset sizes range from 105 to 108. The table reports both runtime (in seconds) and memory usage (in megabytes).

As can be seen, TwinArray Sort performed significantly faster across various input distributions. For instance, TwinArray Sort completed the task in 178.6 s and required 2291.14 MB of memory for a dataset of size 108 with a Random distribution. In comparison to other algorithms, such as Counting Sort, which required significantly higher memory (4577.63 MB) for the same input size and distribution and ran in a much higher amount of time (487.10 s), TwinArray Sort's performance is considerably more efficient. This is approximately 50% of the memory requirement and is nearly 2.7× faster.

Similarly, TwinArray Sort was around 10% faster and consumed 60.6% of the memory that Pigeonhole Sort required, which ran in 196.37 s and used 3814.69 MB of memory under the same conditions. It is evident from analyzing TwinArray Sort's performance in many scenarios that apart from MSD Radix Sort, it consistently used less memory compared to others. For instance, TwinArray Sort used 2332.20 MB for the Reversed distribution and dataset size of 108, while Flashsort and Spreadsort needed 2403.25 MB and 8842.53 MB, respectively. TwinArray Sort consumes significantly less memory in this case—approximately 26.3% of Spreadsort's memory requirements and slightly less than Flashsort, using about 97% of its memory.

TwinArray Sort continued to have an advantage in terms of runtime. The approach works consistently well, often surpassing competing algorithms such as Bucket Sort and Spreadsort, especially in terms of memory efficiency, even with the largest dataset size of 108 across multiple distributions.

In further comparison, TwinArray Sort performed significantly faster for unique element arrays. This behavior is associated with the simplified reconstruction path enabled by the conditional distinct array verifier, which reduces constant-factor overhead but does not change the algorithm's asymptotic complexity. For example, TwinArray Sort used 2322.32 MB of RAM to sort a uniquely distributed random (U_Random) array of size 108 in 66.14 s. On the other hand, Counting Sort required much more memory (4577.63 MB) and took 510.64 s for the same distribution and input size.

When dealing with large inputs, MSD Radix Sort performs competitively in run times (e.g., 724.03 s for 108 size). However, its memory usage is lower (1716.61 MB) than that of TwinArray Sort. While Flashsort, Bucket Sort, and Quicksort exhibit longer execution times and varying memory consumption in our experiments, TwinArray Sort generally provides a favorable overall runtime–memory trade-off among the evaluated methods. For unique-element arrays across the tested distributions and input sizes, TwinArray Sort consistently achieves short runtimes and competitive memory usage relative to the other methods considered.

Table 1. Comparative analysis of sorting algorithms showing runtime and memory usage across different input distributions and dataset sizes, including Quicksort [3], Merge Sort [4], Counting Sort [6], Radix Sort [5], Bucket Sort [12], TimSort [9], Spreadsort [16], and Flashsort [17].

Algorithms	Dist.	Run Times (s)				Memory (MB)			
		$n = 10^5$	$n = 10^6$	$n = 10^7$	$n = 10^8$	$n = 10^5$	$n = 10^6$	$n = 10^7$	$n = 10^8$
TwinArray Sort	Random	0.118	1.111	12.267	178.617	2.361	23.386	230.554	2291.144
	Reversed	0.090	1.083	11.302	177.440	2.323	22.923	230.952	2332.206
	Nsorted	0.108	1.104	11.633	177.939	2.319	23.007	238.275	2347.160
	U_Random	0.010	0.363	4.684	66.141	2.290	23.316	237.556	2322.321
	U_Reversed	0.006	0.069	0.681	15.337	2.290	23.316	237.556	2322.320
	U_Nsorted	0.011	0.106	1.213	19.972	2.290	23.316	237.556	2322.320
Counting Sort	Random	0.255	3.059	35.237	487.108	4.570	45.768	457.757	4577.631
	Reversed	0.237	2.423	25.494	317.697	4.572	45.769	457.757	4577.631
	Nsorted	0.242	2.521	27.876	369.341	4.570	45.769	457.757	4577.630
	U_Random	0.286	3.643	40.947	510.645	4.571	45.769	457.757	4577.631
	U_Reversed	0.216	2.167	22.253	244.145	4.573	45.769	457.757	4577.631
	U_Nsorted	0.242	2.347	25.070	275.164	4.573	45.769	457.757	4577.631
Pigeonhole Sort	Random	0.160	1.631	17.099	196.366	3.808	38.140	381.463	3814.690
	Reversed	0.162	1.834	19.335	230.885	3.807	38.139	381.463	3814.690
	Nsorted	0.160	1.787	19.522	230.637	3.808	38.139	381.463	3814.691
	U_Random	0.167	1.980	20.102	239.526	3.807	38.140	381.463	3814.691
	U_Reversed	0.171	1.550	15.215	158.644	3.807	38.139	381.463	3814.691
	U_Nsorted	0.162	1.591	16.169	165.247	3.807	38.139	381.463	3814.691
MSD Radix Sort	Random	0.315	4.181	47.839	663.023	1.679	16.775	165.645	1650.600
	Reversed	0.240	4.367	49.424	750.084	1.653	16.476	164.225	1656.515
	Nsorted	0.302	4.466	61.134	747.062	1.648	16.636	165.542	1647.953
	U_Random	0.323	4.395	58.749	724.031	1.718	17.167	171.664	1716.618
	U_Reversed	0.319	4.043	53.789	632.245	1.717	17.168	171.664	1716.618
	U_Nsorted	0.312	4.036	54.565	640.853	1.717	17.167	171.664	1716.618
Spreadsort	Random	0.132	1.868	27.347	856.944	8.877	88.990	890.985	8842.531
	Reversed	0.124	1.495	26.053	1052.850	8.838	88.524	891.466	8883.657
	Nsorted	0.188	2.369	27.610	1320.442	8.831	88.611	898.698	8898.529
	U_Random	0.224	3.307	35.031	922.079	9.917	100.034	1009.162	9985.214
	U_Reversed	0.184	1.745	15.993	765.324	9.916	100.030	1009.162	9985.215
	U_Nsorted	0.127	1.294	22.438	929.743	9.916	100.030	1009.166	9985.211
Flashsort	Random	0.685	7.443	76.774	910.031	2.400	24.031	240.324	2403.258
	Reversed	0.642	7.383	70.576	769.748	2.400	24.030	240.324	2403.258
	Nsorted	0.662	7.606	73.095	798.901	2.400	24.031	240.324	2403.258
	U_Random	0.684	8.158	83.678	935.181	2.400	24.031	240.324	2403.258
	U_Reversed	0.646	6.662	67.286	697.841	2.400	24.031	240.324	2403.258
	U_Nsorted	0.653	7.322	71.148	733.189	2.400	24.030	240.324	2403.258
Bucket Sort	Random	0.378	5.248	50.429	1313.397	8.877	88.995	890.985	8842.532
	Reversed	0.347	4.325	48.618	1409.581	8.838	88.529	891.467	8883.663
	Nsorted	0.436	4.926	49.666	1414.722	8.836	88.611	898.703	8898.531
	U_Random	0.487	5.843	61.486	1404.084	9.916	100.035	1009.167	9985.217
	U_Reversed	0.450	4.393	40.355	1103.711	9.916	100.034	1009.167	9985.217
	U_Nsorted	0.365	3.640	47.380	1289.114	9.916	100.034	1009.167	9985.217
Quicksort	Random	0.347	4.534	59.453	967.493	3.695	33.638	279.969	4682.106
	Reversed	0.271	3.742	52.240	942.599	2.435	23.550	241.860	2336.865
	Nsorted	0.277	3.839	56.217	946.405	2.480	24.284	246.428	2426.211
	U_Random	0.488	5.584	65.363	816.290	5.412	44.461	271.089	2598.388
	U_Reversed	0.360	3.320	38.278	472.974	2.436	23.550	241.860	2336.866
	U_Nsorted	0.353	3.862	44.096	504.749	2.482	24.320	246.362	2426.211

Figure 2 displays a grouped bar chart comparing runtime (in seconds) for each sorting algorithm at $n = 10^8$. The six bars per algorithm correspond to U_Reversed, U_Nsorted, U_Random, Reversed, Nsorted, and Random distributions. As indicated by the lowest bar in each group, TwinArray Sort consistently outperforms the other algorithms across all distributions, with the largest margin appearing in the unique-element cases (U_Random, U_Reversed, U_Nsorted). These results align closely with the data presented in Table 1, indicating that TwinArray Sort is the fastest among the evaluated methods in our experiments for both unique and non-unique inputs at large scales.

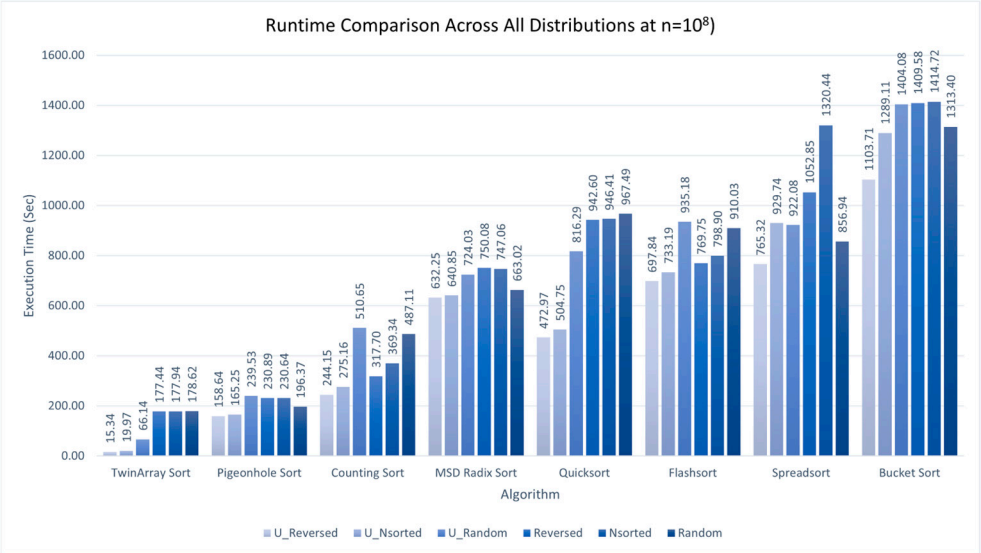

Figure 2. Runtime comparison of sorting algorithms across all distributions at $n = 10^8$.

It was observed that the performance of TwinArray Sort decreases, much like that of Counting Sort, as the range r grows significantly larger than the number of elements n, due to the increasing time and space complexity involved in managing a large count array. When $r \gg n$, it becomes necessary to allocate and process a large array that is mostly empty, leading to memory waste and additional processing time. The results of the investigation (Figure 3) show that there is a virtually perfect positive linear relation (correlation coefficient of approximately 0.992 and 1.0, respectively) between the range and both time and memory.

This suggests that memory use and processing time grow in a directly proportionate way as the range variable increases. In particular, the memory correlation value of 1.0 indicates that memory usage will double along with a doubling of the range. Comparably, the time taken scales roughly linearly with the range, as seen by the strong positive correlation of 0.992 for time. The graph for Time shows that there is some jitteriness in the time data. Although a perfect linear relationship, akin to memory, is what is anticipated, there are a number of reasons why this tiny deviation could occur. There may be small differences in the amount of time it takes to finish the sorting process depending on whether other processes and apps are using the computer's CPU and RAM when the sorting algorithm is running. Furthermore, Python's garbage collection feature for memory management may occasionally halt program execution in order to recover memory, leading to small discrepancies in timing measurements [18]. These variables add to the overall extremely linear and predictable trend, but they also cause the jitteriness in the time measurements that is shown. Thus, TwinArray Sort is less efficient in situations when the number of items is small relative to the wide range of input values because of this

deterioration. It works well in situations when the range of values is proportionate to the number of elements and relatively modest.

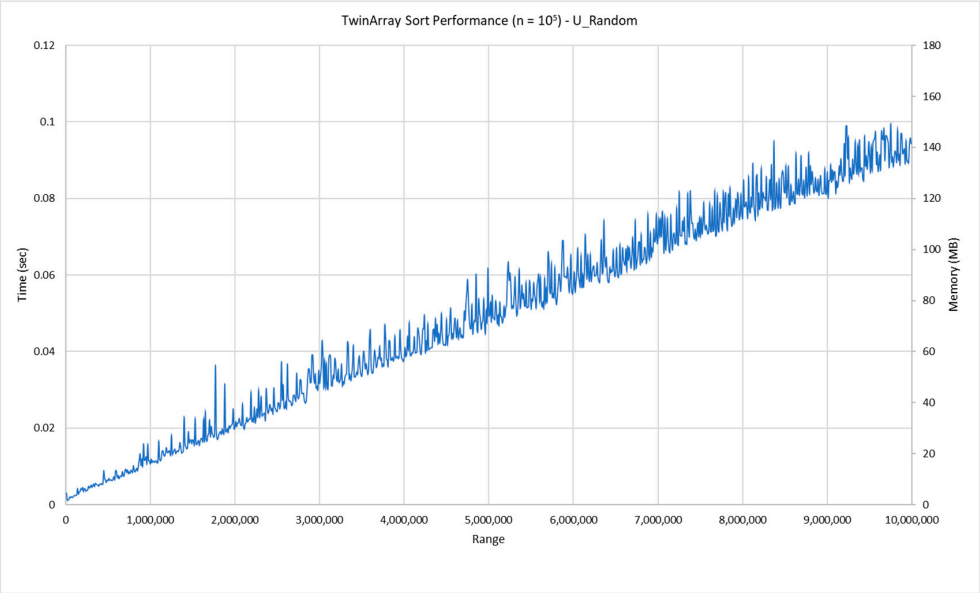

Figure 3. Impact of value range r on TwinArray Sort performance (runtime and memory) for $n = 10^5$ with uniformly random data. A strong linear correlation is observed between range and resource usage.

Like other non-comparison-based sorting algorithms, the TwinArray Sort algorithm showed higher time and memory consumption when compared to other sorting techniques, as seen in Figures 4 and 5, when the range is significantly higher than the number of elements in an array. However, TwinArray Sort performed noticeably better in terms of execution time than Spreadsort, Pigeonhole Sort, and Counting Sort when sorting datasets made up of unique numbers, thanks to its conditional distinct array verifier. Furthermore, TwinArray Sort used less memory than both Spreadsort and Counting Sort, making it a more efficient method of memory consumption.

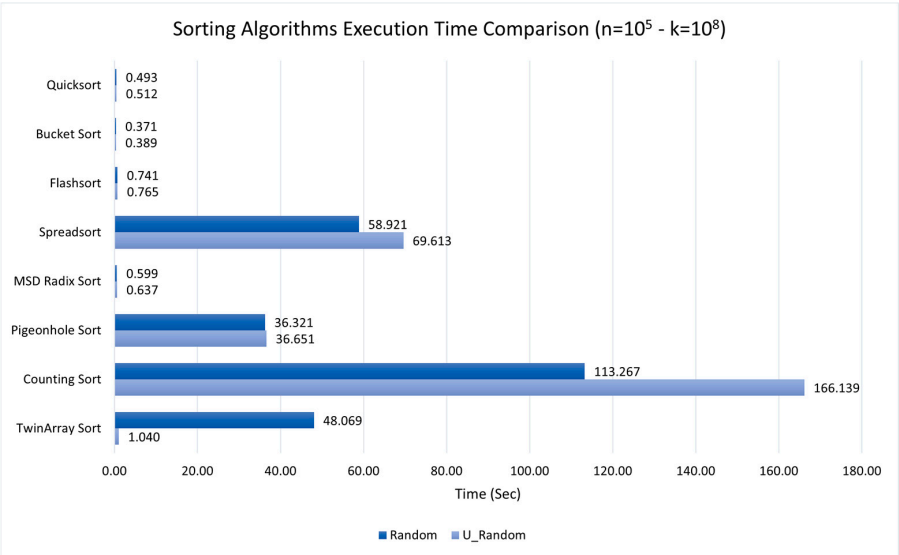

Figure 4. Execution time comparison of TwinArray Sort against other sorting algorithms for both Random and U_Random distributions ($n = 10^5$, $k = 10^8$).

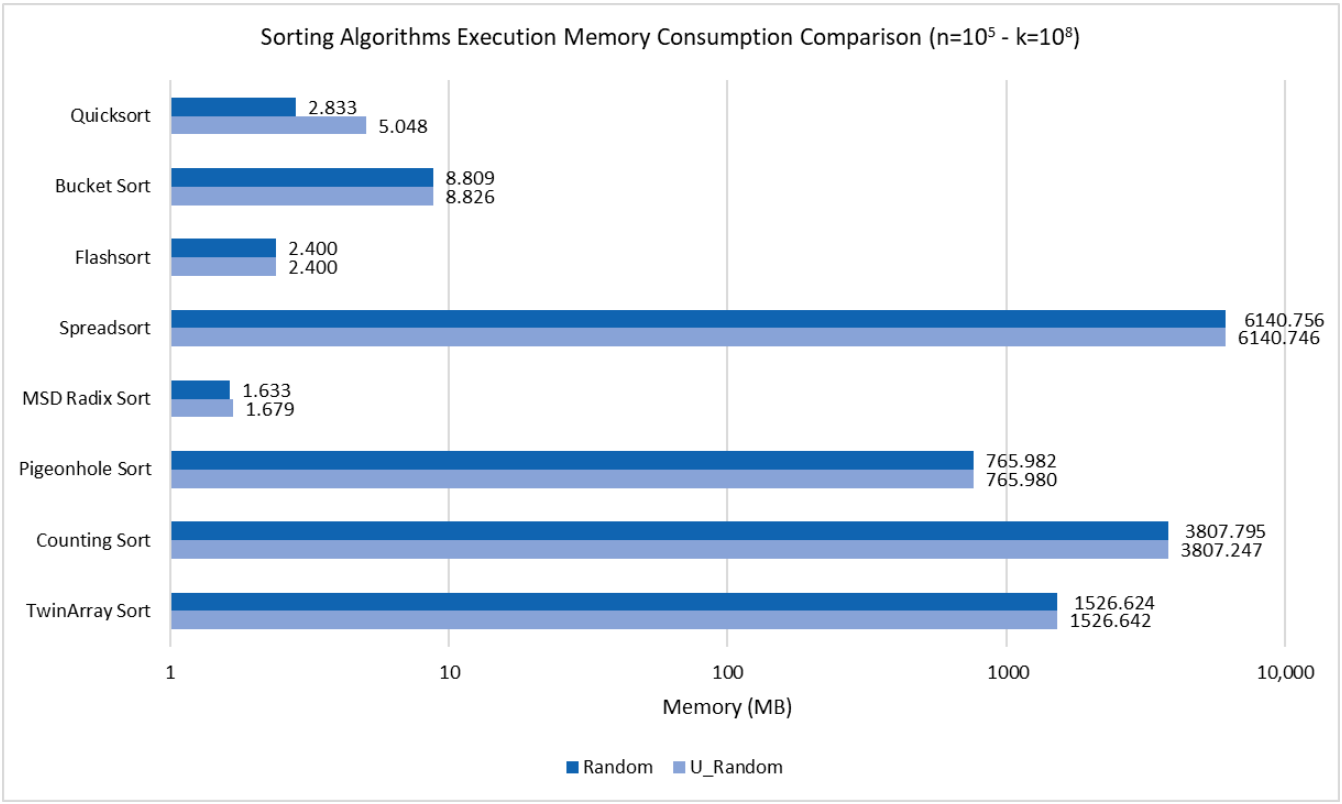

Figure 5. Execution memory comparison of TwinArray Sort against other sorting algorithms for both Random and U Random distributions ($n = 10^5$, $k = 10^8$).

The heatmap in Figure 6 provides a focused analysis of TwinArray Sort under varying input sizes (n) and key ranges (k). This visualisation highlights its scalability patterns, showing consistent performance for smaller and moderate key ranges, with a pronounced increase in execution time when both the dataset size and key range reach their maximum values ($n = 10^6$, $k = 10^6$). This spike is attributable to the algorithm's design, as TwinArray Sort allocates and initialises auxiliary arrays whose size scales linearly with k. When k approaches n at such magnitudes, initialisation costs become significant, and memory accesses lose spatial locality, leading to increased cache miss rates and, in some cases, paging overhead. These effects align with theoretical expectations for counting-based approaches under large key ranges [7,8], indicating that the observed behaviour is inherent to the algorithm rather than an implementation flaw.

Figure 7 isolates TwinArray Sort under Random and Reversed inputs across nine (n, k) settings. Execution time remains low and close between the two distributions when $k \leq 10^5$; Reversed is consistently (but slightly) slower than Random. When both the dataset size and key range reach their maxima ($n = 10^6$, $k = 10^6$), runtime increases sharply (about 1.09–1.11 s), in line with the $O(n + k)$ cost and the memory-hierarchy effects discussed earlier.

Several asymptotic notations can be used to represent the temporal complexity of the TwinArray Sort algorithm in order to characterise its performance (Table 2). The temporal complexity, expressed in Big O notation, $O(n + k)$, indicates that the algorithm must process both the maximum value and every element in the array. This notation provides an upper bound on the algorithm's growth rate. The Big Omega notation, $\Omega(n)$, represents the best-case scenario, where the time required is at least proportional to the number of elements in the array, offering a lower bound on performance. The algorithm's running time is tightly bounded by the Big Theta notation, $\Theta(n + k)$, which encapsulates both the best and worst-case scenarios.

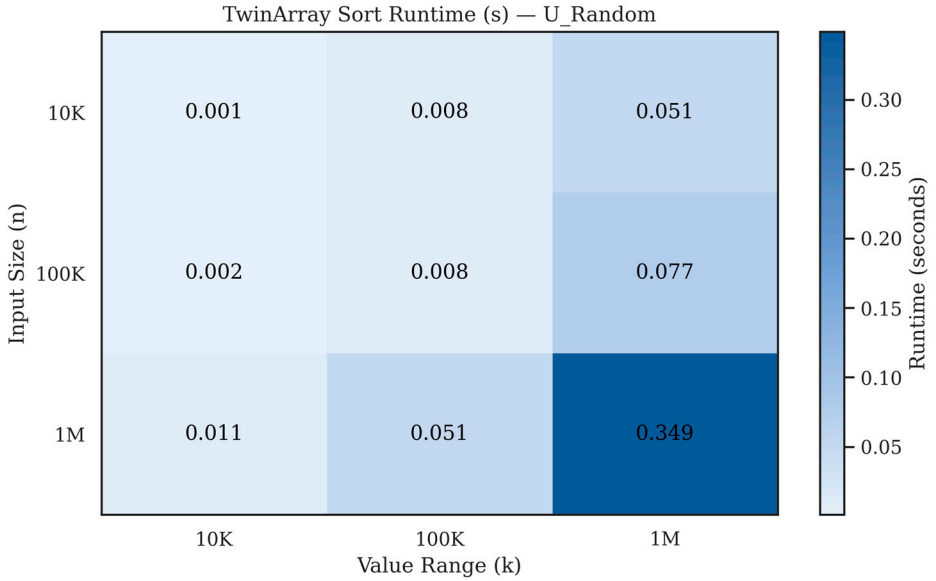

Figure 6. Heatmap of TwinArray Sort runtime performance under varying input sizes (n) and key ranges (k).

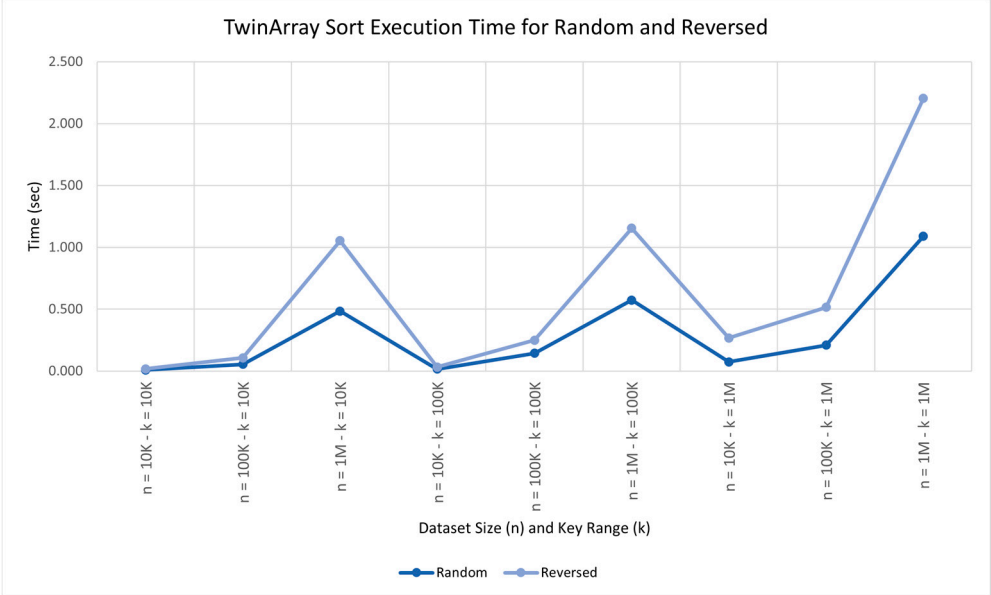

Figure 7. TwinArray Sort execution time (seconds) for Random vs. Reversed inputs across (n, k): [10 K, 10 K], [100 K, 10 K], [1 M, 10 K], [10 K, 100 K], [100 K, 100 K], [1 M, 100 K], [10 K, 1 M], [100 K, 1 M], [1 M, 1 M]. The trajectories show close performance for small/moderate key ranges and a pronounced increase at n = 1 M, k = 1 M.

TwinArray Sort demonstrates superior efficiency compared to standard comparison-based algorithms for large inputs. This is reflected by the Little o notation, o($n$log$n$), indicating that, for sufficiently large n, its growth rate is strictly less than $n$log$n$. Finally, the Little Omega notation, $\omega(n)$, suggests that the algorithm's runtime grows faster than any constant multiple of n; thus, it does not remain constant but increases with input size.

Since the auxiliary arrays scale with k, the efficiency of TwinArray Sort depends on the relationship between n and k. In particular:

Table 2. Asymptotic notations representing the temporal complexity of TwinArray Sort.

Asymptotic Notation	Time Complexity	Description
Big O (O)	$O(n+k)$	Upper bound: The algorithm processes all elements and the maximum value.
Big Omega (Ω)	$\Omega(n)$	Lower bound: In the best case, time is at least proportional to the number of elements.
Big Theta (Θ)	$\Theta(n+k)$	Tight bound: Both best and worst cases involve these terms.
Little o (o)	$o(n\log n)$	The algorithm grows slower than $n\log n$ for large n.
Little omega (ω)	$\omega(n)$	The algorithm grows faster than any constant factor of n.

TwinArray Sort is efficient if $k = O(n)$, equivalently

$$\lim_{n\to\infty}\left(\frac{k}{n}\right) < \infty \tag{5}$$

Moreover, one of the distinctive advantages of TwinArray Sort is its conditional handling of duplicates. Let c_1, c_2, and c_4 be constant coefficients for the input scan, auxiliary traversal, and duplicate reconstruction phases, respectively. There is also a fixed constant overhead c_3 associated with array allocation and initialization; however, since c_3 does not depend on n or k, it is omitted from the following expressions for clarity. The running time can then be expressed as:

$$\begin{aligned} T_{no-dup}(n,k) &= c_1 n \ + \ c_2 k + \ c_3 n \\ T_{dup}(n,k) &= c_1 n \ + \ c_2 k + \ c_4 n \end{aligned} \tag{6}$$

Both cases include the auxiliary-domain scan term c_2k; the difference is in constant-factor reconstruction overhead (extra conditional checks/loop overhead), modelled here as a different linear-in-n coefficient.

$$T_{dup}(n,k) - T_{no-dup}(n,k) = (c_4 - c_3)n > 0 \tag{7}$$

The additional cost in the duplicate case is therefore captured as a constant-factor increase in the linear-in-n reconstruction term, while both cases still include the auxiliary-domain scan term c_2k.

5. Discussion and Conclusions

TwinArray Sort presents a compelling alternative to conventional sorting techniques, demonstrating notable improvements over established non-comparison-based algorithms. Across diverse experimental conditions, it consistently outperformed other sorting methods in terms of execution speed, particularly on datasets with duplicates and moderate key ranges. The algorithm's efficiency is largely attributable to its optimized duplicate handling and the innovative conditional distinct array verification mechanism.

Experimental results show that TwinArray Sort maintains strong performance across a variety of input distributions, including random, reversed, and nearly sorted datasets. Its scalability is evidenced by stable execution times for smaller key ranges and gradual increases when processing larger datasets. TwinArray Sort is best suited for dense-range integer datasets where the key range satisfies $k = O(n)$; performance degradation is expected

when the key range grows significantly larger than the input size. In scenarios involving reversed datasets, its performance advantage becomes more pronounced, indicating that the algorithm is capable of detecting and leveraging data patterns to reduce processing overhead. With respect to memory usage, TwinArray Sort occupies a middle ground: it requires more auxiliary space than in-place algorithms such as Quicksort, Flashsort, and MSD Radix Sort, but consumes considerably less memory than Counting Sort and Spreadsort. This positions it as a competitive option in environments where higher auxiliary memory usage is acceptable in exchange for reduced runtime, particularly when compared against other counting-based methods such as Counting Sort and Spreadsort. TwinArray Sort is not an in-place algorithm, and its memory consumption increases linearly with the key range k; however, under identical conditions it consistently requires significantly less memory than Counting Sort and Spreadsort.

Although the algorithm demonstrates impressive versatility, performance declines in situations where the key range (k) is significantly larger than the dataset size (n). This limitation arises from the increased memory and time complexity associated with constructing large auxiliary arrays. Nevertheless, even under such conditions, TwinArray Sort remains competitive with established methods, and there are promising opportunities to address this drawback through hybrid or adaptive approaches. For example, a dynamic algorithm selection mechanism could switch to comparison-based sorting once k exceeds a threshold, or compressed auxiliary structures could be employed to reduce the memory footprint.

The characteristics of TwinArray Sort make it applicable to a variety of real-world scenarios where high throughput is critical and moderate memory overhead can be tolerated. It is well-suited for distributed log processing in cluster environments, blockchain transaction ordering where massive and partially ordered transaction lists must be processed with minimal latency, and network packet sequencing in high-speed networking applications where processing delays must be avoided. It could also be beneficial in bioinformatics pipelines for sorting genetic sequence identifiers with large and variable key spaces, as well as in Big Data ETL systems where sorting phases often constitute a performance bottleneck.

From a future generation computing perspective, TwinArray Sort shows strong potential for further optimization in emerging architectures. Recent work on integer sorting and systematic performance analysis further emphasizes the importance of memory–time trade-offs in high-performance sorting algorithms [19,20]. Its data-aware execution flow makes it an attractive candidate for GPU acceleration, parallel CPU implementations, and cloud-native deployments where efficiency translates directly into cost savings. The algorithm could also be extended to streaming contexts, enabling incremental sorting in real-time data processing frameworks.

Future work will focus on developing parallel implementations for multi-core CPUs and GPUs, designing hybrid adaptive variants that dynamically choose the most efficient sorting strategy based on runtime profiling, and exploring domain-specific optimizations for time-series, geospatial, and multimedia datasets. Additional research will also investigate energy consumption profiling to assess suitability for green computing applications.

In conclusion, TwinArray Sort represents a significant advancement in sorting algorithm design, delivering a balanced trade-off between computational speed and memory usage. Its adaptability, consistent superiority in experimental evaluations, and applicability to large-scale, diverse datasets make it a promising candidate for deployment in modern high-performance and distributed computing environments. With targeted future optimizations, TwinArray Sort has the potential to become a foundational sorting solution for next-generation computing systems.

Funding: This research received no external funding. The APC was funded by the author.

Data Availability Statement: The datasets generated and/or analyzed during the current study are available from the corresponding author upon reasonable request.

Acknowledgments: During the preparation of this manuscript, the author used ChatGPT (OpenAI) version 4oto improve the readability and language of the manuscript. The author reviewed and edited the output as necessary and takes full responsibility for the content of this publication.

Conflicts of Interest: The author declare no conflict of interest.

References

1. Schaffer, R.; Sedgewick, R. The Analysis of Heapsort. *J. Algorithms* **1993**, *15*, 76–100. [CrossRef]
2. Al-Kharabsheh, K.S.; AlTurani, I.M.; AlTurani, A.M.I.; Zanoon, N.I. Review on Sorting Algorithms A Comparative Study. *Int. J. Comput. Sci. Secur. (IJCSS)* **2013**, *7*, 120–126.
3. Xiang, W. Analysis of the Time Complexity of Quick Sort Algorithm. In *Proceedings of the 2011 International Conference on Information Management, Innovation Management and Industrial Engineering, Shenzhen, China, 26–27 November 2011*; IEEE: New York, NY, USA, 2011; pp. 408–410.
4. Lobo, J.; Kuwelkar, S. Performance Analysis of Merge Sort Algorithms. In *Proceedings of the 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2–4 July 2020*; IEEE: New York, NY, USA, 2020; pp. 110–115.
5. Horsmalahti, P. Comparison of Bucket Sort and RADIX Sort. *arXiv* **2012**, arXiv:1206.3511. [CrossRef]
6. Knuth, D.E. *The Art of Computer Programming. Volume 3, Sorting and Searching*, 2nd ed.; Addison-Wesley: Reading, MA, USA, 1998; Volume 3, ISBN 0201896850.
7. Song, C.; Li, H. Improvement of Counting Sorting Algorithm. *J. Comput. Commun.* **2023**, *11*, 12–22. [CrossRef]
8. Usmani, A.R. A Novel Time and Space Complexity Efficient Variant of Counting-Sort Algorithm. In *Proceedings of the 2019 International Conference on Innovative Computing (ICIC), Lahore, Pakistan, 1–2 November 2019*; IEEE: New York, NY, USA, 2019; pp. 1–6.
9. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009; ISBN 0262033844.
10. Roy, H.; Shafiuzzaman, M.; Samsuddoha, M. SRCS: A New Proposed Counting Sort Algorithm Based on Square Root Method. In *Proceedings of the 2019 22nd International Conference on Computer and Information Technology (ICCIT), Dhaka, Bangladesh, 18–20 December 2019*; IEEE: New York, NY, USA, 2019; pp. 1–6.
11. Baber, M. An Implementation of the Radix Sorting Algorithm on the Touchstone Delta Prototype. In *Proceedings of the Sixth Distributed Memory Computing Conference, 1991. Proceedings*; IEEE: New York, NY, USA, 1991; pp. 458–461.
12. Burnetas, A.; Solow, D.; Agarwal, R. An Analysis and Implementation of an Efficient in-Place Bucket Sort. *Acta Inform.* **1997**, *34*, 687–700. [CrossRef]
13. Mahmoud, H.; Flajolet, P.; Jacquet, P.; Régnier, M. Analytic Variations on Bucket Selection and Sorting. *Acta Inform.* **2000**, *36*, 735–760. [CrossRef]
14. Martínez, C.; Roura, S. Optimal Sampling Strategies in Quicksort and Quickselect. *SIAM J. Comput.* **2001**, *31*, 683–705. [CrossRef]
15. Chaudhuri, R.; Dempster, A.C. A Note on Generating a Worst Case Sequence for Quicksort in Linear Time. In *Proceedings of the 1994 ACM Symposium on Applied Computing—SAC '94*; ACM Press: New York, NY, USA, 1994; pp. 566–567.
16. Ross, S.J. The Spreadsort High-Performance General-Case Sorting Algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*; CSREA Press: Las Vegas, NV, USA, 2002.
17. Neubert, K.D. The Flashsort1 Algorithm. *Dr. Dobb's J.* **1998**, *23*, 123–129.
18. Hertz, M.; Berger, E.D. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*; ACM: New York, NY, USA, 2005; pp. 313–326.
19. Dong, X.; Dhulipala, L.; Gu, Y.; Sun, Y. Parallel Integer Sort: Theory and Practice. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 301–315.
20. Sundaramoorthy, S.; Karunanidhi, G. A Systematic Analysis on Performance and Computational Complexity of Sorting Algorithms. *Discov. Comput.* **2025**, *28*, 250. [CrossRef]