Comparative Analysis of Sorting Algorithms: A Review

Conference Paper · November 2024

DOI: 10.1109/ISCMI63661.2024.10851593

6 authors, including:

Mohammed Alaa Ala'anzy
Suleyman Demirel University
23 PUBLICATIONS 335 CITATIONS
SEE PROFILE

Zhanar Mazhit
Suleyman Demirel University
1 PUBLICATION 1 CITATION
SEE PROFILE

Abdulmohsen Algarni
King Khalid University
62 PUBLICATIONS 1,077 CITATIONS
SEE PROFILE

Ramis Akhmedov
Suleyman Demirel University
6 PUBLICATIONS 18 CITATIONS
SEE PROFILE

Comparative Analysis of Sorting Algorithms: A Review

1st Mohammed Alaa Ala'anzy
Member, IEEE,
Department of Computer Science,
SDU University
Almaty, Kazakhstan
m.alanzy@ieee.org
**Corresponding author*

2nd Zhanar Mazhit
Department of Computer Science,
SDU University
Almaty, Kazakhstan
210103161@stu.sdu.edu.kz

3rd Alaa Fadhil Ala'anzy
Department of Business Administration,
Al-Hikma University College
Baghdad, Iraq
alaa.fadhil@hiuc.edu.iq

4th Abdulmohsen Algarni
Department of Computer Science,
King Khalid University
Abha 61421, Saudi Arabia
a.algarni@kku.edu.sa

5th Ramis Akhmedov
Department of Computer Science,
SDU University
Almaty, Kazakhstan
ramis.akhmedov@sdu.edu.kz

6th Abdiyev Bauyrzhan
Department of Computer and Software Engineering,
Eurasian National University
Astana, Kazakhstan
baurabdi@mail.ru

2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI) | 979-8-3315-1812-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCMI63661.2024.10851593

***Abstract*—In the realm of computer science, sorting algorithms play a pivotal role in optimising data organisation and retrieval processes across various applications. This paper presents a comprehensive examination and comparative analysis of the most commonly used sorting algorithms, focusing on their time, space complexity, and efficiency. By scrutinising both traditional methods like bubble sort, selection sort, and insertion sort, and advanced techniques such as merge sort, quick sort, heap sort, and radix sort, this study sheds light on the performance metrics of each algorithm. Through meticulous implementation and evaluation across datasets of varying magnitudes, ranging from 100 to 100,000 integers, the research provides valuable insights into the strengths and weaknesses of these sorting techniques. The outcomes of this study offer practical guidance for professionals in data engineering and software development, aiding in the informed selection of sorting algorithms tailored to specific application needs. Utilising a rigorous experimental setup with the pseudocode, the study meticulously measures the time complexity of each algorithm, recording CPU time in seconds. The analysis reveals significant performance differentials among the algorithms, showcasing the efficiency of quick sort, shell sort, and heap sort on small datasets, while highlighting the scalability of merge sort, radix sort, and quick sort as dataset sizes increase. This research contributes to the advancement of algorithmic knowledge and provides a valuable resource for practitioners seeking to optimise sorting processes in their computational tasks.**

***Index Terms*—Sorting algorithms, Time complexity, Efficiency.**

I. Introduction

Sorting is a cornerstone operation in computer science, integral to various applications ranging from data organisation and searching to data analysis and beyond [1]. Efficient sorting algorithms are crucial for optimising performance in these tasks, and over the years, numerous algorithms have been developed, each with its own unique strengths and weaknesses. The selection of the most appropriate sorting algorithm for a specific scenario can be complex, influenced by factors such as dataset size, data distribution, stability requirements, and available computational resources [2].

In this research paper, we undertake a comprehensive comparative analysis of thirteen popular sorting algorithms. The algorithms under consideration are insertion sort, selection sort, bubble sort, shell sort, cycle sort, gnome sort, Cocktail Shaker sort, quick sort, heap sort, merge sort, counting sort, radix sort, and bucket sort. These algorithms represent a diverse array of approaches, including simple comparison-based methods, advanced divide-and-conquer strategies, and specialised non-comparison-based techniques.

Optimising the performance of these algorithms is crucial, particularly in environments such as cloud computing [3], [4], big data analytics, Internet of Things (IoT), wireless sensor networks (WSNs) [5], real-time systems, high-performance computing (HPC) [6], database management systems, image and video processing [7], e-commerce platforms, cybersecurity and intrusion detection systems, and many more, where efficient resource utilisation is paramount. Previous studies, such as [8], [9] highlight the importance of selecting appropriate algorithms to enhance system efficiency and performance.

Insertion sort and selection sort are straightforward algorithms, building the sorted list one element at a time and suffering from $O(n^2)$ time complexity, making them less efficient for larger datasets. Bubble sort, while simple, also exhibits $O(n^2)$ time complexity due to its repeated comparisons and swaps of adjacent elements. Shell sort improves upon insertion sort by introducing a gap sequence, enhancing efficiency by reducing comparisons and swaps. Cycle sort minimises write operations by determining element positions within cycles, useful for scenarios prioritising write minimisation. Gnome sort, akin to insertion sort but repositioning elements like a gnome sorting pots, remains inefficient for larger datasets. Cocktail Shaker sort, a bidirectional bubble sort variant, traverses in both directions to slightly optimise comparisons. Quick sort, using a pivot to divide-and-conquer recursively, achieves average $O(n \log n)$ time complexity, though poorly chosen pivots can degrade performance to $O(n^2)$. Heap sort, based on binary heaps, achieves $O(n \log n)$ time complexity by repeatedly extracting maximum elements. merge sort, another divide-and-conquer approach, maintains stability and $O(n \log n)$ time complexity, suitable for large datasets. The counting sort, with linear $O(n + k)$ time complexity, counts occurrences of keys for sorting integers efficiently. Radix sort extends counting sort for digit-by-digit processing, maintaining linear time complexity for large keys. Bucket sort distributes elements into buckets, sorting each individually, effective for uniformly distributed data by leveraging diverse sorting strategies within each bucket.

Our examination encompasses various performance indicators, such as execution time, memory usage, algorithmic complexity, stability, and other essential attributes. The execution time metric offers insights into algorithm efficiency, while memory usage reflects space complexity. Delving into algorithmic complexity aids in understanding theoretical efficiency, and stability analysis evaluates whether algorithms maintain the relative order of elements with identical keys.

Through this thorough comparative examination, our goal is to offer a comprehensive comprehension of these sorting algorithms' performance attributes. Our discoveries will assist developers and practitioners in choosing the most suitable sorting algorithm based on particular requirements and dataset characteristics. Moreover, our research contributes to a broader understanding of sorting algorithms, highlighting their strengths, weaknesses, and trade-offs.

Subsequent sections of this paper offer detailed explanations of sorting algorithms and discuss the datasets used for evaluation. Section II provides a review of related research on sorting algorithms. Section III presents a comprehensive review and comparison of various sorting algorithms. The comparative results are detailed in Section IV, where we analyse algorithm performance based on predefined metrics. Finally, Section V summarises the implications of our findings and offers practical advice for selecting sorting algorithms in real-world scenarios.

II. Related work

To achieve meaningful results in any computer science problem, it is essential to enhance the performance of the chosen sorting algorithm. There are multiple methods available for achieving this goal. Each sorting algorithm can be refined in several ways, and it is crucial to compare these refinements to demonstrate the superiority of our chosen algorithm. The study conducted by [10] offers a thorough comparison of bubble sort and insertion sort algorithms, focusing on their time complexities and runtime efficiencies. Utilising the Java programming language, the research meticulously analyses their computational performance across datasets of varying sizes and complexities. Results consistently highlight insertion sort's superior efficiency over bubble sort, positioning it as the preferred option for sorting tasks across a range of data volumes. This empirical investigation enhances understanding of algorithmic efficiencies and informs strategic algorithm selection for diverse computational applications. A study by [11], trackle sorting and merging algorithms in computer science, comparing the runtime performance of selection and merge algorithms using Java's System nanoTime on an octa-core processor. They find that merge sort outperforms selection sort, particularly with multiple CPU cores and Java's concurrency utilities, highlighting the efficiency gains of multi-core deployment. In [12], the authors conducted a thorough review focusing on two prominent sorting algorithms: quick sort and merge sort, both categorised under divide-and-conquer strategies. While their study offers comprehensive insights into these algorithms, it is limited in scope as it excludes consideration of alternative sorting methods. Thus, a broader evaluation encompassing additional algorithms would enhance the comprehensiveness of their review.

Another study by [13] offers a comparative analysis of three sorting algorithms: comb sort, cocktail sort, and counting sort. However, this study is limited in its scope, focusing solely on these three algorithms without considering a broader range of sorting methods. Additionally, a study conducted by [14] offered a restricted comparative examination of five sorting algorithms: quick sort, merge sort, insertion sort, bubble sort, and selection sort, focusing solely on their time and space complexities.

The study by [15], emphasises the critical role of time and memory complexities in sorting algorithms. Efficient sorting not only enhances algorithm performance but also optimises resource usage. The research explores different sorting algorithms across various data types to determine their optimal efficiency, including notable findings on shell sort and comparisons under diverse dataset conditions. However, the algorithms used were limited to insertion sort, shell sort, quick sort, and selection sort.

Authors in [16] have provided a review of a limited set of sorting algorithms such as bubble sort, insertion sort, quick sort, gnome sort, grouping comparison sort, and enhanced bubble sort in order to come up with a developed new quadratic sorting algorithm to overcome limitations in existing methods. Their approach treats an unsorted data sequence as a collection of disjoint sorted sub-sequences. They introduced a novel method to address the sorting of unsorted data sequences. The algorithm showcases its superior efficiency against existing quadratic sorting methods.

[17] categorised sorting algorithms into value-based and index-based methods, and into comparison-based and non-comparison-based logic. Comparison-based algorithms have a lower bound of $\Omega(n \log n)$, while non-comparison algorithms like counting, radix, and bucket sorts can surpass this bound with additional resources or assumptions. Stable algorithms,

which maintain the initial ordering of elements, are preferred in certain applications. Algorithm selection depends on ease of understanding, value distribution, and range. For example, bucket sort is ideal for evenly distributed values, and radix sort is suitable for large numbers with consistent digit lengths.

[18] evaluated sorting algorithms including quick, bubble, merge, radix, and counting sort based on time and memory complexities. Non-comparison-based algorithms showed superior runtime performance and lower memory usage compared to comparison-based ones, which suffered significant performance degradation as array sizes increased. Quick sort, however, had higher memory consumption due to recursion and pivot handling. The study concluded that non-comparison-based algorithms generally offer more efficient solutions in terms of both time and space complexities. [19] compared stable algorithms (merge, insertion, bubble) with unstable ones (quick, heap, selection), considering platform-dependent and -independent factors. They analyzed computational complexity, memory usage, and stability, finding insertion sort optimal for small datasets among stable options, and merge sort preferable for larger datasets. Among unstable sorts, selection sort is best for smaller datasets, while heap sort suits larger ones. Merge sort is ideal for first-come-first-serve scenarios, offering valuable insights for developers choosing algorithms for various computational needs.

In [20], authors present a thorough comparative analysis of six sorting algorithms: bubble sort, selection sort, insertion sort, merge sort, quick sort, and heap sort. Their study focuses on evaluating time complexities across best, average, and worst-case scenarios, and discusses algorithm stability. Unlike prior device-specific studies, this research adopts a machine-independent approach, providing a generalised perspective through mathematical abstraction. It emphasises the importance of considering factors like data type handling, performance consistency, and code complexity when selecting sorting algorithms tailored to specific application needs. Another study [21], compares six sorting algorithms: quick sort, Tim sort, merge sort, heap sort, radix sort, and shell sort. Quick sort is fast for smaller datasets but less so for larger ones. Tim sort and merge sort consistently perform well. Heap sort is reliable but slower, radix sort works best with smaller datasets, and shell sort maintains consistent performance but can be slower. Choosing the best algorithm depends on dataset size, distribution, stability, memory usage, and time complexity.

In our study, we conducted an extensive review by analysing thirteen sorting algorithms: bubble sort (Bub), insertion sort (Ins), selection sort (Sel), merge sort (Mer), quick sort (Qui), heap sort (Hea), counting sort (Cou), radix sort (Rad), bucket sort (Buc), Cocktail-Shaker sort (Coc), shell sort (She), cycle sort (Cyc), and gnome sort (Gno), as demonstrated in Table I. This comprehensive analysis offers a broad perspective on the efficiency of diverse sorting techniques, providing valuable insights into their performance across varying scenarios and dataset sizes. Furthermore, we present a comparative analysis of these algorithms along with their pseudocodes, followed by a critical discussion of their strengths and limitations.

TABLE I: Comparison of Related Works and Sorting Algorithms

Ref.	**Bub.**	**Ins.**	**Sel.**	**Mer.**	**Qui.**	**Hea.**	**Cou.**	**Rad.**	**Buc.**	**Coc.**	**She.**	**Cyc.**	**Gno.**
[10]	✓	✓											
[11]			✓	✓									
[12]				✓	✓								
[13]							✓			✓			
[14]	✓	✓	✓	✓	✓								
[15]		✓	✓		✓						✓		
[16]	✓	✓			✓								✓
[17]							✓	✓	✓				
[18]	✓			✓	✓		✓	✓					
[19]	✓	✓	✓	✓	✓	✓							
[20]	✓	✓	✓	✓	✓	✓							
[21]				✓	✓	✓		✓			✓		
Ours	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

III. Review of sorting algorithms

A. Bubble sort

Bubble sort is known for its simplicity among sorting algorithms. It operates by traversing the list repeatedly, comparing adjacent elements, and swapping them if they are in the wrong order. This process continues until the list is arranged. Unlike more sophisticated algorithms, bubble sort moves in one direction during each iteration, causing larger elements to progressively move towards their correct positions through successive passes.

The straightforward nature of bubble sort makes it an excellent choice for introducing basic sorting concepts. However, its time complexity of $O(n^2)$ in both average and worst-case scenarios restricts its effectiveness for sorting large datasets. It performs best with small lists or those that are almost sorted, where its simplicity and minimal overhead occasionally result in better performance compared to more complex algorithms. Algorithm 1 demonstrates the algorithm pseudocode.

Algorithm 1: Bubble sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.
for $i = 0$ to $n - 1$ do
    set swapped = false;
    for $j = 0$ to $n - i - 1$ do
        if $A[j] > A[j + 1]$ then
            wap $A[j]$ and $A[j + 1]$;
            set swapped = true;
        end
    end
    if $Swapped = False$ then
        break;
    end
end
```

Strengths:

- Simplicity: bubble sort is easy to grasp and implement, making it an excellent choice for introducing fundamental sorting principles.
- Efficiency for Small Datasets: With very small datasets, bubble sort can demonstrate unexpected efficiency, as the overhead of more complex algorithms may not be justified.

- Adaptability to Nearly sorted Lists: In cases where the list is mostly sorted, bubble sort can outperform its worst-case scenario by terminating early once no swaps are required.

Weaknesses:

- Inefficiency with Large Datasets: bubble sort exhibits a worst-case and average-case time complexity of $O(n^2)$, rendering it highly inefficient for sorting large datasets compared to advanced algorithms like quick sort or merge sort.
- Redundant Operations: The algorithm performs unnecessary comparisons and swaps, particularly in its initial stages, resulting in increased overall execution time.
- Limited Practical Use: Due to its inefficiency, bubble sort sees little practical application where performance is crucial, except as a pedagogical tool or in specific scenarios where its characteristics are advantageous.

Bubble sort, a basic but inefficient sorting algorithm, arranges elements by iterative comparing and swapping adjacent items. Its main advantages are its simplicity and efficacy with very small or nearly sorted datasets. However, its quadratic time complexity renders it unsuitable for large datasets, often surpassed by more sophisticated sorting algorithms in real-world scenarios. While valuable for teaching fundamental sorting concepts, bubble sort sees little use in performance-critical settings.

B. Insertion sort

Insertion sort is a straightforward sorting technique that constructs the final sorted list by gradually inserting one element at a time [22]. It operates by selecting elements from the unsorted list and placing them into their correct positions within the already sorted portion of the list. Starting from the second element, the algorithm compares it with elements in the sorted section, shifting larger elements to the right to accommodate the new element.

Insertion sort is known for its ease of implementation and efficiency on small or nearly sorted datasets. It achieves a best-case time complexity of $O(n)$ when the list is already sorted. However, its average and worst-case time complexity of $O(n^2)$ limits its suitability for large datasets. Nevertheless, its stability and capability for in-place sorting make it suitable for specific applications, especially those involving small arrays or lists that are mostly sorted. Algorithm 2 demonstrates the algorithm pseudocode.

Strengths:

- Ease of Understanding: insertion sort is straightforward to grasp and implement, serving as a suitable introductory algorithm for individuals learning about sorting.
- Efficiency with Small Datasets: Particularly effective with small or nearly sorted datasets, insertion sort demonstrates efficient performance, boasting an average and worst-case time complexity of $O(n^2)$ while showcasing optimal performance with $O(n)$ in the best-case scenario (i.e., with an already sorted list).
- Stability: insertion sort maintains stability by conserving the relative order of equal elements throughout the sorting process.
- In-Place sorting: The algorithm perform sorting operations directly on the provided list, necessitating only a fixed amount of extra memory space.

Weaknesses:

- Inefficiency with Large Datasets: In scenarios involving large datasets, insertion sort's time complexity of $O(n^2)$ in average and worst-case instances renders it less efficient compared to alternatives like quick sort or merge sort.
- Redundant Operations: insertion sort may perform unnecessary comparisons and shifts, particularly when elements are distant from their eventual positions within the sorted sequence.

Algorithm 2: Insertion sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$. set N = A.length;
for $i = 1$ to $n$ do
    set key = A[i];
    set j = i - 1;
    while $j >= 0$ and $A[j] > key$ do
        set A[j + 1] = A[j];
        set j = j - 1;
    end
    set A[j + 1] = key;
end
```

Insertion sort stands out as a simple and reliable sorting method, particularly beneficial for managing small or nearly sorted datasets, owing to its straightforwardness and effectiveness in such contexts. This algorithm constructs a sorted sublist incrementally, meticulously positioning each new element in its rightful place. However, its time complexity of $O(n^2)$ renders it less practical for larger datasets, where more sophisticated algorithms would offer superior performance. Nevertheless, despite its constraints, insertion sort retains value, serving as a valuable educational resource and finding utility in specific applications that capitalise on its inherent traits.

C. Selection sort

Selection sort, a basic comparison-based sorting technique, functions by repeatedly selecting the minimum element from the unsorted section of the list and swapping it with the leftmost unsorted element. This process continues, shifting the boundary between the sorted and unsorted subsets one step to the right, until the entire list is sorted.

The simplicity and straightforward nature of selection sort make it valuable for educational purposes. However, its time complexity of $O(n^2)$ for both average and worst-case scenarios limits its efficiency for sorting large datasets. Furthermore, selection sort lacks stability, meaning that it does not maintain the relative order of identical elements. Despite these limitations, its simplicity and minimal swapping operations can be beneficial in certain scenarios, especially when minimising write operations is crucial. Algorithm 3 demonstrates the algorithm pseudocode.

Strengths:

- Simplicity: selection sort's clear and straightforward approach makes it accessible and effective for teaching essential sorting principles.
- In-Place sorting: It sorts the input array directly, requiring minimal additional memory usage.
- Minimal Swaps: selection sort minimises the number of swaps needed, typically requiring a maximum of $(n - 1)$ swaps, which can be advantageous when minimising write operations is important.

Weaknesses:

- Inefficiency with Large Datasets: Its time complexity of $O(n^2)$ makes it less efficient for sorting large datasets compared to more optimised algorithms like quick sort or merge sort.
- Unstable sorting: selection sort is unstable, meaning it does not ensure the preservation of the relative order of equal elements.
- Redundant Comparisons: The algorithm performs numerous comparisons, even in cases where the list is partially sorted, potentially resulting in inefficiencies.

Algorithm 3: Selection sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.   function
 selection_sort($A, left, right$):
    set N = $A.length$;
    for $i = 0$ to $N - 1$ do
        set min_idx = $i$;
        for $j = i + 1$ to $N$ do
            if $A[min\_idx] > A[j]$ then
                set min_idx = $j$;
            end
        end
        swap $A[i]$ and $A[min\_idx]$;
    end
return
```

Selection sort is a simple sorting method that operates by iteratively choosing the smallest element from the unsorted sublist and exchanging it with the leftmost unsorted element. Its straightforwardness and reduced number of exchanges render it valuable for modest datasets or instructional use. Nonetheless, its $O(n^2)$ time complexity renders it unsuitable for sizable datasets, and its absence of stability restricts its applicability in specific contexts. Despite these limitations, selection sort persists as a foundational algorithm for grasping sorting fundamentals.

D. Cocktail-shaker sort

Cocktail-shaker sort, also known as bidirectional bubble sort or shaker sort, is a variation of the bubble sort algorithm. Unlike the traditional bubble sort that only traverses the list in one direction, cocktail-shaker sort traverses the list in both directions alternately. This bidirectional approach can sometimes result in fewer passes over the list, potentially leading to performance improvements over the classic bubble sort, especially in scenarios where the list elements are initially in a more favorable order.

The cocktail-shaker sort is particularly effective for lists where elements are not entirely out of order but rather slightly disordered. By sorting in both directions, the algorithm can move smaller and larger elements toward their correct positions more efficiently than the unidirectional bubble sort.

The time complexity of the cocktail-shaker sort remains $O(n^2)$ in both average and worst cases, rendering it less suitable for large datasets when compared to more advanced algorithms like quick sort or merge sort. However, its simplicity and bidirectional mechanism can occasionally provide practical advantages in certain use cases or with small datasets. Algorithm 4demonstrates the algorithm pseudocode.

Algorithm 4: Cocktail-Shaker sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.
function shaker_sort($A : array\ of\ items$):
    set n = length($A$);
    set swapped = true;
    set start = 0;
    set end = n - 1;
    while $swapped$ do
        set swapped = false;
        for $i = start\ to\ end - 1$ do
            if $A[i] > A[i + 1]$ then
                swap $A[i]$ and $A[i + 1]$;
                set swapped = true;
            end
        end
        if $swapped = false$ then
            break;
        end
        set swapped = false;
        set end = end - 1;
        for $i = n - 1$ to $start - 1$ do
            if $A[i] > A[i + 1]$ then
                swap $A[i]$ and $A[i + 1]$;
                set swapped = true;
            end
        end
        set start = start + 1;
    end
end
```

Strengths:

- Simplicity: Easy to understand and implement.
- Better than bubble sort for Some Cases: More efficient than bubble sort when dealing with certain types of nearly sorted data.
- Stability: Maintains the relative order of equal elements.

Weaknesses:

- Inefficiency: It has a time complexity of $O(n^2)$ in both the average and worst cases, making it inefficient for large datasets.
- Redundant Comparisons: Like bubble sort, it performs many unnecessary comparisons and swaps.
- Not Suitable for Large Datasets: Due to its quadratic time complexity, it is not practical for sorting large lists.

Cocktail-shaker sort is a straightforward and stable sorting algorithm that can outperform the traditional bubble sort in certain cases, particularly when the list is nearly sorted. However, its $O(n^2)$ time complexity makes it impractical for large datasets.

E. Shell sort

Shell sort is an in-place comparison-based sorting algorithm that extends insertion sort by allowing the exchange of items that are far apart. The fundamental idea is to arrange the elements such that, starting from any point, selecting every $h-th$ element results in a sorted sequence. This process creates an h-sorted list. The algorithm begins by sorting pairs of elements that are far apart and gradually decreases the gap between compared elements.

Shell sort enhances insertion sort by quickly moving values to their appropriate positions, which helps in partially sorting the list and reduces the number of shifts needed during the final insertion sort pass. The performance of shell sort significantly depends on the choice of the increment sequence (gap sequence). Common sequences include the original sequence proposed by shell $(N/2, N/4, \ldots, 1)$ and others like Hibbard's increments and Sedgewick's increments.

The time complexity of shell sort varies based on the gap sequence. In the worst case, it can be $O(n^2)$, but with an optimal gap sequence, it can be reduced to $O(n^{2/3})$ or even $O(nlog^2n)$. Algorithm 5 demonstrates the algorithm pseudocode.

Algorithm 5: Shell sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
 shell_sort($A$ : $array\ of\ items$):
    set n = length($A$);
    set gap = n // 2;
    while $gap > 0$ do
        for $i = gap\ to\ n-1$ do
            set temp = arr[i];
            set j = i;
            while $j >= gap\ and\ A[j-gap] > temp$ do
                set gap = n // 2;
                set j -= gap;
            end
            set A[j] = temp;
        end
        set gap //= 2;
    end
end
```

Strengths:

- Efficiency: Faster than bubble sort, insertion sort, and selection sort for medium-sized arrays.
- In-Place: Requires only a small, constant amount of additional memory.
- Adaptability: Performance improves significantly when applied to nearly sorted data.

Weaknesses:

- Complexity: The choice of increment sequence greatly affects the performance, and optimal sequences are non-trivial to determine.
- Complexity: The choice of increment sequence greatly affects the performance, and optimal sequences are non-trivial to determine.
- Non-Stable: Does not maintain the relative order of equal elements.
- Less Optimal for Large Data: Other algorithms like quick sort or merge sort are generally better for very large datasets.

Shell sort is a powerful extension of the insertion sort that leverages gap sequences to improve sorting efficiency, especially for medium-sized and nearly sorted datasets. The choice of gap sequence is crucial for optimising performance. While it may not be the best choice for very large datasets, its in-place nature and efficiency for certain types of data make it a useful sorting algorithm.

F. Cycle sort

Cycle sort is a comparison-based sorting algorithm that is particularly effective for situations where minimising the number of writes is crucial. It operates by placing each element in its correct position and ensures that each element is written only once. This property makes it particularly useful in scenarios where writing to memory is expensive, such as in embedded systems or when using flash memory.

Cycle sort works by identifying cycles in the permutation of the array. Each cycle can be rotated to place all elements in their correct positions. If an element is already in its correct position, it is skipped. This process is repeated for every element in the array.

The primary advantage of cycle sort is its minimal number of writes, which is at most $O(n-1)$ for an array of n elements. However, the time complexity remains $O(n^2)$ due to the element comparisons required to identify cycles. Despite this, its write efficiency makes it an interesting choice in certain contexts. Algorithm 6 demonstrates the algorithm pseudocode.

Algorithm 6: Cycle sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
 cycle_sort($A$ : $array\ of\ items$):
    set n = length($A$);
    for $cycleStart = 0\ to\ n-2$ do
        set item = A[cycleStart];
        set pos = cycleSstart;
        for $i = cycleStart + 1\ to\ n-1$ do
            if $A[i] < item$ then
                increment pos;
            end
        end
    end
    if $pos == cycleStart$ then
        continue;
    end
    while $item == A[pos]$ do
        increment pos;
    end
    swap $A[pos]$ and $item$;
    while $pos! = cycleStart$ do
        set pos = cycleStart;
        for $i = cycleStart + 1\ to\ n-1$ do
            if $A[i] < item$ then
                increment pos;
            end
            while $item == A[pos]$ do
                increment pos;
            end
        end
        swap $A[pos]$ and $item$;
    end
end
```

Strengths:

- Minimises Writes: Makes the minimal number of writes necessary to sort the array, which is useful for write-sensitive storage.
- In-Place: Requires no additional memory beyond the input array.
- Theoretically Optimal for Writes: Each element is moved to its correct position in at most one write.

Weaknesses:

- Time Complexity: Despite its benefits in minimising write operations, the algorithm has a time complexity of $O(n^2)$ for both average and worst-case scenarios.
- Instability: It does not maintain the relative order of equal elements, making it unstable.
- Implementation Complexity: It demands a more complex implementation compared to simpler algorithms like bubble sort or insertion sort.

Cycle sort is a unique sorting algorithm that minimises the number of writes, making it particularly useful for scenarios where write operations are costly. While it has a quadratic time complexity, its write efficiency is theoretically optimal.

G. Gnome sort

gnome sort, sometimes dubbed "Stupid sort," is a simplistic sorting algorithm akin to insertion sort, albeit with a different approach to arranging elements. It operates by traversing the array, swapping out-of-order elements, and moving backward until the correct order is achieved. While intuitive and straightforward to implement, gnome sort's efficiency diminishes with larger datasets, attributed to its average and worst-case time complexity of $O(n^2)$. Primarily educational in nature, it finds utility in small arrays or as a foundational concept for understanding sorting algorithms. Algorithm 7 demonstrates the algorithm pseudocode.

Algorithm 7: Gnome sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
  gnome_sort($A$ : $array\ of\ items$):
    set n = length($A$);
    set index = 0;
    while $index < n$ do
        if $index == 0$ or $A[index] >= A[index-1]$ then
            increment index;
        end
        else
            swap $A[index]$ and $A[index-1]$;
            decrement index;
        end
    end
end
```

Strengths:

1) Simplicity: Gnome sort is easy to understand and implement.
2) In-Place: It operates directly on the input array, requiring no additional memory.
3) Adaptability: Shows good performance on small datasets or arrays that are nearly sorted.

Weaknesses:

- Inefficiency: With both worst-case and average-case time complexity of $O(n^2)$, it's inefficient for handling large datasets.
- Lack of Stability: Fails to maintain the relative order of equal elements.
- Redundant Comparisons: Involves numerous unnecessary comparisons and swaps, impacting its efficiency.

Gnome sort is a basic sorting algorithm that is easy to implement and understand. It is best used for small datasets or nearly sorted data due to its simplicity. However, its inefficiency for larger datasets limits its practical use, making it more educational than practical.

H. Counting sort

Counting sort diverges from traditional approaches by abstaining from comparing elements; instead, it organises objects based on distinct key values, similar to a hashing mechanism. Following this, it employs arithmetic operations to determine the index position of each object within the output sequence. Counting sort, unlike many sorting algorithms, is specialised and is not intended for general-purpose use.

Counting sort proves highly efficient when the range of values to be sorted is not substantially larger than the number of objects themselves. Notably, it accommodates sorting of negative input values. Irrespective of the input arrangement, the time complexity of counting sort remains constant, operating in $O(n+k)$ time, where n represents the number of elements and k denotes the range.

Compared to comparison-based sorting techniques, counting sort presents advantages due to its avoidance of element comparisons. However, its efficiency diminishes when dealing with very large integers, as the creation of arrays of such magnitude becomes cumbersome. Algorithm 8 demonstrates the algorithm pseudocode.

Algorithm 8: Counting sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
  count_sort($A$):
    set N = A.length;
    set M = 0;
    for $i = 0$ to $N$ do
        set M = $Math.max(M, arr[i]$;
    end
    set countArray = new int[M + 1];
    for $i = 0$ to $N$ do
        increment countArray[A[i]];
    end
    for $i = 1$ to $M$ do
        set countArray[i] += countarray[i-1];
    end
    set outputArray = new int[N];
    for $i = N - 1$ to $0$ do
        set outputArray[countArray[A[i]]] = A[i];
        decrement countArray[A[i]];
    end
    return outputArray;
end
```

Strengths:

- Efficient Time Complexity: counting sort functions in $O(n+k)$ time, with n being the number of elements to sort and k representing the range of input values. This effectiveness is particularly advantageous when the range k closely matches the size of the input n.

- Stability: counting sort maintains the relative order of elements with equal keys, making it a reliable sorting technique. This feature is valuable when preserving the original order of elements with equal values is essential.
- Ease of Understanding: counting sort is straightforward to grasp and implement. Its simple method involves basic counting and array manipulation, making it suitable for educational purposes and swift implementations.

Weaknesses:

- Large Range Issue: counting sort has a space complexity of $O(k)$ due to the need for a counting array of size k. If the range of input values (k) significantly exceeds the number of elements (n), both space and time efficiency diminish, rendering the algorithm impractical.
- Limited Flexibility: The algorithm requires prior knowledge of the input data range, which may not always be available or feasible, especially with data sets of unknown or extremely large ranges.
- Not Suitable for Non-Integer Keys: counting sort is not directly applicable to data with keys that cannot be mapped to non-negative integers. Sorting non-integer data types, such as floating-point numbers or strings, requires additional modifications to the algorithm.

Counting sort is an efficient, non-comparison-based algorithm that sorts integers or items with integer keys in linear time. It works by counting occurrences of each element and using these counts to determine element positions in the sorted sequence. Advantages include linear time complexity, stability, and simplicity. However, it requires extra memory for the counting array and is less practical for large ranges or non-integer keys, making it best suited for bounded numerical data with a predefined range.

I. Radix sort

Radix sort is a methodical integer sorting algorithm that operates independently of comparisons, arranging numbers by scrutinising each digit in isolation. It utilises counting sort as an intermediary step to arrange based on individual digits, commencing with the least significant digit and advancing to the most significant one.

This sorting approach showcases commendable effectiveness, particularly when managing large datasets containing integer keys. Its time complexity is articulated as $O(d{*}(n{+}b))$, where d indicates the digit count, n signifies the element count, and b represents the base of the number system. Algorithm 9 demonstrates the algorithm pseudocode.

Strengths:

- Speed: radix sort can be extremely fast for sorting fixed length numbers, outperforming comparative sorting algorithms like quick sort and merge sort.
- Non-Comparative: It does not compare elements against each other, which can be more efficient in certain scenarios.

Algorithm 9: Radix sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.   function
 radix_sort($A$):
    set max1 = $max(A)$;
    set exp = 1;
    while $max1/exp >= 1$ do
        $count\_sort(A, exp)$;
        set exp *= 10;
    end
end
function count_sort($A, exp1$):
    set n = $length(arr)$;
    set output = [0] * (n);
    set count = [0] * (10);
    for $i = 0$ to $n$ do
        set index = A[i] // exp1;
        set count[index % 10] += 1;
    end
    for $i = 1$ to $10$ do
        set count[i] += count[i - 1];
    end
    set i = n - 1;
    while $1 >= 0$ do
        set index = A[i] // exp1;
        set output[count[index % 10] - 1] = A[i];
        set count[index % 10] -= 1;
        decrement i;
    end
    set i = 0;
    for $i = 0$ to $length(arr)$ do
        set A[i] = output[i];
    end
end
```

Weaknesses:

- Limited Scope: radix sort is applicable only to elements that can be sorted by digit-like keys (e.g., integers, strings).
- Space Requirement: It requires additional space for storing count arrays and the output array, which can be considerable for large data sets.

Radix sort emerges as an efficient solution for sorting data sets characterised by a fixed length, such as integers or strings. Its effectiveness becomes particularly evident when the dataset's range does not significantly surpass the number of items slated for sorting. In scenarios where the variability within the dataset is relatively constrained, radix sort efficiently organises the elements by examining their digits or characters independently, thereby streamlining the sorting process. This approach ensures that radix sort can handle large datasets with remarkable efficiency, offering a reliable solution for a diverse array of sorting tasks.

J. Bucket sort

Bucket sort, alternatively known as Bin sort, functions by partitioning elements into numerous "buckets". Each bucket subsequently undergoes sorting independently, either through the utilisation of a distinct sorting algorithm or by recursively applying bucket sort. Once sorted, the contents of the buckets are merged to yield the ultimate sorted array.

This sorting technique showcases notable efficiency when the input elements exhibit uniform distribution within a specified range. Under optimal conditions, bucket sort achieves linear time complexity, denoted as $O(n)$. However, in adverse scenarios, its performance may deteriorate to $O(n^2)$. Algorithm 10 demonstrates the algorithm pseudocode.

Algorithm 10: Bucket sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array A of length n.   function
 bucket_sort(A, n):
    if $n <= 0$ then
        return;
    end
    set bucket = new array of n empty lists;
    for $i = 0$ to $n - 1$ do
        set bucketIndex = int(A[i] * n);
        bucket[bucketIndex].append(A[i]);
    end
    for $i = 0$ to $n - 1$ do
        bucket[i].sort();
    end
    set index = 0;
    for $i = 0$ to $n - 1$ do
        for $j$ in $bucket[i]$ do
            set A[index] = j;
            set index += 1;
        end
    end
end
```

Strengths:

- Efficient for Uniform Distribution: bucket sort can be very efficient if the input is uniformly distributed over a range.
- Fast: When the additional conditions (uniform distribution and an effective bucket strategy) are met, it can perform faster than $O(nlogn)$ average complexity for many sorting algorithms.

Weaknesses:

- Dependent on Input Distribution: Its performance heavily depends on the distribution of the input data and the number of buckets.
- Additional Space: Requires additional space for the buckets.

Bucket sort is effective for sorting data that is distributed uniformly across a range. It can be extremely fast under the right conditions but requires careful choice of bucket sizes and sorting algorithm for each bucket.

K. Heap sort

Heap sort is a comparison-based sorting technique using a binary heap. It starts by converting the list into a max-heap, where each parent node is greater than or equal to its child nodes. The algorithm then repeatedly extracts the largest element from the root and restructures the heap until the list is sorted. This ensures the largest elements are moved to the end of the list. Heap sort demonstrates efficiency, boasting a time complexity of $O(nlogn)$ for both average and worst-case scenarios, making it suitable for sorting extensive datasets. Additionally, it operates as an in-place sorting algorithm, necessitating only a constant amount of extra memory. However, it lacks stability, as it does not maintain the relative order of identical elements. Nevertheless, owing to its dependable performance and efficient memory usage, heap sort proves invaluable for various sorting applications. Algorithm 11 demonstrates the algorithm pseudocode.

Algorithm 11: Heap sort Algorithm

```
Result: A sorted array in ascending order.
Precondition An unsorted array A of length n.
function sort(A):
    set N = A.length;
    for $i = N/2 - 1$ to 0 do
        $heapify(A, N, i)$;
    end
    for $i = N - 1$ to 0 do
        swap $A[0]$ and $A[i]$;
        $heapify(A, N, largest)$;
    end
end
function heapify(A, N, i):
    set left = 2*i + 1;
    set right = 2*i + 2;
    set largest = i;
    if $left < N and A[left] > A[largest]$ then
        largest = left;
    end
    if $right < N and A[right] > A[largest]$ then
        largest = right;
    end
    if largest != i then
        swap $A[i]$ and $A[largest]$;
        $heapify(A, N, largest)$;
    end
end
```

Strengths:

- Efficient Performance: heap sort boasts a time complexity of $O(n \log n)$, making it well-suited for efficiently handling large datasets.
- Space Optimisation: As an in-place sorting algorithm, heap sort requires only a constant amount of additional memory space.
- Absence of Worst-Case Scenario: Unlike quick sort, heap sort avoids the risk of encountering a worst-case time complexity of $O(n^2)$.

Weaknesses:

- Lack of Stability: heap sort lacks stability, meaning it does not preserve the relative order of equal elements.
- Implementation Complexity: implementing heap sort may be more intricate compared to simpler alternatives like bubble sort.
- Cache Performance Challenges: heap sort's non-sequential memory access patterns may lead to sub-optimal cache performance on modern computer architectures.

Heap sort is a robust and efficient sorting algorithm that leverages the properties of a binary heap to sort elements in $O(nlogn)$ time. While it offers significant performance advantages and is useful for large datasets, it is not stable and can be more complex to implement. Despite these drawbacks, its consistent performance makes it a valuable tool in scenarios where a guaranteed $O(nlogn)$ sort is required.

L. Merge sort

Merge sort is hailed as a highly efficient and stable sorting algorithm due to its sophisticated divide-and-conquer approach to sorting data. The process begins by recursively breaking down the original list into smaller sub-lists, each containing only one element. This division continues until each sub-list is reduced to its basic form.

Once the sub-lists are at their smallest size, the algorithm starts the merging phase. This critical step involves systematically combining adjacent sublists, comparing their elements, and arranging them in ascending order. The merging process ensures that the resulting sub-lists are progressively larger and sorted correctly. Algorithm 12 demonstrates the algorithm pseudocode.

Algorithm 12: Merge sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
 sort($A, left, right$):
    if $left < right$ then
        set middle = left + (right - left) / 2;
        $sort(A, left, middle)$;
        $sort(A, middle + 1, right)$;
        $merge(A, left, middle, right)$;
    end
end
function merge($A, left, middle, right$):
    set n1 = middle - left + 1;
    set n2 = right - middle;
    set arrLeft = new int[n1];
    set arrRight = new int[n2];
    for $i = 0$ to $n1$ do
        set L[i] = A[l+i];
    end
    for $i = 0$ to $n2$ do
        set R[i] = A[m+1+i];
    end
    set i = 0, j = 0;
    set k = left;
    while $i < n1$ and $j < n2$ do
        if $arrLeft[i] <= arrRight[j]$ then
            set A[k] = arrLeft[i];
            increment i;
        end
        else
            set A[k] = arrRight[j];
            increment j;
        end
        increment k;
    end
    while $i < n1$ do
        set arr[k] = arrLeft[i];
        increment i;
        increment k;
    end
    while $j < n2$ do
        set arr[k] = arrRight[j];
        increment j;
        increment k;
    end
end
```

Strengths:

- Efficiency: merge sort consistently achieves a time complexity of $O(nlogn)$ in all scenarios (best, average, and worst), making it highly effective for handling extensive datasets.
- Stability: It maintains the order of elements with identical keys, ensuring stability in sorting.
- Potential for Parallelisation: The algorithm's divide-and-conquer strategy enables easy parallelisation, improving its scalability on multi-core systems.

Weaknesses:

- Space Complexity: merge sort necessitates extra memory space proportional to the array's size during the merging phase, which can be challenging in memory-limited environments.
- Complexity versus Simplicity: Unlike simpler sorting methods such as bubble sort or insertion sort, merge sort is more complex, potentially making it less suitable for educational settings that prioritise simplicity.

Merge sort is a highly efficient and stable algorithm suited for sorting large datasets. Its divide-and-conquer approach ensures that data is sorted in $O(nlogn)$ time complexity, although it requires additional memory for the merging process. Ideal for applications where stability is crucial and memory availability is not a constraint.

M. Quick sort

Quick sort is hailed as a leading sorting algorithm celebrated for its exceptional efficiency and reliance on comparisons, implementing a divide-and-conquer strategy to achieve peak performance [22], [23]. At its essence lies the pivotal task of partitioning, where a pivot element is chosen, and the array is reorganised to ensure elements smaller than the pivot come before it, while those larger follow suit.

Typically, quick sort demonstrates a time complexity of $O(n \log n)$ on average. However, in less-than-ideal scenarios where pivot selection proves suboptimal, its efficiency may plummet to $O(n^2)$. Nevertheless, despite this potential drawback, quick sort often outperforms alternative algorithms with a time complexity of $O(n \log n)$, such as merge sort, in practical applications. Algorithm 13 demonstrates the algorithm pseudocode.

Algorithm 13: Quick sort

```
Result: A sorted array in ascending order.
Precondition An unsorted array $A$ of length $n$.  function
 partition($A, low, high$):
    set pivot = A[high];
    set i = low - 1;
    for $j = low$ to $high$ do
        if $A[j] <= pivot$ then
            increment $i$;
            swap $A[i]$ and $A[j]$;
        end
    end
    swap $A[i + 1]$ and $A[high]$;
    return $i + 1$;
end
function quick_sort($array, low, high$):
    if $low < high$ then
        set pi = $partition(array, low, high$;
        $quick\_sort(A, low, pi - 1)$ $quick\_sort(A, pi + 1, high)$
    end
end
```

Strengths:

- Efficiency: quick sort typically performs well with an average time complexity of $O(nlogn)$, surpassing simpler algorithms like bubble sort or insertion sort.
- In-Place sorting: It sorts directly within the input array, requiring no extra storage space.
- Versatility: quick sort is adaptable to various data types and can be customised for different scenarios.

Weaknesses:

- Worst-Case Complexity: quick sort's time complexity can degrade to $O(n^2)$ in scenarios where pivot selection consistently falters, significantly impacting its efficiency.
- Lack of Stability: quick sort is not stable, potentially altering the relative order of elements with identical keys.

Quick sort is a popular sorting algorithm with good average performance and low memory usage. It is widely used in practice due to its efficiency and simplicity. However, its performance can degrade to quadratic time if care is not taken with pivot selection.

After conducting a comprehensive comparison of various sorting algorithms, the results are summarised in Table II and Table III. This table provides an overview of the time and space complexities, stability, adaptability, and performance characteristics of each algorithm, aiding in the selection of the most suitable sorting technique for different scenarios.

IV. Result and Discussion

In the evaluation of sorting algorithms, the experimental setup was designed using Java as the programming language, utilising different versions of IntelliJ IDEA, on both Windows 11 and macOS platforms. The hardware configurations included a Core i5 Intel processor with 8GB of RAM and a Core i7 Intel processor with 16GB RAM. The datasets used for testing comprised positive integers, divided into four distinct sizes: 100, 1,000, 10,000, and 100,000 numbers, allowing a comprehensive analysis across varying computational loads. The primary focus was on measuring the algorithms' time complexity, with CPU time recorded in seconds. This setup provided a rigorous test environment to evaluate the efficiency and scalability of each algorithm under consistent and controlled conditions, ensuring reliable performance metrics across different system setups and data volumes. Table IV illustrates the CPU time taken by each algorithm.

Figure 1, Figure 2, Figure 3, and Figure 4 highlight significant differences in performance among the algorithms.

A. *Performance on Small Dataset (100 Integers)*

For the smallest dataset of 100 integers (Figure 1), the running times of most algorithms are negligible, demonstrating high efficiency on limited data volumes. Notably, quick sort, shell sort, and heap sort exhibit the shortest times, showcasing their rapid performance even on smaller scales. This efficiency is particularly advantageous in scenarios where quick operations on small datasets are required.

B. *Performance on Medium Dataset (1,000 and 10,000 Integers)*

As the dataset size increases to 1,000 integers (Figure 2) and 10,000 integers (Figure 3), the performance differentiation becomes more pronounced. Algorithms such as quick sort, merge sort, and radix sort start to exhibit significantly shorter running times. These algorithms are well-known for their efficiency and scalability, making them suitable for medium-sized datasets. On the other hand, algorithms like bubble sort, gnome sort, and selection sort show marked increases in running time, highlighting their inefficiency as data volumes grow.

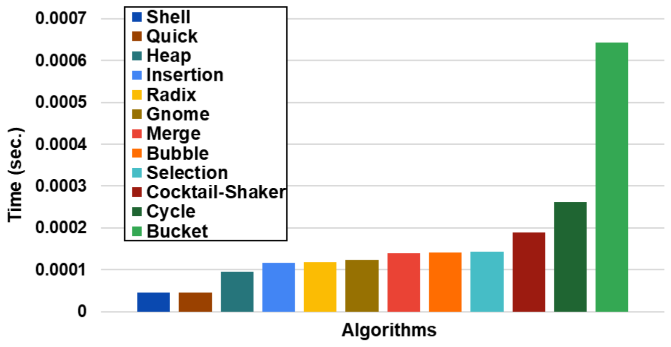

Fig. 1: Runtime for 100 Integers

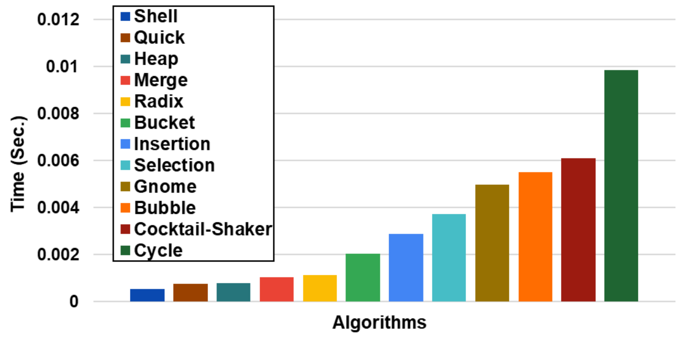

Fig. 2: Runtime for 1000 Integers

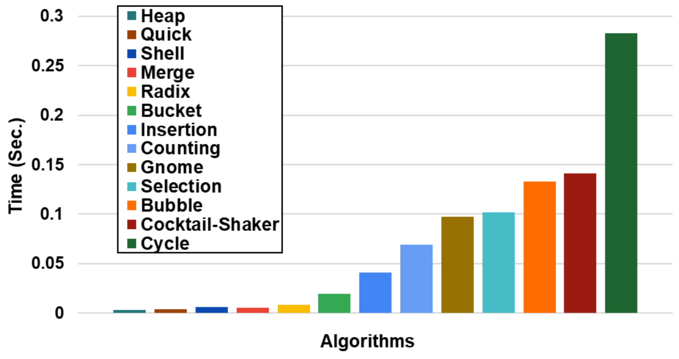

Fig. 3: Runtime for 10,000 Integers

C. *Performance on Large Dataset (100,000 Integers)*

In the largest dataset of 100,000 integers (Figure 4), the performance gap widens further. Quick sort and radix sort maintain highly efficient running times, demonstrating their robust performance and scalability. Merge sort also performs well, albeit slightly slower than quick sort and radix sort. Conversely, algorithms such as bubble sort, Cocktail-Shaker sort, and cycle sort exhibit exponentially longer running times. Bubble sort and gnome sort, in particular, perform poorly, with running times escalating dramatically, underscoring the inefficiency of these algorithms for large datasets.

TABLE II: Comparison metrics for sorting algorithms

Algorithm	Time Complexity	Space Complexity	Stability	Adaptability	Comparisons	Swaps/Moves
Insertion sort	$O(n)$ (best) $O(n^2)$ (avg, worst)	$O(1)$	Stable	Yes	$O(n)$ to $O(n^2)$	$O(n)$ to $O(n^2)$
Selection sort	$O(n^2)$ (best, avg, worst)	$O(1)$	Unstable	No	$O(n^2)$	$O(n)$
Bubble sort	$O(n)$ (best) $O(n^2)$ (avg, worst)	$O(1)$	Stable	Yes	$O(n)$ to $O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$ (best, avg, worst)	$O(n)$	Stable	No	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$ (avg, best) $O(n^2)$ (worst)	$O(\log n)$	Unstable	No	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$ (best, avg, worst)	$O(1)$	Unstable	No	$O(n \log n)$	$O(n \log n)$
Shell sort	$O(n \log^2 n)$ (avg) $O(n^2)$ (worst)	$O(1)$	Unstable	No	Depends	Depends
Counting sort	$O(n + k)$ (avg, best, worst)	$O(k)$	Stable	No	$O(n + k)$	$O(n + k)$
Radix sort	$O(nk)$ (avg, best, worst)	$O(n + k)$	Stable	No	$O(nk)$	$O(n + k)$
Bucket sort	$O(n + k)$ (avg) $O(n^2)$ (worst)	$O(n + k)$	Stable	No	$O(n^2)$	$O(n + k)$
Cocktail Shaker sort	$O(n)$ (best) $O(n^2)$ (avg, worst)	$O(1)$	Stable	Yes	$O(n)$ to $O(n^2)$	$O(n^2)$
Gnome sort	$O(n)$ (best) $O(n^2)$ (avg, worst)	$O(1)$	Stable	Yes	$O(n)$ to $O(n^2)$	$O(n^2)$
Cycle sort	$O(n^2)$ (best) $O(n^2)$ (avg, worst)	$O(1)$	Stable	No	$O(n^2)$	$O(n^2)$

TABLE III: Evaluation criteria for sorting algorithms

Algorithm	In-Place sorting	External sorting	Recursion	Parallelisability	Robustness	Implementation	Application
Insertion	Yes	No	No	Low	High	Very Easy	Small and Nearly sorted Data Sets
Selection	Yes	No	No	Low	High	Easy	Small and Unsorted Data Sets
Bubble	Yes	No	No	Low	Moderate	Very Easy	Educational Purposes and Small Data Sets
Merge	No	Yes	Yes	High	High	Moderate	General-Purpose sorting
Quick	Yes	No	Yes	Moderate	High	Moderate to High	General-Purpose sorting
Heap	Yes	No	No	Limited	High	Moderate	Priority Queue Operations
Shell	Yes	No	No	Limited	Moderate to High	Moderate	Medium-sized Arrays
Counting	No	No	No	Limited	High	Moderate	Small Range of Integer Values
Radix	No	No	Possible	Limited	High	Moderate	Fixed-Length Integer Keys
Bucket	No	No	Possible	High	High	Moderate to High	Uniformly Distributed Keys
Cocktail Shaker	Yes	No	No	Limited	Moderate	Moderate	Small to Moderate Size Arrays
Gnome	Yes	No	No	Limited	Moderate	Moderate	Small to Moderate-Sized Arrays
Cycle	Yes	No	No	Limited	Moderate	Moderate	Small to Moderate-Sized Arrays

TABLE IV: Running time of sorting algorithms (in seconds)

Algorithm	100 Int.	1000 Int.	10 000 Int.	100 000 Int.
Insertion	0.000116256	0.00287381	0.04084433	3.423113273
Merge	0.00013964	0.00103338	0.00514585	0.04304159
Radix	0.0001182	0.0011385	0.0083298	0.0348217
Bucket	0.0006421	0.0020384	0.0193157	0.0583855
Bubble	0.0001414	0.00551	0.1327085	16.1261891
Selection	0.0001436	0.0037447	0.1018653	10.0123196
Shell	0.0000453	0.0005504	0.0058414	0.0180842
Cock.-Sh.	0.0001899	0.0061125	0.1413062	16.01894543
Gnome	0.00012373	0.0049847	0.0971479	8.692618816
Cycle	0.0002617	0.0098478	0.2831037	26.76893115
Quick	0.00004589	0.000759	0.003409	0.0200578
Heap	0.00009591	0.000781	0.003348	0.0220375
Counting	0.0691072	0.0690697	0.06935005	0.08330846

Cock.-Sh.= Cocktail-Shaker, Int.= Integers

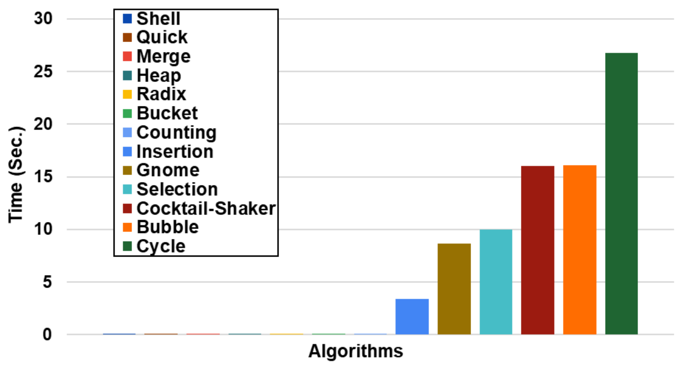

Fig. 4: Runtime for 100,000 Integers

D. Discussion on Algorithm Efficiency and Scalability

The results clearly illustrate the varying efficiency and scalability of different sorting algorithms.

- Quick sort: Demonstrates consistent efficiency across all dataset sizes, making it a versatile choice for both small and large datasets. Its performance is particularly notable on larger datasets, where it outperforms many other algorithms.
- Merge sort: Shows excellent performance, especially for larger datasets. Its divide-and-conquer approach ensures stable running times, making it suitable for datasets where stability is crucial.
- Radix sort: Exhibits high efficiency, particularly for large datasets. Its non-comparative nature allows it to handle large volumes of data swiftly.
- Heap sort and shell sort: Both perform well across various dataset sizes, with shell sort being particularly efficient on smaller datasets.
- Bubble sort, gnome sort, and cycle sort: These algorithms perform poorly on larger datasets. Their simplistic approaches lead to significantly longer running times as data

volume increases, making them unsuitable for large-scale applications.
- Counting sort: While its running time remains relatively stable, it is not efficient for larger datasets due to its high memory consumption and complexity.

These findings emphasise the need to choose the appropriate sorting algorithm based on dataset size and application requirements. Quick sort and radix sort excel with large datasets due to their efficiency and scalability, while simpler algorithms like bubble sort are better suited for educational purposes or very small datasets. In conclusion, the choice of sorting algorithm significantly impacts performance, especially with increasing data volumes. This study underscores the importance of considering algorithm efficiency and scalability to optimise sorting operations across different computational contexts.

V. Conclusion and Future Directions

This study analyses thirteen sorting algorithms, evaluating their performance across different dataset sizes to identify the most efficient techniques. Significant variations in time and space complexity were observed. Quick sort, merge sort, and radix sort consistently performed best for larger datasets due to their $O(n \log n)$ time complexities, while bubble sort, selection sort, and gnome sort were less effective due to their quadratic time complexities. The findings highlight the importance of selecting a sorting algorithm based on application requirements. For large datasets, quick sort and merge sort are preferable due to their lower time complexity. When memory usage is critical, algorithms with minimal space requirements, like heap sort, are advantageous. Specialised algorithms such as counting sort and radix sort offer notable performance benefits for specific data types and distributions. This research serves as a valuable guide for IT professionals and developers in choosing sorting algorithms based on empirical performance data.

Future research should explore additional sorting algorithms and their performance across various datasets. Key areas include integrating sorting with machine learning, analysing programming languages and optimisation impacts, and assessing adaptability to different data structures. Investigating scalability on parallel systems and applications in specialised fields like bioinformatics and finance could provide valuable insights, enhancing our understanding of sorting efficiencies and optimisation strategies.

References

[1] X. Bai and C. Coester, “Sorting with predictions,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[2] Y. Su, Z. Chen, L. Gong, X. Xu, and Y. Yao, “An improved adaptive radar signal sorting algorithm based on dbscan by a novel cvi,” *IEEE Access*, 2024.

[3] Z. Ahmad, A. I. Jehangiri, M. A. Ala’anzy, M. Othman, and A. I. Umar, “Fault-tolerant and data-intensive resource scheduling and management for scientific applications in cloud computing,” *Sensors*, vol. 21, no. 21, p. 7238, 2021.

[4] M. A. Ala’anzy and M. Othman, “Mapping and consolidation of vms using locust-inspired algorithms for green cloud computing,” *Neural Processing Letters*, vol. 54, no. 1, pp. 405–421, 2022.

[5] H. Safi, A. I. Jehangiri, Z. Ahmad, M. A. Ala’anzy, O. I. Alramli, and A. Algarni, “Design and evaluation of a low-power wide-area network (lpwan)-based emergency response system for individuals with special needs in smart buildings,” *Sensors*, vol. 24, no. 11, p. 3433, 2024.

[6] M. Ali, A. I. Jehangiri, O. I. Alramli, Z. Ahmad, R. M. Ghoniem, M. A. Ala’anzy, and R. Saleem, “Performance and scalability analysis of sdn-based large-scale wi-fi networks,” *Applied Sciences*, vol. 13, no. 7, p. 4170, 2023.

[7] M. Khan, A. I. Jehangiri, Z. Ahmad, M. A. Ala’anzy, and A. Umer, “An exploration to graphics processing unit spot price prediction,” *Cluster Computing*, vol. 25, no. 5, pp. 3499–3515, 2022.

[8] M. Alanzy, R. Latip, and A. Muhammed, “Range wise busy checking 2-way imbalanced algorithm for cloudlet allocation in cloud environment,” in *Journal of Physics: Conference Series*, vol. 1018, no. 1. IOP Publishing, 2018, p. 012018.

[9] M. A. Ala’anzy, M. Othman, S. Hasan, S. M. Ghaleb, and R. Latip, “Optimising cloud servers utilisation based on locust-inspired algorithm,” in *2020 7th International Conference on Soft Computing & Machine Intelligence (ISCMI)*. IEEE, 2020, pp. 23–27.

[10] I. M. Al-amin, A. Okeyinka, and A. Ibrahim, “Comparative complexity study of bubble sort and insertion sort using java programming language: A review,” *Journal of Science Innovation and Technology Research*, 2024.

[11] A. Rabiu, E. Garba, B. Baha, and M. Mukhtar, “Comparative analysis between selection sort and merge sort algorithms,” *Nigerian Journal of Basic and Applied Sciences*, vol. 29, no. 1, pp. 43–48, 2021.

[12] O. E. Taiwo, A. O. Christianah, A. N. Oluwatobi, K. A. Aderonke *et al.*, “Comparative study of two divide and conquer sorting algorithms: quicksort and mergesort,” *Procedia Computer Science*, vol. 171, pp. 2532–2540, 2020.

[13] A. H. Elkahlout and A. Y. Maghari, “A comparative study of sorting algorithms comb, cocktail and counting sorting,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 01, 2017.

[14] Y. Chauhan and A. Duggal, “Different sorting algorithms comparison based upon the time complexity,” *International Journal Of Research And Analytical Reviews*, no. 3, pp. 114–121, 2020.

[15] S. Sepahyar, R. Vaziri, and M. Rezaei, “Comparing four important sorting algorithms based on their time complexity,” in *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*, 2019, pp. 320–327.

[16] A. Zutshi and D. Goswami, “Systematic review and exploration of new avenues for sorting algorithm,” *International Journal of Information Management Data Insights*, vol. 1, no. 2, p. 100042, 2021.

[17] S. K. Gill, V. P. Singh, P. Sharma, and D. Kumar, “A comparative study of various sorting algorithms,” *International Journal of Advanced Studies of Scientific Research*, vol. 4, no. 1, 2019.

[18] A. Fenyi, M. Fosu, and B. Appiah, “Comparative analysis of comparison and non comparison based sorting algorithms,” *International Journal of Computer Applications*, 2020.

[19] R. Yadav, R. Yadav, and S. B. Gupta, “Comparative study of various stable and unstable sorting algorithms,” in *Artificial Intelligence and Speech Technology*. CRC Press, 2021, pp. 463–477.

[20] H. H. Aung, “Analysis and comparative of sorting algorithms,” *International Journal of Trend in Scientific Research and Development (IJTSRD)*, vol. 3, no. 5, pp. 1049–1053, 2019.

[21] A. S. Sabah, S. S. Abu-Naser, Y. E. Helles, R. F. Abdallatif, F. Y. A. Samra, A. H. A. Taha, N. M. Massa, and A. A. Hamouda, “Comparative analysis of the performance of popular sorting algorithms on datasets of different sizes and characteristics,” *International Journal of Academic Engineering Research (IJAER)*, 2023.

[22] F. A. Ardhiyani, E. Sudarmilah, and D. A. P. Putri, “An evaluation of quick sort and insertion sort algorithms for categorizing covid-19 cases in jakarta,” in *AIP Conference Proceedings*, vol. 2727, no. 1. AIP Publishing, 2023.

[23] P. E. Harris, J. Kretschmann, and J. C. Martínez Mori, “Lucky cars and the quicksort algorithm,” *The American Mathematical Monthly*, pp. 1–7, 2024.