# Radix Sort with digit buckets

**A Comprehensive Analysis of Implementation, Complexity, and Application**

## Abstract

This report provides a detailed examination of the Radix Sort algorithm, a non-comparative integer sorting technique. It delves into the fundamental mechanics of Radix Sort, which operates by grouping keys by individual digits sharing the same significant position and value. The report covers the algorithm's pseudocode, a visual flowchart, and a step-by-step dry run to illustrate its operational flow. A rigorous asymptotic analysis of its time and space complexity is presented, followed by complete implementation examples in both C++ and Python. Furthermore, the report explores the primary application areas where Radix Sort is most effective and provides a comparative study against traditional sorting algorithms like Quick Sort, Merge Sort, and others. This document serves as a comprehensive academic resource for understanding the theoretical underpinnings and practical utility of Radix Sort.

## 1. Introduction

Sorting is one of the most fundamental and extensively studied problems in computer science. It involves arranging a collection of items into a specific order. Sorting algorithms are broadly categorized into two main groups: comparison-based and non-comparison-based. Comparison-based algorithms, such as Quick Sort and Merge Sort, determine the sorted order by comparing elements with one another. The theoretical lower bound for the time complexity of such algorithms is O(nlogn).

In contrast, non-comparison-based algorithms leverage the internal structure of the data to sort elements without direct comparisons. Radix Sort is a prime example of such an algorithm. It is an integer sorting algorithm that sorts data with integer keys by grouping them by the individual digits which share the same significant position and value. By processing digits from the least significant to the most significant, Radix Sort can achieve linear time complexity under certain conditions, thereby overcoming the O(nlogn) barrier of comparison sorts.

This report provides a comprehensive technical analysis of the Radix Sort algorithm. It begins by detailing its operational mechanics through pseudocode and a step-by-step dry run. A rigorous asymptotic analysis of its time and space complexity is conducted, followed by practical implementations in C++ and Python. The report also explores key application areas and presents a comparative study against other well-known sorting algorithms, concluding with a summary of its strengths and optimal use cases.

### 2. Algorithm Pseudocode

Radix Sort operates by sorting an array of integers one digit at a time, from the least significant to the most significant. It uses a distribution-based approach, placing elements into buckets corresponding to the value of the current significant digit. The process is repeated for all digit positions, ensuring stability at each step.

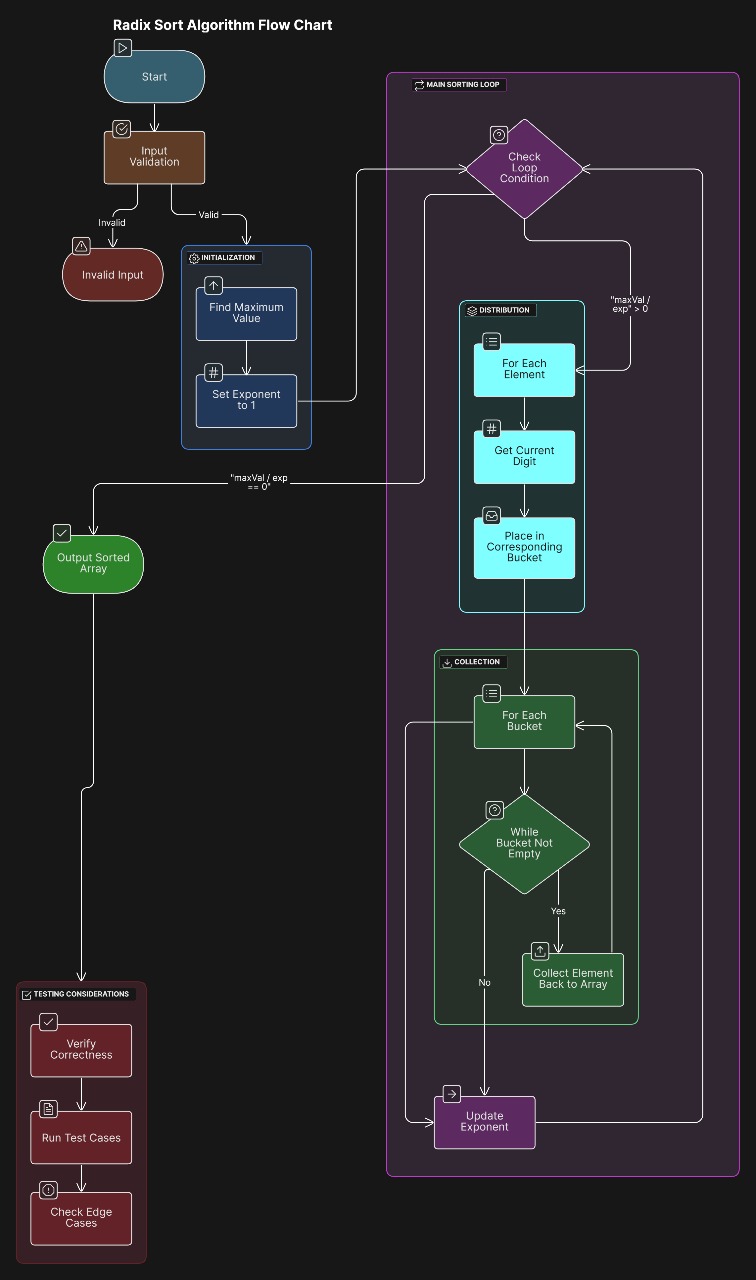
**Algorithm: Radix Sort with Digit Buckets**

|  |
| --- |
| 01. **procedure** RadixSort(A) 02. Input: An **array** A **of** n integers. 03. Output: **Array** A sorted **in** non-decreasing **order**. 04.  05. d ← Find maximum number **of** digits **in** any number **in** **array A** 06.  07. **for** i ← 1 **to** d **do** *// Iterate from the least significant*  08. *digit to the most significant* 09. *// Create 10 empty buckets (one for each digit 0-9)* 10. buckets ← an **array** **of** 10 **empty** lists 11.  12. *// Distribute elements of A into buckets based on the*  13. *i-th digit* 14. **for** **each** number x **in** A **do** 15. digit ← Get the i-th digit **of** x (**from** the right) 16. append x **to** buckets[digit] 17. **end** **for** 18.  19. *// Collect elements from buckets back into A in order* 20. A ← an **empty** list 21. **for** j ← 0 **to** 9 **do** 22. append all elements **of** buckets[j] **to** the **end** **of** A 23. **end** **for** 24. **end** **for** 25.  26. return A |

This procedure ensures that the numbers are correctly sorted by considering each digit's place value sequentially. The stability of the bucketing process (maintaining the relative order of elements with the same digit in a particular pass) is crucial for the algorithm's correctness.

### 3. Flowchart / Block Diagram

The following flowchart illustrates the logical flow of the Radix Sort algorithm. It visualizes the main loop, the distribution of elements into buckets based on the current digit, and the collection of elements back into the array.



### 4. 10-Value Sample Dry Run

Let's trace the Radix Sort algorithm with the following 10-value sample array:

[170, 45, 75, 90, 802, 24, 2, 66, 18, 333]

The maximum value is 802, which has 3 digits. Therefore, the sorting process will have 3 passes.

**Pass 1: Sorting by the Least Significant Digit (1s place)**

* **Input:** [170, 45, 75, 90, 802, 24, 2, 66, 18, 333]
* **Buckets:**
  + Bucket 0: [170, 90]
  + Bucket 2: [802, 2]
  + Bucket 3: [333]
  + Bucket 4: [24]
  + Bucket 5: [45, 75]
  + Bucket 6: [66]
  + Bucket 8: [18]
* **Output of Pass 1:** [170, 90, 802, 2, 333, 24, 45, 75, 66, 18]

**Pass 2: Sorting by the Tens Digit (10s place)**

* **Input:** [170, 90, 802, 2, 333, 24, 45, 75, 66, 18]
* **Buckets:**
  + Bucket 0: [802, 2]
  + Bucket 1: [18]
  + Bucket 2: [24]
  + Bucket 3: [333]
  + Bucket 4: [45]
  + Bucket 6: [66]
  + Bucket 7: [170, 75]
  + Bucket 9: [90]
* **Output of Pass 2:** [802, 2, 18, 24, 333, 45, 66, 170, 75, 90]

**Pass 3: Sorting by the Most Significant Digit (100s place)**

* **Input:** [802, 2, 18, 24, 333, 45, 66, 170, 75, 90]
* **Buckets:**
  + Bucket 0: [2, 18, 24, 45, 66, 75, 90]
  + Bucket 1: [170]
  + Bucket 3: [333]
  + Bucket 8: [802]
* **Output of Pass 3 (Final Sorted Array):** [2, 18, 24, 45, 66, 75, 90, 170, 333, 802]

### 5. Asymptotic Analysis

The performance of Radix Sort is analyzed based on its time and space requirements. Let n be the number of elements in the input array and d be the number of digits in the largest number in the array.

#### Time Complexity

The time complexity is consistent across best, average and worst-case scenarios because the algorithm's operations are not dependent on the initial order of the elements, only on their values and number of digits. The calculation is derived as follows:

|  |  |  |
| --- | --- | --- |
| **Line No.** | **Code Description** | **Complexity** |
| 4 | d ← max\_digits(A) | O(n) – traverse A once to find the max and count its digits. |
| 5 | for i ← 1 to d do | d iterations (outer loop). |
| 6 | buckets ← initialize\_buckets() | O(1) – 10 buckets, fixed number. |
| 7 | for x ∈ A do | O(n) – iterate over all n elements (inner loop). |
| 8 | digit ← digit\_at\_position(x, i) | O(1) – constant time per element to extract the i-th digit. |
| 9 | insert x into buckets[digit] | O(1) – insertion into list. |
| 11 | A ← empty\_list() | O(1) – reset A to empty. |
| 12 | for j ← 0 to 9 do | O(1) – 10 iterations (bucket loop). |
| 13 | for x ∈ buckets[j] do | O(n) in total over all buckets. |
| 14 | append x to A | O(1) per append, O(n) total over all buckets. |

Using thumb’s rule:

T(n) = n + 1 + n + 1 + 1 + 1 + 1 + n + n

= 4n + 5

= 4n = O(n) this is for one pass

If we have d passes, then it will be O(n) \* d = O(n\*d).

|  |  |  |
| --- | --- | --- |
| **Type** | **Complexity** | **Reason** |
| **Best** (Ω) | Ω(n·d) | Must process each digit of each number |
| **Average** (Θ) | Θ(n·d) | Always scans all elements for all digits |
| **Worst** (O) | O(n·d) | All elements with d digits |

#### Space Complexity

The space complexity is determined by the auxiliary storage used by the algorithm.

1. **Buckets:** The primary storage requirement comes from the buckets used to distribute the elements.
2. In the worst-case scenario, all n elements could fall into a single bucket. Therefore, the total space required for the buckets is the space to hold n elements plus the overhead for the b bucket structures themselves. This results in a space complexity of O(n+b).

### 6. C++ Implementation

|  |
| --- |
| #include <iostream> #include <vector> #include <algorithm> #include <queue>  // Function to get the maximum value in arr[] int **getMax**(const std::vector<int>& arr) {  int max = arr[0];  for (size\_t i = 1; i < arr.size(); i++) {  if (arr[i] > max) {  max = arr[i];  }  }  return max; }  // A function to do counting sort of arr[] according to the digit represented by exp. void **countSort**(std::vector<int>& arr, int exp) {  std::vector<int> output(arr.size());  int i;  std::vector<int> count(10, 0);   // Store count of occurrences in count[]  for (i = 0; i < arr.size(); i++) {  count[(arr[i] / exp) % 10]++;  }   // Change count[i] so that count[i] now contains actual  // position of this digit in output[]  for (i = 1; i < 10; i++) {  count[i] += count[i - 1];  }   // Build the output array  for (i = arr.size() - 1; i >= 0; i--) {  output[count[(arr[i] / exp) % 10] - 1] = arr[i];  count[(arr[i] / exp) % 10]--;  }   // Copy the output array to arr[], so that arr[] now  // contains sorted numbers according to current digit  for (i = 0; i < arr.size(); i++) {  arr[i] = output[i];  } }   // The main function to that sorts arr[] of size n using Radix Sort void **radixSort**(std::vector<int>& arr) {  // Find the maximum number to know number of digits  if (arr.empty()) {  return;  }  int m = getMax(arr);   // Do counting sort for every digit. Note that instead  // of passing digit number, exp is passed. exp is 10^i  // where i is current digit number  for (int exp = 1; m / exp > 0; exp \*= 10) {  countSort(arr, exp);  } }  // A utility function to print an array void **printArray**(const std::vector<int>& arr) {  for (int i : arr) {  std::cout << i << " ";  }  std::cout << std::endl; }  // Driver code int **main**() {  std::vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66, 18, 333};  std::cout << "Original array: ";  printArray(arr);   radixSort(arr);   std::cout << "Sorted array: ";  printArray(arr);  return 0; }  **Original array**: 170 45 75 90 802 24 2 66 18 333 **Sorted array**: 2 18 24 45 66 75 90 170 333 802 |

### 7. Python Implementation

|  |
| --- |
| # Python program for implementation of Radix Sort  def **counting\_sort**(arr, exp):  """  A stable sort for a specific digit represented by exp.  """  n = len(arr)  output = [0] \* n  count = [0] \* 10   # Store count of occurrences in count[]  for i in range(n):  index = (arr[i] // exp)  count[index % 10] += 1   # Change count[i] so that count[i] now contains the actual  # position of this digit in the output array  for i in range(1, 10):  count[i] += count[i - 1]   # Build the output array  i = n - 1  while i >= 0:  index = (arr[i] // exp)  output[count[index % 10] - 1] = arr[i]  count[index % 10] -= 1  i -= 1   # Copy the output array to arr[], so that arr now  # contains sorted numbers  for i in range(n):  arr[i] = output[i]  def **radix\_sort**(arr):  """  Main function to implement Radix Sort.  """  if not arr:  return   # Find the maximum number to know the number of digits  max1 = max(arr)   # Do counting sort for every digit. Note that instead of passing digit  # number, exp is passed. exp is 10^i where i is the current digit number  exp = 1  while max1 // exp > 0:  counting\_sort(arr, exp)  exp \*= 10  # Driver code to test above if \_\_name\_\_ == "\_\_main\_\_":  data = [170, 45, 75, 90, 802, 24, 2, 66, 18, 333]  print(f"Original array: {data}")  radix\_sort(data)  print(f"Sorted array: {data}")  **Original array**: [170, 45, 75, 90, 802, 24, 2, 66, 18, 333]  **Sorted array**: [2, 18, 24, 45, 66, 75, 90, 170, 333, 802] |

### 8. Application Areas

Radix Sort is particularly efficient under specific conditions and finds use in various domains:

* **Sorting large integer keys:** It excels at sorting large sets of integers, especially when the number of digits is relatively small.
* **Suffix array construction:** Used in bioinformatics and string processing algorithms for constructing suffix arrays.
* **Parallel Computing:** The algorithm can be parallelized effectively, as each digit-based sort can be handled independently to some extent.
* **Fixed-size key sorting:** Ideal for sorting data where keys have a fixed length, such as sorting employee IDs or other fixed-format identifiers.

### 9. Comparative Study

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best Case Time** | **Average Case Time** | **Worst Case Time** | **Space Complexity** | **Stable** |
| **Radix Sort** | O(d(n+b)) | O(d(n+b)) | O(d(n+b)) | O(n+b) | Yes |
| **Quick Sort** | O(nlogn) | O(nlogn) | O(n2) | O(logn) | No |
| **Merge Sort** | O(nlogn) | O(nlogn) | O(nlogn) | O(n) | Yes |
| **Bubble Sort** | O(n) | O(n2) | O(n2) | O(1) | Yes |
| **Insertion Sort** | O(n) | O(n2) | O(n2) | O(1) | Yes |
| **Selection Sort** | O(n2) | O(n2) | O(n2) | O(1) | No |

Radix Sort stands out as a non-comparative algorithm with a linearithmic complexity that can outperform comparison-based sorts like Quick Sort and Merge Sort when the key length (d) is small, and the keys are uniformly distributed.

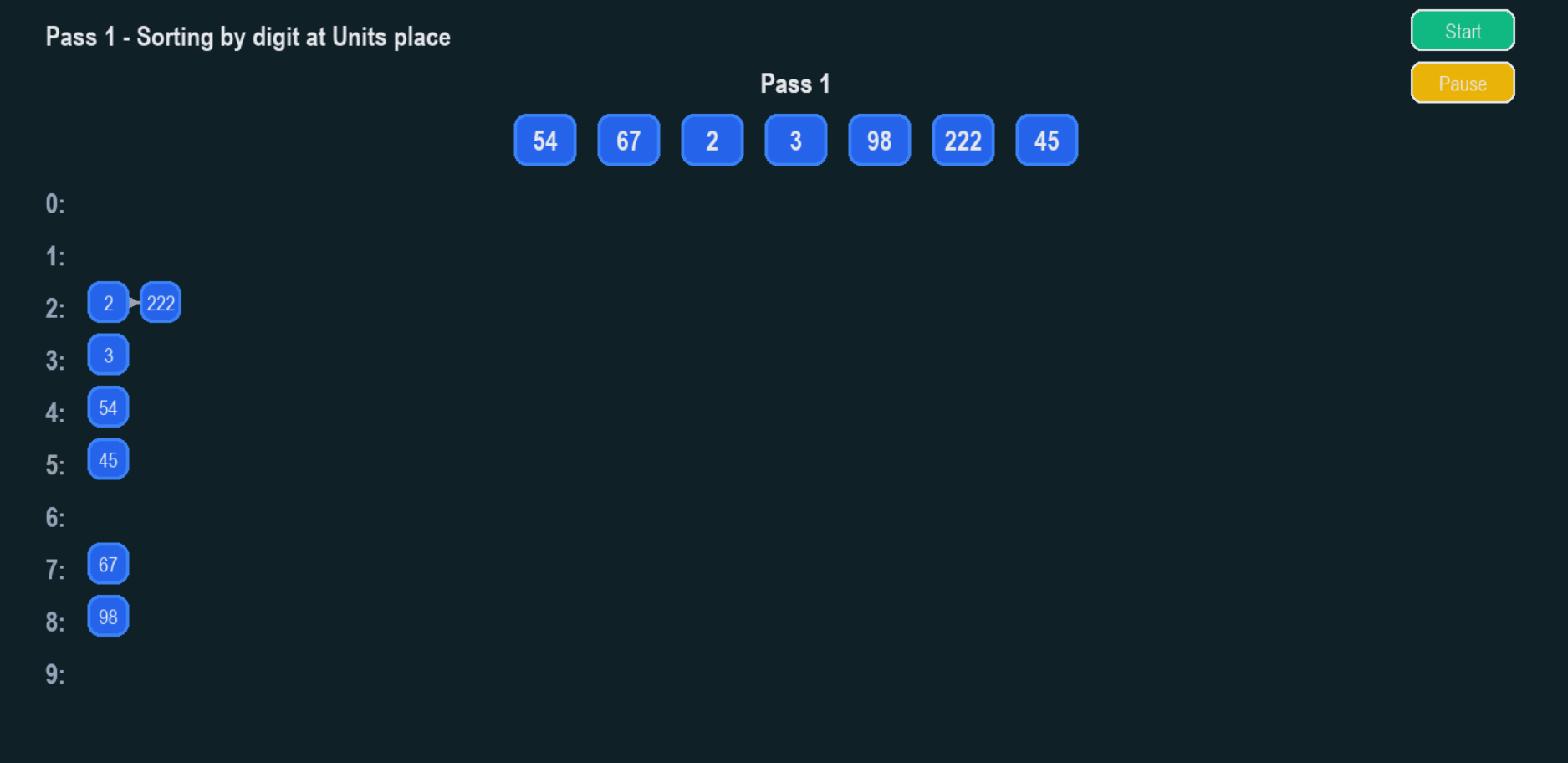
### 10. Execution Snapshots

Below are representative snapshots of simulation results.



A screenshot of a computer

AI-generated content may be incorrect.

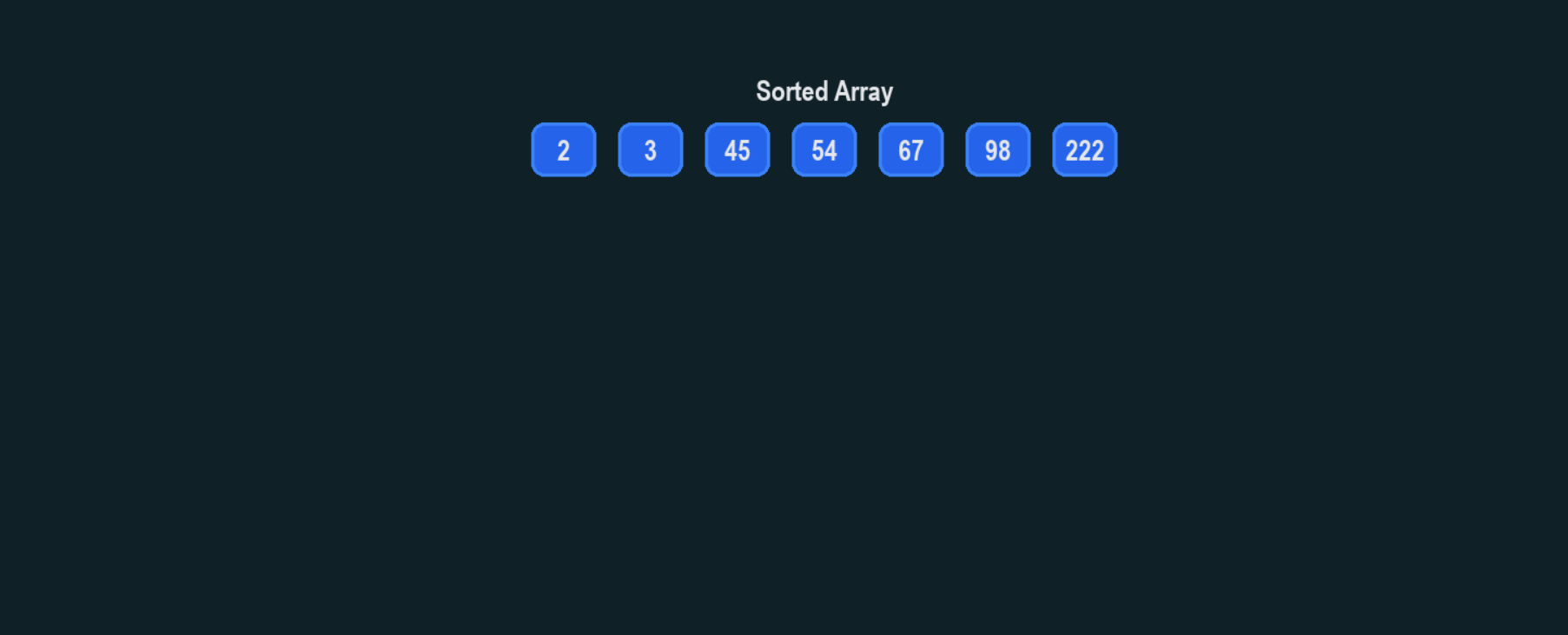


A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



### 11. Conclusion

Radix Sort is a powerful and efficient sorting algorithm for specific use cases, particularly with integer data or fixed-length strings. Its non-comparative nature allows it to achieve linear time complexity under favorable conditions, making it faster than comparison-based sorting algorithms for certain datasets. This report has provided a thorough analysis of its mechanics, complexity, and practical implementations, establishing its place as a valuable tool in a programmer's and computer scientist's toolkit.

### 12. References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley Professional.
3. GeeksforGeeks. "Radix Sort." Retrieved from <https://www.geeksforgeeks.org/radix-sort/>
4. Wikipedia contributors. "Radix Sort." *Wikipedia, The Free Encyclopedia*. Retrieved from <https://en.wikipedia.org/wiki/Radix_sort>