Radix Sort

Varun Kumar

July 5, 2025

01. Logic

Radix Sort is a non-comparative sorting algorithm that sorts integers by processing individual digits. It processes digits from least significant digit (LSD) to most significant digit (MSD), or vice versa.

Key Idea

Sort the numbers digit by digit using a stable sorting algorithm (like Counting Sort), starting from the least significant digit.

02. Number of Passes and Behavior

Let n be the number of elements, and d be the number of digits in the maximum number.

- The number of passes $= d$
- Each pass uses a stable sort (usually Counting Sort)

03. Optimal Behavior

Radix Sort is efficient when the range of digits (d) is small and the number of elements (n) is large. It avoids comparisons and is ideal for sorting integers and fixed-length strings.

04. Pseudocode

```
function radixSort(arr):
    max_digit = number of digits in the largest number
    for digit_pos in range(0, max_digit):
        stable_sort(arr, digit_pos)
```

Stability Note

Radix Sort is stable if the sorting algorithm used in each digit pass is stable (e.g., Counting Sort).

05. Example Walkthrough

Given: `[170, 45, 75, 90, 802, 24, 2, 66]`

Pass 1 (LSD - unit digit)

`[170, 90, 802, 2, 24, 45, 75, 66]`

Pass 2 (tens digit)

`[802, 2, 24, 45, 66, 170, 75, 90]`

Pass 3 (hundreds digit)

`[2, 24, 45, 66, 75, 90, 170, 802]`

06. Python Code with Explanation

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    # Count digits
    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    # Accumulate counts
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Build output
    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1

    # Copy to arr
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort(arr, exp)
        exp *= 10
```

07. Why is Radix Sort Stable?

Radix Sort processes digits one place at a time starting from the least significant digit (LSD), and it uses a **stable sorting algorithm** (like Counting Sort) in each pass. Stability in these intermediate passes ensures that:

- Elements with the same digit maintain their original relative order.
- When a more significant digit is sorted later, it doesn't disturb the previous ordering.

Hence, by using a stable sort at each digit level, the overall Radix Sort becomes stable.

08. Is Radix Sort In-Place?

Radix Sort is **not in-place** because it requires additional memory for:

- Temporary output array (same size as input) to store the sorted elements after each digit pass.
- Counting array (usually of size 10 for base-10 digits) used in Counting Sort.

  Thus, its space complexity is $O(n + k)$, where:

  - n = number of elements
  - k = range of digits (usually 10 for decimal numbers)

09. Can Radix Sort Be Made In-Place?

In general, **Radix Sort is not in-place** due to its reliance on an auxiliary output array to maintain stability across digit-wise passes.

Why not in-place?

- Requires $O(n)$ extra space for output in each pass.
- Avoiding extra space would break the **stability** or **increase complexity**.

Some theoretical attempts exist to make Radix Sort in-place, but they are not practical due to complexity, instability, or loss of linear-time performance.

10. Why can't Radix Sort be made in-place without breaking stability or increasing complexity?

Answer: Radix Sort works by sorting digits one at a time (typically starting from the least significant digit), and in each digit pass, it uses a **stable sort** like Counting Sort to preserve the relative order of elements with equal digits.

To maintain **stability**, we must:

- Use an auxiliary output array to store the sorted result.
- Copy back this output into the original array after each pass.

Removing the auxiliary array (to make it in-place) would:

1. **Break Stability:** Direct in-place swapping can rearrange equal elements, violating stability.
2. **Increase Complexity:** Advanced in-place stable sorting techniques exist, but they require extra logic (like complex cycle-leader transformations or linked-index simulation), which leads to:
    - More comparisons and swaps
    - Harder implementation
    - Loss of linear-time advantage

Conclusion: While in theory, it is possible to design an in-place version of Radix Sort, it is *not practical* because:

- It would no longer run in linear time.
- It may no longer remain stable.
- It would require complex and error-prone logic.

Hence, **standard Radix Sort is not in-place**, and efforts to make it in-place come at the cost of either **breaking stability** or **losing linear time efficiency**.

11. Best Case and Worst Case Using a Single Example

Let us consider the array:

`[170, 45, 75, 90, 802, 24, 2, 66]`

We will use this same example to illustrate both the best-case and worst-case scenarios for Radix Sort.

Best Case Scenario

- All elements are uniformly distributed over digit positions.
- Digit range is small (e.g., base 10), and all numbers have nearly the same number of digits.
- **No significant digit collisions** at each place value, so Counting Sort per digit is efficient.
- Array size $n = 8$, maximum digits $d = 3$ (e.g., 802 has 3 digits).

Best Case Time Complexity

$$T(n) = O(d \cdot (n + k)) = O(3 \cdot (8 + 10)) = O(1) \cdot O(n) = O(n)$$

Where:

- d = number of digits (3)
- k = base/range of digits (10)

Worst Case Scenario

- Numbers have large variation in digit lengths (e.g., mixing 3-digit and 10-digit numbers).
- Very large base k or inefficient implementation of Counting Sort.
- Overhead in each digit pass increases due to long digit ranges.
- Still better than $O(n^2)$ algorithms, but constants can degrade performance.

Worst Case Time Complexity

$$T(n) = O(d \cdot (n + k)) \quad \text{where } d \text{ is large, e.g., 10 or more digits.}$$

Even with same 8 elements, if max number = 9876543210, then $d = 10$:

$$T(n) = O(10 \cdot (8 + 10)) = O(180)$$

Summary

Scenario	Condition	Time Complexity
Best Case	Uniform digit length, small d	$O(n)$
Worst Case	Large d, large base k	$O(d \cdot (n+k))$

12. Time & Space Complexity and Its Properties

Case	Complexity	Property	Value
Best Case	$O(n \cdot k)$	Stable	Yes
Average Case	$O(n \cdot k)$	In-place	No
Worst Case	$O(n \cdot k)$	Adaptive	No
Space Complexity	$O(n+k)$	Recursive	No

Note:

- Actually $O(k \cdot (n+b)) \approx O(n \cdot k)$
  - Where n = number of elements, k = number of digits and b = base
- Radix Sort is faster than comparison-based $O(n \log n)$ sorts when k is small and digits are bounded.
- Not adaptive: it does the same number of passes regardless of initial order.
- Not in-place: requires auxiliary arrays (e.g., output array in Counting Sort).

14. C++ Implementation of Radix Sort (LSD)

Code with Explanation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Get the maximum element in the array
int getMax(const vector<int>& arr) {
    return *max_element(arr.begin(), arr.end());
}

// Counting Sort used for a specific digit (exp = 1, 10, 100, ...)
void countingSort(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);       // Output array to store sorted values
    int count[10] = {0};         // Count array for digits 0-9

    // Count occurrences of digits at 'exp' place
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Convert count[] to actual positions
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build output[] using reverse traversal for stability
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy back to original array
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

// Main Radix Sort function
void radixSort(vector<int>& arr) {
    int maxVal = getMax(arr);
```

```

    // Sort each digit (unit, tens, hundreds, etc.)
    for (int exp = 1; maxVal / exp > 0; exp *= 10)
        countingSort(arr, exp);
}

// Utility function to print array
void printArray(const vector<int>& arr) {
    for (int num : arr)
        cout << num << " ";
    cout << endl;
}

// Driver code
int main() {
    vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66};

    cout << "Original array:\n";
    printArray(arr);

    radixSort(arr);

    cout << "Sorted array:\n";
    printArray(arr);

    return 0;
}
```

Listing 1: Radix Sort in C++

Explanation of the Code

- `getMax()` finds the largest number to determine the number of digit passes.
- `countingSort()` is called for each digit position (units, tens, hundreds, etc.) using `exp` as the digit place.
- Counting Sort is stable because we fill the `output[]` array from the end of the original array.
- `radixSort()` loops over digit places until all digits of the largest number are processed.
- `printArray()` is a helper to display the array before and after sorting.

Properties of This Implementation

- **Stable:** Yes, due to reverse traversal in Counting Sort.
- **Not In-Place:** Uses extra memory for the output array.
- **Time Complexity:** $O(n \cdot d)$ where d is the number of digits in the max element.
- **Space Complexity:** $O(n + k)$ where $k = 10$ (digits from 0 to 9).
- **Limitation:** Only supports non-negative integers in this version.

15. Handling Negative Numbers in Radix Sort

Problem

Standard Radix Sort (especially LSD variant) works only on non-negative integers because it processes digits. Negative numbers contain a sign, and digit-wise processing is undefined for negative values.

Solution Strategy

To extend Radix Sort to handle both positive and negative integers:

1. **Separate the array into:**
   - Negative numbers
   - Non-negative numbers
2. **Take absolute values** of negative numbers before sorting.
3. **Apply Radix Sort:**
   - On positive numbers as-is
   - On absolute values of negative numbers
4. **Reverse the sorted negative list** (because more negative values come first).

5. **Convert back to negative** by multiplying with -1.
6. **Concatenate:**
   - Final array = Sorted negatives (in reverse) + sorted positives

Example

$$\text{Input: } [-170, 45, -75, 90, -802, 24, 2, -66]$$

- Negatives: `[-170, -75, -802, -66]`
- Positives: `[45, 90, 24, 2]`
- Absolute Negatives: `[170, 75, 802, 66]`

Apply Radix Sort:

- Sorted Positives: `[2, 24, 45, 90]`
- Sorted Absolute Negatives: `[66, 75, 170, 802]`

Reverse and restore negatives:

$$\text{Sorted Negatives: } [-802, -170, -75, -66]$$

$$\textbf{Final Output: } [-802, -170, -75, -66, 2, 24, 45, 90]$$

Conclusion

This method:

- Preserves Radix Sort's efficiency: $O(n \cdot d)$
- Requires temporary arrays for splitting and merging
- Ensures correct sorting of negative values without modifying the core algorithm