**NON-COMPARISON SORTING**

# Introduction

## Overview

While algorithms like MergeSort and QuickSort are great for general-purpose sorting, they are limited by the comparison model, meaning they need to compare elements to determine their order.

However, there are certain scenarios where we can beat O(n log n) by avoiding comparisons altogether. These are called non-comparison sorting algorithms.

## Key Idea

Non-comparison sorting algorithms, such as Counting Sort and Radix Sort, take advantage of the internal representation of data in computers (in binary format) instead of comparing elements directly.

This leads to more efficient algorithms when sorting certain types of data, specifically integers within a limited range.

## Why Can’t We Beat O(n log n)?

For general comparison-based algorithms, it is mathematically proven that the best performance we can achieve is O(n log n).

The reason for this is that we must perform comparisons between all elements in order to determine their relative order. However, non-comparison sorts bypass this limitation by exploiting the internal structure of the data.

## When Non-Comparison Sorting Works Best

These algorithms are most effective when:

* Sorting integers or fixed-length data.
* The range of the data is small (e.g., sorting integers between 0 and 100).
* We don't need to compare the elements directly.

# Couting sort

## How It Works

- Counting Sort counts the number of occurrences of each value in the input data and uses this count to determine the position of each element in the sorted output.

- It is highly efficient when the range of numbers (k) is small relative to the number of elements (n).

## Time Complexity:

- O(n + k) Where n is the number of elements and k is the range of input values.

- This is much faster than comparison-based algorithms when k is small.

## Use Case:

- Sorting integers within a fixed range. Example: Sorting a list of exam scores (ranging from 0 to 100).

# Radix Sort

## How It Works

- Radix Sort works by sorting the numbers digit by digit, starting from the least significant digit (LSD) to the most significant (MSD) or vice versa.

- It uses another stable sorting algorithm, like Counting Sort, as a subroutine to sort based on each digit.

- The process is repeated for each digit until the entire list is sorted.

## Time Complexity

- O(n \* k) Where n is the number of elements and k is the number of digits in the largest number.

- This makes it efficient for sorting fixed-length data like integers.

## Use Case

Sorting large sets of numbers, especially when the number of digits is small or the range is known. Example: Sorting phone numbers or dates.

# Pros and cons of Non-comparison sorting algorithms

|  |  |
| --- | --- |
| Pros | Cons |
| Faster for specific data types: Can outperform O(n log n) algorithms for integers within a limited range. | Limited to specific data types: Only works with integers or fixed-length data. |
| Efficient with small range data: Works well when the range of numbers (k) is small. | Memory usage: Can require extra space proportional to the range of data (k). |
| Better time complexity for certain cases: Time complexity of O(n + k) or O(n \* k) in some cases. | Not universal: Does not work for all types of data (e.g., strings, floats). |
| Stable sorting: Some algorithms like Counting Sort maintain the relative order of equal elements. | Not suitable for large ranges: Becomes inefficient when the range (k) is large. |

A screenshot of a computer screen

Description automatically generated

Slide 1: Array sorting algorithms