Kabul University
Computer Science Faculty
Software Engineering Department

Tim Sort

< A Hybrid Sorting Algorithm >

Presented by: Abdullah “Nooristani”
Date of Presentation: 2025/5/24

Table of contents

1. Introduction

- Why are Sorting Algorithms Important?
- Overview of Common Sorting Algorithms
- History and Inventor
- Introduction to Tim Sort

2. Technical Analysis

- Hybrid Nature of Tim Sort
- Runs and Minimize Run Size
- Sorting Runs with Insertion Sort
- Merging Runs
- Time & Space Complexity

3. Application & Summary

- Usage in Python & Java, Android
- Real World Applications
- Summary & Q&A

01 { . .

Introduction

- Why are Sorting Algorithms Important?
- Overview of Common Sorting Algorithms
- History and Inventor
- Introduction to Tim Sort

} . .

Why are Sorting Algorithms Important?

- Faster Searching: Binary Search requires sorted data
- Easier data analysis and reporting
- Better user Experience (e.g., sorted contact list)
- Essential in databases, search engines, and apps

Real World Examples

- Google search results sorted by relevance
- E-commerce sites sorting products by price or popularity
- School report cards sorted by grades

Overview of Common Sorting Algorithms

Bubble Sort: Simple but slow, compares adjacent items repeatedly. Best only for educational purposes.

Insertion Sort: Efficient for small or nearly sorted data, builds the sorted array one element at all time.

Merge Sort: Divide and conquer algorithm, Split arrays into halves, sorts each recursively, then merges.

Quick Sort: Also Divide and Conquer , Chooses a pivot, partitions array into two subarrays. Not stable.

Selection Sort: Repeatedly finds the minimum element and moves it to the beginning , not stable, performs well on small data sets.

Counting Sort: Non-comparison-based sort, works for integers in a fixed range, counts occurrences of each value and reconstructs sorted array, extremely fast for small range data.

Radix Sort: Non-comparison sort that processes numbers digit by digit, uses Counting sort internally at each digit level.

Very Efficient for integers and strings with uniform length, stable.

Comparison Analysis of Sorting Algorithms

When it comes to sorting algorithms, none of them are perfect. Take a look at the following table of running times for the sorting algorithms in the next slide:

Name	Average Time	Best Time	Worst Time
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Couting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix Sort	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$

Continue….

These sorting algorithms that seem perfect often show really poor behavior in worst-case scenarios. Their best-case performance may be suboptimal as well. Take quicksort for example. Quicksort looks pretty good, right? On average it runs at a very respectable **O(n log n)** time. Now, what if we ask it to sort a fully sorted collection of values, and our pivot selection is having an off day? In this case, quicksort will slow to a crawl and sort our input with a **$O(n^2)$** time. That makes it as bad as some of our slowest sorting algorithms.

Continue...

The opposite holds true as well. Our slowest sorting algorithms, like Selection Sort, Insertion Sort, or Bubble Sort have the potential to run really fast. For example, Bubble Sort normally runs at **$O(n^2)$**. If the values we ask it to sort happen to already be sorted, Bubble Sort will sort the values at a blazing fast **$O(n)$** time. That's even faster than Quicksort's best sorting time:

Anything could happen

Slow sorting algorithms could run really fast. Fast sorting algorithms could run really slow.

What if we could avoid this speculation and choose the right sorting algorithm based on what exactly our unsorted data is doing?

Continue…

How well our sorting algorithms work depends largely on the arrangement of our unsorted input values:

1. Are the input values random?
2. Are they already sorted?
3. Are they sorted in reverse?
4. Is the range of values inside them large or small?

Meet Hybrid Sorting Algorithms

Instead of using a single sorting algorithm for our data, we can choose from a variety of sorting algorithms optimized for the kind of data we are dealing with. We can use something known as a **hybrid sorting algorithm**. A hybrid sorting algorithm takes advantage of the strengths of multiple sorting algorithms to create some sort of a super algorithm

By relying on multiple sorting algorithms, we can **minimize the worst-case behavior of individual sorting algorithms** by switching between algorithms based on the characteristics of the unsorted input data being sorted.

In this Presentation, we are going to learn about one of the greatest hybrid sorting algorithms ever created. We are going to learn about **Tim sort**

Introduction to Tim Sort

- Tim sort is a hybrid sorting algorithm that combines the best features of two other sorting algorithms:
    - Insertion Sort for small data pieces.
    - Merge Sort for merging larger sorted pieces.
- Designed to be adaptive to real-world data, which is often partially sorted
- Developed by Tim Peters in 2002.
- Used in Python's built-in sort, Java's Array.sort(), and Android.

History and Inventor of Tim Sort

- Created by Tim Peters, a Python Developer
- Officially introduced in Python 2.3 (2002)
- Designed to optimized sorting for real-world data, exploiting existing order.
- Quickly adopted by other programming languages due to efficiency and stability.
- Named “Tim Sort” after its inventor.

02 { . .

Technical Analysis

- Hybrid Nature of Tim Sort
- The Way Time sort Works
- Sorting Runs with Insertion Sort
- Merging Runs
- Time & Space Complexity

} . .

The Hybrid Nature of Tim Sort

- Tim Sort is a hybrid algorithm because it combines two strategies:
  - Insertion Sort for small data pieces, nearly chunks of data
  - Merge Sort efficient for merging larger sorted chunks.

How the Combination Works?

1. The array is divided into small "runs" (naturally ordered sequences).
2. Insertion Sort is applied to each run individually (because it's fast on small data)
3. The sorted runs are then merged using a modified merge process.

What makes it Unique?

- Doesn't treat the array as totally unsorted.
- Tries to detect already sorted regions first.
- Uses optimal tools for each stage:
  - Small Size → Insertion Sort
  - Large merge → Merge Sort

Advantages of Hybrid Approach

- Adaptive: Uses less time when data is already partially sorted.
- Efficient: Outperforms Quick Sort and Merge Sort in many real-world cases.
- Stable: Maintains the order of equal elements.
- Practical: Combines real-world performance with theoretical robustness.

Sorting Runs with Insertion Sort

Why Insertion Sort for Runs?

Very fast for small-sized or nearly sorted data

Tim Sort uses insertion sort to sort each detected run

Helps make each run completely sorted before merging begins

Example:

Sort: [4, 2, 6]

Compare 2 to 4 → shift 4

Insert 2 before 4

Result: [2, 4, 6]

How Insertion Sort Works(Quick Recap)

1. Start from the second element
2. Compare it to previous elements
3. Shift larger elements to the right
4. Insert the current element in its correct position

Visualization of a Run begin Sorted

Before insertion sort: [3, 1, 4]

- After sorting: [1, 3, 4] → ready to be merged with other sorted runs

Merging Runs the Heart of Tim Sort

What is Merging Sort

Merging Sort is the process of combining two sorted arrays (runs) into one larger sorted arrays.

- Tim Sort repeatedly merges runs to form bigger and bigger sorted segments.
- Similar to merge sort- but smarter and more adaptive.

Tim Sort's Merging strategy

Tim Sort doesn't merge runs randomly. It uses smart rules to balance performance and minimize memory usage.

Example:

Let's say we have these sorted runs:

[1, 3, 5] , [2, 4, 6]

→ Merged result: [1, 2, 3, 4, 5, 6]

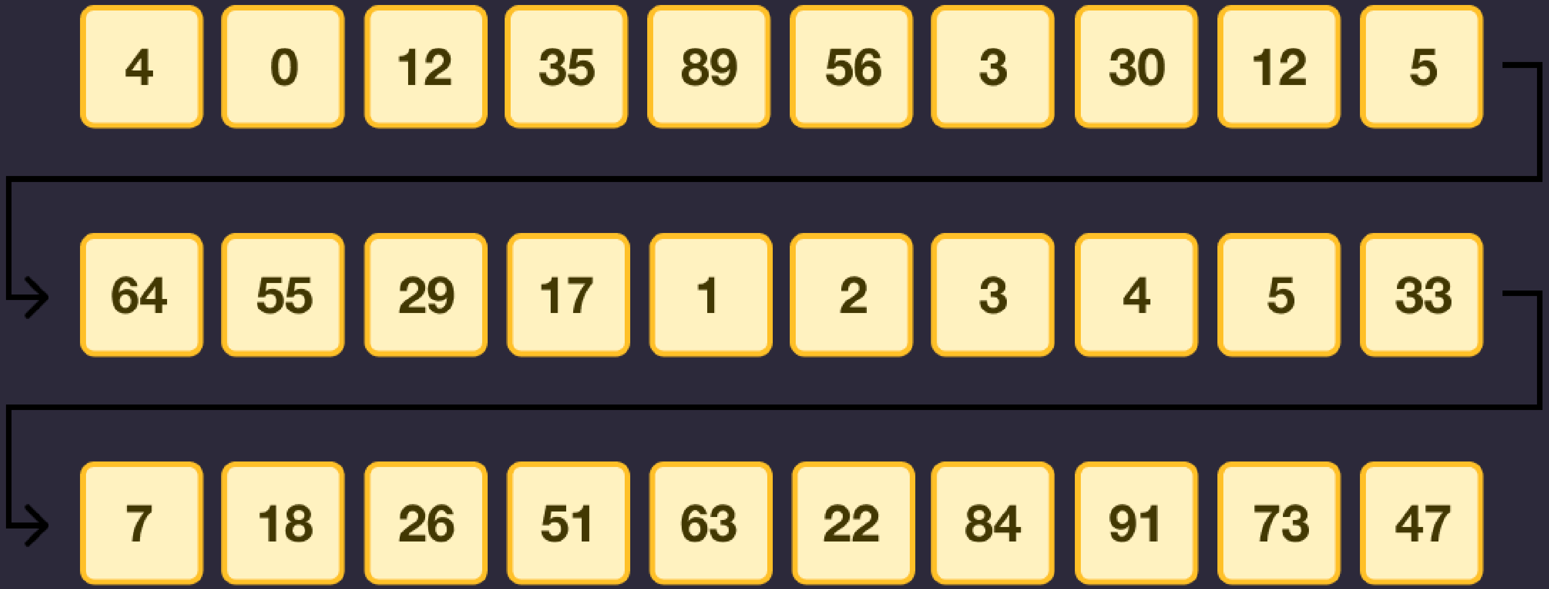
Time And Space Complexity of Tim Sort

Case	Time Complexity	Explanation	Space Complexity
Best Case	O(n)	Inputs is already sorted (or nearly sorted) → detects as runs minimal merging.	O(n) auxiliary space: • Tim sort uses extra memory to store and merge runs • Similar to merge sort in space usage. • Extra arrays help maintain stability and efficient merging.
Average Case	O(n log n)	Typical case with mixed ordering → runs + merging with balancing rules.	
Worst Case	O(n log n)	Completely reversed or random data → insertion sort on runs + full merges.	

Comparison with Other Algorithms

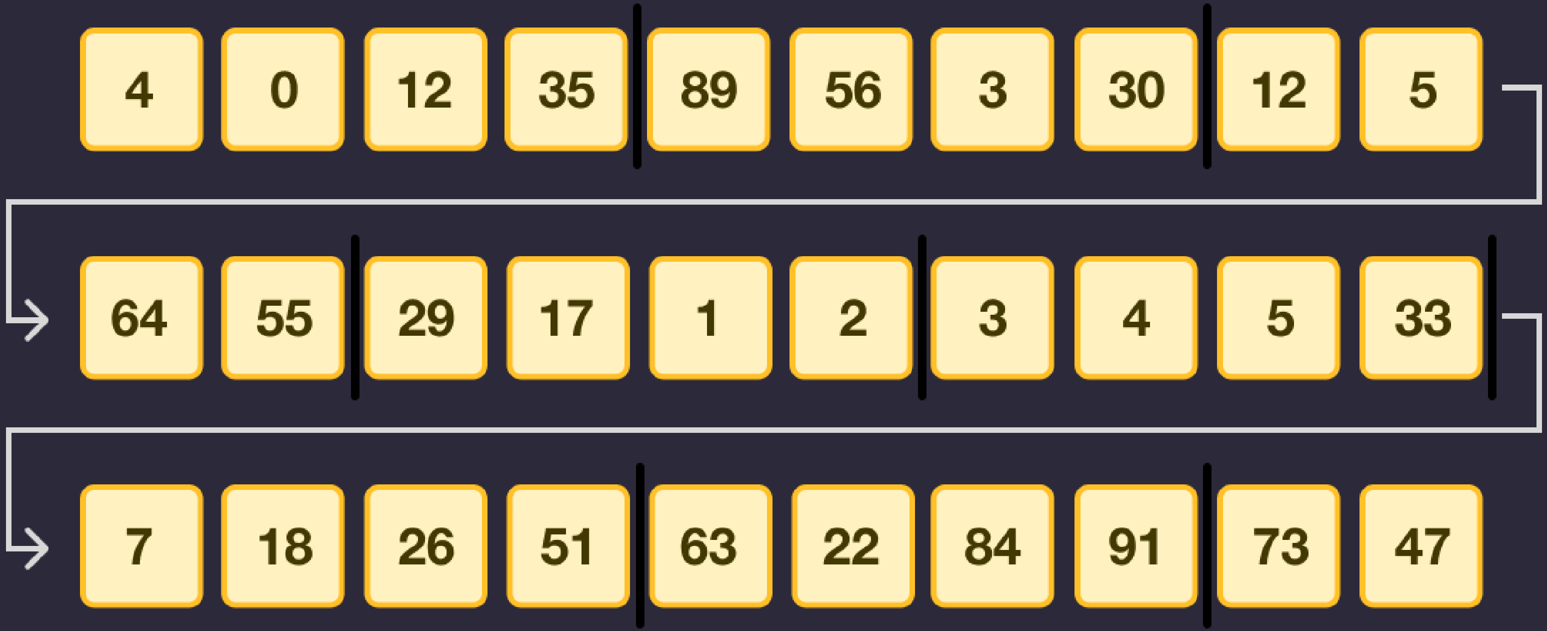
Algorithm	Time(Worst)	Space	Stable
Quick Sort	$O(n^2)$	O(log n)	No
Merge Sort	O(n log n)	O(n)	Yes
Tim Sort	O(n log n)	O(n)	Yes

Assume an array like this:



4	0	12	35	89	56	3	30	12	5
64	55	29	17	1	2	3	4	5	33
7	18	26	51	63	22	84	91	73	47

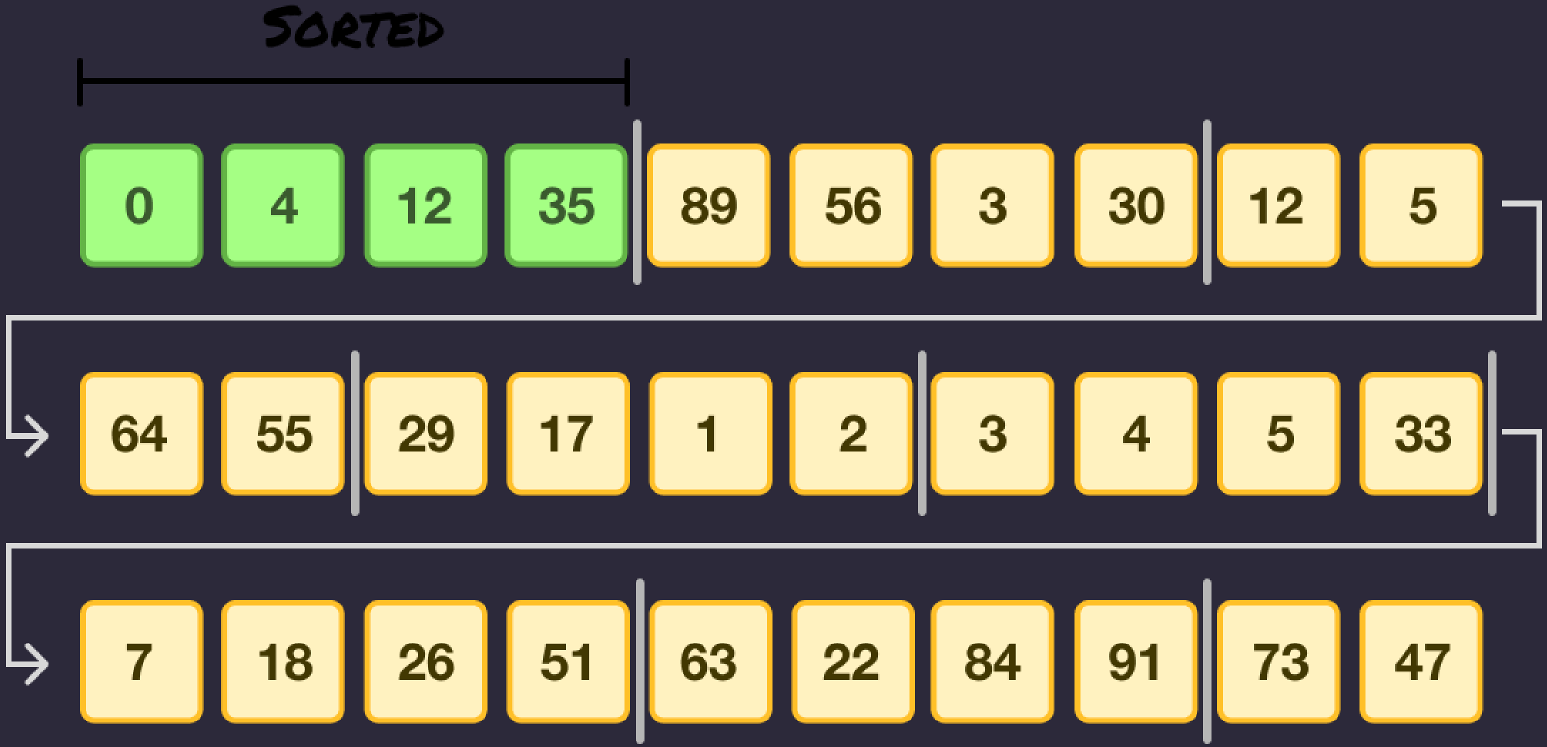
Sorting Runs



4 0 12 35 | 89 56 3 30 | 12 5

64 55 | 29 17 1 2 | 3 4 5 33 |

7 18 26 51 | 63 22 84 91 | 73 47

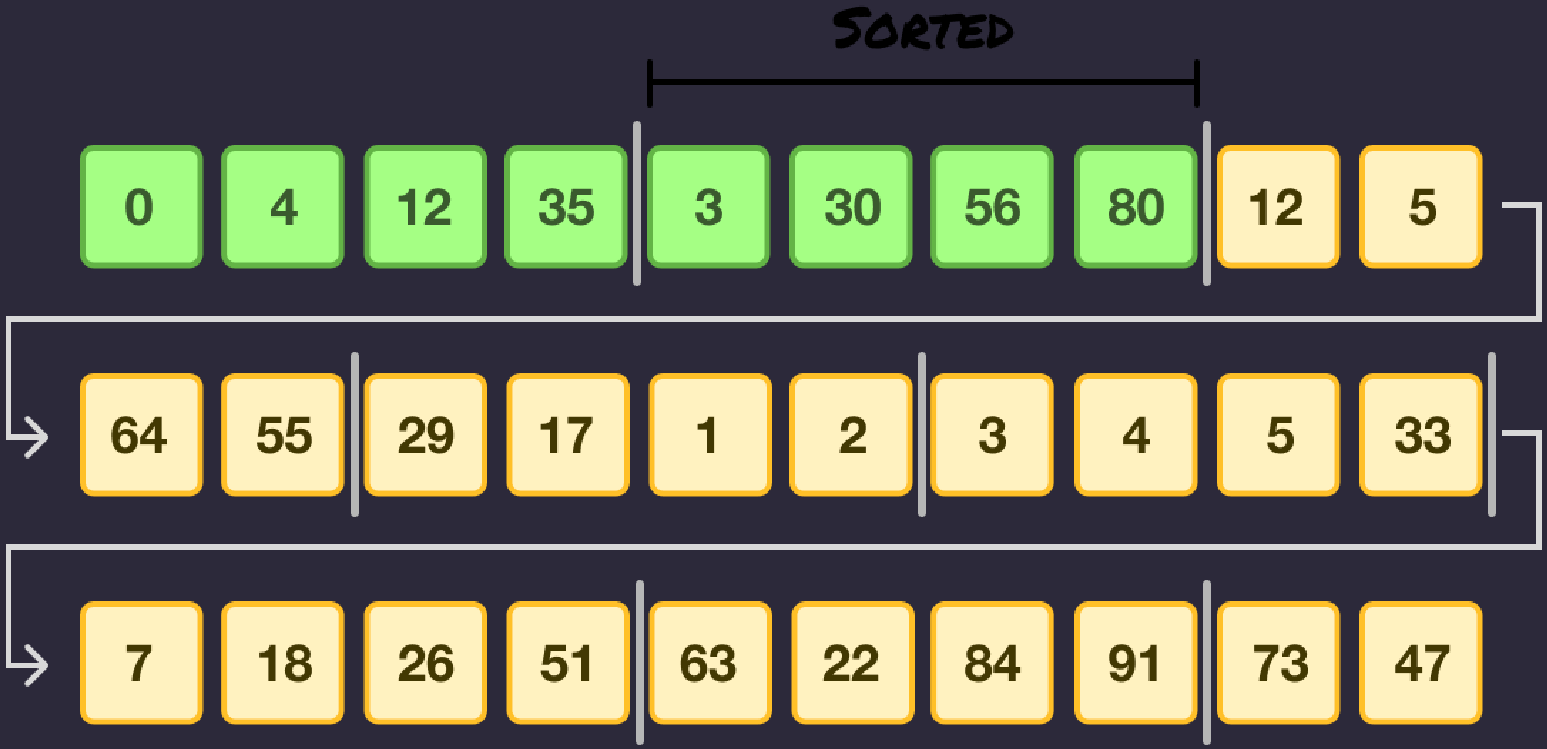


SORTED

0 4 12 35 89 56 3 30 12 5

64 55 29 17 1 2 3 4 5 33

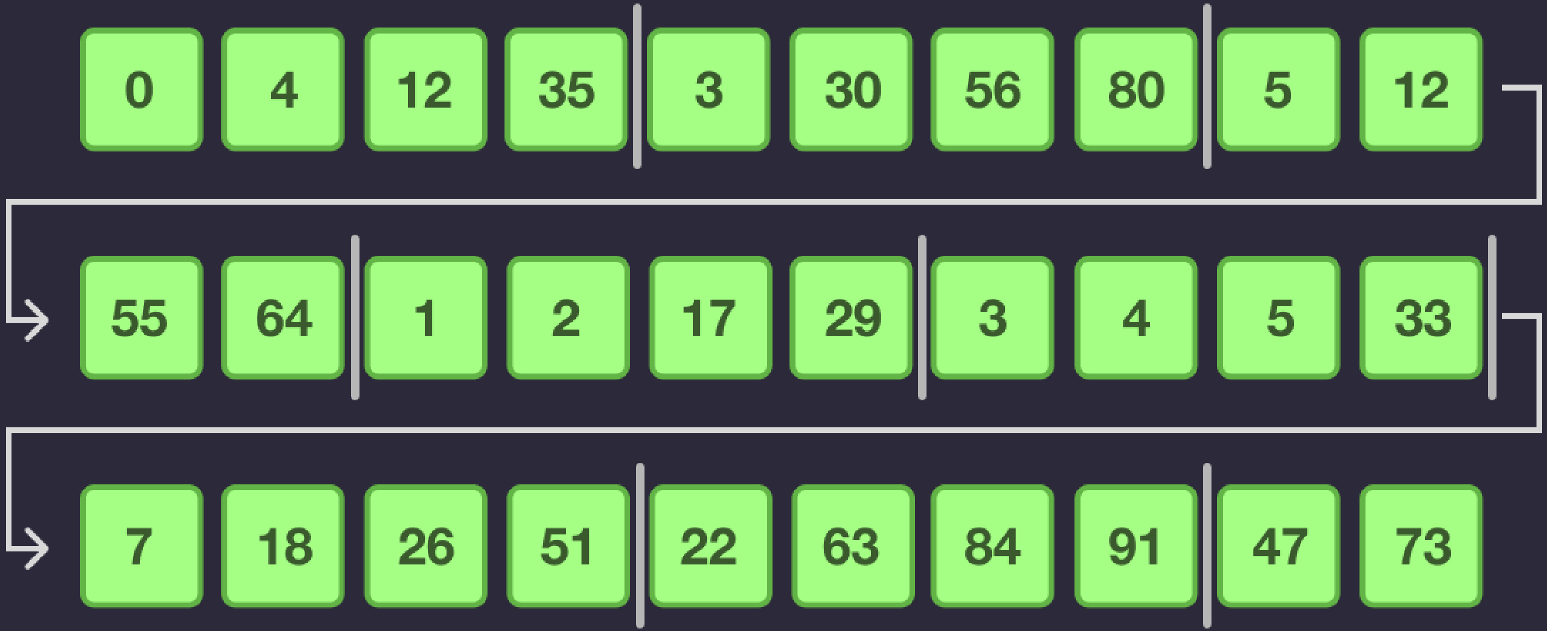
7 18 26 51 63 22 84 91 73 47



0 4 12 35 3 30 56 80 12 5

64 55 29 17 1 2 3 4 5 33

7 18 26 51 63 22 84 91 73 47



0 4 12 35 3 30 56 80 5 12

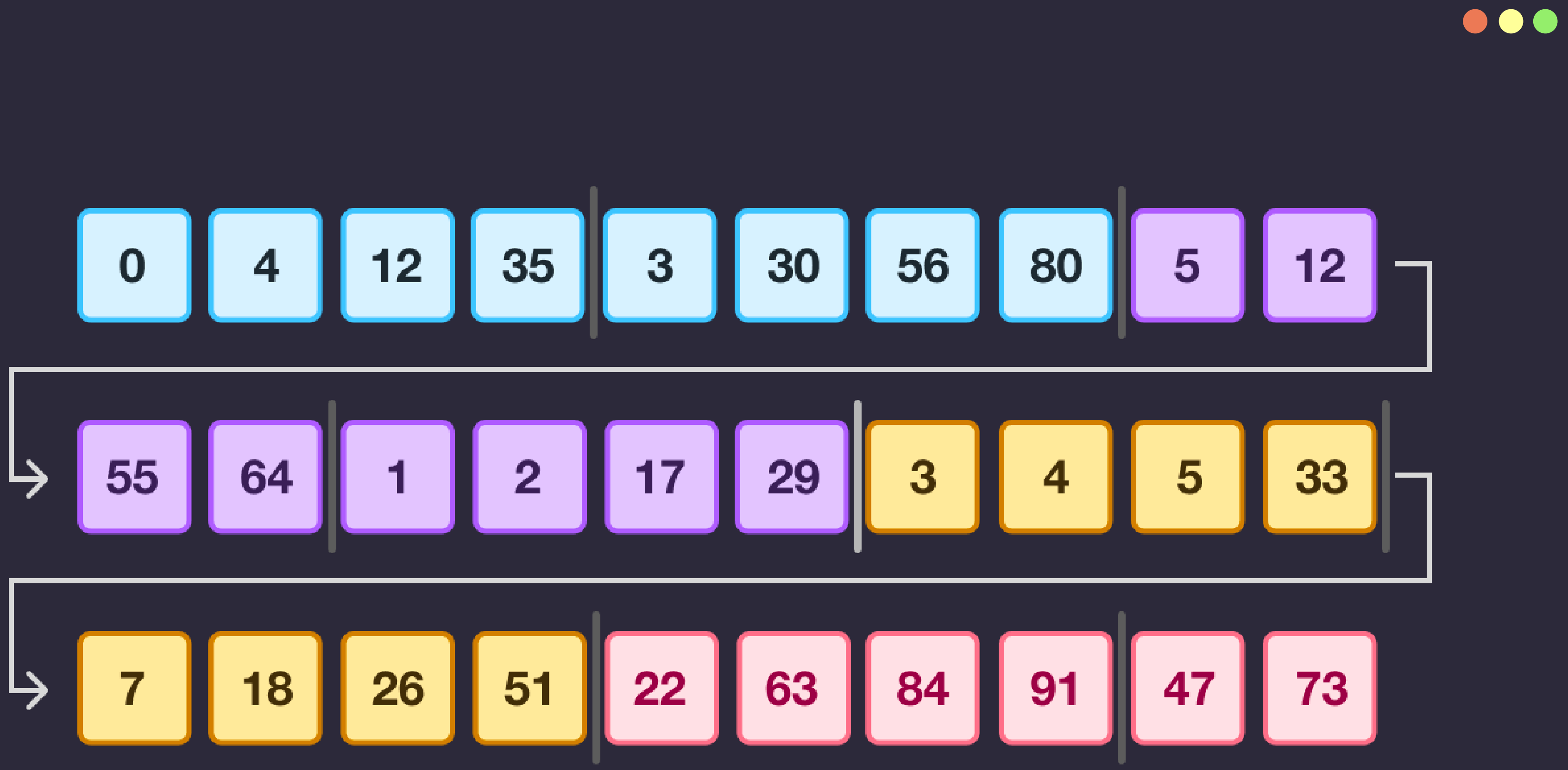
55 64 1 2 17 29 3 4 5 33

7 18 26 51 22 63 84 91 47 73

Merging Runs

The final step is for us to take our individually sorted runs and merge them into larger and larger sorted runs. At the end of all this, the final result will be one combined sorted collection of data.

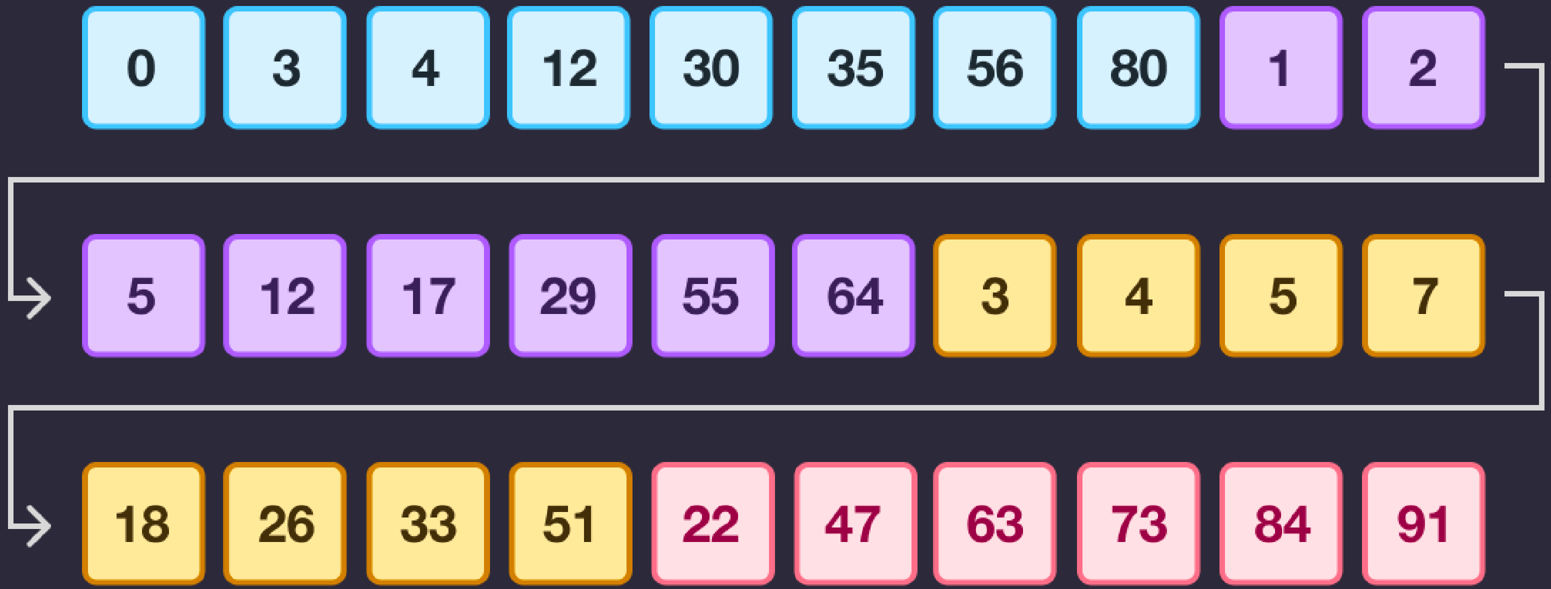
We start by merging adjacent runs together, and I have color-coded the adjacent runs that will be merged first:



0 4 12 35 | 3 30 56 80 | 5 12

55 64 | 1 2 17 29 | 3 4 5 33 |

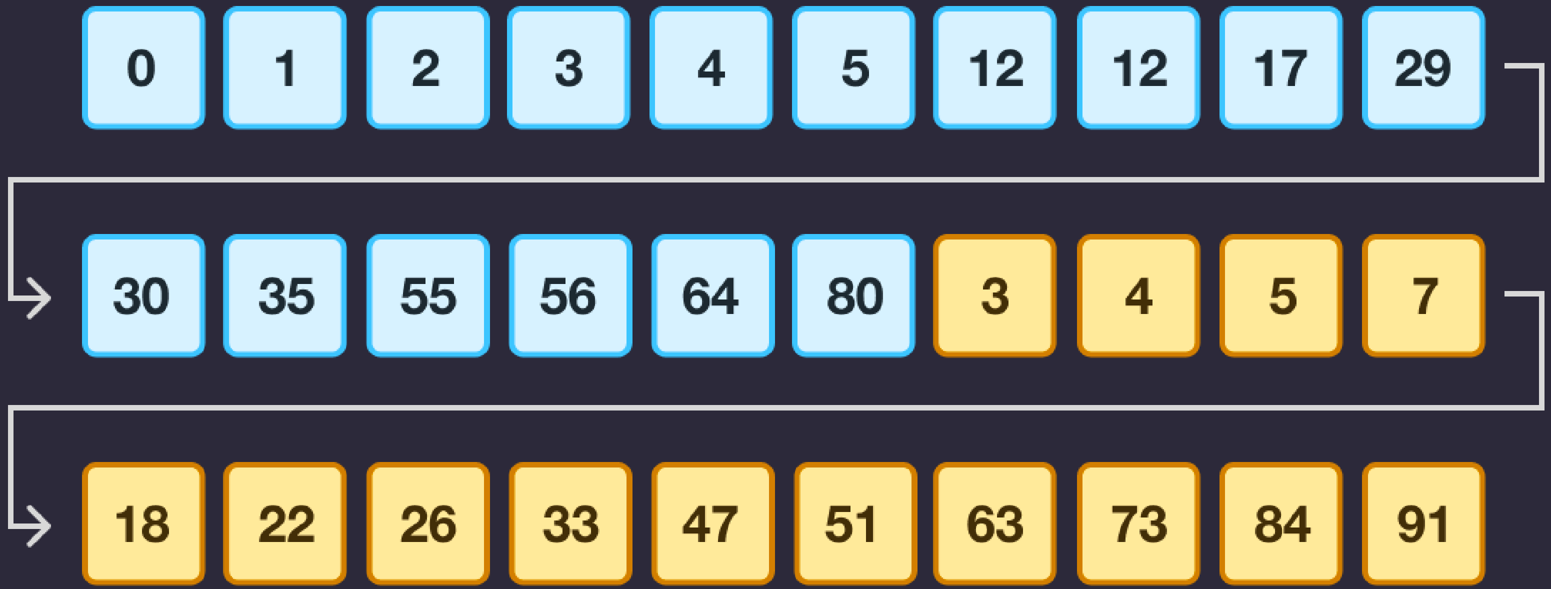
7 18 26 51 | 22 63 84 91 | 47 73



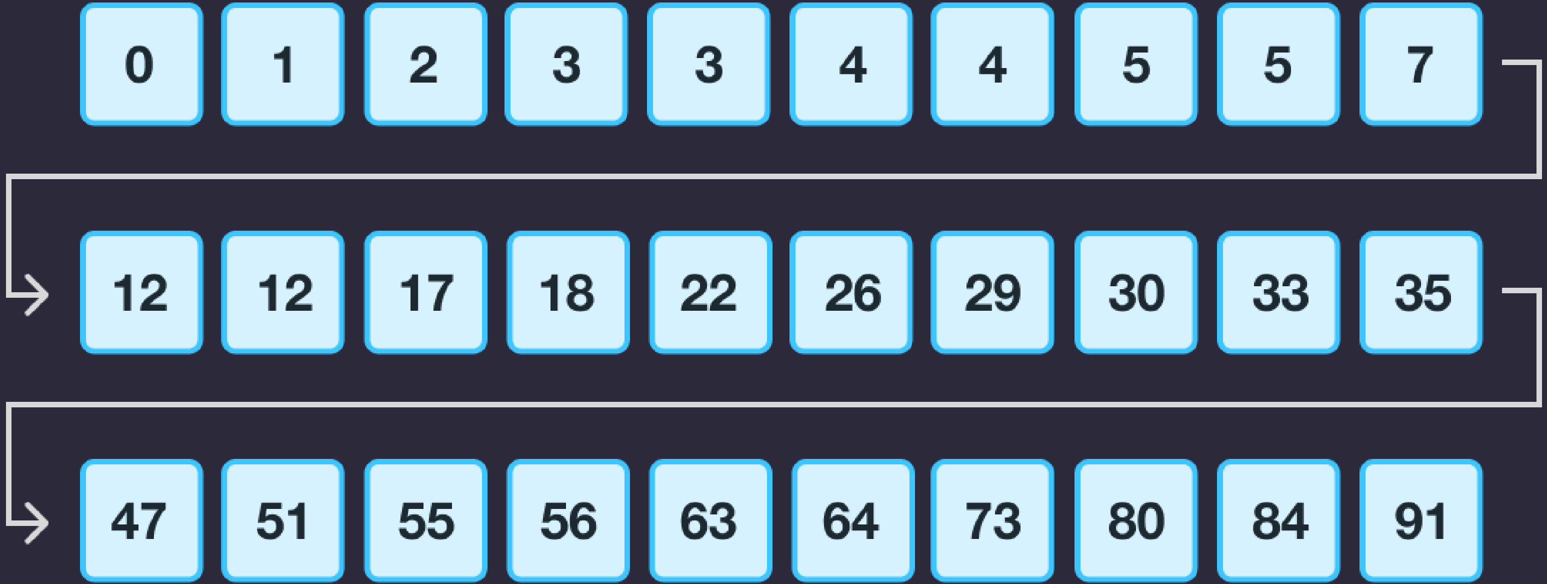
0 3 4 12 30 35 56 80 1 2

5 12 17 29 55 64 3 4 5 7

18 26 33 51 22 47 63 73 84 91



0	1	2	3	4	5	12	12	17	29
30	35	55	56	64	80	3	4	5	7
18	22	26	33	47	51	63	73	84	91



0 1 2 3 3 4 4 5 5 7

12 12 17 18 22 26 29 30 33 35

47 51 55 56 63 64 73 80 84 91

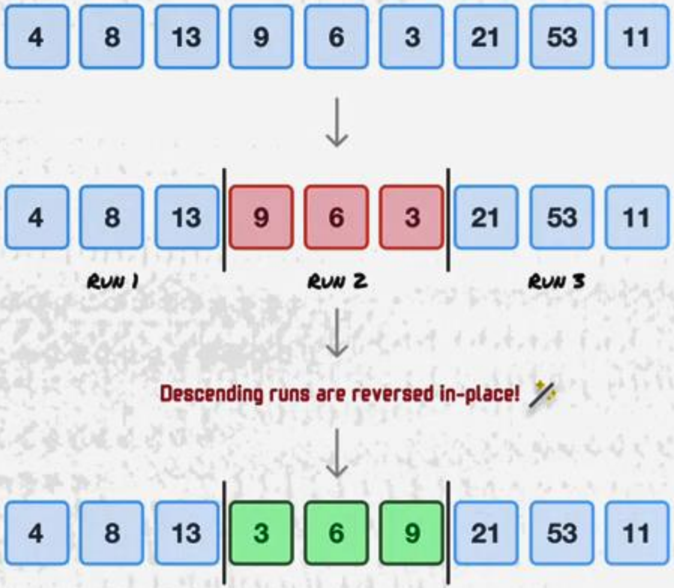
Optimization

Running insertion sort and merging the results is only part of the performance story.

There are additional optimizations at play.

These optimizations revolve around detecting common patterns is our data and customizing the next steps to account for how the data is structured.

Detecting Ascending and Descending Runs



Descending runs are reversed in-place! 🪄

Galloping Mode

Galloping mode (also known as ***binary search insertion***) is an optimization technique used in the Timsort algorithm **during the merging phase**.

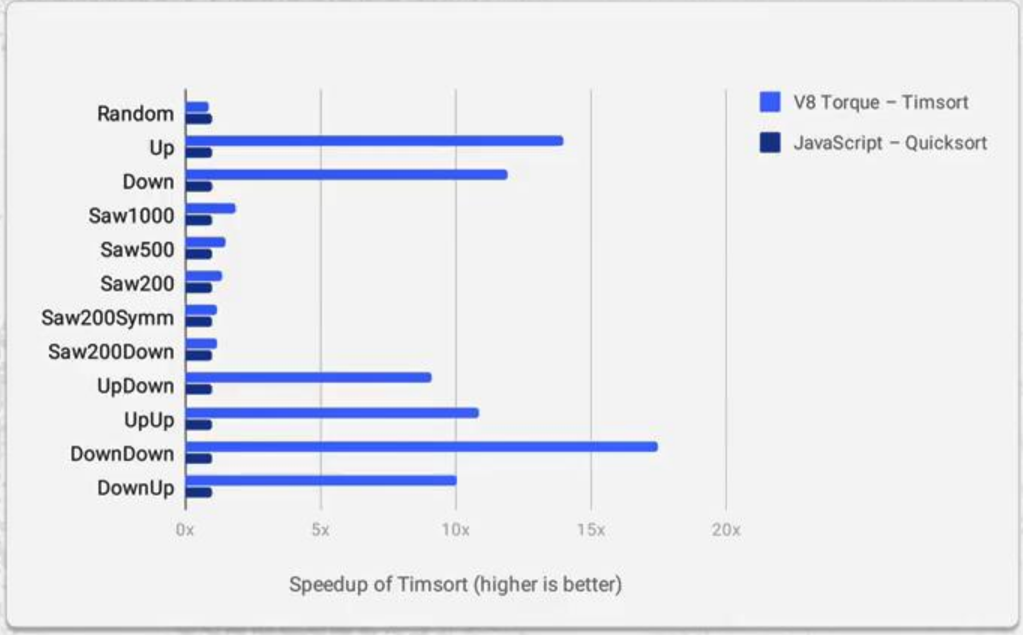
It's designed to handle cases where a run has many elements that are already in their final sorted position relative to the other run.

Performance Characteristics

Timsort is one of the best sorting algorithms out there, and we can see it live up to its grandness when we summarize its time and memory complexity below:

Scenario	**Time Complexity**	**Space Complexity**
Best case	O(n)	O(n)
Worst case	O(n log n)	O(n)
Average case	O(n log n)	O(n)

Performance Characteristics



03 { . .

Application & Summary

- Usage in Python & Java, Android
- Real World Applications
- Summary & Q&A

} . .

Usage in Python, Java, and Android

Python: Default sorting algorithm in sorted() and .sort() since version

Java: Used in Arrays.sort() for objects since Java 7.

Android: Utilized in system-level sorting for UI efficiency .

Note: Tim sort's stability is crucial for predictable sort in moder apps.

Thanks!

< Do you have any questions? >

deepseek

ChatGPT