

5

Quick Sort

MEDIUM

What You Will Learn

- ✓ How the partition step works — the heart of Quick Sort
- ✓ Why choosing a good pivot matters so much
- ✓ How Quick Sort achieves $O(n \log n)$ average
- ✓ The worst case scenario and how to avoid it

Intuition First

Grab any element from the array — call it the pivot. Now rearrange everything: smaller elements go LEFT of pivot, bigger ones go RIGHT. The pivot is now in its exact final position forever. Recursively sort left and right sides.

The Partition Step

- Pick a pivot (we'll use last element for simplicity)
- i tracks the boundary of 'smaller' section (starts at -1)
- j scans from start to pivot-1
- If $arr[j] \leq pivot$: increment i , swap $arr[i]$ and $arr[j]$
- After loop: swap $arr[i+1]$ with pivot. Pivot is now in place!

Partition Visual on [10, 80, 30, 90, 40, 50, 70]

Pivot = 70 (last element)

$i = -1$, scan j from 0 to 5:

```
j=0: 10 <= 70 → i=0, swap(0,0). [10, 80, 30, 90, 40, 50 | 70]
j=1: 80 > 70 → skip. [10, 80, 30, 90, 40, 50 | 70]
j=2: 30 <= 70 → i=1, swap(1,2). [10, 30, 80, 90, 40, 50 | 70]
j=3: 90 > 70 → skip. [10, 30, 80, 90, 40, 50 | 70]
j=4: 40 <= 70 → i=2, swap(2,4). [10, 30, 40, 90, 80, 50 | 70]
j=5: 50 <= 70 → i=3, swap(3,5). [10, 30, 40, 50, 80, 90 | 70]
Final: swap pivot with arr[i+1=4]:
```

70 is in its FINAL position ✓ — left: smaller, right: bigger

10	30	40	50	70	90	80
----	----	----	----	----	----	----

■ Java Implementation

```
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIdx = partition(arr, low, high);
        quickSort(arr, low, pivotIdx - 1); // Sort left
        quickSort(arr, pivotIdx + 1, high); // Sort right
    }
}

public static int partition(int[] arr, int low, int high) {
    int pivot = arr[high]; // Last element as pivot
    int i = low - 1; // i = boundary of smaller section
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
        }
    }
    // Place pivot in correct position
    int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
    return i + 1; // Return pivot's final index
}
```

■ ■ Time & Space Complexity

Case	Time	Space	What happens?
Best	$O(n \log n)$	$O(\log n)$	Pivot always lands in the middle
Average	$O(n \log n)$	$O(\log n)$	Random pivots give balanced partitions
Worst	$O(n^2)$	$O(n)$	Pivot is always smallest/largest (sorted array!)

■ ■ The Worst Case Problem

- Sorted or reverse-sorted array + last-element pivot = $O(n^2)$
- Fix 1: Random pivot — pick random index, swap with last, then partition
- Fix 2: Median-of-3 — pick median of first, mid, last as pivot

■ Pro Tips

- Quick Sort is often faster than Merge Sort in practice (better cache performance)
- Java's `Arrays.sort()` for primitives uses Dual-Pivot QuickSort
- NOT stable — equal elements may swap positions
- Use random pivot to avoid worst case in interviews

■ Practice Problems

[Easy] Trace partition step on [3, 6, 8, 10, 1, 2, 1] with pivot=1

[Med] Implement Quick Sort with random pivot selection

[Med] Dutch National Flag Problem — 3-way partition

[Hard] Find kth largest element using QuickSelect ($O(n)$ avg)

■ Quick Revision

- Pick pivot, partition array around it
- Pivot ends up in its exact final position after partition
- Average: $O(n \log n)$ | Worst: $O(n^2)$ with bad pivots
- In-place: $O(\log n)$ stack space
- NOT stable — fastest in practice for primitives