

# Quick Sort Algorithm

## Divide and Conquer Strategy

### Algorithm Notes

## 1 Introduction

Quick Sort is a highly efficient sorting algorithm that follows the **Divide and Conquer** strategy. It works by selecting a 'pivot' element and partitioning the array around it.

- **Divide:** The array is partitioned into two sub-arrays.
- **Conquer:** The sub-arrays are sorted recursively.
- **Combine:** No significant work is needed to combine; the array is sorted in place.

## 2 Partitioning Logic

The Partition algorithm is the heart of Quick Sort. The goal is to place the **Pivot** element in its correct sorted position.

### Partition Rules

We use two pointers, **P** and **Q**, and a **Pivot** (usually the first element).

#### 1. Initialize:

- **Pivot** =  $A[low]$
- $P = low + 1$  (starts after pivot)
- $Q = high$  (starts at end)

#### 2. Move P (Increment): Move $P$ right until an element **greater** than Pivot is found.

$$A[P] > \text{Pivot} \rightarrow \text{Stop}$$

#### 3. Move Q (Decrement): Move $Q$ left until an element **smaller or equal** to Pivot is found.

$$A[Q] \leq \text{Pivot} \rightarrow \text{Stop}$$

#### 4. Action:

- If  $P < Q$ : **Swap**  $A[P]$  **and**  $A[Q]$ .
- If  $P \geq Q$  (Crossed): **Swap Pivot with**  $A[Q]$ . This places the Pivot in its final position.

### 3 Step-by-Step Visualization

#### Initial State

Input Array:

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 35 | 50 | 15 | 25 | 80 | 20 | 90 | 45 |
|----|----|----|----|----|----|----|----|

(Assume an imaginary  $\infty$  exists at the end to stop P from going out of bounds)

#### Pass 1: Partitioning

##### Step 1: Initialization

- **Pivot = 35** (Index 0)
- $P$  starts at Index 1 (Value 50),  $Q$  starts at Index 7 (Value 45).

##### Step 2: Pointer Movement

- **Check P:** Is  $50 > 35$ ? Yes. Stop P at 50.
- **Check Q:** Is  $45 \leq 35$ ? No.  $90 \leq 35$ ? No.  $20 \leq 35$ ? Yes. Stop Q at 20.

**Current State:**  $P$  is at 50,  $Q$  is at 20. Since  $P < Q$ , we **SWAP**.

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 35 | 20 | 15 | 25 | 80 | 50 | 90 | 45 |
|----|----|----|----|----|----|----|----|

50 and 20 have been swapped.

##### Step 3: Resume Movement

- **Move P:**
  - $15 > 35$ ? No.
  - $25 > 35$ ? No.
  - $80 > 35$ ? Yes. Stop P at 80.
- **Move Q:**
  - $50 \leq 35$ ? No.
  - $80 \leq 35$ ? No.
  - $25 \leq 35$ ? Yes. Stop Q at 25.

**Current State:**  $P$  is at 80 (Index 4),  $Q$  is at 25 (Index 3).

**Condition Check:**  $P > Q$ . The pointers have crossed!

**Step 4: Final Swap** Swap the **Pivot (35)** with **A[Q] (25)**.

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 25 | 20 | 15 | 35 | 80 | 50 | 90 | 45 |
|----|----|----|----|----|----|----|----|

## Result of Pass 1

The array is now partitioned.

| Left Sub-array            | Sorted           | Right Sub-array        |
|---------------------------|------------------|------------------------|
| [25, 20, 15]              | 35               | [80, 50, 90, 45]       |
| (All elements $\leq 35$ ) | (Fixed Position) | (All elements $> 35$ ) |

The algorithm now recursively applies the same logic to the Left and Right sub-arrays.

## 4 Complexity Analysis

### Time Complexity

The performance depends on the pivot selection:

- **Best/Average Case ( $O(n \log n)$ ):** Occurs when the pivot divides the array into roughly equal halves.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- **Worst Case ( $O(n^2)$ ):** Occurs when the array is already sorted (ascending or descending). The pivot divides the array into size 0 and  $n - 1$ .

$$T(n) = T(n - 1) + n$$

## 5 Implementation Note

**The Role of Infinity ( $\infty$ ):** In implementation, we conceptually assume an infinite value exists at  $A[n]$ . This acts as a **sentinel**. If the Pivot is the largest element,  $P$  keeps incrementing. Without  $\infty$ ,  $P$  would access memory outside the array bounds. The sentinel guarantees  $P$  stops.