Let's break down the code step by step and explain it using the example array arr = [64, 34, 25, 12, 22, 11, 90].

**Code and Explanation**

def quick\_sort(arr):

# Base case: If the array has 1 or no elements, it's already sorted

if len(arr) <= 1:

return arr

* **Explanation:** If the array has 1 or 0 elements, it's already sorted, so we return it as is.
* **Example:** For arr = [11] or arr = [], the function will return [11] or [].

# Choose the middle element as the pivot

pivot = arr[len(arr) // 2]

* **Explanation:** The pivot is the middle element of the array. It helps divide the array into three parts.
* **Example:** For arr = [64, 34, 25, 12, 22, 11, 90], len(arr) // 2 = 7 // 2 = 3, so the pivot is arr[3] = 12.

**Divide the array into three parts:**

**1. Elements less than the pivot**

**2. Elements equal to the pivot**

**3. Elements greater than the pivot**

**left = [x for x in arr if x < pivot]**

**middle = [x for x in arr if x == pivot]**

**right = [x for x in arr if x > pivot]**

* **Explanation:** The array is divided into:
  + left: All elements smaller than the pivot.
  + middle: All elements equal to the pivot.
  + right: All elements greater than the pivot.
* **Example:**
  + For arr = [64, 34, 25, 12, 22, 11, 90] and pivot = 12:
    - left = [11] (elements smaller than 12),
    - middle = [12] (elements equal to 12),
    - right = [64, 34, 25, 22, 90] (elements greater than 12).

# Recursively sort the left and right parts, and combine them with the middle

return quick\_sort(left) + middle + quick\_sort(right)

* **Explanation:** Recursively call quick\_sort on left and right parts, then combine the results with middle.
* **Example (Step by Step):**
  1. Start with arr = [64, 34, 25, 12, 22, 11, 90], pivot = 12:
     + left = [11], middle = [12], right = [64, 34, 25, 22, 90].
     + Recursively call:
       - quick\_sort(left) = quick\_sort([11]) → [11] (base case).
       - quick\_sort(right) = quick\_sort([64, 34, 25, 22, 90]).
  2. Sorting right = [64, 34, 25, 22, 90], pivot = 25:
     + left = [22], middle = [25], right = [64, 34, 90].
     + Recursively call:
       - quick\_sort(left) = quick\_sort([22]) → [22] (base case).
       - quick\_sort(right) = quick\_sort([64, 34, 90]).
  3. Sorting right = [64, 34, 90], pivot = 34:
     + left = [], middle = [34], right = [64, 90].
     + Recursively call:
       - quick\_sort(left) = quick\_sort([]) → [] (base case).
       - quick\_sort(right) = quick\_sort([64, 90]).
  4. Sorting right = [64, 90], pivot = 90:
     + left = [64], middle = [90], right = [].
     + Recursively call:
       - quick\_sort(left) = quick\_sort([64]) → [64] (base case).
       - quick\_sort(right) = quick\_sort([]) → [] (base case).
* Combine results:
  1. right = [64] + [90] + [] = [64, 90].
  2. right = [] + [34] + [64, 90] = [34, 64, 90].
  3. right = [22] + [25] + [34, 64, 90] = [22, 25, 34, 64, 90].
  4. Final: left = [11], middle = [12], right = [22, 25, 34, 64, 90] → [11, 12, 22, 25, 34, 64, 90].

# Example usage

arr = [64, 34, 25, 12, 22, 11, 90]

sorted\_arr = quick\_sort(arr)

print("Quick Sorted Array:", sorted\_arr)

* **Output:** Quick Sorted Array: [11, 12, 22, 25, 34, 64, 90].

The **Quick Sort** algorithm is classified as a **divide-and-conquer** algorithm because it breaks the problem into smaller subproblems, solves each of them, and then combines the results. Here's a detailed explanation:

### ****1. Divide:****

The algorithm splits the input array into smaller parts based on a pivot element:

* It identifies three parts:
  + **Left**: Elements smaller than the pivot.
  + **Middle**: Elements equal to the pivot.
  + **Right**: Elements larger than the pivot.

#### Example:

For arr = [64, 34, 25, 12, 22, 11, 90], with pivot = 12:

* left = [11]
* middle = [12]
* right = [64, 34, 25, 22, 90]

This "dividing" process creates smaller subproblems to solve.

### ****2. Conquer:****

Each of the left and right parts is recursively sorted by applying the same quick sort algorithm.

* The base case stops the recursion when the subarrays have 1 or 0 elements (since they are already sorted).

#### Example:

For right = [64, 34, 25, 22, 90], the process repeats:

* Divide: Pivot = 25, left = [22], middle = [25], right = [64, 34, 90].
* Conquer: Recursively sort [22] and [64, 34, 90].

### ****3. Combine:****

The algorithm combines the sorted subarrays into a single sorted array:

* Combine left + middle + right.

#### Example:

For the initial split:

* left = [11]
* middle = [12]
* right = [22, 25, 34, 64, 90] (after sorting recursively) Final result: [11, 12, 22, 25, 34, 64, 90].

### ****Why Divide-and-Conquer?****

1. **Problem Reduction:** It reduces the sorting problem into smaller, independent subproblems.
2. **Recursive Processing:** Each subproblem is solved recursively in isolation, making it manageable.
3. **Combining Results:** Once subproblems are solved, their results are combined to produce the final sorted array.

The "divide" step ensures the array is broken down, while the "conquer" and "combine" steps recursively handle and reassemble the parts. This is why Quick Sort exemplifies the divide-and-conquer strategy.

The name **"Quick Sort"** comes from the algorithm's efficiency and speed in sorting arrays, particularly when compared to other traditional sorting algorithms like **Bubble Sort** or **Selection Sort**. Here are the reasons behind the name:

**1. Divide-and-Conquer Structure:**

* Quick Sort works by dividing the problem into smaller subproblems and solving them recursively. This divide-and-conquer approach often reduces the number of comparisons needed to sort the data compared to simpler algorithms.

**2. Performance in Average Cases:**

* Quick Sort is generally faster than other algorithms because:
  + **Efficient partitioning:** By using a pivot element to divide the array, large sections of the data can be processed in fewer steps.
  + **Logarithmic depth:** In the average case, it reduces the problem size logarithmically, requiring O(nlog⁡n)O(n \log n) comparisons.
* For example:
  + Sorting 1,000 elements might require far fewer operations in Quick Sort than Bubble Sort, which takes O(n2)O(n^2) in the worst case.

**3. Practical Observations:**

* **In-place Sorting:** Quick Sort uses less memory than algorithms like Merge Sort because it sorts the array "in place" (without additional memory for temporary arrays).
* **Empirical Speed:** In real-world scenarios and for typical datasets, Quick Sort is often faster than other algorithms, including Merge Sort, due to lower overhead and cache-friendly behavior.

**4. Historical Context:**

* Quick Sort was developed in 1961 by **Tony Hoare**, who observed its practical speed during his work with early computers. The algorithm’s implementation outperformed many others at the time, leading to its reputation as a "quick" way to sort.

**5. Why Not Always the Quickest?**

* The name "Quick Sort" doesn't imply that it is always the fastest. For example:
  + In the **worst case** (if the pivot always splits poorly), Quick Sort runs in O(n2)O(n^2). However, with good pivot selection strategies (like random or median-of-three), this is rare.

In summary, Quick Sort is named for its **empirical speed**, **divide-and-conquer efficiency**, and ability to outperform many other sorting methods in practical applications, especially for large datasets.