Quick sort

1. Quick Sort performs a varying number of comparisons based on the choice of the pivot element.

It makes approximately O(n\*log(n)) comparisons on average.

In the worst case (when the pivot choice is poor), it can make O(n^2) comparisons.

1. Number of Swaps:

Quick Sort minimizes the number of swaps compared to other sorting algorithms like Bubble Sort or Selection Sort.

The average number of swaps is relatively low, and in most practical cases, it's very efficient.

1. Number of Basic Operations:

Quick Sort's basic operations mainly involve comparisons and swaps.

Apart from that, there are some other basic operations like arithmetic operations and array element access.

Overall, Quick Sort has relatively fewer basic operations compared to some other sorting algorithms.

//primitive operations

1. Running Time in Milliseconds:

Quick Sort is known for its average-case time complexity of O(n\*log(n)), which makes it one of the fastest sorting algorithms in practice.

However, in the worst case, it can degrade to O(n^2), but this is rare with good pivot selection strategies.

The actual running time in milliseconds depends on the implementation, input data distribution, and hardware.

1. Memory Used:

Quick Sort is an in-place sorting algorithm, meaning it doesn't require additional memory proportional to the input size (it uses a small amount of stack space for recursion).

This makes it memory-efficient compared to algorithms like Merge Sort, which require additional memory for merging.

Quick Sort is generally considered one of the fastest sorting algorithms for most practical use cases due to its average-case time complexity of O(n\*log(n)).

Overall, Quick Sort is a well-balanced sorting algorithm that provides a good trade-off between speed and memory efficiency. However, the choice of pivot strategy is crucial for its performance.

Improvement of Quick sort implementation

* Randomized Pivot Selection**:** One of the most common improvements to Quick Sort is to use a randomized pivot selection strategy. Instead of always choosing the first or last element as the pivot, you randomly select a pivot from within the subarray. This helps avoid worst-case scenarios and ensures better average-case performance.
* Median Pivot**:** This improvement involves selecting the pivot as the median of three elements (e.g., the first, middle, and last elements of the subarray). This helps mitigate issues with extreme values and contributes to more balanced partitions.

Placing the median value at the end of the array improves the pivot selection strategy, reduces the likelihood of worst-case scenarios, and simplifies the implementation by making swap operations more convenient during the partition step.

* Hybrid Sorting Algorithms: In practice, Quick Sort is often combined with other sorting algorithms, such as Insertion Sort or Heap Sort, to improve performance for small subarrays.

When the size of the subarray is below a certain threshold, the algorithm switches to Insertion Sort

* Tail Recursion Elimination: The function's return value is directly derived from the recursive call without any further computation or processing. Tail Recursion Elimination works by reusing the current function's stack frame (activation record) for the next function call, rather than creating a new stack frame for each recursive call. This optimization reduces the overhead associated with function calls and stack frame creation, ultimately making the code more memory-efficient and potentially faster.

Time complexity:

* The time complexity for the recursive part is *O*(*n*log*n*) in the best and average cases.
* The **partition** function has a time complexity of *O*(*n*).
* The **insertion\_sort** function has a time complexity of *O*(*k*), where *k* is the sorted subarray size.
* Considering all parts, the overall time complexity is **O(n log n)** in the best and average cases, and **O(n^2)** in the worst case

\*\*The improvements made can lead to a reduction in the number of recursive calls and comparisons, improving the average performance of the algorithm.

Improvement of Merge sort implementation

* In-Place Merge Sort**:** It is a variation of the traditional Merge Sort algorithm that sorts an array without using additional memory, except for a small, constant amount of auxiliary memory. In the standard Merge Sort, a separate array is typically used for merging, while in the in-place version, the merging is done directly within the original array. These variants optimize the memory usage aspect.
* Parallel Merge Sort: In modern computing environments, parallelization is crucial for performance. Parallel versions of Merge Sort distribute the sorting work among multiple processors or cores to speed up the process.
* Optimizations for Small Arrays: Merge Sort can be optimized for small subarrays by switching to a more efficient sorting algorithm, such as Insertion Sort, when the size of the subarray falls below a certain level. The program uses Insertion Sort for small subarrays (size <= threshold). The worst-case time complexity of Insertion Sort is *O*(*n^*2), but for small subarrays, it can be more efficient than the O(*n*log*n*) Merge Sort.

Time complexity:  
The time complexity of the Merge Sort algorithm is O(*n*log*n*) in all cases (best, worst, and average). This is because the array is recursively divided into halves, and then merged back together, with each merge operation taking *O*(*n*) time.