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.

2)

Merge Sort

a. \*\*Number of Comparisons:\*\*

- Merge Sort performs a fixed number of comparisons regardless of the input data distribution.

- It always makes approximately O(n\*log(n)) comparisons in all cases, including the worst case.

b. \*\*Number of Swaps:\*\*

- Merge Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements.

- Merge Sort operates by creating temporary arrays and copying elements between them, but this is not counted as swaps.

c. \*\*Number of Basic Operations:\*\*

- Merge Sort's basic operations primarily involve comparisons, element copying between arrays, and recursive function calls.

- Other basic operations include arithmetic operations and array element access.

- The number of basic operations in Merge Sort is relatively higher compared to some other sorting algorithms due to the need for merging.

d. \*\*Running Time in Milliseconds:\*\*

- Merge Sort is known for its consistent and predictable time complexity of O(n\*log(n)) in all cases.

- It provides a stable and guaranteed performance, making it suitable for scenarios where worst-case performance is critical.

- The actual running time in milliseconds depends on the implementation, input data size, and hardware, but it scales efficiently with large datasets.

e. \*\*Memory Used:\*\*

- Merge Sort is not an in-place sorting algorithm; it requires additional memory proportional to the input size.

- It creates temporary arrays for merging, which can consume additional memory.

- While it uses more memory than in-place algorithms like Quick Sort, it is still considered memory-efficient compared to some other sorting algorithms like Heap Sort.

Merge Sort is known for its stability, predictable performance, and guaran teed worst-case time complexity of O(n\*log(n)). It does not involve swaps and is memory-efficient for large datasets. While it may not be the fastest sorting algorithm in terms of runtime, its reliability and consistent performance make it a popular choice, especially when sorting large datasets where worst-case behavior needs to be avoided.

Overall, Merge Sort is well-suited for scenarios where a stable and worst-case-efficient sorting algorithm is required.

3) in place heap sort

Worst Case Time Complexity of Heap Sort

The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we would need to call max-heapify every time we remove an element. In such a case, considering there are 'n' number of nodes-

The number of swaps to remove every element would be log(n), as that is the max height of the heap

Considering we do this for every node, the total number of moves would be n \* (log(n)).

Therefore, the runtime in the worst case will be O(n(log(n)).

Best Case Time Complexity of Heap Sort

The best case for heapsort would happen when all elements in the list to be sorted are identical. In such a case, for 'n' number of nodes-

Removing each node from the heap would take only a constant runtime, O(1). There would be no need to bring any node down or bring max valued node up, as all items are identical.

Since we do this for every node, the total number of moves would be n \* O(1).

Therefore, the runtime in the best case would be O(n).

Average Case Time Complexity of Heap Sort

In terms of total complexity, we already know that we can create a heap in O(n) time and do insertion/removal of nodes in O(log(n)) time. In terms of average time, we need to take into account all possible inputs, distinct elements or otherwise. If the total number of nodes is 'n', in such a case, the max-heapify function would need to perform:

log(n)/2 comparisons in the first iteration (since we only compare two values at a time to build max-heap)

log(n-1)/2 in the second iteration

log(n-2)/2 in the third iteration

and so on

So mathematically, the total sum would turn out to be-

(log(n))/2 + (log(n-1))/2 + (log(n-2))/2 + (log(n-3))/2 + ...

Upon approximation, the final result would be

=1/2(log(n!))

=1/2(n∗log(n)−n+O(log(n)))

a. \*\*Number of Comparisons:\*\*

- In-Place Heap Sort performs a varying number of comparisons depending on the structure of the input data and the order of the elements.

- In the worst case, it can make up to O(n\*log(n)) comparisons, which is similar to other efficient sorting algorithms like Merge Sort and Quick Sort.

b. \*\*Number of Swaps:\*\*

- Heap Sort is known for its in-place nature, which means it minimizes the number of swaps compared to some other sorting algorithms.

- It primarily swaps elements within the array to maintain the heap structure.

- The average and worst-case number of swaps are both O(n\*log(n)), making it more efficient in terms of swaps than some other algorithms like Bubble Sort.

c. \*\*Number of Basic Operations:\*\*

- Heap Sort's basic operations mainly involve comparisons, swaps, and adjustments to maintain the heap property.

- It also includes arithmetic operations and array element access.

- The number of basic operations is relatively moderate compared to some other sorting algorithms.

d. \*\*Running Time in Milliseconds:\*\*

- In-Place Heap Sort has an average and worst-case time complexity of O(n\*log(n)), which makes it an efficient sorting algorithm.

- It provides a balance between speed and memory efficiency.

- The actual running time in milliseconds depends on the implementation, input data distribution, and hardware but scales efficiently with large datasets.

e. \*\*Memory Used:\*\*

- Heap Sort is an in-place sorting algorithm, meaning it does not require additional memory proportional to the input size.

- It operates directly on the input array without creating additional data structures.

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

In-Place Heap Sort is known for its efficiency in terms of both comparisons and swaps. It has a consistent average and worst-case time complexity of O(n\*log(n)), making it suitable for various scenarios. Its in-place nature makes it memory-efficient, and it is often used when both stable sorting and minimal memory usage are required.

Overall, In-Place Heap Sort is a well-balanced sorting algorithm that provides good performance and is particularly useful for sorting large datasets with limited memory resources.

4)

Insertion sort

a. \*\*Number of Comparisons:\*\*

- Insertion Sort performs a varying number of comparisons depending on the input data's order.

- In the best case (when the input is already sorted), it makes only n-1 comparisons, where n is the number of elements.

- In the worst case (when the input is reverse sorted), it can make up to O(n^2) comparisons, where n is the number of elements.

- In the average case, it makes approximately (n^2)/4 comparisons.

b. \*\*Number of Swaps:\*\*

- Insertion Sort is known for its simplicity and minimal swaps.

- In the best case (already sorted input), it makes zero swaps.

- In the worst case, it can make up to O(n^2) swaps.

- In the average case, it makes approximately (n^2)/4 swaps.

c. \*\*Number of Basic Operations:\*\*

- Insertion Sort's basic operations mainly involve comparisons and swaps.

- It also includes array element access and arithmetic operations.

- The number of basic operations is moderate and depends on the input data's order.

d. \*\*Running Time in Milliseconds:\*\*

- Insertion Sort has an average and worst-case time complexity of O(n^2), where n is the number of elements.

- It is generally considered less efficient than other sorting algorithms like Quick Sort and Merge Sort, especially for large datasets.

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

e. \*\*Memory Used:\*\*

- Insertion Sort is an in-place sorting algorithm, meaning it operates directly on the input array without requiring additional memory.

- It is memory-efficient compared to algorithms like Merge Sort, which require additional memory for merging.

Insertion Sort is often used for small datasets or when the input data is nearly sorted. It has a simple and easy-to-understand implementation, making it suitable for educational purposes. However, it is less efficient than other sorting algorithms for large datasets due to its quadratic time complexity.

Overall, Insertion Sort is a straightforward sorting algorithm that performs well for small datasets or when elements are already partially sorted. It is not the most efficient sorting algorithm for large datasets but has its practical use cases where simplicity and minimal memory usage are priorities.

5)

Bucket sort

a. \*\*Number of Comparisons:\*\*

- Bucket Sort is a non-comparative sorting algorithm. It doesn't perform traditional element comparisons as seen in comparison-based sorts like Quick Sort or Merge Sort.

- Instead, it distributes elements into buckets based on their values.

b. \*\*Number of Swaps:\*\*

- Bucket Sort doesn't involve swaps between elements like Insertion Sort or Selection Sort.

- It focuses on distributing elements into buckets and then sorting each bucket individually.

- Within each bucket, another sorting algorithm may be used, which can involve swaps, but the number of swaps depends on the specific sorting algorithm used for the buckets.

c. \*\*Number of Basic Operations:\*\*

- Basic operations in Bucket Sort mainly include:

- Distributing elements into buckets based on their values (which involves array element access and arithmetic operations).

- Sorting each bucket (which can involve comparisons and swaps depending on the chosen sorting algorithm).

- The number of basic operations depends on the input data's distribution, the number of buckets, and the sorting algorithm used for the buckets.

d. \*\*Running Time in Milliseconds:\*\*

- The running time of Bucket Sort depends on several factors:

- The number of elements to be sorted (n).

- The number of buckets (k).

- The distribution of elements within the buckets.

- The sorting algorithm used for each bucket.

- On average, Bucket Sort has a linear time complexity, O(n + k), where n is the number of elements to be sorted and k is the number of buckets.

- In the worst case, when elements are not evenly distributed into buckets, the time complexity can degrade to O(n^2) if a slow sorting algorithm is used for individual buckets.

- The actual running time in milliseconds can vary based on these factors and the hardware.

e. \*\*Memory Used:\*\*

- Bucket Sort requires additional memory for the buckets themselves, which can vary depending on the implementation.

- It is not an in-place sorting algorithm like Quick Sort or Insertion Sort.

- The additional memory required for the buckets makes it less memory-efficient for large datasets compared to some other in-place sorting algorithms.

Bucket Sort is particularly useful when the input data is uniformly distributed across a range of values and can be divided into relatively equal-sized buckets. It can be efficient for sorting real numbers in a specific range, such as grades or exam scores. The choice of sorting algorithm used for individual buckets can impact its performance, so selecting an appropriate algorithm is essential. Overall, Bucket Sort is an effective sorting method for certain scenarios but may not be as versatile as some other sorting algorithms.

6)

Radix sort

a. \*\*Number of Comparisons:\*\*

- Radix Sort is a non-comparative sorting algorithm, which means it doesn't perform element comparisons.

- Instead, it relies on counting and distributing elements based on their digits or bytes.

b. \*\*Number of Swaps:\*\*

- Radix Sort doesn't involve swapping elements, as it is a non-comparative sorting algorithm.

- Elements are directly moved to their respective positions based on their digits or bytes.

c. \*\*Number of Basic Operations:\*\*

- Radix Sort's basic operations involve counting, distributing, and accessing individual digits or bytes of elements.

- These operations are generally simple and don't include comparisons or swaps.

d. \*\*Running Time in Milliseconds:\*\*

- Radix Sort has a time complexity of O(nk) in the worst and average cases, where n is the number of elements to be sorted, and k is the number of digits or bytes in the maximum element.

- The actual running time in milliseconds depends on the value of k and the size of the dataset.

- For fixed-size integers, k is constant, making Radix Sort a linear time complexity sorting algorithm (O(n)).

e. \*\*Memory Used:\*\*

- Radix Sort is an in-place sorting algorithm, meaning it operates directly on the input array without requiring additional memory.

- It is memory-efficient, particularly when compared to algorithms like Merge Sort that require additional memory for merging.

Radix Sort is a specialized sorting algorithm that works well for fixed-size integers or strings with a consistent number of digits or bytes. It is not a comparison-based sorting algorithm, making it suitable for scenarios where element comparisons are expensive or not applicable. Radix Sort is particularly efficient for sorting large datasets of integers, where it can outperform comparison-based sorting algorithms like Quick Sort or Merge Sort due to its linear time complexity. However, its performance may degrade for datasets with varying digit or byte lengths.

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.