**Sorting Algorithms**

**Selection Sort:** a simple and efficient sorting algorithm that repeatedly selects the smallest (or largest) element from the list's unsorted portion and moves it to the sorted portion.

Simple and easy to understand.

Works well with small datasets.

**Algorithm**:

1. **Start** with the first element of the array.
2. **Find** the smallest element in the unsorted portion of the array.
3. **Swap** this smallest element with the first element of the unsorted portion.
4. **Move** the boundary between the sorted and unsorted portions one element to the right.
5. Repeat steps 2-4 until the entire array is sorted.

The **time complexity** of Selection Sort is **O(N2)** as there are two nested loops

* One loop to select an element of Array one by one = O(N)
* Another loop to compare that element with every other Array element = O(N)
* Therefore overall complexity = O(N) \* O(N) = O(N\*N) = O(N2)

**Disadvantages:**

* Selection sort has a time complexity of O(n^2) in the worst and average case.
* Does not work well on large datasets.
* Does not preserve the relative order of items with equal keys which means it is not stable.

**Bubble Sort:** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Bubble sort is easy to understand and implement.

It does not require any additional memory space.

It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

**Algorithm**:

1. **Start** at the beginning of the list.
2. **Compare** each pair of adjacent (If the first element is greater than the second element, they are swapped) elements.
3. **Swap** the elements if they are in the wrong order.
4. **Move** to the next pair and repeat the process.
5. **Repeat** the entire process for the remaining elements until no swaps are needed, indicating that the list is sorted.

*In the Bubble Sort algorithm,*

* *traverse from* ***left*** *and compare adjacent elements and the higher one is placed at right side.*
* *In this way, the largest element is moved to the rightmost end at first.*
* *This process is then continued to find the second largest and place it and so on until the data is sorted.*

**Time Complexity:**O(N2)

Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.

**Disadvantages:**

* It is very slow for large data sets because time complexity is O(N2).
* Bubble sort involves a large number of swaps, which can be costly in terms of time.

**Insertion Sort**

*works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a****stable sorting****algorithm.*

**Algorithm**:

1. We have to start with second element of the array as first element in the array is assumed to be sorted.
2. Compare second element with the first element and check if the second element is smaller then swap them.
3. Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
4. Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
5. Repeat until the entire array is sorted.

**Time Complexity**:O(N^2)

* Stable sorting algorithm.
* Efficient for small lists and nearly sorted lists.

**Disadvantages**

* Inefficient for large lists.
* Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.
* Can be useful when the array is already almost sorted

**Merge Sort**

It follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Merge sort consistently performs well.

it performs well even on large datasets.

The divide-and-conquer approach is straightforward.

**Algorithm**:

1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

**Time-Complexity:** O(n log n)

**Disadvantages:**

Merge sort requires additional space proportional to the size of the input array, making it less space-efficient compared to some in-place sorting algorithms like quicksort.

**Quick Sort:**

Quicksort is a sorting algorithm based on the divide-and-conquer approach.

It is a divide-and-conquer algorithm that makes it easier to solve problems.

It is efficient on large data sets.

It has a low overhead, as it only requires a small amount of memory to function.

Fastest general-purpose algorithm for large data when stability is not required.

**Algorithm**:

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).  
     
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

**Time Complexity:**

**Best Case and Average Case**: O (N log (N))

**Worst Case:**O (N ^ 2)

**Disadvantages**:

* It has a worst-case time complexity of O (N 2 ), which occurs when the pivot is chosen poorly.
* It is not a good choice for small data sets.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of a quick sort

**Heap Sort**

comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Efficient Time Complexity

Memory usage can be minimal (by writing an iterative happify() instead of a recursive one)

It is simpler to understand than other equally efficient sorting algorithms

**Algorithm**:

First, convert the array into heap data structure using Heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then happify the root of the heap. Repeat this process until size of the heap is greater than 1.

* Build a heap from the given input array.
* Repeat the following steps until the heap contains only one element:
  + Swap the root element of the heap (which is the largest element) with the last element of the heap.
  + Remove the last element of the heap (which is now in the correct position).
  + Heapify the remaining elements of the heap.
* The sorted array is obtained by reversing the order of the elements in the input array.

**Time Complexity**: O (n log n)

**Disadvantages:**

1. Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is O(n Log n) for both.
2. Heap sort is unstable. It might rearrange the relative order.
3. Heap Sort is not very efficient when working with highly complex data.