MERGE SORT

Introduction:

Sorting is a fundamental operation in Computer Science used to arrange elements in a particular order (ascending to descending). One of the most efficient and widely used sorting algorithms is merge sort. It follows the divide and Conquer approach, breaking a problem into smaller subproblems, solving them independently, and combining their solutions.

Concept of Merge Sort:

Merge sort works by recursively dividing the array into two halves until each subarray contains a single element. Then it merges the sort subarrays to form a sorted array.

Steps Involved in Merge Sort:

Merge sort is works on divide and Conquer approach.

Divide: Split the array into two halves

Conquer: Recursively sort both halves using merge sort

Combine: Merge the sorted halves to get single sorted array

Algorithms for Merge Sort:

* If the array has one or zero elements return (It is already sorted)
* Find the middle index of the array
* Recursively apply merge sort to the left half.
* Recursively apply merge sort to the right half.
* Merge the two sorted halves into one sorted array.

Implementation of Merge Sort:

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    leftHalf = mergeSort(arr[:mid])
```

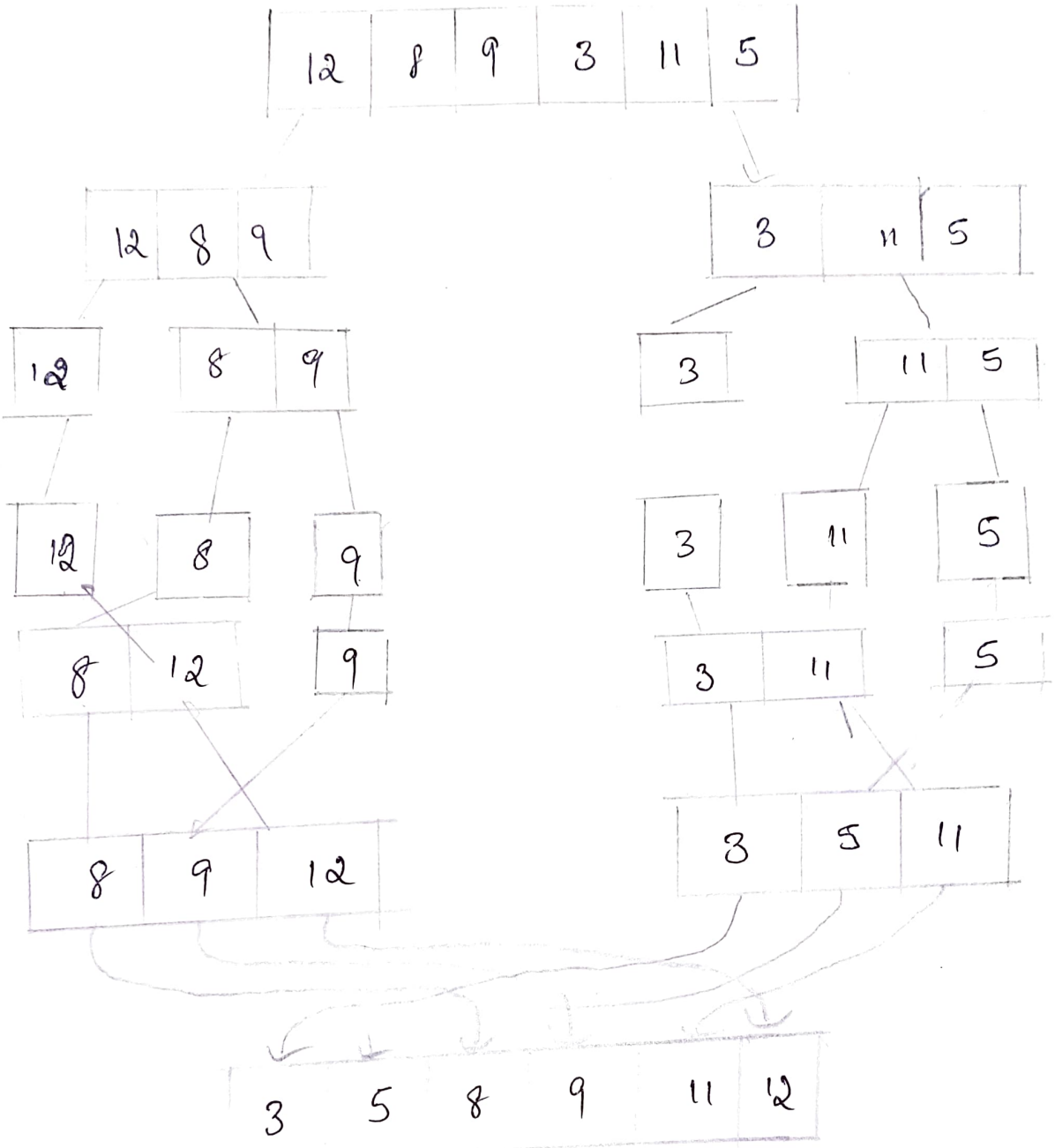
```
    right_half = mergesort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    sorted_arr = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])

arr = [38, 27, 43, 3, 9, 82]
sorted_arr = mergesort(arr)
Print("Sorted array:", sorted_arr)
```

Time Complexity:

Best Case: $O(n \log n)$

Average Case: $O(n \log n)$

Worst Case: $O(n \log n)$



Space Complexity:

Space Merge Sort requires additional memory for the temporary arrays used in merging. Its Space Complexity is O(n).

Advantages of Merge Sort:

* Stable Sorting algorithm: It maintains the order of equal elements.

* Guaranteed Performance: It performs efficiently even in worst-case scenarios.

* Useful for Linked list: It is an excellent choice for sorting linked list.

Disadvantages of merge sort:

* Extra space: It requires additional memory, making it inefficient for large datasets in memory-constrained environments.

* Slower of small data: Simple sorting algorithms like insertion sort might perform better for smaller arrays.

Application of Merge sort:

* Sorting large datasets.
* External sorting (handling large files on disk)
* Sorted linked list efficiently.
* Used in the implementation of other complex sorting algorithms.

Quick Sort

Introduction:

Sorting is a fundamental operation in data structures and algorithms used to arrange elements in specific order. Quick Sort is the one of the most efficient sorting algorithms, following the divide and Conquer approach. It is widely used due to its speed and in Place sorting Capability.

Working Principle of Quick Sort:

Quick Sort works by selecting a Pivot element and Partitioning that array such that:

* Elements smaller than the Pivot are placed on the left.
* Elements greater than the Pivot are placed on the right
* The Process is recursively applied to both Partitions until the array is sorted.

Steps of Quick Sort algorithm:

1. Choose a Pivot: Select an element as the Pivot (eg. first, last, middle or random element)
2. Partitioning: Rearrange the array such the elements smaller than the Pivot move to the left and elements greater than the Pivot move to the right.
3. Recursive Sorting: Apply Quick sort to the left and right Partitions.
4. Base Case: when the array has one or zero elements it is already sorted.

Algorithm for Quick Sort:

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1
def quickSort(arr, low, high):
```

If low < high:

Pi = Partition (arr, low, high).

QuickSort (arr, low, Pi-1)

QuickSort (arr, Pi+1, high)

arr = [10, 80, 30, 90]

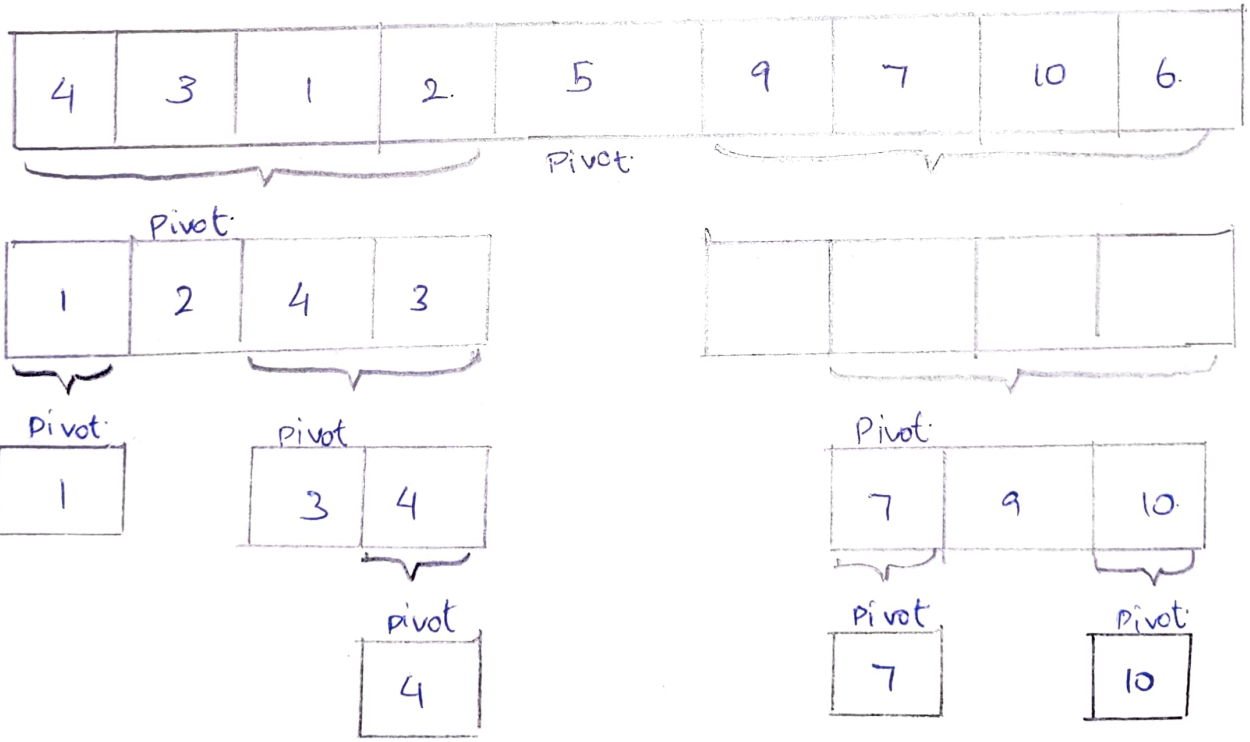
n = len (arr)

Print ("original array", arr)

QuickSort (arr, 0, n-1)

Print ("Sorted array", arr)

Diagram representation of quick sort.



Time Complexity:

Best Case: $O(n \log n)$ (when the Pivot divides the array evenly)

Average Case: $O(n \log n)$

Worst Case: $O(n^2)$ (occurs when the Pivot is always the smallest or largest element)

Space Complexity:

* $O(\log n)$ for the recursive stack in the best case.
* $O(n)$ is the worst case

Advantage of Quick Sort:

1. Fast Performance: on average, Quick Sort is faster than merge Sort
2. In-Place Sorting: Requires very little extra memory.
3. Efficient large data: Ideal in many real-world applications.

Disadvantages of Quick Sort:

1. Worst Case Complexity of $O(n^2)$: happens when the Pivot selection is Poor.

2. Not a Stable Sort: Equal elements might not maintain their original order.

Applications of Quick Sort:

* Database and file Sorting
* Competitive Programming and System applications
* Used in the Implementation of Standards Sorting libraries like Sort()

SELECTION SORT

Introduction:

Selection Sort is a Simple and Intuitive Sorting algorithm that repeatedly Selects the Smallest element from an unsorted Part of the array and Swaps it with the first element of the unsorted Part.

Working Principle of Selection Sort:

Selection Sort works by dividing the array into two Parts:

* Sorted Part: Initialy empty, elements are added one by one.

* Unsorted Part: Initially Contains all elements, and the Smallest element is selected and moved to the Sorted Part.

Steps of Selection Sort algorithm:

1. Start from the first element and assumes it as the minimum.

2. Traverse the unsorted Part to find the Smallest element.

3. Swap the Smallest element with the first element of the unsorted part.

4. Move the boundary between Sorted and unsorted Sections one Step to the right.

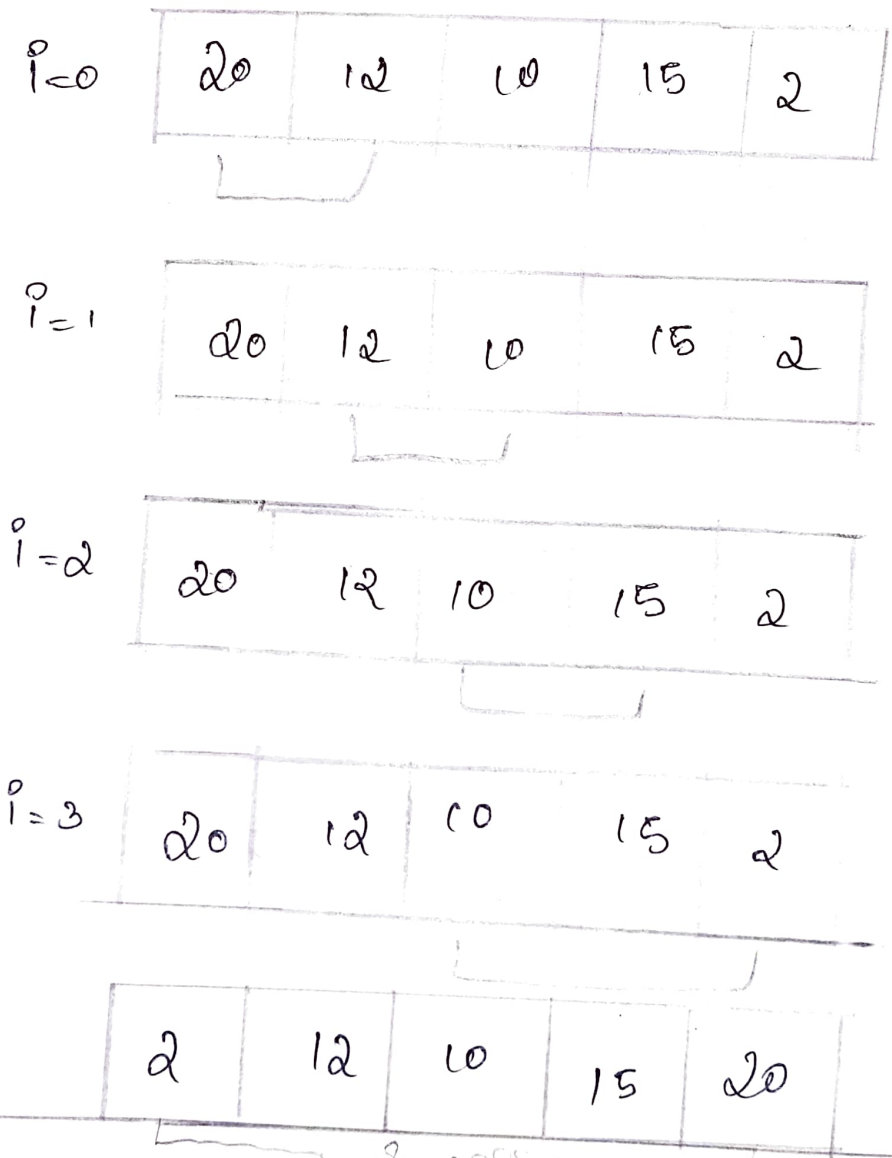
5. Repeat the Process until the entire array is Sorted.

Algorithm for Selection Sort:

```
def Selection Sort (arr):
    n = len(arr)
    for i in range (n-1):
        min index = i
```

```
for j in range (i+1, n):
    if arr[j] < arr[min_index]:
        min_index = j
    arr[i], arr[min_index] = arr[min_index], arr[i]
arr = [64, 25, 12, 22, 11]
Print ("Original array", arr)
Selection Sort (arr)
Print ("Sorted array", arr)
```

Diagram representation of selection sort:



Time Complexity:

* Base Case: $O(n^2)$ Even if already sorted all elements are compared.
* Average Case: $O(n^2)$
* Worst Case: $O(n^2)$ occures when the array is sorted in reverse order.

Space Complexity:

$O(1)$ is selection sort in an in-place sorting algorithm. and does not require extra memory.

Applications of Selection sort:

* useful when memory space is limited
* used in cases where swapping cost is minimal.
* Suitable for small datasets or when simplicity is Prefered over efficiency.

Comparison of Sorting algorithms:

Sorting algorithms Play a Crucial role in data structures and algorithms. by arranging elements in a specific order. Different sorting algorithms have varying time complexities.

Space Complexities and Use Cases

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stable?	In-Place?	Best use case.
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes	Small datasets, teaching purpose.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes	Small data sets, when swaps are costly.
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes	Nearly sorted dataset, small dataset
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No	large data sets, linked list.
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (Best), $O(n)$	No	Yes	General Purpose efficient for large dataset.

Choosing the right sorting algorithm:

Scenario	Recommended algorithm.
Small dataset ($n < 100$)	Insertion, Selection, Bubble Sort
large dataset ($n > 100$)	merge, Quick, heap sort.
Nearly sorted	Insertion Sort
General Purpose	Quick, merge Sort
In place sorting	Quick, Selection Sort.