**UNIT IV – DIVIDE & CONQUER, GREEDY AND DYNAMIC PROGRAMMING**

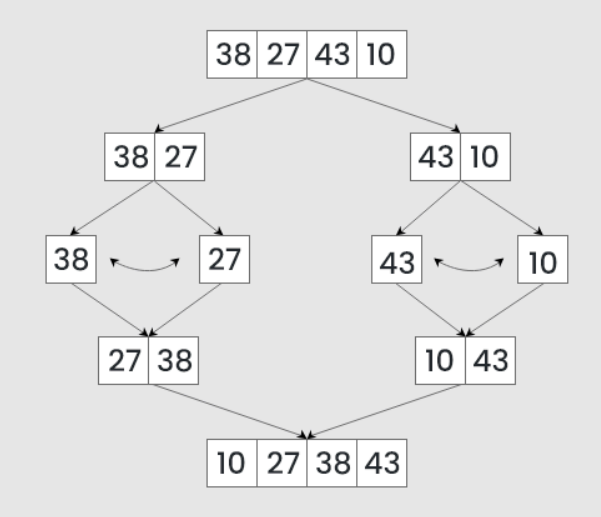
Divide and conquer: Merge sort - Quick sort - Binary search. Greedy method: Knapsack problem- Job sequencing with deadlines - Minimum cost spanning tree – Single source shortest path. dynamic programming: All pair shortest path - Knapsack problem – Traveling salesman problem - Flow shop scheduling.

---

## DIVIDE AND CONQUER: MERGE SORT

Merge sort is a sorting algorithm that 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.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



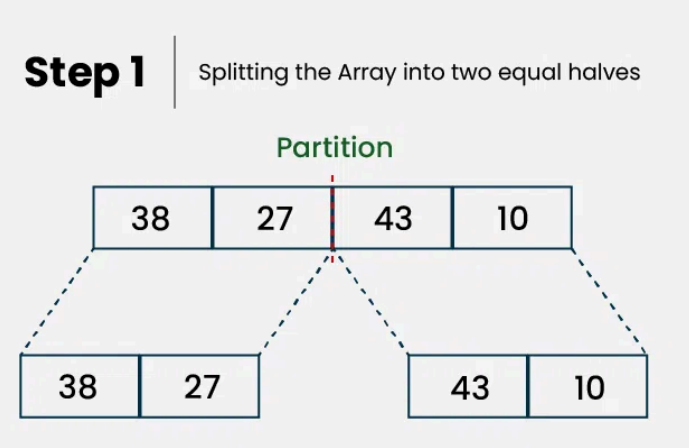

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach to sort a given array of elements.

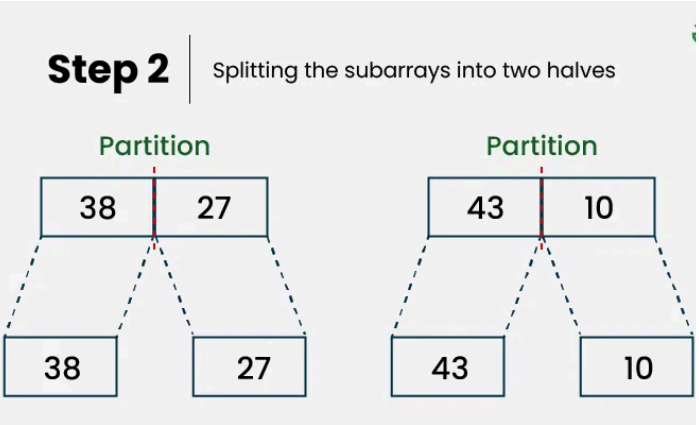
**Working of Merge sort:**

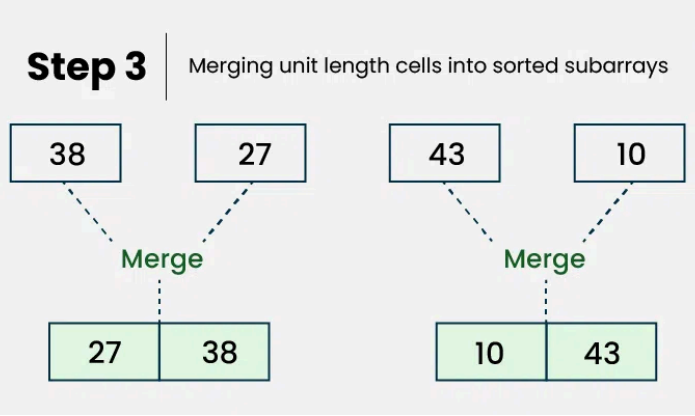
- **Divide**: Divide the list or array recursively into two halves until it can no longer be divided.
- **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
- **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

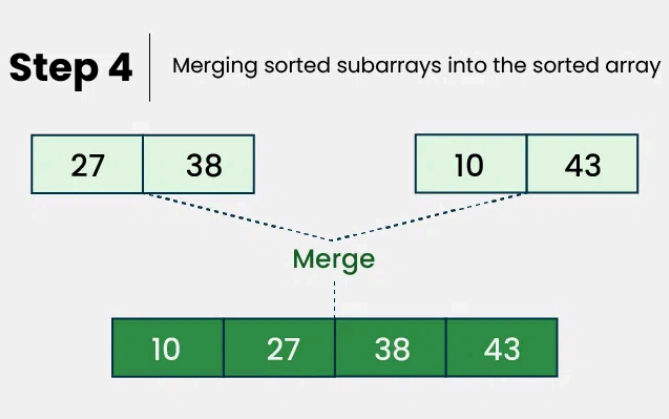
**Illustration of Merge Sort:**

Let’s sort the array or list [38, 27, 43, 10] using Merge Sort






**Divide:**

[38, 27, 43, 10] is divided into [38, 27 ] and [43, 10] .

[38, 27] is divided into [38] and [27] .

[43, 10] is divided into [43] and [10] .

**Conquer:**

[38] is already sorted.

[27] is already sorted.

[43] is already sorted.

[10] is already sorted.

**Merge:**

Merge [38] and [27] to get [27, 38] .

Merge [43] and [10] to get [10,43] .

Merge [27, 38] and [10,43] to get the final sorted list [10, 27, 38, 43]

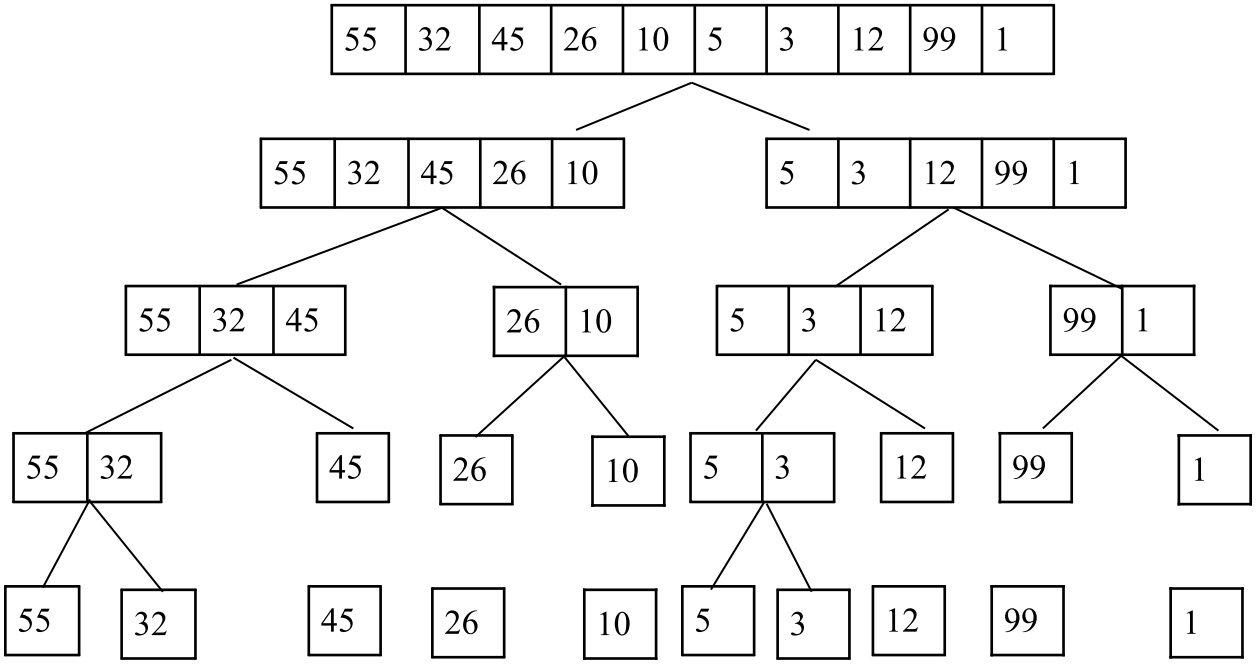
Therefore, the sorted list is **[10, 27, 38, 43]** .

**Implementation:**

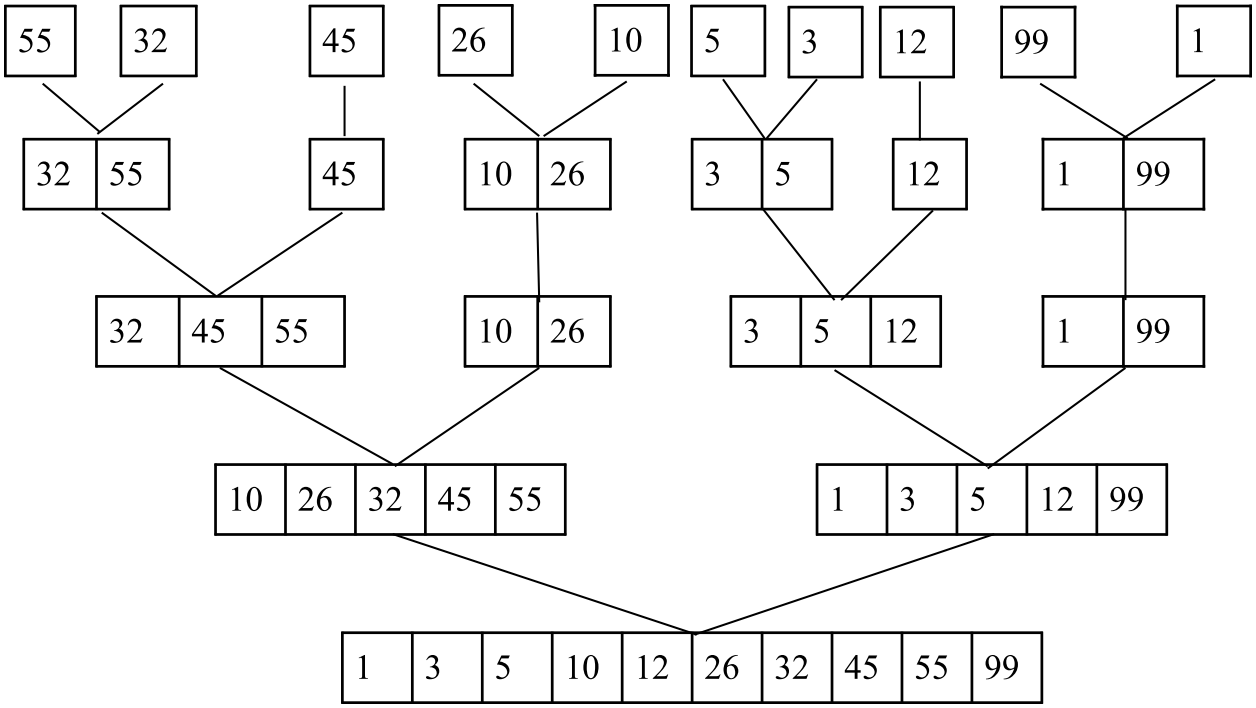
```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
```

```
	while (i < n1 && j < n2)
	{
		if (L[i] <= R[j])
		{
			arr[k] = L[i];
			i++;
		}
		else
		{
			arr[k] = R[j];
			j++;
		}
		k++;
	}
	while (i < n1)
	{
		arr[k] = L[i];
		i++;
		k++;
	}
	while (j < n2)
	{
		arr[k] = R[j];
		j++;
		k++;
	}
}
void mergeSort(int arr[], int l, int r)
{
	if (l < r)
	{
		int m = l + (r - l) / 2;
		mergeSort(arr, l, m);
		mergeSort(arr, m + 1, r);
		merge(arr, l, m, r);
	}
}
```

Example: Sort the elements 55, 32, 45, 26, 10, 5, 3, 12, 99, 1 using merge sort




**Merging**




**Complexity Analysis of Merge Sort:**

**Time Complexity:**

Best Case: O(n log n), When the array is already sorted or nearly sorted.

Average Case: O(n log n), When the array is randomly ordered.

Worst Case: O(n log n), When the array is sorted in reverse order.

**Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

---

## DIVIDE AND CONQUER: QUICK SORT

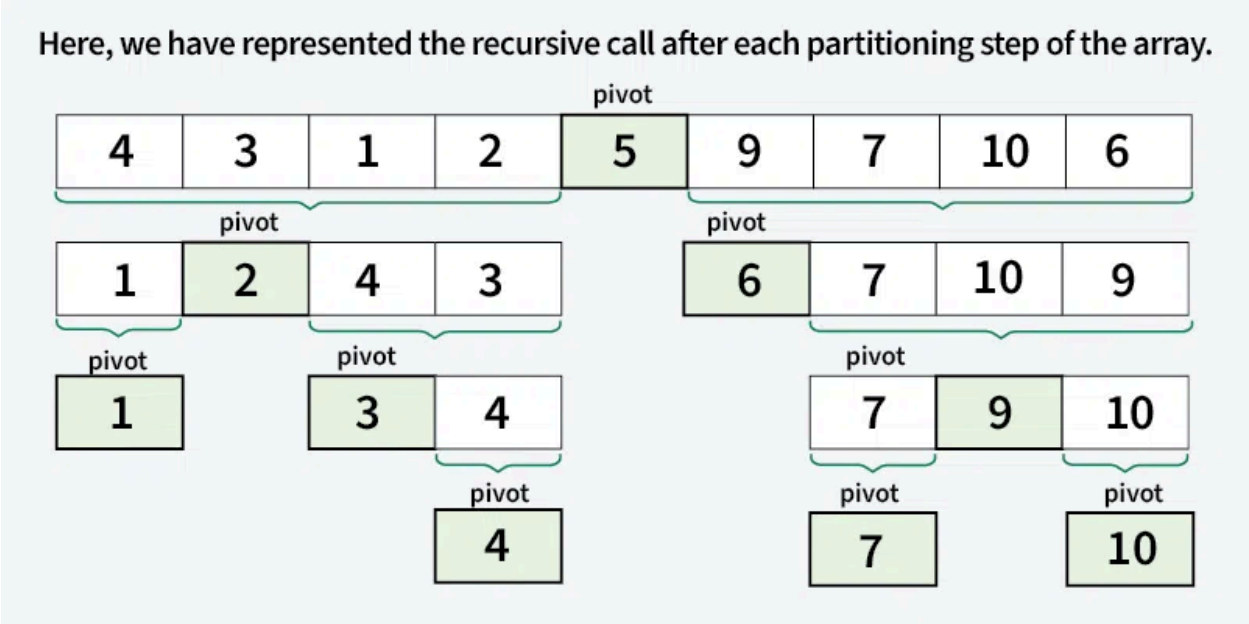
QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

### Working of QuickSort Algorithm

QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

- **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
- **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
- **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.




### Choice of Pivot

There are many different choices for picking pivots.

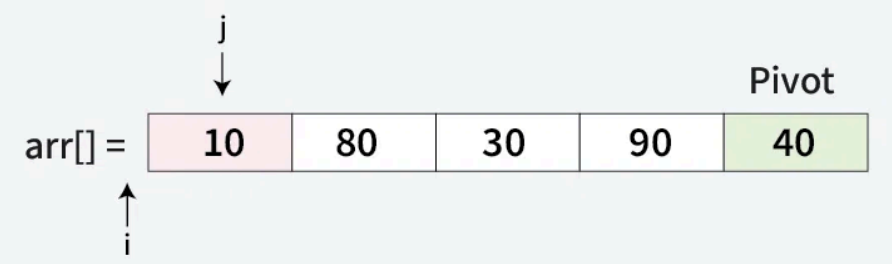
- **Always pick the first (or last) element as a pivot.** The below implementation picks the last element as pivot. The problem with this approach is it ends up in the worst case when the array is already sorted.
- **Pick a random element as a pivot.** This is a preferred approach because it does not have a pattern for which the worst case happens.
- **Pick the median element is pivot.** This is an ideal approach in terms of time complexity as we can find the median in linear time and the partition function will always divide the input array into two halves. But it is low on average as median finding has high constants.

**Partition Algorithm**

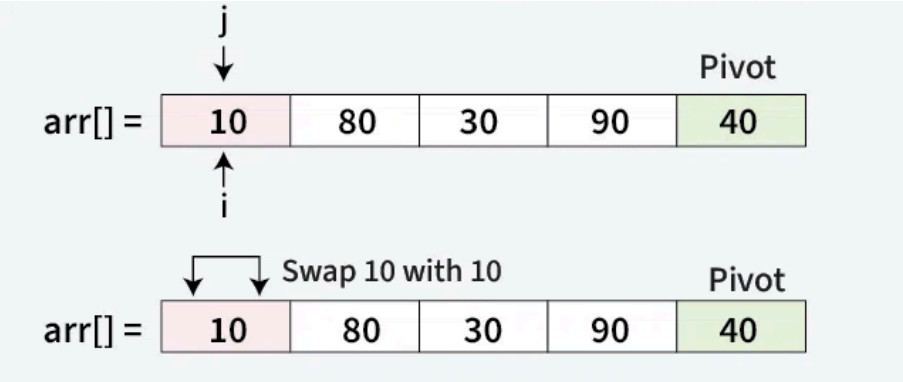
The key process in quickSort is a partition(). There are three common algorithms to partition. All these algorithms have O(n) time complexity.

- **Naive Partition:** Here we create a copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to the original array. This requires O(n) extra space.
- **Lomuto Partition:** We have used this partition in this article. This is a simple algorithm, we keep track of the index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.
- **Hoare's Partition:** This is the fastest of all. Here we traverse array from both sides and keep swapping greater elements on left with smaller on right while the array is not partitioned.

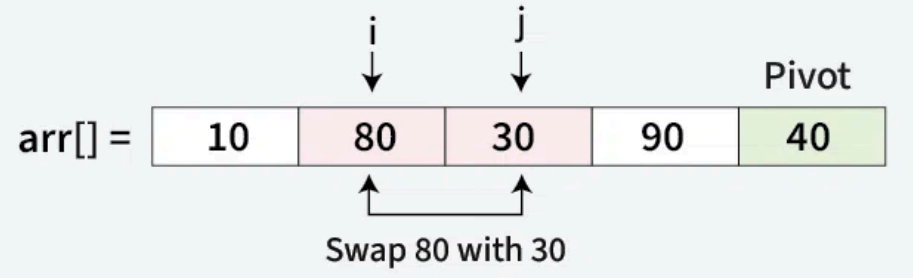
**Step 1:** Pivot selection: The last element arr[4] = 40 is chosen as the pivot. Initial pointers i = -1 and j = 0



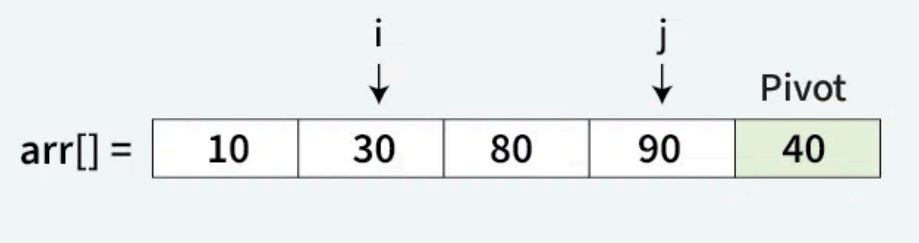

**Step 2:** Since arr[j] < pivot (i.e) 10 < 40, Increment j by 1



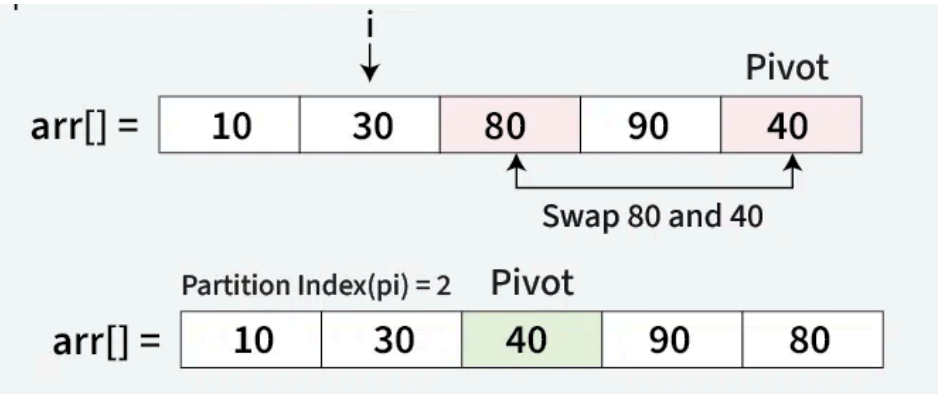

**Step 3:** Since, arr[j] < pivot (i.e.) 30 < 40, Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1




**Step 4:** Since arr[j] > pivot i.e.) 90 < 40, No need to swap and increment j by 1




**Step 5:** Since traversal of j has ended, Now move the pivot to its correct position, Swap arr[i+1] = arr[2] with arr[4] = 40.



Now the elements present on the left side of the pivot are less than 40 and elements present on the right side of pivot are greater than 40. Recursively follow the same steps to sort the arrays on the left side and right side of pivot.

**Illustration of QuickSort Algorithm**

```
void swap(int* a, int* b);
int partition(int arr[], int low, int high)
{
   int pivot = arr[high];
   int i = low - 1;
   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j] < pivot)
      {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}
void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
int main()
{
```

```
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

**Time Complexity:**

- Best Case: ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- Average Case ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- Worst Case: ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

**Auxiliary Space:** O(n), due to recursive call stack

---