✱ ✱ Merge Sort:

- It's another sorting algorithm that uses divide & conquer approach.

→ It divides a Array into two parts (1)

→ Recursion call sorts the two parts. (2) Recursive

→ Both arrays after sorted will be merged to get the sorted Array ✓ (3)

Eg: [8, 3, 4], [12, 5, 6]

3, 4, 8 5, 6, 12

→ After sorted, merge

⇒ [3, 4, 5, 6, 8, 12] ✓

you compare the sorted sub-Arrays and create a new sorted Array.

merging is done by comparing two arrays.

Pseudo:

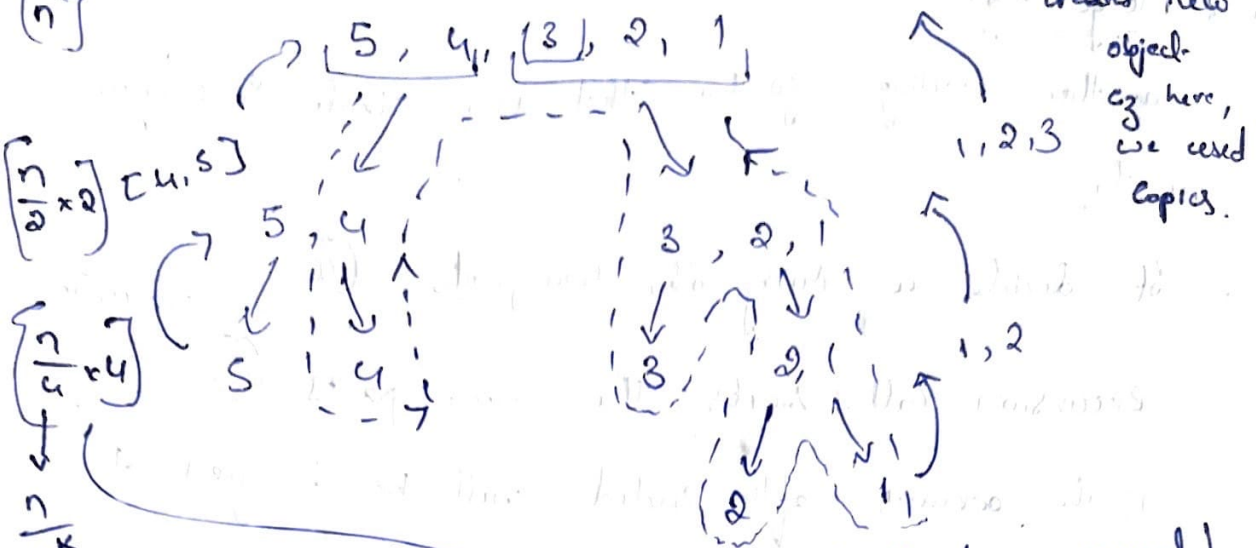
This is for method where we make copies of the Array. ✱

```
mergesort(arr){
    if(Arr.length == 1){
        return Arr;
    }
    mid = length/2;
    subArr1 = mergesort(copy of Array(arr, range till mid))
    " 2 = " " (arr, range from mid)

    return merge(SA1, SA2);
}

merge(arr, left, Right){
    // merge the Arrays }
```

(7)



$\rightarrow k = \log_2 N$ (At every level, N elements are merged).

Time Complexity: $O(N * \log_2 N)$ //

Space / Auxilary : $O(N)$ //

merge Pseudo Code: // Debug in IDE for better understanding

```
func (Arr1, Arr2) {

    final Arr = new [Arr1 + Arr2];

// now, we need take two variables for sake of comp.

// 1 variable for pointer for new array.

// check till any of them run out of bound.

    while (i < Arr1.length && j < arr.length) {

        // compare to check if i or j, which is smaller.

        if [arr1[i] < arr2[j]) {

            FArr = arr1[i];

            i++;

        } else {

            FArr = Arr2[j];

            j++;

        }

        k++ ——→ // either case, k moves up

    }
```

// when loop terminated but a array is not completed

// Add all rem. elements.

```
while (i < Arr1.length) {
        FArr[k] = first[i];
        i++;
        k++;
}
// we do the same for another Array
while (j < Arr2.length) {
        // Same as Above
}
// Only one of Above two will execute or be possible
// The end, So we return final Array.
```

2nd method: Without creating copies of Arrays

→ we just pass the indexes of the Array.

→ manipulate the range using indexes.

Aka. Inplace method //.

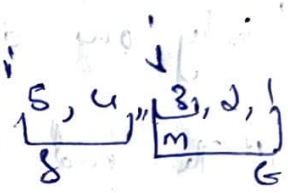
```
func(Arr, s, m, end){
        // }
```

we'll need to create a new array for getting final sorted array

Pseudo Code:

```
mergeSort(arr, s, e) {
    // base
    if (e - s == 1) {
        ret;
    }
    mid = (s + e)/2
    mergeSort(arr, s, mid);
    //    (arr, mid, e);
    mergeInplace(arr, s, m, e);
}

mergeInplace(arr, s, m, e) {
    mix = new [e-s];
    // variables for compare & sort
    i, j, k;
    while (i < m & j < e) {
        mix[k] =
        // Same as 1st method
        little modification
    }
    // Add the elments to mix
    for (l = 0; l < mix.length; l++) {
        arr[s+l] = mix[l] //.
    }
}
```



// Explain: To modify org. Array

Test it // Run & Debug it.

★★★ Quick Sort (V imp. for interviews)

- Working → It's a sorting Algorithm

we take a pivot:

↓

★ The what, why, how :

Totally Random ← • Choose any element as pivot

→ After 1st pass, all elements < p will be on left side of pivot & the elements > p lie on right side.

Eg: 5, (4), 3, 2, 1

↓ Pivot (Totally random)

⇒ 3, 2, 1, 4, 5 // Pivot in right place & Above condition is met, irrespective of the fact that if they are sorted or not.

↓ right index

- This is where, recursion comes in.
- After every pass, the pivot is put in its right place.

★ In merge sort, even if array was sorted, it'll go till the very end and check. Quick sort won't do that.

→ Quick sort is just more efficient when compared to merge.

★ How to select a pivot and put it at it's correct position.

Arr [5, (4), 3, 2, 1]

s e p

we need to check & see that elements on left of pivot are smaller & greater are on the right.

- we check using start & end, if cond. violated, we swap them.

outer loop (s < e)

```
while (A[s] < p) {
        s++
} // end of it start > p.

while (n[e] > p) {
        e--;
} // violation means end ≤ p
```

The whole idea is to put the pivot in it's place.

→ what will the recursion call be?

Eg: Arr = 5, 4, 3, 2, 1 (s at 5, e at 1, p = 4)

After 1 pass,

low   s,e   high

3, 2, 1, 4, 5 (p = 4)

→ recursion calls

now, Arr will be divided from low till end

Arr divided from s to High

4 variables: s, e, low, high

left side = low, end | right side = start, high

→ low & high are imp., tell us which part we work on.

* How to pick pivot?

(i) Random (ii) Corner elements (iii) middle elements

Complexity comparison of pivot positions:

Arr = {s . . . p . . . e} — T(K), p, T(N-K-1)

normal case

$T(N) = T(K) + T(N-K-1) + O(N)$

worst case: Picking corner elements. (when K=0)

Because now instead of divide by 2 or something, the array is only reduced by (n-1)

$T(N) = T(0) + T(N-1) + O(N)$

$\Rightarrow T(N) = T(N-1) + O(N)$

worst $\Rightarrow O(N^2)$ //.

Best Case: middle element (value $k = N/2$)

$T(N) = T(\frac{N}{2}) + T(\frac{N}{2}) + O(N)$

$\Rightarrow 2\left[T\left(\frac{N}{2}\right)\right] + O(N)$

Best $\Rightarrow O(N \cdot \log N)$ //. (Same as merge Sort).

Note:
- QS is not stable
- in-place MS is taking $O(N)$ extra space
- MS is better for linked lists due to memory allocation (not continuous).
- Hybrid Sorting Algos (Tim Sort) → uses both MS & IS
  - ↳ Internal Sorting algo, Available in python in Java as collections, C++ as STL.

** Sorting can be done in any own way, try your own.