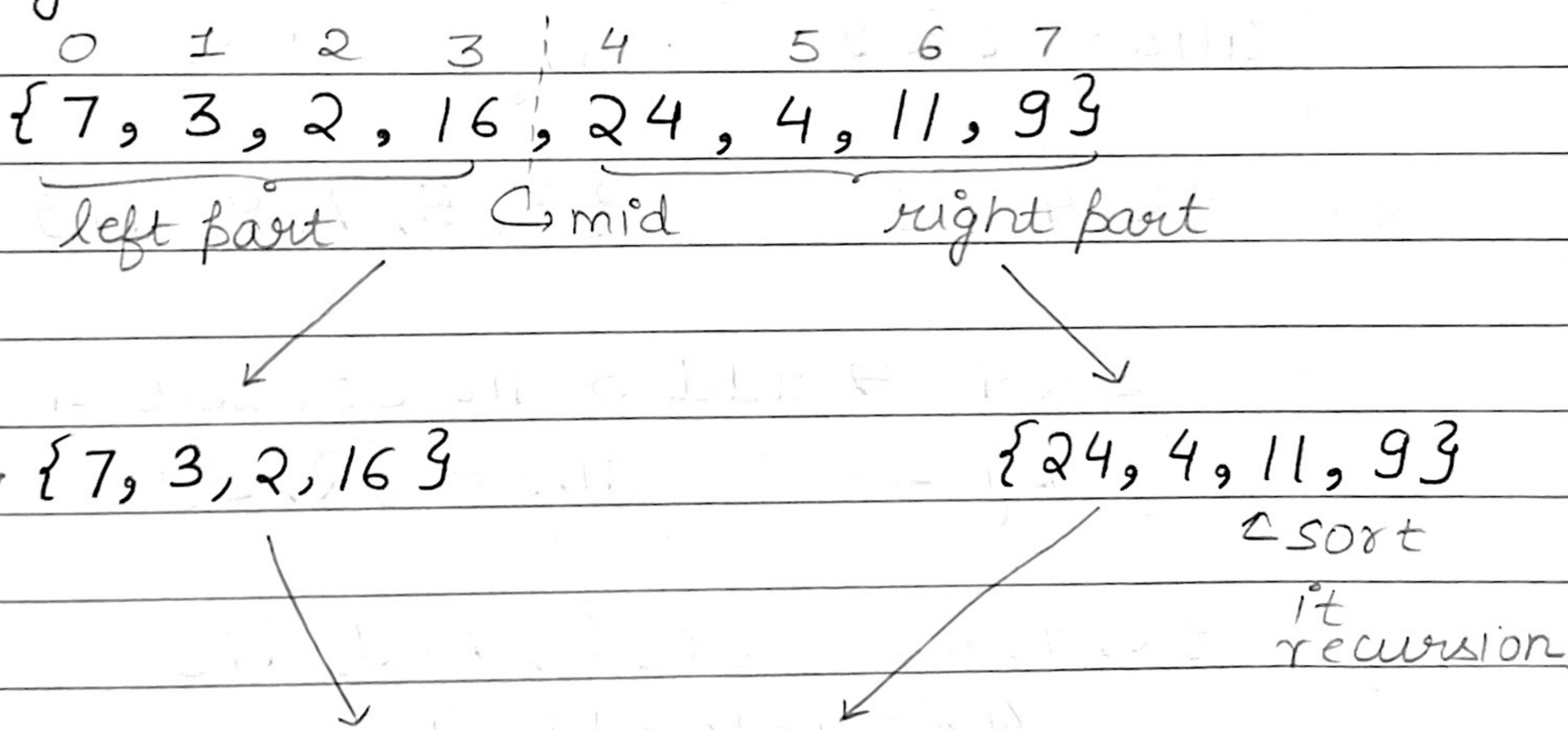
17/03/2023

Merge Sort

We will be given an array & we have to sort it in the increasing order. This is based on divide & conquer.



Divide ⇒ Divide the array
Conquer ⇒ Sort the arrays.

We know everything in this question like finding mid, send recursive calls etc. but we don't

know how to merge the arrays. So first let's solve the question of merge 2 sorted arrays.

Q → Merge 2 sorted arrays

array 1 → {2, 4, 6}
array 2 → {3, 5, 7, 9, 11}
ans = ?

Dry run ⇒ By 2 pointer approach

1) {2, 4, 6} {3, 5, 7, 9, 11}
i j

2 < 3 ⇒ add 2 in sorted ans & increase i by 1.
ans = {2}

2) {2, 4, 6} {3, 5, 7, 9, 11}
i j

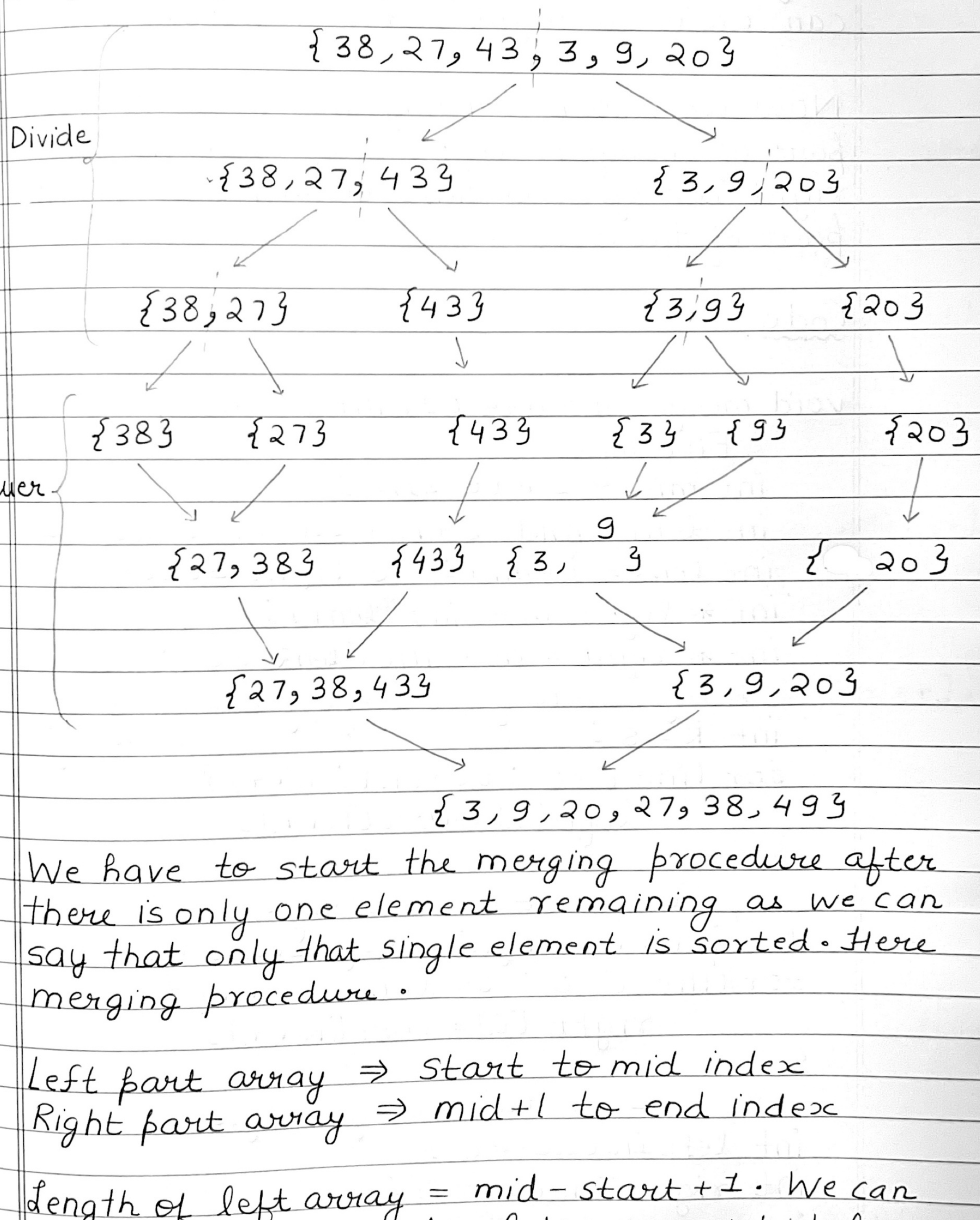
3 < 4 ⇒ add 3 in sorted ans & increase j by 1. ans = {2, 3}

3) {2, 4, 6} {3, 5, 7, 9, 11}
i j

4 < 5 ⇒ add 4 in sorted ans & increase i by 1.
ans = {2, 3, 4}

Now we will continue this until any one array finishes. If any of the array finishes simply copy the values of other array to the ans array. Hence we have merged the 2 sorted arrays.

Dry run for merge sort



We have to start the merging procedure after there is only one element remaining as we can say that only that single element is sorted. Here merging procedure.

Left part array ⇒ start to mid index
Right part array ⇒ mid+1 to end index

Length of left array = mid - start + 1. We can make a new array for left array which has size mid - start + 1.

length of right array = end - mid & hence we can create a right array of length end - mid.

Now we will be copying the values of left part of array to left array & similarly right array will have elements of right part of the array.

Code

```
void merge (int arr [], int s, int e) {
    //Find mid
    int mid = s + (e-s)/2;
    int len1 = mid - s + 1;   } Length of left &
    int len2 = e - mid;       } right subarrays.
    int * left = new int [len1];   } Dyanamic
    int * right = new int [len2];  } allocation.
    //Array starts from s. Copy values to left
    int k = s;                          array.
    for (int i = 0; i < len1; i++) {
        left [i] = arr [k++];
    }
    //Copy values to right array.
    k = mid + 1; //right array start index.
    for (int i = 0; i < len2; i++) {
        right [i] = arr [k++];
    }
    //Merge 2 sorted arrays logic
    int leftIndex = 0;
    int rightIndex = 0;
    int mainArrayIndex = s;
```

```
// Run loop until any of the array finishes
    while (leftIndex < len1 && rightIndex < len2){
    // Left array has smaller value
        if (left [leftIndex] < right [rightIndex]) {
                arr [mainArrayIndex++] = left [leftIndex++];
    }
    // Right array has smaller value.
        else {
        arr [mainArrayIndex++] = right[rightIndex++];
        }
}
// Copy left array values if right array has finished.
while (leftIndex < len1) {
        arr [mainArrayIndex++] = left [leftIndex ++];
 }
// Copy right array value if left array has finished
while (rightIndex < len2) {
    arr [mainArrayIndex++] = right [rightIndex ++];
}
    //Deallocation of left & right array can be done
}
// Merge sort function
void mergeSort (int arr [], int s, int e){
            // s == e → single element, s > e → invalid
        if (s >= e)
            return;
//Divide the array
int mid = s + (e-s)/2;
// Recursive call for left array
mergeSort (arr, s, mid);
//Recursive call for right array
mergeSort (arr, mid+1, e);
```

```
//Merge 2 sorted arrays.
merge (arr, s, e);
}
```

Note → `int * left = new int [5];`

new returns the address. Array of size 5 will be created & left is pointing to the address. This is basically the dyanamic allocation of array in memory.

Logic of length of left & right array

0	1	2	3	4	5
{38,	27,	43,	3,	9,	20}
			↑		
			mid		

Length of left array = $\underset{\uparrow}{2} - \underset{\uparrow}{0} + 1 = 3$
(2 = mid, 0 = s)

Length of right array = $\underset{\uparrow}{5} - \underset{\uparrow}{2} = 3$
(5 = e, 2 = mid)

Time complexity of merge sort
We need to find the recursive relation of the code.

$$T(n) = \underset{\uparrow}{k} + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \underset{\uparrow}{O(n)}$$

k ↑ Base case; $T\left(\frac{n}{2}\right)$ ↑ Left part; $T\left(\frac{n}{2}\right)$ ↑ Right part; $O(n)$ ↑ merge

$$T(n) = 2T\left(\frac{n}{2}\right) + k + n * k$$

↳ ignore in front of $n * k$.

$$T(n) = 2T\left(\frac{n}{2}\right) + n * k$$

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + \frac{n}{2} * k$ } $\times 2$

$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + \frac{n}{4} * k$ } $\times 4$

$T\left(\frac{n}{8}\right) = T\left(\frac{n}{16}\right) + \frac{n}{8} * k$ } $\times 8$

⋮

$T(1) = k$

↳ constant time to sort array of size 1.

→ Check $a = \log_2 n$

$T(n) = (a-1) \times n \times k + k$

$T(n) = (\log n - 1)\, n \times k + k$

$T(n) = k \times n \times \log n - nk$

$T(n) = nk \log n \rightarrow O(n \log n)$

	n	
	↓	
	$\frac{n}{2}$	
$a-1$	↓	a
	$\frac{n}{4}$	
	⋮	
	1	

$\frac{n}{2^a} = 1$

$n = 2^a$

$a = \log n$ } used in above equation.

Note → Merge sort will have $O(n \log n)$ time complexity in best, worst & average case.