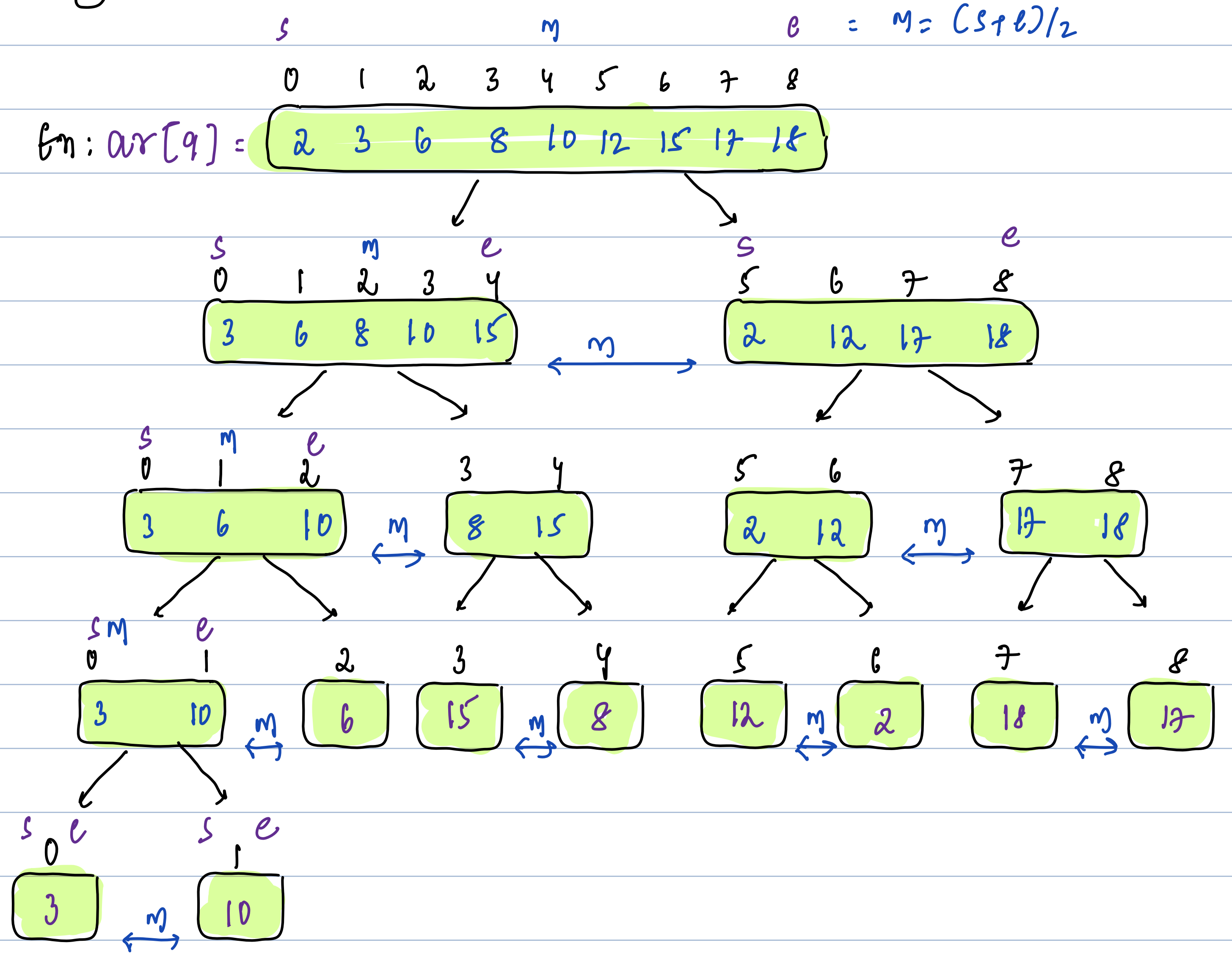
Todays Content

1. Merge Sort : TC & SC
2. Inversion Count

Merge Sort:

Keep dividing arr[] in 2 halfs, till it contains 1 element & Merge.

Tracing:



Ass: Given ar[] sort it from s...e & return nothing.

→ merge space

→ Stack Size

```
void mergeSort(int ar[], int s, int e){ TC: O(N log_2 N)  SC: O(N + log_2 N)
    if(s==e){return;}
    int m = (s+e)/2;
    mergeSort(ar, s, m);
    mergeSort(ar, m+1, e);
    merge(ar, s, m, e); # Merge both subarray [s..m] & [m+1..e]
}
```

ar[] : {.. s s+1 ... m | m+1 ... e ... }

tmp[ ]

ar[] : {.. s s+1 ... m | m+1 ... e ... }

```
void merge(int ar[], int s, int m, int e){
    int c[e-s+1];
    int p1=s, p2=m+1, p3=0;

    while(p1<=m && p2<=e){ # 1st sub: [s..m]  2nd sub= [m+1..e]
        if(A[p1] < A[p2]){ c[p3]=A[p1]; p3++; p1++;}
        else{ c[p3] = A[p2]; p3++; p2++;}
    }

    while(p1<=m){
        c[p3]=A[p1]; p3++; p1++;
    }
    while(p2<=e){
        c[p3] = A[p2]; p3++; p2++;
    }

    for(int i=s; i<=e; i++){
        A[i]= c[i-s];
    }
}
```

Recursive Relation:

$$T(N) = 2T(N/2) + O(N) \qquad T(1) = 1.$$

$$T(N/2) = 2T(N/4) + N/2$$

$$= 2[2T(N/4) + N/2] + N$$

$$= 4T(N/4) + N + N$$

$$= 4T(N/4) + 2N$$

$$T(N/4) = 2T(N/8) + N/4$$

$$= 4[2T(N/8) + N/4] + 2N$$

$$= 8T(N/8) + N + 2N$$

$$= 8T(N/8) + 3N$$

Generalized

$$T(N) = 2^k T(N/2^k) + kN \qquad T(1) = 1$$

$$N/2^k = 1, \quad N = 2^k, \quad k = \log_2^N$$

$$= N\,T(N/N) + \log_2^N * N$$

$$= N + \log_2^N * N$$

$$T(N) = O(N \log_2^N)$$

Inversion Count:

Given an ar[N] calculate no: of pairs (i, j) such that i < j && ar[i] > ar[j]

	0	1	2	3	4
Ex1: ar[5] :	10	3	8	15	6

i	j	
0 < 1		ar[0] > ar[1]
0 < 2		ar[0] > ar[2]
0 < 4		ar[0] > ar[4]
2 < 4		ar[2] > ar[4]
3 < 4		ar[3] > ar[4]

	0	1	2	3	4	5	6	7	8	9	
Ex2: ar[10]: {	10	3	8	15	6	12	2	18	8	1 }	
	6	2	3	5	2	3	1	2	1	0	= 25

Idea1:

Generate all pairs, if it's valid pair: Inc count;

```
int pairs(int[] ar){ TC: O(N^2)  SC: O(1)
    int c=0;
    for(int i=0; i<N; i++){
        for(int j=i+1; j<N; j++){
            if( ar[i] > ar[j]){
                c++;
            }
        }
    }
} return c;
```

Idea2: Uses idea mergeSort

	0	1	2	3	4	5	6	7	8	9
ar[10]:	10	3	8	15	6	12	2	18	8	1

0	1	2	3	4	5	6	7	8	9
10	3	8	15	6	12	2	18	8	1

sort left part — sort right part

indice left & indices right

0	1	2	3	4	5	6	7	8	9
~~3~~	~~6~~	~~8~~	~~10~~	~~15~~	~~1~~	~~2~~	~~8~~	~~12~~	~~18~~

P_1 P_2

+5	5	0	0	0	+2	+0	+1	+0	+0

= Merge 13 pairs.

tmp[ ] =	1	2	3	6	8	8	10	12	15	18

P_3

#obs1: if(ar[P_1] <= ar[P_2]) {
  keep ar[P_1]
}
else{ # ar[P_1] > ar[P_2]
  keep ar[P_2] ; c = c + No: of Elements in left
}

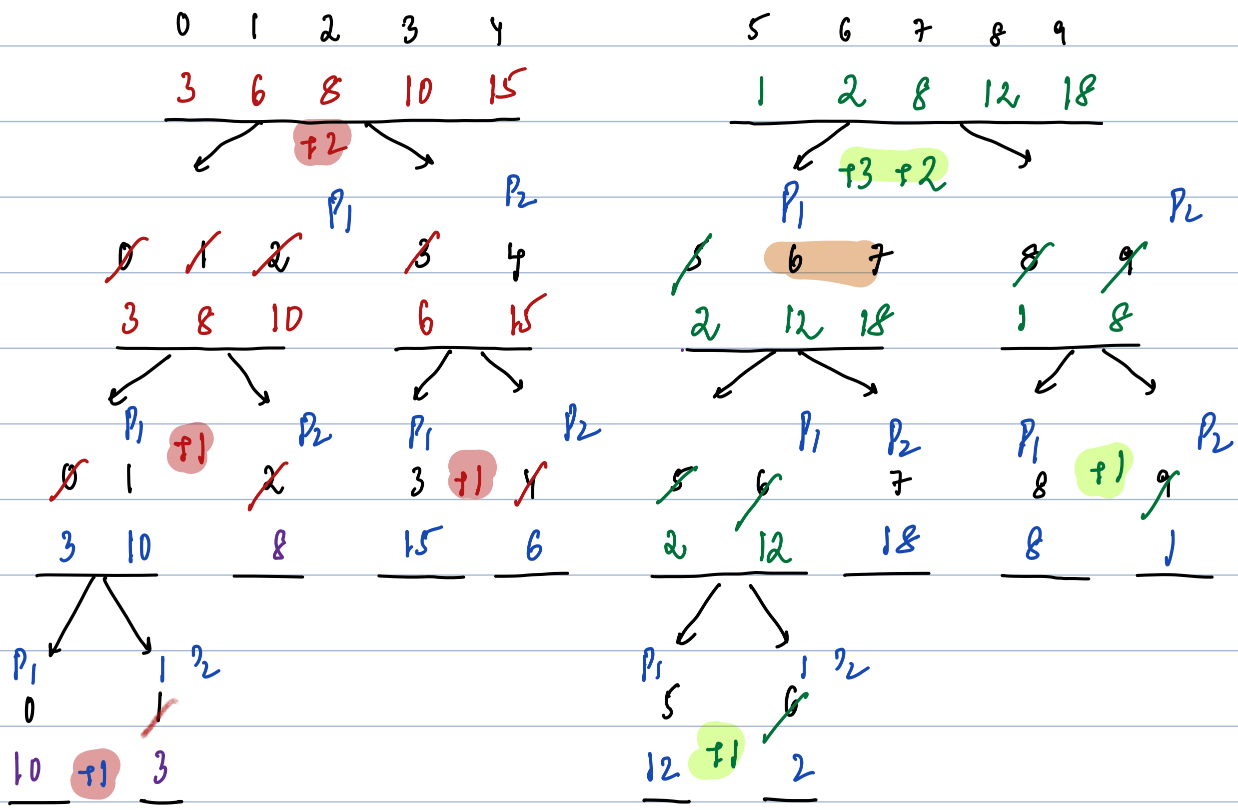
#obs2: Total pairs =
Total left pairs
Total right pairs
Total merge pairs

Total = left + right + m = 5 + 7 + 13 = 25.

Continue:

0	1	2	3	4	5	6	7	8	9
10	3	8	15	6	12	2	18	8	1

left = 5   m = 13   right = 7



Ass: Given arr[] sort it from s ... e & return count of pairs

TC: $O(N\log_2^N)$ SC: $O(N+\log_2^N)$

```
int mergeSort(int arr[], int s, int e){ TC: O(N log2 N)  SC: O(N + log2 N)
    if(s==e){return;}
    int m = (s+e)/2;
    int lc= mergeSort(arr, s, m);
    int rc= mergeSort(arr, m+1, e);
    int mc= Merge(arr, s, m, e); # Merge both subarray [s..m] & [m+1..e]
}   return lc+rc+mc;
```

arr[] : { .. [s s+1 ... m] [m+1. — . .. e] ... }

tmp[ ]

arr[] : { .. [s s+1 ... m] [m+1. — . .. e] ... }

```
int Merge(int arr[], int s, int m, int e){
    int c[e-s+1];
    int p1=s, p2=m+1, p3=0;
    int mc=0;
    while( p1<=m && p2<=e){ # 1st sub: [s..m]  2nd sub= [m+1..e]
        if( A[p1] <= A[p2]) { c[p3]=A[p1]; p3++; p1++;}
        else{ c[p3] = A[p2]; p3++; p2++; mc=mc+ m-p1+1; }
    }

    while( p1<=m){
        c[p3]=A[p1]; p3++; p1++;
    }
    while( p2<=e){
        c[p3] = A[p2]; p3++; p2++;
    }

    for(int i=s; i<=e; i++){
        A[i]= c[i-s];
    }

    return mc;
}
```

arr[] : { s .. [p1 — .. m] m+1 .... e

```
int mc=0;
int solve(int[] ar, int N){
    mc=0; # Any global variable re-initialize before function calls.
    mergeSort(ar, 0, N-1);
    return mc;
}
```

Note: All testcases run on global variables without re-initialization, hence we reinitialize

```
void mergeSort(int ar[], int s, int e){ TC: O(N log_2 N)  SC: O(N + log_2 N)
    if(s==e){return;}
    int m = (s+e)/2;
    mergeSort(ar, s, m);
    mergeSort(ar, m+1, e);
    merge(ar, s, m, e); # Merge both subarray [s..m] & [m+1..e]
}
```

ar[] : { .. [s s+1 ... m] [m+1. _ ... e] ... }

tmp[ ]

ar[] : { .. [s s+1 ... m] [m+1. _ ... e] ... }

```
void merge(int ar[], int s, int m, int e){
    int c[e-s+1];
    int p1=s, p2=m+1, p3=0;

    while(p1<=m && p2<=e){ # 1st sub: [s..m]  2nd sub= [m+1.. e]
        if(A[p1]<=A[p2]){ c[p3]=A[p1]; p3++; p1++;}
        else{ c[p3] = A[p2]; p3++; p2++; mc=mc+ m-p1+1; }
    }
    while(p1<=m){
        c[p3]=A[p1]; p3++; p1++;
    }
    while(p2<=e){
        c[p3] = A[p2]; p3++; p2++;
    }

    for(int i=s; i<=e; i++){
        A[i] = c[i-s];
    }
}
```

ar[] : { s .. [p1_ ... m] m+1 .... e

Note: All merge pairs are added in mc;