Todays Content:

- Quick sort
- Countsort

1Q] Given ar[N] elements, re-arrange the array such that

→ ar[0] should goto correct sorted position
→ All elements < ar[0] leftside of ar[0]
→ All elements > ar[0] rightside of ar[0]

} Expected SC: O(1)

	0	1	2	3	4	5	6	7	8	9	10
ar[11] = {	10	3	8	15	6	12	2	18	7	15	14 }

Ideal:

Sort ar[]

TC: N log N

{ 2 3 6 7 8 | 10 | 12 14 15 15 18 }

(2 3 6 7 8 : < 10 ; 12 14 15 15 18 : > 10)

Idea2:

	0	1	2	3	4	5	6	7	8	9	10
{	10	3	8	~~15~~ 7	6	~~12~~ 2	~~2~~ 12	18	~~7~~ 15	15	14 }
						P_2	P_1				

	0	1	2	3	4	5	6	7	8	9	10
{	10	3	8	7	6	2	12	18	15	15	14 }

Swap ar[0] & ar[P_2]

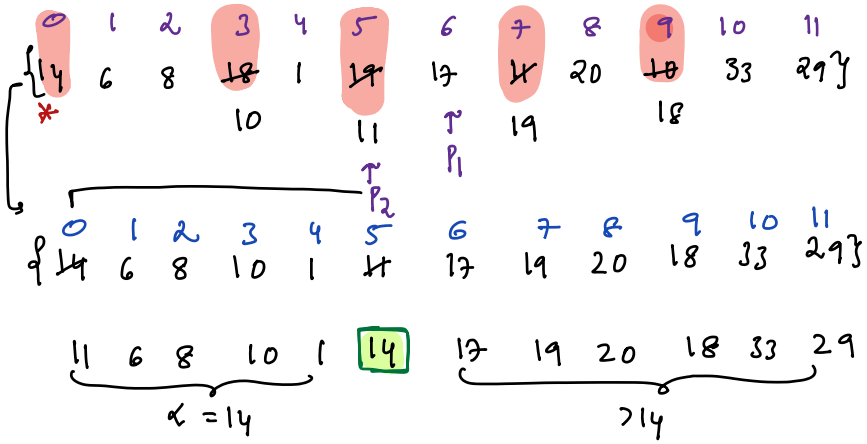
{ 2 3 8 7 6 | 10 | 12 18 15 15 14 }

(2 3 8 7 6 : < 10 ; 12 18 15 15 14 : > 10)

P_1	P_2	
1	10	
2	10	
3	10	
*	↓	
3	9	
↓	↓	
3	8	: Swap P_1++ P_2--
↓	↓	
4	7	
↓	:	
5	7	
	↓	
5	6	: Swap P_1++, P_2--
↓	↓	
6	5	: $P_1 > P_2$ break : Swap (ar[0] & ar[P_2])

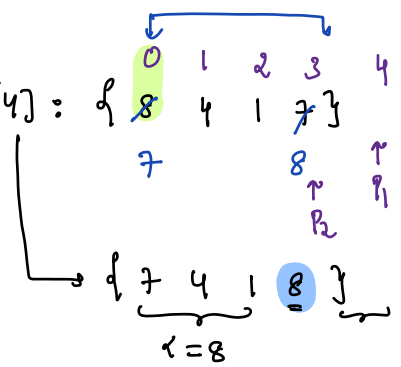
Eg1:

ar[12] = {14 6 8 18 1 14 17 11 20 18 33 29}



P1	P2	
1	11	
2	11	
3	11	
*		
3	10	
3	9	: swap (ar[3] & ar[9]) P1++, P2--
*	*	
4	8	
5	8	
*		
5	7	: swap (ar[5] & ar[7]) P1++, P2--
*	*	
6	6	
*		
6	5	: P1 > P2 break : swap (ar[0] & ar[P2])

Eg3: ar[4] : {8 4 1 7}



P1	P2	
1	3	
2	3	
3	3	
4	3	: P1 > P2 break, swap ar[0] & ar[P2]

```
void re-arrange(int ar[N]){   TC: O(N)  SC: O(1)

    // ref = ar[0]
    P1 = 1, P2 = N-1
    while( P1 <= P2 ){
        if (ar[P1] <= ar[0]){
            // P1 is correct pos
            P1 = P1+1
        }
        else if (ar[P2] > ar[0]){
            // P2 is correct pos
            P2 = P2-1
        }
        else{ // Both P1 & P2 are wrng
            swap(ar[P1], ar[P2])
            P1++, P2--
        }
    }
    swap(ar[0] & ar[P2])
}
```

20)

Given ar[N] & subarray [s, e] Re-arrange sub[s - e] such that

→ ar[s] should come to correct pos in [s e]

→ all elements <= ar[s] should left

→ all elements > ar[s] should right

→ & finally return correct pos of ar[s]

{s=2 e=7}

Ex: ar[10] = { 10 3 8 6 14 7 4 12 7 1 }

P1	P2
3	7
4	7
4	6
5	5
6	5

4 6 : swap ar[4] & ar[6] P1++, P2--

6 5 : P1 > P2 break : swap (ar[P2] & ar[s])

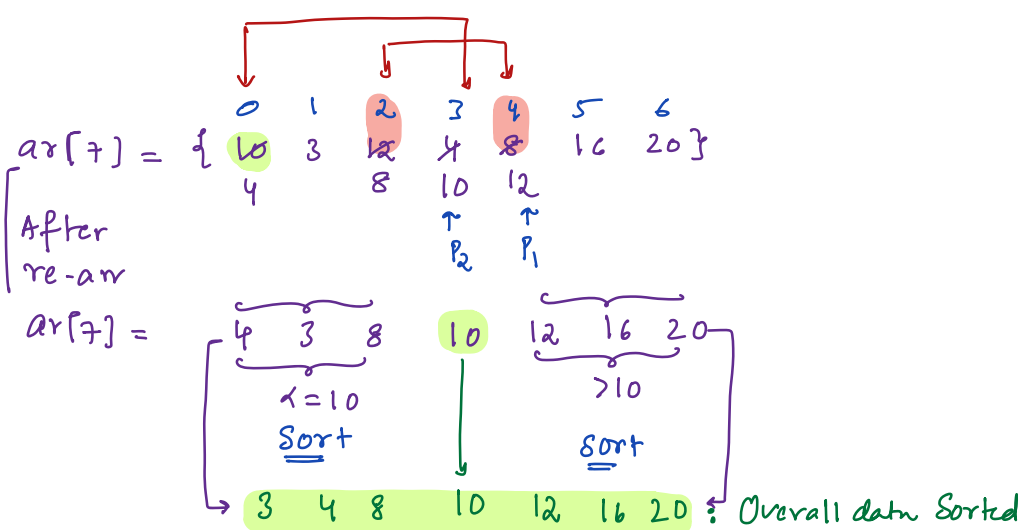
{ 10 3 8 6 14 7 4 12 7 1 }

{ 10 3 8 6 4 7 14 12 7 1 }

{ 7 6 4 8 14 12 7 1 }

<= 8 | > 8

QuickSort Idea :



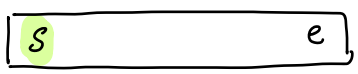
```
int re-arrange( int ar[N], int s, int e){ TC: O(N) SC: O(1)
    // ref = ar[s]
    P1 = s+1 , P2 = e
    while( P1 <= P2 ){
        if ( ar[P1] <= ar[s] ){ // P1 is corret pos
            P1 = P1+1
        }
        else if ( ar[P2] > ar[s] ){ // P2 is corret pos
            P2 = P2-1
        }
        else{ // Both P1 & P2 are wrng
            swap(ar[P1], ar[P2])
            P1++, P2--
        }
    }
    swap ( ar[s] & ar[P2])
    return P2 // correct ind position of ar[s]
}
```

10:20 → 10:30pm

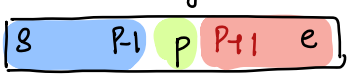
Sort entire subarray

Ass: Given subarray [s - e] in ar[] sort the subarray [s e]

```
void QuickSort ( int ar[], int s, int e){
    if(e <= s) {return}
    p = re-arrange(ar, s, e)
        // correct index pos of ar[s]
        with in subarray [s - e]
    Quicksort(ar, s, p-1)
    Quicksort (ar, p+1, e)
}
```



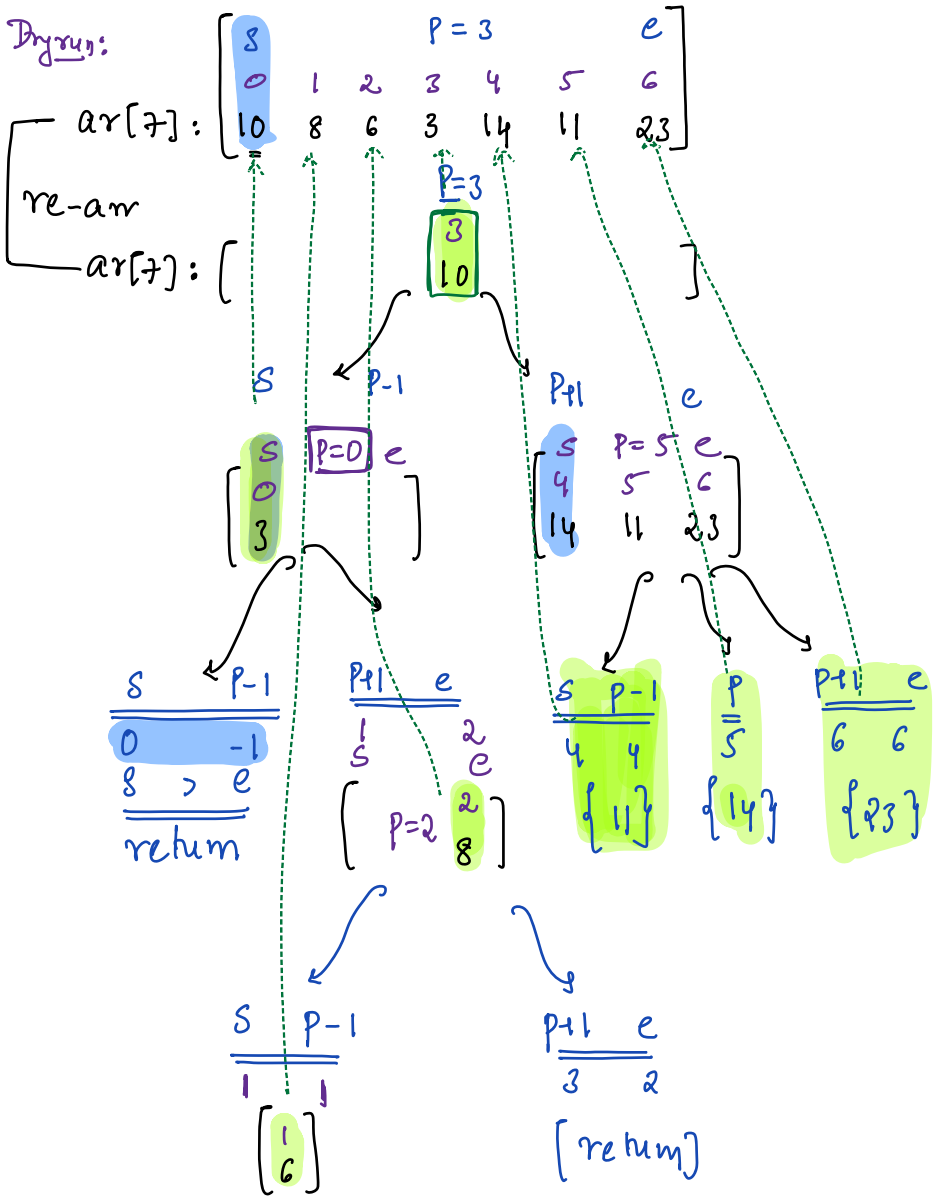
// re-arrange sub [s e]
such ar[s] goes to corr pos = p



<= ar[p] Sort

> ar[p] Sort

Sorted <= ar[p] & Sorted



ar[ ] = { 3 6 8 10 11 14 23 } sorted data

Time Complenity Quick Sort

```
void QuickSort(int ar[], int s, int e){
    if(e <= s) {return;}
    p = re-arrange(ar, s, e)
        // correct index pos of ar[s]
        with in subarray [s - e]
    Quicksort(ar, s, p-1) }
    Quicksort(ar, p+1, e) }
}
```

SC: O(1) + Recursive Call Stack Size ≈ (log N)

N

unstable : TODO

// Say we sort N elements.

: assume time taken to sort N elements is f(N)

Best Case:

: Equal divism into 2 sides

$f(N) = N + 2f(N/2)$

$f(N) = N \log N$

Avg Time

≈ O(N log N)

Worst Case:

: All N-1 elements going to 1 side

$f(N) = N + f(N-1)$

$f(N) = O(N^2)$

Inbuild :

Java → Tim Sort : mergeSort + QuickSort & 2 reference/pivot

C++ → IntroSort : QuickSort + HeapSort
↳ & future

python → Tim Sort : Insertion, Merge

→ pick k min : Selection

→ pick max in stable : Bubble

→ insert in sorted[] : Insertion

→ Sort entire data : Inbuilt sort → O(NlogN) SC: O(1)
↳ in your language of choice inbuilt method

CountSort / Freq Sort /

// Given ar[N], all elements in range [1-5] sort arr[]

	0	1	2	3	4	5	6	7	8	9
ar[] = {	3	1	5	5	2	1	3	3	2	1 }

Idea1: Sort arr[] TC: O(NlogN)

Idea2: Store all freq in hashmap

{ <3, 3>, <1, 3>, <5, 2>, <2, 2> }

Iterate from 1-5:

1 : freq[1] = 3

2 : freq[2] = 2

3 : freq[3] = 3

4 : freq[4] = 0

5 : freq[5] = 2

Output:

	0	1	2	3	4	5	6	7	8	9
{	3	x	3	3	2	x	3	3	2	x }
	1	1	1	2	2	3	3	3	5	5

// data sorted

Pseudocode:

1) Insert all ar[] → hm : O(N)

// Iterate in given data range

```
k=0
for(i=1; i<=5; i++) {
    // freq of i get from hashmap hm
    n = hm[i] // freq of i
    // If i is not there, assu n=0
    for(j=0; j<n; j++) {
        ar[k] = i, k++
    }
}
```

Q8) : Data in range [s → e] ? TC: O(N + N + e - s + 1) ⇒

→ TC: O(N + e - s + 1)

Range of data // e - s + 1 = R

→ TC: O(N + R)

SC: O(N)

Storing all arr[] ele in hashmap

1) Insert all ar[] → hm : O(N)

// Iterate in given data range

```
k=0
for(i=s ; i<=e ; i++){ → TC: N + e - s + 1
    // freq of i get from hashmap hm
    n = hm[i] // freq of i
    // If i is not there, assu n=0
    for(j=0 ; j<n ; j++){
        ar[k] = i , k++
    }
}
```

Ex: ar[10] all elements in range [s 1, e 100]

ar[10] = { 2 2 3 1 1 4 1 3 4 100 }

↳ // Insert in hashmap

hm : { 2:2 3:2 1:3 4:2 100:1 }

i	j : freq of i times	Total iterations
1	freq[1] = 3	3
2	freq[2] = 2	2
3	freq[3] = 2	2
4	freq[4] = 2	2
5	freq[5] = 0	1
6	freq[6] = 0	1
7	freq[7] = 0	1
⋮		
100	freq[100] = 1	1

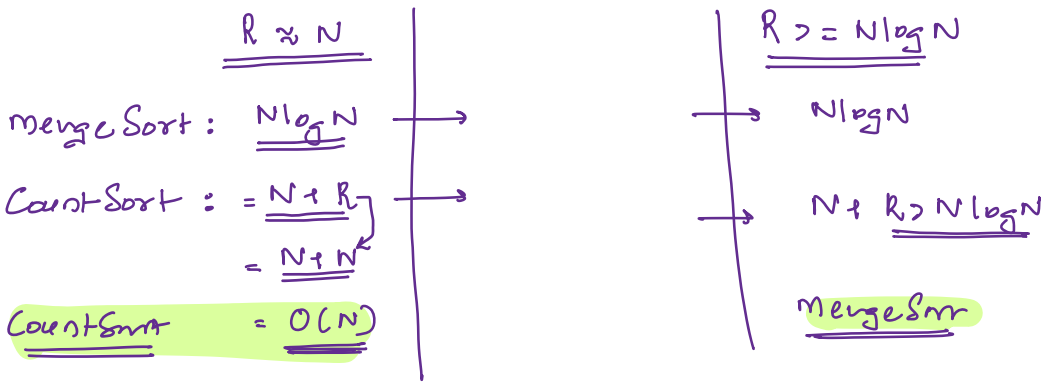
sum of all freq = N

sum of is ≈ no of elem in range ≈ [s e]

→ e - s + 1

Total Iterations = N + e - s + 1

// ar[N] : CountSort : ∃ TC: (N+R) // R = dataRange, N = no: of array elements



TODO: CountSort only works on Integers