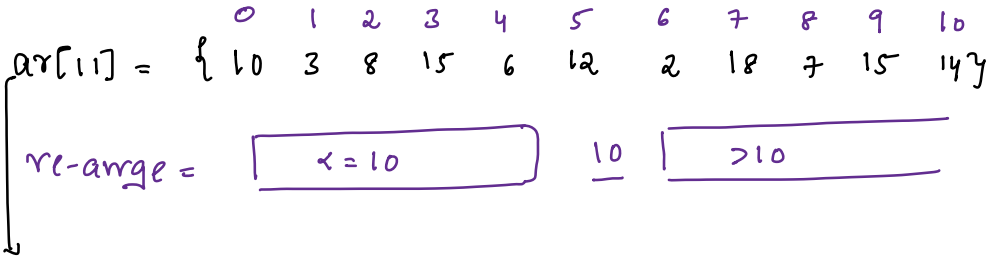
Todays Content:

- Quick sort
- Countsort
- Radix sort :

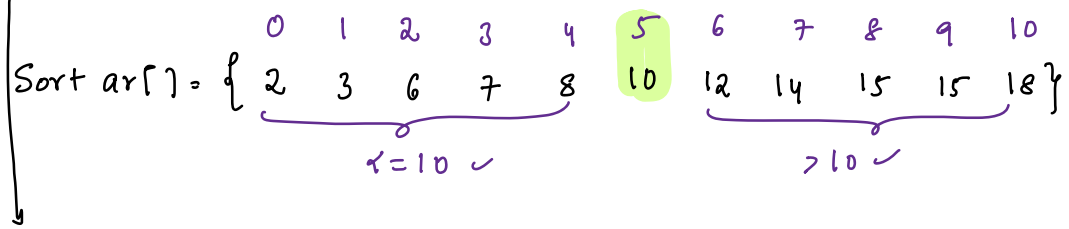
Q] Given ar[N] elements, re-arrange the array such that

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

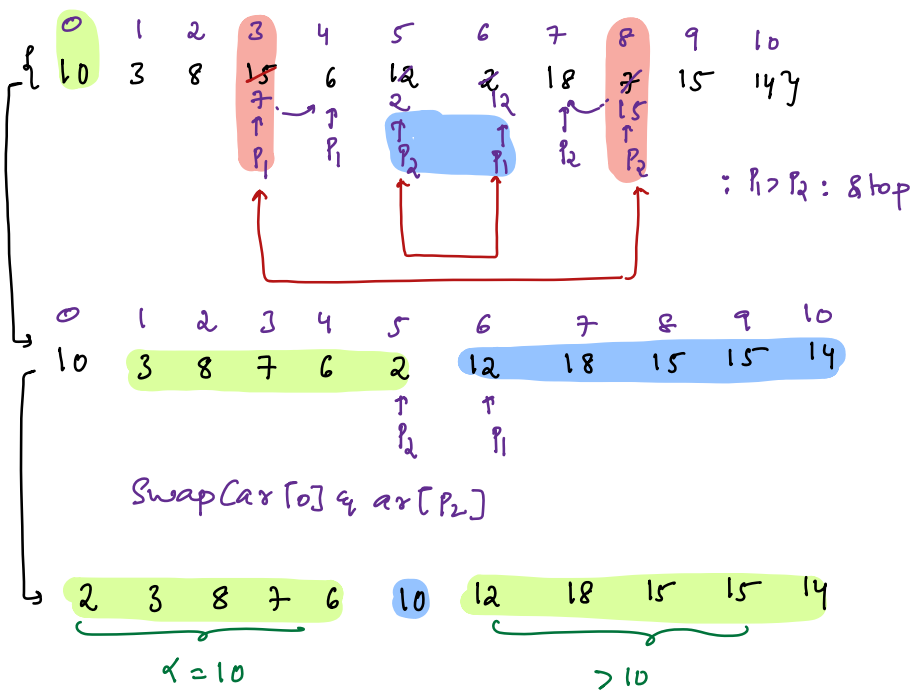
} Expected SC: O(1)



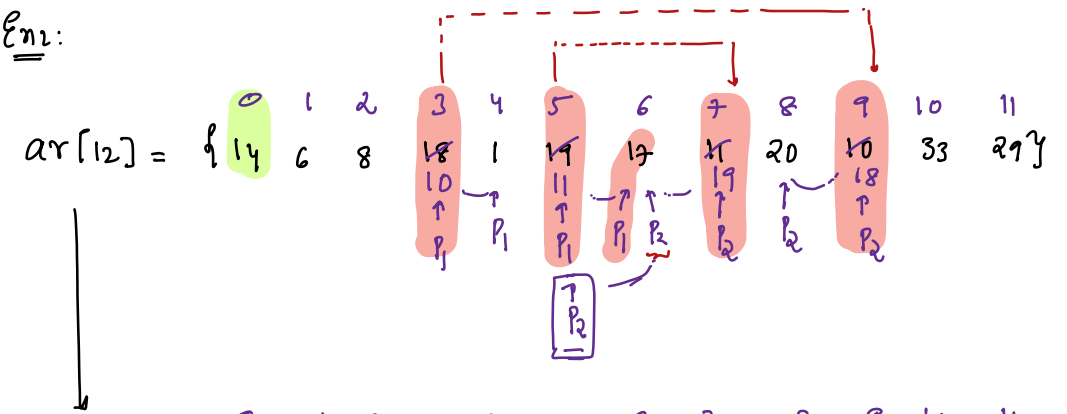
Idea1: Sort ar[], TC: O(N log N)



Idea2:



Eg 2:



ar[12] = { 14 6 8 10 1 11 17 19 20 18 33 29 }

0 1 2 3 4 5 6 7 8 9 10 11

P_2 P_1

Swap ar[0] → ar[P_2]

ar[12] = { 11 6 8 10 1 14 17 19 20 18 33 29 }

0 1 2 3 4 5 6 7 8 9 10 11

P_2 P_1

Eg 3: ar[4] : { 8 4 1 7 } → no element, $P_1 = 4$, $P_2 = 3$: break

0 1 2 3 4

P_2 P_1

Swap ar[0] → ar[P_2]

ar[4] : { 7 4 1 8 }

0 1 2 3

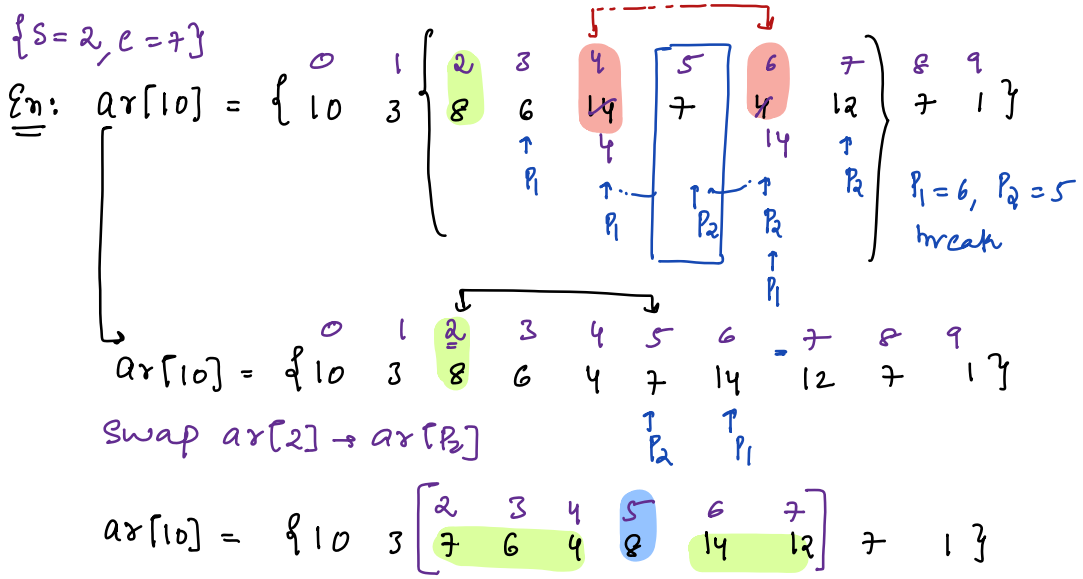
```
void re-arrange(int ar[N]){   TC: O(N)  SC: O(1)
    //reference = ar[0]
    P1 = 1, P2 = n-1
    while( P1 <= P2 ){
        if( ar[P1] <= ar[0] ) { P1++;}
        else if( ar[P2] > ar[0] ) { P2--;}
        else{
            swap (ar[P1], ar[P2])
            P1++, P2--
        }
    }
    swap( ar[0] & ar[P2])
}
```

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

→ ar[s] should come to correct pos

→ all elements <= ar[s] should left

→ all elements > ar[s] should right

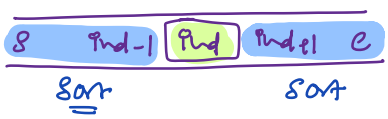



```
Pnt re-arrange(Pnt ar[N], Pnt s, Pnt e){   TC: O(N)  SC: O(1)
    //reference = ar[s]
    P1 = s+1, P2 = e
    While( P1 <= P2 ){
        if( ar[P1] <= ar[s] ) { P1++; }
        else if( ar[P2] > ar[s] ) { P2--; }
        else{
            Swap(ar[P1], ar[P2])
            P1++, P2--
        }
    }
    Swap( ar[s] & ar[P2] )
    return P2; // returning index at which we keep ar[s]
}
```

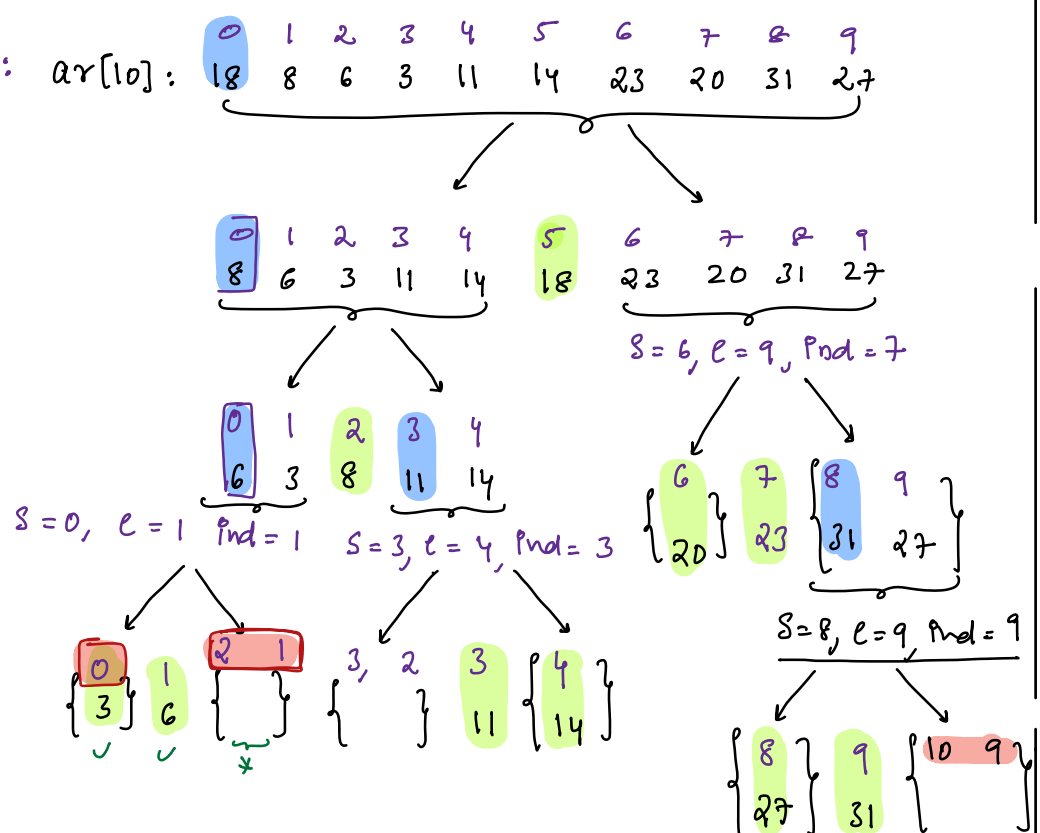
3@) Sort entire array:

Ans: Sort entire subarray from [s e]

```
void QuickSort(int ar[], int s, int e){
    if(s >= e) { return } => { Doubt }
    int pind = rearrange(ar, s, e) // keep ar[s] in corrt pos
    sort(ar, s, pind-1)
}   sort(ar, pind+1, e)
```



Dryrun: ar[10]:



Final data:

→ 0 1 2 3 4 5 6 7 8 9

3 6 8 11 14 18 20 23 27 31

8:20 → 8:30am

```
void QuickSort(int ar[], int s, int e){
    if(s >= e) { return } => { Doubt }
    int ind = rearrange(ar, s, e)
    Sort(ar, s, ind-1)
    Sort(ar, ind+1, e)
}
```

Recurrence Relations:

Best: $T(N) = N + 2T(N/2)$

$\therefore T(N) = N \log_2 N$

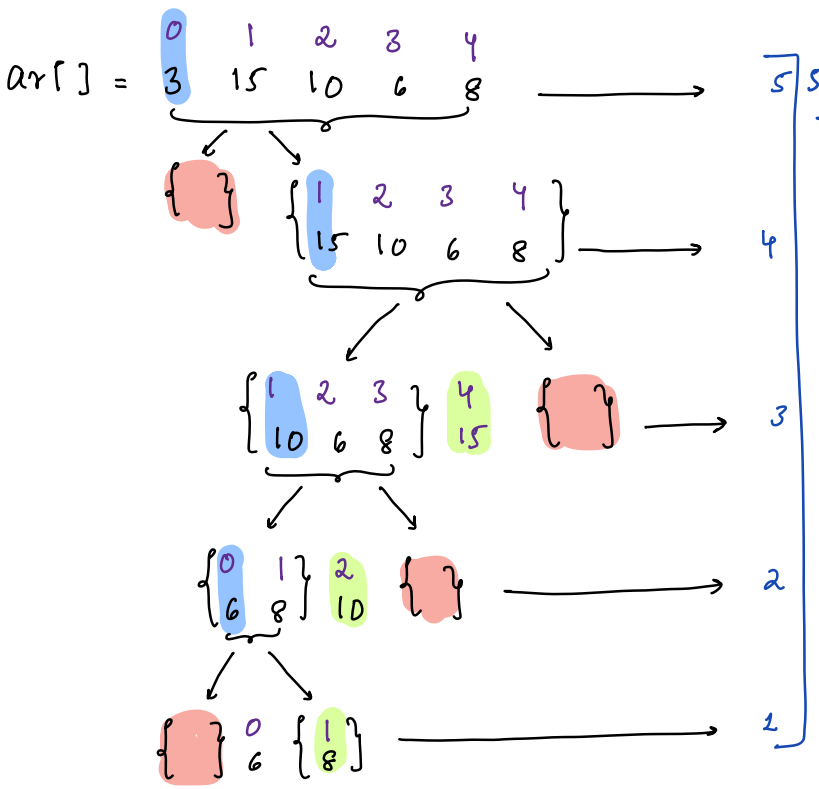
Worst: $T(N) = N + T(N-1)$

all elements goes to one side

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

Worst Case:

re-arrange iterations

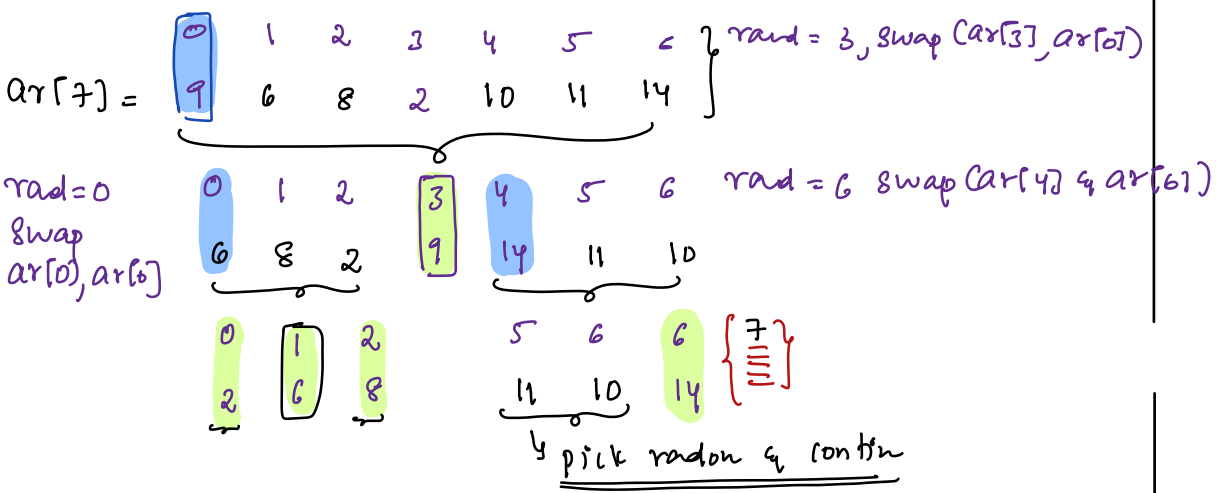


$5+4+3+2+1 \approx 15$

Worst Case $\approx O(N^2)$

As reference of we pick smallest a largest element at every re-arrange we will get worst Case scenario = $O(N^2)$

Optimized: pick any random ind ele in given range as ref



Final Quick Sort

```
Pnt  re-arrange (Pnt ar[N], Pnt s, Pnt e) {   TC: avg O(N log N)
    //radom index from [s - e]
    [r = rand(s, e) //return random number
    Swap (ar[s], ar[r])
    P1 = s+1, P2 = e
    while ( P1 <= P2 ) {
        if ( ar[P1] <= ar[s] ) { P1++;}
        else if (ar[P2] > ar[s] ) { P2--;}
        else{
            Swap (ar[P1], ar[P2])
            P1++, P2--
        }
    }
    Swap ( ar[s] & ar[P2])
    return P2 ; // returning index at which we keep ar[s]
}
```

Inbuild :

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

C++ → IntroSort : QuickSort + HeapSort
(4 & future)

python → Tim Sort : Insertion + Merge

→ pick k min : Selection

→ pick max in stable : Bubble

→ insert in sorted[] : Insertion

→ Sort entire data : Inbuilt sort

CountSort:

// Given ar[N], all elements in range [1-4] sort ar[]

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

Idea1: Sort arr[]

TC: NlogN

Idea2: Store freq of all elements [1-4] in hashmap

ele : freq				
1 : 3	1	1	1	
2 : 2	2	2		
3 : 3	3	3	3	
4 : 2	4	4		

1 1 1 2 2 3 3 3 4 4

Pseudo code:

Step1: Store all elements with freq in hashmap

Step2:

```
k = 0
for(i = 1; i <= 4; i++) {        // data in range [1 4]
    // how many time i occurs = hm[i]
    for(j = 0; j < hm[i]; j++) {
        ar[k] = i, k++;
    }
}
// ar[] is sorted
```

SQ) : Data in range [s e] ? TC: O(N) SC: O(N)

Step1: Store all elements with freq in Hashmap → O(N)

Step2: k=0

```
for(i=s; i<=e; i++){
    // how many time i occurs = hm[i]
    for(j=0; j<hm[i]; j++){
        ar[k] = i, k++;
    }
}
// ar[] is sorted
```

→ data in range [s e]

→ O(N+R)

TC O(N+R)

R = e-s

R: depends on data

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

↳ { 2 6 100 2 2 9 80 50 1 9 }

[1 100]

i = s → e j: freq of [i]

i	j: freq of [i]
1	freq[1]
2	freq[2]
3	2
4	1
5	1
6	freq[6]
7	1
9	freq[9]
:	:
50	freq[50]
:	:
100	freq[100]

Total: [Sum of all freq() = N + Sum of all is = e-s]

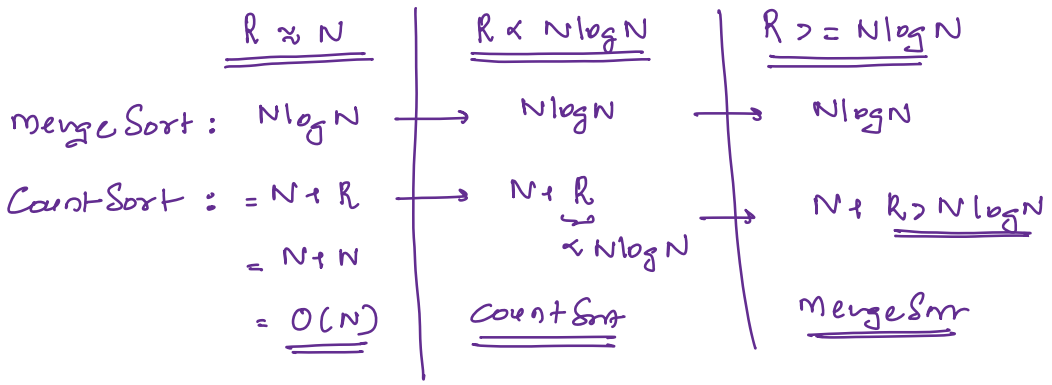
TC: O(N + e-s)

Ingeneral e-s is called Range R

TC: O(N+R)

Sorting data with freq = CountSort

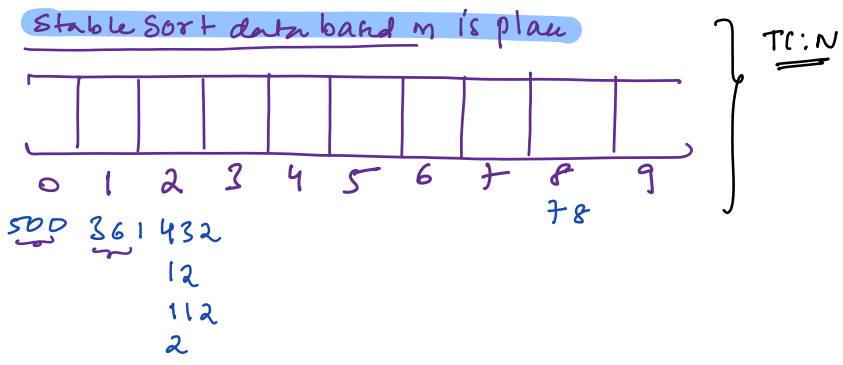
// ar[N] : Range R = max - min of array



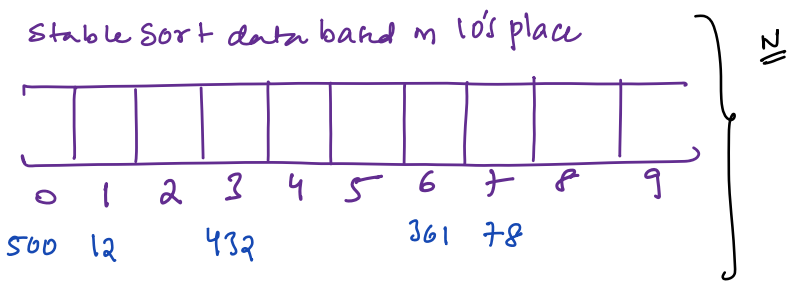
If Range R ≈ N : Good to go with CountSort

RadixSort: Sort data digit by digit from one place → ---

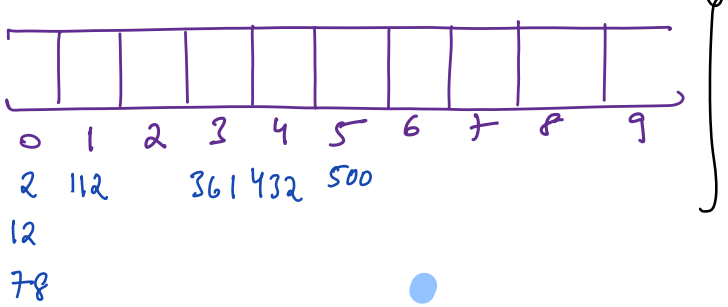
ar[] = 361 432 12 78 500 112 2



ar[]: 500 361 432 12 112 2 78



ar[]: 500 2 12 112 432 361 78



ar[]: 2 12 78 112 361 432 500

If given integers ≈ 10^9 → digits ≈ 10 → radix Sort = N*10 → O(N)

sc = O(N)

TODO: Radix Sort real time applications TODO

→ BS + 2P + Hashing + Strings : 9

→ Linked / Stacks : 7

→ Trees → 6

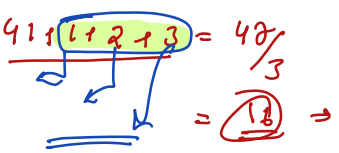
→ Heaps / Tries / Greedy = 5

→ Backtracking = 2

→ Dp → 7

→ graphs → 5

(41) : ——



Calendar:

Sep: 27

↓

Oct: 4

↓

11

↓

18

↓

25

Nov: 1

↓

8

↓

15

↓

22

↓

29

Dec: 6

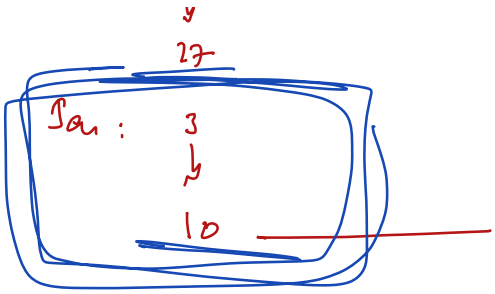
↓

13

↓

20

|


y
27
Ja :
3
10