Today's Content :

- Search in a Row-wise Column-wise Sorted matrix [HW]
- Merge Intervals [HW]
- Given N Elements find first missing Natural Number
  - ↳ daut Session
- Maximum Absolute Difference
  - = {always +ve}

$|-2| = +(+2) = 2$

$$|a| = \begin{cases} a>0 : a \\ a<0 : -a \end{cases}$$

$$|a| = \max(a, -a)$$

$$|a| - b = \begin{cases} a>0 : a-b \\ a<0 : -a-b \end{cases}$$

$$|a|-b = \max(a-b, -a-b)$$

```
// int abs(int a){
     if(a>0)
        return a
     else
        return -1*a
   }
```

abs(2) → 2

abs(-2) → -2

return max(a, -a)

18) Find the first missing Natural Number?

[1, 2, ...]

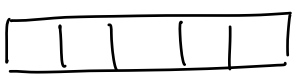
Ex1: ar[5] = {3, -2, 1, 2, 7} : 4

Ex2: ar[7] = {-9, 2, 6, 4, -8, 1, 3} : 5

Ex3: ar[6] = {1, 2, 5, 6, 4, 3} : 7

Ex4: ar[5] = {-4, 8, 3, -1, 0} : 1

Ex5: ar[4] = {4, 2, 1, 3} : 5

Idea1: ar[5] : 

1 ⟶ ✓

2 ⟶ ✓

3 ⟶ ✓

4 ⟶ ✓

5 ⟶ ✓

6 :→ ans

Obs: int ar[N] → min: 1

→ max: N+1

Idea:

```
given N
for(i=1; i<=N; i=i+1){
  // Search if i present in array
  for(j=0; j<N; j++){
    if(ar[j]==i){
      |
    }
  }
  if i not present return i
}
return (N+1)
```

TC: O(N^2)

SC: O(1)

Idea2: Using hashset

→ Insert all Elements in hashset hs

```
for(i = 1; i <= N; i = i+1){
    // if (i is not in hs){
        return i;
    }
}
return N+1
```

TC: O(N)

SC: O(N) → Because of hashset

Condition: No Extra Space.

Idea3: → Sort & Iterate & get first Missing Number

O(NlogN + N) ≈ TC = O(NlogN) SC = O(1)

ar[7] = { -9, 2, 6, 4, -8, 1, 3 }

Sort array : { -9 -8 1 2 3 4 6 }

x x ✓ ✓ ✓ ✓ (5) → ans = 5

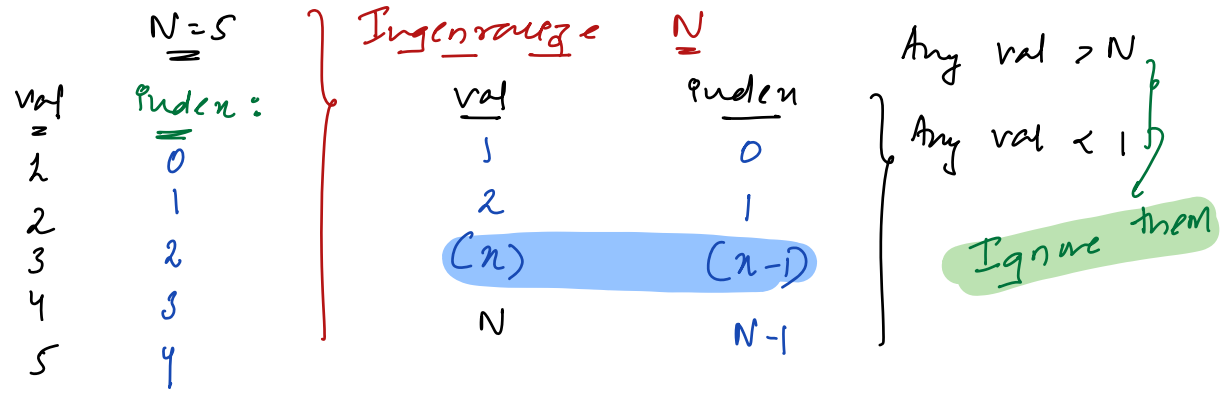
ar[8] = { -3, 1, 5, 8, 14, 2, 7, 3 }

Sort array : { -3 1 2 3 5 7 8 14 }

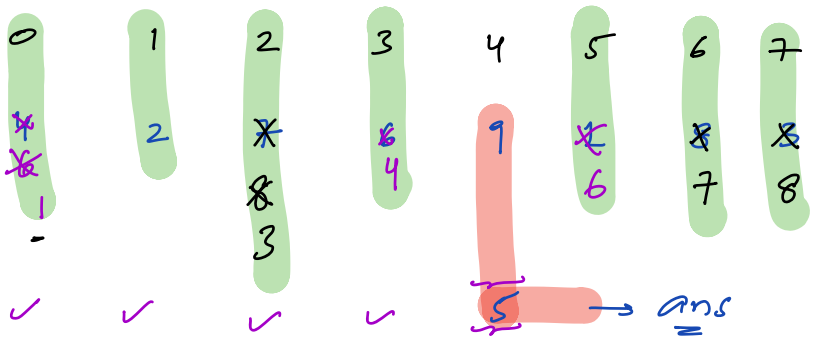
x ✓ ✓ ✓ (4) → 4 is missing → ans

Condition: Optimize more : Expected TC: O(N) SC: O(1)

Idea: Keep an element in it's correct position.



ar[8] =



i ar[i]

0 : 4 ⇒ Swap (ar[0], ar[3]) ⇒ Swap (ar[0], ar[5]) : Set

1 : 2 ⇒ Set

2 : 7 ⇒ Swap (ar[2], ar[6]) ⇒ Swap (ar[2], ar[7]) : Set

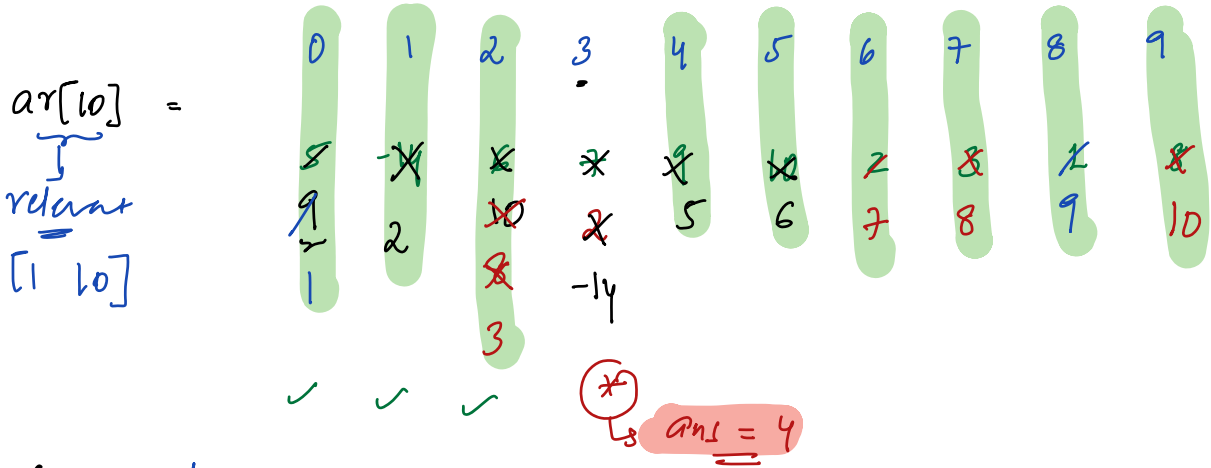
3 : 4 ✓

4 : 9 : (Irrelevant data, Break)

5 : 6 ✓

6 : 7 ✓

7 : 8 ✓



i : val

0 : 5 → swap(ar[0], ar[4]) → swap(ar[0], ar[8]) : (Break)

1 : -14 → (Break)

2 : 6 → swap(ar[2], ar[5]) → swap(ar[2], ar[9])

swap(ar[2], ar[7]) : Set : (Break) ✓

3 : 7 → swap(ar[3], ar[6]) → swap(ar[3], ar[1]) →

4

5 Irrelevant data at ar[3] hence break ✓

6

7

8

9

→ Iterating from start & getting 1st missing Natural Number

ar[5] = {1, 15, 12, 18, 20}

[1-5] → 1 → return 13

Pseudocode:

// Given ar[N]

```
for (i = 0; i < N; i++) {
    while ( ar[i] > 0 && ar[i] <= N && ar[i] != (i+1) ) {
        int val = ar[i]
        if ( ar[i] == ar[val-1]) { break }
        swap ar[i] & ar[val-1]
    }
}
```

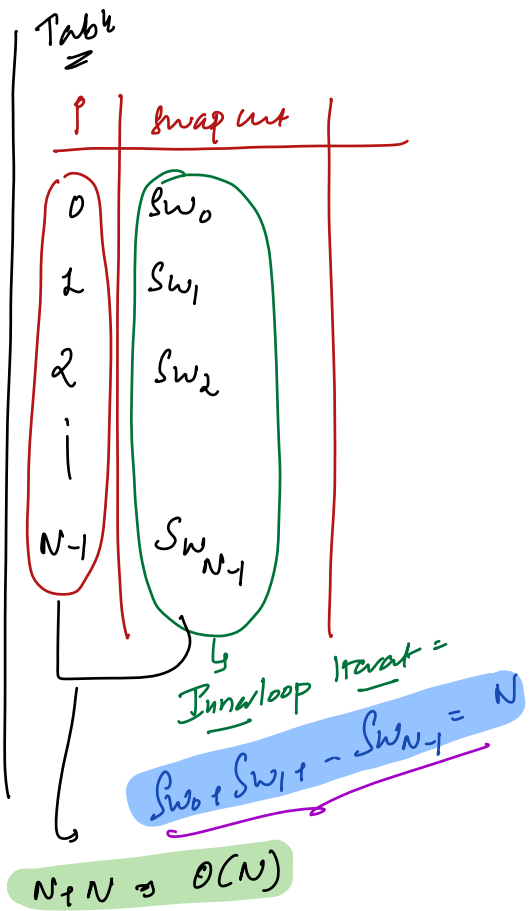
TC: O(N)

(ar[i] > 0 && ar[i] <= N: relevant data; ar[i] != (i+1): not set; swap count)

// Brute & get missing number

```
for (i = 0; i <= N; i++) {
    if ( ar[i] != (i+1) {
        return (i+1)
    }
}
return (N+1)
```

TC: O(N)

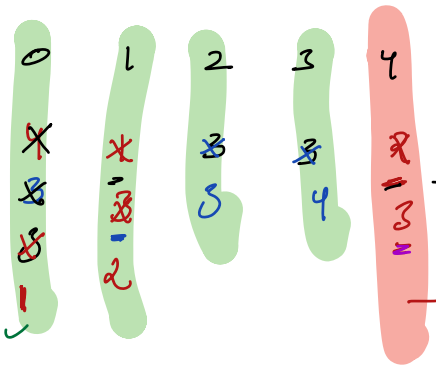


TC: O(N)

SC: O(1)

Eg4: ar[5] : { 0 1 2 3 4 } → ans = 5

val



i=0: 4 → swap(ar[0], ar[3]) → ∞ loop ?

3 → swap(ar[0], ar[2])

3 → swap(ar[0], ar[2])

3 → swap(ar[0], ar[2])

ar[0] == ar[2] Bcoz of this it is going to ∞ loop :(Break)

i: 1 → swap(ar[1], ar[0])

3 → swap(ar[1], ar[2]) {Break}

i: 2 → skip ✓

i: 3 → skip

i: 4 : 2 → swap(ar[4], ar[1])

3 → swap(ar[4], ar[2]) {Break}

10: SD break

// Given a [N][M] 2D matrix, every row sorted, every column sorted

find an element k

N sorted elements : Binary Search k
TC: $\log_2^N$

k = 15

	0	1	2	3	4	5	
0	-1	2	4	5	9	11	< 15 x
1	1	4	7	8	10	14	< 15 x
2	3	7	9	10	12	18	≥ 15 ✓
3	6	10	12	14	16	20	> 15 ✓
4	9	13	16	19	22	24	> 15 ✓
5	11	15	19	21	24	27	> 15 ✓
6	14	20	25	29	31	39	> 15 ✓
7	18	24	29	32	34	42	x

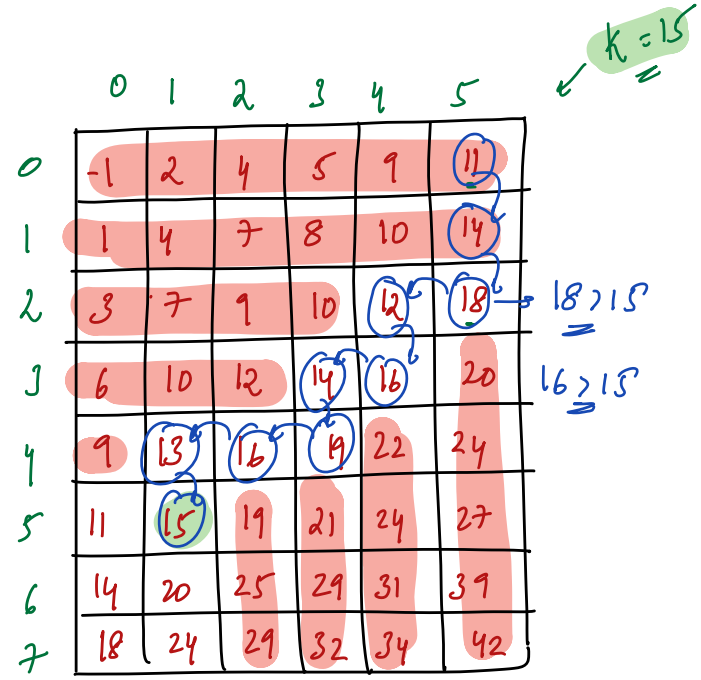
→ Idea 1: Traverse in Entire Mat
TC: N*M SC: O(1)

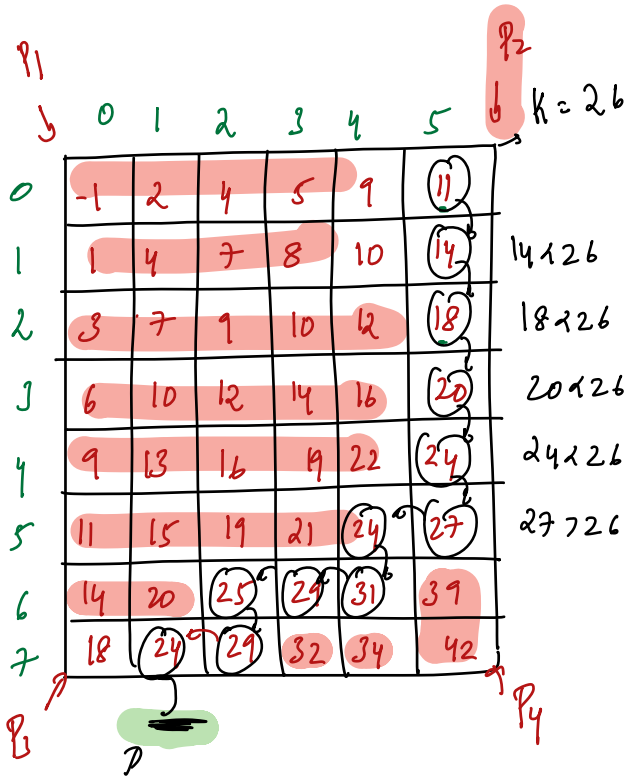
Idea 2: Apply BS in every row
TC: N * $\log_2^M$

Idea 3: Apply BS in every col
TC: M * ($\log_2^N$)

Idea 4: Compare k with first & last element of a row Based in that apply BS in that row

TC: N * ($\log_2^M$)





// (Try from P_1, P_3, P_4)

```
i = 0, j = M-1
while (i < N && j >= 0) {
    if (ar[i][j] < k) {
        // skip row
        i++;
    }
    else if (ar[i][j] > k) {
        // skip col
        j--
    }
    else
        return True
}
return false;
```

Idea: At every step we either skip a row or a column

Total no. of steps: (N + M)

Rows = N

Cols = M

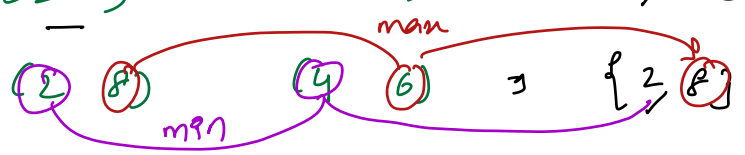
Merge Intervals ?

I_1 : [a b]

// overlapping → { Even if they are intersecting at 1 element, that is overlapping }

Ex: I_1 I_2 Merged Interval

(2 6) (3 7) → {2, 7}



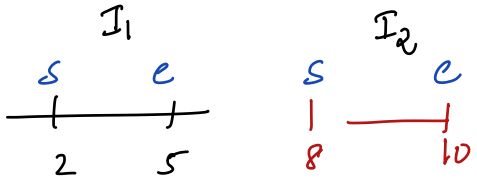
(2 8) (4 6) → {2, 8}

(3 7) (4 10) → {3, 10}

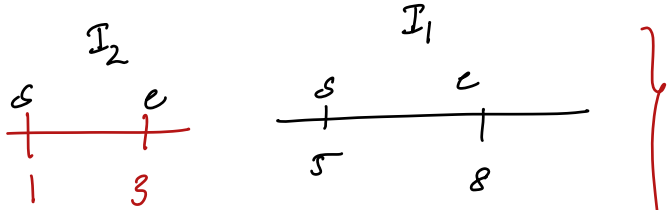
(3 6) (6 10) → {3 10}

(2 5) (8 10) → { Not overlapping }

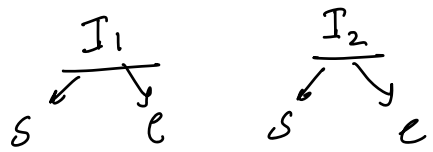
(5, 8) (1, 3) → { Not overlapping }



$I_2.s > I_1.e$: not overlap



$I_2.e < I_1.s$: not overlap



⇒ Merged Inter → s, e

= {min(I1.s, I2.s) max(I1.e, I2.e)}

//

Given N non-overlapping intervals, & They are sorted Based on start, (New Interval I comes) ⇒ Merge all ⇒

N = 7

New Interval

[1 3]

[4 7]

[10 14] → [12 22] : [10 22]

[16 19] → [10 22] : [10 22]

[21 24] → [10 22] : [10 24]

[27 30] → [10 24] :

[32 35]

[38 41]

New non overlapping Intervals

[1 3]

[4 7]

[10 24] :

[27 30]

[32 35]

[38 41]

Final ans

N = 5:

Intervals

ans

[1 5] [12, 22]

[8 10]

[11 14] [12 22] : [11, 22]

[15 20] [11, 22] : [11 22]

[20 24] [11 22] : [11 24]

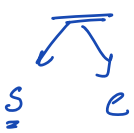
[1 5]

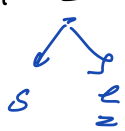
[8 10]

[11 24]

// Pseudo Code :

Say ar[N] Intervals, Interval I comes





```
Intervals []  merge ( Intervals ar[], Interval I){

    for(i = 0; i < N; i = i+1){                    TC : O(N)

        if Interval = ar[i],                        I

        if(ar[i].e < I.s){
            ans.insert( ar[i] )
        }
        else if( I.e < ar[i].s){
            ans.insert( I);
            for(j = i; j<N; j++){
                ans.insert(ar[j])
            }
            return ans;
        }
        else { // overlapping
            ar[i] & I
            I.s = min(ar[i].s, I.s)
            I.e = max(ar[i].e, I.e)
        }
    }
    ans.insert(I)
    return ans;
}
```