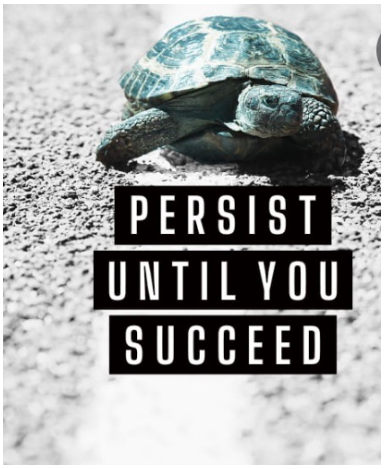


Today's content

① Given N elements . find first missing +ve no.

② Search in row-wise & column-wise sorted matrix.

③ Merge Intervals

Q) Find the first missing natural number. [1, 2, 3, ---]

Eg1: arr[5]: [3, -1, 1, 2, 7] : 4

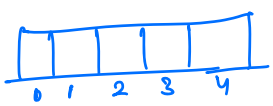
Eg2: arr[7]: [-9, 2, 6, 4, -8, 1, 3] : 5

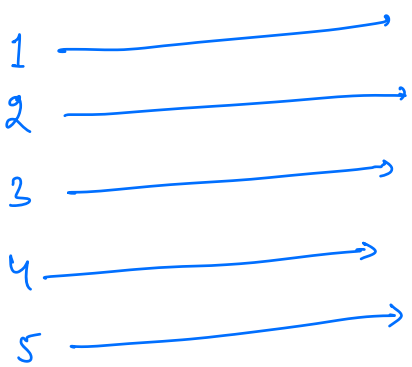
Q: arr[6]: [1, 2, 5, 6, 4, 3] : 7

Q: arr[6]: [1, 0, -5, -6, 4, 2] : 3.

idea-1

arr[5] →





[6 → ans.]

obs. → min : 1
→ max : N+1.

idea-1.

```
for( i = 1 ; i <= N ; i++) {
    //check if ith. No. is present in arr[] ?

    for( j = 0 ; j < N ; j++){
        if(arr[j] == i){

        }
    }
    // if ith. no. is not present in arr[]
        // return i
}
return N+1 ;
```

T.C → O(N²)
S.C → O(1)

idea-2. Using Hashset

1. // Insert all the elements in hashset (hs)

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

T.C → O(N)
S.C → O(N)

Condition → No extra space is allowed.

idea-3 Sort & then iterate to get first missing natural number.

① arr[7]: [-9, 2, 6, 4, -8, 1, 3]

↓

arr[7]: [-9 -8 1 2 3 4 6]

(+1 +1 +1 +1)

✗ ✗ ✓ ✓ ✓ ✓ (5) → [ans = 5]

② arr[8]: [-3, 1, 5, 8, 14, 2, 7, 3]

T.C → O(NlogN)
S.C → O(1)

After sorting ↓

arr[8]: [-3, 1, 2, 3, 5, 7, 8, 14]

(+1 +1 +1)

✗ ✓ ✓ ✓ (4) → [ans = 4]

→ [ignore -ve number & duplicates]

Implementation (todo)

Optimization. O(NlogN) → O(N). , S.C → O(1)

idea-4 Keep the element at its correct position.

N = 5.

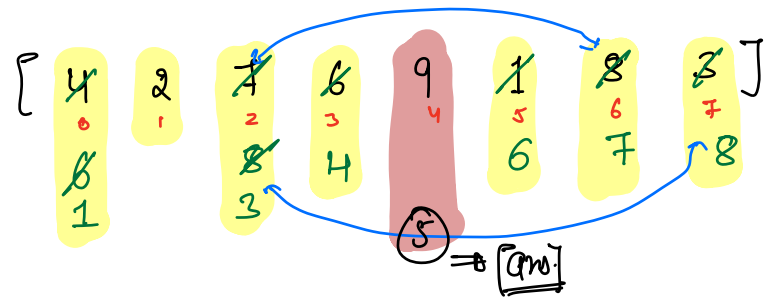
i = 0	val = 1
i = 1	val = 2
i = 2	val = 3
i = 3	val = 4
i = 4	val = 5

Generalize.

val.	idx
1	0
2	1
3	2
4	3
⋮	⋮
x	(x-1)
⋮	⋮
N	(N-1)

if
val > N.
val < 1.
{ignore them}

arr[8] :



i	val.
0	swap(arr[0], arr[3]), swap(arr[0], arr[5]) stop.
1.	2. (nothing)
2.	swap(arr[2], arr[6]), swap(arr[2], arr[7]) stop.
3	4 ✓
4	9 [ignore]
5.	✓
6	✓
7.	✓

arr[ip]: [8 -14 6 7 9 10 2 3 8 12 1]

0 1 2 3 4 5 6 7 8 9 10

2 3 -14 5 6 7 8 9 10 12

ans = 4 → first missing natural no.

i	:	val
0	:	swap(arr[0], arr[4]), swap(arr[0], arr[8]), swap(arr[0], arr[7]) swap(arr[0], arr[2]), swap(arr[0], arr[5]), swap(arr[0], arr[9]) stop
1	:	ignore.
2	:	3 ✓
3	:	swap(arr[3], arr[6]), swap(arr[3], arr[1]) stop.
4	:	✓
5	:	✓
6	:	✓
7	:	✓
8	:	✓
9	:	✓
10	:	swap(arr[10], arr[0]) stop.

observation: In single swap, at least one element will be shifted at its correct position.

At max, how many swaps are possible? ⇒ N swaps.

pseudo-code

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

(arr[i] > 0 && arr[i] <= N : relevant data; arr[i] != i+1 : not satisfied.)

T.C → O(N)
S.C → O(1)

// iterate again & then find first missing Natural no.

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

i	swap.count
0	s_1
1	s_2
2	s_3
⋮	⋮
N-1	s_N

$= s_1 + s_2 + s_3 + \cdots s_N$

iterations. = 2N

Eg → arr[5] : [4 3 3 2 3]

i	val.
0	swap(arr[0] and arr[3]), swap(arr[0] and arr[2])
	swap(arr[0] and arr[2]), swap(arr[0] & arr[2], - -
—	—— T.L.E (infinite loop)

if (arr[i] == arr[val-1])
we need to break
otherwise, infinite loop will be there.

Q) Given a matrix where every row & column are sorted.

Find an element K.

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

K=15.

idea-1: traverse the whole matrix & check if arr[i][j] == K or not.

T.C → O(N·M), S.C → O(1)

Binary search → search for any element in sorted array.

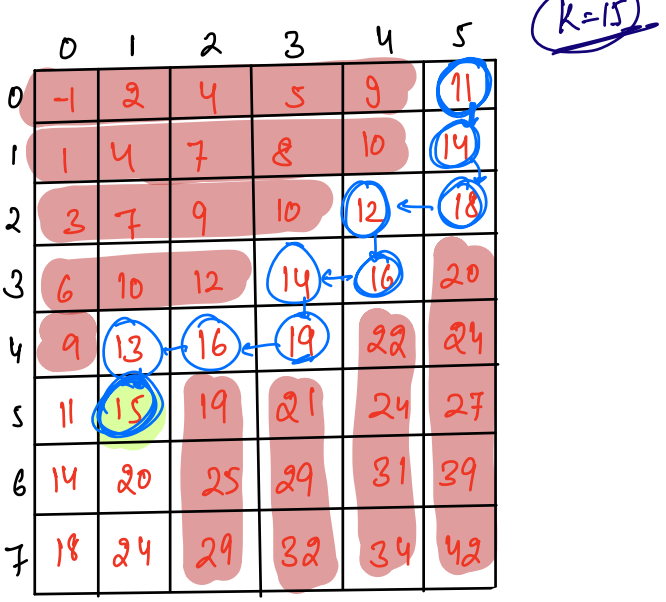
↳ logN.

idea-2 → In every row, apply binary search.

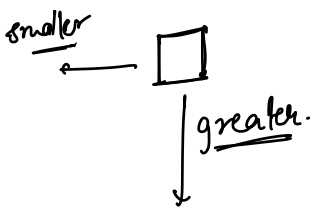
T.C → O(N log M)

idea-3 → In every col, apply binary search

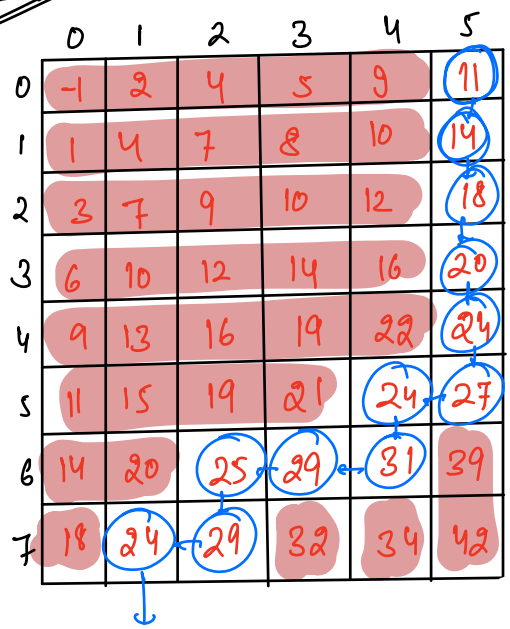
T.C → O(M log N)



K



k=26



Todo # by with bottom-left corner.

pseudo-code:

```
i = 0 , j = M-1

while (i<N && j≥0){
    if (arr[i][j] < k) {
        i++
    }
    else if (arr[i][j] > k){
        j--
    }
    else {
        return true
    }
}
return false;
```

observation

- At every step we are either skipping a row or a column.

no. of steps = N + M

T.C → O(N+m)
S.C → O(1)

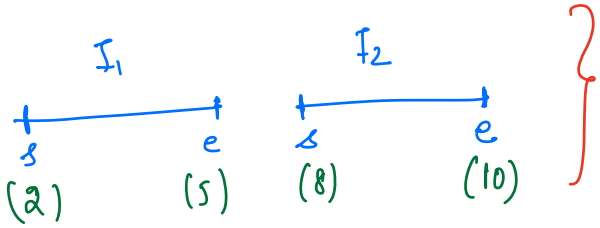
Merge Intervals

↳ range / boundary / [a, b].

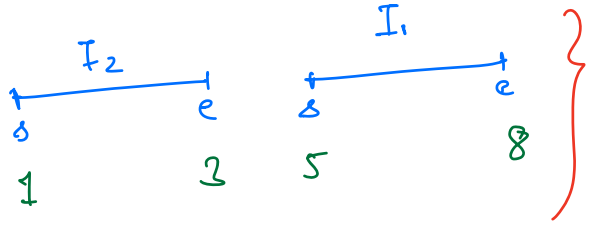
Eg:

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]	{No overlapping}
[5 8]	[1 3]	{No overlapping}

Non-overlapping.

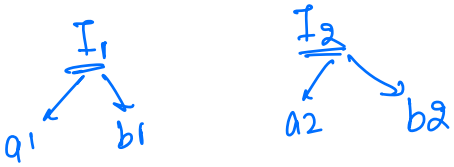


$I_2.s > I_1.e$



$I_2.e < I_1.s$

if I_1 & I_2 are overlapping. (Generalisation for merged interval)



⟹ merged interval

$\{\min(a1, a2), \max(b1, b2)\}$

Given N non-overlapping intervals & they are sorted based on start-time. You are given a new interval. Merge all the intervals.

Eg. N = 8

New interval

[1 3]

[4 7]

[10 14] + [10,22] ⇒ [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 req. ans.

N=5

Interval

[1, 5]

[8, 10]

[11, 14] + [12, 22] ⇒ [11, 22]

[15, 20] + [11, 22] ⇒ [11, 22]

[20, 24] + [11, 22] ⇒ [11, 24]

New non-overlapping intervals

[1, 5]

[8, 10]

[11, 24]

} final req. ans.

arr[i] with interval I

s e

s e

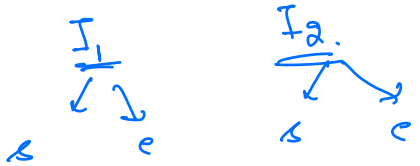
pseudo-code-

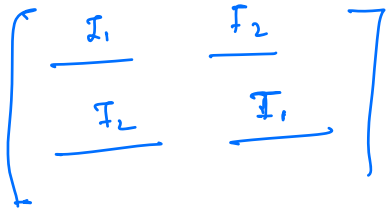
```
Interval[] mergeIntervals( arr[], Interval I ){

for( i=0 ; i<N ; i++){
        // arr[i] → interval in the array.
        if( arr[i].e < I.s){
            // they are non-overlapping, update ans
            ans.insert(arr[i])
        }
        else if( arr[i].s > I.e){
            // they are non-overlapping
            ans.insert(I);
            for( j=i ; j<N ; j++){
                ans.insert(arr[j]);
            }
            return ans
        }
        else{ //overlapping
            //merge arr[i] & I^th
            I.s = min(arr[i].s, I.s)
            I.e = max(arr[i].e, I.e)
        }
}
ans.insert(I);
return ans.
```

T.C → O(N)
S.C → O(1)

Doubts :





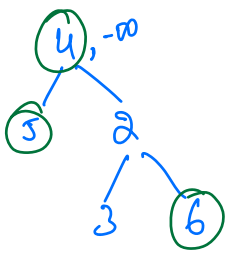
non-overlapping. ① $I_1.s < I_2.s$.

check if ($I_1.e < I_2.s$)

② $I_2.s < I_1.s$

check if ($I_2.e < I_1.s$)

Count of nodes with more value than all its ancestors



Keep an track of maximum element