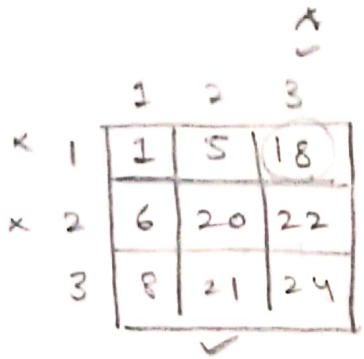
Array 2.

Q. In a 2D matrix, all rows are sorted & all columns are sorted.

↓

given x, find the indices of x



(6), (21), (17)

all unique elements

↓

assume no duplicates.

↓

* check at the top right corner or bottom left corner then 2 cases are possible. The element at that index could be either greater or less than that element. If greater, then in that column, the element can't be found. If smaller, then it can't be found in that row. we keep ruling out columns & rows in this manner and in the end we can conclude if present or not.

↓

Ruling out is done by moving in one direction, i.e. left or down (if started from top right corner)

↓

OCR+C) (moving from top corner to bottom corner by going to bottom right first & then to bottom left)

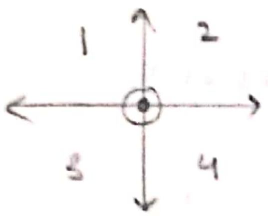
* Binary search in sorted row & column matrix

↓

pick middle element, and depending on its value, i.e. it is greater than or less than x.

we discard one quadrant of the matrix. (top left or bottom right)

and then make a recursive call on other three blocks.



element < mid, call on 1, 2, 3; discard 4.

↓

Divide & conquer approach.

$$T(n*n) = O(1) + 3^* T(n/2 * n/2)$$ → 3 half sided matrices

$$T(n/2 * n/2) = O(1) + 3^* T(n/4 * n/4)$$

$$T(n/4 * n/4) = O(1) + 3^* T(n/8 * n/8)$$

⋮

$$\frac{r^n - 1}{r - 1}$$

$$T(1) = O(1)$$

↓

$$T(n*n) = O(1) + 3^* (O(1) + 3^* T(n/4 * n/4))$$

$$T(n*n) = O(1) + 3^* (O(1) + 3^* (O(1) + 3^* T(n/8 * n/8)))$$

$$= O(1) + 3\,O(1) + 9(O(1)) + 27(O(1)) + \ldots$$

$$= O(1) + 3^* O(1) [1 + 3 + 9 + 27 + \ldots]$$

$$= O(1) + 3^* (O(1)) \left[\frac{3^{\log_2 n} - 1}{2} \right]$$

$$= c_1 + c_2 \frac{n^{\log_3 3} - 1}{2}$$

$$= c_1 + c_2 \frac{n-1}{2}$$

$$= c_1 + \frac{c_2}{2} n - \frac{c_2}{2}$$

$$= c_1 - \frac{c_2}{2} + \frac{c_2}{2} n$$

$$= O(n)$$ (same order complexity as the previous solution)

more complicated and would require recursion for implementation ↑

Scanned by CamScanner

(one kind of observation)

(**) * checking on extremities can actually lead to a solution in a matrix or array based problems.

(pattern)

↓

pointer approaches.

many a time rely on checking on the extremities

↓

(true for dp & graph as well)

* breaking the problem into sub problems

Q. Given an array, find if there is any element that is a majority element

not necessary that it is present (any element is in majority)

(which occurs more than n/2 times)

↓

(i) hashmap → storing the frequencies of the element and after the traversal, we iterate over the hashmap to check the frequencies, if such an element matches the constraint, then we return it.

avg → O(n) ✓

worstcase → O(n²) ✗

↓

unordered_map

in case of a lot of collisions

time complexity ↑ ← unordered_map → hashing with chaining → dependent on the function used

map → red black tree (self balancing ordered tree with other properties)

↓

avg, worst: O(nlogn)

* Assuming the majority element always exists, then if we have 2 elements at a time, if they are different, then we disregard them

(majority element still stays the same)

↓

(Moore's voting Algorithm)

and after obtaining this majority element.
Run another loop to check if it really is majority or not. → compare the count of the found element with the size of the array

↓ extension

(Q) find whether there exist an element whose frequency is more than $n/3$?
If yes, return that element, else return -1.

---

majority element ($n/2$)

$(\frac{n}{2}+1)$

3 2 1 1 4 1 5 1 1

↑ ↑ ↑ ↑

---

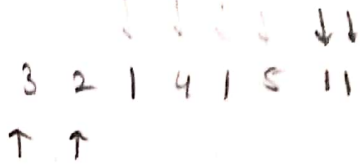
* if array is of size N and assuming majority element exist, if we encounter 2 distinct elements, removing / discarding them would still yield the same majority element, since in worst case one of those elements would be the majority element. and ∴, if M is a majority element, then

$$M \rightarrow \frac{N}{2}+1$$

and for M to stay majority element in $N-2$ based array, it has to occur more than $\frac{N-2}{2} = \frac{N}{2}-1$ times.

↓

if M is one of the elements then we can remove it once, and then the new frequency becomes $\frac{N}{2}$, which is more than the req. frequency.



c1 = 1
c2 = 0
c1 = 2
c2 = 0

↓

can be done using two pointers and count variable

```
int majority (vector<int> &v){

    i=0, j=1, count=1, N = v.size();
    while (j < N){

        if (v[i]==v[j]){
            count++;
            j++;
        }
        else if (count>1){
            count--;
            j--;
        }
        else{
            i=j+1;
            j=i+1;
        }
    }
```

count of both $\frac{N}{3}$ element if present will reduce by 1.

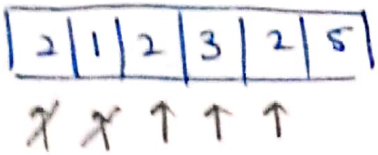
$\left(\frac{n}{3}+1\right)$

$\frac{n-3}{3} = \frac{n}{3} - 1$

N/3 version : i) removing 3 distinct elements.

ii) more than one element can be more than $\frac{N}{3}$.

m_1, count1
m_2, count2,



$m_1 = 2, c1 = 1$
$m_2 = 1, c2 = 1$

(i) return the element with higher count.

(ii) check 1 element at a time

```
int maT3(vector<int> & v){
  int N = v.size();
    int m1, c1=0,m2, c2=0;
    m1 = v[0]; c1 = 1;
    int ptr = 1;
    while (ptr < N){
      if (c1==0){
          m1 = v[ptr];
          c1++;
        }
    else if(c2==0){
          if (v[ptr]==m1){
            c1++;
          }
        else{
          m2 = v[ptr];
          c2++;
        }
    else if(v[ptr]==m2){
          c2++;
      }
    else{
      c1--;
    } c2--;
```

```
if(count1==0 &&
   m1==v[ptr]){
    count1++;
  }
```

ptr++;

}

and c2 as well. (whichever is more)

if (c1 ≥ 1) check for m1; (O(N)) → and accordingly return the value

else if (c2 ≥ 1) check for m2; (O(N))

else return (-1);

↓

no $\frac{N}{2}$ majority element.

1 2 2 2 3 2

↓

if the element with high frequency doesn't satisfy then others don't as well.

n/k majority → track k-1 variables, and accordingly work around it.