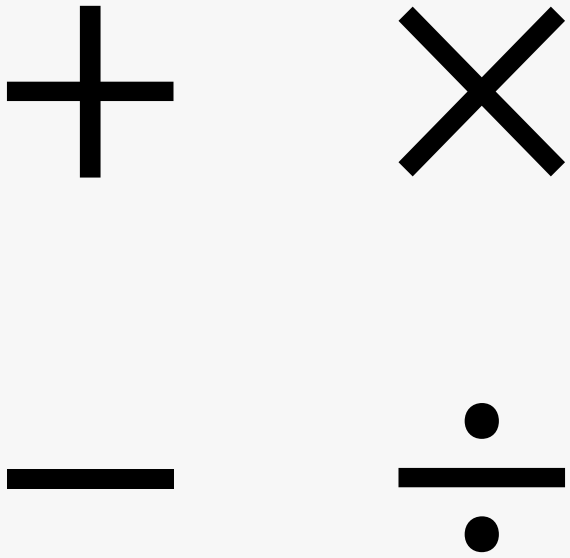


What is time complexity?

Amount of time taken by an algorithm to run as a function of length of input.

input - N

```
for(int i=0; i<N; i++){
    // operations
    print("Hello");
}
```

CPU performs the task N times. If we increase N, CPU also will increase the number of operations.

So the time taken by the function to perform operation is directly proportional to input N.

Here, time means CPU operations and not actual time.
Time complexity of the above function is O(N).

What is space complexity?

Amount of space taken by an algorithm to run as a function of length of input.

Declaring variables like below

```
int a=1;
int b[5];
```

these do not change as you increase the value of N

Hence they are not considered in our space complexity

and we can say the complexity is O(1)

```
int n ; // input
int[] b = new int[n];
```

Here, as you increase n, the space required by an array also increases. so it takes O(N) space.

Unit to represent complexities

1. Big - O : Upper Bound
2. Theta θ : Average case
3. Omega Ω : Lower bound

Eg., Search - to find an item in an array

1	2	3	4	5	6	7

To find 1 in the above array, time complexity would be O(1) as it is found in the first memory address. This is the lower bound Ω(1).

To find 7 in the above array, time complexity would be O(N) as you need to traverse till the end and the time complexity would be O(N) in the worst case.

In best case, time complexity of our code was O(1) and in worst case was O(N), hence in an average if you can find an element in the middle of the array, the time complexity would be O(N/2) or Θ(N/2)

Big O : Complexities

1. Constant time : O(1)
2. Linear time : O(N)
3. Logarithmic time : O(log N)
4. Quadratic time : O(N^2)
5. Cubic time : O(N^3)

Constant time does not depend on the size of your input.
for ex: int a = 5;
If you increase the input size N, it has no impact on the variable used above. So we say they take constant time.

Linear time means if you increase the value of your input the time taken by your program would also increase linearly.
for ex:
```
for(int i=0; i<N; i++){
    print(i);
}
```

Quadratic time will come with a nested loop. For every value of i till N, j travels till N. In worst case scenario it would take $O(N^2)$ time.

Eg.,
```
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){

    }
}
```

Cubic time will come with 3 levels of nesting.

Eg.,
```
for (int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        for(int k=0; k<N; k++){
        }
    }
}
```

For each value of i, j runs N times and in turn for each value of j, k runs N times.

For cases where we have separate for loops,

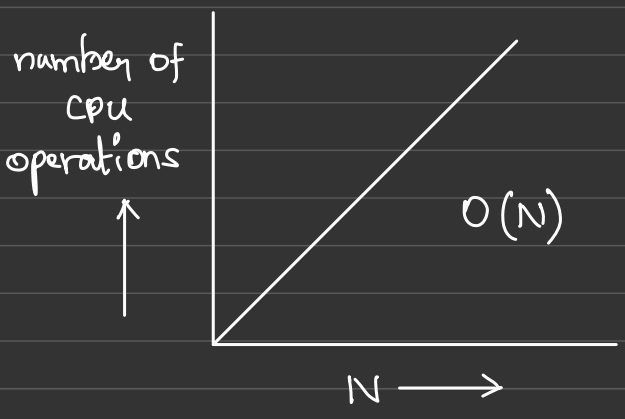
`for (int i=0; i<N; i++){ }` → runs O(N) times

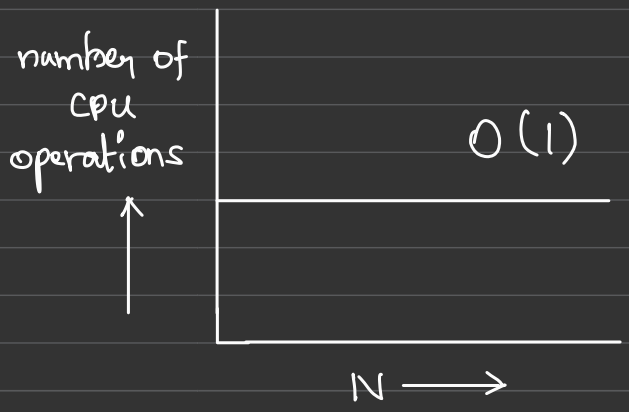
`for (int j=0; j<N; j++){ }` → runs O(N) times

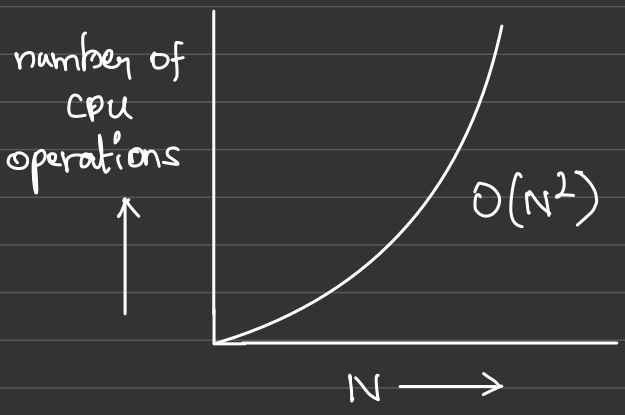
So in total it takes $O(N) + O(N) = O(2N)$

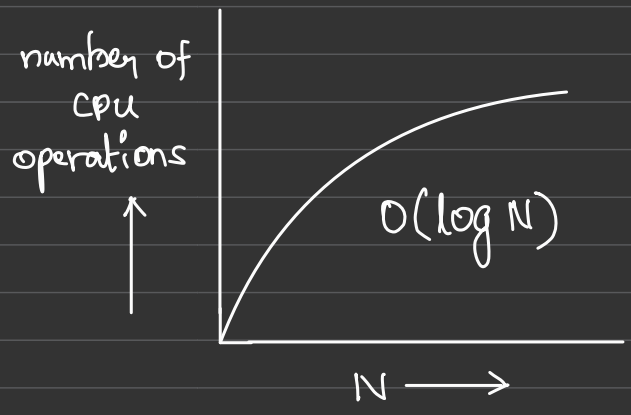
Since constants are anyway ignored, we can say it takes $O(N)$ time in worst case.

Some example graphs of complexities









Eg: ① $f(n) = 2n^2 + 3n \Rightarrow O(2n^2) \Rightarrow O(n^2)$

② $f(n) = 4n^4 + 3n^3 \Rightarrow O(4n^4) \Rightarrow O(n^4)$

③ $f(n) = n^2 + \log n \Rightarrow O(n^2)$

④ $f(n) = 200 \Rightarrow O(200) \Rightarrow O(1)$

⑤ $f(n) = O(n/4) \Rightarrow O(n)$

Least complex to most complex algorithms list

$O(1)$ → least complex algorithm

$O(\log n)$

$O(\sqrt{n})$

$O(n)$

$O(n \cdot \log n)$

$O(n^2)$

$O(n^3)$

$O(2^n)$

$O(n!)$

$O(n^n)$ → most complex algorithm

Eg., ① $O(\log_2 n)$

1	2	3	4	5	6	7

assuming you have an array of sorted elements and you want to search for an element. We have various search algorithms :

① **Linear Search** - it goes in search of an element one after the other starting from the beginning in a linear fashion. Hence it takes $O(N)$ time.

② Binary Search - To find an element you reach to the mid element. If the element to search is greater than the middle element, you ignore first half of the array and search only in the other half. If not found then we repeat this process again until you find the element. In each iteration, we discard half of the array and hence the algorithm takes $O(\log N)$ time.

Starting from input $N \rightarrow N/2 \rightarrow N/4 \rightarrow N/8 \rightarrow \ldots \rightarrow 1$

General format would be $N/2^k = 1$

when you run a loop for N times you perform N operations. Like wise you are running a loop k times since you are dividing a given array k times.

$$\therefore \quad N = 2^k$$

$$\log_2 N = \log_2 2^k$$

$$\boxed{k = \log_2 N}$$

where k is number of operations.

Leetcode - Rotate Array - 189

input an array - {1, 2, 3, 4, 5, 6, 7}

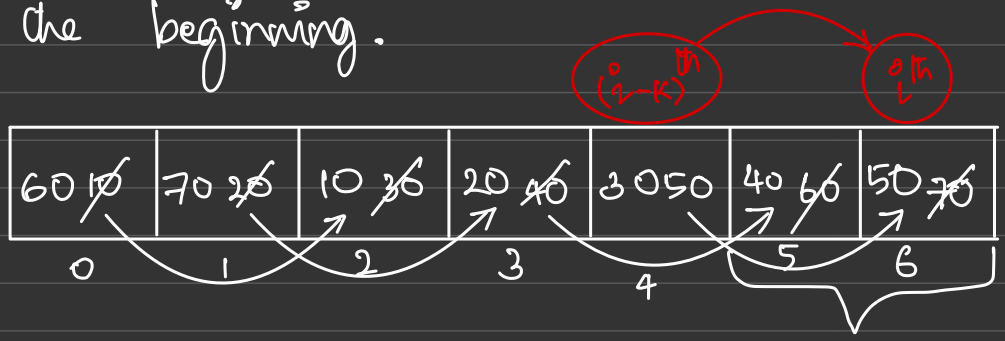
k - 3

output - {5, 6, 7, 1, 2, 3, 4}

(a) copy the last k elements into a temp array.

(b) Run a for loop from n-1 to >=k and copy arr['i-k] inside arr['i].

(c) Copy paste your temp array in the actual array at the beginning.



6th index has 4th index element
5th index has 3rd index element
4th index has 2nd index element

} so the formula would be (index - k)

Using modulus approach

For any circular problems, we may use the modulus approach.

i/p:

10	20	30	40	50	60
0	1	2	3	4	5

K = 2

o/p:

50	60	10	20	30	40
0	1	2	3	4	5

n = 6

The pattern here is :

0^{th} index → 2^{nd} index
1^{st} index → 3^{rd} index
2^{nd} index → 4^{th} index
3^{rd} index → 5^{th} index
4^{th} index → 0^{th} index
5^{th} index → 1^{st} index

(index + K)

(index + K) % n

matches

(0 + 2) % 6 → 2 % 6 → 2
(1 + 2) % 6 → 3 % 6 → 3
(2 + 2) % 6 → 4 % 6 → 4
(3 + 2) % 6 → 5 % 6 → 5
(4 + 2) % 6 → 6 % 6 → 0
(5 + 2) % 6 → 7 % 6 → 1

XOR approach - XOR can be used for swapping.

we can solve in 3 steps:

a) reverse the whole array (0 to n-1)

b) reverse the first half (0 to K-1)

c) reverse the second half (K to n-1)

```
public static void rotate(int[] arr, int K){
    int n = arr.length;
    K = K % n;
    if(n <= K || K==0){ return; }
    reverse(arr, 0, n-1);
    reverse(arr, 0, K-1);
    reverse(arr, K, n-1);
}
private static void reverse(int[] arr, int i, int j){
    while(i < j){
        arr[i] = arr[i] ^ arr[j];
        arr[j] = arr[j] ^ arr[i];
        arr[i] = arr[i] ^ arr[j];
        i++;
        j--;
    }
}
```

```
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
```

Leetcode - Missing Number - 268

Given an array nums containing n distinct numbers return the only number in the range that is missing from the array.

1	7	3	2	5	6	8

[1 → 8] 4 is missing

approaches:

a) search for an element in the given array in a linear manner. Time complexity would be $O(n^2)$.

b) sort the given array and check if two adjacent numbers have a difference of 1. Time complexity would be $O(n.\log n)$.

NOTE - Arithmetic Progression formula to find sum of all numbers is $\frac{n}{2}(a+l)$

where n is the no of elements/terms.
a is the first element/term.
l is the last element/term.

for our example, it should be

$$\frac{8}{2}(1+8) = 4 \times 9 = 36$$

In our problem, first term is 1 and last term is n. So, our formula would then be

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

Leetcode - Rearrange Array Elements by sign - 2149

Given a even length array, rearrange the array elements such that

a) array begins with a positive integer.
b) every consecutive pair have opposite signs.
c) all integers with same sign, we preserve the order.

solution

we separate positive & negative elements into a different arrays and later join them together in a resultant array.

while we copy the elements into the resultant array, make sure to start with positive number and jump 2 places to repeat the same process.

Leetcode - 2643 - Row with maximum ones

Return an array containing the index of the row, and the number of ones in it.

mat →

	0	1	2	3	
0	1	0	0	0	→ 1s - 1
1	0	1	1	0	→ 1s - 2
2	0	1	1	0	→ 1s - 2
3	1	1	1	0	→ 1s - 3
4	0	0	1	0	→ 1s - 1

keep a variable to count number of 1's. As you get the max number of 1's you also store the row number.

In 2-D arrays.

```
for(int i=0 ; i < mat.length ; i++) {
```

→ provides the total number of rows.

```
for(int j=0; j < mat[i].length ; j++){
```

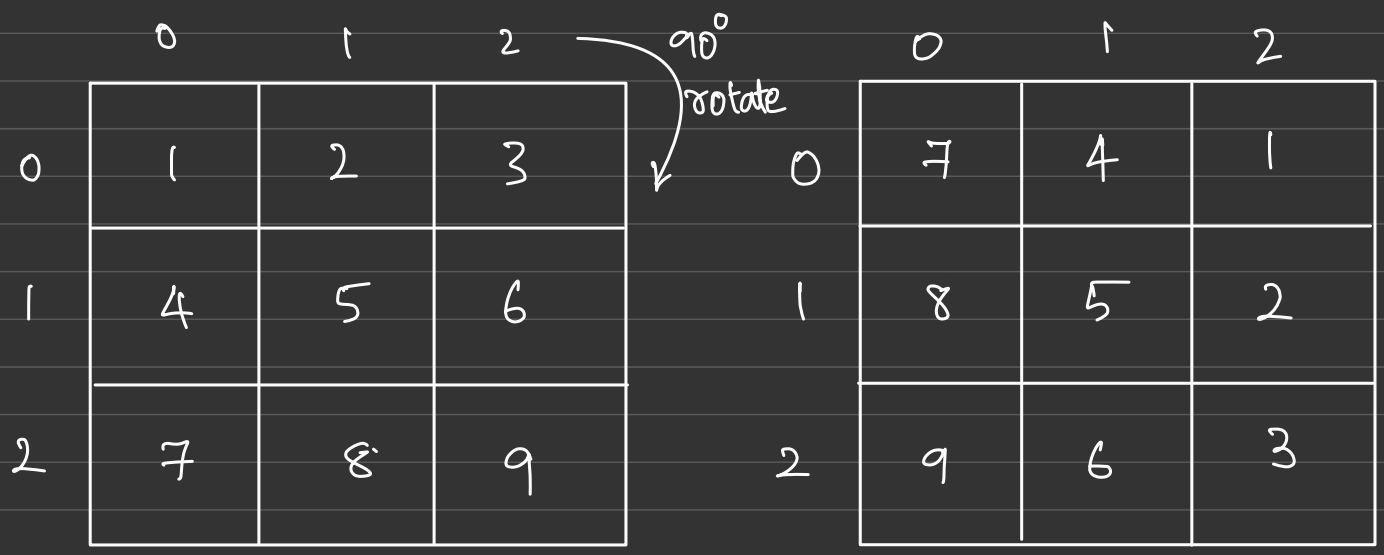
→ since `i` provides us the current row we are traversing

`mat[i]` provides us the ith row and the column count of the ith row would be `mat[i].length`

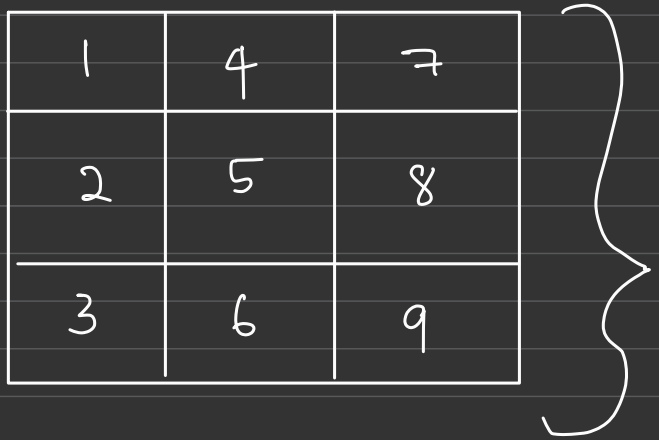
Time complexity is `O(i * j)` or `O(m*n)`

Leetcode - 48 - Rotate Image

Given an $n * n$ 2D matrix representing an image, rotate the image by 90° (clockwise) in-place.



90° rotation - we can **transpose** a matrix and when we do so we get the following.



if we carefully observe, **reversing** this matrix would give us our deserved output.

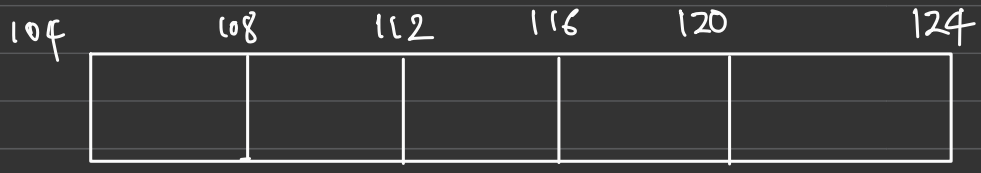
2 steps to follow:

Transpose ⟶ Reverse

Arrays

Behind The Scenes

`int[] arr = new int[5];` — 1D arrays

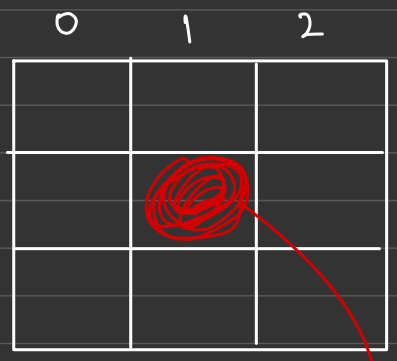


`int[][] arr = new int[3][3];`

Inside the memory, it is stored same as 1D arrays. However, there is a formula to calculate the next byte of integer in the memory.

`c * i + j`

where c is number of columns
i is row number
j is column number



To find the address for arr[1][1], it would be

$3 * 1 + 1 = 4$

As per the formula, column is mandatory while initializing an array.

2-D arrays row wise access

```
for(int i=0 ; i<row; i++)
{
   for(int j=0 ; j<col; j++)
   {
      print (arr[i][j];
   }
}
```

	0	1	2
0	(0,0) 10	(0,1) 20	(0,2) 30
1	(1,0) 40	(1,1) 50	(1,2) 60
2	(2,0) 11	(2,1) 21	(2,2) 31
3	(3,0) 41	(3,1) 51	(3,2) 61

2-D arrays column wise access

Here the column index remains same for each of the column (top-down printing).

```
for(int i=0; i<col; i++){
   for(int j=0; j<row; j++){
      print (arr[j][i]);
   }
}
```

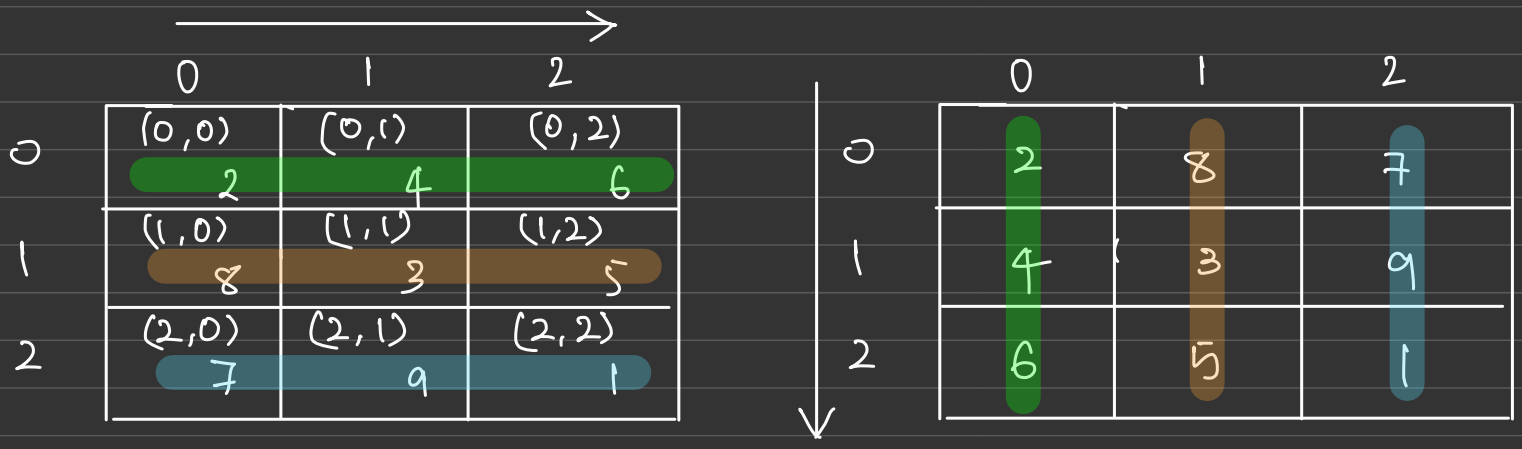
row value

column value

	0	1	2
0	(0,0) 10	(0,1) 20	(0,2) 30
1	(1,0) 40	(1,1) 50	(1,2) 60
2	(2,0) 11	(2,1) 21	(2,2) 31
3	(3,0) 41	(3,1) 51	(3,2) 61

Transpose of a matrix

Transpose - converts rows to columns and vice versa.



we are basically swapping the numbers when we say transpose.

arr[0][0] ⟷ arr[0][0]

arr[0][1] ⟷ arr[1][0]

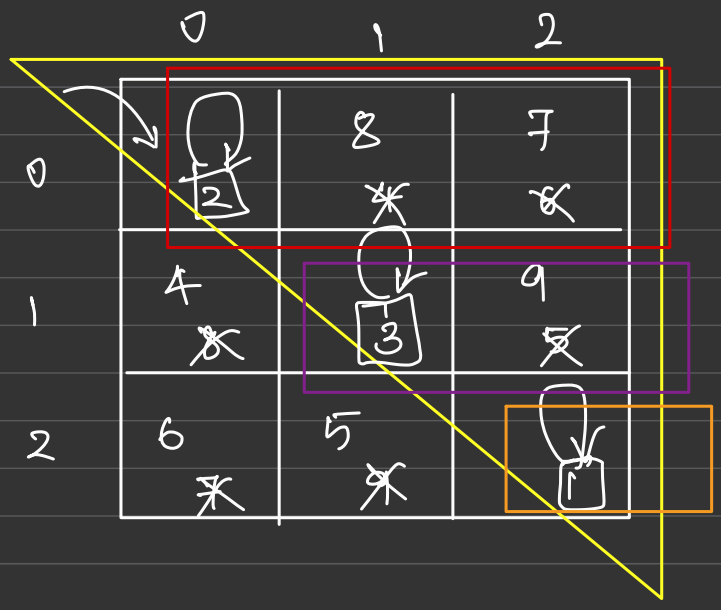
arr[0][2] ⟷ arr[2][0]

arr[1][0] ⟷ arr[0][1]

generalizing this, we can say we are swapping

arr[i][j] with arr[j][i]

while transposing, we might swap elements multiple times which would give us wrong answers.



(0,0)

a[0][0] ⟷ a[0][0]

(0,1)

a[0][1] ⟷ a[1][0]

(0,2)

a[0][2] ⟷ a[2][0]

(1,0)

a[1][0] ⟷ a[0][1] → this would swap back already swapped elements. So we need to avoid this.

we need to only process the right half triangle

The for-loop should be changed to

```
for(int i=0 ; i< row ; i++)
{
    for(int j=i ; j< col; j++)
    {

    }
}
```

Arrays assignments

1. **Key Pair** - Find two sum as in a pair of numbers that would equal to a target number.

input: arr = {1, 4, 45, 6, 10, 8}

target = 16

output: Yes → arr[3] + arr[4] = 6 + 10 = 16

Approaches

* we can use two for-loops by using two pointers. One on 0^{th} index and the other on 1^{st} index. Iterate the second pointer and check if the pair is equal to the target number

Time complexity - $O(n^2)$ - would get T.L.E

* or we can use two pointers on either side of a sorted array. Once sorted all smaller numbers would be on the left side and bigger ones on the right. Check if the pair equals the target. If it is greater than target reduce right pointer otherwise reduce the left pointer.

Time complexity - $O(n \cdot \log n)$

2. Find Pivot Index

Pivot index is the index where the sum of all the numbers strictly to the left of the index is equal to the sum of all the numbers exactly to the index's right.

input : arr = {1, 7, 3, 6, 5, 6}

Approaches

a. **Brute Force** - Find the sum of elements from the current index to its left and right. If sum are same we get our answer. We don't consider the current element in the sum.

Time complexity - $O(n^2)$

Space complexity - $O(1)$

b. **Prefix Sum** - Create 2 arrays one each to calculate the sum of left and right sum of the array

nums = {1, 7, 3, 6, 5, 6}						
lSum =	0	1	8	11	17	22
rSum =	27	20	17	11	6	0

This is called as **Prefix Sum** method

Formula would be :

```
lSum[i] = lSum[i-1] + nums[i-1];
```

```
rSum[i] = rSum[i+1] + nums[i+1];
```

3. Sort colors

Given an array nums with n objects colored red, white or blue, sort them in-place so that objects of the same color are adjacent

0 represents red

1 represents white

2 represents blue

i/p: nums = {2,0,2,1,1,0}

o/p: [0,0,1,1,2,2]

Approaches

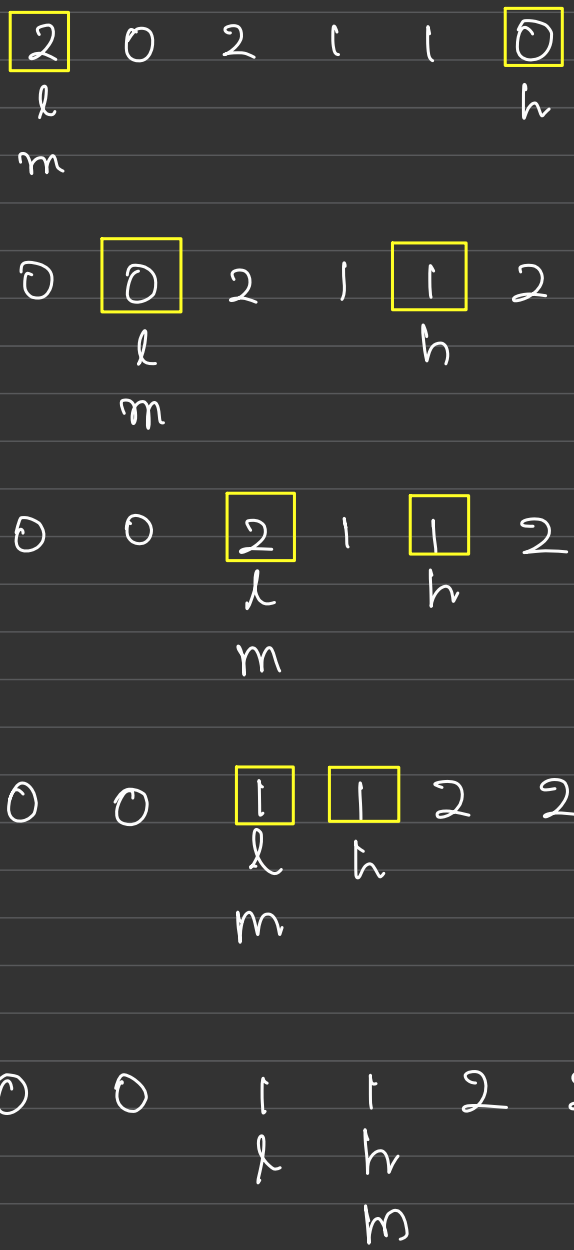
a). Using in-built sort() method.

T.C - O(n · logn) S.C - O(n)

b) Using counting method.- count the number of 0's, 1's and 2's and using a new array spread the number of 0's, 1's and 2's. But this uses another array but we need to solve it in-place.

c. **In-place approach**

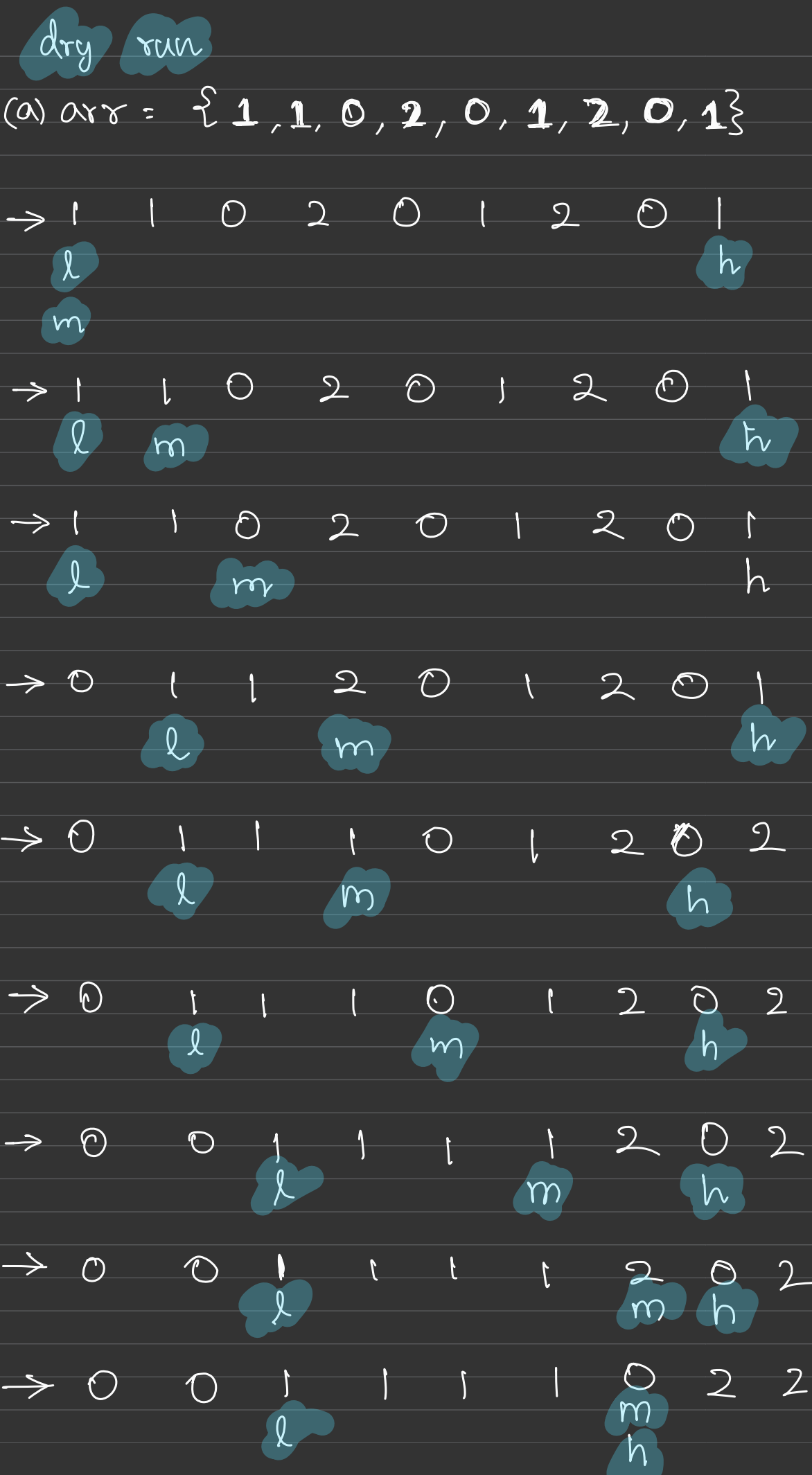
use 3-pointer approach, low, mid and high where low and mid points to first element and high points to last element.




```
int n = nums.length;
int lo=0, mi=0, hi = n-1;
while (mi<= hi){
  if(nums[mi] ==0){
     swap(nums, lo, mi);
     lo++;
     mi++;
  }
  else if(nums[mi] == 1){
     mi++;
  }
  else{
    swap(nums, hi, mi);
    hi--;
  }.
}
```

dry run

(a) arr = {1, 1, 0, 2, 0, 1, 2, 0, 1}



→ 0 0 0 1 1 1 1 2 2 — sorted array

l h m

(b) arr = {2, 0, 1}

→ 2 0 1

l h

m

→ 1 0 2

l h

m

→ 1 0 2 } to overcome this scenario we use $mi <= hi$ instead of $mi < hi$

l h

m

→ 0 1 2 → sorted array

h m

l

4. Missing Number

Return the only missing number in the range (0, n)

Approaches

a. Sorting method

sort the given array and check the current element with the index. If they match then continue, otherwise return the index.

T.C - O(n . log n)

S.C - O(1)

b. XOR method

In XOR, A ^ A = 0.

XOR all the array elements with all the elements in the range (0, n).

Ex: 9 ^ 6 ^ 4 ^ 2 ^ 3 ^ 5 ^ 7 ^ 0 ^ 1

(^) 0 ^ 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9

---

8 remains which is our missing element

T.C - O(n)

S.C - O(1)

5. Move all negative numbers to left side of an array

arr = {1, 2, -3, 4, -5, 6} - order need not be maintained.

Assumptions

a. Sort in ascending order.

T.C - O(n.logn) S.C - O(n)

b. Dutch National Flag algorithm / Two pointer approach

0	1	2	3	4	5
1	2	-3	4	-5	6
l					h

(maintains -ve) (maintains +ve)

```
if (a[l] < 0)
    l++;
else if (a[h] > 0)
    h--;
else
    swap(a, l, h);
```

dry run

→

1	2	-3	4	-5	6
l					h

→

1	2	-3	4	-5	6
l				h	

⇒

-5	2	-3	4	1	6
l				h	

→ -5 2 -3 4 1 6
l h

→ -5 2 -3 4 1 6
l h

→ -5 2 -3 4 1 6
l h

→ -5 -3 2 4 1 6
l h

→ -5 -3 2 4 1 6 → moved all negatives to left side of an array.
l
h

so our exit condition must be l <= h

T.C - O(n) S.C - O(1)

6. Find duplicate number

Eg. 1 3 4 2 2

o/p: 2

N+1 = 5 nums[i] ∈ [1, n]

N = 4

a. sort the Array

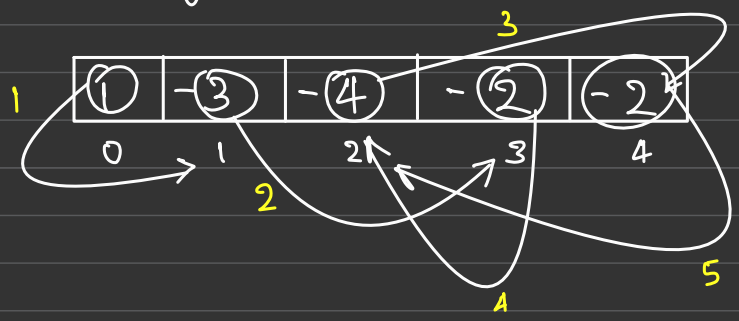
1 ② ② 3 4

if two adjacent elements then there are duplicates

T.C - O(n.logn) S.C - O(n)

b. Negative Marking method

* Consider the array element as an index.
* visit the index and mark visited (make it negative).
* already visited then return it.



T.C - O(n) S.C - O(1)

this approach modifies the given array which is not good.

c. Positioning method

nums - 1 3 4 2 2

consider the array elements as index and swap them to their respective index since nums[i] ∈ [1, n]

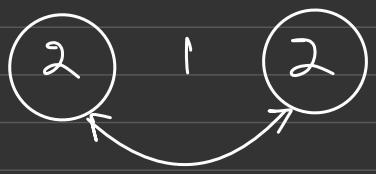
0 1 2 3 4

1 3 4 2 2

→ 3 1 4 2 2

→ 2 1 4 3 2

→ 4 1 2 3 2

→  3 4

both are same as it is **duplicate** so we return it.