1. Two Sum

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Question : Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int size = nums.length;
        int [] result  = new int[2];
        HashMap<Integer, Integer> hashMap = new HashMap<>();
        for(int i=0; i<size; i++){
            int key  = target - nums[i];
            if(hashMap.containsKey(key)){
                result[0] = i;
                result[1] = hashMap.get(key);
                break;
            }
            hashMap.put(nums[i], i);
        }
        Arrays.sort(result);
        return result;
    }
}
```

---

4. Median of Two Sorted Arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

You may assume **nums1** and **nums2** cannot be both empty.

Example 1:

```
nums1 = [1, 3]
nums2 = [2]

The median is 2.0
```

Example 2:

```
nums1 = [1, 2]
nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5
```

---

4. Median of Two Sorted Arrays Hard

5131

740

Favorite

Share There are two sorted arrays nums1 and nums2 of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

You may assume nums1 and nums2 cannot be both empty.

Example 1:

nums1 = [1, 3] nums2 = [2]

The median is 2.0 Example 2:

nums1 = [1, 2] nums2 = [3, 4]

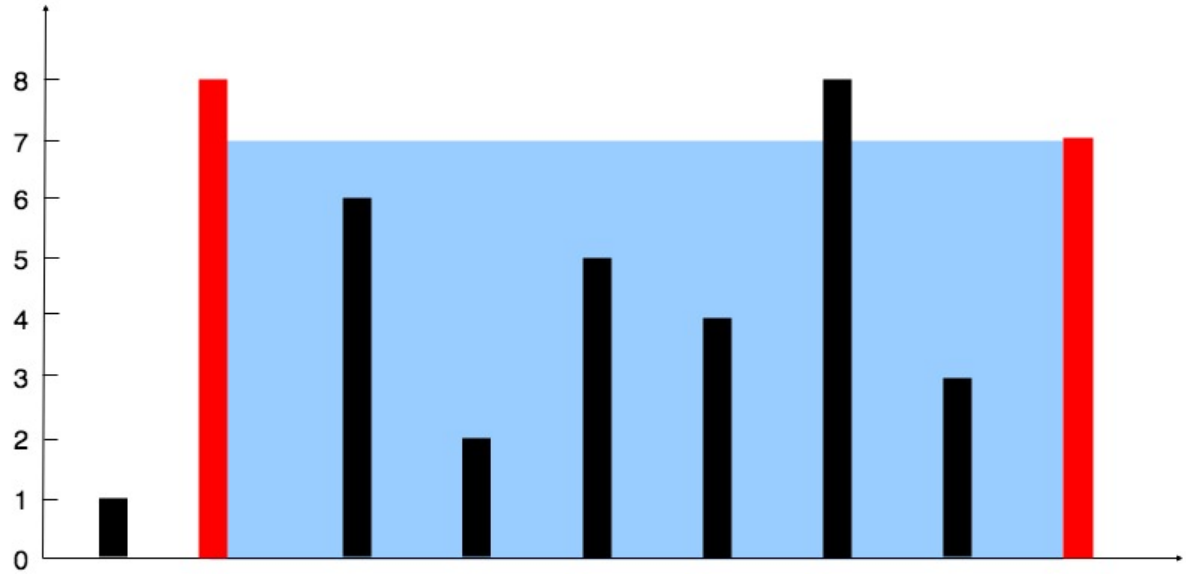
The median is (2 + 3)/2 = 2.5

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int x = nums1.length;
        int y = nums2.length;
        if(x > y){
            return findMedianSortedArrays(nums2, nums1);
        }
        int low = 0;
        int high = x;
        while(low <= high){
            int partitionX = (low + high)/2;
            int partitionY = (x + y + 1)/2 - partitionX;
            int maxLeftX = (partitionX == 0)? Integer.MIN_VALUE : nums1[partition
X -1];
            int minRightX = (partitionX == x)? Integer.MAX_VALUE : nums1[partitio
nX];

            int maxLeftY = (partitionY == 0)? Integer.MIN_VALUE : nums2[partition
Y - 1];
            int minRightY = (partitionY == y)? Integer.MAX_VALUE : nums2[partitio
nY];

            if((maxLeftX <= minRightY) && (maxLeftY <= minRightX)){
                if((x+y)%2 == 0){
                    return ((double) Math.max(maxLeftX, maxLeftY) + Math.min(minR
ightX, minRightY))/2;
                }else{
                    return (double)Math.max(maxLeftX, maxLeftY);
                }
            }else if (maxLeftX > minRightY){
                high = partitionX - 1;
            }else{
                low = partitionX + 1;
            }
        }
        return -1;
    }
}
```

11. Container With Most Water

Given *n* non-negative integers a_1, a_2, ..., a_n , where each represents a point at coordinate (i, a_i). *n* vertical lines are drawn such that the two endpoints of line *i* is at (i, a_i) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.



The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example:

```
Input: [1,8,6,2,5,4,8,3,7]
Output: 49
```

---

11. Container With Most Water Medium

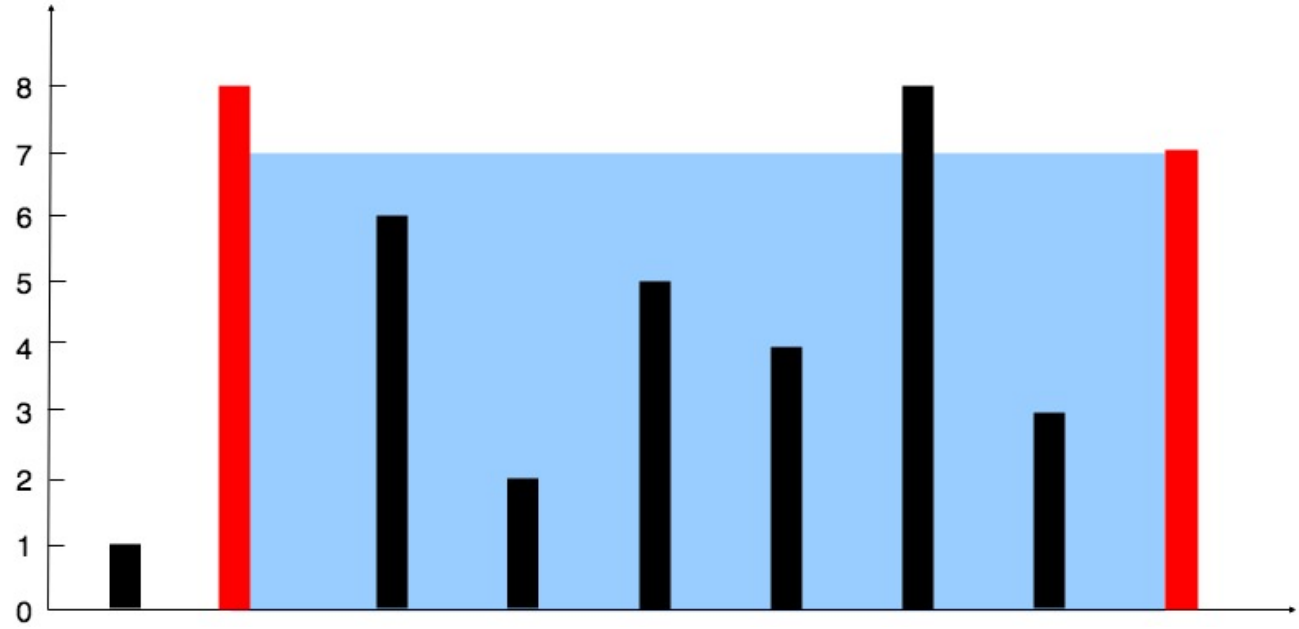
4774

512

Add to List

Share Given n non-negative integers a1, a2, ..., an , where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and n is at least 2.



How this approach works?

Initially we consider the area constituting the exterior most lines. Now, to maximize the area, we need to consider the area between the lines of larger lengths. If we try to move the pointer at the longer line inwards, we won't gain any increase in area, since it is limited by the shorter line. But moving the shorter line's pointer could turn out to be beneficial, as per the same argument, despite the reduction in the width. This is done since a relatively longer line obtained by moving the shorter line's pointer might overcome the reduction in area caused by the width reduction.

Proof

```
public int maxArea(int[] height) {
int len = height.length, low = 0, high = len -1 ;
int maxArea = 0;
while (low < high) {
maxArea = Math.max(maxArea, (high - low) * Math.min(height[low], height[high]));
if (height[low] < height[high]) {
low++;
} else {
high--;
}
}
return maxArea;
```

} Here is the proof. Proved by contradiction:

Suppose the returned result is not the optimal solution. Then there must exist an optimal solution, say a container with a_ol and a_or (left and right respectively), such that it has a greater volume than the one we got. Since our algorithm stops only if the two pointers meet. So, we must have visited one of them but not the other. WLOG, let's say we visited a_ol but not a_or. When a pointer stops at a_ol, it won't move until

The other pointer also points to a_ol. In this case, iteration ends. But the other pointer must have visited a_or on its way from right end to a_ol. Contradiction to our assumption that we didn't visit a_or.

The other pointer arrives at a value, say a_rr, that is greater than a_ol before it reaches a_or. In this case, we does move a_ol. But notice that the volume of a_ol and a_rr is already greater than a_ol and a_or (as it is wider and heigher), which means that a_ol and a_or is not the optimal solution -- Contradiction!

Both cases arrive at a contradiction.

```
class Solution {
    public int maxArea(int[] height) {
        int high = height.length - 1;
        int low = 0;
        int area = 0;
        while(low<high){
            area = Math.max(area, Math.min(height[low], height[high]) * (high - l
ow));
            if(height[low] < height[high])
                low++;
            else
                high--;
        }
        return area;
    }
}
```

---

16. 3Sum Closest

Given an array `nums` of *n* integers and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example:

```
Given array nums = [-1, 2, 1, -4], and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Given an array nums of n integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example:

Given array nums = [-1, 2, 1, -4], and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        int size = nums.length;
        int result = nums[0] + nums[1] + nums[size -1];
        Arrays.sort(nums);
        for(int i=0; i<size-2; i++){
            int l = i+1;
            int h = size-1;
            int x = nums[i];
            while(l<h){
                int sum = x + nums[l] + nums[h];
                if(sum == target)  // This inclued to make more efficeint code.
                    return sum;
                if(sum < target){
                    l++;
                }else{
                    h--;
                }
                if(Math.abs(sum - target) < Math.abs(result - target))
                    result = sum;
            }
        }
        return result;
    }
}
```

18. 4Sum

Given an array `nums` of *n* integers and an integer `target`, are there elements *a*, *b*, *c*, and *d* in `nums` such that *a* + *b* + *c* + *d* = `target`? Find all unique quadruplets in the array which gives the sum of `target`.

Note:

The solution set must not contain duplicate quadruplets.

Example:

```
Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:
[
  [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]
]
```

18. 4Sum Medium

1282

256

Favorite

Share Given an array nums of n integers and an integer target, are there elements a, b, c, and d in nums such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

Note:

The solution set must not contain duplicate quadruplets.

Example:

Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is: [[-1, 0, 0, 1], [-2, -1, 1, 2], [-2, 0, 0, 2]]

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;
        HashMap<Integer, Pair> hashMap = new HashMap<>();
        for(int i=0;i<n-1; i++){
            for(int j =i; j<n;j++){
                int sum  = nums[i] + nums[j];
                hashMap.put(sum, new Pair(i, j));
            }
        }
        for(int i=0; i<n-1; i++){
            for(int j =i; j<n; j++){
                int x = target - (nums[i] + nums[j]);
                if(hashMap.containsKey(x)){
                    Pair  pair = hashMap.get(x);
                    if(pair.x != i && pair.x != j && pair.y != i && pair.y != j){
                        result.add(Arrays.asList(nums[i], nums[j], nums[pair.x],
nums[pair.y]));
                    }
                }
            }
        }
        return result;
    }
}
class Pair{
    int x;
    int y;
    Pair(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

OutPut

```
Your input
[1,0,-1,0,-2,2]
0
Output
[[1,1,0,-2],[1,0,-1,0],[1,-1,-2,2],[1,-2,-1,2],[1,2,-1,-2],[0,0,-2,2],[0,0,-2,2],
[0,-2,0,2],[0,2,0,-2],[-1,-1,0,2],[-1,-2,1,2],[0,0,-2,2]]
Expected
[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

So above solution is also count duplicates. Need to remove duplicates.

---

25. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.

Example:

Given this linked list: `1->2->3->4->5`

For *k* = 2, you should return: `2->1->4->3->5`

For *k* = 3, you should return: `3->2->1->4->5`

Note:

- Only constant extra memory is allowed.
- You may not alter the values in the list's nodes, only nodes itself may be changed.

---

25. Reverse Nodes in k-Group Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

Example:

Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if(head == null || k == 1)
            return head;
        ListNode newHead = head;
        ListNode firstNode = head;
        ListNode prevNode = null;
        ListNode kthNode = null;
        while(firstNode != null){
            kthNode = firstNode;
            int count = k;
            while(count >0 && kthNode != null ){
                kthNode = kthNode.next;
                count--;
            }
            if(count > 0)
                break;
            ListNode tempNode = reverseList(firstNode, kthNode);
            if(newHead == head)
                newHead = tempNode;
            else
                prevNode.next = tempNode;
            prevNode =  firstNode;
            firstNode = kthNode;

        }
        return newHead;
    }
    private ListNode reverseList(ListNode start, ListNode last){
        ListNode cur = start;
        ListNode prev = last;
        ListNode next = null;
        while(cur != last){
            next = cur.next;
            cur.next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }
}
```

Second Variation Reverse a Linked List in groups of given size | Set 1

```
Node reverse(Node head, int k)
{
   Node current = head;
   Node next = null;
   Node prev = null;

   int count = 0;

   /* Reverse first k nodes of linked list */
   while (count < k && current != null)
   {
       next = current.next;
       current.next = prev;
       prev = current;
       current = next;
       count++;
   }

   /* next is now a pointer to (k+1)th node
      Recursively call for the list starting from current.
      And make rest of the list as next of first node */
   if (next != null)
      head.next = reverse(next, k);

   // prev is now head of input list
   return prev;
}
```

---

31. Next Permutation

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be **in-place (http://en.wikipedia.org/wiki/In-place_algorithm)** and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

---

31. Next Permutation Medium

2603

880

Add to List

Share Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2 3,2,1 → 1,2,3 1,1,5 → 1,5,1

```
class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length -2;
        while( i>=0 && nums[i+1] <= nums[i]){
            i--;
        }
        if(i >= 0){
            int j = nums.length -1;
            while(j> i && nums[j] <= nums[i]){ // Need equality because 1,5,1 tes
t case will failed.
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i+1);
    }
    private void swap(int [] nums, int i, int j){
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
    private void reverse(int [] nums, int start){
        int i = start, j = nums.length - 1;
        while(i < j){
            swap(nums, i, j);
            i++;
            j--;
        }
    }
}
```

34. Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return `[-1, -1]`.

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

---

34. Find First and Last Position of Element in Sorted Array Medium

2413

111

Add to List

Share Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8 Output: [3,4] Example 2: Input: nums = [5,7,7,8,8,10], target = 6 Output: [-1,-1]

```
class Solution {
    public int [] searchRange(int [] array, int target){
        int firstPosition = getFirstPosition(array, 0, array.length -1, target);
        int lastPosition = getLastPosition(array, 0, array.length -1, target);
        return new int[]{firstPosition, lastPosition};
    }

    public int getFirstPosition(int []array, int start, int end, int target){
        if(start <= end){
            int mid = start + (end - start) / 2;
            if( array[mid] == target  && ( mid == 0 || array[mid -1] < target))
                return mid;
            if(target > array[mid]){
                return getFirstPosition(array, mid + 1, end, target);
            }else{
                return getFirstPosition(array, start, mid -1, target);
            }
        }
        return -1;
    }

    public int getLastPosition(int [] array, int start, int end, int target){
        if(start <= end){
            int mid = start + (end - start) / 2;
            if(array[mid] == target && (mid == array.length -1 || array[mid] < ar
ray[mid + 1]))
                return mid;
            if(array[mid] > target)
                return getLastPosition(array, start, mid -1, target);
            else
                return getLastPosition(array, mid +1 , end, target);
        }
        return -1;
    }
}
```

Input: nums = [5,7,7,8,8,10], target = 6 Output: [-1,-1]``

---

42. Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

Example:

```
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

---

42. Trapping Rain Water Hard

5341

104

Add to List

Share Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Example:

Input: [0,1,0,2,1,0,1,3,2,1,2,1] Output: 6

```
class Solution {
    public int trap(int[] height) {
        Stack<Integer> stack = new Stack<>();
        int trapWaterSum =0;
        int len = height.length;
        for(int i=0; i < len; i++){
            if(stack.isEmpty() || height[stack.peek()] >= height[i]){
                stack.push(i);
            }else{
                while(height[stack.peek()] < height[i]){
                    int popValue = stack.pop();
                    if(stack.isEmpty()){ // not able hold water. Please refer the
image.
                        break;
                    }
                    int peekValue = stack.peek();
                    int min = Math.min(height[peekValue], height[i]);
                    trapWaterSum += (min - height[popValue]) * ( i - (peekValue +
1));
                }
                stack.push(i);
            }
        }
        return trapWaterSum;
    }
}
```

56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

Example 1:

```
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

56. Merge Intervals Medium

3170

242

Add to List

Share Given a collection of intervals, merge all overlapping intervals.

Example 1:

Input: [[1,3],[2,6],[8,10],[15,18]] Output: [[1,6],[8,10],[15,18]] Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6]. Example 2:

Input: [[1,4],[4,5]] Output: [[1,5]] Explanation: Intervals [1,4] and [4,5] are considered overlapping.

```
class Solution {
    public int[][] merge(int[][] intervals) {
        List<Interval> intervalList = new ArrayList<>();
        int size = intervals.length;
        for(int i=0; i<size; i++){
            intervalList.add(new Interval(intervals[i][0], intervals[i][1]));
        }
        Collections.sort(intervalList);
        for(int i = 0; i < intervalList.size() - 1 ; ){
            int j = i + 1;
            if(intervalList.get(i).finishTime >= intervalList.get(j).startTime){
                if(intervalList.get(i).finishTime < intervalList.get(j).finishTim
e){
                    intervalList.get(i).finishTime = intervalList.get(j).finishTi
me;
                }
                intervalList.remove(j);
            }else{
                i++;
            }
        }
        int [][] newInterval = new int[intervalList.size()][2];
        int i=0;
        for(Interval interval : intervalList){
            newInterval[i][0] = interval.startTime;
            newInterval[i][1] = interval.finishTime;
            i++;
        }
        return newInterval;
    }
}

class Interval implements Comparable<Interval>{
    int startTime;
    int finishTime;
    public Interval(int startTime, int finishTime){
        this.startTime =  startTime;
        this.finishTime = finishTime;
    }
    public int compareTo(Interval interval){
        return this.startTime - interval.startTime;
    }
}
```

63. Unique Paths II

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

Note: *m* and *n* will be at most 100.

Example 1:

```
Input:
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
Output: 2
Explanation:
There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

---

63. Unique Paths II Medium

1266

209

Add to List

Share A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?


```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        int [][] array = new int[m][n];
        for(int i=0; i<m; i++){
            if(obstacleGrid[i][0] == 1){
                while(i < m){
                    array[i][0] = 0;
                    i++;
                }
            }else{
                array[i][0] = 1;
            }
        }

        for(int j=0; j<n ;j++){
            if(obstacleGrid[0][j] == 1){
                while(j < n){
                    array[0][j] = 0;
                    j++;
                }
            }else{
                array[0][j] =1;
            }
        }

        for(int i=1; i<m; i++){
            for(int j =1; j<n;j++){
                if(obstacleGrid[i][j] == 1){
                    array[i][j] = 0;
                }else{
                    array[i][j] = array[i-1][j] + array[i][j-1];
                }
            }
        }
        return array[m-1][n-1];
    }
}
```

80. Remove Duplicates from Sorted Array II

Given a sorted array *nums*, remove the duplicates **in-place** (https://en.wikipedia.org/wiki/In-place_algorithm) such that duplicates appeared at most *twice* and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place (https://en.wikipedia.org/wiki/In-place_algorithm)** with O(1) extra memory.

Example 1:

```
Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being
1, 1, 2, 2 and 3 respectively.

It doesn't matter what you leave beyond the returned length.
```

Example 2:

```
Given nums = [0,0,1,1,1,1,2,3,3],

Your function should return length = 7, with the first seven elements of nums
 being modified to 0, 0, 1, 1, 2, 3 and 3 respectively.

It doesn't matter what values are set beyond the returned length.
```

Clarification:

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means modification to the input array will be known to the caller as well.

Internally you can think of this:

```
// nums is passed in by reference. (i.e., without making a copy)
int len = removeDuplicates(nums);

// any modification to nums in your function would be known by the caller.
// using the length returned by your function, it prints the first len elements.
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

---

80. Remove Duplicates from Sorted Array II Medium

875

635

Add to List

Share Given a sorted array nums, remove the duplicates in-place such that duplicates appeared at most twice and return the new length.

Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

Example 1:

Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being 1, 1, 2, 2 and 3 respectively.

It doesn't matter what you leave beyond the returned length. Example 2:

Given nums = [0,0,1,1,1,1,2,3,3],

Your function should return length = 7, with the first seven elements of nums being modified to 0, 0, 1, 1, 2, 3 and 3 respectively.

It doesn't matter what values are set beyond the returned length. Clarification:

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by reference, which means modification to the input array will be known to the caller as well.

Internally you can think of this:

// nums is passed in by reference. (i.e., without making a copy) int len = removeDuplicates(nums);

// any modification to nums in your function would be known by the caller. // using the length returned by your function, it prints the first len elements. for (int i = 0; i < len; i++) { print(nums[i]); }

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int size = nums.length;
        int j= 1, count = 1;
        for(int i=1; i<size; i++){
            if(nums[i-1] == nums[i]){
                count++;
            }else{
                count =1;
            }
            if(count<=2){
                nums[j++] = nums[i];
            }

        }
        return j;
    }
}
```

---

91. Decode Ways

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given a **non-empty** string containing only digits, determine the total number of ways to decode it.

Example 1:

```
Input: "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

Example 2:

```
Input: "226"
Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
```

---

91. Decode Ways Medium

2051

2327

Add to List

Share A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1 'B' -> 2 ... 'Z' -> 26 Given a non-empty string containing only digits, determine the total number of ways to decode it.

Example 1:

Input: "12" Output: 2 Explanation: It could be decoded as "AB" (1 2) or "L" (12). Example 2:

Input: "226" Output: 3 Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

```
class Solution {
    public int numDecodings(String s) {
        if(s.charAt(0) == '0')
            return 0;
        int [] count = new int[s.length()];
        count[0] = 1;
        for(int i=1; i< s.length(); i++){
            int value = s.charAt(i) - '0';
            if(value >=1 && value <= 9){
                count[i] = count[i-1];
            }
            int preValue = s.charAt( i -1) - '0';
            if(preValue !=0 ){
                int twoDigit = Integer.parseInt(s.substring(i-1, i +1));
                if(twoDigit >= 10 && twoDigit <=26){
                    count[i] = i !=1 ? count[i-2] + count[i] : count[i] +1;
                }
            }
        }
        return count[s.length() -1];
    }
}
```

Output: 3 Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

---

100. Same Tree

Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:

```
Input:     1         1
          / \       / \
         2   3     2   3

        [1,2,3],   [1,2,3]

Output: true
```

Example 2:

```
Input:     1         1
          /           \
         2             2

        [1,2],     [1,null,2]

Output: false
```

Example 3:

```
Input:     1         1
          / \       / \
         2   1     1   2

        [1,2,1],   [1,1,2]

Output: false
```

---

100. Same Tree Easy

1432

46

Favorite

Share Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:

Input: 1 1 / \ /
2 3 2 3

```
    [1,2,3],   [1,2,3]
```

Output: true Example 2:

Input: 1 1 /
2 2

```
    [1,2],     [1,null,2]
```

Output: false Example 3:

Input: 1 1 / \ /
2 1 1 2

```
    [1,2,1],   [1,1,2]
```

Output: false

Recursion Solution:

```
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p==null && q == null)
            return true;
        if(p!=null && q!= null){
            return p.val == q.val && isSameTree(p.left, q.left) && isSameTree(p.r
ight, q.right);
        }
        return false;
    }
}
```

Iterative Solution:

```
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == null && q == null){
            return true;
        }
        //structure
        if(p == null || q == null){
            return false;
        }
        //val
        if(p.val != q.val){
            return false;
        }

        Stack<TreeNode> stk1 = new Stack<TreeNode>();
        Stack<TreeNode> stk2 = new Stack<TreeNode>();
        stk1.push(p);
        stk2.push(q);

        while( !stk1.isEmpty() && !stk2.isEmpty() ){
            TreeNode tn1 = stk1.pop();
            TreeNode tn2 = stk2.pop();

            if(tn1.val != tn2.val){
                return false;
            }
            //structure different
            if(tn1.left == null && tn2.left != null){
                return false;
            }else if (tn1.left != null && tn2.left == null){
                return false;
            }else if (tn1.right == null && tn2.right != null){
                return false;
            }else if (tn1.right != null && tn2.right == null) {
                return false;
            }

            if(tn1.left != null && tn2.left != null){
                stk1.push(tn1.left);
                stk2.push(tn2.left);
            }
```

```
            if(tn1.right != null && tn2.right != null){
                stk1.push(tn1.right);
                stk2.push(tn2.right);
            }
        }

        return true;

    }
}
```

122. Best Time to Buy and Sell Stock II

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 =
             Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit =
```

Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 =
             Note that you cannot buy on day 1, buy on day 2 and sell them later, a
             engaging multiple transactions at the same time. You must sell before
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

22. Best Time to Buy and Sell Stock II Easy

1302

1430

Favorite

Share Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

Input: [7,1,5,3,6,4] Output: 7 Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4. Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3. Example 2:

Input: [1,2,3,4,5] Output: 4 Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4. Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again. Example 3:

Input: [7,6,4,3,1] Output: 0 Explanation: In this case, no transaction is done, i.e. max profit = 0.

Is this question a joke?

public class Solution { public int maxProfit(int[] prices) { int total = 0; for (int i=0; i< prices.length-1; i++) { if (prices[i+1]>prices[i]) total += prices[i+1]-prices[i]; }

```
return total;
```

} A simple code like this. The designer of this question must thought of something too complicated.

---

125. Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Note: For the purpose of this problem, we define empty string as valid palindrome.

Example 1:

```
Input: "A man, a plan, a canal: Panama"
Output: true
```

Example 2:

```
Input: "race a car"
Output: false
```

---

125. Valid Palindrome Easy

887

2448

Add to List

Share Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Note: For the purpose of this problem, we define empty string as valid palindrome.

Example 1:

Input: "A man, a plan, a canal: Panama" Output: true Example 2:

Input: "race a car" Output: false

```
public class Solution {
    public boolean isPalindrome(String string) {
        if(string.isEmpty()){
            return true;
        }
        int start =0;
        int end = string.length()-1;
        while (start<=end){
            char startChar = string.charAt(start);
            char endChar = string.charAt(end);
            if(!Character.isLetterOrDigit(startChar)){
                start++;
            }else if(!Character.isLetterOrDigit(endChar)){
                end--;
            }else {
                if(Character.toLowerCase(startChar)!=Character.toLowerCase(endCha
r)){
                    return false;
                }
                start++;
                end--;
            }
        }
        return true;

    }
}
```

---

146. LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache (https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU). It should support the following operations: `get` and `put`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`put(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a **positive** capacity.

Follow up:
Could you do both operations in **O(1)** time complexity?

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1. put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a positive capacity.

Follow up: Could you do both operations in O(1) time complexity?

Example:

LRUCache cache = new LRUCache(2 /* capacity */);

cache.put(1, 1); cache.put(2, 2); cache.get(1); // returns 1 cache.put(3, 3); // evicts key 2 cache.get(2); // returns -1 (not found) cache.put(4, 4); // evicts key 1 cache.get(1); // returns -1 (not found) cache.get(3); // returns 3 cache.get(4); // returns 4

```
class DLinkedListNode {
    int key;
    int value;
    DLinkedListNode prev;
    DLinkedListNode next;
    public String toString() {
        return "key: " + this.key + ", value: " + this.value ;
    }
}

class LRUCache {
    HashMap<Integer, DLinkedListNode> hashMap;
    DLinkedListNode head = null, tail = null;
    private int size;
    private int capacity;
    public LRUCache(int capacity) {
        hashMap = new HashMap<>();
        this.capacity = capacity;
    }
    private void addNodeAtTop(DLinkedListNode node){
        node.next = head;
        node.prev = null;
        if(head != null)
            head.prev = node;
        head = node;
        if(tail == null){
            tail = node;
        }
    }

    private void removeNode(DLinkedListNode node){
        if(node.prev != null){
            node.prev.next = node.next;
        }else{
            head = node.next;
        }

        if(node.next != null){
            node.next.prev = node.prev;
        }else{
            tail = node.prev;
        }
    }

    public int get(int key) {
        if(hashMap.containsKey(key)){
            DLinkedListNode node = hashMap.get(key);
            removeNode(node);
            addNodeAtTop(node);
            return node.value;
        }
        return -1;
    }

    public void put(int key, int value) {
        if(hashMap.containsKey(key)){
            DLinkedListNode node = hashMap.get(key);
```

```
            node.value = value;
            removeNode(node);
            addNodeAtTop(node);
        }else{
            DLinkedListNode  newNode = new DLinkedListNode();
            newNode.key = key;
            newNode.value = value;
            newNode.prev = null;
            newNode.next = null;
            if(hashMap.size() >= capacity){
                hashMap.remove(tail.key);
                removeNode(tail); // Order is important. First remove tails from
HashMap then remove from DDL
            }
            addNodeAtTop(newNode);
            hashMap.put(key, newNode);
        }
    }
}
```

---

189. Rotate Array

Given an array, rotate the array to the right by *k* steps, where *k* is non-negative.

Example 1:

```
Input: [1,2,3,4,5,6,7] and k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: [-1,-100,3,99] and k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Note:

- Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
- Could you do it in-place with O(1) extra space?

---

89. Rotate Array Easy

2097

732

Add to List

Share Given an array, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: [1,2,3,4,5,6,7] and k = 3 Output: [5,6,7,1,2,3,4] Explanation: rotate 1 steps to the right: [7,1,2,3,4,5,6] rotate 2 steps to the right: [6,7,1,2,3,4,5] rotate 3 steps to the right: [5,6,7,1,2,3,4] Example 2:

Input: [-1,-100,3,99] and k = 2 Output: [3,99,-1,-100] Explanation: rotate 1 steps to the right: [99,-1,-100,3] rotate 2 steps to the right: [3,99,-1,-100] Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem. Could you do it in-place with O(1) extra space?

```
class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        rotateArray(nums, 0, nums.length -1);
        rotateArray(nums, 0, k-1);
        rotateArray(nums, k, nums.length -1);
    }
    public void rotateArray(int [] nums, int start, int end){
        while(start < end){
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }
}
```

Explanation

This is a great solution, but it is weird that no one tried to prove the solution in a mathematical way in this discussion. I will try to prove its correctness, as I feel more comfortable to understand it in this way, other than a bunch of examples.

k %= nums.length; makes sure that k is less than the length of the array. There are two parts of the array that we need to care about:

Goal1: Assume range1 = [0, n - k - 1]. Members of this range only need to move to the right by k steps. For any member i in this range, the targeted position is i + k. In other words, we need to move every member i in range1 to position i + k Goal2: Assume range2 = [n - k, n - 1]. Members of this range will have to move beyond the boundary of the array, thus for any member i in this range, the targeted position is (i + k) % n , which is equivalent to i + k - n. In other words, we need to move every member i in range2 to position i + k - n. For any member i, after the first reverse(...) call, its new position j will be n - i - 1.

By replacing the i in n - i - 1, we can calculate the new value of range1 from [0, n - k - 1] to [n - (n - k - 1) - 1, n - 1], which is [k, n - 1] (as range1 is reversed, its left and right bounds also need to be reversed). The similar procedure can be applied to range2, the new range2 now becomes [0, k - 1]

For any member j in the new range2, the second reverse(...) call will assign the member j to a new position k - 1 - j. Notice that, this j is actually equal to n - i - 1, where i is the original position. So, the new position now becomes k - 1 - (n - i - 1) = k - n + i, which meets Goal2.

The similar procedure can be applied to range1, with the third reverse(...) call, which will meet Goal1.

200. Number of Islands

Given a 2d grid map of `'1'` s (land) and `'0'` s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input:
11110
11010
11000
00000

Output: 1
```

Example 2:

```
Input:
11000
11000
00100
00011

Output: 3
```

---

200. Number of Islands Medium

3452

127

Favorite

Share Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input: 11110 11010 11000 00000

Output: 1 Example 2:

Input: 11000 11000 00100 00011

Output: 3

```
**Following is good solution, just remove the all Island by making it 0 while vis
iting. Nice solution :)
Op's thought process - Just drown every island - one by one!!**
```

public class Solution {

private int n; private int m;

public int numIslands(char[][] grid) { int count = 0; n = grid.length; if (n == 0) return 0; m = grid[0].length; for (int i = 0; i < n; i++){ for (int j = 0; j < m; j++) if (grid[i][j] == '1') { DFSMarking(grid, i, j); ++count; } } return count; }

private void DFSMarking(char[][] grid, int i, int j) { if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') return; grid[i][j] = '0'; DFSMarking(grid, i + 1, j); DFSMarking(grid, i - 1, j); DFSMarking(grid, i, j + 1); DFSMarking(grid, i, j - 1); } }

```
My Solution:
```

class Solution { public int numIslands(char[][] grid) { int count = 0; int ROW = grid.length; if(ROW == 0) return 0; int COL = grid[0].length; boolean [][] visited = new boolean[ROW][COL]; for(int i = 0; i < ROW; i++) for(int j = 0; j < COL; j++) visited[i][j] = false; for(int i=0; i<ROW; i++){ for(int j=0; j<COL; j++){ if(grid[i][j] == '1' && !visited[i][j]){ dfs(grid, i, j, visited); ++count; } } } return count; }

```
private void dfs(char [][] grid, int row, int col, boolean [][] visited){
    int rowNbr[] = new int[] { -1,  0,  0,  1 };
    int colNbr[] = new int[] {  0, -1,  1,  0 };
    visited[row][col] = true;
    for(int k = 0; k < 4; k++){
         int x = rowNbr[k]+row;
        int y = colNbr[k]+col;
        if(isSafe(grid, x, y, visited))
            dfs(grid, x, y, visited);
    }
}
private boolean isSafe(char [][] grid, int row, int col, boolean [][] visited){
    int ROW = grid.length;
    int COL = grid[0].length;
    if(row >= 0 && col >= 0 && row < ROW && col < COL && grid[row][col] == '1' &&
!visited[row][col])
        return true;
    return false;
}
```

}

283. Move Zeroes

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Example:

```
Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

---

283. Move Zeroes Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Example:

Input: [0,1,0,3,12] Output: [1,3,12,0,0] Note:

You must do this in-place without making a copy of the array. Minimize the total number of operations.

```
class Solution {
    public void moveZeroes(int[] nums) {
        int j = 0;
        int size = nums.length;
        for(int i=0; i<size; i++){
            if(nums[i] != 0)
                nums[j++] = nums[i];
        }
        while(j < size){
            nums[j++] = 0;
        }
    }
}
```

---

348. Design Tic-Tac-Toe

Design a Tic-tac-toe game that is played between two players on a *n* x *n* grid.

You may assume the following rules:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves is allowed.
3. A player who succeeds in placing *n* of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

```
Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.

TicTacToe toe = new TicTacToe(3);

toe.move(0, 0, 1); -> Returns 0 (no one wins)
|X| | |
| | | |    // Player 1 makes a move at (0, 0).
| | | |

toe.move(0, 2, 2); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 2 makes a move at (0, 2).
| | | |

toe.move(2, 2, 1); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 1 makes a move at (2, 2).
| | |X|

toe.move(1, 1, 2); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 2 makes a move at (1, 1).
| | |X|

toe.move(2, 0, 1); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 1 makes a move at (2, 0).
|X| |X|

toe.move(1, 0, 2); -> Returns 0 (no one wins)
|X| |O|
|O|O| |    // Player 2 makes a move at (1, 0).
|X| |X|

toe.move(2, 1, 1); -> Returns 1 (player 1 wins)
|X| |O|
|O|O| |    // Player 1 makes a move at (2, 1).
|X|X|X|
```

Follow up:
Could you do better than O(n^2) per `move()` operation?

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:

A move is guaranteed to be valid and is placed on an empty block. Once a winning condition is reached, no more moves is allowed. A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

```
class TicTacToe {

    /** Initialize your data structure here. */
    int rows [], cols [], diagonal, anti_diagonal;
    public TicTacToe(int n) {
        rows = new int[n];
        cols = new int[n];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */

    /**
    Explaination:
    Here we need only four variables int [] rows, int [] cols, int diagonal, int antiDiagonal; There these variables are used to take care of Number of ways any player can win.
    For example If Player1 is trying to crease cross for rows[0], then for win its value should be rows[0] = n, where n is the size of row or col of tic-tac-toe. If player2 made any changes in rows[0] then player1 can't win by rows[0].
    */
    public int move(int row, int col, int player) {
        int size = cols.length;
        int addNum = player == 1 ? 1 : -1;
        rows[row] += addNum;// Every add of -1 or 1 is the entry by player1 or player2. If rows[1] =2, if player2 enters in rows[1], then Player 1 can win by rows 1.
        cols[col] += addNum;
        if(row == col){
            diagonal += addNum;
        }
        if (col == size - 1 - row){
            anti_diagonal += addNum;
        }
        if(Math.abs(rows[row]) == size || Math.abs(cols[col]) == size || Math.abs(diagonal) == size || Math.abs(anti_diagonal) == size){
            return player;
        }

        return 0;
    }
}

/**
 * Your TicTacToe object will be instantiated and called as such:
 * TicTacToe obj = new TicTacToe(n);
 * int param_1 = obj.move(row,col,player);
 */
```

387. First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.

s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

---

387. First Unique Character in a String Easy

1413

98

Add to List

Share Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

s = "leetcode" return 0.

s = "loveleetcode", return 2. Note: You may assume the string contain only lowercase letters.

```
class Solution {
    public int firstUniqChar(String s) {
        int size = s.length();
        int [] freq = new int[26];
        for(int i=0; i<size; i++){
            int index = s.charAt(i) - 'a';
            if(freq[index] != 0){
                freq[index] = -1;
            }else{
                freq[index] = i + 1;//to handle the case when 'leetcode', where i
ndex['l' -'a'] = 0 (index of 'l').
            }
        }
        int min = Integer.MAX_VALUE;
        for(int i=0; i< 26; i++){
            if(freq[i] > 0){
                min = Math.min(min, freq[i]);
            }
        }
        return min == Integer.MAX_VALUE? -1 : min -1;
    }
}
```

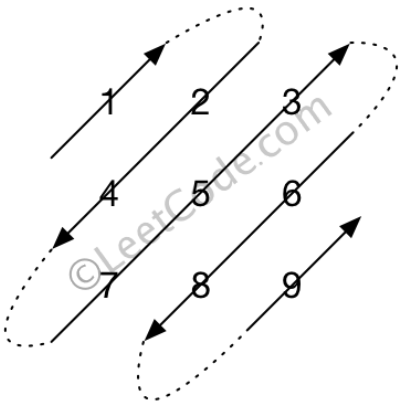
---

498. Diagonal Traverse

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]

Output:  [1,2,4,7,5,3,6,8,9]

Explanation:
```



Note:

The total number of elements of the given matrix will not exceed 10,000.

/Question:

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image. Example:

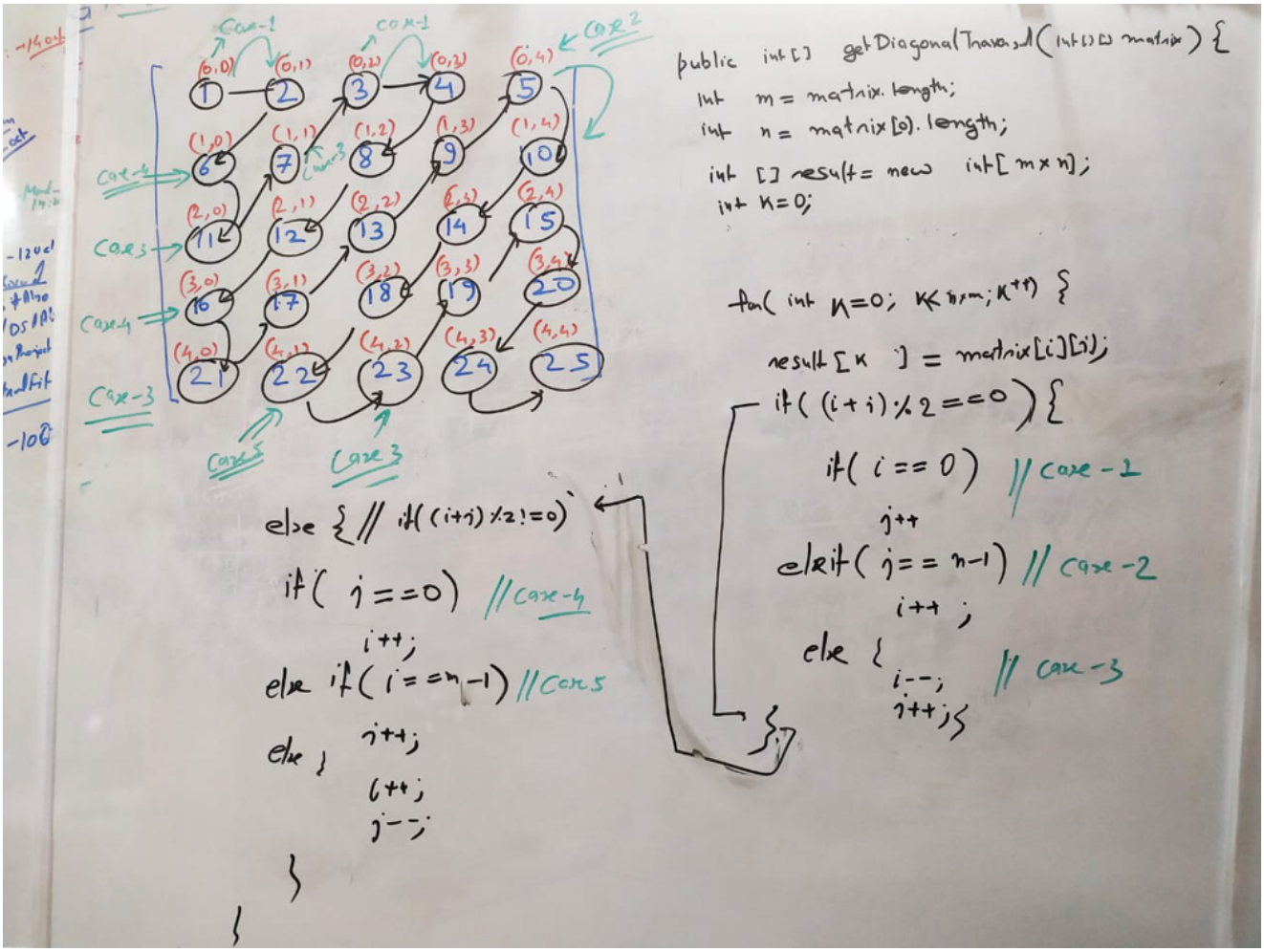
Input: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Output: [1,2,4,7,5,3,6,8,9]

Explanation: Note:The total number of elements of the given matrix will not exceed 10,000.

Solution: There are six test cases which are explained the image, please check the following image. There are `if (row + col) %2 ==0` **Case 1 :** Top Left to Right Traverse , just increase the column. **Case 2 :** Right most Top to Bottom Traverse, just increase the row. **Case 3:** Digonal Traveral Bottom to Top, just decrease row and increase column.

`if (row + col) %2 !=0` **Case 4 :** Left Most Top to bottom Traverse , just increase the row. **Case 5 :** Bottom most left to Right Traverse, just column the row. **Case 6:** Digonal Traveral top to Bottom, just increase row and decrease column In following image, cases are shown in green color, and (row, col) are shown in red color. I forget to mention case 6 in image but it is obvious.



```
class Solution {
    public int[] findDiagonalOrder(int[][] matrix) {
        if(matrix.length == 0)
            return new int[0];
        int m = matrix.length;
        int n = matrix[0].length;
        int [] result = new int[m * n];
        int i=0, j =0;
        for(int k =0; k< m*n; k++){
            result[k] = matrix[i][j];
            if( (i + j) %2 == 0){
                if(j == n -1){  // this condition should come before i==0,
                    i++;
                }else if(i ==0){
                    j++;
                }else {
                    i--;
                    j++;
                }
            }else{
                if(i == m-1){
                    j++;
                }else if(j ==0){
                    i++;
                }else{
                    i++;
                    j--;
                }
            }
        }
        return result;
    }
}
```

518. Coin Change 2

You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Example 1:

```
Input: amount = 5, coins = [1, 2, 5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

Example 2:

```
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.
```

Example 3:

```
Input: amount = 10, coins = [10]
Output: 1
```

Note:

You can assume that

- 0 <= amount <= 5000
- 1 <= coin <= 5000
- the number of coins is less than 500
- the answer is guaranteed to fit into signed 32-bit integer

You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Example 1:

Input: amount = 5, coins = [1, 2, 5] Output: 4 Explanation: there are four ways to make up the amount: 5=5 5=2+2+1 5=2+1+1+1 5=1+1+1+1+1 Example 2:

Input: amount = 3, coins = [2] Output: 0 Explanation: the amount of 3 cannot be made up just with coins of 2. Example 3:

Input: amount = 10, coins = [10] Output: 1

Note:

You can assume that

0 <= amount <= 5000 1 <= coin <= 5000 the number of coins is less than 500 the answer is guaranteed to fit into signed 32-bit integer

```
class Solution {
    public int change(int amount, int[] coins) {
        int [] totalCombination = new int[amount + 1];
        totalCombination[0] = 1;
        int noOfCoins = coins.length;
        for(int i=0; i<noOfCoins; i++){
            for(int j =1; j<= amount; j++){
                if(j >=coins[i]){
                    totalCombination[j] += totalCombination[j - coins[i]];
                }
            }
        }
        return totalCombination[amount];
    }
}
```

525. Contiguous Array

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example 1:

```
Input: [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1
```

Example 2:

```
Input: [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number
```

Note: The length of the given binary array will not exceed 50,000.

---

525. Contiguous Array Medium

979

49

Add to List

Share Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example 1: Input: [0,1] Output: 2 Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1. Example 2: Input: [0,1,0] Output: 2 Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1. Note: The length of the given binary array will not exceed 50,000.

```
class Solution {
    public int findMaxLength(int[] nums) {
        HashMap<Integer, Integer> counts  = new HashMap<>();
        int maxLength = 0;
        int count = 0;
        counts.put(0, -1);
        for(int i=0; i< nums.length; i++){
            if(nums[i] == 0){
                count--;
            }else{
                count++;
            }
            if(counts.containsKey(count)){
                maxLength = Math.max(maxLength, i - counts.get(count));
            }else{
                counts.put(count, i);
            }
        }
        return maxLength;
    }
}
```

Other Solution

Algorithm Create "deltas" array. Each value in this array will represent (number of 1s) minus (number of 0s) seen so far. Create a HashMap<Integer, Integer>, that for each delta, it saves the first index we saw this delta in our array "key" is the "delta" calculated earlier "value" is first index of this delta in original array Example index: [0 1 2 3 4 5 6 7] input: [0 0 1 0 0 0 1 1] deltas: [-1 -2 -1 -2 -3 -4 -3 -2] In our deltas array, if we encounter the same value twice, it means the number of 0s and the number of 1s are equal between these indices. The largest subarray corresponds to the largest j-i where deltas[i] == deltas[j]. In this case, i=1, j=7, deltas[i] == deltas[j] == -2, and j-i = 7-1 = 6. This corresponds to subarray (i, j] which is [1, 0, 0, 0, 1, 1] Solution class Solution { public int findMaxLength(int[] array) { if (array == null) { return 0; } int[] deltas = findDeltas(array); Map<Integer, Integer> map = new HashMap<>(); map.put(0, -1); // for testcases such as [1, 0], [1, 0, 0] int maxLength = 0; for (int i = 0; i < array.length; i++) { if (!map.containsKey(deltas[i])) { map.put(deltas[i], i); } else { maxLength = Math.max(maxLength, i - map.get(deltas[i])); } } return maxLength; }

```
private int[] findDeltas(int[] array) {
    int[] deltas = new int[array.length];
    int delta = 0;
    for (int i = 0; i < array.length; i++) {
        if (array[i] == 1) {
            delta++;
        } else if (array[i] == 0) {
            delta--;
        }
        // if array[i] is not a 1 or 0, delta does not change

        deltas[i] = delta;
    }
    return deltas;
}
```

} Time/Space Complexity Time Complexity: O(n) Space Complexity: O(n)

1150. Check If a Number Is Majority Element in a Sorted Array

Given an array `nums` sorted in **non-decreasing** order, and a number `target`, return `True` if and only if `target` is a majority element.

A *majority element* is an element that appears **more than `N/2`** times in an array of length `N`.

Example 1:

```
Input: nums = [2,4,5,5,5,5,5,6,6], target = 5
Output: true
Explanation:
The value 5 appears 5 times and the length of the array is 9.
Thus, 5 is a majority element because 5 > 9/2 is true.
```

Example 2:

```
Input: nums = [10,100,101,101], target = 101
Output: false
Explanation:
The value 101 appears 2 times and the length of the array is 4.
Thus, 101 is not a majority element because 2 > 4/2 is false.
```

Note:

1. `1 <= nums.length <= 1000`
2. `1 <= nums[i] <= 10^9`
3. `1 <= target <= 10^9`

1150. Check If a Number Is Majority Element in a Sorted Array Easy

67

14

Add to List

Share Given an array nums sorted in non-decreasing order, and a number target, return True if and only if target is a majority element.

A majority element is an element that appears more than N/2 times in an array of length N.

Example 1:

Input: nums = [2,4,5,5,5,5,5,6,6], target = 5 Output: true Explanation: The value 5 appears 5 times and the length of the array is 9. Thus, 5 is a majority element because 5 > 9/2 is true. Example 2:

Input: nums = [10,100,101,101], target = 101 Output: false Explanation: The value 101 appears 2 times and the length of the array is 4. Thus, 101 is not a majority element because 2 > 4/2 is false.

```
class Solution {
    public boolean isMajorityElement(int[] nums, int target) {
        int size = nums.length;
        int start  = 0;
        int end = size -1;
        int mid = start + (end - start) / 2;
        int index = findIndexOfFirstOccurance(nums, target);
        if(index + size/2 >= size || index == -1)
            return false;
        return (nums[index + size/2] == target) ? true : false;

    }
    private int findIndexOfFirstOccurance(int [] nums, int target){
        int start = 0;
        int end = nums.length -1;
        while(start <= end){
            int mid = start + (end - start) /2;
            if((mid == 0 || nums[mid -1] < target) && nums[mid] == target)
                return mid;
            if(nums[mid] < target)
                start = mid + 1;
            else
                end = mid -1;

        }
        return -1;
    }
}
```

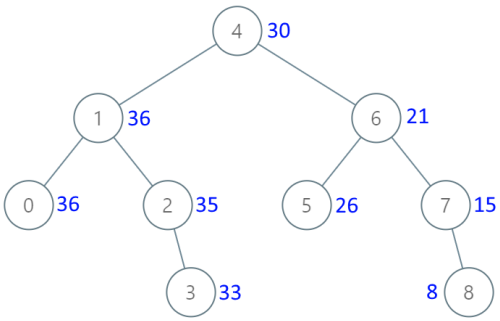
1038. Binary Search Tree to Greater Sum Tree

Given the root of a binary **search** tree with distinct values, modify it so that every `node` has a new value equal to the sum of the values of the original tree that are greater than or equal to `node.val`.

As a reminder, a *binary search tree* is a tree that satisfies these constraints:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



```
Input: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

Note:

1. The number of nodes in the tree is between `1` and `100`.
2. Each node will have value between `0` and `100`.
3. The given tree is a binary search tree.

1038. Binary Search Tree to Greater Sum Tree Medium

454

71

Add to List

Share Given the root of a binary search tree with distinct values, modify it so that every node has a new value equal to the sum of the values of the original tree that are greater than or equal to node.val.

As a reminder, a binary search tree is a tree that satisfies these constraints:

The left subtree of a node contains only nodes with keys less than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key. Both the left and right subtrees must also be binary search trees.

** Recursive Apporach** `class Solution { int sum =0; public TreeNode bstToGst(TreeNode root) { if(root == null) return root; bstToGst(root.right); sum += root.val; root.val = sum; bstToGst(root.left); return root; } }`

Iterative Approach

```
public TreeNode bstToGst(TreeNode root){
        if(root == null)
            return root;
        List<TreeNode> list = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.add(cur);
                cur = cur.right;
            }
            cur = stack.pop();
            list.add(cur);
            cur = cur.left;
        }
        int sum =0;
        for(TreeNode node: list){
            sum += node.val;
            node.val = sum;
        }
        return root;
    }
```

1167. Minimum Cost to Connect Sticks

You have some `sticks` with positive integer lengths.

You can connect any two sticks of lengths `X` and `Y` into one stick by paying a cost of `X + Y`. You perform this action until there is one stick remaining.

Return the minimum cost of connecting all the given `sticks` into one stick in this way.

Example 1:

```
Input: sticks = [2,4,3]
Output: 14
```

Example 2:

```
Input: sticks = [1,8,3,5]
Output: 30
```

Constraints:

- `1 <= sticks.length <= 10^4`
- `1 <= sticks[i] <= 10^4`

1167. Minimum Cost to Connect Sticks

You have some sticks with positive integer lengths.

You can connect any two sticks of lengths X and Y into one stick by paying a cost of X + Y. You perform this action until there is one stick remaining.

Return the minimum cost of connecting all the given sticks into one stick in this way.

Example 1:

Input: sticks = [2,4,3] Output: 14 Example 2:

Input: sticks = [1,8,3,5] Output: 30 `class Solution { public int connectSticks(int[] sticks) { PriorityQueue<Integer> queue = new PriorityQueue<>(); int result = 0; for(int value: sticks){ queue.add(value); } while(queue.size() > 1){ int cost = queue.poll() + queue.poll(); result += cost; queue.add(cost); } return result; } }`

1116. Print Zero Even Odd

Suppose you are given the following code:

```
class ZeroEvenOdd {
  public ZeroEvenOdd(int n) { ... }      // constructor
  public void zero(printNumber) { ... }  // only output 0's
  public void even(printNumber) { ... }  // only output even numbers
  public void odd(printNumber) { ... }   // only output odd numbers
}
```

The same instance of `ZeroEvenOdd` will be passed to three different threads:

1. Thread A will call `zero()` which should only output 0's.
2. Thread B will call `even()` which should only ouput even numbers.
3. Thread C will call `odd()` which should only output odd numbers.

Each of the threads is given a `printNumber` method to output an integer. Modify the given program to output the series `010203040506` ... where the length of the series must be 2*n*.

Example 1:

```
Input: n = 2
Output: "0102"
Explanation: There are three threads being fired asynchronously. One of them calls
```

Example 2:

```
Input: n = 5
Output: "0102030405"
```

1116. Print Zero Even Odd Medium

118

74

Add to List

Share Suppose you are given the following code:

class ZeroEvenOdd { public ZeroEvenOdd(int n) { ... } // constructor public void zero(printNumber) { ... } // only output 0's public void even(printNumber) { ... } // only output even numbers public void odd(printNumber) { ... } // only output odd numbers } The same instance of ZeroEvenOdd will be passed to three different threads:

Thread A will call zero() which should only output 0's. Thread B will call even() which should only ouput even numbers. Thread C will call odd() which should only output odd numbers. Each of the threads is given a printNumber method to output an integer. Modify the given program to output the series 010203040506... where the length of the series must be 2n.

Example 1:

Input: n = 2 Output: "0102" Explanation: There are three threads being fired asynchronously. One of them calls zero(), the other calls even(), and the last one calls odd(). "0102" is the correct output. Example 2:

Input: n = 5 Output: "0102030405"

```
class ZeroEvenOdd {
    private int n;
    Semaphore zero = new Semaphore(1);
    Semaphore one = new Semaphore(0);
    Semaphore two = new Semaphore(0);
    public ZeroEvenOdd(int n) {
        this.n = n;
    }

    // printNumber.accept(x) outputs "x", where x is an integer.
    public void zero(IntConsumer printNumber) throws InterruptedException {
        for(int i=0; i<n; i++){
            zero.acquire();
            printNumber.accept(0);
            if(i %2 == 0){
                one.release();
            }else{
                two.release();
            }
        }
    }

    public void even(IntConsumer printNumber) throws InterruptedException {
        for(int i=2; i<=n; i=i+2){
            two.acquire();
            printNumber.accept(i);
            zero.release();
        }
    }

    public void odd(IntConsumer printNumber) throws InterruptedException {
        for(int i=1; i<=n; i=i+2){
            one.acquire();
            printNumber.accept(i);
            zero.release();
        }
    }
}
```