



LeetCode

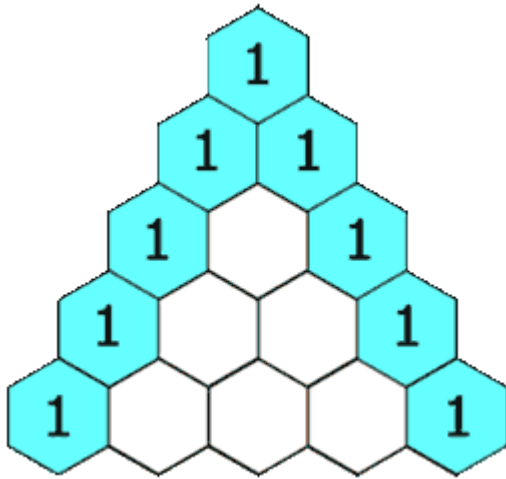
Practice Set I

More Contents on [GitHub.com/AyeRaj](https://github.com/AyeRaj)

118. Pascal's Triangle

Given an integer `numRows`, return the first numRows of **Pascal's triangle**. [View Question](#)

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



Example 1:

Input: numRows = 5

Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

```
class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> result = new ArrayList<>();


        List<Integer> list, pre=null;
        for(int i=0; i<numRows; i++){
            list = new ArrayList<>();
            for(int j=0; j <=i; j++){
                if(j==0 || j==i){
                    list.add(1);
                }
                else{
                    list.add(pre.get(j-1)+pre.get(j));
                }
            }
            pre = list;
            result.add(list);
        }
        return result;
    }
}
```

73. Set Matrix Zeroes

Given an $m \times n$ integer matrix `matrix`, if an element is 0, set its entire row and column to 0's.

Example 1:

1	1	1
1	0	1
1	1	1




1	0	1
0	0	0
1	0	1

Input: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`

Output: `[[1,0,1],[0,0,0],[1,0,1]]`

Example 2:

0	1	2	0
3	4	5	2
1	3	1	5



0	0	0	0
0	4	5	0
0	3	1	0

Input: `matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]`

Output: `[[0,0,0,0],[0,4,5,0],[0,3,1,0]]`

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;
        int[][] arr = new int[m][n];
        for(int i=0; i<m; i++){
            for(int j = 0; j<n;j++){
                arr[i][j]=matrix[i][j];
            }
        }
        for(int i=0; i<m; i++){
            for(int j = 0; j<n;j++){
                if(arr[i][j]==0)
                    helper(i, j, matrix);
            }
        }
        public void helper(int a, int b, int[][] matrix){
            for(int k=0; k<matrix[0].length; k++){
                matrix[a][k]=0;
            }
            for(int l=0; l<matrix.length; l++){
                matrix[l][b]=0;
            }
        }
    }
}
```

31. Next Permutation

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for `arr = [1, 2, 3]`, the following are considered permutations of `arr`: `[1, 2, 3]`, `[1, 3, 2]`, `[3, 1, 2]`, `[2, 3, 1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1, 2, 3]` is `[1, 3, 2]`.
- Similarly, the next permutation of `arr = [2, 3, 1]` is `[3, 1, 2]`.
- While the next permutation of `arr = [3, 2, 1]` is `[1, 2, 3]` because `[3, 2, 1]` does not have a lexicographical larger rearrangement.

Given an array of integers `nums`, find the next permutation of `nums`.

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

Example 1:

Input: `nums = [1, 2, 3]`

Output: `[1, 3, 2]`

Example 2:

Input: `nums = [3, 2, 1]`

Output: `[1, 2, 3]`

Example 3:

Input: `nums = [1, 1, 5]`

Output: `[1, 5, 1]`

```
class Solution {
    public void nextPermutation(int[] nums) {
        if(nums == null || nums.length <= 1)
            return;
        int i = nums.length - 2;
        while(i >= 0 && nums[i] >= nums[i+1])
            i--;
        if(i >= 0){
            int j = nums.length - 1;
            while(nums[j] <= nums[i])
                j--;
            swap(nums, i, j);
        }
    }
}
```

```

        reverse(nums, i+1, nums.length -1);
    }
    public void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
    public void reverse(int[] nums, int i, int j){
        while(i<j){
            swap(nums, i++, j--);
        }
    }
}

```

1. Find the largest index k such that $nums[k] < nums[k + 1]$.
If no such index exists, just reverse $nums$ and done.
2. Find the largest index $l > k$ such that $nums[k] < nums[l]$.
3. Swap $nums[k]$ and $nums[l]$.
4. Reverse the sub-array $nums[k + 1:]$.

Input: [1, 2, 3]

{
 [1, 2, 3]
 [1, 3, 2]
 [2, 1, 3]
 [2, 3, 1]
 [3, 1, 2]
 [3, 2, 1]
 ↓

0. Initial sequence

0	1	2	5	3	3	0
---	---	---	---	---	---	---

1. Find longest non-increasing suffix

0	1	2	5	3	3	0
---	---	---	---	---	---	---

2. Identify pivot

0	1	2	5	3	3	0
---	---	---	---	---	---	---

3. Find rightmost successor to pivot in the suffix

0	1	2	5	3	3	0
---	---	---	---	---	---	---

4. Swap with pivot

0	1	3	5	3	2	0
---	---	---	---	---	---	---

5. Reverse the suffix

0	1	3	0	2	3	5
---	---	---	---	---	---	---

6. Done

0	1	3	0	2	3	5
---	---	---	---	---	---	---

53. Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

Example 1:

Input: `nums = [-2,1,-3,4,-1,2,1,-5,4]`

Output: 6

Explanation: `[4,-1,2,1]` has the largest sum = 6.

Example 2:

Input: `nums = [1]`

Output: 1

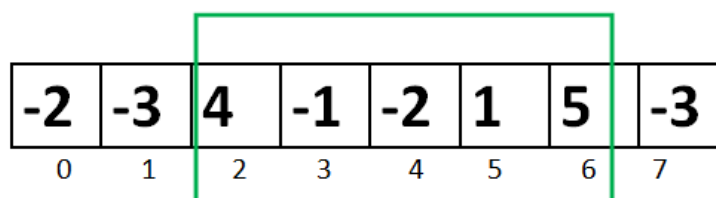
Example 3:

Input: `nums = [5,4,-1,7,8]`

Output: 23

```
class Solution {
    public int maxSubArray(int[] nums) {
        int max=Integer.MIN_VALUE;
        int sum = 0;
        for(int i=0; i<nums.length; i++){
            sum=Math.max(nums[i], nums[i]+sum);
            max=Math.max(sum,max);
        }
        return max;
    }
}
```

Largest Subarray Sum Problem



$$4 + (-1) + (-2) + 1 + 5 = 7$$

Maximum Contiguous Array Sum is 7

75. 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, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

Input: `nums = [2,0,2,1,1,0]`

Output: `[0,0,1,1,2,2]`

Example 2:

Input: `nums = [2,0,1]`

Output: `[0,1,2]`

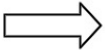
```
class Solution {
    public void sortColors(int[] nums) {
        int n = 0;
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == 0){
                swap(nums, i, n);
                n++;
            }
        }
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == 1){
                swap(nums, i, n);
                n++;
            }
        }
    }
    public void swap(int[] nums, int i, int n){
        int temp = nums[i];
        nums[i] = nums[n];
        nums[n] = temp;
    }
}
```


48. Rotate Image

You are given an $n \times n$ 2D `matrix` representing an image, rotate the image by **90** degrees (clockwise). You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:

1	2	3
4	5	6
7	8	9




7	4	1
8	5	2
9	6	3

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

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

Example 2:

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16



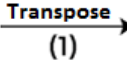
15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

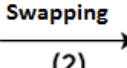
```
class Solution {
    public void rotate(int[][] matrix) {
        for(int i=0; i<matrix.length; i++){ // Transpose the matrix
            for(int j=i; j<matrix[0].length; j++){
                int temp=0;
                temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i]=temp;
            }
        } // To rotate clockwise - replace half matrix vertically
        int n = matrix.length,m = matrix[0].length;
        for(int i=0; i<n; i++){
            for(int j=0; j<m/2; j++){
                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][m-j-1];
                matrix[i][m-j-1] = temp;
            }
        }
    }
}
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16



13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

56. Merge Intervals

Given an array of `intervals` where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example 1:

Input: `intervals = [[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: `intervals = [[1,4],[4,5]]`

Output: `[[1,5]]`

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

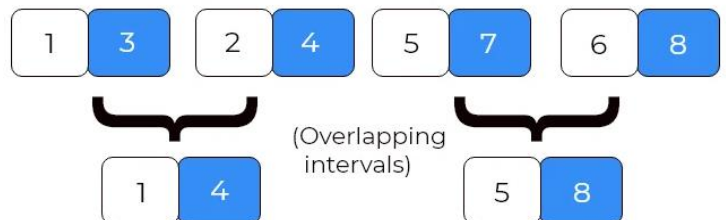
```
class Solution {
    public int[][] merge(int[][] intervals) {
        if(intervals.length<=1)
            return intervals;

        Arrays.sort(intervals, (a,b)-> a[0] - b[0]);

        List<int[]> result = new ArrayList<>();

        int start = intervals[0][0];
        int end = intervals[0][1];

        for(int[] interv : intervals){
            if(interv[0] <= end)
                end = Math.max(end, interv[1]);
            else{
                result.add(new int[]{start, end});
                start = interv[0];
                end = interv[1];
            }
        }
        result.add(new int[]{start, end});
        return result.toArray(new int[0][]);
    }
}
```



An efficient approach is to first sort the intervals according to the starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if `interval[i]` doesn't overlap with `interval[i-1]`, then `interval[i+1]` cannot overlap with `interval[i-1]` because starting time of `interval[i+1]` must be greater than or equal to `interval[i]`. Following is the detailed step by step algorithm.

88. Merge Sorted Array

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to 0 and should be ignored. `nums2` has a length of `n`.

Example 1:

Input: `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`

Output: `[1,2,2,3,5,6]`

Explanation: The arrays we are merging are `[1,2,3]` and `[2,5,6]`.

The result of the merge is `[1,2,2,3,5,6]` with the underlined elements coming from `nums1`.

Example 2:

Input: `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`

Output: `[1]`

Explanation: The arrays we are merging are `[1]` and `[]`.

The result of the merge is `[1]`.

Example 3:

Input: `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`

Output: `[1]`

Explanation: The arrays we are merging are `[]` and `[1]`.

The result of the merge is `[1]`.

Note that because `m = 0`, there are no elements in `nums1`. The 0 is only there to ensure the merge result can fit in `nums1`.

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int p = m-1, q = n-1, r = m+n-1;
        while(q >= 0){
            if(p >= 0 && nums1[p]>nums2[q])
                nums1[r--] = nums1[p--];
            else
                nums1[r--] = nums2[q--];
        }
    }
}
```

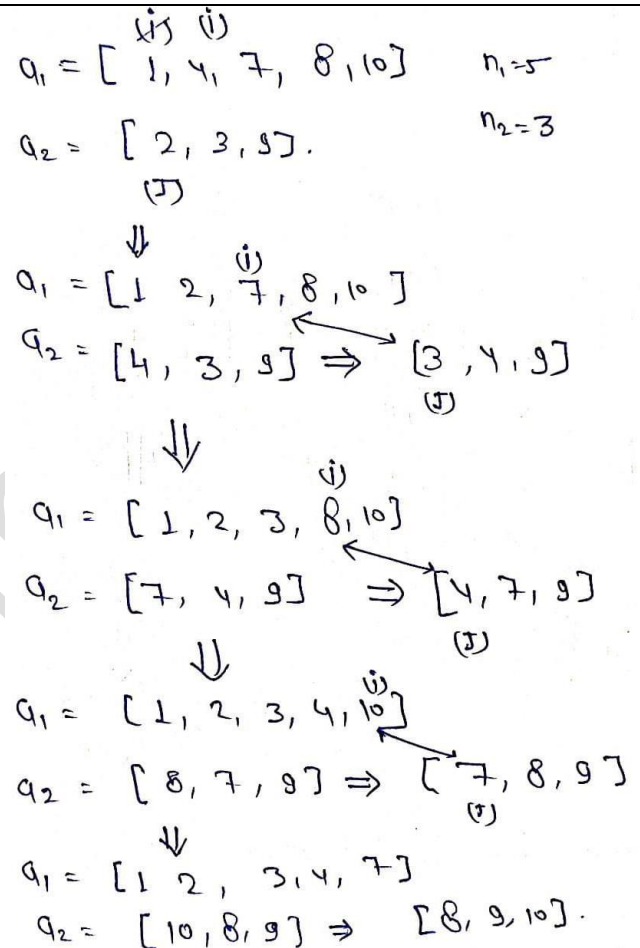
Solution 2: Without using space

Intuition: We can think of iterating in arr1 and whenever we encounter an element that is greater than the first element of arr2, just swap it. Now rearrange the arr2 in a sorted manner, after completion of the loop we will get elements of both the arrays in non-decreasing order.

Approach:

- Use a for loop in arr1.
- Whenever we get any element in arr1 which is greater than the first element of arr2, swap it.
- Rearrange arr2.
- Repeat the process.

```
public class tuf {  
  
    public static void main(String[] args) {  
        int arr1[] = {1,4,7,8,10};  
        int arr2[] = {2,3,9};  
        System.out.println("Before merge:");  
        for (int i = 0; i < arr1.length; i++) {  
            System.out.print(arr1[i] + " ");  
        }  
        System.out.println();  
        for (int i = 0; i < arr2.length; i++) {  
            System.out.print(arr2[i] + " ");  
        }  
        System.out.println();  
        merge(arr1, arr2, arr1.length, arr2.length);  
        System.out.println("After merge:");  
        for (int i = 0; i < arr1.length; i++) {  
            System.out.print(arr1[i] + " ");  
        }  
        System.out.println();  
        for (int i = 0; i < arr2.length; i++) {  
            System.out.print(arr2[i] + " ");  
        }  
    }  
  
    static void merge(int[] arr1, int arr2[], int n, int m) {  
        // code here  
        int i, k;  
        for (i = 0; i < n; i++) {  
            // take first element from arr1  
            // compare it with first element of second array  
            // if condition match, then swap  
            if (arr1[i] > arr2[0]) {  
                int temp = arr1[i];  
                arr1[i] = arr2[0];  
                arr2[0] = temp;  
            }  
  
            // then sort the second array  
            // put the element in its correct position
```



```

// so that next cycle can swap elements correctly
int first = arr2[0];
// insertion sort is used here
for (k = 1; k < m && arr2[k] < first; k++) {
    arr2[k - 1] = arr2[k];
}
arr2[k - 1] = first;
}
}
}

```

Solution 3: Gap method

Approach:

- Initially take the gap as $(m+n)/2$;
- Take as a pointer1 = 0 and pointer2 = gap.
- Run a loop from pointer1 & pointer2 to $m+n$ and whenever $arr[pointer2] < arr[pointer1]$, just swap those.
- After completion of the loop reduce the gap as $gap = gap/2$.
- Repeat the process until $gap > 0$.

```

class Main{
    static void swap(int a,int b){
        int temp=a;
        a=b,b=temp;
    }
    static void merge(int ar1[], int ar2[], int n, int m) {
int gap =(int) Math.ceil((double)(n + m) / 2.0);
while (gap > 0) {
    int i = 0,j = gap;
    while (j < (n + m)) {
        if (j < n && ar1[i] > ar1[j]) {
            swap(ar1[i], ar1[j]);
        } else if (j >= n && i < n && ar1[i] > ar2[j - n]) {
            swap(ar1[i], ar2[j - n]);
        } else if (j >= n && i >= n && ar2[i - n] > ar2[j - n]) {
            swap(ar2[i - n], ar2[j - n]);
        }
        j++;
        i++;
    }
    if (gap == 1)
        gap = 0;
    else
        gap =(int) Math.ceil((double) gap / 2.0);
}
}

public static void main(String[] args) {
    int arr1[] = {1,4,7,8,10};
    int arr2[] = {2,3,9};
    System.out.println("Before merge:");
    for (int i = 0; i < arr1.length; i++) {
        System.out.print(arr1[i] + " ");
    }
    System.out.println();
    for (int i = 0; i < arr2.length; i++) {
        System.out.print(arr2[i] + " ");
    }
    System.out.println();
    merge(arr1, arr2, arr1.length, arr2.length);
    System.out.println("After merge:");
    for (int i = 0; i < arr1.length; i++) {
        System.out.print(arr1[i] + " ");
    }
    System.out.println();
    for (int i = 0; i < arr2.length; i++) {
        System.out.print(arr2[i] + " ");
    }
}
}

```

287. Find the Duplicate Number

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

Input: `nums = [1,3,4,2,2]`

Output: 2

Example 2:

Input: `nums = [3,1,3,4,2]`

Output: 3

Approach: 1

```
class Solution {
    public int findDuplicate(int[] nums) {
        int len = nums.length;
        int[] temp = new int[len+1];

        for(int i=0; i<len; i++){
            temp[nums[i]]++;
            if(temp[nums[i]]>1){
                return nums[i];
            }
        }
        return len;
    }
}
```

Approach: 2

```
public static int findDuplicate_set(int[] nums) {
    Set<Integer> set = new HashSet<>();
    int len = nums.length;
    for (int i = 0; i < len; i++) {
        if (!set.add(nums[i])) {
            return nums[i];
        }
    }

    return len;
}
```

268. Missing Number

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return *the only number in the range that is missing from the array*.

Example 1:

Input: `nums = [3,0,1]`

Output: 2

Explanation: `n = 3` since there are 3 numbers, so all numbers are in the range `[0,3]`. 2 is the missing number in the range since it does not appear in `nums`.

Example 2:

Input: `nums = [0,1]`

Output: 2

Explanation: `n = 2` since there are 2 numbers, so all numbers are in the range `[0,2]`. 2 is the missing number in the range since it does not appear in `nums`.

Example 3:

Input: `nums = [9,6,4,2,3,5,7,0,1]`

Output: 8

Explanation: `n = 9` since there are 9 numbers, so all numbers are in the range `[0,9]`. 8 is the missing number in the range since it does not appear in `nums`.

```
class Solution {
    public int missingNumber(int[] nums) {
        int len = nums.length;
        int[] temp = new int[len+1];
        for(int i=0; i<len; i++){
            temp[nums[i]]++;
        }
        for(int i=0; i<len; i++){
            if(temp[i]==0)
                return i;
        }
        return len;
    }
}
```

```
class Solution {
    public int missingNumber(int[] nums) {
        int sum=0;
        for(int val:nums){
            sum+=val;
        }
        return (nums.length*(nums.length+1)/2 - sum);
    }
}
```



```
}  
}
```

```
class Solution {  
    public int missingNumber(int[] nums) {  
        int xor = 0, i = 0;  
        for (i = 0; i < nums.length; i++) {  
            xor = xor ^ i ^ nums[i];  
        }  
        return xor ^ i;  
    }  
}
```

Repeat and Missing Number Array

You are given a read only array of n integers from 1 to n.

Each integer appears exactly once except A which appears twice and B which is missing.

Return A and B.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Note that in your output A should precede B.

Example:

Input:[3 1 2 5 3]

Output:[3, 4]

A = 3, B = 4

```
public class Solution {
    public ArrayList<Integer> repeatedNumber(final List<Integer> A) {

        ArrayList<Integer> res = new ArrayList<>();

        Collections.sort(A);
        int n = A.size();
        int rep = -1;
        int miss = -1;
        long sum = A.get(0);

        for (int i = 1; i < n; i++) {
            if (A.get(i).intValue() == A.get(i - 1).intValue()) {
                rep = A.get(i);
            }
            sum += A.get(i);
        }

        miss = (int) ((1L * n * (1L * n + 1)) / 2 - sum + rep);

        res.add(rep);
        res.add(miss);

        return res;
    }
}
```

```
public class Solution {
    public ArrayList<Integer> repeatedNumber(final List<Integer> A) {

        ArrayList<Integer> out = new ArrayList<Integer>();
```

```

double l = A.size();
double sum = (l*(l+1))/2;
long sumA = 0;
int a=0;
HashSet<Integer> ASet = new HashSet<Integer>();
for(int i=0;i<A.size();i++) {
    if(ASet.contains(A.get(i))) {
        a=A.get(i);
    }
    ASet.add(A.get(i));
    sumA = sumA+A.get(i);
}
double diff = sumA - sum;
int b = a - (int)diff;
out.add(a);
out.add(b);
return out;
}
}

```

```

public class Solution {
    public ArrayList<Integer> repeatedNumber(final List<Integer> A) {

        ArrayList<Integer> out = new ArrayList<Integer>();
        double l = A.size();
        double sum = (l*(l+1))/2;
        long sumA = 0;
        int a=0;
        HashSet<Integer> ASet = new HashSet<Integer>();
        for(int i=0;i<A.size();i++) {
            if(ASet.contains(A.get(i))) {
                a=A.get(i);
            }
            ASet.add(A.get(i));
            sumA = sumA+A.get(i);
        }
        double diff = sumA - sum;
        int b = a - (int)diff;
        out.add(a);
        out.add(b);
        return out;
    }
}

```

74. Search a 2D Matrix

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`

Output: `true`

Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 13`

Output: `false`

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int n = matrix.length, m = matrix[0].length;
        for(int i=0; i<n; i++){
            if(matrix[i][m-1]>=target){
                for(int j = 0; j<m; j++){
                    if(matrix[i][j]==target)
                        return true;
                }
            }
        }
        return false;
    }
}
```

```
class Solution {  
    public boolean searchMatrix(int[][] matrix, int target) {  
        int i = 0, j = matrix[0].length - 1;  
        while(i < matrix.length && j >= 0) {  
            if(matrix[i][j] == target)  
                return true;  
            else if(matrix[i][j] > target)  
                j --;  
            else if(matrix[i][j] < target)  
                i ++;  
        }  
        return false;  
    }  
}
```

169. Majority Element

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.

Example 1:

Input: `nums = [3,2,3]`

Output: `3`

Example 2:

Input: `nums = [2,2,1,1,1,2,2]`

Output: `2`

```
class Solution {  
    public int majorityElement(int[] nums) {  
        Arrays.sort(nums);  
        return nums[nums.length/2];  
    }  
}
```

```
class Solution {  
  
    public int majorityElement(int[] nums) {  
  
        int count=0, ret = 0;  
  
        for (int num: nums) {  
  
            if (count==0)  
                ret = num;  
  
            if (num!=ret)  
                count--;  
  
            else  
                count++;  
  
        }  
  
        return ret;  
  
    }  
}
```

229. Majority Element II

Given an integer array of size n , find all elements that appear more than $\lfloor n/3 \rfloor$ times.

Example 1:

Input: `nums = [3,2,3]`

Output: `[3]`

Example 2:

Input: `nums = [1]`

Output: `[1]`

```
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        int num1 = -1, num2 = -1, count1 = 0, count2 = 0, len = nums.length;
        for(int i=0; i<len; i++){
            if(nums[i]==num1)
                count1++;
            else if(nums[i]==num2)
                count2++;
            else if(count1==0){
                num1=nums[i];
                count1=1;
            }else if(count2==0){
                num2=nums[i];
                count2=1;
            }else{
                count1--;
                count2--;
            }
        }
        List<Integer> list = new ArrayList<Integer>();
        count1=0;
        count2=0;
        for(int i=0; i<len; i++){
            if(nums[i]==num1)
                count1++;
            else if(nums[i]==num2)
                count2++;
        }
        if(count1>len/3)
            list.add(num1);
        if(count2>len/3)
            list.add(num2);
        return list;
    }
}
```

62. Unique Paths

There is a robot on an $m \times n$ grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers m and n , return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

Example 1:



Input: $m = 3, n = 7$

Output: 28

Example 2:

Input: $m = 3, n = 2$

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

```
class Solution {
    public int uniquePaths(int m, int n) {
        int N = m + n - 2;
        int r = m - 1;
        double res = 1;
        for (int i = 1; i <= r; i++) {
            res = res * (N - r + i) / i;
        }
        return (int) res;
    }
}
```



```

class Solution {
    public int uniquePaths(int m, int n) {
        if(m == 1 || n == 1)
            return 1;

        m--;
        n--;

        if(m < n) {    // Swap, so that m is the bigger number
            m = m + n;
            n = m - n;
            m = m - n;
        }

        long res = 1;
        int j = 1;

        for(int i = m+1; i <= m+n; i++, j++){ // Instead of taking factorial, keep on
multiply & divide
            res *= i;
            res /= j;
        }

        return (int)res;
    }
}

```

121. Best Time to Buy and Sell Stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: `prices = [7,1,5,3,6,4]`

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

```
class Solution {
    public int maxProfit(int[] prices) {
        int currmax = 0;
        int max = 0;
        for(int i=1; i<prices.length; i++){
            currmax=Math.max(0, currmax+=prices[i]-prices[i-1]);
            max=Math.max(currmax,max);
        }
        return max;
    }
}
```

122. Best Time to Buy and Sell Stock II

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

Example 1:

Input: `prices = [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 = $6 - 3 = 3$.

Total profit is $4 + 3 = 7$.

Example 2:

Input: `prices = [1,2,3,4,5]`

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = $5 - 1 = 4$.

Total profit is 4.

Example 3:

Input: `prices = [7,6,4,3,1]`

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices==null || prices.length<=1)
            return 0;
        int profit=0;
        for(int i=1; i<prices.length;i++){
            if(prices[i-1]<prices[i])
                profit+=prices[i]-prices[i-1];
        }
        return profit;
    }
}
```

123. Best Time to Buy and Sell Stock III

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

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

Example 1:

Input: `prices = [3,3,5,0,0,3,1,4]`

Output: 6

Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = $3 - 0 = 3$.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = $4 - 1 = 3$.

Example 2:

Input: `prices = [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: `prices = [7,6,4,3,1]`

Output: 0

Explanation: In this case, no transaction is done, i.e. max profit = 0.

```
class Solution {
    public int maxProfit(int[] prices) {
        int sell1 = 0, sell2 = 0, buy1 = Integer.MIN_VALUE, buy2 = Integer.MIN_VALUE;
        for (int i = 0; i < prices.length; i++) {
            buy1 = Math.max(buy1, -prices[i]);
            sell1 = Math.max(sell1, buy1 + prices[i]);
            buy2 = Math.max(buy2, sell1 - prices[i]);
            sell2 = Math.max(sell2, buy2 + prices[i]);
        }
        return sell2;
    }
}
```

```
//2nd Approach

class Solution {

    public int maxProfit(int[] prices) {

        int hold1 = Integer.MIN_VALUE, hold2 = Integer.MIN_VALUE;

        int release1 = 0, release2 = 0;

        for(int i:prices){    // Assume we only have 0 money at first

            release2 = Math.max(release2, hold2+i);    // The maximum if we've just sold
2nd stock so far.

            hold2 = Math.max(hold2,release1-i);    // The maximum if we've just buy 2nd
stock so far.

            release1 = Math.max(release1, hold1+i);    // The maximum if we've just sold
1nd stock so far.

            hold1= Math.max(hold1,-i);    // The maximum if we've just buy 1st stock so far.

        }

        return release2;    ///Since release1 is initiated as 0, so release2 will always
higher than release1.

    }

}
```

74. Subarray Sums Divisible by K

Given an integer array `nums` and an integer `k`, return the number of non-empty **subarrays** that have a sum divisible by `k`.

A **subarray** is a **contiguous** part of an array.

Example 1:

Input: `nums = [4,5,0,-2,-3,1]`, `k = 5`

Output: 7

Explanation: There are 7 subarrays with a sum divisible by `k = 5`:

`[4, 5, 0, -2, -3, 1]`, `[5]`, `[5, 0]`, `[5, 0, -2, -3]`, `[0]`, `[0, -2, -3]`, `[-2, -3]`

```
class Solution {
    public int subarraysDivByK(int[] A, int K) {
        int[] map = new int[K];
        map[0] = 1;
        int count = 0, sum = 0;
        for(int a : A) {
            sum = (sum + a) % K;
            if(sum < 0) sum += K;
            count += map[sum];
            map[sum]++;
        }
        return count;
    }
}
```

```
class Solution {
    public int subarraysDivByK(int[] A, int K) {
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);
        int count = 0, sum = 0;
        for(int a : A) {
            sum = (sum + a) % K;
            if(sum < 0) sum += K; // Because -1 % 5 = -1, but we need the positive mod 4
            count += map.getOrDefault(sum, 0);
            map.put(sum, map.getOrDefault(sum, 0) + 1);
        }
        return count;
    }
}
```

1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to `target`*.

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

You can return the answer in any order.

Example 1:

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

Output: `[0,1]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int[] result = new int[2];
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < numbers.length; i++) {
            if (map.containsKey(target - numbers[i])) {
                result[1] = i;
                result[0] = map.get(target - numbers[i]);
                return result;
            }
            map.put(numbers[i], i);
        }
        return result;
    }
}
```

15. 3Sum

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: `nums = [-1,0,1,2,-1,-4]`

Output: `[[-1,-1,2],[-1,0,1]]`

Example 2:

Input: `nums = []`

Output: `[]`

Example 3:

Input: `nums = [0]`

Output: `[]`

```
class Solution {
    public List<List<Integer>> threeSum(int[] num) {
        Arrays.sort(num);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < num.length-2; i++) {
            if (i == 0 || (i > 0 && num[i] != num[i-1])) {
                int lo = i+1, hi = num.length-1, sum = 0 - num[i];
                while (lo < hi) {
                    if (num[lo] + num[hi] == sum) {
                        res.add(Arrays.asList(num[i], num[lo], num[hi]));
                        while (lo < hi && num[lo] == num[lo+1]) lo++;
                        while (lo < hi && num[hi] == num[hi-1]) hi--;
                        lo++; hi--;
                    } else if (num[lo] + num[hi] < sum) lo++;
                    else hi--;
                }
            }
        }
        return res;
    }
}
```


18. 4Sum

Given an array `nums` of `n` integers, return an array of all the unique quadruplets `[nums[a], nums[b], nums[c], nums[d]]` such that:

- `0 <= a, b, c, d < n`
- `a, b, c, and d` are distinct.
- `nums[a] + nums[b] + nums[c] + nums[d] == target`

You may return the answer in any order.

Example 1:

Input: `nums = [1,0,-1,0,-2,2], target = 0`

Output: `[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]`

Example 2:

Input: `nums = [2,2,2,2,2], target = 8`

Output: `[[2,2,2,2]]`

Constraints:

- `1 <= nums.length <= 200`
- `-109 <= nums[i] <= 109`
- `-109 <= target <= 109`

```
class Solution {
    public List<List<Integer>> fourSum(int[] arr, int target) {
        List<List<Integer>> res = new ArrayList<>();
        int n = arr.length;
        if(n<4){ return res; }
        Arrays.sort(arr);
        for(int i = 0;i<=n-4;i++){
            if(i != 0 && arr[i] == arr[i-1]) continue;
            int var1 = arr[i];
            List<List<Integer>> subRes = threeSum(arr,target - var1 , i+1);
            for(List<Integer> list : subRes){
                list.add(var1);
                res.add(list);
            }
        }
        return res;
    }

    public static List<List<Integer>> twoSum(int arr[],int si, int ei,int target){
        int left = si;
        int right = ei;
        List<List<Integer>> res = new ArrayList<>();
        while(left<right){
            if(left != si && arr[left]==arr[left-1]){
```

```

        left++;
        continue;
    }
    int sum = arr[left] + arr[right];
    if(sum == target){
        List<Integer> subres = new ArrayList<>();
        subres.add(arr[left]);
        subres.add(arr[right]);
        res.add(subres);

        left++;
        right--;

    }else if(sum > target){
        right--;
    }else{
        left++;
    }
}
return res;
}

public List<List<Integer>> threeSum(int[] arr , int target, int si) {
    List<List<Integer>> res = new ArrayList<>();
    int n = arr.length;
    if(n-si<3){
        return res;
    }
    for(int i = si;i<=n-3;i++){
        if(i != si && arr[i] == arr[i-1]) continue;
        int var1 = arr[i];
        int targ = target - var1;
        List<List<Integer>> subRes = twoSum(arr,i+1, n-1, targ);
        for(List<Integer> list : subRes){
            list.add(var1);
            res.add(list);
        }
    }
    return res;
}
}

```

442. Find All Duplicates in an Array

Given an integer array `nums` of length `n` where all the integers of `nums` are in the range `[1, n]` and each integer appears **once** or **twice**, return *an array of all the integers that appears **twice***.

You must write an algorithm that runs in $O(n)$ time and uses only constant extra space.

Example 1:

Input: `nums = [4,3,2,7,8,2,3,1]`

Output: `[2,3]`

Example 2:

Input: `nums = [1,1,2]`

Output: `[1]`

Example 3:

Input: `nums = [1]`

Output: `[]`

```
class Solution {
    public List<Integer> findDuplicates(int[] nums) {
        // First Approach
        int[] f = new int[nums.length+1];
        List<Integer> list = new ArrayList<Integer>();
        for(int i=0; i<nums.length; i++){
            f[nums[i]]++;
            if(f[nums[i]]>=2){
                list.add(nums[i]);
            }
        }
        return list;
        // Second Approach
        List<Integer> list = new ArrayList<Integer>();
        for(int i=0; i<nums.length; ++i){
            int index=Math.abs(nums[i])-1;
            if(nums[index]<0){
                list.add(index+1);
            }
            nums[index]=-nums[index];
        }
        return list;
    }
}
```

26. Remove Duplicates from Sorted Array

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array `nums`. More formally, if there are `k` elements after removing the duplicates, then the first `k` elements of `nums` should hold the final result. It does not matter what you leave beyond the first `k` elements.

Return `k` after placing the final result in the first `k` slots of `nums`.

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

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

```
class Solution {
    public int removeDuplicates(int[] nums) {
        int n = nums.length;
        if (n==0 || n == 1){
            return n;
        }
        int j = 0;
        for(int i = 0; i<n-1; i++){
            if (nums[i] != nums[i+1]){
                nums[j++] = nums[i];
            }
        }
        nums[j++] = nums[n - 1];
        return j;
    }
}
```

11. Container with Most Water

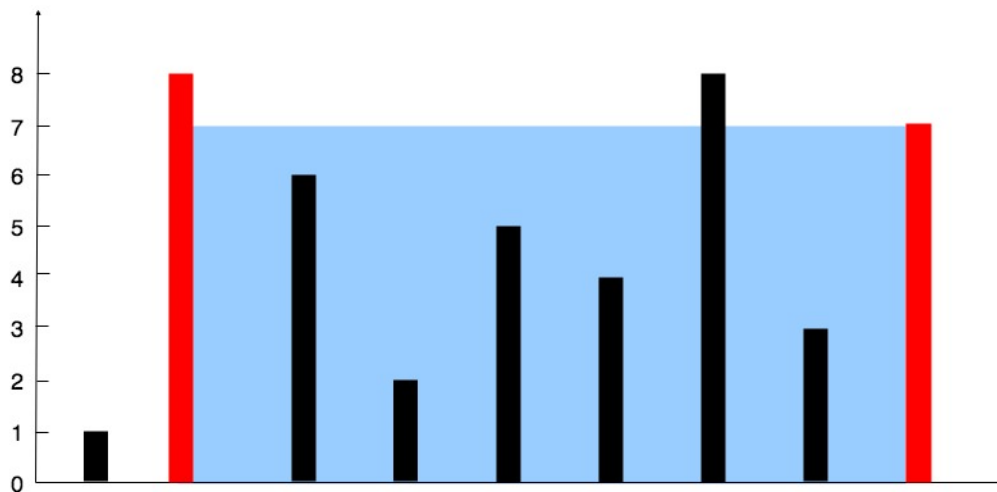
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are $(i, 0)$ and $(i, \text{height}[i])$.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:



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

Output: 49

Explanation: 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 2:

Input: `height = [1,1]`

Output: 1

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

1423. Maximum Points You Can Obtain from Cards

There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array `cardPoints`.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly `k` cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array `cardPoints` and the integer `k`, return the *maximum score* you can obtain.

Example 1:

Input: `cardPoints = [1,2,3,4,5,6,1]`, `k = 3`

Output: 12

Explanation: After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of $1 + 6 + 5 = 12$.

Example 2:

Input: `cardPoints = [2,2,2]`, `k = 2`

Output: 4

Explanation: Regardless of which two cards you take, your score will always be 4.

Example 3:

Input: `cardPoints = [9,7,7,9,7,7,9]`, `k = 7`

Output: 55

Explanation: You have to take all the cards. Your score is the sum of points of all cards.

```
class Solution {
    public int maxScore(int[] cardPoints, int k) {
        int l=0, r=cardPoints.length-1;
        int left=0, right=0, result=0;
        for(int i=r; i>r-k; --i){
            right+=cardPoints[i];
        }
        result = right;

        int i=-1, j=r-k;
        for(int m=0; m<k; ++m){
            ++i;
            ++j;
            left+=cardPoints[i];
            right-=cardPoints[j];
            result=Math.max(result, right+left);
        }
        return result;
    }
}
```

560. Subarray Sum Equals K

Given an array of integers `nums` and an integer `k`, return *the total number of subarrays whose sum equals to `k`*.

A subarray is a contiguous **non-empty** sequence of elements within an array.

Example 1:

Input: `nums = [1,1,1]`, `k = 2`

Output: 2

Example 2:

Input: `nums = [1,2,3]`, `k = 3`

Output: 2

```
class Solution {
    public int subarraySum(int[] nums, int k) {
        int result=0;
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0,1);
        int sum=0;
        for(int i=0; i<nums.length; i++){
            sum+=nums[i];
            if(map.containsKey(sum-k)){
                result+=map.get(sum-k);
            }
            map.put(sum, map.getOrDefault(sum, 0)+1);
        }
        return result;
    }
}
```

54. Spiral Matrix

Given an $m \times n$ matrix, return all elements of the matrix in spiral order.

Example 1:

1 →	2 →	3 ↓
4 →	5	6 ↓
↑ 7 ←	8 ←	9

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

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

Example 2:

1 →	2 →	3 →	4 ↓
5 →	6 →	7	8 ↓
↑ 9 ←	10 ←	11 ←	12

Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

Output: [1,2,3,4,8,12,11,10,9,5,6,7]


```

class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> list = new ArrayList<>();

        int top=0;
        int left=0;
        int right=matrix[0].length-1;
        int bottom = matrix.length-1;
        int dir=0;
        while(top<=bottom && left<=right){
            if(dir==0){
                for(int i=left; i<=right; i++){
                    list.add(matrix[top][i]);
                }
                top++;
            }
            if(dir==1){
                for(int i=top; i<=bottom; i++){
                    list.add(matrix[i][right]);
                }
                right--;
            }
            if(dir==2){
                for(int i=right; i>=left; i--){
                    list.add(matrix[bottom][i]);
                }
                bottom--;
            }

            if(dir==3){
                for(int i=bottom; i>=top; i--){
                    list.add(matrix[i][left]);
                }
                left++;
            }
            dir=(dir+1)%4;
        }
        return list;
    }
}

```

283. Move Zeroes

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

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: `nums = [0,1,0,3,12]`

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

Example 2:

Input: `nums = [0]`

Output: `[0]`

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

79. Word Search

Given an $m \times n$ grid of characters `board` and a string `word`, return `true` if `word` exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

A	B	C	E
S	F	C	S
A	D	E	E

Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, `word = "ABCCED"`

Output: `true`

Example 2:

A	B	C	E
S	F	C	S
A	D	E	E

Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, `word = "SEE"`

Output: `true`

Example 3:

A	B	C	E
S	F	C	S
A	D	E	E

Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, `word = "ABCB"`

Output: `false`

```

class Solution {

    static boolean[][] visited;

    public boolean exist(char[][] board, String word) {

        visited = new boolean[board.length][board[0].length];

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

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

                if((word.charAt(0) == board[i][j]) && search(board, word, i, j, 0)){

                    return true;

                }

            }

        } return false;

    }

    public boolean search(char[][] board, String word, int i, int j, int index){

        if(index == word.length()){

            return true;

        }

        if(i >= board.length || i < 0 || j >= board[i].length || j < 0 || board[i][j] !=
word.charAt(index) || visited[i][j]){

            return false;

        }

        visited[i][j] = true;

        if(search(board, word, i-1, j, index+1) || search(board, word, i+1, j, index+1) ||
        search(board, word, i, j-1, index+1) || search(board, word, i, j+1, index+1)){

            return true;

        }

        visited[i][j] = false;

        return false;

    }

}

```

55. Jump Game

You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` if you can reach the last index, or `false` otherwise.

Example 1:

Input: `nums = [2,3,1,1,4]`

Output: `true`

Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: `nums = [3,2,1,0,4]`

Output: `false`

Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

```
// Approach 1
class Solution {
    public boolean canJump(int[] nums) {
        int max = 0;
        for(int i = 0; i < nums.length; i++){
            if(max < i){
                return false;
            }
            max = (i + nums[i]) > max ? i + nums[i] : max;
        }
        return true;
    }
}
```

```
// Approach 2
class Solution {
    public boolean canJump(int[] nums) {
        int lastIndexCanReach = 0;
        for(int i=0; i <= lastIndexCanReach; i++){
            lastIndexCanReach = Math.max(lastIndexCanReach, nums[i]+i);
            if(lastIndexCanReach >= nums.length-1) return true;
        }
        return false;
    }
}
```

66. Plus One

You are given a **large integer** represented as an integer array `digits`, where each `digits[i]` is the *i*th digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: `digits = [1,2,3]`

Output: `[1,2,4]`

Explanation: The array represents the integer 123.

Incrementing by one gives $123 + 1 = 124$.

Thus, the result should be `[1,2,4]`.

Example 2:

Input: `digits = [4,3,2,1]`

Output: `[4,3,2,2]`

Explanation: The array represents the integer 4321.

Incrementing by one gives $4321 + 1 = 4322$.

Thus, the result should be `[4,3,2,2]`.

Example 3:

Input: `digits = [9]`

Output: `[1,0]`

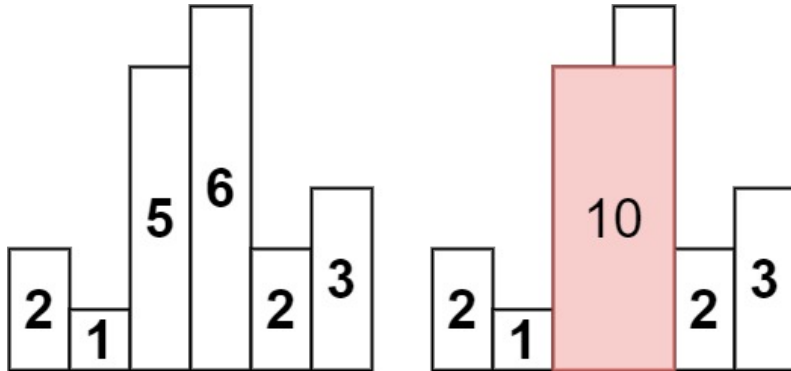
```
class Solution {
    public int[] plusOne(int[] digits) {
        int n = digits.length;
        for(int i=n-1; i>=0; i--) {
            if(digits[i] < 9) {
                digits[i]++;
                return digits;
            }
            digits[i] = 0;
        }
        int[] newNumber = new int [n+1];
        newNumber[0] = 1;

        return newNumber;
    }
}
```

Largest Rectangle in Histogram

Given an array of integers `heights` representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:



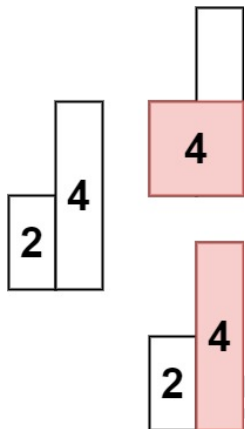
Input: heights = [2,1,5,6,2,3]

Output: 10

Explanation: The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:



Input: heights = [2,4]

Output: 4

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        int maxArea = Integer.MIN_VALUE;

        int[] prevSmaller = getPrevSmaller(heights);
        //System.out.println(Arrays.toString(prevSmaller));
        int[] nextSmaller = getNextSmaller(heights);
        //System.out.println(Arrays.toString(nextSmaller));
```

```

/**
 *      1  6  4  4  6  6      --> (indices of next smaller element)
 *      <-- -1 -1  1  2  1  4      (indices of prev smaller element)
 *      2, 1, 5, 6, 2, 3      (Array of heights)
 * Indices = 0  1  2  3  4  5
 */
for (int i = 0; i < heights.length; i++) {

    /**
     * At every height, find the next smaller height and prev smaller height
     * and subtract 1 from the distance between the 2 smaller heights so that
     * we EXCLUDE BOTH the smaller heights (prev AND next) from either sides.
     *
     * Once we find that horizontal distance, multiply that with the current
     * height to get the current max area
     */
    int currentMaxArea = (nextSmaller[i] - prevSmaller[i] - 1) * heights[i];
    maxArea = Math.max(maxArea, currentMaxArea);
}

return maxArea;
}

private int[] getNextSmaller(int[] h) {
    int[] nextSmaller = new int[h.length];

    Stack<Integer> s = new Stack<Integer>();

    /**
     *      1  6  4  4  6  6      --> (indices of next smaller element)
     *      2, 1, 5, 6, 2, 3 (Array of heights)
     * Indices = 0  1  2  3  4  5
     */
    int len = h.length;
    for (int i = len - 1; i >= 0; i--) {
        while (!s.isEmpty() && h[i] <= h[s.peek()]) {
            s.pop(); // Keep removing from stack till stack is empty OR we find
// element just smaller or equal to h[i]
        }
        if (s.isEmpty()) { // If stack becomes empty
            nextSmaller[i] = len; // Add next smaller element LOCATION as length,
// which would be 1 index more than the last index
        } else {
            nextSmaller[i] = s.peek(); // Else add LOCATION as top element in stack
        }
        s.push(i);
    }

    return nextSmaller;
}

```



```

private int[] getPrevSmaller(int[] h) {
    int[] prevSmaller = new int[h.length];
    Stack<Integer> s = new Stack<Integer>();
    /**
     *      <-- -1 -1  1  2  1  4 (indices of prev smaller element)
     *          2, 1, 5, 6, 2, 3 (Array of heights)
     * Indices = 0  1  2  3  4  5
     */
    for (int i = 0; i < h.length; i++) {
        while (!s.isEmpty() && h[i] <= h[s.peek()]) {
            s.pop(); // Keep removing from stack till stack is empty OR we find
element just smaller or equal to h[i]
        }
        if (s.isEmpty()) { // If stack becomes empty
            prevSmaller[i] = -1; // Add prev smaller element LOCATION as -1
        } else {
            prevSmaller[i] = s.peek(); // Else add LOCATION as top element in stack
        }
        s.push(i);
    }

    return prevSmaller;
}
}

```

```

class Solution {
    public int largestRectangleArea(int[] heights) {
        int len = heights.length;
        int[] left = new int[len];
        left[0] = -1;
        int[] right = new int[len];
        right[len-1] = len;
        for (int i = 1; i < len; ++i){
            int j = i-1;
            while (j >= 0 && heights[j] >= heights[i]) j = left[j];
            left[i] = j;
        }
        for (int i = len-2; i >= 0; --i){
            int j = i+1;
            while (j < len && heights[j] >= heights[i]) j = right[j];
            right[i] = j;
        }
        int res = 0;
        for (int i = 0; i < len; ++i) res = Math.max(res, heights[i]*(right[i]-left[i]-
1));
        return res;
    }
}

```

289. Game of Life

According to [Wikipedia's article](#): "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."


The board is made up of an $m \times n$ grid of cells, where each cell has an initial state: **live** (represented by a 1) or **dead** (represented by a 0). Each cell interacts with its [eight neighbors](#) (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the $m \times n$ grid `board`, return *the next state*.

Example 1:

0	1	0
0	0	1
1	1	1
0	0	0




0	0	0
1	0	1
0	1	1
0	1	0

Input: `board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]`

Output: `[[0,0,0],[1,0,1],[0,1,1],[0,1,0]]`

Example 2:

1	1
1	0



1	1
1	1

Input: `board = [[1,1],[1,0]]`

Output: `[[1,1],[1,1]]`

```

class Solution {
    public void gameOfLife(int[][] board) {
        int[][] mark = new int[board.length][board[0].length];
        for(int i=0; i<board.length; i++){
            for(int j=0; j<board[i].length; j++){
                int neighbour = countNeighbour(board, i, j);
                // First Rule
                if(board[i][j]==1 && neighbour<2){
                    mark[i][j]=0;
                }
                // Second Rule
                else if(board[i][j]==1 && (neighbour == 2 || neighbour == 3)){
                    mark[i][j]=1;
                }
                // Third Rule
                else if(board[i][j]==1 && neighbour>3){
                    mark[i][j]=0;
                }
                // Fourth Rule
                else if(board[i][j]==0 && neighbour == 3){
                    mark[i][j]=1;
                }
            }
        }
        for(int i=0; i< board.length; i++){
            for(int j=0; j<board[0].length; j++){
                board[i][j]=mark[i][j];
            }
        }
    }

    public int countNeighbour(int[][] board, int i, int j){
        int count=0;
        // All 8 direction co-ordinates
        int[][] direction =
        {
            {0, 1},    // right
            {0, -1},   // left
            {-1, 0},   // top
            {1, 0},    // bottom
            {1, -1},   // bottom-left
            {1, 1},    // bottom-right
            {-1, 1},   // top-right
            {-1, -1}}; // top-left
        for(int[] dir:direction){
            int x = i+dir[0]; // Adding the resultant x co-ordinates
            int y = j+dir[1]; // Adding the resultant y co-ordinates
            // Checking the corner case condition
            if(x>=0 && y>=0 && x<board.length && y<board[0].length){
                count+=board[x][y];
            }
        }
        return count;
    }
}

```

2133. Check if Every Row and Column Contains All Numbers

An $n \times n$ matrix is **valid** if every row and every column contains **all** the integers from 1 to n (**inclusive**).

Given an $n \times n$ integer matrix `matrix`, return `true` if the matrix is **valid**. Otherwise, return `false`.

Example 1:

1	2	3
3	1	2
2	3	1

Input: `matrix = [[1,2,3],[3,1,2],[2,3,1]]`

Output: `true`

Explanation: In this case, $n = 3$, and every row and column contains the numbers 1, 2, and 3.

Hence, we return `true`.

```
class Solution {
    public boolean checkValid(int[][] matrix) {
        int n = matrix.length;
        for(int i=0; i<n; ++i){
            Set<Integer> setRow = new HashSet<>();
            Set<Integer> setCol = new HashSet<>();
            for(int j=0; j<n; ++j){
                if(!setRow.add(matrix[i][j]) || !setCol.add(matrix[j][i])){
                    return false;
                }
            }
        }
        return true;
    }
}
```

746. Min Cost Climbing Stairs

You are given an integer array `cost` where `cost[i]` is the cost of i^{th} step on a staircase. Once you pay the cost, you can either climb one or two steps. You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor*.

Example 1:

Input: `cost = [10,15,20]`

Output: 15

Explanation: You will start at index 1.

- Pay 15 and climb two steps to reach the top.

The total cost is 15.

Example 2:

Input: `cost = [1,100,1,1,1,100,1,1,100,1]`

Output: 6

Explanation: You will start at index 0.

- Pay 1 and climb two steps to reach index 2.

- Pay 1 and climb two steps to reach index 4.

- Pay 1 and climb two steps to reach index 6.

- Pay 1 and climb one step to reach index 7.

- Pay 1 and climb two steps to reach index 9.

- Pay 1 and climb one step to reach the top.

The total cost is 6.

```
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int n = cost.length;
        int[] dp = new int[n];
        for(int i=0; i<n; i++){
            if(i<2){
                dp[i] = cost[i];
            }else{
                // current cost + Minimum of previous two steps
                dp[i] = cost[i] + Math.min(dp[i-1], dp[i-2]);
            }
        }
        return Math.min(dp[n-1], dp[n-2]);
    }
}
```

1654. Minimum Jumps to Reach Home

A certain bug's home is on the x-axis at position `x`. Help them get there from position `0`.

The bug jumps according to the following rules:

- It can jump exactly `a` positions **forward** (to the right).
- It can jump exactly `b` positions **backward** (to the left).
- It cannot jump backward twice in a row.
- It cannot jump to any `forbidden` positions.

The bug may jump forward **beyond** its home, but it **cannot jump** to positions numbered with **negative** integers.

Given an array of integers `forbidden`, where `forbidden[i]` means that the bug cannot jump to the position `forbidden[i]`, and integers `a`, `b`, and `x`, return *the minimum number of jumps needed for the bug to reach its home*. If there is no possible sequence of jumps that lands the bug on position `x`, return `-1`.

Example 1:

Input: `forbidden = [14,4,18,1,15]`, `a = 3`, `b = 15`, `x = 9`

Output: `3`

Explanation: 3 jumps forward (`0 -> 3 -> 6 -> 9`) will get the bug home.

Example 2:

Input: `forbidden = [8,3,16,6,12,20]`, `a = 15`, `b = 13`, `x = 11`

Output: `-1`

Example 3:

Input: `forbidden = [1,6,2,14,5,17,4]`, `a = 16`, `b = 9`, `x = 7`

Output: `2`

Explanation: One jump forward (`0 -> 16`) then one jump backward (`16 -> 7`) will get the bug home.

```
class Solution {  
    class Position {  
        int val;  
        int direction;  
  
        Position(int val, int direction) {  
            this.val = val;  
            this.direction = direction;  
        }  
  
        @Override  
        public String toString() {  
            return this.val + " " + this.direction;  
        }  
    }  
}
```

```

    }
}

public int minimumJumps(int[] forbidden, int a, int b, int x) {
    if(x==0){
        return 0;
    }
    int steps = 0;
    int furthest = 10000;
    Set<Integer> forbiddenSpots = new HashSet();
    Set<String> visitedSet = new HashSet<>();

    for(int i=0; i<forbidden.length; i++){
        forbiddenSpots.add(forbidden[i]);
    }
    Queue<Position> queue = new LinkedList();
    queue.add(new Position(0,0));
    visitedSet.add(0+"-"+0);
    steps++;
    // BFS Starts
    while(!queue.isEmpty()){
        int size = queue.size();

        for(int i=0; i<size; i++){
            Position current_position = queue.poll();

            int next_a = current_position.val + a;
            if(next_a == x){
                return steps;
            }
            if(next_a>=0 && next_a<= furthest && !forbiddenSpots.contains(next_a) &&
visitedSet.add(next_a + "-" + 1)){
                queue.add(new Position(next_a,1));
            }
            if(current_position.direction==1){
                int next_b = current_position.val-b;
                if(next_b==x){
                    return steps;
                }
                if(next_b>=0 && next_b<=furthest && !forbiddenSpots.contains(next_b)
&& visitedSet.add(next_b + "-" + a)){
                    queue.add(new Position(next_b, 0));
                }
            }
        }
        steps++;
    }
    return -1;
}
}

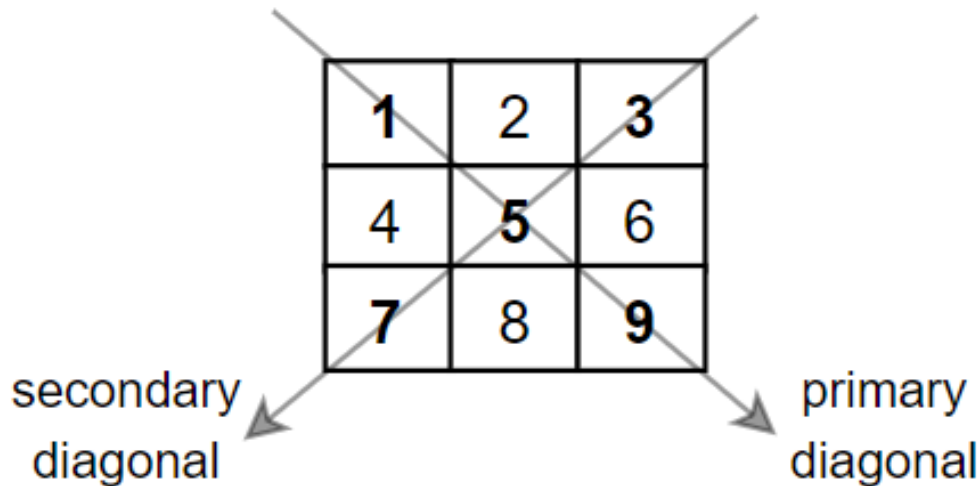
```

1572. Matrix Diagonal Sum

Given a square matrix `mat`, return the sum of the matrix diagonals.

Only include the sum of all the elements on the primary diagonal and all the elements on the secondary diagonal that are not part of the primary diagonal.

Example 1:



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

Output: 25

Explanation: Diagonals sum: $1 + 5 + 9 + 3 + 7 = 25$

Notice that element `mat[1][1] = 5` is counted only once.

Example 2:

Input: `mat = [[5]]`

Output: 5

```
class Solution {
    public int diagonalSum(int[][] mat) {
        int sum=0;
        int n = mat.length;
        for(int i=0; i<n; i++){
            // sum+=(topLeft-bottomRight)+(topRight-bottomLeft)
            sum+=mat[i][i]+mat[i][n-1-i];
        }
        // even matrix have common diagonal, remove that element from sum
        return n%2==0? sum : (sum-mat[n/2][n/2]);
    }
}
```



```
class Solution {
    public int diagonalSum(int[][] mat) {
        int sum=0;
        int n = mat.length;
        int j = n-1,i=0;
        for(i=0; i<j; i++, j--){
            // sum+=(topleft-center)+(topRight-center)+(bottomLeft-center)+(bottomRight-center)
            sum+=mat[i][i]+mat[i][j]+mat[j][i]+mat[j][j];
        }
        // add the center diagonal
        if(i==j){
            sum+=mat[i][i];
        }
        return sum;
    }
}
```

350. Intersection of Two Arrays II

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

Example 1:

Input: `nums1 = [1,2,2,1]`, `nums2 = [2,2]`

Output: `[2,2]`

Example 2:

Input: `nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`

Output: `[4,9]`

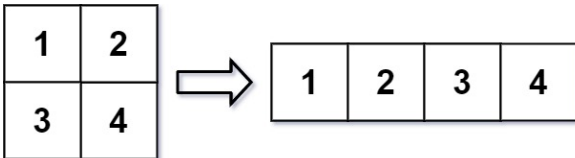
Explanation: `[9,4]` is also accepted.

```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        List<Integer> list = new ArrayList<>();
        int[] f = new int[1001];
        for(int i=0; i<nums1.length; i++){
            f[nums1[i]]++;
        }
        for(int i=0; i<nums2.length; i++){
            if(f[nums2[i]]>0){
                list.add(nums2[i]);
                f[nums2[i]]--;
            }
        }
        int[] ans = new int[list.size()];
        for(int i=0; i<list.size(); i++){
            ans[i]=list.get(i);
        }
        return ans;
    }
}
```

566. Reshape the Matrix

In MATLAB, there is a handy function called `reshape` which can reshape an $m \times n$ matrix into a new one with a different size $r \times c$ keeping its original data. You are given an $m \times n$ matrix `mat` and two integers `r` and `c` representing the number of rows and the number of columns of the wanted reshaped matrix. The reshaped matrix should be filled with all the elements of the original matrix in the same row-traversing order as they were. If the `reshape` operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

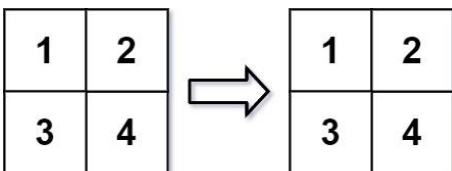
Example 1:



Input: `mat = [[1,2],[3,4]]`, `r = 1`, `c = 4`

Output: `[[1,2,3,4]]`

Example 2:



Input: `mat = [[1,2],[3,4]]`, `r = 2`, `c = 4`

Output: `[[1,2],[3,4]]`

```
class Solution {
    public int[][] matrixReshape(int[][] mat, int r, int c) {
        if((mat.length*mat[0].length)!= (r*c)){
            return mat;
        }
        if(mat.length==r && mat[0].length==c){
            return mat;
        }
        int[] table = new int[(mat.length)*(mat[0].length)];
        int n = mat[0].length;
        for(int i=0; i<mat.length; i++){
            for(int j=0; j<mat[0].length; j++){
                table[n*i+j]=mat[i][j];
            }
        }
        int[][] result = new int[r][c];
        for(int i=0; i<table.length; i++){
            result[i/c][i%c] = table[i];
        }
        return result;
    }
}
```

GitHub.com/AyeRaj