**[287. Find the Duplicate Number](https://leetcode.com/problems/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

**Example 3:**

**Input:** nums = [3,3,3,3,3]

**Output:** 3

**Constraints:**

* 1 <= n <= 105
* nums.length == n + 1
* 1 <= nums[i] <= n
* All the integers in nums appear only **once** except for **precisely one integer** which appears **two or more** times.

**Follow up:**

* How can we prove that at least one duplicate number must exist in nums?
* Can you solve the problem in linear runtime complexity?
* public class Solution {
* public int findDuplicate(int[] nums) {
* int slow = nums[0];
* int fast = nums[0];
* do {
* slow = nums[slow];
* fast = nums[nums[fast]];
* } while (slow != fast);
* slow = nums[0];
* while (slow != fast) {
* slow = nums[slow];
* fast = nums[fast];
* }
* return slow;
* }
* }

[**75. Sort Colors**](https://leetcode.com/problems/sort-colors/)

Given an array nums with n objects colored red, white, or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)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]

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* nums[i] is either 0, 1, or 2.

**Follow up:** Could you come up with a one-pass algorithm using only constant extra space?

class Solution {

    public void sortColors(int[] nums) {

        int left= 0;

        int right= nums.length-1;

        int ptr;

        int zero=0;

        int one=0;

        int two=0;

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

            if(nums[i]==0) zero++;

            else if(nums[i]==1) one++;

            else two++;

        }

        ptr=zero;

        while(ptr<=right){

            if(nums[ptr]==0){

                int temp=nums[ptr];

                nums[ptr]=nums[left];

                nums[left]=temp;;

                left++;

            }

            else if(nums[ptr]==2){

                int temp=nums[ptr];

                nums[ptr]=nums[right];

                nums[right]=temp;

                right--;

            }

            else{

                ptr++;

            }

        }

    }

}

[**26. Remove Duplicates from Sorted Array**](https://leetcode.com/problems/remove-duplicates-from-sorted-array/)

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in*nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

* Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
* Return k.

**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**.

**Example 1:**

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

**Output:** 2, nums = [1,2,\_]

**Explanation:** Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**Example 2:**

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

**Output:** 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

**Explanation:** Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -100 <= nums[i] <= 100
* nums is sorted in **non-decreasing** order.
* class Solution {
* public int removeDuplicates(int[] nums) {
* int j=1;
* for(int i=1;i<nums.length;i++){
* if(nums[i]!=nums[i-1]){
* nums[j]=nums[i];
* j++;
* }
* }
* return j;
* }

}

[**73. Set Matrix Zeroes**](https://leetcode.com/problems/set-matrix-zeroes/)

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

You must do it [in place](https://en.wikipedia.org/wiki/In-place_algorithm).

**Example 1:**



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

**Output:** [[1,0,1],[0,0,0],[1,0,1]]

**Example 2:**



**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]]

**Constraints:**

* m == matrix.length
* n == matrix[0].length
* 1 <= m, n <= 200
* -231 <= matrix[i][j] <= 231 - 1

**Follow up:**

* A straightforward solution using O(mn) space is probably a bad idea.
* A simple improvement uses O(m + n) space, but still not the best solution.
* Could you devise a constant space solution?
* class Solution {
* public void setZeroes(int[][] matrix) {
* int m = matrix.length;
* int n = matrix[0].length;
* boolean firstRowHasZero = false;
* boolean firstColHasZero = false;
* for (int j = 0; j < n; j++) {
* if (matrix[0][j] == 0) {
* firstRowHasZero = true;
* break;
* }
* }
* for (int i = 0; i < m; i++) {
* if (matrix[i][0] == 0) {
* firstColHasZero = true;
* break;
* }
* }
* for (int i = 1; i < m; i++) {
* for (int j = 1; j < n; j++) {
* if (matrix[i][j] == 0) {
* matrix[i][0] = 0;
* matrix[0][j] = 0;
* }
* }
* }
* for (int i = 1; i < m; i++) {
* for (int j = 1; j < n; j++) {
* if (matrix[i][0] == 0 || matrix[0][j] == 0) {
* matrix[i][j] = 0;
* }
* }
* }
* if (firstRowHasZero) {
* for (int j = 0; j < n; j++) {
* matrix[0][j] = 0;
* }
* }
* if (firstColHasZero) {
* for (int i = 0; i < m; i++) {
* matrix[i][0] = 0;
* }
* }
* }
* }

[**283. Move Zeroes**](https://leetcode.com/problems/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]

**Constraints:**

* 1 <= nums.length <= 104
* -231 <= nums[i] <= 231 - 1

**Follow up:** Could you minimize the total number of operations done?

class Solution {

    public void moveZeroes(int[] nums) {

        int n= nums.length;

        int i=0,j=0;

        while(i<n && j<n){

            if(nums[i]==0 && nums[j]!=0){

                int temp= nums[i];

                nums[i]=nums[j];

                nums[j]=temp;

                i++;

                j++;

            }

            else if(nums[i]!=0){

                i++;

                j++;

            }

            else if(nums[i]==0){

                j++;

            }

        }

    }

}

[**121. Best Time to Buy and Sell Stock**](https://leetcode.com/problems/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 ith 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.

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104
* class Solution {
* public int maxProfit(int[] prices) {
* int n= prices.length;
* int curr=Integer.MAX\_VALUE;
* int profit=0;
* for(int i=0;i<n;i++){
* if(prices[i]<curr){
* curr=prices[i];
* }
* else{
* profit= Math.max(profit,prices[i]-curr);
* }
* }
* return profit;
* }
* }

Chocolate Distribution problem

Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :  
1. Each student gets **exactly** one packet.  
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

**Example 1:**

**Input:**

N = 8, M = 5

A = {3, 4, 1, 9, 56, 7, 9, 12}

**Output:** 6

**Explanation:** The minimum difference between maximum chocolates and minimum chocolates is 9 - 3 = 6 by choosing following M packets :{3, 4, 9, 7, 9}.

**Example 2:**

**Input:**

N = 7, M = 3

A = {7, 3, 2, 4, 9, 12, 56}

**Output:** 2

**Explanation:** The minimum difference between maximum chocolates and minimum chocolates is 4 - 2 = 2 by choosing following M packets :{3, 2, 4}.

**Your Task:**  
You don't need to take any input or print anything. Your task is to complete the function **findMinDiff()**which takes array A[ ], N and M as input parameters and returns the minimum possible difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student.

**Expected Time Complexity:**O(N\*Log(N))  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ T ≤ 100  
1 ≤ N ≤ 105  
1 ≤ Ai ≤ 109  
1 ≤ M ≤ N

class Solution

{

public long findMinDiff (ArrayList<Integer> a, int n, int m)

{

// your code here

Collections.sort(a);

long res= Integer.MAX\_VALUE;

for(int i=m-1;i<n;i++){

res= Math.min(res, a.get(i)-a.get(i-m+1));

}

return res;

}

}

[**122. Best Time to Buy and Sell Stock II**](https://leetcode.com/problems/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 ith 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.

**Constraints:**

* 1 <= prices.length <= 3 \* 104
* 0 <= prices[i] <= 104
* class Solution {
* public int maxProfit(int[] prices) {
* int res=0;
* for(int i=0;i<prices.length-1;i++){
* int profit=prices[i+1]-prices[i];
* if(profit>0) res+=profit;
* }
* return res;
* }
* }

[**974. Subarray Sums Divisible by K**](https://leetcode.com/problems/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]

**Example 2:**

**Input:** nums = [5], k = 9

**Output:** 0

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* 2 <= k <= 104
* class Solution {
* public int subarraysDivByK(int[] nums, int k) {
* int n = nums.length;
* Map<Integer, Integer> sumCount = new HashMap<>();
* sumCount.put(0, 1);
* int sum = 0;
* int ans = 0;
* for (int i = 0; i < n; i++) {
* sum += nums[i];
* int mod = ((sum % k) + k) % k;
* if (sumCount.containsKey(mod)) {
* ans += sumCount.get(mod);
* }
* sumCount.put(mod, sumCount.getOrDefault(mod, 0) + 1);
* }
* return ans;
* }
* }

[**442. Find All Duplicates in an Array**](https://leetcode.com/problems/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:** []

**Constraints:**

* n == nums.length
* 1 <= n <= 105
* 1 <= nums[i] <= n
* Each element in nums appears **once** or **twice**.
* class Solution {
* public List<Integer> findDuplicates(int[] nums) {
* List<Integer> res= new ArrayList<>();
* for(int i=0;i<nums.length;i++){
* int ind= Math.abs(nums[i]);
* if(nums[ind-1]<0){
* res.add(ind);
* }
* nums[ind-1]\*=-1;
* }
* return res;
* }
* }

[**11. Container With Most Water**](https://leetcode.com/problems/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 ith line are (i, 0) and (i, 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

**Constraints:**

* n == height.length
* 2 <= n <= 105
* 0 <= height[i] <= 104
* class Solution {
* public int maxArea(int[] height) {
* int max=0;
* int l=0;
* int r=height.length-1;
* while(l<r){
* int area=Math.min(height[l],height[r])\*(r-l);
* max=Math.max(max,area);
* if(height[l]<height[r]){
* l++;
* }
* else{
* r--;
* }
* }
* return max;
* }
* }

[**15. 3Sum**](https://leetcode.com/problems/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]]

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

**Input:** nums = [0,1,1]

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

**Example 3:**

**Input:** nums = [0,0,0]

**Output:** [[0,0,0]]

**Explanation:** The only possible triplet sums up to 0.

**Constraints:**

* 3 <= nums.length <= 3000
* -105 <= nums[i] <= 105
* class Solution {
* public List<List<Integer>> threeSum(int[] nums) {
* List<List<Integer>> res = new ArrayList<>();
* Arrays.sort(nums);
* int n = nums.length;
* for (int i = 0; i < n - 2; i++) {
* if (i > 0 && nums[i] == nums[i - 1]) continue;
* int j = i + 1;
* int k = n - 1;
* while (j < k) {
* int sum = nums[i] + nums[j] + nums[k];
* if (sum == 0) {
* res.add(Arrays.asList(nums[i], nums[j], nums[k]));
* while (j < k && nums[j] == nums[j + 1]) j++;
* while (j < k && nums[k] == nums[k - 1]) k--;
* j++;
* k--;
* }
* else if (sum < 0) {
* j++;
* } else {
* k--;
* }
* }
* }
* return res;
* }
* }

[**18. 4Sum**](https://leetcode.com/problems/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[] nums, int target) {

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

        if(nums.length<4) return res;

        Arrays.sort(nums);

        // System.out.println(Arrays.toString(nums));

        int i=0;

        int j=1;

        int k=2;

        int l=nums.length-1;

        while(i<nums.length-3){

            j=i+1;

            while(j<nums.length-2){

                k=j+1;

                l=nums.length-1;

                while(k<l){

                    // System.out.println(i+" "+j+" "+k+" "+l);

                    long sum=(long)nums[i]+nums[j]+nums[k]+nums[l];

                    if(sum==target){

                        res.add(Arrays.asList(nums[i],nums[j],nums[k],nums[l]));

                        while(k<nums.length-1 && nums[k+1]==nums[k]){

                            k++;

                        }

                        while(l>k && nums[l-1]==nums[l]){

                            l--;

                        }

                        k++;

                        l--;

                    }

                    else if(sum<target){

                        while(k<nums.length-1 && nums[k+1]==nums[k]){

                            k++;

                        }

                        k++;

                    }

                    else{

                        while(l>k && nums[l-1]==nums[l]){

                            l--;

                        }

                        l--;

                    }

                }

                while(j<nums.length-1 && nums[j+1]==nums[j]){

                    j++;

                }

                j++;

            }

            while(i<nums.length-1 && nums[i+1]==nums[i]){

                i++;

            }

            i++;

        }

        return res;

    }

}

[**1423. Maximum Points You Can Obtain from Cards**](https://leetcode.com/problems/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.

**Constraints:**

* 1 <= cardPoints.length <= 105
* 1 <= cardPoints[i] <= 104
* 1 <= k <= cardPoints.length
* class Solution {
* public int maxScore(int[] cardPoints, int k) {
* int n= cardPoints.length;
* int l= n-k;
* int res= 0;
* int sum=0;
* int curr=0;
* for(int cardPoint:cardPoints){
* sum+=cardPoint;
* }
* // System.out.println(sum);
* for(int i=0;i<l;i++){
* curr+=cardPoints[i];
* }
* res= sum-curr;
* // System.out.println(res);
* for(int i=l;i<n;i++){
* curr-=cardPoints[i-l];
* curr+=cardPoints[i];
* res=Math.max(res,sum-curr);
* // System.out.println(res);
* }
* return res;
* }
* }

[**560. Subarray Sum Equals K**](https://leetcode.com/problems/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

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* -1000 <= nums[i] <= 1000
* -107 <= k <= 107
* class Solution {
* public int subarraySum(int[] nums, int k) {
* int sum=0;
* int ans=0;
* Map<Integer, Integer> hm= new HashMap<>();
* hm.put(0,1);
* for(int num:nums){
* sum+=num;
* if(hm.containsKey(sum-k)){
* ans+=hm.get(sum-k);
* }
* hm.put(sum,hm.getOrDefault(sum,0)+1);
* }
* return ans;
* }
* }

[**54. Spiral Matrix**](https://leetcode.com/problems/spiral-matrix/)

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**



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

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

**Example 2:**



**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]

**Constraints:**

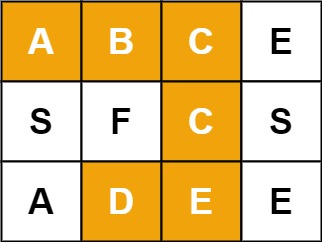
* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 10
* -100 <= matrix[i][j] <= 100
* class Solution {
* public List<Integer> spiralOrder(int[][] matrix) {
* int m= matrix.length;
* int n= matrix[0].length;
* int left=0;
* int right=n-1;
* int top=0;
* int bottom= m-1;
* List<Integer> res= new ArrayList<>();
* while(top<=bottom && left<=right){
* for(int i=left;i<=right;i++){
* res.add(matrix[top][i]);
* }
* top++;
* for(int i=top;i<=bottom;i++){
* res.add(matrix[i][right]);
* }
* right--;
* if(top<=bottom){
* for(int i=right;i>=left;i--){
* res.add(matrix[bottom][i]);
* }
* bottom--;
* }
* if(left<=right){
* for(int i=bottom;i>=top;i--){
* res.add(matrix[i][left]);
* }
* left++;
* }
* }
* return res;
* }
* }

[**79. Word Search**](https://leetcode.com/problems/word-search/)

Given an m x 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.

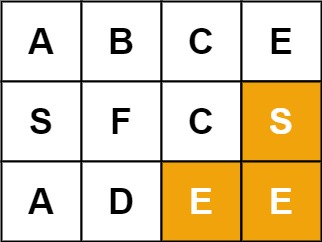
**Example 1:**



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

**Output:** true

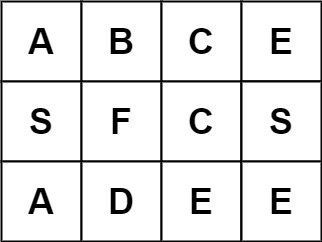
**Example 2:**



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

**Output:** true

**Example 3:**



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

**Output:** false

**Constraints:**

* m == board.length
* n = board[i].length
* 1 <= m, n <= 6
* 1 <= word.length <= 15
* board and word consists of only lowercase and uppercase English letters.

**Follow up:** Could you use search pruning to make your solution faster with a larger board?

class Solution {

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

        int m= board.length;

        int n= board[0].length;

        boolean flag=false;

        for(int i=0;i<m;i++){

            for(int j=0;j<n;j++){

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

                    flag=dfs(i,j,board,word, 0);

                    if(flag) return flag;

                }

            }

        }

        return flag;

    }

    boolean dfs(int i, int j, char board[][], String word, int ind){

        if(ind==word.length()){

            return true;

        }

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

            return false;

        }

            char c= board[i][j];

            board[i][j]='.';

            boolean found = dfs(i + 1, j, board, word, ind + 1) ||

                        dfs(i - 1, j, board, word, ind + 1) ||

                        dfs(i, j + 1, board, word, ind + 1) ||

                        dfs(i, j - 1, board, word, ind + 1);

            board[i][j]=c;

        return found;

    }

}

[**88. Merge Sorted Array**](https://leetcode.com/problems/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.

**Constraints:**

* nums1.length == m + n
* nums2.length == n
* 0 <= m, n <= 200
* 1 <= m + n <= 200
* -109 <= nums1[i], nums2[j] <= 109

**Follow up:**Can you come up with an algorithm that runs in O(m + n) time?

class Solution {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        int i=m-1,j=n-1,k=m+n-1;

        while(j>=0){

            if(i>=0 && nums1[i]>nums2[j]){

                nums1[k--]=nums1[i--];

            }

            else{

                nums1[k--]=nums2[j--];

            }

        }

    }

}

[**55. Jump Game**](https://leetcode.com/problems/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.

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 105
* class Solution {
* public boolean canJump(int[] nums) {
* int reachable = 0;
* for(int i = 0; i < nums.length; i ++) {
* if(i > reachable) return false;
* reachable = Math.max(reachable, i + nums[i]);
* }
* return true;
* }
* }

[**169. Majority Element**](https://leetcode.com/problems/majority-element/)

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

The majority element is the element that appears more than ⌊n / 2⌋ 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

**Constraints:**

* n == nums.length
* 1 <= n <= 5 \* 104
* -109 <= nums[i] <= 109

**Follow-up:** Could you solve the problem in linear time and in O(1) space?

class Solution {

    public int majorityElement(int[] nums) {

        int count = 0;

        int candidate = 0;

        for (int num : nums) {

            if (count == 0) {

                candidate = num;

            }

            if (num == candidate) {

                count++;

            }

            else {

                count--;

            }

        }

        return candidate;

    }

}

[**493. Reverse Pairs**](https://leetcode.com/problems/reverse-pairs/)

Given an integer array nums, return *the number of****reverse pairs****in the array*.

A **reverse pair** is a pair (i, j) where:

* 0 <= i < j < nums.length and
* nums[i] > 2 \* nums[j].

**Example 1:**

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

**Output:** 2

**Explanation:** The reverse pairs are:

(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 \* 1

(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 \* 1

**Example 2:**

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

**Output:** 3

**Explanation:** The reverse pairs are:

(1, 4) --> nums[1] = 4, nums[4] = 1, 4 > 2 \* 1

(2, 4) --> nums[2] = 3, nums[4] = 1, 3 > 2 \* 1

(3, 4) --> nums[3] = 5, nums[4] = 1, 5 > 2 \* 1

**Constraints:**

* 1 <= nums.length <= 5 \* 104
* -231 <= nums[i] <= 231 - 1
* class Solution {
* public int reversePairs(int[] nums) {
* int low=0;
* int high=nums.length-1;
* return mergeSort(low,high, nums);
* }
* int mergeSort(int low, int high, int nums[]){
* int cnt=0;
* if(low>=high) return cnt;
* int mid= low+(high-low)/2;
* cnt+=mergeSort(low,mid,nums);
* cnt+=mergeSort(mid+1,high,nums);
* cnt+=countPairs(nums, low, mid, high);
* merge(nums, low, mid, high);
* return cnt;
* }
* int countPairs(int nums[], int low, int mid, int high){
* int cnt=0;
* int j=mid+1;
* for(int i=low;i<=mid;i++){
* while(j<=high && (long)nums[i]>2L\*nums[j]){
* j++;
* }
* cnt+=(j-(mid+1));
* }
* return cnt;
* }
* void merge(int nums[], int low, int mid, int high){
* int temp[]= new int[high-low+1];
* int ind=0;
* int i=low;
* int j=mid+1;
* while(i<=mid && j<=high){
* if(nums[i]<nums[j]){
* temp[ind++]=nums[i++];
* }
* else{
* temp[ind++]=nums[j++];
* }
* }
* while(i<=mid){
* temp[ind++]=nums[i++];
* }
* while(j<=high){
* temp[ind++]=nums[j++];
* }
* for(i=0;i<temp.length;i++){
* nums[low+i]= temp[i];
* }
* }
* }

[**47. Permutations II**](https://leetcode.com/problems/permutations-ii/)

Given a collection of numbers, nums, that might contain duplicates, return *all possible unique permutations****in any order****.*

**Example 1:**

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

**Output:**

[[1,1,2],

[1,2,1],

[2,1,1]]

**Example 2:**

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

**Output:** [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

**Constraints:**

* 1 <= nums.length <= 8
* -10 <= nums[i] <= 10
* class Solution {
* public List<List<Integer>> permuteUnique(int[] nums) {
* List<List<Integer>> res = new ArrayList<>();
* Arrays.sort(nums);
* backtrack(res, nums, new boolean[nums.length], new ArrayList<>());
* return res;
* }
* public void backtrack(List<List<Integer>> res, int[] nums, boolean[] visited,
* List<Integer> curr) {
* if (curr.size() == nums.length && !res.contains(curr)) {
* res.add(new ArrayList<>(curr));
* return;
* }
* for (int i = 0; i < nums.length; i++) {
* if (visited[i]) {
* continue;
* }
* visited[i] = true;
* curr.add(nums[i]);
* backtrack(res, nums, visited, curr);
* curr.remove(curr.size() - 1);
* visited[i] = false;
* }
* }
* }

[**289. Game of Life**](https://leetcode.com/problems/game-of-life/)

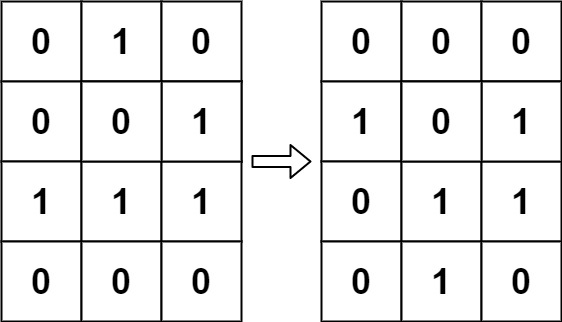
According to [Wikipedia's article](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life): "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an m x 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](https://en.wikipedia.org/wiki/Moore_neighborhood) (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 x n grid board, return *the next state*.

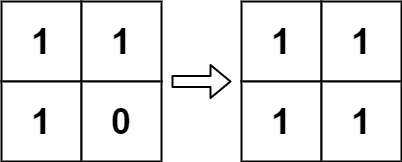
**Example 1:**



**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:**



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

**Output:** [[1,1],[1,1]]

**Constraints:**

* m == board.length
* n == board[i].length
* 1 <= m, n <= 25
* board[i][j] is 0 or 1.

**Follow up:**

* Could you solve it in-place? Remember that the board needs to be updated simultaneously: You cannot update some cells first and then use their updated values to update other cells.
* In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches upon the border of the array (i.e., live cells reach the border). How would you address these problems?
* class Solution {
* public void gameOfLife(int[][] board) {
* int m= board.length;
* int n= board[0].length;
* for(int i=0;i<m;i++){
* for(int j=0;j<n;j++){
* int cnt=0;
* if(i>0){
* if(board[i-1][j]%2==1){
* cnt++;
* }
* }
* if(j>0){
* if(board[i][j-1]%2==1){
* cnt++;
* }
* }
* if(i>0 && j>0){
* if(board[i-1][j-1]%2==1){
* cnt++;
* }
* }
* if(i<m-1){
* if(board[i+1][j]%2==1){
* cnt++;
* }
* }
* if(j<n-1){
* if(board[i][j+1]%2==1){
* cnt++;
* }
* }
* if(i<m-1 && j<n-1){
* if(board[i+1][j+1]%2==1){
* cnt++;
* }
* }
* if(i>0 && j<n-1){
* if(board[i-1][j+1]%2==1){
* cnt++;
* }
* }
* if(i<m-1 && j>0){
* if(board[i+1][j-1]%2==1){
* cnt++;
* }
* }
* if(board[i][j]%2==0 && cnt==3){
* board[i][j]+=2;
* }
* else if(board[i][j]%2==1 && (cnt==2 || cnt==3)){
* board[i][j]+=2;
* }
* else{
* board[i][j]+=0;
* }
* }
* }
* for(int i=0;i<m;i++){
* for(int j=0;j<n;j++){
* board[i][j]=board[i][j]/2;
* }
* }
* }
* }