LeetCode

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].

// var twoSum = function(nums, target) {

// for(let i=0; i<nums.length-1; i++){

// for(let j=i+1; j<nums.length; j++){

// if(nums[i]+nums[j]===target){

// return [i,j];

// }

// }

// }

// return;

// };

var twoSum = function(nums,target){

let comp = new Map();

let len = nums.length;

for(let i=0;i<len;i++){

if(comp[nums[i]] >=0){

return [comp[nums[i]],i]

}

comp[target - nums[i]] = i;

}

}

let nums = [2,7,11,15];

let target = 9;

console.log(twoSum(nums, target));

**26. Remove Duplicates from Sorted Array (two pointer method used)**

**https://www.youtube.com/watch?v=DEJAZBq0FDA**

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**.

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**](https://en.wikipedia.org/wiki/In-place_algorithm) 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**.

**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).

var removeDuplicates = function(nums) {

let left = 1;

let right = 1;

let len = nums.length;

while(right < len){

if(nums[right] != nums[right-1]){

nums[left] = nums[right];

left++;

}

right++;

}

return left;

console.log(nums)

};

**27. Remove Element**

Given an integer array nums and an integer val, remove all occurrences of val in nums [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm). The relative order of the elements may be changed.

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**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

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

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

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

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

var removeElement = function(nums, val) {

let left = 0;

let right = 0;

let len = nums.length;

while(right < len){

if(nums[right] != val){

nums[left] = nums[right];

left++;

}

right++;

}

console.log(nums)

return left;

};

**1512. Number of Good Pairs**

/\*

Given an array of integers nums, return the number of good pairs.

A pair (i, j) is called good if nums[i] == nums[j] and i < j.

Example 1:

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

Output: 4

Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.

\*/

var numIdenticalPairs = function(nums) {

let map = new Map();

let ans = 0;

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

if(map.has(nums[i])){

ans = ans + map.get(nums[i]);

map.set(nums[i], map.get(nums[i]) + 1);

}else {

map.set(nums[i],1);

}

}

return ans;

};

1365. How Many Numbers Are Smaller Than the Current Number

/\*

Given the array nums, for each nums[i] find out how many numbers in the array are smaller than it. That is, for each nums[i] you have to count the number of valid j's such that j != i and nums[j] < nums[i].

Return the answer in an array.

Example 1:

Input: nums = [8,1,2,2,3]

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

Explanation:

For nums[0]=8 there exist four smaller numbers than it (1, 2, 2 and 3).

For nums[1]=1 does not exist any smaller number than it.

For nums[2]=2 there exist one smaller number than it (1).

For nums[3]=2 there exist one smaller number than it (1).

For nums[4]=3 there exist three smaller numbers than it (1, 2 and 2).

with sorting method

\*/

var smallerNumbersThanCurrent = function(nums) {

    let map = new Map();

    let unsortedArr = [...nums];

    let ans=[];

    nums.sort((a,b)=> a-b);

    for(let i=0;i<nums.length;i++){

        // in this case check if item doesn't exist alrready.even if there are duplicate numbers from index of 1st occurence of ele, we can determin the count before it

        if(!map.has(nums[i])){

            map.set(nums[i], i);

        }

    }

    for(let i=0;i<unsortedArr.length;i++){

        ans.push(map.get(unsortedArr[i]))

    }

    return ans;

};

**167. Two Sum II - Input Array Is Sorted** with constant space (medium)

(two pointer approach)

Given a **1-indexed** array of integers numbers that is already **sorted in non-decreasing order**, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return the indices of the two numbers, index1 and index2, ***added by one*** as an integer array [index1, index2] of length 2.

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

**Example 1:**

**Input:** numbers = [2,7,11,15], target = 9

**Output:** [1,2]

**Explanation:** The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

**Example 2:**

**Input:** numbers = [2,3,4], target = 6

**Output:** [1,3]

**Explanation:** The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].

Point left pointer at 0 & right at last

If sum < target increase left

If sum > target decrease right

If sum==target return indexes.

var twoSum = function(numbers, target) {

let left = 0;

let right = numbers.length-1;

while(left<right){

let sum = numbers[left] + numbers[right];

if(sum < target){

left++;

} else if(sum > target){

right--;

}else if(sum == target){

return [left + 1,right + 1];

}

}

};

**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.

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 105

/\*\*

\* @param {number[]} nums

\* @return {boolean}

\*/

var canJump = function(nums) {

let maxReach = nums[0];

for(let i=1; i<nums.length;i++){

let jumps = i + nums[i];

if(jumps >= maxReach && maxReach >= i){

maxReach = jumps;

}

}

if(maxReach >= nums.length -1){

return true;

}else{

return false;

}

};

/\*

**53. Maximum Subarray Medium**

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

\*/

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var maxSubArray = function(nums) {

let maxSumSoFar = nums[0];

let currSum = nums[0];

for(let i=1;i<nums.length;i++){

if(currSum < 0){

currSum = 0;

}

currSum = currSum + nums[i];

if(currSum > maxSumSoFar){

maxSumSoFar = currSum;

}

}

return maxSumSoFar;

};