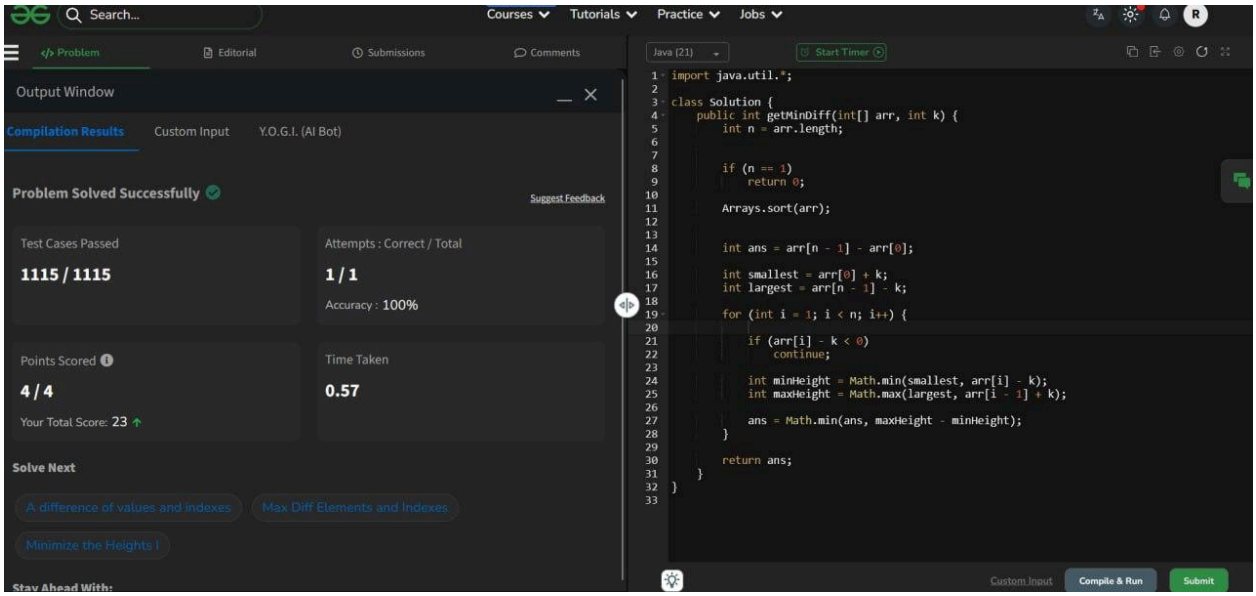
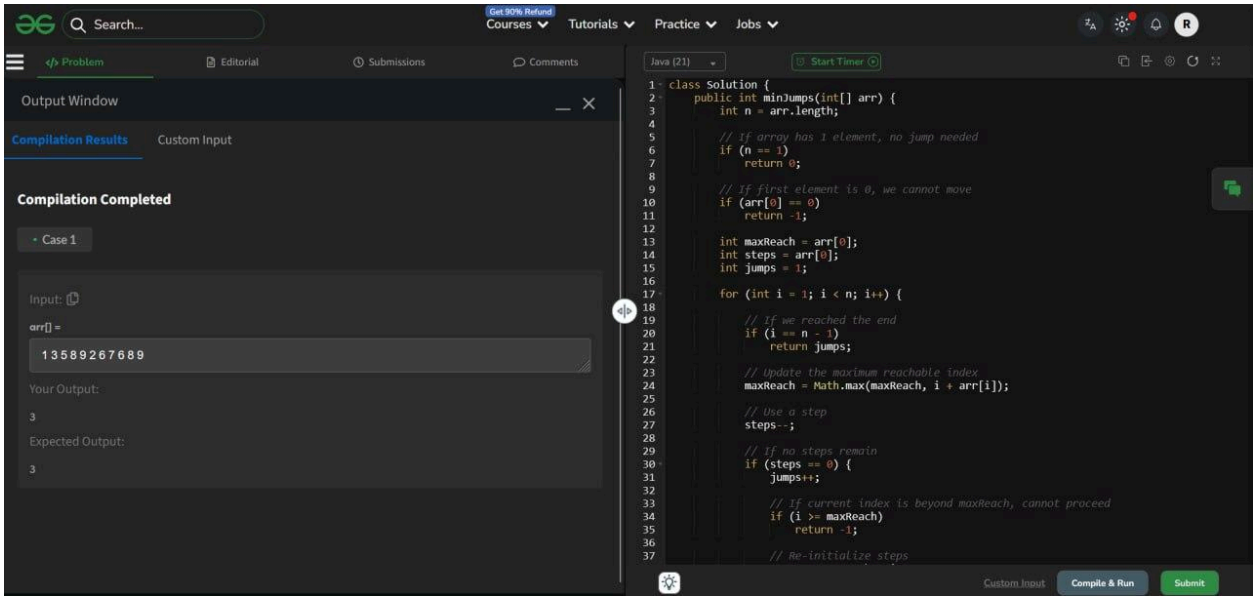
Given an array arr[] denoting heights of n towers and a positive integer k.
For each tower, you must perform exactly one of the following operations exactly once.
Increase the height of the tower by k
Decrease the height of the tower by k
Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.
You can find a slight modification of the problem here.
Note: It is compulsory to increase or decrease the height by k for each tower. After the operation,
the resultant array should not contain any negative integers.
Examples :
Input: k = 2, arr[] = [1, 5, 8, 10]
Output: 5
Explanation: The array can be modified as [1+k, 5-k, 8-k, 10-k] = [3, 3, 6, 8]. The difference between the largest and the smallest is 8-3 = 5.
Input: k = 3, arr[] = [3, 9, 12, 16, 20]
Output: 11
Explanation: The array can be modified as [3+k, 9+k, 12-k, 16-k, 20-k] = [6, 12, 9, 13, 17]. The difference between the largest and the smallest is 17-6 = 11.
Constraints
1 ≤ k ≤ 107
1 ≤ n ≤ 105
1 ≤ arr[i] ≤ 107



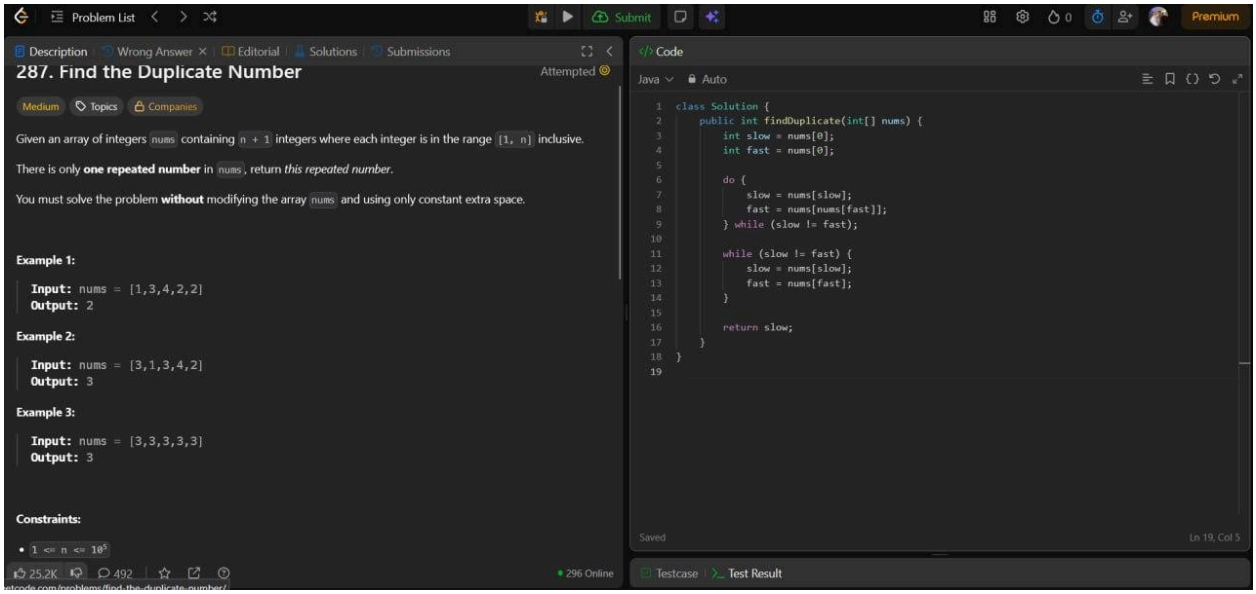
You are given an array arr[] of non-negative numbers. Each number tells you the maximum number of steps you can jump forward from that position.
For example:
If arr[i] = 3, you can jump to index i + 1, i + 2, or i + 3 from position i.
If arr[i] = 0, you cannot jump forward from that position.
Your task is to find the minimum number of jumps needed to move from the first position in the array to the last position.
Note: Return -1 if you can't reach the end of the array.
Examples :
Input: arr[] = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]
Output: 3
Explanation: First jump from 1st element to 2nd element with value 3. From here we jump to 5th element with value 9, and from here we will jump to the last.
Input: arr = [1, 4, 3, 2, 6, 7]
Output: 2
Explanation: First we jump from the 1st to 2nd element and then jump to the last element.
Input: arr = [0, 10, 20]
Output: -1
Explanation: We cannot go anywhere from the 1st element.
Constraints:
2 ≤ arr.size() ≤ 105
0 ≤ arr[i] ≤ 105

✅ Core Idea (Greedy)
While moving through the array, we track:
maxReach → farthest index we can reach so far
steps → steps remaining in the current jump
jumps → total jumps taken
We only increase jumps when steps become 0.


Search...
Get 90% Refund
Courses
Tutorials
Practice
Jobs
Problem
Editorial
Submissions
Comments
Output Window
Compilation Results
Custom Input
Compilation Completed
Case 1
Input:
arr[] =
13589267689
Your Output:
3
Expected Output:
3
Java (21)
Start Timer
class Solution {
public int minJumps(int[] arr) {
int n = arr.length;
// If array has 1 element, no jump needed
if (n == 1)
return 0;
// If first element is 0, we cannot move
if (arr[0] == 0)
return -1;
int maxReach = arr[0];
int steps = arr[0];
int jumps = 1;
for (int i = 1; i < n; i++) {
// If we reached the end
if (i == n - 1)
return jumps;
// Update the maximum reachable index
maxReach = Math.max(maxReach, i + arr[i]);
// Use a step
steps--;
// If no steps remain
if (steps == 0) {
jumps++;
// If current index is beyond maxReach, cannot proceed
if (i >= maxReach)
return -1;
// Re-initialize steps
Custom Input
Compile & Run
Submit

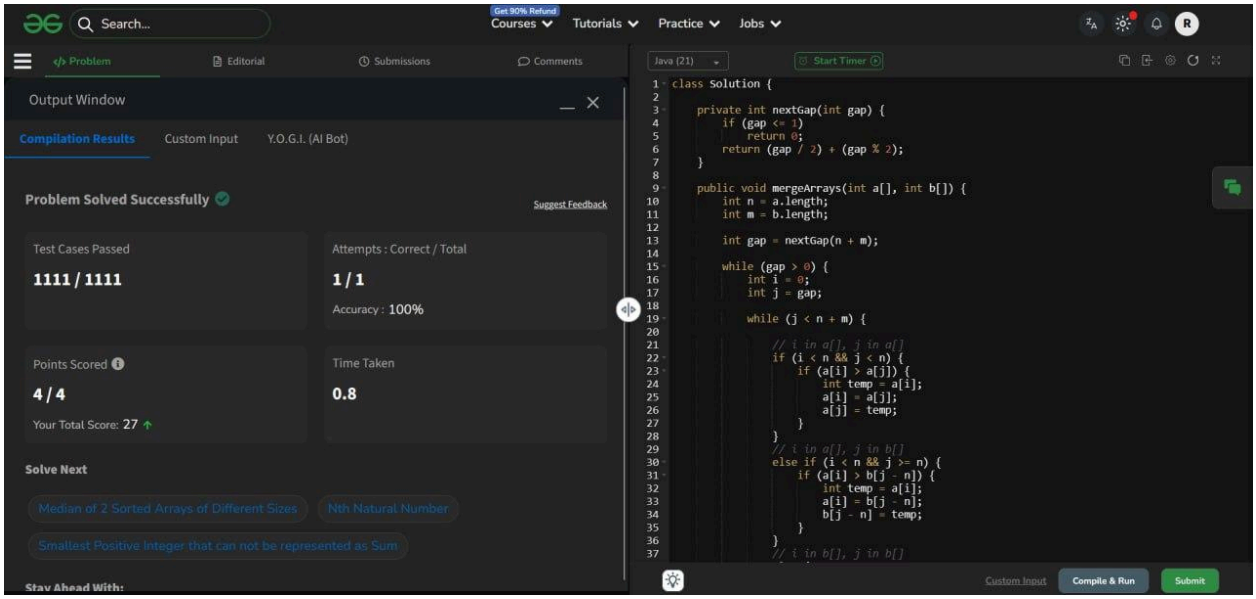
Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n]
inclusive.
There is only one repeated number in nums, return this repeated number.
You must solve the problem without modifying the array nums and using only constant extra space.
Example 1:
Input: nums = [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.

Algorithm (Floyd's Cycle Detection)
Use two pointers:
slow → moves 1 step
fast → moves 2 steps
First phase: find the meeting point
Second phase: move one pointer to start
Move both 1 step at a time
Where they meet again = duplicate number


Problem List
Submit
Premium
Description
Wrong Answer
Editorial
Solutions
Submissions
287. Find the Duplicate Number
Attempted
Medium
Topics
Companies
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 using 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 <= 10^5
25.2K
492
296 Online
Code
Java
Auto
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);
while (slow != fast) {
slow = nums[slow];
fast = nums[fast];
}
return slow;
}
}
Saved
Ln 19, Col 5
Testcase
Test Result

Given two sorted arrays a[] and b[] of size n and m respectively, the task is to merge them in sorted order without using any extra space. Modify a[] so that it contains the first n elements and modify b[] so that it contains the last m elements.
Examples:
Input: a[] = [2, 4, 7, 10], b[] = [2, 3]
Output: a[] = [2, 2, 3, 4], b[] = [7, 10]
Explanation: After merging the two non-decreasing arrays, we get, [2, 2, 3, 4, 7, 10]
Input: a[] = [1, 5, 9, 10, 15, 20], b[] = [2, 3, 8, 13]
Output: a[] = [1, 2, 3, 5, 8, 9], b[] = [10, 13, 15, 20]
Explanation: After merging two sorted arrays we get [1, 2, 3, 5, 8, 9, 10, 13, 15, 20].
Input: a[] = [0, 1], b[] = [2, 3]
Output: a[] = [0, 1], b[] = [2, 3]
Explanation: After merging two sorted arrays we get [0, 1, 2, 3].
Constraints:
1 ≤ n, m ≤ 105
0 ≤ a[i], b[i] ≤ 107

Core Idea (Gap Method)
Treat both arrays as a single combined array
Start with a gap = ceil((n + m) / 2)
Compare elements that are gap apart and swap if needed
Reduce the gap until it becomes 0
This avoids using any extra array.



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] overlap, 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.
Example 3:
Input: intervals = [[4,7],[1,4]]
Output: [[1,7]]
Explanation: Intervals [1,4] and [4,7] are considered overlapping.
Constraints:
1 <= intervals.length <= 104
intervals[i].length == 2
0 <= starti <= endi <= 104

Approach
Sort intervals by start time
Initialize a result list
Traverse intervals:
If current interval overlaps with last interval in result → merge them
Else → add as a new interval
Two intervals [a,b] and [c,d] overlap if c ≤ b