

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

$1 \leq k \leq 107$

$1 \leq n \leq 105$

$1 \leq arr[i] \leq 107$

The screenshot shows a LeetCode problem submission page. The problem title is "Minimize the Height Difference". The submission status is "Problem Solved Successfully" with a green checkmark. The code editor contains the following Java code:

```
1 import java.util.*;
2
3 class Solution {
4     public int getMinDiff(int[] arr, int k) {
5         int n = arr.length;
6
7         if (n == 1)
8             return 0;
9
10        Arrays.sort(arr);
11
12
13        int ans = arr[n - 1] - arr[0];
14
15        int smallest = arr[0] + k;
16        int largest = arr[n - 1] - k;
17
18        for (int i = 1; i < n; i++) {
19            if (arr[i] - k < 0)
20                continue;
21
22            int minHeight = Math.min(smallest, arr[i] - k);
23            int maxHeight = Math.max(largest, arr[i - 1] + k);
24
25            ans = Math.min(ans, maxHeight - minHeight);
26
27        }
28
29        return ans;
30    }
31
32 }
```

The performance metrics shown are: Test Cases Passed (1115 / 1115), Attempts: Correct / Total (1 / 1), Accuracy: 100%, Points Scored (4 / 4), Time Taken (0.57), and Your Total Score: 23.

You are given an array $\text{arr}[]$ of non-negative numbers. Each number tells you the maximum number of steps you can jump forward from that position.

For example:

If $\text{arr}[i] = 3$, you can jump to index $i + 1$, $i + 2$, or $i + 3$ from position i .

If $\text{arr}[i] = 0$, 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: $\text{arr}[] = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]$

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: $\text{arr} = [1, 4, 3, 2, 6, 7]$

Output: 2

Explanation: First we jump from the 1st to 2nd element and then jump to the last element.

Input: $\text{arr} = [0, 10, 20]$

Output: -1

Explanation: We cannot go anywhere from the 1st element.

Constraints:

$2 \leq \text{arr.size()} \leq 105$

$0 \leq \text{arr}[i] \leq 105$

Core Idea (Greedy)

While moving through the array, we track:

$\text{maxReach} \rightarrow$ farthest index we can reach so far

$\text{steps} \rightarrow$ steps remaining in the current jump

$\text{jumps} \rightarrow$ 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[] =
1 3 5 8 9 2 6 7 6 8 9

Your Output: 3

Expected Output: 3

Java (21) Start Timer

```
1- class Solution {
2-     public int minJumps(int[] arr) {
3-         int n = arr.length;
4-
5-         // If array has 1 element, no jump needed
6-         if (n == 1)
7-             return 0;
8-
9-         // If first element is 0, we cannot move
10-        if (arr[0] == 0)
11-            return -1;
12-
13-        int maxReach = arr[0];
14-        int steps = arr[0];
15-        int jumps = 1;
16-
17-        for (int i = 1; i < n; i++) {
18-            // If we reached the end
19-            if (i == n - 1)
20-                return jumps;
21-
22-            // Update the maximum reachable index
23-            maxReach = Math.max(maxReach, i + arr[i]);
24-
25-            // Use a step
26-            steps--;
27-
28-            // If no steps remain
29-            if (steps == 0) {
30-                jumps++;
31-
32-                // If current index is beyond maxReach, cannot proceed
33-                if (i >= maxReach)
34-                    return -1;
35-
36-                // Re-initialize steps
37-            }
}
}
```

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:

$1 \leq n \leq 105$

`nums.length == n + 1`

$1 \leq \text{nums}[i] \leq n$

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 < > ✎ Problem List

Description Wrong Answer ✗ Editorial Solutions Submissions Attempted

287. Find the Duplicate Number

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:

- $1 \leq n \leq 10^5$

25.2K 492 | Testcase Test Result

```
Java Auto
1 class Solution {
2     public int findDuplicate(int[] nums) {
3         int slow = nums[0];
4         int fast = nums[0];
5
6         do {
7             slow = nums[slow];
8             fast = nums[nums[fast]];
9         } while (slow != fast);
10
11         while (slow != fast) {
12             slow = nums[slow];
13             fast = nums[fast];
14         }
15
16         return slow;
17     }
18 }
19
```

Saved Ln 19, Col 5

<https://leetcode.com/problems/find-the-duplicate-number/>

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:

$1 \leq n, m \leq 105$

$0 \leq a[i], b[i] \leq 107$

Core Idea (Gap Method)

Treat both arrays as a single combined array

Start with a gap = $\text{ceil}((n + m) / 2)$

Compare elements that are gap apart and swap if needed

Reduce the gap until it becomes 0

This avoids using any extra array.

The screenshot shows a LeetCode problem page for "Merge Sorted Array". The code editor contains the following Java solution:

```
1 class Solution {
2     private int nextGap(int gap) {
3         if (gap <= 1)
4             return 0;
5         return (gap / 2) + (gap % 2);
6     }
7
8     public void mergeArrays(int a[], int b[]) {
9         int n = a.length;
10        int m = b.length;
11
12        int gap = nextGap(n + m);
13
14        while (gap > 0) {
15            int i = 0;
16            int j = gap;
17
18            while (j < n + m) {
19
20                // i in a[], j in a[]
21                if (i < n && j < n) {
22                    if (a[i] > a[j]) {
23                        int temp = a[i];
24                        a[i] = a[j];
25                        a[j] = temp;
26                    }
27                }
28                // i in a[], j in b[]
29                else if (i < n && j >= n) {
30                    if (a[i] > b[j - n]) {
31                        int temp = a[i];
32                        a[i] = b[j - n];
33                        b[j - n] = temp;
34                    }
35                }
36            }
37        }
38    }
39}
```

The sidebar displays the following information:

- Output Window
- Compilation Results: Problem Solved Successfully
- Test Cases Passed: 1111 / 1111
- Attempts: Correct / Total: 1 / 1
- Accuracy: 100%
- Points Scored: 4 / 4
- Time Taken: 0.8
- Your Total Score: 27
- Solve Next: Median of 2 Sorted Arrays of Different Sizes, Nth Natural Number, Smallest Positive Integer that can not be represented as Sum
- Stay Ahead With: Custom Input, Compile & Run, Submit

Given an array of intervals where $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$, merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example 1:

Input: $\text{intervals} = [[1,3],[2,6],[8,10],[15,18]]$

Output: $[[1,6],[8,10],[15,18]]$

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Example 2:

Input: $\text{intervals} = [[1,4],[4,5]]$

Output: $[[1,5]]$

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

Example 3:

Input: $\text{intervals} = [[4,7],[1,4]]$

Output: $[[1,7]]$

Explanation: Intervals [1,4] and [4,7] are considered overlapping.

Constraints:

$1 \leq \text{intervals.length} \leq 104$

$\text{intervals}[i].length == 2$

$0 \leq \text{start}_i \leq \text{end}_i \leq 104$

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 $c \leq b$

The screenshot shows a code editor interface with a navigation bar at the top. The main area is divided into two panes: 'Code' on the right and 'Runtime' and 'Memory' metrics on the left. The 'Code' pane displays a Java class named 'Solution' with a single method 'merge'. The 'Runtime' pane shows a bar chart with a peak at 9ms, labeled 'Beats 38.70%'. The 'Memory' pane shows a bar chart with a peak at 49.15 MB, labeled 'Beats 50.97%'. Below the code editor, there is a smaller preview window showing the same code.

```
1 class Solution {
2     public int[][] merge(int[][] intervals) {
3         Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
4
5         List<int[]> result = new ArrayList<>();
6
7         // Add first interval
8         result.add(intervals[0]);
9
10        for (int i = 1; i < intervals.length; i++) {
11            int[] last = result.get(result.size() - 1);
12            int[] current = intervals[i];
13
14            // Overlapping intervals
15            if (current[0] <= last[1]) {
16                last[1] = Math.max(last[1], current[1]);
17            } else {
18                result.add(current);
19            }
20        }
21
22        // Convert List to array
23        return result.toArray(new int[result.size()][]);
24    }
25 }
```

Code | Java

```
1 class Solution {
2     public int[][] merge(int[][] intervals) {
3         Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
4
5         List<int[]> result = new ArrayList<>();
6
7         for (int i = 1; i < intervals.length; i++) {
8             int[] last = result.get(result.size() - 1);
9             int[] current = intervals[i];
10
11            if (current[0] <= last[1]) {
12                last[1] = Math.max(last[1], current[1]);
13            } else {
14                result.add(current);
15            }
16        }
17
18        return result.toArray(new int[result.size()][]);
19    }
20 }
```

Accepted 172 / 172 testcases passed
duefoxx submitted at Feb 02, 2026 22:47

Editorial Solution

Runtime: 9 ms | Beats 38.70%
Memory: 49.15 MB | Beats 50.97%

Code | Java

Testcase | Test Result