

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 coding platform interface. On the left, the 'Output Window' displays 'Problem Solved Successfully' with a green checkmark. Below this, it shows 'Test Cases Passed: 1115 / 1115', 'Attempts: Correct / Total: 1 / 1', 'Accuracy: 100%', 'Points Scored: 4 / 4', and 'Time Taken: 0.57'. At the bottom, there are buttons for 'Solve Next' and 'Stay Ahead With:'. On the right, the 'Editorial' tab is active, showing a Java solution. The code is as follows:

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

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:

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

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

✓ 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.

36

Search...

Get 90% Refund

Courses

Tutorials

Practice

Jobs

7A

R

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

Custom Input

Compile & Run

Submit

```
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
19            // If we reached the end
20            if (i == n - 1)
21                return jumps;
22
23            // Update the maximum reachable index
24            maxReach = Math.max(maxReach, i + arr[i]);
25
26            // Use a step
27            steps--;
28
29            // If no steps remain
30            if (steps == 0) {
31                jumps++;
32
33                // If current index is beyond maxReach, cannot proceed
34                if (i >= maxReach)
35                    return -1;
36
37                // Re-initialize steps
```

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:

$1 \leq n \leq 10^5$

`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 \rightarrow moves 1 step

fast \rightarrow 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

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:

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

25.2K 492 29% Online

Code

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 In 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:

$1 \leq n, m \leq 105$

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

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 coding platform interface with a dark theme. On the left, the 'Output Window' displays 'Problem Solved Successfully' with statistics: 11/11 test cases passed, 1/1 attempts, 100% accuracy, 4/4 points scored, and a time taken of 0.8 seconds. The main editor on the right shows a Java solution for the 'Merge Two Sorted Arrays' problem. The code implements the Gap Method, starting with a gap of $\text{ceil}((n+m)/2)$ and iteratively swapping elements that are gap apart until the gap becomes 0. The solution is enclosed in a class named 'Solution' with a 'mergeArrays' method.

```
1 class Solution {
2
3     private int nextGap(int gap) {
4         if (gap <= 1)
5             return 0;
6         return (gap / 2) + (gap % 2);
7     }
8
9     public void mergeArrays(int a[], int b[]) {
10        int n = a.length;
11        int m = b.length;
12
13        int gap = nextGap(n + m);
14
15        while (gap > 0) {
16            int i = 0;
17            int j = gap;
18
19            while (j < n + m) {
20
21                // i in a[], j in a[]
22                if (i < n && j < n) {
23                    if (a[i] > a[j]) {
24                        int temp = a[i];
25                        a[i] = a[j];
26                        a[j] = temp;
27                    }
28                }
29
30                // i in a[], j in b[]
31                else if (i < n && j >= n) {
32                    if (a[i] > b[j - n]) {
33                        int temp = a[i];
34                        a[i] = b[j - n];
35                        b[j - n] = temp;
36                    }
37                }
38
39                // i in b[], j in b[]
40            }
41
42            gap = nextGap(gap);
43        }
44    }
45 }
```

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:

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

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

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

Approach

Sort intervals by start time

Initialize a result list

Traverse intervals:

If current interval overlaps with last interval in result \rightarrow merge them

Else \rightarrow add as a new interval

Two intervals $[a,b]$ and $[c,d]$ overlap if $c \leq b$

The screenshot shows the LeetCode submission interface for the 'Merge Intervals' problem. The submission is 'Accepted' with 172/172 test cases passed. The runtime is 9 ms, which beats 38.70% of submissions, and the memory usage is 49.15 MB, which beats 50.97%. A bar chart shows the runtime distribution. The code editor displays a Java solution that sorts intervals by start time and merges overlapping ones.

```
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

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

        // Add first interval
        result.add(intervals[0]);

        for (int i = 1; i < intervals.length; i++) {
            int[] last = result.get(result.size() - 1);
            int[] current = intervals[i];

            // Overlapping intervals
            if (current[0] <= last[1]) {
                last[1] = Math.max(last[1], current[1]);
            } else {
                result.add(current);
            }
        }

        // Convert List to array
        return result.toArray(new int[result.size()][]);
    }
}
```