Triplet Sum in Array

Difficulty: **Medium** Accuracy: **35.0%** Submissions: **362K+** Points: **4** Average Time: **15m**

Given an array **arr[]** and an integer **target**, determine if there exists a triplet in the array whose sum equals the given **target**.

Return **true** if such a triplet exists, otherwise, return **false**.

Examples:

Input: arr[] = [1, 4, 45, 6, 10, 8], target = 13
Output: true
Explanation: The triplet {1, 4, 8} sums up to 13.

Input: arr[] = [1, 2, 4, 3, 6, 7], target = 10
Output: true
Explanation: The triplets {1, 3, 6} and {1, 2, 7} both sum to 10.

Input: arr[] = [40, 20, 10, 3, 6, 7], target = 24
Output: false
Explanation: No triplet in the array sums to 24.

Constraints:

$3 \leq arr.size() \leq 5*10^3$

$0 \leq arr[i], target \leq 10^5$

```java
class Solution {
    public static boolean find3Numbers(int[] arr, int target) {
        int n = arr.length;
        Arrays.sort(arr);

        for (int i = 0; i < n - 2; i++) {
            int left = i + 1;
            int right = n - 1;

            while (left < right) {
                int sum = arr[i] + arr[left] + arr[right];

                if (sum == target) {
                    return true;
                } else if (sum < target) {
                    left++;
                } else {
                    right--;
                }
            }
        }
        return false;
    }

    // For testing
    public static void main(String[] args) {
        int[] arr = {1, 4, 45, 6, 10, 8};
        int target = 13;
        System.out.println(find3Numbers(arr, target)); // true
    }
}
```

Problem | Editorial | Submissions | Comments

Array Subset

Difficulty: **Basic** Accuracy: **44.05%** Submissions: **521K+** Points: **1** Average Time: **20m**

Given two arrays **a[]** and **b[]**, your task is to determine whether **b[]** is a subset of **a[]**.

Examples:

Input: a[] = [11, 7, 1, 13, 21, 3, 7, 3], b[] = [11, 3, 7, 1, 7]
Output: true
Explanation: b[] is a subset of a[]

Input: a[] = [1, 2, 3, 4, 4, 5, 6], b[] = [1, 2, 4]
Output: true
Explanation: b[] is a subset of a[]

Input: a[] = [10, 5, 2, 23, 19], b[] = [19, 5, 3]
Output: false
Explanation: b[] is not a subset of a[]

Constraints:

$1 <= a.size(), b.size() <= 10^5$

$1 <= a[i], b[j] <= 10^6$

Java (21) Start Timer

```java
class Solution {
    public boolean isSubset(int[] a, int[] b) {
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int x : a)
            map.put(x, map.getOrDefault(x, 0) + 1);

        for (int x : b) {
            if (!map.containsKey(x) || map.get(x) == 0)
                return false;
            map.put(x, map.get(x) - 1);
        }
        return true;
    }
}
```

Factorials of large numbers

Difficulty: **Medium** Accuracy: **36.57%** Submissions: **177K+** Points: **4** Average Time: **20m**

Given an integer **n,** find its factorial. Return a list of integers denoting the digits that make up the factorial of n.

Examples:

Input: n = 5
Output: [1, 2, 0]
Explanation: 5! = 1*2*3*4*5 = 120

Input: n = 10
Output: [3, 6, 2, 8, 8, 0, 0]
Explanation: 10! = 1*2*3*4*5*6*7*8*9*10 = 3628800

Input: n = 1
Output: [1]
Explanation: 1! = 1

Constraints:
$1 \leq n \leq 10^3$

Java (21) Start Timer

```java
// User function Template for Java

class Solution {
    static ArrayList<Integer> factorial(int n) {
        ArrayList<Integer> res = new ArrayList<>();
        res.add(1);

        for (int x = 2; x <= n; x++) {
            int carry = 0;
            for (int i = 0; i < res.size(); i++) {
                int val = res.get(i) * x + carry;
                res.set(i, val % 10);
                carry = val / 10;
            }
            while (carry > 0) {
                res.add(carry % 10);
                carry /= 10;
            }
        }

        Collections.reverse(res);
        return res;
    }
}
```

Problem | Editorial | Submissions | Comments

Common in 3 Sorted Arrays

Difficulty: **Easy** Accuracy: **22.16%** Submissions: **440K+** Points: **2**

Given three sorted arrays in **non-decreasing** order, print all common elements in **non-decreasing** order across these arrays. If there are no such elements return an empty array. In this case, the output will be -1.

Note: can you handle the duplicates without using any additional Data Structure?

Examples :

Input: arr1 = [1, 5, 10, 20, 40, 80] , arr2 = [6, 7, 20, 80, 100] , arr3 = [3, 4, 15, 20, 30, 70, 80, 120]
Output: [20, 80]
Explanation: 20 and 80 are the only common elements in arr1, arr2 and arr3.

Input: arr1 = [1, 2, 3, 4, 5] , arr2 = [6, 7] , arr3 = [8,9,10]
Output: [-1]
Explanation: There are no common elements in arr1, arr2 and arr3.

Input: arr1 = [1, 1, 1, 2, 2, 2], arr2 = [1, 1, 2, 2, 2], arr3 = [1, 1, 1, 1, 2, 2, 2, 2]
Output: [1, 2]
Explanation: We do not need to consider duplicates

Java (21) Start Timer

```java
// User function Template for Java

class Solution {
    // Function to find common elements in three arrays.
    public List<Integer> commonElements(List<Integer> arr1, List<Integer> arr2,
                                        List<Integer> arr3) {
        int i = 0, j = 0, k = 0;
        List<Integer> ans = new ArrayList<>();

        while (i < arr1.size() && j < arr2.size() && k < arr3.size()) {

            int a = arr1.get(i);
            int b = arr2.get(j);
            int c = arr3.get(k);

            if (a == b && b == c) {
                if (ans.isEmpty() || ans.get(ans.size() - 1) != a)
                    ans.add(a);
                i++; j++; k++;
            }
            else if (a < b) i++;
            else if (b < c) j++;
            else k++;
        }

        if (ans.isEmpty()) ans.add(-1);
        return ans;
    }
}
```

Problem List | Submit | Premium

Description | Editorial | Solutions | Submissions

56. Merge Intervals

Solved

Medium | Topics | Companies

Given an array of `intervals` where `intervals[i] = [start_i, 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: 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]]
```

Code

Java | Auto

```java
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a,b) -> a[0] - b[0]);
        List<int[]> res = new ArrayList<>();

        for (int[] i : intervals) {
            if (res.isEmpty() || res.get(res.size()-1)[1] < i[0])
                res.add(i);
            else
                res.get(res.size()-1)[1] = Math.max(res.get(res.size()-1)[1], i[1]);
        }
        return res.toArray(new int[res.size()][]);
    }
}
```

Saved

Ln 1, Col 1

Problem | Editorial | Submissions | Comments

Merge Without Extra Space

Difficulty: **Medium** Accuracy: **32.01%** Submissions: **327K+** Points: **4** Average Time: **20m**

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:

$1 \le n, m \le 10^5$

$0 \le a[i], b[i] \le 10^7$

Java (21) | Start Timer

```java
class Solution {
    public void mergeArrays(int[] a, int[] b) {
        int n = a.length, m = b.length;
        for (int gap = nextGap(n + m); gap > 0; gap = nextGap(gap)) {
            for (int i = 0, j = gap; j < n + m; i++, j++) {
                if (i < n && j < n && a[i] > a[j]) swap(a, i, a, j);
                else if (i < n && j >= n && a[i] > b[j - n]) swap(a, i, b,
                else if (i >= n && b[i - n] > b[j - n]) swap(b, i - n, b, j
            }
        }
    }

    int nextGap(int g) {
        return g <= 1 ? 0 : (g / 2) + (g % 2);
    }

    void swap(int[] x, int i, int[] y, int j) {
        int t = x[i]; x[i] = y[j]; y[j] = t;
    }
}
```

Problem List | Submit | Premium

Description | Editorial | Solutions | Submissions

287. Find the Duplicate Number

Solved

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]
```

Code

Java | Auto

```java
class Solution {
    public int findDuplicate(int[] nums) {
        int slow = nums[0];
        int fast = nums[0];

        // Phase 1: Detect cycle
        do {
            slow = nums[slow];
            fast = nums[nums[fast]];
        } while (slow != fast);

        // Phase 2: Find cycle entrance
        slow = nums[0];
        while (slow != fast) {
            slow = nums[slow];
            fast = nums[fast];
        }

        return slow;
    }
}
```

Saved

Ln 1, Col 1

Problem | Editorial | Submissions | Comments

Minimum Jumps

Difficulty: **Medium** Accuracy: **11.91%** Submissions: **1.1M** Points: **4**

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.

Java (21) | Start Timer

```java
class Solution {
    public int minJumps(int[] arr) {
        int n = arr.length;
        if (n <= 1) return 0;
        if (arr[0] == 0) return -1;

        int maxReach = arr[0], steps = arr[0], jumps = 1;

        for (int i = 1; i < n; i++) {
            if (i == n - 1) return jumps;

            maxReach = Math.max(maxReach, i + arr[i]);
            steps--;

            if (steps == 0) {
                jumps++;
                if (i >= maxReach) return -1;
                steps = maxReach - i;
            }
        }
        return -1;
    }
}
```

Custom Input | Compile & Run | Submit

Minimize the Heights II

Difficulty: **Medium** Accuracy: **15.06%** Submissions: **772K+** Points: **4** Average Time: **25m**

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.

Java (21) Start Timer

```java
class Solution {
    public int getMinDiff(int[] arr, int k) {
        int n = arr.length;
        if (n == 1) return 0;

        Arrays.sort(arr);

        int ans = arr[n - 1] - arr[0];

        int small = arr[0] + k;
        int big = arr[n - 1] - k;

        if (small > big) {
            int temp = small;
            small = big;
            big = temp;
        }

        for (int i = 1; i < n - 1; i++) {
            int sub = arr[i] - k;
            int add = arr[i] + k;

            if (sub < 0) continue;

            if (sub >= small || add <= big) continue;

            if (big - sub <= add - small)
                small = sub;
            else
                big = add;
        }

        return Math.min(ans, big - small);
```

Custom Input Compile & Run Submit

Problem | Editorial | Submissions | Comments

Kth Smallest

Difficulty: **Medium** Accuracy: **35.17%** Submissions: **739K+** Points: **4** Average Time: **25m**

Given an integer array **arr[]** and an integer **k**, your task is to find and return the **k^{th} smallest** element in the given array.

Note: The kth smallest element is determined based on the sorted order of the array.

Examples :

Input: arr[] = [10, 5, 4, 3, 48, 6, 2, 33, 53, 10], k = 4
Output: 5
Explanation: 4th smallest element in the given array is 5.

Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3
Output: 7
Explanation: 3rd smallest element in the given array is 7.

Constraints:

$1 \leq arr.size() \leq 10^5$

$1 \leq arr[i] \leq 10^5$

$1 \leq k \leq arr.size()$

Java (21) Start Timer

```java
class Solution {
    public int kthSmallest(int[] arr, int k) {
        Arrays.sort(arr);
        return arr[k - 1];
    }
}
```

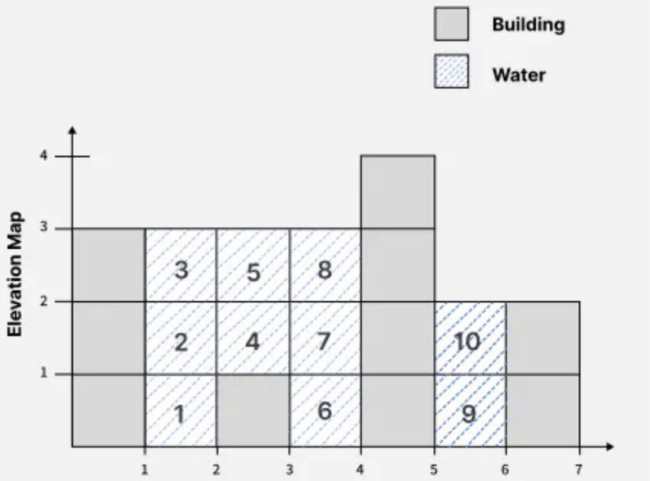
Trapping Rain Water

Difficulty: **Hard** Accuracy: **33.14%** Submissions: **498K+** Points: **8** Average Time: **20m**

Given an array **arr[]** with non-negative integers representing the height of blocks. If the width of each block is 1, compute how much water can be trapped between the blocks during the rainy season.

Examples:

Input: arr[] = [3, 0, 1, 0, 4, 0 2]
Output: 10
Explanation: Total water trapped = 0 + 3 + 2 + 3 + 0 + 2 + 0 = 10 units.



Java (21)
Start Timer

```java
class Solution {
    public int maxWater(int arr[]) {
        int n = arr.length;
        int left = 0, right = n - 1;
        int leftMax = 0, rightMax = 0;
        int water = 0;

        while (left < right) {
            if (arr[left] <= arr[right]) {
                if (arr[left] >= leftMax) {
                    leftMax = arr[left];
                } else {
                    water += leftMax - arr[left];
                }
                left++;
            } else {
                if (arr[right] >= rightMax) {
                    rightMax = arr[right];
                } else {
                    water += rightMax - arr[right];
                }
                right--;
            }
        }
        return water;
    }
}
```

Custom Input
Compile & Run
Submit