Harshitha R AIDS A 22AD041 20/11/24

1. **3Sum Closest**

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Example 2:**

**Input:** nums = [0,0,0], target = 1

**Output:** 0

**Explanation:** The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

**Constraints:**

* 3 <= nums.length <= 500
* -1000 <= nums[i] <= 1000
* -104 <= target <= 104

**Code**

class Solution {

public int threeSumClosest(int[] nums, int target) {

Arrays.sort(nums);

int closest\_sum = Integer.MAX\_VALUE / 2;

for (int i = 0; i < nums.length - 2; ++i) {

int left = i + 1, right = nums.length - 1;

while (left < right) {

int current\_sum = nums[i] + nums[left] + nums[right];

if (Math.abs(current\_sum - target) < Math.abs(closest\_sum - target)) {

closest\_sum = current\_sum;

}

if (current\_sum < target) {

++left;

} else if (current\_sum > target) {

--right;

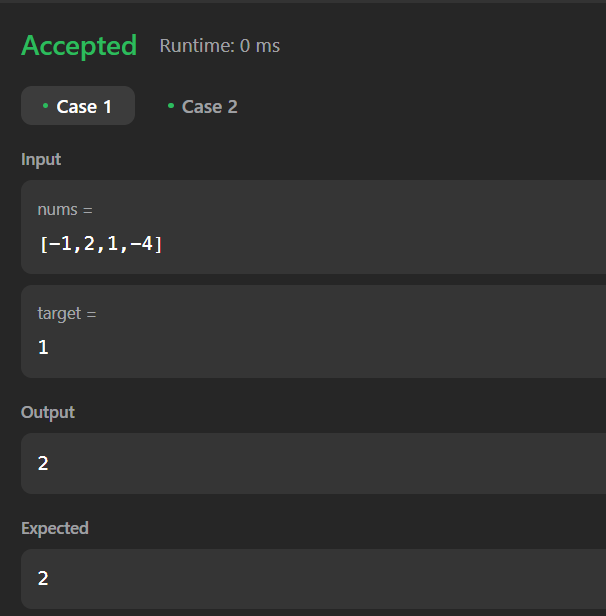
} else {

return current\_sum;

}}}

return closest\_sum;

}}

**Output**

**Time Complexity**

O(n^2)

1. **Jump Game II**

You are given a **0-indexed** array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

* 0 <= j <= nums[i] and
* i + j < n

Return *the minimum number of jumps to reach* nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

**Example 1:**

**Input:** nums = [2,3,1,1,4]

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** nums = [2,3,0,1,4]

**Output:** 2

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 1000
* It's guaranteed that you can reach nums[n - 1].

**Code**

**class Solution {**

**public int jump(int[] nums) {**

**int near = 0, far = 0, jumps = 0;**

**while (far < nums.length - 1) {**

**int farthest = 0;**

**for (int i = near; i <= far; i++) {**

**farthest = Math.max(farthest, i + nums[i]);**

**}**

**near = far + 1;**

**far = farthest;**

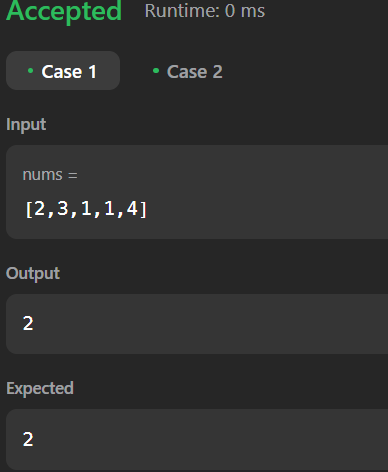
**jumps++;**

**}**

**return jumps;**

**}}**

**Output**



**Time Complexity**

O(n)

1. **Group Anagrams**

Given an array of strings strs, group the

anagrams

together. You can return the answer in **any order**.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Explanation:**

* There is no string in strs that can be rearranged to form "bat".
* The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
* The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

**Code**

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

Map<String, List<String>> map = new HashMap<>();

for (String word : strs) {

char[] chars = word.toCharArray();

Arrays.sort(chars);

String sortedWord = new String(chars);

if (!map.containsKey(sortedWord)) {

map.put(sortedWord, new ArrayList<>());

}

map.get(sortedWord).add(word);

}

return new ArrayList<>(map.values());

}

}

**Output**

**Time Complexity**

O(n log n)

1. **Decode Ways**

You have intercepted a secret message encoded as a string of numbers. The message is **decoded** via the following mapping:

"1" -> 'A'

"2" -> 'B'

...

"25" -> 'Y'

"26" -> 'Z'

However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").

For example, "11106" can be decoded into:

* "AAJF" with the grouping (1, 1, 10, 6)
* "KJF" with the grouping (11, 10, 6)
* The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode.  
  
Given a string s containing only digits, return the **number of ways** to **decode** it. If the entire string cannot be decoded in any valid way, return 0.

The test cases are generated so that the answer fits in a **32-bit** integer.

**Example 1:**

**Input:** s = "12"

**Output:** 2

**Explanation:**

"12" could be decoded as "AB" (1 2) or "L" (12).

**Example 2:**

**Input:** s = "226"

**Output:** 3

**Explanation:**

"226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

**Example 3:**

**Input:** s = "06"

**Output:** 0

**Explanation:**

"06" cannot be mapped to "F" because of the leading zero ("6" is different from "06"). In this case, the string is not a valid encoding, so return 0.

**Constraints:**

* 1 <= s.length <= 100
* s contains only digits and may contain leading zero(s).

**Code**

class Solution {

public int numDecodings(String s) {

int strLen = s.length();

int[] dp = new int[strLen + 1];

dp[0] = 1;

if (s.charAt(0) != '0') {

dp[1] = 1;

} else {

return 0;

}

for (int i = 2; i <= strLen; ++i) {

if (s.charAt(i - 1) != '0') {

dp[i] += dp[i - 1];

}

if (s.charAt(i - 2) == '1' ||

(s.charAt(i - 2) == '2' && s.charAt(i - 1) <= '6')) {

dp[i] += dp[i - 2];

}

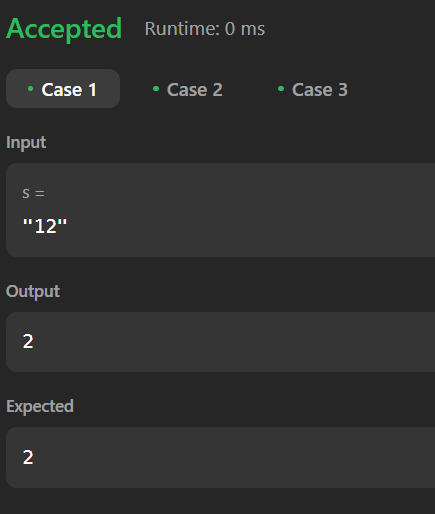
}

return dp[strLen];

}

}

**Output**



**Time Complexity**

O(n)

1. **Best Time to buy and sell stock II**

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the* ***maximum*** *profit you can achieve*.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

**Example 2:**

**Input:** prices = [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Total profit is 4.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

**Constraints:**

* 1 <= prices.length <= 3 \* 104
* 0 <= prices[i] <= 104

**Code**

class Solution {

public int maxProfit(int[] prices) {

int profit = 0;

for (int i = 1; i < prices.length; i++) {

if (prices[i] > prices[i - 1]) {

profit += prices[i] - prices[i - 1];

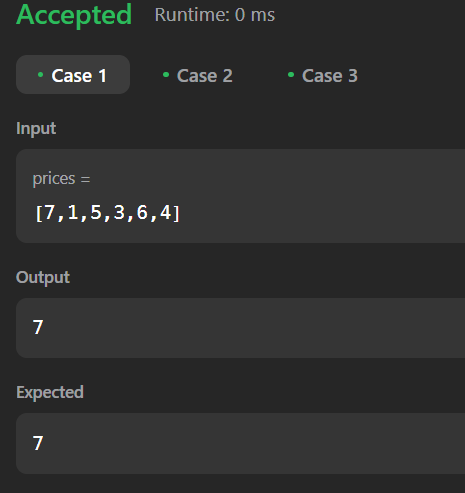
}

}

return profit;

}

}

**Output**

**Time Complexity**

O(n)

1. **Number of islands**

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 300
* grid[i][j] is '0' or '1'.

**Code**

class Solution {

public int numIslands(char[][] grid) {

int islands = 0;

int rows = grid.length;

int cols = grid[0].length;

Set<String> visited = new HashSet<>();

int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

for (int r = 0; r < rows; r++) {

for (int c = 0; c < cols; c++) {

if (grid[r][c] == '1' && !visited.contains(r + "," + c)) {

islands++;

bfs(grid, r, c, visited, directions, rows, cols);

}

}

}

return islands;

}

private void bfs(char[][] grid, int r, int c, Set<String> visited, int[][] directions, int rows, int cols) {

Queue<int[]> q = new LinkedList<>();

visited.add(r + "," + c);

q.add(new int[]{r, c});

while (!q.isEmpty()) {

int[] point = q.poll();

int row = point[0], col = point[1];

for (int[] direction : directions) {

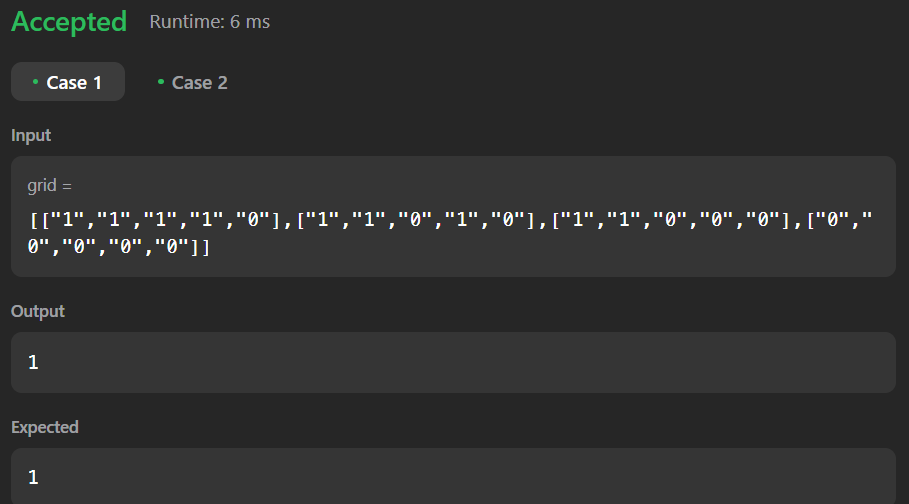
int nr = row + direction[0], nc = col + direction[1];

if (nr >= 0 && nr < rows && nc >= 0 && nc < cols && grid[nr][nc] == '1' && !visited.contains(nr + "," + nc)) {

q.add(new int[]{nr, nc});

visited.add(nr + "," + nc);

}}}}}

**Output**

**Time Complexity**

O(r \* c)

1. **Quick Sort**

**Difficulty: MediumAccuracy: 55.23%Submissions: 236K+Points: 4**

**Implement Quick Sort, a Divide and Conquer algorithm, to sort an array, arr[] in ascending order. Given an array, arr[], with starting index low and ending index high, complete the functions partition() and quickSort(). Use the last element as the pivot so that all elements less than or equal to the pivot come before it, and elements greater than the pivot follow it.**

**Note: The low and high are inclusive.**

**Examples:**

**Input: arr[] = [4, 1, 3, 9, 7]**

**Output: [1, 3, 4, 7, 9]**

**Explanation: After sorting, all elements are arranged in ascending order.**

**Input: arr[] = [2, 1, 6, 10, 4, 1, 3, 9, 7]**

**Output: [1, 1, 2, 3, 4, 6, 7, 9, 10]**

**Explanation: Duplicate elements (1) are retained in sorted order.**

**Input: arr[] = [5, 5, 5, 5]**

**Output: [5, 5, 5, 5]**

**Explanation: All elements are identical, so the array remains unchanged.**

**Constraints:  
1 <= arr.size() <= 103  
1 <= arr[i] <= 104**

**Code**

class Solution {

public int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

public void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

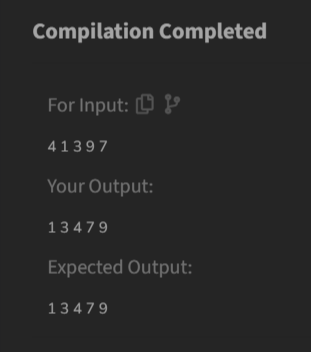
public int[] sortArray(int[] arr) {

quickSort(arr, 0, arr.length - 1);

return arr;

}

}

**Output**

**Time Complexity**

O(n log n)

1. **Merge Sort**

Difficulty: MediumAccuracy: 54.1%Submissions: 205K+Points: 4

Given an array arr[], its starting position l and its ending position r. Sort the array using the merge sort algorithm.

Examples:

Input: arr[] = [4, 1, 3, 9, 7]

Output: [1, 3, 4, 7, 9]

Input: arr[] = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

Output: [1, 2, 3]

Constraints:  
1 <= arr.size() <= 105  
1 <= arr[i] <= 105

**Code**

class Solution {

public void mergeSort(int[] arr, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

private void merge(int[] arr, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int[] L = new int[n1];

int[] R = new int[n2];

for (int i = 0; i < n1; ++i)

L[i] = arr[l + i];

for (int j = 0; j < n2; ++j)

R[j] = arr[m + 1 + j];

int i = 0, j = 0;

int k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

public void processTestCases() {

Scanner scanner = new Scanner(System.in);

int t = scanner.nextInt();

for (int testCase = 1; testCase <= t; testCase++) {

int n = scanner.nextInt();

int[] arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = scanner.nextInt();

}

mergeSort(arr, 0, n - 1);

System.out.print("Sorted array for Test Case " + testCase + ": ");

printArray(arr);

}

scanner.close();

}

public void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

System.out.println();

}

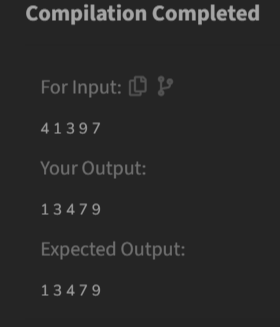
public static void main(String[] args) {

Solution solution = new Solution();

solution.processTestCases();

}

}

**Output**

**Time Complexity**

O(n log n)

1. **Ternary Search**

**Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.**

**You must write an algorithm with O(log n) runtime complexity.**

**Example 1:**

**Input: nums = [-1,0,3,5,9,12], target = 9**

**Output: 4**

**Explanation: 9 exists in nums and its index is 4**

**Example 2:**

**Input: nums = [-1,0,3,5,9,12], target = 2**

**Output: -1**

**Explanation: 2 does not exist in nums so return -1**

**Constraints:**

* **1 <= nums.length <= 104**
* **-104 < nums[i], target < 104**
* **All the integers in nums are unique.**
* **nums is sorted in ascending order.**

**Code**

class Solution {

public int search(int[] nums, int target) {

int left = 0;

int right = nums.length - 1;

while (left <= right) {

int mid1 = left + (right - left) / 3;

int mid2 = right - (right - left) / 3;

if (nums[mid1] == target) {

return mid1;

}

if (nums[mid2] == target) {

return mid2;

}

if (target < nums[mid1]) {

right = mid1 - 1;

} else if (target > nums[mid2]) {

left = mid2 + 1;

} else {

left = mid1 + 1;

right = mid2 - 1;

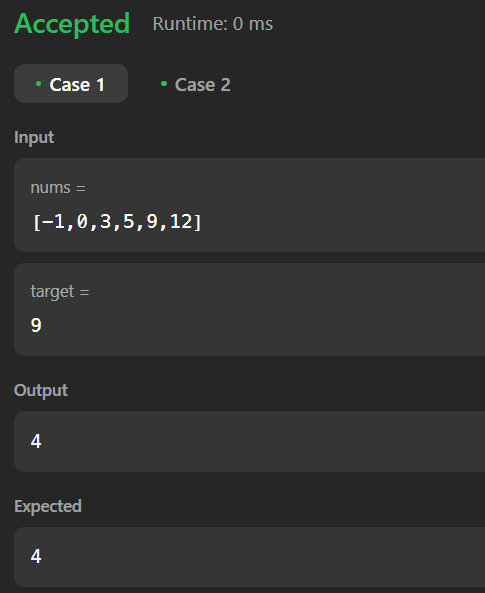
}

}

return -1;

}

}

**Output**

**Time Complexity**

O(log n)

1. **Interpolation Search**

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Constraints:**

* 1 <= nums.length <= 104
* -104 < nums[i], target < 104
* All the integers in nums are **unique**.
* nums is sorted in ascending order.

**Code**

import java.util.\*;

class Solution {

public int search(int[] nums, int target) {

int left = 0;

int right = nums.length - 1;

while (left <= right && target >= nums[left] && target <= nums[right]) {

if (left == right) {

if (nums[left] == target) return left;

return -1;

}

int pos = left + ((target - nums[left]) \* (right - left)

/ (nums[right] - nums[left]));

if (pos < left || pos > right) {

return -1;

}

if (nums[pos] == target) {

return pos;

}

if (nums[pos] < target) {

left = pos + 1;

}

else {

right = pos - 1;

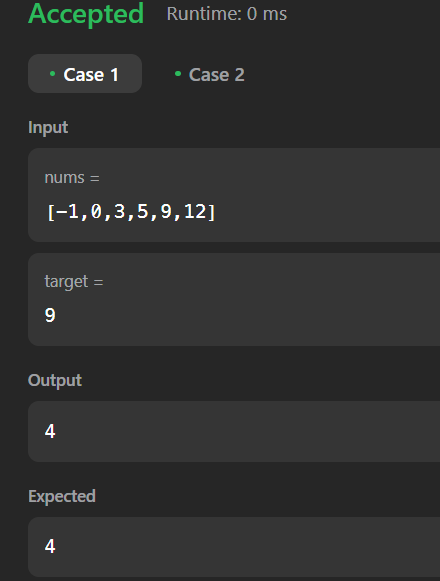
}

}

return -1;

}

}

**Output**

**Time Complexity**

O(log log n)