Harshitha R AIDS A 22AD041 21/11/24

1. **Valid Palindrome**

**A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.**

**Given a string s, return true *if it is a palindrome, or* false *otherwise*.**

**Example 1:**

**Input: s = "A man, a plan, a canal: Panama"**

**Output: true**

**Explanation: "amanaplanacanalpanama" is a palindrome.**

**Example 2:**

**Input: s = "race a car"**

**Output: false**

**Explanation: "raceacar" is not a palindrome.**

**Example 3:**

**Input: s = " "**

**Output: true**

**Explanation: s is an empty string "" after removing non-alphanumeric characters.**

**Since an empty string reads the same forward and backward, it is a palindrome.**

**Constraints:**

* **1 <= s.length <= 2 \* 105**
* **s consists only of printable ASCII characters.**

**Code**

**class Solution {**

**public boolean isPalindrome(String s) {**

**String a = s.toLowerCase();**

**StringBuilder lst = new StringBuilder();**

**for (char i : a.toCharArray()) {**

**if ((i >= 'a' && i <= 'z') || (i >= '0' && i <= '9')) {**

**lst.append(i);**

**}**

**}**

**String b = lst.toString();**

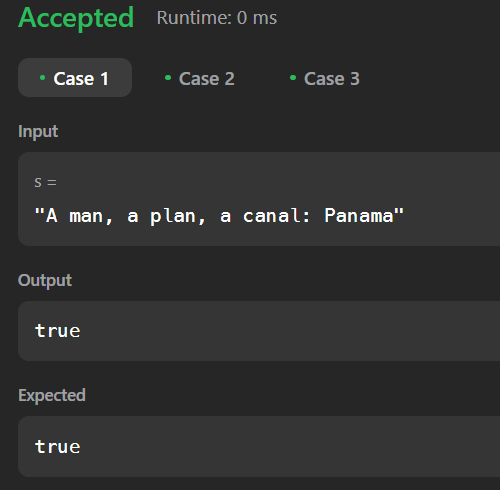
**String c = new StringBuilder(b).reverse().toString();**

**return b.equals(c);**

**}**

**}**

**Output**



**Time Complexity**

O(n)

1. **Is Subsequence**

**Given two strings s and t, return true *if* s *is a subsequence of* t*, or* false *otherwise*.**

**A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).**

**Example 1:**

**Input: s = "abc", t = "ahbgdc"**

**Output: true**

**Example 2:**

**Input: s = "axc", t = "ahbgdc"**

**Output: false**

**Constraints:**

* **0 <= s.length <= 100**
* **0 <= t.length <= 104**
* **s and t consist only of lowercase English letters.**

**Code**

**class Solution {**

**public boolean isSubsequence(String s, String t) {**

**int sp = 0;**

**int tp = 0;**

**while (sp < s.length() && tp < t.length()) {**

**if (s.charAt(sp) == t.charAt(tp)) {**

**sp++;**

**}**

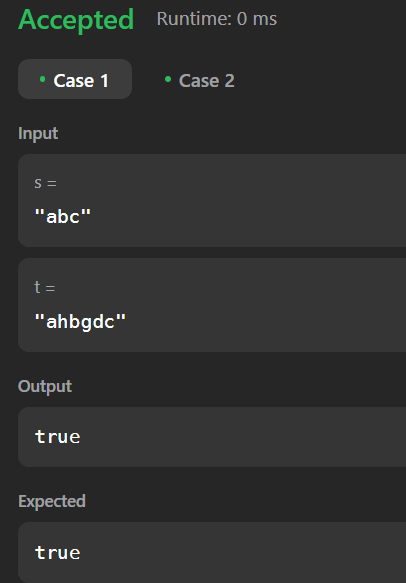
**tp++;**

**}**

**return sp == s.length();**

**}**

**}**



**Output**

**Time Complexity**

O(n)

**2) Two Sum II - Input Array Is Sorted**

**Given a 1-indexed array of integers numbers that is already *sorted in non-decreasing order*, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.**

**Return *the indices of the two numbers,* index1 *and* index2*, added by one as an integer array* [index1, index2] *of length 2.***

**The tests are generated such that there is exactly one solution. You may not use the same element twice.**

**Your solution must use only constant extra space.**

**Example 1:**

**Input: numbers = [2,7,11,15], target = 9**

**Output: [1,2]**

**Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].**

**Example 2:**

**Input: numbers = [2,3,4], target = 6**

**Output: [1,3]**

**Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].**

**Example 3:**

**Input: numbers = [-1,0], target = -1**

**Output: [1,2]**

**Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].**

**Constraints:**

* **2 <= numbers.length <= 3 \* 104**
* **-1000 <= numbers[i] <= 1000**
* **numbers is sorted in non-decreasing order.**
* **-1000 <= target <= 1000**
* **The tests are generated such that there is exactly one solution.**

**Code**

**import java.util.\*;**

**class Solution {**

**public int[] twoSum(int[] numbers, int target) {**

**for (int i = 0; i < numbers.length - 1; i++) {**

**for (int j = i + 1; j < numbers.length; j++) {**

**if (numbers[i] + numbers[j] == target) {**

**return new int[] {i + 1, j + 1};**

**}**

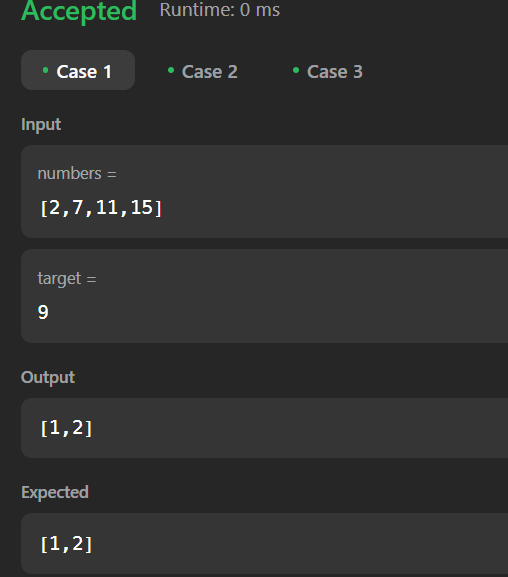
**}**

**}**

**return new int[] {-1, -1};**

**}**

**}**

**Output**

**Time Complexity**

O(n^2)

**4) Container with most water**

**You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).**

**Find two lines that together with the x-axis form a container, such that the container contains the most water.**

**Return *the maximum amount of water a container can store*.**

**Notice that you may not slant the container.**

**Example 1:**

****

**Input: height = [1,8,6,2,5,4,8,3,7]**

**Output: 49**

**Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.**

**Example 2:**

**Input: height = [1,1]**

**Output: 1**

**Constraints:**

* **n == height.length**
* **2 <= n <= 105**
* **0 <= height[i] <= 104**

**Code**

**class Solution {**

**public int maxArea(int[] height) {**

**int maxArea=0;**

**int l=0;**

**int r=height.length-1;**

**while(l<r){**

**int currentArea=Math.min(height[l],height[r])\*(r-l);**

**maxArea=Math.max(maxArea,currentArea);**

**if(height[l]<height[r]){**

**l++;**

**}**

**else{**

**r--;**

**}**

**}**

**return maxArea;**

**}**

**public static void main(String[] args){**

**int[]height1={1,8,6,2,5,4,8,3,7};**

**Solution solution=new Solution();**

**int maxArea1= solution.maxArea(height1);**

**System.out.println(maxArea1);**

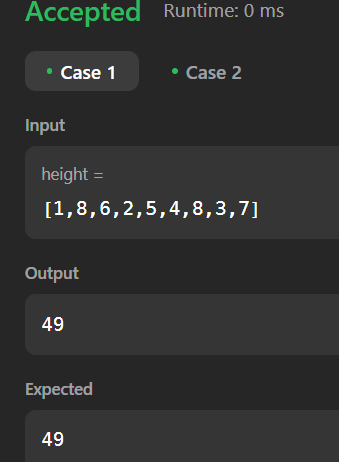
**int[]height2={1,1};**

**int maxArea2=solution.maxArea(height2);**

**System.out.println(maxArea2);**

**}**

**}**

**Output**

**Time Complexity**

O(n)

**5) 3Sum**

**Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.**

**Notice that the solution set must not contain duplicate triplets.**

**Example 1:**

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

**Output: [[-1,-1,2],[-1,0,1]]**

**Explanation:**

**nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.**

**nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.**

**nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.**

**The distinct triplets are [-1,0,1] and [-1,-1,2].**

**Notice that the order of the output and the order of the triplets does not matter.**

**Example 2:**

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

**Output: []**

**Explanation: The only possible triplet does not sum up to 0.**

**Example 3:**

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

**Output: [[0,0,0]]**

**Explanation: The only possible triplet sums up to 0.**

**Constraints:**

* **3 <= nums.length <= 3000**
* **-105 <= nums[i] <= 105**

**Code**

**import java.util.ArrayList;**

**import java.util.Arrays;**

**import java.util.List;**

**class Solution {**

**public List<List<Integer>> threeSum(int[] nums) {**

**Arrays.sort(nums);**

**List<List<Integer>>result= new ArrayList<>();**

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

**{**

**if(i==0 || (i>0 && nums[i]!= nums[i-1]))**

**{**

**int t= -nums[i];**

**int l=i+1;**

**int r=nums.length-1;**

**while(l<r){**

**int sum=nums[l]+nums[r];**

**if(sum==t){**

**result.add(Arrays.asList(nums[i],nums[l],nums[r]));**

**while(l<r && nums[l]==nums[l+1])**

**{**

**l++;}**

**while(l<r && nums[r]==nums[r-1])**

**{**

**r--;}**

**l++;**

**r--;**

**}**

**else if(sum<t){**

**l++;**

**}**

**else{**

**r--;**

**}**

**}**

**}**

**}**

**return result;**

**}**

**public static void main(String[] args){**

**int[]nums={-1,0,1,2,-1,-4};**

**Solution solution = new Solution();**

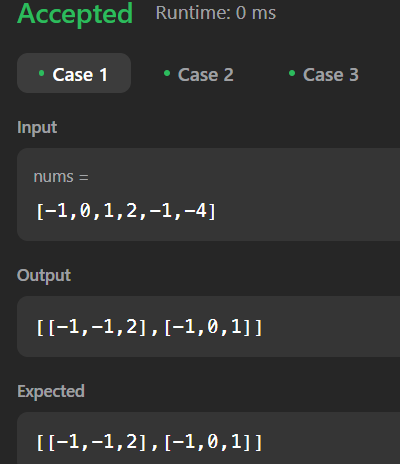
**List<List<Integer>> result=solution.threeSum(nums);**

**System.out.println(result);**

**}**

**}**

**Output**

****

**Time Complexity**

O(n^2)

**6) Minimum Size Subarray Sum**

Given an array of positive integers nums and a positive integer target, return *the* ***minimal length*** *of a*

*subarray*

*whose sum is greater than or equal to* target. If there is no such subarray, return 0 instead.

**Example 1:**

**Input:** target = 7, nums = [2,3,1,2,4,3]

**Output:** 2

**Explanation:** The subarray [4,3] has the minimal length under the problem constraint.

**Example 2:**

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

**Output:** 1

**Example 3:**

**Input:** target = 11, nums = [1,1,1,1,1,1,1,1]

**Output:** 0

**Constraints:**

* 1 <= target <= 109
* 1 <= nums.length <= 105
* 1 <= nums[i] <= 104

**Code**

**class Solution {**

**public int minSubArrayLen(int target, int[] nums) {**

**int left=0,right=0,sum =0;**

**int ans = Integer.MAX\_VALUE;**

**for(right=0;right<nums.length;right++){**

**sum +=nums[right];**

**while(sum>=target){**

**ans=Math.min(ans,right-left+1);**

**sum -=nums[left++];**

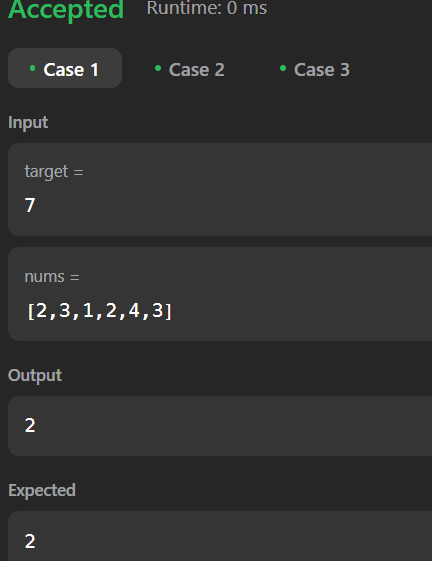
**}**

**}**

**return ans == Integer.MAX\_VALUE ? 0:ans;**

**}**

**}**



**Output**

**Time Complexity**

O(n)

**7) Longest Substring Without Repeating Characters**

**Given a string s, find the length of the longest**

**substring**

**without repeating characters.**

**Example 1:**

**Input: s = "abcabcbb"**

**Output: 3**

**Explanation: The answer is "abc", with the length of 3.**

**Example 2:**

**Input: s = "bbbbb"**

**Output: 1**

**Explanation: The answer is "b", with the length of 1.**

**Example 3:**

**Input: s = "pwwkew"**

**Output: 3**

**Explanation: The answer is "wke", with the length of 3.**

**Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.**

**Constraints:**

* **0 <= s.length <= 5 \* 104**
* **s consists of English letters, digits, symbols and spaces.**

**Code**

**class Solution {**

**public int lengthOfLongestSubstring(String s) {**

**int left = 0;**

**int maxLength = 0;**

**HashSet<Character> charSet = new HashSet<>();**

**for (int right = 0; right < s.length(); right++) {**

**while (charSet.contains(s.charAt(right))) {**

**charSet.remove(s.charAt(left));**

**left++;**

**}**

**charSet.add(s.charAt(right));**

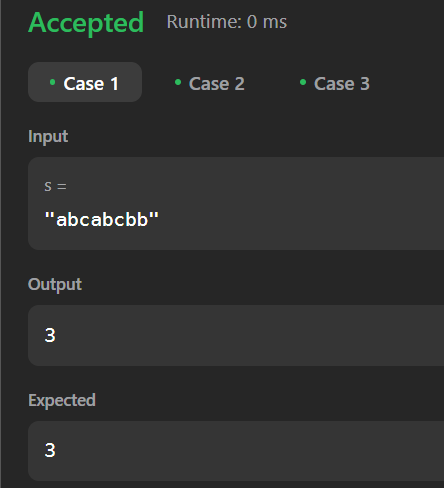
**maxLength = Math.max(maxLength, right - left + 1);**

**}**

**return maxLength;**

**}**

**}**

**Output**

**Time Complexity**

O(n)

**8) Substring with concatenation of all words**

**You are given a string s and an array of strings words. All the strings of words are of the same length.**

**A concatenated string is a string that exactly contains all the strings of any permutation of words concatenated.**

* **For example, if words = ["ab","cd","ef"], then "abcdef", "abefcd", "cdabef", "cdefab", "efabcd", and "efcdab" are all concatenated strings. "acdbef" is not a concatenated string because it is not the concatenation of any permutation of words.**

**Return an array of *the starting indices* of all the concatenated substrings in s. You can return the answer in any order.**

**Example 1:**

**Input: s = "barfoothefoobarman", words = ["foo","bar"]**

**Output: [0,9]**

**Explanation:**

**The substring starting at 0 is "barfoo". It is the concatenation of ["bar","foo"] which is a permutation of words.  
The substring starting at 9 is "foobar". It is the concatenation of ["foo","bar"] which is a permutation of words.**

**Example 2:**

**Input: s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]**

**Output: []**

**Explanation:**

**There is no concatenated substring.**

**Example 3:**

**Input: s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]**

**Output: [6,9,12]**

**Explanation:**

**The substring starting at 6 is "foobarthe". It is the concatenation of ["foo","bar","the"].  
The substring starting at 9 is "barthefoo". It is the concatenation of ["bar","the","foo"].  
The substring starting at 12 is "thefoobar". It is the concatenation of ["the","foo","bar"].**

**Constraints:**

* **1 <= s.length <= 104**
* **1 <= words.length <= 5000**
* **1 <= words[i].length <= 30**
* **s and words[i] consist of lowercase English letters.**

**Code**

**class Solution**

**{**

**public List<Integer> findSubstring(String s, String[] words)**

**{**

**if(words[0].length()\*words.length>s.length())**

**return new ArrayList<>();**

**Map<String,Integer> word\_frq=new HashMap<>();**

**List<Integer> ans=new ArrayList<>();**

**for(String str:words)**

**word\_frq.put(str,word\_frq.getOrDefault(str,0)+1);**

**int wordlen=words[0].length();**

**String[] str=new String[s.length()];**

**for(int i=0;i<wordlen;i++)**

**{**

**Map<String,Integer> frq=new HashMap<>();**

**int begin=i,size=0;**

**for(int j=i;j<=s.length()-wordlen;j+=wordlen)**

**{**

**str[j]=s.substring(j,j+wordlen);**

**if(word\_frq.containsKey(str[j]))**

**{**

**begin= begin==-1? j:begin;**

**frq.put(str[j],frq.getOrDefault(str[j],0)+1);**

**size++;**

**if(size==words.length)**

**{**

**if(frq.equals(word\_frq))**

**ans.add(begin);**

**frq.put(str[begin],frq.get(str[begin])-1);**

**begin+=wordlen;**

**size--;**

**}**

**}**

**else**

**{**

**begin=-1;**

**size=0;**

**frq.clear();**

**}**

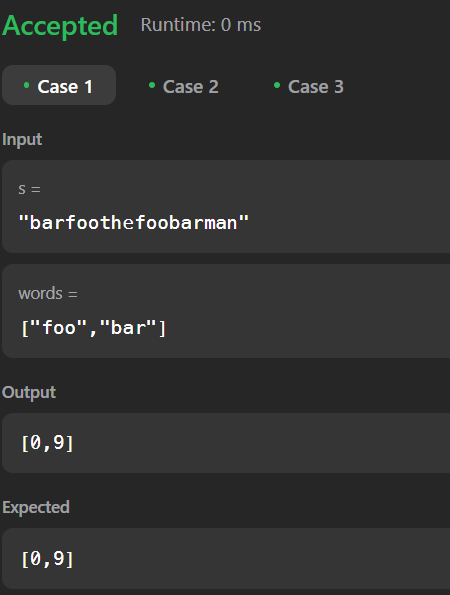
**}**

**}**

**return ans;**

**}**

**}**

**Output**

**Time Complexity**

O(n \* m \* k)

**9) Minimum Window Substring**

**Given two strings s and t of lengths m and n respectively, return *the minimum window***

***substring***

***of* s *such that every character in* t *(including duplicates) is included in the window*. If there is no such substring, return *the empty string* "".**

**The testcases will be generated such that the answer is unique.**

**Example 1:**

**Input: s = "ADOBECODEBANC", t = "ABC"**

**Output: "BANC"**

**Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.**

**Example 2:**

**Input: s = "a", t = "a"**

**Output: "a"**

**Explanation: The entire string s is the minimum window.**

**Example 3:**

**Input: s = "a", t = "aa"**

**Output: ""**

**Explanation: Both 'a's from t must be included in the window.**

**Since the largest window of s only has one 'a', return empty string.**

**Constraints:**

* **m == s.length**
* **n == t.length**
* **1 <= m, n <= 105**
* **s and t consist of uppercase and lowercase English letters.**

**Code**

**class Solution {**

**public String minWindow(String s, String t) {**

**if (s.length() < t.length()) {**

**return "";**

**}**

**Map<Character, Integer> charCount = new HashMap<>();**

**for (char ch : t.toCharArray()) {**

**charCount.put(ch, charCount.getOrDefault(ch, 0) + 1);**

**}**

**int targetCharsRemaining = t.length();**

**int[] minWindow = {0, Integer.MAX\_VALUE};**

**int startIndex = 0;**

**for (int endIndex = 0; endIndex < s.length(); endIndex++) {**

**char ch = s.charAt(endIndex);**

**if (charCount.containsKey(ch) && charCount.get(ch) > 0) {**

**targetCharsRemaining--;**

**}**

**charCount.put(ch, charCount.getOrDefault(ch, 0) - 1);**

**if (targetCharsRemaining == 0) {**

**while (true) {**

**char charAtStart = s.charAt(startIndex);**

**if (charCount.containsKey(charAtStart) && charCount.get(charAtStart) == 0) {**

**break;**

**}**

**charCount.put(charAtStart, charCount.getOrDefault(charAtStart, 0) + 1);**

**startIndex++;**

**}**

**if (endIndex - startIndex < minWindow[1] - minWindow[0]) {**

**minWindow[0] = startIndex;**

**minWindow[1] = endIndex;**

**}**

**charCount.put(s.charAt(startIndex), charCount.getOrDefault(s.charAt(startIndex), 0) + 1);**

**targetCharsRemaining++;**

**startIndex++;**

**}**

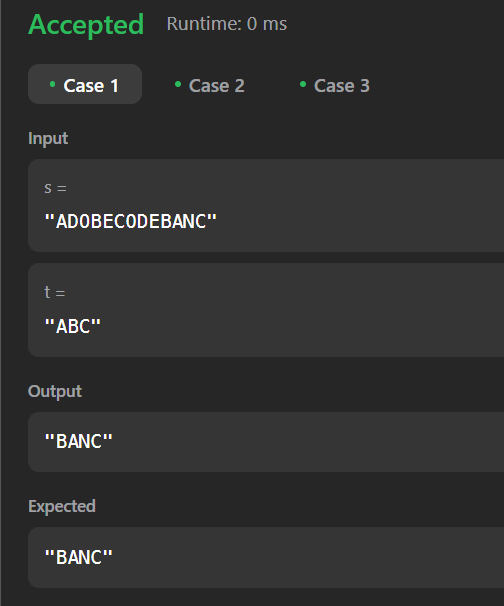
**}**

**return minWindow[1] >= s.length() ? "" : s.substring(minWindow[0], minWindow[1] + 1);**

**}**

**}**

**Output**



**Time Complexity**

O(S + T)

**10) Valid Parantheses**

**Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.**

**An input string is valid if:**

1. **Open brackets must be closed by the same type of brackets.**
2. **Open brackets must be closed in the correct order.**
3. **Every close bracket has a corresponding open bracket of the same type.**

**Example 1:**

**Input: s = "()"**

**Output: true**

**Example 2:**

**Input: s = "()[]{}"**

**Output: true**

**Example 3:**

**Input: s = "(]"**

**Output: false**

**Example 4:**

**Input: s = "([])"**

**Output: true**

**Constraints:**

* **1 <= s.length <= 104**
* **s consists of parentheses only '()[]{}'.**

**Code**

**class Solution {**

**public boolean isValid(String s) {**

**Stack<Character> stack = new Stack<Character>();**

**for (char c : s.toCharArray()) {**

**if (c == '(')**

**stack.push(')');**

**else if (c == '{')**

**stack.push('}');**

**else if (c == '[')**

**stack.push(']');**

**else if (stack.isEmpty() || stack.pop() != c)**

**return false;**

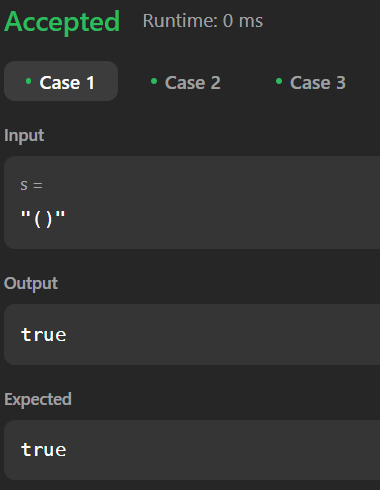
**}**

**return stack.isEmpty();**

**}**

**}**

**Output**



**Time Complexity**

O(n)

**11) Simplify Path**

You are given an *absolute* path for a Unix-style file system, which always begins with a slash '/'. Your task is to transform this absolute path into its **simplified canonical path**.

The *rules* of a Unix-style file system are as follows:

* A single period '.' represents the current directory.
* A double period '..' represents the previous/parent directory.
* Multiple consecutive slashes such as '//' and '///' are treated as a single slash '/'.
* Any sequence of periods that does **not match** the rules above should be treated as a **valid directory or** **file name**. For example, '...' and '....' are valid directory or file names.

The simplified canonical path should follow these *rules*:

* The path must start with a single slash '/'.
* Directories within the path must be separated by exactly one slash '/'.
* The path must not end with a slash '/', unless it is the root directory.
* The path must not have any single or double periods ('.' and '..') used to denote current or parent directories.

Return the **simplified canonical path**.

**Example 1:**

**Input:** path = "/home/"

**Output:** "/home"

**Explanation:**

The trailing slash should be removed.

**Example 2:**

**Input:** path = "/home//foo/"

**Output:** "/home/foo"

**Explanation:**

Multiple consecutive slashes are replaced by a single one.

**Example 3:**

**Input:** path = "/home/user/Documents/../Pictures"

**Output:** "/home/user/Pictures"

**Explanation:**

A double period ".." refers to the directory up a level (the parent directory).

**Example 4:**

**Input:** path = "/../"

**Output:** "/"

**Explanation:**

Going one level up from the root directory is not possible.

**Example 5:**

**Input:** path = "/.../a/../b/c/../d/./"

**Output:** "/.../b/d"

**Explanation:**

"..." is a valid name for a directory in this problem.

**Constraints:**

* 1 <= path.length <= 3000
* path consists of English letters, digits, period '.', slash '/' or '\_'.
* path is a valid absolute Unix path.

**Code**

public class Solution {

public static String simplifyPath(String path) {

Stack<String> stack = new Stack<>();

String[] components = path.split("/");

for (String component : components) {

if (component.equals("") || component.equals(".")) {

continue;

}

if (component.equals("..")) {

if (!stack.isEmpty()) {

stack.pop();

}

} else {

stack.push(component);

}

}

if (stack.isEmpty()) {

return "/";

}

StringBuilder result = new StringBuilder();

for (String dir : stack) {

result.append("/").append(dir);

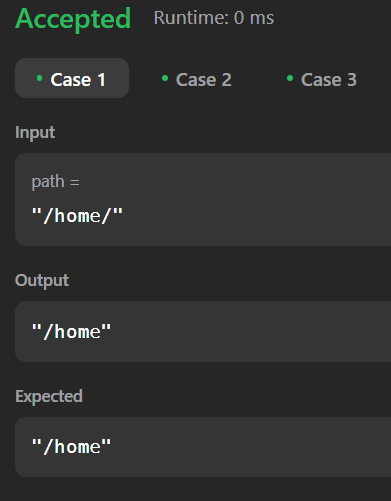
}

return result.toString();

}

}

**Output**



**Time Complexity**

O(n)

**12) Min Stack**

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

* MinStack() initializes the stack object.
* void push(int val) pushes the element val onto the stack.
* void pop() removes the element on the top of the stack.
* int top() gets the top element of the stack.
* int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

**Example 1:**

**Input**

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

**Output**

[null,null,null,null,-3,null,0,-2]

**Explanation**

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top(); // return 0

minStack.getMin(); // return -2

**Constraints:**

* -231 <= val <= 231 - 1
* Methods pop, top and getMin operations will always be called on **non-empty** stacks.
* At most 3 \* 104 calls will be made to push, pop, top, and getMin.

**Code**

class MinStack {

Stack<Integer> allValues = new Stack<>();

Stack<Integer> minStack = new Stack<>();

HashMap<Integer,Integer> unique = new HashMap<>();

int min = Integer.MAX\_VALUE;

public MinStack() {

}

public void push(int val) {

if(allValues.isEmpty() || val<min){

min = val;

minStack.push(val);

}

allValues.push(val);

unique.put(val, unique.getOrDefault(val, 0)+1);

}

public void pop() {

if(allValues.peek() == min && unique.get(allValues.peek()) == 1){

minStack.pop();

if(!minStack.isEmpty()){

min = minStack.peek();

}

}

else{

unique.put(allValues.peek(), unique.get(allValues.peek())-1);

}

allValues.pop();

}

public int top() {

return allValues.peek();

}

public int getMin() {

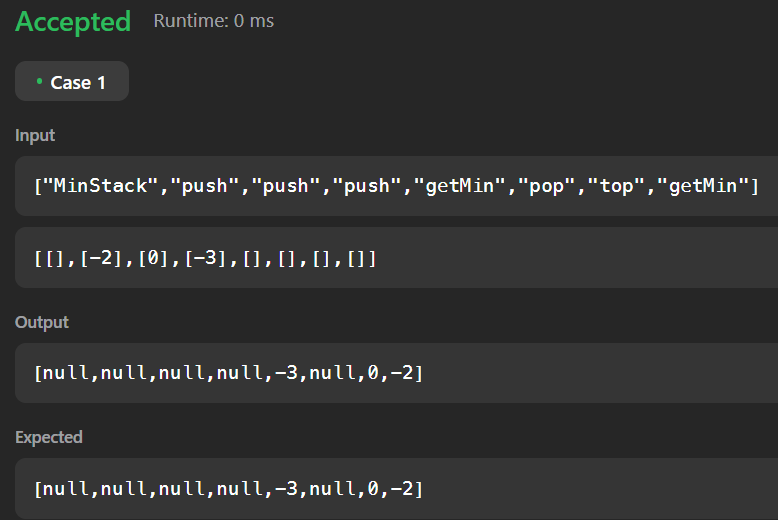
return minStack.peek();

}

}

**Time Complexity**

O(1)

**Output**

**13) Evaluate reverse polish notation**

You are given an array of strings tokens that represents an arithmetic expression in a [Reverse Polish Notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Evaluate the expression. Return *an integer that represents the value of the expression*.

**Note** that:

* The valid operators are '+', '-', '\*', and '/'.
* Each operand may be an integer or another expression.
* The division between two integers always **truncates toward zero**.
* There will not be any division by zero.
* The input represents a valid arithmetic expression in a reverse polish notation.
* The answer and all the intermediate calculations can be represented in a **32-bit** integer.

**Example 1:**

**Input:** tokens = ["2","1","+","3","\*"]

**Output:** 9

**Explanation:** ((2 + 1) \* 3) = 9

**Example 2:**

**Input:** tokens = ["4","13","5","/","+"]

**Output:** 6

**Explanation:** (4 + (13 / 5)) = 6

**Example 3:**

**Input:** tokens = ["10","6","9","3","+","-11","\*","/","\*","17","+","5","+"]

**Output:** 22

**Explanation:** ((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

**Constraints:**

* 1 <= tokens.length <= 104
* tokens[i] is either an operator: "+", "-", "\*", or "/", or an integer in the range [-200, 200].

**Code**

class Solution {

public int evalRPN(String[] tokens) {

Stack<String> st = new Stack<>();

for(int i=0;i<tokens.length;i++)

{

if(!tokens[i].equals("+") && !tokens[i].equals("-") && !tokens[i].equals("\*") && !tokens[i].equals("/"))

st.push(tokens[i]);

else

{

int ans=0;

int val1=Integer.parseInt(st.pop());

int val2=Integer.parseInt(st.pop());

if(tokens[i].equals("+"))

ans = val1+val2;

else if(tokens[i].equals("-"))

ans = val2-val1;

else if(tokens[i].equals("\*"))

ans = val1\*val2;

else if(tokens[i].equals("/"))

ans = val2/val1;

String temp = Integer.toString(ans);

st.push(temp);

}

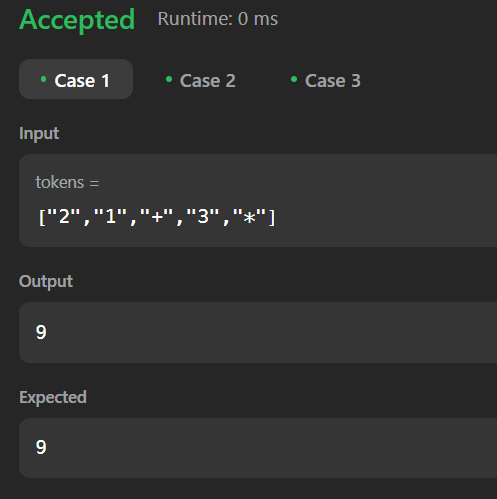
}

return Integer.parseInt(st.peek());

}

}

**Output**



**Time Complexity**

O(n)

**14) Basic Calculator**

Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

**Note:** You are **not** allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

**Example 1:**

**Input:** s = "1 + 1"

**Output:** 2

**Example 2:**

**Input:** s = " 2-1 + 2 "

**Output:** 3

**Example 3:**

**Input:** s = "(1+(4+5+2)-3)+(6+8)"

**Output:** 23

**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of digits, '+', '-', '(', ')', and ' '.
* s represents a valid expression.
* '+' is **not** used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).
* '-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).
* There will be no two consecutive operators in the input.
* Every number and running calculation will fit in a signed 32-bit integer.

**Code**

class Solution {

public int calculate(String s) {

Stack<Integer>st=new Stack<>();

int n=s.length();

int num=0;

int ans=0;

int sign=1;

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

char ch=s.charAt(i);

if(Character.isDigit(ch)){

num=10\*num+(ch-'0');

}

else if(ch=='+'){

ans=ans+(num\*sign);

sign=1;

num=0;

}

else if(ch=='-'){

ans=ans+(num\*sign);

sign=-1;

num=0;

}

else if(ch=='('){

st.push(ans);

st.push(sign);

ans=0;

sign=1;

}

else if(ch==')'){

ans=ans+(num\*sign);

num=0;

ans=ans\*(st.pop());

ans+=st.pop();

}

}

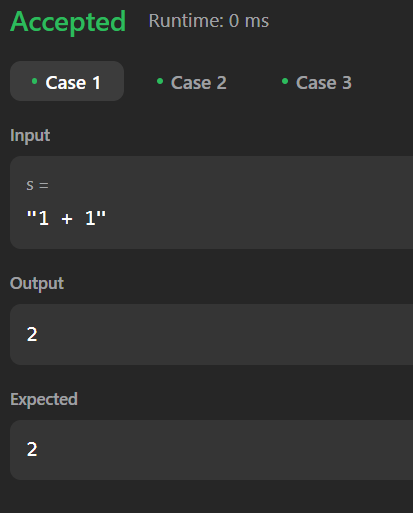
ans=ans+(num\*sign);

return ans;

}

}

**Output**



**Time Complexity**

O(n)

**15) Search insert position**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

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

**Example 1:**

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

**Example 2:**

**Input:** nums = [1,3,5,6], target = 2

**Output:** 1

**Example 3:**

**Input:** nums = [1,3,5,6], target = 7

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums contains **distinct** values sorted in **ascending** order.
* -104 <= target <= 104

**Code**

class Solution {

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

int left = 0;

int right = nums.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) {

return mid;

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

right = mid - 1;

} else {

left = mid + 1;

}

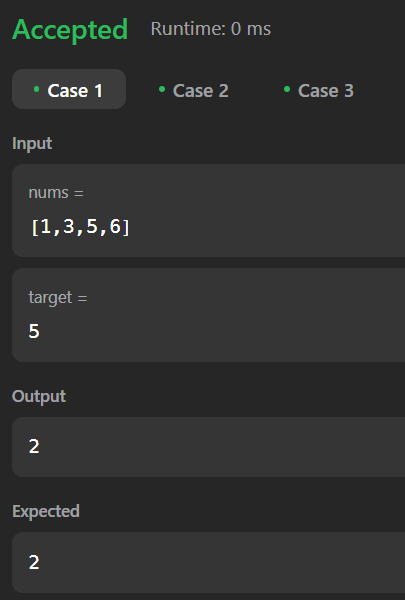
}

return left;

}

}

**Output**



**Time Complexity**

O(log n)

**16) Search a 2D matrix**

You are given an m x n integer matrix matrix with the following two properties:

* Each row is sorted in non-decreasing order.
* The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m \* n)) time complexity.

**Example 1:**

****

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Example 2:**

****

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 100
* -104 <= matrix[i][j], target <= 104

**Code**

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

int top = 0;

int bot = matrix.length - 1;

while (top <= bot) {

int mid = (top + bot) / 2;

if (matrix[mid][0] < target && matrix[mid][matrix[mid].length - 1] > target) {

break;

} else if (matrix[mid][0] > target) {

bot = mid - 1;

} else {

top = mid + 1;

}

}

int row = (top + bot) / 2;

int left = 0;

int right = matrix[row].length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (matrix[row][mid] == target) {

return true;

} else if (matrix[row][mid] > target) {

right = mid - 1;

} else {

left = mid + 1;

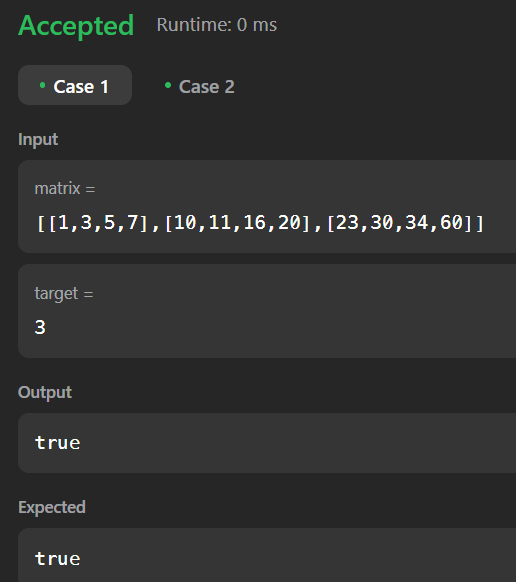
}

}

return false;

}

}

**Output**

**Time Complexity**

O(log(m) + log(n))

**17) Find peak element**

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

**Example 1:**

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

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

**Example 2:**

**Input:** nums = [1,2,1,3,5,6,4]

**Output:** 5

**Explanation:** Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

**Constraints:**

* 1 <= nums.length <= 1000
* -231 <= nums[i] <= 231 - 1
* nums[i] != nums[i + 1] for all valid i.

**Code**

class Solution {

public int findPeakElement(int[] nums) {

int left = 0;

int right = nums.length - 1;

while (left < right) {

int mid = (left + right) / 2;

if (nums[mid] > nums[mid + 1]) {

right = mid;

} else {

left = mid + 1;

}

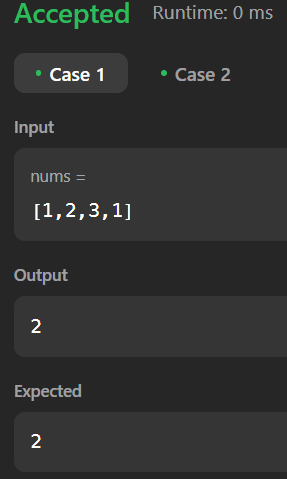
}

return left;

}

}

**Output**



**Time Complexity**

O(log n)

**18) Search in rotated sorted array**

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

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

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2], target = 3

**Output:** -1

**Example 3:**

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

**Output:** -1

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is an ascending array that is possibly rotated.
* -104 <= target <= 104

**Code**

**class Solution {**

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

**int left = 0;**

**int right = nums.length - 1;**

**while (left <= right) {**

**int mid = (left + right) / 2;**

**if (nums[mid] == target) {**

**return mid;**

**} else if (nums[mid] >= nums[left]) {**

**if (nums[left] <= target && target <= nums[mid]) {**

**right = mid - 1;**

**} else {**

**left = mid + 1;**

**}**

**} else {**

**if (nums[mid] <= target && target <= nums[right]) {**

**left = mid + 1;**

**} else {**

**right = mid - 1;**

**}**

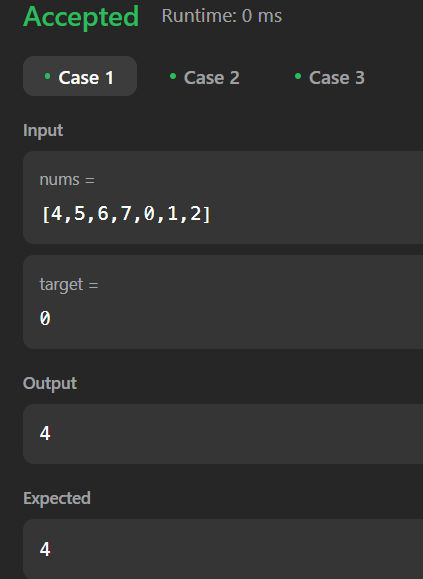
**}**

**}**

**return -1;**

**}**

**}**

**Output**

**Time Complexity**

O(log n)

**19) Find First and Last Position of element in sorted array**

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

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

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

**Example 2:**

**Input:** nums = [5,7,7,8,8,10], target = 6

**Output:** [-1,-1]

**Example 3:**

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

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

**Code**

**class Solution {**

**public int[] searchRange(int[] nums, int target) {**

**int[] result = {-1, -1};**

**int left = binarySearch(nums, target, true);**

**int right = binarySearch(nums, target, false);**

**result[0] = left;**

**result[1] = right;**

**return result;**

**}**

**private int binarySearch(int[] nums, int target, boolean isSearchingLeft) {**

**int left = 0;**

**int right = nums.length - 1;**

**int idx = -1;**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**if (nums[mid] > target) {**

**right = mid - 1;**

**} else if (nums[mid] < target) {**

**left = mid + 1;**

**} else {**

**idx = mid;**

**if (isSearchingLeft) {**

**right = mid - 1;**

**} else {**

**left = mid + 1;**

**}**

**}**

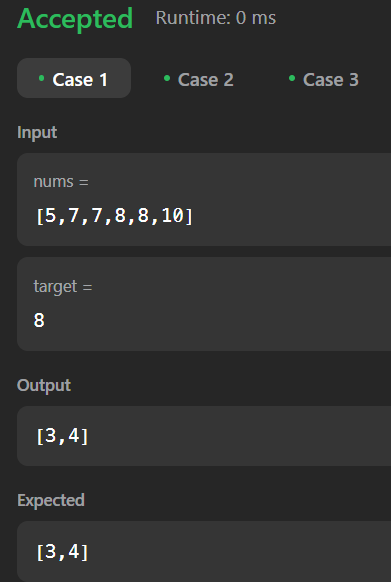
**}**

**return idx;**

**}**

**}**

**Output**



**Time Complexity**

O(log n)

**20) Find minimum in rotated sorted array**

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

**Example 1:**

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

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2]

**Output:** 0

**Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

**Example 3:**

**Input:** nums = [11,13,15,17]

**Output:** 11

**Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.

**Constraints:**

* n == nums.length
* 1 <= n <= 5000
* -5000 <= nums[i] <= 5000
* All the integers of nums are **unique**.
* nums is sorted and rotated between 1 and n times.

**Code**

**class Solution {**

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

**int left = 0;**

**int right = nums.length-1;**

**while(nums[left] > nums[right]){**

**int mid = left + (right - left)/2;**

**if(nums[mid+1] < nums[mid]){**

**return nums[mid+1];**

**}**

**if(nums[mid-1] > nums[mid]){**

**return nums[mid];**

**}**

**if(nums[mid] < nums[right]){**

**right = mid - 1;**

**}**

**if(nums[mid] > nums[left]){**

**left = mid + 1;**

**}**

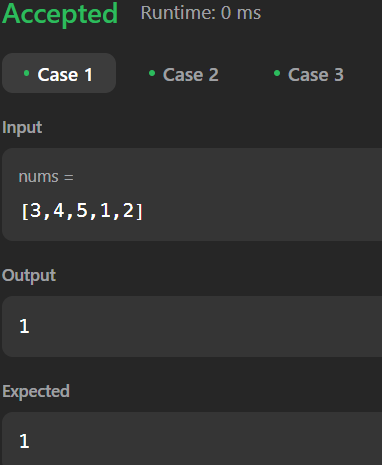
**}**

**return nums[left];**

**}**

**}**

**Output**



**Time Complexity**

O(log n)