GATE TECHNICAL TRAINING – DSA CODING PRACTICE PROBLEMS 2026

DATE: 21-11-2024 **NAME: S Yathissh– CSBS**

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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class ValidPalindrome {
	public static boolean isValid(String s) {
		StringBuilder sb =  new StringBuilder();
		for(char c : s.toCharArray()) {
			if(Character.isLetterOrDigit(c)) {
```

```
                        sb.append(Character.toLowerCase(c));
                    }
                }

                String forward = sb.toString();
                String reversed = sb.reverse().toString();
                return forward.equals(reversed);

        }
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the string: ");
                String s = sc.next();

                if(isValid(s))
                        System.out.println("It is a valid palindrome");

                else
                        System.out.println("It is not a valid palindrome");

        }

}
```

OUTPUT:

```
Enter the string:
race a car
It is not a valid palindrome
```

```
Enter the string:
A man, a plan, a canal: Panama
It is a valid palindrome
```

TIME COMPLEXITY: O(n)

2. 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

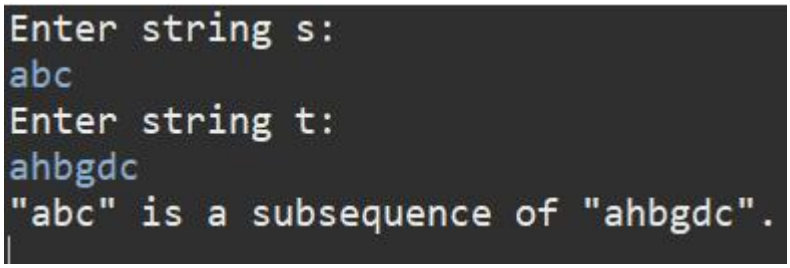
0 <= t.length <= 104

s and t consist only of lowercase English letters.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.Scanner;

class IsSubsequence {
    public boolean isSubsequence(String s, String t) {
        int i = 0, j = 0;
        while (i < s.length() && j < t.length()) {
            if (s.charAt(i) == t.charAt(j)) {
                i++;
            }
            j++;
        }
        return i == s.length();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter string s: ");
        String s = sc.nextLine();
        System.out.println("Enter string t: ");
        String t = sc.nextLine();

        IsSubsequence solution = new IsSubsequence();
        boolean result = solution.isSubsequence(s, t);

        if (result) {
            System.out.println("\"" + s + "\" is a subsequence of \"" + t + "\".");
        } else {
            System.out.println("\"" + s + "\" is NOT a subsequence of \"" + t + "\".");
        }

        sc.close();
    }
}
```

OUTPUT:



TIME COMPLEXITY: O(n)

3. 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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.Scanner;

public class TwoSumII {
    public static int[] twoSum(int[] numbers, int target) {
        int left = 0;
        int right = numbers.length - 1;
        while (left < right) {
            int currSum = numbers[left] + numbers[right];
            if (target == currSum) {
                return new int[] { left + 1, right + 1 };
            } else if (currSum > target) {
                right--;
            } else {
                left++;
            }
        }
        return new int[] { -1, -1 };
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] numbers = new int[n];

        System.out.println("Enter the elements in the array: ");
        for (int i = 0; i < n; i++) {
            numbers[i] = sc.nextInt();
        }

        System.out.println("Enter the target value: ");
        int target = sc.nextInt();

        int[] result = twoSum(numbers, target);

        if (result[0] != -1) {
            System.out.println("The Two sum is : " + "["+ result[0] +"," + result[1]+
"]");
        } else {
            System.out.println("No solution found.");
        }

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
4
Enter the elements in the array:
2
7
11
15
Enter the target value:
9
The Two sum is : [1,2]
```

TIME COMPLEXITY: O(n)

4. CONTAINER WITH MOST WATER

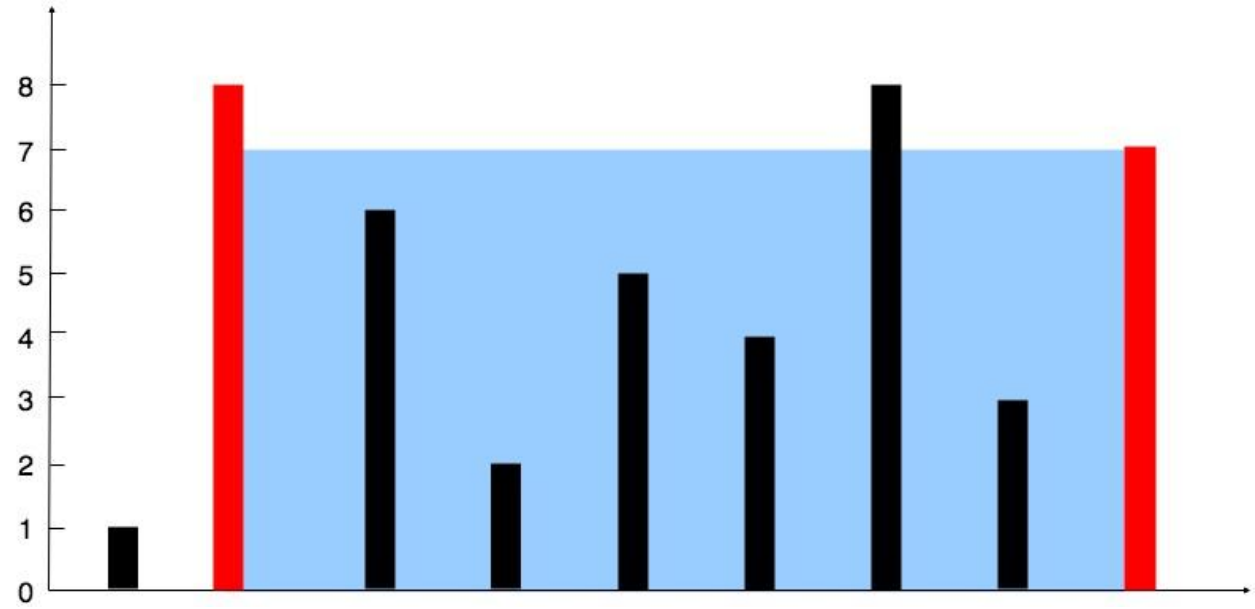
You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

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

PROGRAM:

```
package dsaPracticeProblems;
import java.util.Scanner;

class ContainerWithMostWater {
    public static int maxArea(int[] height) {
        int maxArea = 0;
        int left = 0;
        int right = height.length - 1;

        while (left < right) {
            maxArea = Math.max(maxArea, (right - left) * Math.min(height[left],
height[right]));

            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertical lines: ");
        int n = sc.nextInt();
        int[] height = new int[n];

        System.out.println("Enter the heights of the vertical lines: ");
        for (int i = 0; i < n; i++) {
            height[i] = sc.nextInt();
        }

        int maxArea = maxArea(height);

        System.out.println("The maximum area is: " + maxArea);
```

```
        sc.close();
    }
}
```

OUTPUT:

```
Enter the number of vertical lines:
9
Enter the heights of the vertical lines:
1 8 6 2 5 4 8 3 7
The maximum area is: 49
```

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

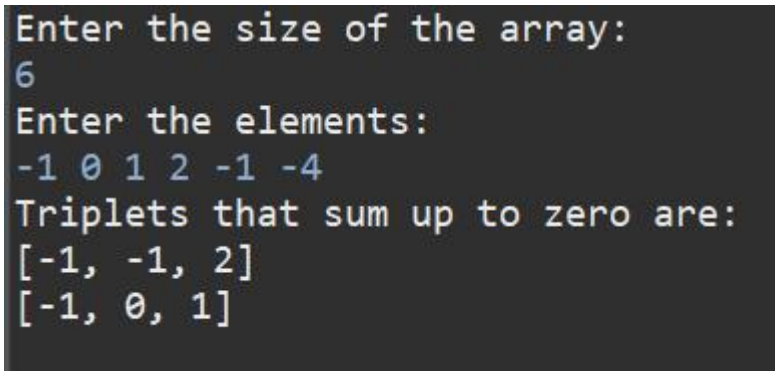
-105 <= nums[i] <= 105

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

class ThreeSum {
    public static List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> n = new ArrayList<>();
        int l = nums.length;
        Arrays.sort(nums);

        for (int i = 0; i < l; i++) {
            if (i > 0 && nums[i] == nums[i - 1])
                continue;

            int left = i + 1;
            int right = l - 1;

            while (left < right) {
                int currSum = nums[i] + nums[left] + nums[right];
                if (currSum > 0)
                    right--;
                else if (currSum < 0)
                    left++;
                else {
                    n.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    left++;
                    while (left < right && nums[left] == nums[left - 1])
                        left++;
                }
            }
        }
        return n;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }

        List<List<Integer>> result = threeSum(nums);

        System.out.println("Triplets that sum up to zero are: ");
        for (List<Integer> triplet : result) {
```

```
            System.out.println(triplet);
        }

        sc.close();
    }
}
```

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

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.*;

class MinimumSizeSubarraySum {
    public static int minSubArrayLen(int target, int[] nums) {
        int minLength = Integer.MAX_VALUE;
        int left = 0;
        int currSum = 0;

        for (int right = 0; right < nums.length; right++) {
            currSum += nums[right];
            while (currSum >= target) {
                minLength = Math.min(minLength, right - left + 1);
                currSum -= nums[left];
                left++;
            }
        }
        return minLength != Integer.MAX_VALUE ? minLength : 0;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the target sum: ");
        int target = sc.nextInt();

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter " + n + " integers: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }

        int result = minSubArrayLen(target, nums);

        if (result > 0) {
            System.out.println("The minimal length of a subarray is: " + result);
        } else {
            System.out.println("No valid subarray found.");
        }

        sc.close();
    }
}
```

OUTPUT:

```
Enter the target sum:
7
Enter the size of the array:
6
Enter 6 integers:
2 3 1 2 4 3
The minimal length of a subarray is: 2
```

TIME COMPLEXITY: O(n)

7. LONGEST SUBSTRING WITHOUT REPEATING THE 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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;
```

```java
public class LengthofLongestSubstringWithoutRepeating {
    public static int lengthOfLongestSubstring(String s) {
        int left = 0;
        int maxLength = 0;
        HashSet<Character> newSet = new HashSet<>();

        for (int right = 0; right < s.length(); right++) {
            while (newSet.contains(s.charAt(right))) {
                newSet.remove(s.charAt(left));
                left++;
            }
            newSet.add(s.charAt(right));
            maxLength = Math.max(maxLength, right - left + 1);
        }
        return maxLength;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter a string: ");
        String s = sc.nextLine();

        int result = lengthOfLongestSubstring(s);

        System.out.println("The length of the longest substring without repeating
characters is: " + result);

        sc.close();
    }
}
```

OUTPUT:

```
Enter a string:
abacbbacccab
The length of the longest substring without repeating characters is: 3
```

TIME COMPLEXITY: O(n)

8. SUBSTRING WITH CONCATENATION

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.

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.*;

public class FindSubstringWithConcatenation {

    public static List<Integer> findSubstring(String s, String[] words) {
        List<Integer> indices = new ArrayList<>();
        if (s == null || words == null || words.length == 0) {
            return indices;
        }
```

```
        Map<String, Integer> wordCount = new HashMap<>();
        for (String word : words) {
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
        }

        int wordLength = words[0].length();
        int windowLength = words.length * wordLength;
        int n = s.length();

        for (int i = 0; i < wordLength; i++) {
            Map<String, Integer> currentCount = new HashMap<>();
            int left = i, right = i, totalWords = 0;

            while (right + wordLength <= n) {
                String word = s.substring(right, right + wordLength);
                right += wordLength;

                if (wordCount.containsKey(word)) {
                    currentCount.put(word, currentCount.getOrDefault(word, 0) + 1);
                    totalWords++;

                    while (currentCount.get(word) > wordCount.get(word)) {
                        String removedWord = s.substring(left, left + wordLength);
                        currentCount.put(removedWord, currentCount.get(removedWord) -
1);

                        totalWords--;
                        left += wordLength;
                    }

                    if (totalWords == words.length) {
                        indices.add(left);
                    }
                } else {
                    currentCount.clear();
                    totalWords = 0;
                    left = right;
                }
            }
        }

        return indices;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the string 's':");
        String s = scanner.nextLine();

        System.out.println("Enter the number of words:");
        int wordCount = scanner.nextInt();
        scanner.nextLine();

        String[] words = new String[wordCount];
        System.out.println("Enter the words:");
        for (int i = 0; i < wordCount; i++) {
            words[i] = scanner.nextLine();
        }
```

```
        List<Integer> result = findSubstring(s, words);

        if (result.isEmpty()) {
            System.out.println("No valid substring found.");
        } else {
            System.out.println("Starting indices of substrings: " + result);
        }

        scanner.close();
    }
}
```

OUTPUT:

```
Enter the string 's':
barfoothefoobarman
Enter the number of words:
2
Enter the words:
foo
bar
Starting indices of substrings: [0, 9]
```

TIME COMPLEXITY: O(n)

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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.Scanner;

public class MinimumWindowSubstring {

    public static String minWindow(String s, String t) {
        if (s == null || t == null || s.length() == 0 || t.length() == 0 || s.length()
< t.length()) {
            return "";
        }

        int[] map = new int[128];
        for (char c : t.toCharArray()) {
            map[c]++;
        }

        int start = 0, end = 0, count = t.length();
        int minLen = Integer.MAX_VALUE, startIndex = 0;
        char[] chs = s.toCharArray();

        while (end < chs.length) {
            if (map[chs[end]] > 0) {
                count--;
            }
            map[chs[end]]--;
            end++;

            while (count == 0) {
                if (end - start < minLen) {
                    startIndex = start;
                    minLen = end - start;
                }

                map[chs[start]]++;
                if (map[chs[start]] > 0) {
```

```
                    count++;
                }
                start++;
            }
        }

        return minLen == Integer.MAX_VALUE ? "" : s.substring(startIndex, startIndex +
minLen);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the string 's':");
        String s = scanner.nextLine();

        System.out.println("Enter the string 't':");
        String t = scanner.nextLine();

        String result = minWindow(s, t);

        if (result.isEmpty()) {
            System.out.println("No valid window exists.");
        } else {
            System.out.println("The minimum window substring is: " + result);
        }

        scanner.close();
    }
}
```

OUTPUT:

```
Enter the string 's':
ADOBECODEBANC
Enter the string 't':
ABC
The minimum window substring is: BANC
```

TIME COMPLEXITY: O(n)

10. VALID PARENTHESES

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

An input string is valid if:

Open brackets must be closed by the same type of brackets.

Open brackets must be closed in the correct order.

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 '()[]{}'.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class ValidParentheses {
    public static boolean isValid(String s) {
        Stack<Character> st = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(') {
                st.push(')');
            } else if (c == '{') {
                st.push('}');
            } else if (c == '[') {
                st.push(']');
            } else if (st.isEmpty() || st.pop() != c) {
                return false;
            }
        }
        return st.isEmpty();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter a string: ");
        String s = sc.nextLine();

        boolean result = isValid(s);
```

```
        if (result) {
            System.out.println("The string of parentheses is valid.");
        } else {
            System.out.println("The string of parentheses is not valid.");
        }

        sc.close();
    }
}
```

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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class SimplifyPath {
    public static String simplifyPath(String path) {
        Stack<String> st = new Stack<>();
        String[] direct = path.split("/");
```

```
        for (String d : direct) {
            if (d.equals(".") || d.isEmpty()) {
                continue;
            } else if (d.equals("..")) {
                if (!st.isEmpty()) {
                    st.pop();
                }
            } else {
                st.push(d);
            }
        }
        return "/" + String.join("/", st);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter a Unix-style file path: ");
        String path = sc.nextLine();

        String simplifiedPath = simplifyPath(path);

        System.out.println("Simplified Path: " + simplifiedPath);

        sc.close();
    }
}
```

OUTPUT:

TIME COMPLEXITY:

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.

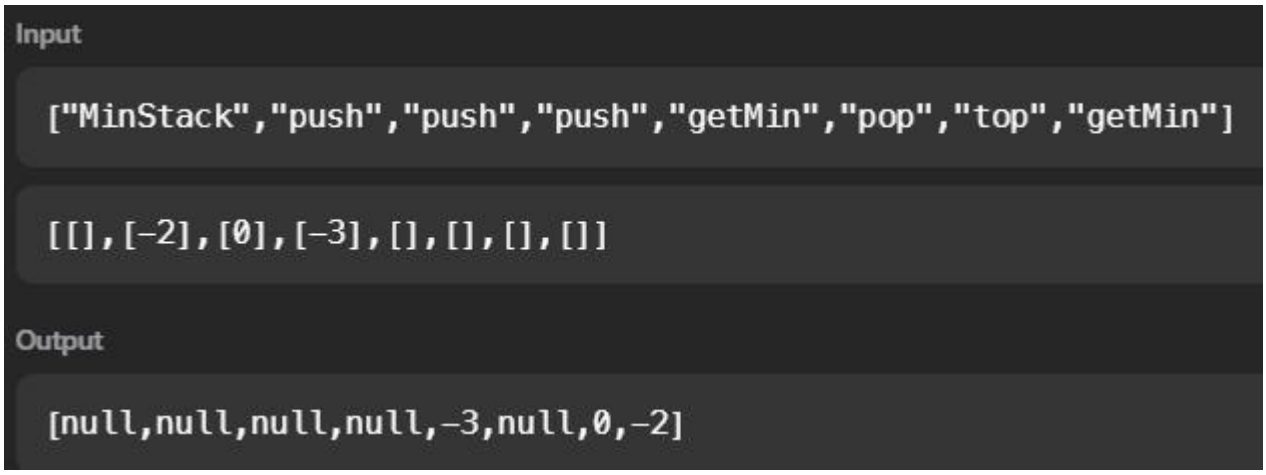
At most 3 * 104 calls will be made to push, pop, top, and getMin.

PROGRAM:

```
class MinStack {

    Stack<Integer> st;
    Stack<Integer> minSt;

    public MinStack() {
        st = new Stack<>();
        minSt = new Stack<>();
    }

    public void push(int val) {
        st.push(val);
        if(minSt.isEmpty() || val<=minSt.peek())
            minSt.push(val);
    }

    public void pop() {
        if(st.pop().equals(minSt.peek()))
            minSt.pop();
```

```
    }

    public int top() {
        return st.peek();
    }

    public int getMin() {
        return minSt.peek();
    }
}
```

OUTPUT:



TIME COMPLEXITY: O(1)

13. EVALUATE REVERSE POLISH

You are given an array of strings tokens that represents an arithmetic expression in a 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].

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.*;

public class RPN {
    public static int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();

        for (String s : tokens) {
            if (s.equals("+")) {
                stack.push(stack.pop() + stack.pop());
            } else if (s.equals("-")) {
                int second = stack.pop();
                int first = stack.pop();
                stack.push(first - second);
            } else if (s.equals("*")) {
                stack.push(stack.pop() * stack.pop());
```

```
            } else if (s.equals("/")) {
                int second = stack.pop();
                int first = stack.pop();
                stack.push(first / second);
            } else {
                stack.push(Integer.parseInt(s));
            }
        }

        return stack.peek();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the Reverse Polish Notation:");
        String input = sc.nextLine();

        String[] tokens = input.split("\\s+");

        int result = evalRPN(tokens);

        System.out.println("The result of the RPN expression is: " + result);

        sc.close();
    }
}
```

OUTPUT:

```
Enter the Reverse Polish Notation:
2 1 + 3 *
The result of the RPN expression is: 9
```

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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class BasicCalculator {
    public static int calculate(String s) {
        int res = 0;
        int operator = 1;
        int num = 0;
        Stack<Integer> st = new Stack<>();

        for (char c : s.toCharArray()) {
            if (Character.isDigit(c)) {
                num = num * 10 + (c - '0');
            } else if (c == '+') {
                res += operator * num;
                num = 0;
                operator = 1;
            } else if (c == '-') {
                res += operator * num;
                num = 0;
                operator = -1;
            } else if (c == '(') {
                st.push(res);
                st.push(operator);
                res = 0;
                operator = 1;
            } else if (c == ')') {
                res += operator * num;
                num = 0;
                res *= st.pop();
                res += st.pop();
            }
```

```
        }
        return res + (operator * num);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the string: ");
        String s = sc.nextLine();

        int result = calculate(s);

        System.out.println("The result of the calculation is: " + result);

        sc.close();
    }
}
```

OUTPUT:

```
Enter the string:
(1+(4+5+2)-3)+(6+8)
The result of the calculation is: 23
```

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

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class SearchInsertPosition {
    public static int searchInsert(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] > target) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }

        System.out.println("Enter the target value: ");
        int target = sc.nextInt();

        int result = searchInsert(nums, target);

        System.out.println("The target would be inserted at index: " + result);

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
4
Enter the elements:
1
3
6
9
Enter the target value:
4
The target would be inserted at index: 2
```

TIME COMPLEXITY: O(logn)

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:

1	3	5	7
10	11	16	20
23	30	34	60

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

Output: true

Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

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

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class SearchIn2DMatrix {
    public static boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length;
        int n = matrix[0].length;
        int i = 0;
        int j = n - 1;
        while (i < m && j >= 0) {
            if (matrix[i][j] == target) return true;
            if (matrix[i][j] > target) {
                j--;
            } else {
                i++;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of rows:");
        int m = sc.nextInt();
```

```
        System.out.println("Enter the number of columns:");
        int n = sc.nextInt();

        int[][] matrix = new int[m][n];
        System.out.println("Enter the elements of the matrix:");
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        System.out.println("Enter the target value:");
        int target = sc.nextInt();

        boolean found = searchMatrix(matrix, target);
        if (found) {
            System.out.println("The target " + target + " is found in the matrix.");
        } else {
            System.out.println("The target " + target + " is not found in the
matrix.");
        }

        sc.close();
    }
}
```

OUTPUT:

```
Enter the number of rows:
3
Enter the number of columns:
4
Enter the elements of the matrix:
1 3 5 7
10 11 16 20
23 30 34 60
Enter the target value:
3
The target 3 is found in the matrix.
```

TIME COMPLEXITY: O(log(m * 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.

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.*;

public class PeakElement {
    public static 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;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];
```

```
        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }

        int peakIndex = findPeakElement(nums);

        System.out.println("A peak element is at index: " + peakIndex);

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
4
Enter the elements:
1 2 3 1
A peak element is at index: 2
```

TIME COMPLEXITY: O(logn)

18. SEARCH IN ROTATED SORTED

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

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.*;

public class SearchInARotatedSortedArray {
    public static int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;

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

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

            if (nums[left] <= nums[mid]) {
                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;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
```

```
        System.out.println("Enter the target value: ");
        int target = sc.nextInt();

        int result = search(nums, target);

        if (result != -1) {
            System.out.println("The target is at index: " + result);
        } else {
            System.out.println("The target is not in the array.");
        }

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
7
Enter the elements:
4 5 6 7 0 1 2
Enter the target value:
0
The target is at index: 4
```

TIME COMPLEXITY: O(logn)

19. FIND FIRST AND LAST POSITION

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]

PROGRAM:

```java
package dsaPracticeProblems;
import java.util.Scanner;

public class SubstringWithConcatenation {
    private static 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;
      }

    public static int[] searchRange(int[] nums, int target) {
        int[] result = { -1, -1 };
        if (nums.length == 0) {
            return result;
        }

        int left = binarySearch(nums, target, true);
        int right = binarySearch(nums, target, false);

        result[0] = left;
        result[1] = right;
        return result;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
```

```
        System.out.println("Enter the target value: ");
        int target = sc.nextInt();

        int[] result = searchRange(nums, target);

        System.out.println("The range of the target is: [" + result[0] + ", " +
result[1] + "]");

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
6
Enter the elements:
5 7 7 8 8 10
Enter the target value:
8
The range of the target is: [3, 4]
```

TIME COMPLEXITY: O(logn)

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.

PROGRAM:

```
package dsaPracticeProblems;
import java.util.*;

public class MinimumInRotatedSortedArray {
    public static int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = (left + right) / 2;

            if (nums[mid] <= nums[right]) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return nums[left];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
```

```
        int result = findMin(nums);

        System.out.println("The minimum element in the array is: " + result);

        sc.close();
    }
}
```

OUTPUT:

```
Enter the size of the array:
5
Enter the elements:
3 4 5 1 2
The minimum element in the array is: 1
```

TIME COMPLEXITY: O(logn)