DSA PRACTICE -9(21.09.2024)

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.

Solution:

```
import java.util.Scanner;


class Validpalindrome {
    public boolean isPalindrome(String s) {
        s = s.toLowerCase().replaceAll("[^a-zA-Z0-9]", "");
        StringBuilder str = new StringBuilder(s);
        str.reverse();
        System.out.println("Reversed string: " + str);
        return str.toString().equals(s);
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string to check if it is a palindrome:");
        String input = sc.nextLine();
        Validpalindrome solution = new Validpalindrome();
        boolean result = solution.isPalindrome(input);
        System.out.println("Is the string a palindrome? " + result);
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac Validpalindrome.java

F:\DSA PRACTICE\day 9>java Validpalindrome
Enter a string to check if it is a palindrome:
abba
Reversed string: abba
Is the string a palindrome? true
```

Timecomplexity:O(n)

Spacecomplexity: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 <= 10^4
- s and t consist only of lowercase English letters.

Solution:

```
import java.util.Scanner;
class Subsequence {
   public boolean isSubsequence(String s, String t) {
      int i = 0;
      int 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.print("Enter string s: ");
      String s = sc.nextLine();
      System.out.print("Enter string t: ");
      String t = sc.nextLine();
       Subsequence solution = new  Subsequence();
      boolean result = solution.isSubsequence(s, t);
      System.out.println("Is '" + s + "' a subsequence of '" + t + "'? " + result);
   }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac  Subsequence.java

F:\DSA PRACTICE\day 9>java  Subsequence
Enter string s: abbbdkdcdk
Enter string t: abbeje
Is 'abbbdkdcdk' a subsequence of 'abbeje'? false
```

Timecomplexity:O(max(s.length(),t.length()))

Spacecomplexity:O(1)

3.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[$index_1$] and numbers[$index_2$] where 1 <= $index_1$ < $index_2$ <= numbers.length.

Return *the indices of the two numbers,* $index_1$ *and* $index_2$*, added by one as an integer array* [$index_1$, $index_2$] *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, $index_1$ = 1, $index_2$ = 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 $index_1$ = 1, $index_2$ = 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 $index_1$ = 1, $index_2$ = 2. We return [1, 2].

Constraints:

- 2 <= numbers.length <= $3 * 10^4$
- -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.

Solution:

```
import java.util.HashMap;
import java.util.Scanner;
class TwoSum{
    public int[] twoSum(int[] numbers, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < numbers.length; i++) {
            int complement = target - numbers[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement) + 1, i + 1 };
            }
            map.put(numbers[i], i);
        }
        return null;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] numbers = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            numbers[i] = sc.nextInt();
```

```
        }
        System.out.print("Enter the target value: ");
        int target = sc.nextInt();
        TwoSum sol = new TwoSum();
        int[] result = sol.twoSum(numbers, target);
        if (result != null) {
            System.out.println("Indices of numbers adding to target: " + result[0] + ", " +
result[1]);
        } else {
            System.out.println("No solution found.");
        }
        sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac TwoSum.java

F:\DSA PRACTICE\day 9>java TwoSum
Enter the size of the array: 4
Enter the elements of the array:
2 3 1 5
Enter the target value: 6
Indices of numbers adding to target: 3, 4
```

Timecomplexity:O(n)

Spacecomplexity: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 i^{th} 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

Solution:

```
import java.util.Scanner;
class Containerwithmostwater {
    public int maxArea(int[] height) {
        int i = 0;
        int j = height.length - 1;
        int ans = 0;
        while (i < j) {
            int h = Math.min(height[i], height[j]);
            int b = j - i;
            ans = Math.max(ans, h * b);
            if (height[i] < height[j]) {
                i++;
            } else {
                j--;
            }
        }
        return ans;
    }
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] height = new int[n];
        System.out.println("Enter the heights: ");
        for (int i = 0; i < n; i++) {
            height[i] = sc.nextInt();
        }
        Containerwithmostwater sol = new Containerwithmostwater();
        int result = sol.maxArea(height);
        System.out.println("Maximum area of water container: " + result);
        sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac Containerwithmostwater.java

F:\DSA PRACTICE\day 9>java Containerwithmostwater
Enter the size of the array: 5
Enter the heights:
1 4 5 6 4
Maximum area of water container: 12
```

Timecomplexity:O(N)

Spacecomplexity:O(1)

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
- -10^5 <= nums[i] <= 10^5

Solution:

```
import java.util.*;
class ThreeSum {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums); // Sort the array
        Set<List<Integer>> arr = new HashSet<>();
        for (int i = 0; i < nums.length - 2; i++) {
            int j = i + 1;
            int k = nums.length - 1;
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k];
```

```
            if (sum == 0) {
                arr.add(Arrays.asList(nums[i], nums[j], nums[k]));
                j++;
            } else if (sum < 0) {
                j++;
            } else {
                k--;
            }
        }
    }
    return new ArrayList<>(arr);
}
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the size of the array: ");
    int n = sc.nextInt();
    int[] nums = new int[n];
    System.out.println("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        nums[i] = sc.nextInt();
    }
    ThreeSum sol = new ThreeSum();
    List<List<Integer>> result = sol.threeSum(nums);
    if (!result.isEmpty()) {
        System.out.println("Unique triplets with sum 0:");
        for (List<Integer> triplet : result) {
            System.out.println(triplet);
        }
    }
```

```
        else {
            System.out.println("No triplets found.");
        }
        sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac  ThreeSum.java

F:\DSA PRACTICE\day 9>java  ThreeSum
Enter the size of the array: 5
Enter the elements of the array:
3 4 6 7 2
No triplets found.
```

TimeComplexity : O(n^2)

Spacecomplexity: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:

- $1 <= target <= 10^9$
- $1 <= nums.length <= 10^5$
- $1 <= nums[i] <= 10^4$

Solution:

```
import java.util.Scanner;
class MinimumSize {
    public int minSubArrayLen(int target, int[] nums) {
        int i = 0;
        int j = 0;
        int sum = 0;
        int minLength = Integer.MAX_VALUE;
        while (j < nums.length) {
            sum += nums[j];
            j++;
            while (sum >= target) {
                minLength = Math.min(minLength, j - i);
                sum -= nums[i];
                i++;
            }
        }
        return (minLength == Integer.MAX_VALUE) ? 0 : minLength;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the target value: ");
        int target = sc.nextInt();
```

```
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] nums = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
        MinimumSize sol = new MinimumSize();
        int result = sol.minSubArrayLen(target, nums);
        if (result == 0) {
            System.out.println("No valid subarray found.");
        } else {
            System.out.println("Minimum length of subarray with sum >= target: " + result);
        }
        sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac MinimumSize.java

F:\DSA PRACTICE\day 9>java MinimumSize
Enter the target value: 7
Enter the size of the array: 6
Enter the elements of the array:
2 3 1 2 4 3
Minimum length of subarray with sum >= target: 2
```

Timecomplexity:O(n)

Spacecomplexity:O(1)

7. Longest Substring Without Repeating Characters

Given a string s, find the length of the longest substringwithout 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 <= $5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

Solution:

```
import java.util.HashSet;
import java.util.Scanner;
class LongestSubstring {
   public int lengthOfLongestSubstring(String s) {
      int i = 0;
      int max = 0;
      HashSet<Character> c = new HashSet<>();
      for (int j = 0; j < s.length(); j++) {
         while (c.contains(s.charAt(j))) {
            c.remove(s.charAt(i));
            i++;
         }
```

```
            c.add(s.charAt(j));
            max = Math.max(max, j - i + 1);
        }
        return max;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String s = sc.nextLine();
        LongestSubstring sol = new LongestSubstring();
        int result = sol.lengthOfLongestSubstring(s);
        System.out.println("Length of the longest substring without repeating characters: " +
result);
        sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac LongestSubstring.java

F:\DSA PRACTICE\day 9>java LongestSubstring
Enter a string: abbcabdef
Length of the longest substring without repeating characters: 6
```

Timecomplexity:O(n)

Spacecomplexity:O(m)

8. Valid Parentheses

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 <= 10^4
- s consists of parentheses only '()[]{}'.

Solution:

```
import java.util.Scanner;
import java.util.Stack;
class ValidParenthesis{
   public boolean isValid(String s) {
      Stack<Character> st = new Stack<>();
      for (int i = 0; i < s.length(); i++) {
         char ch = s.charAt(i);
         if (ch == '(' || ch == '{' || ch == '[') {
            st.push(ch);
         }
         else {
```

```
            if (st.isEmpty()) {
                return false;
            } else if ((ch == ')' && st.peek() == '(') ||
                    (ch == '}' && st.peek() == '{') ||
                    (ch == ']' && st.peek() == '[')) {
                st.pop();
            } else {
                return false;
            }
          }
       }
       return st.isEmpty();
    }
    public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       System.out.print("Enter a string of brackets: ");
       String input = sc.nextLine();
       ValidParenthesis sol = new  ValidParenthesis();
       boolean isValid = sol.isValid(input);
       if (isValid) {
          System.out.println("The string is valid.");
       } else {
          System.out.println("The string is not valid.");
       }
       sc.close();
    }
}
```

Output:

```
F:\DSA PRACTICE\day 9>javac  ValidParenthesis.java

F:\DSA PRACTICE\day 9>java  ValidParenthesis
Enter a string of brackets: (({{))
The string is not valid.
```

Timecomplexity:O(n)

Spacecomplexity:O(n)