**CSE231:COMPETITIVE PROGRAMMING**

L: 0 T: 0 P: 2

Course Weightage - **ATT: 5 CAP: 45 ETP: 50**

Reference Books

1. **DATA STRUCTURES AND ALGORITHMS : CONCEPTS, TECHNIQUES AND APPLICATIONS** by ***G. A. V. PAI*** & Published by ***Mc Graw Hill Education***
2. **CRACKING THE CODING INTERVIEW** by ***GAYLE LAAKMANN MCDOWELL*** & Published by  ***CAREERCUP***

CA Grading:

1. Best 2 of 3 CAP (CA practicals) online coding tests (each of 50 Marks)

Reference websites for practice:

1. Codechef.com
2. tutorialhorizon.com/algorithms/
3. leetcode.com

Programming Language Preferences:

1. cpp (most popular, fastest)
2. Java

**! Important Note:**

1. When writing an algo, try to minimise the usage of inbuilt methods/functions as much as possible where your function is directly or within a limited step solving the whole problem.
2. We generally prefer minimizing time over space complexity i.e., tarde-off space with time unless otherwise stated

## **Practicals for Day1/Week1**

1. String Reversal - Reversing a string means arranging its characters in the opposite order.

How to do it? If given a string, construct a new string and add characters to it from the back of the original string

Or, convert the string into char array and then swap the first & last, 2nd & 2nd last and so on till mid of that array. Once done, convert back to string.

Or, recursively concatenate the chars from the back

Possible Solutions in Java

(1)

public class StringReversal {

public static String reverseStringIterative(String s) {

StringBuilder reversed = new StringBuilder();

for (int i = s.length() - 1; i >= 0; i--) {

reversed.append(s.charAt(i));

}

return reversed.toString();

}

public static void main(String[] args) {

String input = "Hello, World!";

String reversed = reverseStringIterative(input);

System.out.println("Original: " + input);

System.out.println("Reversed: " + reversed);

}

}

(2)

public class StringReversal {

public static String reverseStringWithCharArray(String s) {

char[] charArray = s.toCharArray();

int length = charArray.length;

// Create a new array to store reversed characters

char[] reversedArray = new char[length];

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

reversedArray[i] = charArray[length - 1 - i];

}

return new String(reversedArray);

}

public static void main(String[] args) {

String input = "Hello, World!";

String reversed = reverseStringWithCharArray(input);

System.out.println("Original: " + input);

System.out.println("Reversed: " + reversed);

}

}

(3)

public class StringReversal {

public static String reverseStringInPlace(String s) {

char[] charArray = s.toCharArray();

int start = 0;

int end = charArray.length - 1;

// Swap characters from start to end

while (start <= end) {

char temp = charArray[start];

charArray[start] = charArray[end];

charArray[end] = temp;

start++;

end--;

}

return new String(charArray);

}

public static void main(String[] args) {

String input = "Hello, World!";

String reversed = reverseStringInPlace(input);

System.out.println("Original: " + input);

System.out.println("Reversed: " + reversed);

}

}

(4)

public class StringReversal {

public static String reverseStringRecursive(String s) {

if (s.isEmpty()) {

return s;

}

return reverseStringRecursive(s.substring(1)) + s.charAt(0);

}

public static void main(String[] args) {

String input = "Hello, World!";

String reversed = reverseStringRecursive(input);

System.out.println("Original: " + input);

System.out.println("Reversed: " + reversed);

}

}

1. Anagram Check: An anagram is a word or phrase formed by rearranging the letters of another. To check if two strings are anagrams, we need to ensure that they contain the same characters with the same frequencies, regardless of the order.

(1) Using sorting the both strings

import java.util.Arrays;

public class AnagramCheck {

public static boolean areAnagramsSorting(String s1, String s2) {

// Check if lengths are different

if (s1.length() != s2.length()) {

return false;

}

// Convert strings to character arrays and sort them

char[] charArray1 = s1.toCharArray();

char[] charArray2 = s2.toCharArray();

Arrays.sort(charArray1);

Arrays.sort(charArray2);

// Compare sorted arrays

return Arrays.equals(charArray1, charArray2);

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

System.out.println("Are \"" + str1 + "\" and \"" + str2 + "\" anagrams? "

+ areAnagramsSorting(str1, str2));

}

}

(2) Using frequency count (hashmap)

import java.util.HashMap;

import java.util.Map;

public class AnagramCheck {

public static boolean areAnagramsFrequencyCount(String s1, String s2) {

// Check if lengths are different

if (s1.length() != s2.length()) {

return false;

}

// Count character frequencies using a map

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

// Increment frequencies for characters in the first string

for (char c : s1.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

// Decrement frequencies for characters in the second string

for (char c : s2.toCharArray()) {

int count = frequencyMap.getOrDefault(c, 0);

if (count == 0) {

// Character not found in the first string

return false;

}

frequencyMap.put(c, count - 1);

}

// All frequencies should be zero for anagrams

return frequencyMap.values().stream().allMatch(count -> count == 0);

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

System.out.println("Are \"" + str1 + "\" and \"" + str2 + "\" anagrams? "

+ areAnagramsFrequencyCount(str1, str2));

}

}

(3) Using Char array

public class AnagramCheck {

public static boolean areAnagramsCountDifference(String s1, String s2) {

// Check if lengths are different

if (s1.length() != s2.length()) {

return false;

}

// Use an array to store character counts

int[] charCount = new int[26]; // Assuming only lowercase alphabets

// Increment counts for characters in the first string

for (char c : s1.toCharArray()) {

charCount[c - 'a']++;

}

// Decrement counts for characters in the second string

for (char c : s2.toCharArray()) {

charCount[c - 'a']--;

}

// Check if all counts are zero

for (int count : charCount) {

if (count != 0) {

return false;

}

}

return true;

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

System.out.println("Are \"" + str1 + "\" and \"" + str2 + "\" anagrams? "

+ areAnagramsCountDifference(str1, str2));

}

}

(4) Using Xor

public class AnagramCheck {

public static boolean areAnagramsXOR(String s1, String s2) {

// Check if lengths are different

if (s1.length() != s2.length()) {

return false;

}

int xorResult = 0;

// XOR all characters in both strings

for (char c : s1.toCharArray()) {

xorResult ^= c;

}

for (char c : s2.toCharArray()) {

xorResult ^= c;

}

// If both strings are anagrams, the XOR result should be zero

return xorResult == 0;

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

System.out.println("Are \"" + str1 + "\" and \"" + str2 + "\" anagrams? "

+ areAnagramsXOR(str1, str2));

}

}

1. Longest Substring Without Repeating Characters - The problem involves finding a contiguous substring without any repeating characters. As we iterate through the string, we need to keep track of the characters in the current substring and update the maximum length whenever we encounter a repeating character.

import java.util.HashMap;

import java.util.Map;

public class Solution {

public int lengthOfLongestSubstring(String s) {

if (s.length() < 2) {

return s.length();

}

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

int start = 0, maxLength = 0;

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

if (used.containsKey(s.charAt(i))) {

start = Math.max(start, used.get(s.charAt(i)) + 1);

}

maxLength = Math.max(maxLength, i - start + 1);

used.put(s.charAt(i), i);

}

return maxLength;

}

public static void main(String[] args) {

Solution solution = new Solution();

String input = "abcabcbb";

int result = solution.lengthOfLongestSubstring(input);

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

}

}

1. Reverse the individual words of the string - Convert the string to a char array for in-place manipulation. Iterate through each character and identify word boundaries (spaces). Reverse each individual word by using the reverse method, which takes the start and end indices of the word.

public class ReverseWords {

public static String reverseWords(String s) {

char[] chars = s.toCharArray();

int start = 0;

for (int end = 0; end < chars.length; end++) {

if (chars[end] == ' ') {

reverse(chars, start, end - 1);

start = end + 1;

}

}

// Reverse the last word

reverse(chars, start, chars.length - 1);

return new String(chars);

}

private static void reverse(char[] chars, int start, int end) {

while (start < end) {

char temp = chars[start];

chars[start] = chars[end];

chars[end] = temp;

start++;

end--;

}

}

public static void main(String[] args) {

String input = "Hello World";

System.out.println("Original: " + input);

System.out.println("Reversed: " + reverseWords(input));

}

}

## **Practicals for Day2/Week2**

1. Decimal number to Roman Numeral The Roman numeral system traditionally uses a limited set of symbols to represent numbers. The system is based on certain combinations of these symbols, and there are no specific symbols for numbers beyond a certain range. The largest numeral that can be represented using the traditional symbols is 3,999. Iterate through a predefined list of Roman numeral symbols and their corresponding values. Repeatedly append the Roman numeral symbol to the result string as long as the input decimal number is greater than or equal to the current value. Then subtract the value from the input number and proceed to the next symbol. Continue this process until the input number becomes zero. Finally, the result string containing the Roman numeral representation of the input decimal number is returned

**public class DecimalToRoman {**

**public static String intToRoman(int num) {**

**if (num <= 0 || num > 3999) {**

**return "Invalid input";**

**}**

**String[] romanSymbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};**

**int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};**

**StringBuilder result = new StringBuilder();**

**for (int i = 0; i < values.length; i++) {**

**while (num >= values[i]) {**

**result.append(romanSymbols[i]);**

**num -= values[i];**

**}**

**}**

**return result.toString();**

**}**

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

**int decimalNumber = 3549;**

**System.out.println("Roman numeral for " + decimalNumber + ": " + intToRoman(decimalNumber));**

**}**

**}**

1. Implement a stack using one queue Enqueue elements in reverse order when pushing. When pushing an element, first add it to the end of the queue and then move all existing elements one by one to the end, effectively placing the newly pushed element at the front. When popping or accessing the top element, simply dequeue from the front of the queue.

**import java.util.LinkedList;**

**import java.util.Queue;**

**public class StackUsingQueue {**

**private Queue<Integer> queue;**

**public StackUsingQueue() {**

**queue = new LinkedList<>();**

**}**

**public void push(int x) {**

**queue.add(x);**

**int size = queue.size();**

**while (size > 1) {**

**queue.add(queue.remove());**

**size--;**

**}**

**}**

**public int pop() {**

**return queue.remove();**

**}**

**public int top() {**

**return queue.peek();**

**}**

**public boolean empty() {**

**return queue.isEmpty();**

**}**

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

**StackUsingQueue stack = new StackUsingQueue();**

**stack.push(1);**

**stack.push(2);**

**stack.push(3);**

**System.out.println("Top element: " + stack.top());**

**System.out.println("Popped element: " + stack.pop());**

**System.out.println("Is stack empty? " + stack.empty());**

**}**

**}**