Q1.public class Solution {

public int minimumDeleteSum(String s1, String s2) {

int m = s1.length();

int n = s2.length();

int[][] dp = new int[m+1][n+1];

// Initialize the first row and first column

for (int i = 1; i <= m; i++) {

dp[i][0] = dp[i-1][0] + s1.charAt(i-1);

}

for (int j = 1; j <= n; j++) {

dp[0][j] = dp[0][j-1] + s2.charAt(j-1);

}

// Fill the dp array

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (s1.charAt(i-1) == s2.charAt(j-1)) {

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

} else {

dp[i][j] = Math.min(dp[i-1][j] + s1.charAt(i-1), dp[i][j-1] + s2.charAt(j-1));

}

}

}

return dp[m][n];

}

}

Q2.import java.util.Stack;

public class Solution {

public boolean checkValidString(String s) {

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

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

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

char c = s.charAt(i);

if (c == '(') {

stack.push(i);

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

starStack.push(i);

} else {

if (!stack.isEmpty()) {

stack.pop();

} else if (!starStack.isEmpty()) {

starStack.pop();

} else {

return false;

}

}

}

while (!stack.isEmpty() && !starStack.isEmpty()) {

if (stack.pop() > starStack.pop()) {

return false;

}

}

return stack.isEmpty();

}

}

Q3.public class Solution {

public int minDistance(String word1, String word2) {

int m = word1.length();

int n = word2.length();

// Create a 2D array to store the lengths of the LCS

int[][] dp = new int[m + 1][n + 1];

// Fill the DP array

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (word1.charAt(i - 1) == word2.charAt(j - 1)) {

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

} else {

dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

}

}

}

// Calculate the minimum number of steps

int lcsLength = dp[m][n];

int steps = m - lcsLength + n - lcsLength;

return steps;

}

}

Q4.class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

private int index = 0; // global index to keep track of current position in the string

public TreeNode str2tree(String s) {

if (s.isEmpty()) {

return null;

}

int num = getNum(s); // get the integer value at the current position

TreeNode root = new TreeNode(num); // create a new tree node with the value

// check if there is a left child

if (index < s.length() && s.charAt(index) == '(') {

index++; // move to the next position

root.left = str2tree(s); // recursively construct the left subtree

}

// check if there is a right child

if (index < s.length() && s.charAt(index) == '(') {

index++; // move to the next position

root.right = str2tree(s); // recursively construct the right subtree

}

index++; // move past the closing parenthesis

return root;

}

private int getNum(String s) {

int start = index;

// check if the number is negative

if (s.charAt(index) == '-') {

index++;

}

// find the end of the number

while (index < s.length() && Character.isDigit(s.charAt(index))) {

index++;

}

return Integer.parseInt(s.substring(start, index));

}

}

Q5.public int compress(char[] chars) {

int readPtr = 0; // pointer for reading the original array

int writePtr = 0; // pointer for writing the compressed characters

int count = 1; // count of consecutive repeating characters

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

// check if the current character is equal to the previous one

if (i < chars.length && chars[i] == chars[i - 1]) {

count++;

} else {

chars[writePtr++] = chars[readPtr]; // write the current character

// write the count if it is greater than 1

if (count > 1) {

String countStr = Integer.toString(count);

// write each digit of the count separately

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

chars[writePtr++] = c;

}

}

readPtr = i; // move the read pointer to the next character

count = 1; // reset the count

}

}

return writePtr; // return the new length of the array

}

Q6.import java.util.ArrayList;

import java.util.List;

public List<Integer> findAnagrams(String s, String p) {

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

if (s.length() < p.length()) {

return result; // empty result if s is shorter than p

}

int[] pCount = new int[26]; // array to store character count of p

int[] sCount = new int[26]; // array to store current window character count of s

// Count the characters in p

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

pCount[c - 'a']++;

}

int left = 0; // left pointer of the window

// Iterate over s using the right pointer

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

// Increment the count of the current character in the window

sCount[s.charAt(right) - 'a']++;

// Shrink the window if its size is greater than p's length

if (right - left + 1 > p.length()) {

sCount[s.charAt(left) - 'a']--; // decrement the count of the character at the left end

left++; // move the left pointer to the right

}

// Check if the counts of characters in pCount and sCount are equal

if (right - left + 1 == p.length() && isEqual(pCount, sCount)) {

result.add(left); // add the index of the start of the anagram window to the result

}

}

return result;

}

// Function to check if two arrays are equal

private boolean isEqual(int[] arr1, int[] arr2) {

for (int i = 0; i < arr1.length; i++) {

if (arr1[i] != arr2[i]) {

return false;

}

}

return true;

}

Q7.import java.util.Stack;

public String decodeString(String s) {

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

Stack<StringBuilder> stringStack = new Stack<>();

StringBuilder currentString = new StringBuilder();

int count = 0;

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

if (Character.isDigit(c)) {

count = count \* 10 + (c - '0');

} else if (c == '[') {

countStack.push(count);

stringStack.push(currentString);

currentString = new StringBuilder();

count = 0;

} else if (c == ']') {

StringBuilder decodedString = stringStack.pop();

int repeatCount = countStack.pop();

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

decodedString.append(currentString);

}

currentString = decodedString;

} else {

currentString.append(c);

}

}

return currentString.toString();

}

Q8.public boolean buddyStrings(String s, String goal) {

if (s.length() != goal.length()) {

return false;

}

if (s.equals(goal)) {

// Check if there are any repeated characters in s

int[] count = new int[26];

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

count[c - 'a']++;

if (count[c - 'a'] > 1) {

return true;

}

}

return false;

}

int first = -1; // Position of first mismatched character

int second = -1; // Position of second mismatched character

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

if (s.charAt(i) != goal.charAt(i)) {

if (first == -1) {

first = i;

} else if (second == -1) {

second = i;

} else {

// More than two characters are different, can't perform the swap

return false;

}

}

}

return (second != -1 && s.charAt(first) == goal.charAt(second)

&& s.charAt(second) == goal.charAt(first));

}