String

1.Reverse a String

You are given a string s, and your task is to reverse the string.

Examples:

Input: s = "Geeks"
Output: "skeeG"

Input: s = "for"
Output: "rof"

Input: s = "a"
Output: "a"

Constraints:

```java
class Solution {
    public static String reverseString(String s) {
        // code here
        char []ch=s.toCharArray();
        int l=0;
        int h=ch.length-1;
        while(l<h)
        {
            char temp=ch[l];
            ch[l]=ch[h];
            ch[h]=temp;
            l++;
            h--;

        }
        return new String(ch);
    }
}
```

```java
//User function Template for Java


class Reverse
{
    // Complete the function
    // str: input string
    public static String reverseWord(String str)
    {
        // Reverse the string str
        StringBuilder sb=new StringBuilder(str);

        return sb.reverse().toString();
    }
}
```

2.Palindrome String

You are given a string **s**. Your task is to determine if the string is a palindrome. A string is considered a palindrome if it reads the same forwards and backwards.

Examples :

Input: s = "abba"
Output: true
Explanation: "abba" reads the same forwards and backwards, so it is a palindrome.

Input: s = "abc"
Output: false
Explanation: "abc" does not read the same forwards and backwards, so it is not a palindrome.

Input: s = "a"
Output: true
Explanation: A single-character string is always a palindrome.

```
class Solution {
  public:
    // Function to check if a string is a palindrome.
    bool isPalindrome(string& s) {
      int i=0;
      int j=s.length()-1;
      while(i<=j){
        if(s[i]!=s[j]){
          return false;
        }
        i++;
        j--;
      }
      return true;
    }
};
```

```
class Solution {
    boolean isPalindrome(String s) {
        // code here
        String str = s.toLowerCase();
        StringBuilder rev = new StringBuilder(str);
        rev.reverse();
        if(str.equals(rev.toString())){
            return true;
        }
        else{
            return false;
        }
    }
};
```

3.Print all duplicate character in a string

Given a string **S**, the task is to print all the duplicate characters with their occurrences in the given string.

Example:

Input: S = "geeksforgeeks"
Output:
e, count = 4
g, count = 2
k, count = 2
s, count = 2

```
static void printDups(String str)
{
  Map<Character, Integer> count = new HashMap<>();
  for (int i = 0; i < str.length(); i++) {
    if(count.containsKey(str.charAt(i)))
      count.put(str.charAt(i) , count.get(str.charAt(i))+1);
    else count.put(str.charAt(i),1);
    //increase the count of characters by 1
  }

  for (Map.Entry<Character,Integer> mapElement : count.entrySet()) {
//iterating through the unordered map
    if (mapElement.getValue() > 1)   //if the count of characters is
greater than 1 then duplicate found
      System.out.println(mapElement.getKey() + ", count = " +
mapElement.getValue());
  }
}
```

4.String rotation of each other

You are given two strings of equal lengths, **s1** and **s2**. The task is to check if **s2** is a rotated version of the string **s1**.

Note: The characters in the strings are in lowercase.

Examples :

Input: s1 = "abcd", s2 = "cdab"
Output: true
Explanation: After 2 right rotations, s1 will become equal to s2.

Input: s1 = "aab", s2 = "aba"
Output: true
Explanation: *After 1 left rotation, s1 will become equal to s2.*

Input: s1 = "abcd", s2 = "acbd"
Output: false
Explanation: Strings are not rotations of each other.

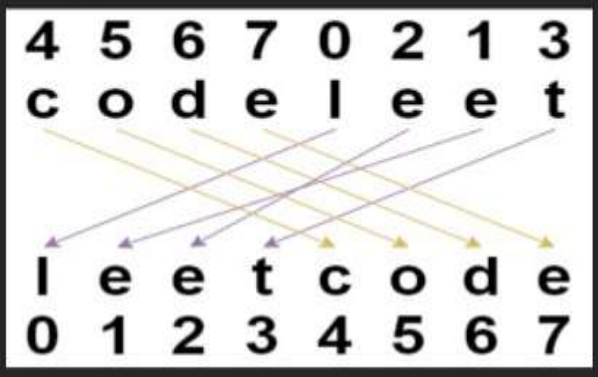
```
class Solution {
    // Function to check if two strings are rotations of each other or not.
    public static boolean areRotations(String s1, String s2) {
        // Your code here
        String str=s1+s1;
        int n=s1.length();
        int m=s2.length();
        int i=0,j=0;

        while(i<str.length() && j<m){
            if(str.charAt(i)==s2.charAt(j)){
                i++;
                j++;
            }else{
                i++;
            }
        }
        return (j==m);
    }
}
```

5.Shuffle String

You are given a string `s` and an integer array `indices` of the **same length**. The string `s` will be shuffled such that the character at the i^{th} position moves to `indices[i]` in the shuffled string.

Return *the shuffled string*.

Example 1:




```
Input: s = "codeleet", indices = [4,5,6,7,0,2,1,3]
Output: "leetcode"
Explanation: As shown, "codeleet" becomes "leetcode" after shuffling.
```

Example 2:

```
Input: s = "abc", indices = [0,1,2]
Output: "abc"
Explanation: After shuffling, each character remains in its position.
```

```java
class Solution {
    public String restoreString(String s, int[] indices) {

        int n=s.length();
        char arr[]=new char[n];

        for(int i=0;i<n;i++){
            arr[indices[i]]=s.charAt(i);
        }
        String res=String.valueOf(arr);
        return res;

    }
}
```

6.Longest Palindrome in a string

Given a string **s**, your task is to find the longest palindromic substring within s.

A **substring** is a **contiguous** sequence of characters within a string, defined as s[i...j] where $0 \leq i \leq j < \text{len}(s)$.

A **palindrome** is a string that reads the **same** forward and backward. More formally, s is a palindrome if reverse(s) == s.

Note: If there are multiple palindromic substrings with the same length, return the **first occurrence** of the longest palindromic substring from left to right.

Examples :

Input: s = "forgeeksskeegfor"
Output: "geeksskeeg"
Explanation: There are several possible palindromic substrings like "kssk", "ss", "eeksskee" etc. But the substring "geeksskeeg" is the longest among all.

Input: s = "Geeks"
Output: "ee"
Explanation: "ee" is the longest palindromic substring of "Geeks".

Input: s = "abc"
Output: "a"
Explanation: "a", "b" and "c" are longest palindromic substrings of same length. So, the first occurrence is returned.

```java
class Solution {
     static String expandAround(String s,int i,int j){
        while(i>=0 && j<s.length() && s.charAt(i)==s.charAt(j)){
            i--;
            j++;
        }
        return s.substring(i+1,j);
    }
    static String longestPalindrome(String s) {
        // code here
        if(s.length()==0||s==null)return "";
        String longestPalin="";
        for(int i=0;i<s.length();i++){
            String pal1=expandAround(s,i,i+1);
            if(pal1.length()>longestPalin.length()){
                longestPalin=pal1;
            }
            String pal2=expandAround(s,i,i);
            if(pal2.length()>longestPalin.length()){
                longestPalin=pal2;
            }
        }
        return longestPalin;
    }
}
```

7.Longest Repeating Subsequence

Given string str, find the length of the longest repeating subsequence such that it can be found twice in the given string.

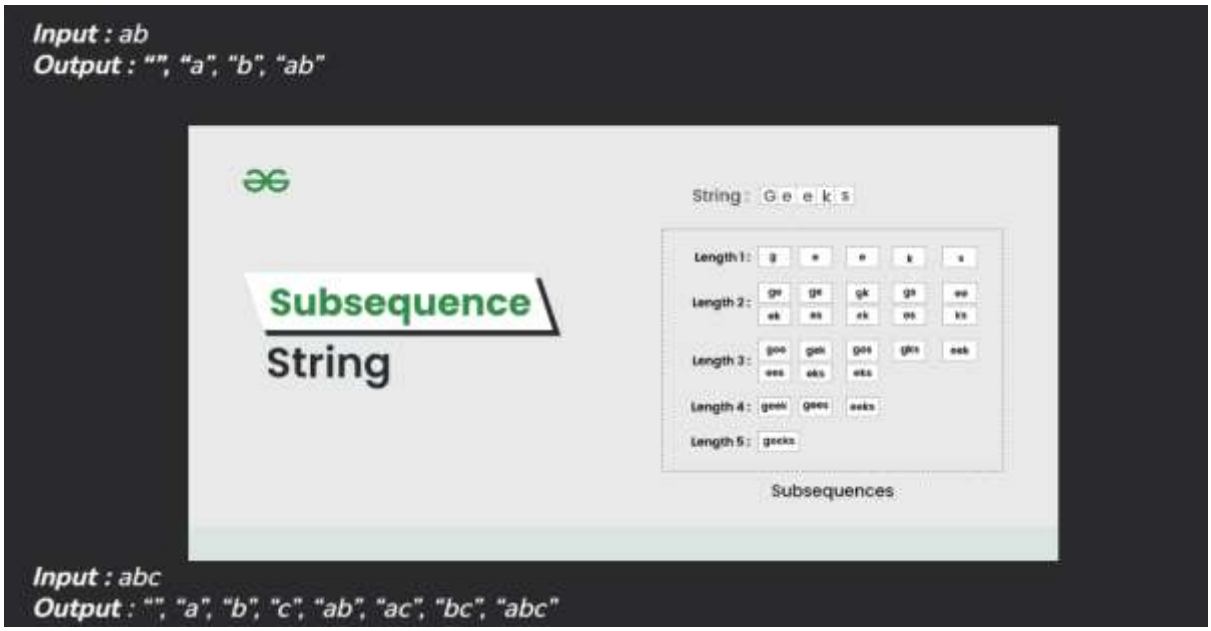
The two identified subsequences A and B can use the same ith character from string **s** if and only if that ith character has different indices in A and B. For example, A = "xax" and B = "xax" then the index of the first "x" must be different in the original string for A and B.

Input: s = "axxzxy"
Output: 2
Explanation: The given array with indexes looks like
a x x z x y
0 1 2 3 4 5
The longest subsequence is "xx". It appears twice as explained below.
subsequence A
x x
0 1 <-- index of subsequence A
------
1 2 <-- index of s
subsequence B
x x
0 1 <-- index of subsequence B
------
2 4 <-- index of s
We are able to use character 'x' (at index 2 in s) in both subsequences as it appears on index 1 in subsequence A and index 0 in subsequence B.

Input: s = "axxxy"
Output: 2
Explanation: The given array with indexes looks like
a x x x y
0 1 2 3 4
The longest subsequence is "xx". It appears twice as explained below.
subsequence A
x x
0 1 <-- index of subsequence A
------
1 2 <-- index of s
subsequence B
x x
0 1 <-- index of subsequence B
------
2 3 <-- index of s
We are able to use character 'x' (at index 2 in s) in both subsequencesas it appears on index 1 in subsequence A and index 0 in subsequence B.

```java
class Solution {
    public int LongestRepeatingSubsequence(String s) {
        // code here
        String a=new String(s);
        int m=s.length();
        int n=a.length();
        int[][] dp=new int[m+1][n+1];
        int ans=lcsUtil(s,a,m,n,dp);
        return ans;
    }
    static int lcsUtil(String a,String b,int m,int n,int[][] dp){
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                if(a.charAt(i-1)==b.charAt(j-1) && i!=j){
                    dp[i][j]=1+dp[i-1][j-1];
                }
                else{
                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

8.Print all Subsequence of a string




```
    static List<String> al = new ArrayList<>();
    public static void main(String[] args)
    {
        String s = "abcd";
        findsubsequences(s, ""); // Calling a function
        System.out.println(al);
    }

    private static void findsubsequences(String s,
                                         String ans)
    {
        if (s.length() == 0) {
            al.add(ans);
            return;
        }

        // We add adding 1st character in string
        findsubsequences(s.substring(1), ans + s.charAt(0));

        // Not adding first character of the string
        // because the concept of subsequence either
        // character will present or not
        findsubsequences(s.substring(1), ans);
    }
}
```

9.Permutation of String

Given a string **s**, which may contain duplicate characters, your task is to generate and return an array of all **unique** permutations of the string. You can return your answer in **any** order.

Examples:

Input: s = "ABC"
Output: ["ABC", "ACB", "BAC", "BCA", "CAB", "CBA"]
Explanation: Given string ABC has 6 unique permutations.

Input: s = "ABSG"
Output: ["ABGS", "ABSG", "AGBS", "AGSB", "ASBG", "ASGB", "BAGS", "BASG", "BGAS", "BGSA", "BSAG", "BSGA", "GABS", "GASB", "GBAS", "GBSA", "GSAB", "GSBA", "SABG", "SAGB", "SBAG", "SBGA", "SGAB", "SGBA"]
Explanation: Given string ABSG has 24 unique permutations.

Input: s = "AAA"
Output: ["AAA"]
Explanation: No other unique permutations can be formed as all the characters are same.

```java
class Solution {
    public ArrayList<String> findPermutation(String s) {
        // Code here
        Set<String >ans=new HashSet<>();
        boolean visited[]=new boolean[s.length()];
        makePermutation(s,ans,"",visited);
        return new ArrayList<>(ans);
    }
    static void makePermutation(String s,Set<String> ans,String cur,boolean visited[]){
        if(cur.length()==s.length()){
            ans.add(cur);
            return;
        }
        for(int i=0;i<s.length();i++){
            if(!visited[i]){
                visited[i]=true;
                makePermutation(s,ans,cur+s.charAt(i),visited);
                visited[i]=false;
            }
        }
    }
}
```

10.Split the binary String into substring with equal number of 0's and 1's

Given a binary string **str** of length **N**, the task is to find the maximum count of consecutive substrings **str** can be divided into such that all the substrings are balanced i.e. they have equal number of **0s** and **1s**. If it is not possible to split **str** satisfying the conditions then print **-1**.

Example:

Input: *str = "0100110101"*
Output: *4*
The required substrings are "01", "0011", "01" and "01".
Input: *str = "0111100010"*
Output: *3*

Input: *str = "0000000000"*
Output: *-1*

```java

class Solution {
    public static int maxSubStr(String str) {
        // Write your code here
        int count=0;
        int ans=0;
        int i=0;
        while(i<str.length()){
            if(str.charAt(i)=='0'){
                count--;
            }
            else{
                count++;
            }
            if(count==0){
                ans++;
            }
            i++;
        }
        return count==0?ans:-1;
    }
}

// } Driver Code Ends
```

11.Edit Distance

Given two strings **s1** and **s2.** Return the minimum number of operations required to convert **s1** to **s2**.
The possible operations are permitted:

1. **Insert** a character at any position of the string.
2. **Remove** any character from the string.
3. **Replace** any character from the string with any other character.

Examples:

Input: s1 = "geek", s2 = "gesek"
Output: 1
Explanation: One operation is required, inserting 's' between two 'e' in s1.

Input: s1 = "gfg", s2 = "gfg"
Output: 0
Explanation: Both strings are same.

Input: s1 = "abcd", s2 = "bcfe"
Output: 3
Explanation: We can convert s1 into s2 by removing 'a', replacing 'd' with 'f' and

```java
class Solution {
    public int editDistance(String s1, String s2) {
        // Code here
         int n=s1.length();
        int m=s2.length();
        int[] dp=new int[m+1];
        int[] curr=new int[m+1];
        for(int j=0;j<=m;j++)dp[j]=j;
        for(int i=1;i<=n;i++){
            curr[0]=i;
            for(int j=1;j<=m;j++){
                if(s1.charAt(i-1)==s2.charAt(j-1))curr[j]=dp[j-1];
                else curr[j]=Math.min(1+dp[j-1],Math.min(1+dp[j],1+curr[j-1]));
            }
            dp=curr.clone();
        }
        return dp[m];
    }
}
```

12.Next Permutation

Given an array of integers **arr[]** representing a permutation, implement the **next permutation** that rearranges the numbers into the lexicographically next greater permutation. If no such permutation exists, rearrange the numbers into the lowest possible order (i.e., sorted in ascending order).

Note - A permutation of an array of integers refers to a specific arrangement of its elements in a sequence or linear order.

Examples:

Input: arr = [2, 4, 1, 7, 5, 0]
Output: [2, 4, 5, 0, 1, 7]
Explanation: The next permutation of the given array is {2, 4, 5, 0, 1, 7}.

Input: arr = [3, 2, 1]
Output: [1, 2, 3]
Explanation: As arr[] is the last permutation, the next permutation is the lowest one.

Input: arr = [3, 4, 2, 5, 1]
Output: [3, 4, 5, 1, 2]
Explanation: The next permutation of the given array is [3, 4, 5, 1, 2].

```java
class Solution {
    void nextPermutation(int[] arr) {
        // code here
        int n=arr.length;
        int ind=-1;
        for(int i=n-2;i>=0;i--){
            if(arr[i]<arr[i+1]){
                ind=i;
                break;
            }
        }
        if(ind==-1){
            reverse(arr,0,n-1);
            return;
        }
        for(int i=n-1;i>ind;i--){
            if(arr[i]>arr[ind]){
                int temp=arr[i];
                arr[i]=arr[ind];
                arr[ind]=temp;
                break;
            }
        }
        reverse(arr,ind+1,n-1);
    }
    void reverse(int arr[],int start,int end){
    }
    void reverse(int arr[],int start,int end){
        while(start<end){
            int temp=arr[start];
            arr[start]=arr[end];
            arr[end]=temp;
            start++;
            end--;
        }
    }
}
```

13.Parenthesis Checker

Given a string **s**, composed of different combinations of '(' , ')', '{', '}', '[', ']', verify the validity of the arrangement.
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.

Input: s = "[{()}]"
Output: true
Explanation: All the brackets are well-formed.

Input: s = "[()()]{}"
Output: true
Explanation: All the brackets are well-formed.

Input: s = "([]"
Output: False
Explanation: The expression is not balanced as there is a missing ')' at the end.

Input: s = "([{}])"
Output: False
Explanation: The expression is not balanced as there is a closing ']' before the closing ')'.

```
static boolean ispar(String x)
{
    Stack<Character> st = new Stack<>();

    for (int i = 0; i < x.length(); i++) {
        char ch = x.charAt(i);

        if (ch == '(' || ch == '{' || ch == '[') {
            st.push(ch);
        } else {
            // If stack is empty when we encounter a closing bracket
            if (st.isEmpty()) {
                return false;
            } else {
                // Check for matching pairs
                if (ch == ')' && st.peek() != '(') {
                    return false;
                } else if (ch == '}' && st.peek() != '{') {
                    return false;
                } else if (ch == ']' && st.peek() != '[') {
                    return false;
                } else {
                    st.pop();
                }
            }
        }
    }

    // Return true if stack is empty (all brackets matched), false otherwise
    return st.isEmpty();
}
```

14.Rabin Karp

Given two strings, one is a **text** string and the other is a **pattern** string. The task is to print the starting indexes of all the occurrences of the pattern string in the text string. For printing, the Starting Index of a string should be taken as 1. The strings will only contain lowercase English alphabets (**'a'** to **'z'**).

Example 1:

Input:
text = "birthdayboy"
pattern = "birth"
Output:
[1]
Explanation:
The string "birth" occurs at index 1 in text.

Example 2:

Input:
text = "geeksforgeeks"
pattern = "geek"
Output:
[1, 9]
Explanation:
The string "geek" occurs twice in text, one starts are index 1 and the other at index 9.

```java
class Solution {

    ArrayList<Integer> search(String pattern, String text) {
        // your code here
         ArrayList<Integer> lis=new ArrayList<Integer>();

        for(int i=0;i<=text.length()-pattern.length();i++)
        {

            if(pattern.equals(text.substring(i,i+pattern.length())))
            {

                lis.add(i+1);
            }
        }
        return(lis);
    }
}
```

15.Longest Prefix Sum

Given a string of characters **s**, find the length of the longest prefix which is also a suffix.
Note: Prefix and suffix can be overlapping but they should not be equal to the entire string.

Examples :

Input: s = "abab"
Output: 2
Explanation: "ab" is the longest prefix and suffix.

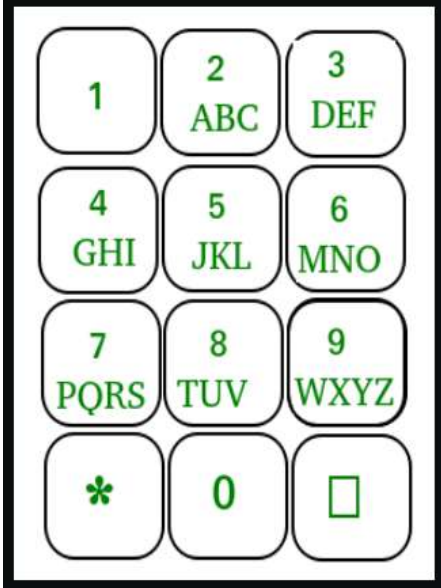
Input: s = "aabcdaabc"
Output: 4
Explanation: The string "aabc" is the longest prefix and suffix.

Input: s = "aaaa"
Output: 3
Explanation: "aaa" is the longest prefix and suffix.

```
class Solution {
    int longestPrefixSuffix(String s) {
        // code here
        int[] lps = new int[s.length()];

        int pre = 0, suf = 1;
        while (suf < s.length()) {
            if (s.charAt(pre) == s.charAt(suf)) {
                lps[suf] = pre + 1;
                pre++;
                suf++;
            } else {
                if (pre == 0) {
                    lps[suf] = 0;
                    suf++;
                } else {
                    pre = lps[pre - 1];
                }
            }
        }
        return lps[s.length() - 1];
    }
}
```

16.Convert a sentence into its equivalent mobile numeric keypad sequence

Given a sentence in the form of a string in uppercase, convert it into its equivalent mobile numeric keypad sequence. Please note there might be spaces in between the words in a sentence and we can print spaces by pressing 0.



Example 1:

Input:
S = "GFG"
Output: 43334
Explanation: For 'G' press '4' one time.
For 'F' press '3' three times.

Example 2:

Input:
S = "HEY U"
Output: 4433999088
Explanation: For 'H' press '4' two times.
For 'E' press '3' two times. For 'Y' press '9'
three times. For white space press '0' one time.
For 'U' press '8' two times.

```java
// User function Template for Java
class Solution {
    String printSequence(String S) {
        // code here
            String[]num = {"2", "22", "222",
        "3", "33", "333",
        "4", "44", "444",
        "5", "55", "555",
        "6", "66", "666",
        "7", "77", "777","7777",
        "8", "88", "888",
        "9", "99","999", "9999"
        };



        StringBuilder s = new StringBuilder();
        for(int i=0; i<S.length(); i++){
            char c = S.charAt(i);
            if(c==' '){
                s.append(0);
            }else{
               s.append(num[c-'A']);

            }
        }
        return s.toString();
    }
}
```

Custom Input

17.Count the Reversals

Given a string **s** consisting of only opening and closing curly brackets **'{'** and **'}'**, find out the **minimum** number of reversals required to convert the string into a balanced expression. A reversal means changing **'{'** to **'}'** or vice-versa.

Examples:

Input: s = "}{{}}{{{"
Output: 3
Explanation: One way to balance is: "{{{}}{}}". There is no balanced sequence that can be formed in lesser reversals.

Input: s = "{{}{{{}{{}}{{"
Output: -1
Explanation: There's no way we can balance this sequence of braces.

```java
class Solution {
    public int countMinReversals(String s) {
        // code here
        if (s.length() % 2 != 0) {
            return -1;
        }

        int size = 0;
        int openBrackets = 0;
        for(char ch : s.toCharArray()) {
            if(ch == '{') {
                size++;
            }
            else if(size > 0) {
                size--;
            }
            else {
                openBrackets++;
            }
        }

        return (int) (Math.ceil(openBrackets / 2.0) + Math.ceil(size / 2.0));
    }
}
```

18.Count Palindromic Subsequence

Given a string **s**, you have to find the number of palindromic subsequences (need not necessarily be distinct) present in the string s.

Examples:

Input: s = "abcd"
Output: 4
Explanation: palindromic subsequence are : 'a' ,'b', 'c' ,'d'

Input: s = "aab"
Output: 4
Explanation: palindromic subsequence are : 'a', 'a', 'b', 'aa'

Input: s = "b"
Output: 1
Explanation: palindromic subsequence are : 'b'

```java
class Solution {
    int countPS(String s) {
        // Your code here
        int n = s.length();
        if(n==1) return 1;

        int[][] dp = new int[n][n];

        for(int g=0; g<n; g++) {
            for(int i=0,j=g; j<n; i++,j++) {
                if(g==0) {
                    dp[i][j] = 1;
                } else if (g==1) {
                    if(s.charAt(i) == s.charAt(j)) {
                        dp[i][j] = 3;
                    } else {
                        dp[i][j] = 2;
                    }
                } else {
                    if(s.charAt(i) == s.charAt(j)) {
                        dp[i][j] = dp[i][j-1] + dp[i+1][j] + 1;
                    } else {
                        dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];
                    }
                }
            }
        }

        return dp[0][n-1];
    }
}
```

Custom Input Compile & Run

19.Count Occurrence of a given word in 2-d array

Find the number of occurrences of a given search word in a 2d-Array of characters where the word can go up, down, left, right, and around 90-degree bends.
Note: While making a word you can use one cell only once.

Example 1:

Input:
R = 4, C = 5
mat = {{S,N,B,S,N},
{B,A,K,E,A},
{B,K,B,B,K},
{S,E,B,S,E}}
target = SNAKES
Output:
3
Explanation:
S N B S N
B A K E A
B K B B K
S E B S E
Total occurrence of the word is 3
and denoted by color.

Example 2:

Input:
R = 3, C = 3
mat = {{c,a,t},
{a,t,c},
{c,t,a}}
target = cat
Output:
5
Explanation: Same explanation as first example.

```java
class Solution{
    int countoc = 0;
    public int findOccurrence(char mat[][], String target){
        int m = mat.length;
        int n = mat[0].length;
        boolean vis [][] = new boolean[m][n];
        for(int r = 0; r< m; r++){
            for(int c = 0; c < n; c++){
                if(mat[r][c] == target.charAt(0)) countoc += find(mat,target,vis,r,c,0);
            }
        }
        return countoc;
    }
    int find(char mat[][],String word, boolean vis[][],int r, int c, int index){
        if(r < 0 || c < 0 || r == vis.length || c == vis[0].length || vis[r][c] || word.charAt(index) != mat[r][c]){
            return 0;
        }
        if(index == word.length()-1) return 1;
        vis[r][c] = true;
        int top  = find(mat,word,vis,r-1,c,index+1);
        int down = find(mat,word,vis,r+1,c,index+1);
        int right= find(mat,word,vis,r,c+1,index+1);
        int left = find(mat,word,vis,r,c-1,index+1);
        vis[r][c] = false;
        return top + down + right + left;
    }
}
```

20.Find the String in grid

Given a 2D grid of **n*m** of characters and a **word**, find all occurrences of given word in grid. A word can be matched in **all 8 directions** at any point. Word is said to be found in a direction if all characters match in this direction (not in zig-zag form). The 8 directions are, **horizontally left**, **horizontally right**, **vertically up**, **vertically down**, and **4 diagonal directions**.

Note: The returning list should be **lexicographically smallest**. If the word can be found in multiple directions starting from the same coordinates, the list should contain the coordinates only once.

Example 1:

Input:
grid = {{a,b,c},{d,r,f},{g,h,i}},
word = "abc"
Output:
{{0,0}}
Explanation:
From (0,0) we can find "abc" in horizontally right direction.

Example 2:

Input:
grid = {{a,b,a,b},{a,b,e,b},{e,b,e,b}}
word = "abe"
Output:
{{0,0},{0,2},{1,0}}
Explanation:
From (0,0) we can find "abe" in right-down diagonal.
From (0,2) we can find "abe" in left-down diagonal.
From (1,0) we can find "abe" in horizontally right direction.

```
class Solution
{
    public boolean search(int i,int j,int dir,int n,int m,char[][] grid,String word,int k,int[] dx,int[] dy){
        if(k==word.length()){
            return true;
        }
        if(i<0 || i>=n || j<0 || j>=m || grid[i][j] != word.charAt(k)){
            return false;
        }

        return search(i+dx[dir],j+dy[dir],dir,n,m,grid,word,k+1,dx,dy);
    }
    public int[][] searchWord(char[][] grid, String word)
    {
```

```
        return search(i+dx[dir],j+dy[dir],dir,n,m,grid,word,k+1,dx,dy);
    }
    public int[][] searchWord(char[][] grid, String word)
    {
        ArrayList<ArrayList<Integer>> list=new ArrayList<>();
        int n=grid.length;
        int m=grid[0].length;
        int[] dx={-1,-1,0,1,1,1,0,-1};
        int[] dy={0,1,1,1,0,-1,-1,-1};
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(grid[i][j] == word.charAt(0)){
                    for(int d=0;d<8;d++){
                        int row=i+dx[d];
                        int col=j+dy[d];
                        if(search(i,j,d,n,m,grid,word,0,dx,dy)){
                            ArrayList<Integer> t=new ArrayList<>();
                            t.add(i);
                            t.add(j);
                            list.add(t);
                            break;
                        }
                    }
                }
            }
        }
        int[][] ans=new int[list.size()][2];
        for(int i=0;i<list.size();i++){
            ans[i][0]=list.get(i).get(0);
            ans[i][1]=list.get(i).get(1);
        }
        return ans;
    }
}
```

21.Pattern Searching

Given a string **txt** and a pattern **pat**. Your task is to return **true** if the pattern is present in the given string otherwise return **false**.

Examples:

Input: txt = "abcdefh", pat = "bcd"
Output: true
Explanation: The pattern "bcd" exist in "**abcd**efh".

Input: txt = "axzy", pat = "xy"
Output: false
Explanation: xy is not present in txt.

Constraints:

```
bool searchPattern(string str, string pat)
{
   // your code here

   if(str.length()==pat.length()){
      if(str==pat){
         return true;
      }
      return false;
   }
   else{
   for (int i = 0; i < str.length(); i++) {
      for (int j = 0; j < str.length(); j++) {
         string st=str.substr(i,j);
         if(st==pat){
            return true;
         }
      }
   }
   return false;
   }
}
```

```
class Solution {
    public static boolean searchPattern(String txt, String pat) {
        // code here
          if(txt.contains(pat)){
              return true;

          }
          return false;
    }
}
```

22.Roman Number to Integer

Given a string in Roman number format (s), your task is to convert it to an integer. Various symbols and their values are given below.
Note: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000

Examples:

Input: s = "IX"
Output: 9
Explanation: IX is a Roman symbol which represents 10 – 1 = 9.

Input: s = "XL"
Output: 40
Explanation: XL is a Roman symbol which represents 50 – 10 = 40.

Input: s = "MCMIV"
Output: 1904
Explanation: M is 1000, CM is 1000 – 100 = 900, and IV is 4. So we have total as 1000 + 900 + 4 = 1904.

```java
class Solution {
    // Finds decimal value of a given roman numeral
    public int romanToDecimal(String s) {
        // code here
        HashMap<Character,Integer> map=new HashMap<>();
        map.put('I',1);
        map.put('V',5);
        map.put('X',10);
        map.put('L',50);
        map.put('C',100);
        map.put('D',500);
        map.put('M',1000);

        int n=s.length();
        int res=0;
        for(int i=0;i<n-1;i++){
            if(map.get(s.charAt(i)) < map.get(s.charAt(i+1))){
                res-=map.get(s.charAt(i));
            }else{
                res+=map.get(s.charAt(i));
            }

        }
        return res+=map.get(s.charAt(n-1));
    }
}
```

23.Longest Common Prefix of string

Given an array of strings **arr[]**. Return the **longest common prefix** among each and every strings present in the array. If there's no prefix common in all the strings, return "".

Examples :

Input: arr[] = ["geeksforgeeks", "geeks", "geek", "geezer"]
Output: "gee"
Explanation: "gee" is the longest common prefix in all the given strings.

Input: arr[] = ["hello", "world"]
Output: ""
Explanation: There's no common prefix in the given strings.

```java
// User function Template for Java
class Solution {
    public String longestCommonPrefix(String arr[]) {
        // code here
          if(arr.length==1){
            return arr[0];
        }
        String prefix=arr[0];
        String newPrefix="";

        for(int i=1;i<arr.length;i++){
            String s=arr[i];
            newPrefix=getNewPrefix(prefix,s);
            if(newPrefix.equals("")){

                return newPrefix;
            }

        }
        return newPrefix;
    }
     public String getNewPrefix(String prefix,String s){
        char[] ch1=prefix.toCharArray();
        char[] ch2=s.toCharArray();
        StringBuffer sb=new StringBuffer();

        for(int i=0;i<ch1.length;i++){
            if((i<ch1.length && i<ch2.length) && (ch1[i]==ch2[i])){
                sb.append(ch1[i]);
            }else{
                return new String(sb);
            }
        }

        return new String(sb);
    }
```

24.Min Number of Flips

Given a binary string, that is it contains only 0s and 1s. We need to make this string a sequence of alternate characters by flipping some of the bits, our goal is to minimize the number of bits to be flipped.

Example 1:

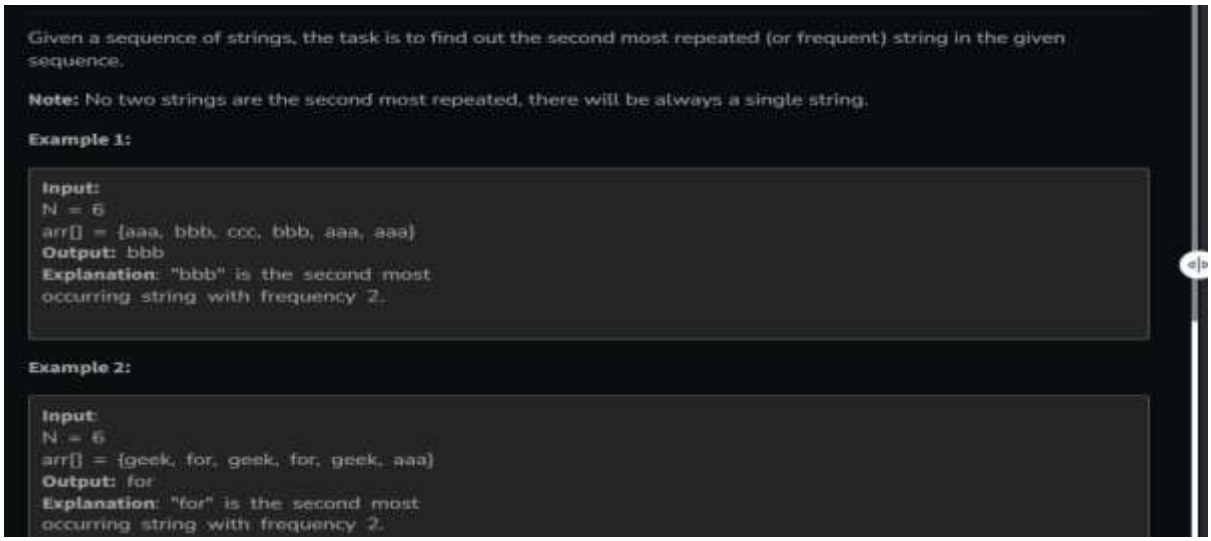
Input:
S = "001"
Output: 1
Explanation:
We can flip the 0th bit to 1 to have
101.

Example 2:

Input:
S = "0001010111"
Output: 2
Explanation: We can flip the 1st and 8th bit
bit to have "0101010101"
101.

```java
class Solution {
    public int minFlips(String S) {
        // case1: Count flips needed to make the string alternate starting with '0' (i.e., "0101...")
        // case2: Count flips needed to make the string alternate starting with '1' (i.e., "1010...")
        int case1 = 0;
        int case2 = 0;

        for (int i = 0; i < S.length(); i++) {
            char expectedCharCase1 = (i % 2 == 0) ? '0' : '1'; // pattern "0101..."
            char expectedCharCase2 = (i % 2 == 0) ? '1' : '0'; // pattern "1010..."

            if (S.charAt(i) != expectedCharCase1) {
                case1++; // Flip needed for pattern starting with '0'
            }
            if (S.charAt(i) != expectedCharCase2) {
                case2++; // Flip needed for pattern starting with '1'
            }
        }

        // Return the minimum number of flips required
        return Math.min(case1, case2);
    }

}
```

25.Second Most repeated string in sequence



Given a sequence of strings, the task is to find out the second most repeated (or frequent) string in the given sequence.

Note: No two strings are the second most repeated, there will be always a single string.

Example 1:

Input:
N = 6
arr[] = {aaa, bbb, ccc, bbb, aaa, aaa}
Output: bbb
Explanation: "bbb" is the second most occurring string with frequency 2.

Example 2:

Input:
N = 6
arr[] = {geek, for, geek, for, geek, aaa}
Output: for
Explanation: "for" is the second most occurring string with frequency 2.

```java
class Solution {
    String secFrequent(String arr[], int N) {
        // your code here
        // String  str= "";
       StringBuilder sb = new StringBuilder();
        Map<String , Integer> hm = new HashMap<>();

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

            if(hm.containsKey(arr[i])){
                hm.put(arr[i], hm.get(arr[i])+1);

            }
            else{
                hm.put(arr[i],1);
            }
        }
        ArrayList<Integer> al = new ArrayList<>();
        for( String e : hm.keySet()){
            al.add(hm.get(e));

        }
        al.sort(Comparator.reverseOrder());
       // System.out.println(al);
        for( Map.Entry<String, Integer> entry  : hm.entrySet()){
         if( hm.get(entry.getKey()) == al.get(1)){
           //  str+= entry.getKey();
           sb.append(entry.getKey());
            break;

         }
        }
        return sb.toString();
    }
}
```

26.Minimum Swaps for bracket balancing

You are given a string **s** of **2*n** characters consisting of **n** '[' brackets and **n** ']' brackets. A string is considered **balanced** if it can be represented in the form **a[b]** where **a** and **b** are balanced strings. We can make an unbalanced string balanced by swapping **adjacent** characters. Calculate the **minimum number of swaps** necessary to make a string balanced.
Note - Strings **a** and **b** can be **empty**.

Examples :

Input: s = "[]][["
Output: 2
Explanation: First swap: Position 3 and 4 [][][, Second swap: Position 5 and 6 [][][]

Input: s = "[][]"
Output : 0
Explanation: String is already balanced.

Input: s = "[[][][]]"
Output: 0

```java
// User function Template for Java
class Solution {
    static int minimumNumberOfSwaps(String s) {
        // code here
        int value = 0;
        int ans =0 ;
        for(int  i =0;i<s.length();i++){
            if(s.charAt(i)=='['){
                value++;
            }else{
             value--;
             if(value <0){
                 ans = ans-value;
             }

            }
        }


        return  ans;
    }
}
```

27.Longest Common Subsequence

Given two strings **s1** and **s2**, return the length of their **longest common subsequence** (LCS). If there is no common subsequence, return **0**.

A subsequence is a sequence that can be derived from the given string by deleting some or no elements without changing the order of the remaining elements. For example, "ABE" is a subsequence of "ABCDE".

Examples:

Input: s1 = "ABCDGH", s2 = "AEDFHR"
Output: 3
Explanation: The longest common subsequence of "ABCDGH" and "AEDFHR" is "ADH", which has a length of 3.

Input: s1 = "ABC", s2 = "AC"
Output: 2
Explanation: The longest common subsequence of "ABC" and "AC" is "AC", which has a length of 2.

Input: s1 = "XYZW", s2 = "XYWZ"
Output: 3
Explanation: The longest common subsequences of "XYZW" and "XYWZ" are "XYZ" and "XYW", both of length 3.

```java

class Solution {
    static int lcs(String s1, String s2) {
        // code here
         int n = s1.length();
        int m = s2.length();
        int[][] dp = new int[n + 1][m + 1];

        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                if(s1.charAt(i-1)==s2.charAt(j-1)){
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                }
                else{
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[n][m];
    }
}
```

28.Generate IP Address

Given a string **s** containing only digits, your task is to restore it by returning all possible valid IP address combinations. You can return your answer in **any** order.

A **valid** IP address must be in the form of A.B.C.D, where A, B, C, and D are numbers from 0-255(inclusive).

Note: The numbers cannot be 0 prefixed unless they are 0. For example, 1.1.2.11 and 0.11.21.1 are valid IP addresses while 01.1.2.11 and 00.11.21.1 are not.

Examples:

Input: s = "255678166"
Output: ["25.56.78.166", "255.6.78.166", "255.67.8.166", "255.67.81.66"]
Explanation: These are the only valid possible IP addresses.

Input: s = "25505011535"
Output: []
Explanation: We cannot generate a valid IP address with this string.

```java
class Solution {
    public ArrayList<String> generateIp(String s) {
        // code here
         ArrayList<String> result = new ArrayList<>();

        // If the string length is not between 4 and 12, it's not possible to form a valid IP address
        if (s.length() < 4 || s.length() > 12) {
            return result;
        }

        // Start backtracking to find all valid combinations
        backtrack(s, 0, new StringBuilder(), result, 0);

        return result;
    }

    private void backtrack(String s, int start, StringBuilder current, List<String> result, int count) {
```

```
    return result;
}

private void backtrack(String s, int start, StringBuilder current, List<String> result, int count) {
    // If we have 4 segments and we've used up the string
    if (count == 4) {
        if (start == s.length()) {
            result.add(current.toString());
        }
        return;
    }

    // Try to form a valid segment (one to three digits long)
    for (int len = 1; len <= 3; len++) {
        if (start + len > s.length()) {
            break;
        }

        String segment = s.substring(start, start + len);

        // Check if the segment is valid:
        if ((segment.charAt(0) == '0' && segment.length() > 1) || Integer.parseInt(segment) > 255) {
            continue;
        }

        int currentLength = current.length();

        // If we're not on the first segment, add a dot
        if (count > 0) {
            current.append('.');
        }

        // Add the valid segment
        current.append(segment);

        // Recurse for the next segment
        backtrack(s, start + len, current, result, count + 1);

        // Backtrack: Remove the segment and the dot
        current.setLength(currentLength);
    }
}
```

29.Smallest distinct window

Given a string **str,** your task is to find the **length** of the **smallest** window that contains **all** the characters of the given string at least once.

Example:

Input: str = "aabcbcdbca"
Output: 4
Explanation: Sub-String "dbca" has the smallest length that contains all the characters of str.

Input: str = "aaab"
Output: 2
Explanation: Sub-String "ab" has the smallest length that contains all the characters of str.

Input: str = "geeksforgeeks"
Output: 7
Explanation: There are multiple substring with smallest length that contains all characters of str, "eksforg" and "ksforge".

```java
class Solution {
    public int findSubString(String str) {
        // code here
        Set<Character> hs=new HashSet<>();
        for(char ch:str.toCharArray()){
            hs.add(ch);
        }
         int ans = Integer.MAX_VALUE;
        int totalUniqueCharacters=hs.size();
        Map<Character,Integer> hm=new HashMap<>();
        int j=0;
        for(int i=0;i<str.length();i++){
            hm.put(str.charAt(i),hm.getOrDefault(str.charAt(i),0)+1);

            if(hm.size()==totalUniqueCharacters){
                while(hm.get(str.charAt(j))>1){
                    int freq=hm.get(str.charAt(j));
                    hm.put(str.charAt(j),freq-1);
                    j++;
                }
                ans=Math.min(ans,i-j+1);
            }

        }
        return ans;
    }
}
```

30.Reaarange Character

Given a string **s** with repeated characters, the task is to rearrange characters in a string such that no two adjacent characters are the same.
Note: The string has only lowercase English alphabets and it can have multiple solutions. Return any one of them. If there is no possible solution, then print empty string ("").

Examples:

Input : s = "aaabc"
Output: 1
Explanation: "aaabc" can rearranged to "abaca" or "acaba" as no two adjacent characters are same in the output string.

Input : s= "aaabb"
Output: 1
Explanation: "aaabb" can rearranged to "ababa" as no two adjacent characters are same in the output string.

Input : s = "aaaabc"
Output: 0
Explanation: No combinations possible such that two adjacent characters are different.

```java
class Solution {
    public static String rearrangeString(String s) {
        // code here
           int n = s.length();

        Map<Character, Integer> freq = new HashMap<>();
        for (char c : s.toCharArray())
            freq.put(c, freq.getOrDefault(c, 0) + 1);

        PriorityQueue<Map.Entry<Character, Integer>> pq =
            new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());
        pq.addAll(freq.entrySet());

        StringBuilder res = new StringBuilder();

        Map.Entry<Character, Integer> prev = new AbstractMap.SimpleEntry<>('#', -1);

        while (!pq.isEmpty()) {

            Map.Entry<Character, Integer> p = pq.poll();
            res.append(p.getKey());

            if (prev.getValue() > 0)
                pq.offer(prev);

            prev = new AbstractMap.SimpleEntry<>(p.getKey(), p.getValue() - 1);
        }

        if (res.length() != n)
            return "";

        return res.toString();
    }
}
```

31.Min Char to add for Palindrome

Given a string **s**, the task is to find the minimum characters to be added at the front to make the string palindrome.

Note: A palindrome string is a sequence of characters that reads the same forward and backward.

Examples:

Input: s = "abc"
Output: 2
Explanation: Add 'b' and 'c' at front of above string to make it palindrome : "**cb**abc"

Input: s = "aacecaaaa"
Output: 2
Explanation: Add 2 a's at front of above string to make it palindrome : "**aa**aacecaaaa"

```java
class Solution {
    public static int minChar(String s) {
        // Write your code here
        StringBuilder rev = new StringBuilder();
        rev.append(s);
        rev.reverse();

        StringBuilder newStr = new StringBuilder();
        newStr.append(s).append('$').append(rev.toString());
        String complete = newStr.toString();
        int n = complete.length();
        int i = 0, j = 1;
        int arr[] = new int[n];

        Arrays.fill(arr, 0);

        while(j < n){
            if(complete.charAt(j) == complete.charAt(i)) arr[j++] = ++i;
            else if(i != 0) i = arr[i - 1];
            else j++;
        }
        return s.length() - i;
    }
}
```

32.Print Anagram Together

Given an array of strings, return all groups of strings that are anagrams. The strings in each group must be arranged in the order of their appearance in the original array. Refer to the sample case for clarification.

Examples:

Input: arr[] = ["act", "god", "cat", "dog", "tac"]
Output: [["act", "cat", "tac"], ["god", "dog"]]
Explanation: There are 2 groups of anagrams "god", "dog" make group 1. "act", "cat", "tac" make group 2.

Input: arr[] = ["no", "on", "is"]
Output: [["is"], ["no", "on"]]
Explanation: There are 2 groups of anagrams "is" makes group 1. "no", "on" make group 2.

Input: arr[] = ["listen", "silent", "enlist", "abc", "cab", "bac", "rat", "tar", "art"]
Output: [["abc", "cab", "bac"], ["listen", "silent", "enlist"], ["rat", "tar", "art"]]
Explanation:
Group 1: "abc", "bac", and "cab" are anagrams.
Group 2: "listen", "silent", and "enlist" are anagrams.
Group 3: "rat", "tar", and "art" are anagrams.

```java
class Solution {
    public ArrayList<ArrayList<String>> anagrams(String[] arr) {
        // code here
        // Initialize a map to store sorted string as key and list of strings as values
         Map<String, List<String>> res = new HashMap<>();

        // Iterate over the list of strings
        for (String temp : arr) {
            // Convert the string to a char array, sort it, and convert it back to a string
            char[] chars = temp.toCharArray();
            Arrays.sort(chars);
            String sortedTemp = new String(chars);

            // Add the original string to the corresponding sorted key in the map
            res.putIfAbsent(sortedTemp, new ArrayList<>());
            res.get(sortedTemp).add(temp);
        }

        // Convert the map values to an ArrayList<ArrayList<String>>
        ArrayList<ArrayList<String>> ans = new ArrayList<>();
        for (List<String> group : res.values()) {
            ans.add(new ArrayList<>(group));
        }

        return ans;
    }
}
```

33.Smallest window containing all character of another string

Given two strings **s1** and **s2**. Find the smallest window in the string **s1** consisting of all the characters(**including duplicates**) of the string **s2.** Return "" in case no such window is present. If there are multiple such windows of the same length, return the one with the **least starting index**.
Note: All characters are in lowercase letters.

Examples:

Input: s1 = "timetopractice", s2 = "toc"
Output: "toprac"
Explanation: "toprac" is the smallest substring in which "toc" can be found.

Input: s1 = "zoomlazapzo", **s2** = "oza"
Output: "apzo"
Explanation: "apzo" is the smallest substring in which "oza" can be found.

Input: s1 = "zoom", s2 = "zooe"
Output: ""
Explanation: No window is present containing all characters of s2.

```java
class Solution
{
    //Function to find the smallest window in the string s consisting
    //of all the characters of string p.
    public static String smallestWindow(String s, String p)
    {
        // Your code here
         int[] freq = new int[26];
        int[] store = new int[26];

        for(char ch : p.toCharArray()) {
            freq[ch - 'a']++;
            store[ch - 'a']++;
        }

        int start = 0;
        int end = 0;
        int min = Integer.MAX_VALUE;
        int i = 0;
        int j = 0;
        int len = 0;

        while(j < s.length()) {
            if(store[s.charAt(j) - 'a'] > 0) {
                freq[s.charAt(j) - 'a']--;
                if(freq[s.charAt(j) - 'a'] >= 0) {
                    len++;
                }
            }

            while(len == p.length()) {
                if(j - i + 1 < min) {
                    start = i;
                    end = j;
```

```
    int min = Integer.MAX_VALUE;
    int i = 0;
    int j = 0;
    int len = 0;

    while(j < s.length()) {
        if(store[s.charAt(j) - 'a'] > 0) {
            freq[s.charAt(j) - 'a']--;
            if(freq[s.charAt(j) - 'a'] >= 0) {
                len++;
            }
        }

        while(len == p.length()) {
            if(j - i + 1 < min) {
                start = i;
                end = j;
                min = j - i + 1;
            }
            if(store[s.charAt(i) - 'a'] > 0) {
                freq[s.charAt(i) - 'a']++;
                if(freq[s.charAt(i) - 'a'] > 0)
                    len--;
            }
            i++;
        }
        j++;
    }
    return (min == Integer.MAX_VALUE) ? "-1" : s.substring(start,end + 1);
}
```

34.Remove Consecutive Character

You are given a string **s**, consisting of lowercase alphabets. Your task is to remove consecutive duplicate characters from the string.

Example:

Input: s = "aabb"
Output: "ab"
Explanation:
The character 'a' at index 2 is the same as 'a' at index 1, so it is removed.
Similarly, the character 'b' at index 4 is the same as 'b' at index 3, so it is removed.
The final string is "ab".

Input:s = "aabaa"
Output: "aba"
Explanation:
The character 'a' at index 2 is the same as 'a' at index 1, so it is removed.
The character 'a' at index 5 is the same as 'a' at index 4, so it is removed.
The final string is "aba".

Input: s = "abcddcba"
Output: "abcdcba"
Explanation:
The character 'd' at index 5 is the same as 'd' at index 4, so it is removed.
No other consecutive duplicates exist.
The final string is "abcdcba".

```java
class Solution {
    public String removeConsecutiveCharacter(String s) {
        // code here
        StringBuilder sb = new StringBuilder();
        char prev ='\0';
        for(char ch : s.toCharArray()) {
            if(ch!=prev) {

            sb.append(ch);
        }
        prev = ch;
    }

        return sb.toString();
    }
}
```

35.Transform String

Given two strings A and B. Find the minimum number of steps required to transform string A into string B. The only allowed operation for the transformation is selecting a character from string A and inserting it in the beginning of string A.

Example 1:

Input:
A = "abd"
B = "bad"
Output: 1
Explanation: The conversion can take place in 1 operation: Pick 'b' and place it at the front.

Example 2:

Input:
A = "GeeksForGeeks"
B = "ForGeeksGeeks"
Output: 3
Explanation: The conversion can take place in 3 operations:
Pick 'r' and place it at the front.
A = "rGeeksFoGeeks"
Pick 'o' and place it at the front.
A = "orGeeksFGeeks"
Pick 'F' and place it at the front.
A = "ForGeeksGeeks"

```java
class Solution {
    int transform(String A, String B) {
        // code here
        HashMap<Character,Integer> map1 =new HashMap<>();
        HashMap<Character,Integer> map2 =new HashMap<>();
        if(A.length()!=B.length()){
            return -1;
        }
        char chA[] =A.toCharArray();
        char chB[] =B.toCharArray();

         for(Character e: chA){
             map1.put(e,map1.getOrDefault(e,0)+1);
         }
         for(Character e: chB){
             map2.put(e,map2.getOrDefault(e,0)+1);
         }
         if(!map1.equals(map2)) return -1;
        int count =0;
        int i=A.length()-1;
        int j =B.length()-1;
        while(i>=0 && j>=0){
            if(A.charAt(i)==B.charAt(j)){
                i--;
                j--;
            }else{
                count++;
                i--;
            }
        }
        return count;
    }
}
```

36.Isomorphic String

Given two strings **s1** and **s2**, check if these two strings are **isomorphic** to each other.

If the characters in s1 can be changed to get s2, then two strings, s1 and s2, are isomorphic. A character must be completely swapped out for another character while maintaining the order of the characters. A character may map to itself, but no two characters may map to the same character.

Examples:

Input: s1 = "aab", s2 = "xxy"
Output: true
Explanation: There are two different characters in aab and xxy, i.e a and b with frequency 2 and 1 respectively.

Input: s1 = "aab", s2 = "xyz"
Output: false
Explanation: There are two different characters in aab but there are three different charactersin xyz. So there won't be one to one mapping between s1and s2.

Input: s1 = "aac", s2 = "xyz"
Output: false
Explanation: There are two different characters in aab but there are three different charactersin xyz. So there won't be one to one mapping between s1and s2.

```java
class Solution {
    // Function to check if two strings are isomorphic.
    public static boolean areIsomorphic(String s1, String s2) {
        // Your code here
        HashMap<Character,Character> map1=new HashMap<>();
        HashMap<Character,Character> map2=new HashMap<>();

        if(s1.length()!=s2.length())
            return false;
        for(int i=0;i<s1.length();i++){
            char ch1=s1.charAt(i);
            char ch2=s2.charAt(i);

            if(map1.containsKey(ch1)){
                if(map1.get(ch1)!=ch2)
                    return false;
            }else{
                map1.put(ch1,ch2);
            }

            if(map2.containsKey(ch2)){
                if(map2.get(ch2)!=ch1)
                    return false;
            }else{
                map2.put(ch2,ch1);
            }
        }
        return true;
    }
}
```