```
package Love_Babbar;

public class Revision {
    // ---Reverse a String
    // ---Check whether a String is Palindrome or not
    // ---Find Duplicate characters in a string NA
    // Why strings are immutable in Java? NA
    // ---Write a Code to check whether one string is a rotation of another(Join
    // String)
    // ---Write a Program to check whether a string is a valid shuffle of two
    // strings or not NA(3 Pointer)
    // ---Count and Say problem
    // ---Write a program to find the longest Palindrome in a string.[ Longest
    // palindromic Substring](Even Odd Method)
    // ---Find Longest Recurring Subsequence in String(LCS Variation)
    // ---Print all Subsequences of a string.
    // --- Print all the permutations of the given string(IMPORTANT)
    // ---Split the Binary string into two substring with equal 0's and 1's NA
    // Word Wrap Problem [VERY IMP].
    // --- EDIT Distance [Very Imp]
    // ---Find next greater number with same set of digits. [Very Very IMP]
    // ---Balanced Parenthesis problem.[Imp]
    // ---Word break Problem[ Very Imp]
    // Rabin Karp Algorithm
    // KMP Algorithm
    // --- Convert a Sentence into its equivalent mobile numeric keypad sequence.
    // ---Minimum number of bracket reversals needed to make an expression balanced.
    // ---Count All Palindromic Subsequence in a given String.
    // ---Count of number of given string in 2D character array
    // ---Search a Word in a 2D Grid of characters.
    // Boyer Moore Algorithm for Pattern Searching.
    // ---Converting Roman Numerals to Decimal
    // ---Longest Common Prefix
    // ---Number of flips to make binary string alternate
    // Find the first repeated word in string.
    // ---Minimum number of swaps for bracket balancing.
    // Find the longest common subsequence between two strings.
    // ---Program to generate all possible valid IP addresses from given string.
    // ---Write a program to find the smallest window that contains all characters
    // of
    // string itself.
    // Rearrange characters in a string such that no two adjacent are same
    // ---Minimum characters to be added at front to make string palindrome
    // ---Given a sequence of words, print all anagrams together
    // ---Find the smallest window in a string containing all characters of another
    // string
    // ---Recursively remove all adjacent duplicates
    // ---String matching where one string contains wildcard characters
    // Function to find Number of customers who could not get a computer NA
```

```
    // ---Transform One String to Another using Minimum Number of Given Operation
    // ---Check if two given strings are isomorphic to each other
    // ---Recursively print all sentences that can be formed from list of word lists
    // NA
}

public class String {
    class Strings {
        // String
        // Reverse a String
        class Solution {
            // Complete the function
            // str: input string
            public static String reverseWord(String str) {
                // Reverse the string str
                StringBuilder str2 = new StringBuilder(str);
                return str2.reverse().toString();
            }
        }

        // Check whether a String is Palindrome or not
        class Solution {
            int isPalindrome(String S) {
                // code here
                StringBuilder str = new StringBuilder(S);
                String S2 = str.reverse().toString();
                return S2.equals(S) ? 1 : 0;
            }
        };

        // Find Duplicate characters in a string
        class Solution {
            public static void main(String[] args) {
                Map<Character,Integer> mp=new HashMap<>();
                int n=s.length();
                for(int i=0;i<n;i++){mp.put(s.charAt(i),mp.getOrDefault(s.charAt(i),0)+1);}
                for(Character ch: mp.keySet){
                    if(mp.get(ch)>1){System.out.println(ch+" "+mp.get(ch))}
                }
            }
        }

        // Why strings are immutable in Java?
        // Write a Code to check whether one string is a rotation of another
        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
                int l1 = s1.length();
```

```
        int l2 = s2.length();
        String base = s1 + s1;
        if (l1 != l2) {
            return false;
        }
        if (base.contains(s2) == true) {
            return true;
        } else
            return false;
    }

}

// Write a Program to check whether a string is a valid shuffle of two strings
// or not
class Solution {
    public void main(String S1,String S2,String S3){
            int n1=S1.length();
            int n2=S2.length();
            int n3=S3.length();
            if((n1+n2!)=n3)){return 0;}
            while(i<n1&&j<n2){
                if(S1.charAt(i)==S3.charAt(k)){i++;k++;}
                else if(S2.charAt(j)==S3.charAt(k)){j++;k++;}
                else{f=1;break;}
            }
            if(j<n2&&i<n1){return 0;}
            return 1;
        }
}

// Count and Say problem
class Solution {
    static String lookandsay(int n) {
        // your code here
        if (n == 1) {
            return "1";
        }
        if (n == 2) {
            return "11";
        }
        String pat = new String("11");

        for (int i = 3; i <= n; i++) {

            String sb = new String();
            int c = 1;
            pat += "$";
            // System.out.println(pat);
```

```
            for (int j = 1; j < pat.length(); j++) {
                if (pat.charAt(j) == pat.charAt(j - 1)) {
                    c++;
                } else {
                    sb += Integer.toString(c) + "";
                    sb += pat.charAt(j - 1) + "";

                    c = 1;
                }
            }

            pat = sb;
        }
        return pat;
    }
}

// Write a program to find the longest Palindrome in a string.[ Longest
// palindromic Substring]
class Solution {
    static String longestPalin(String S) {
        // code here
        int l1 = S.length();
        int l, h, start = 0, end = 1;
        for (int i = 1; i < l1; i++) {
            // Even Palindrome
            l = i - 1;
            h = i;
            while (l >= 0 && h < l1 && S.charAt(l) == S.charAt(h)) {
                if (h - l + 1 > end) {
                    start = l;
                    end = h - l + 1;
                }
                l--;
                h++;
            }
            // Odd Substring
            l = i - 1;
            h = i + 1;

            while (l >= 0 && h < l1 && S.charAt(l) == S.charAt(h)) {
                if (h - l + 1 > end) {
                    start = l;
                    end = h - l + 1;
                }
                l--;
                h++;
            }
        }
```

```
            return S.substring(start, start + end);
        }
    }

    // Find Longest Recurring Subsequence in String
    class Solution {
        public int LongestRepeatingSubsequence(String str) {
            // code here
            int n = str.length();
            int dp[][] = new int[n + 1][n + 1];
            for (int rows[] : dp) {
                Arrays.fill(rows, -1);
            }
            return f(str, str, n - 1, n - 1, dp);
        }

        public int f(String s1, String s2, int ind1, int ind2, int dp[][]) {

            if (ind1 < 0 || ind2 < 0) {
                return 0;
            }
            if (dp[ind1][ind2] != -1) {
                return dp[ind1][ind2];
            }
            if (s1.charAt(ind1) == s2.charAt(ind2) && ind1 != ind2) {
                return dp[ind1][ind2] = 1 + f(s1, s2, ind1 - 1, ind2 - 1, dp);
            }
            return dp[ind1][ind2] = Math.max(f(s1, s2, ind1 - 1, ind2, dp), f(s1, s2, ind1,
ind2 - 1, dp));
        }

        int LongestRepeatingSubsequence(string str) {
            int n = str.length();
            // Create and initialize DP table
            int dp[n+1][n+1];
            for (int i=0; i<=n; i++){
                for (int j=0; j<=n; j++){
                    dp[i][j] = 0;
                }
            }
            // Fill dp table (similar to LCS loops)
            for (int i=1; i<=n; i++){
                for (int j=1; j<=n; j++){
            // If characters match and indexes are not same
                    if (str[i-1] == str[j-1] && i!=j)
                            dp[i][j] = 1 + dp[i-1][j-1];
            // If characters do not match
                    else
                            dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
```

```
                }
            }
        return dp[n][n];
        }
}

// Print all Subsequences of a string.
class Solution {
    public static ArrayList<String> subsequences(String str) {
        // Write your code here
        ArrayList<String> res = new ArrayList<>();
        int n = str.length();
        f(0, n, str, res, "");
        return res;
    }

    public static void f(int ind, int n, String str, ArrayList<String> res, String t) {
        if (ind == n) {
            if (t.equals("")) {
                return;
            } else {
                res.add(t);
                return;
            }
        }
        f(ind + 1, n, str, res, t);
        f(ind + 1, n, str, res, t + str.charAt(ind));
    }
}

// Print all the permutations of the given string
class Solution {
    public List<String> find_permutation(String S) {
        // Code here
        List<String> ls = new ArrayList<>();
        ls.add(S.charAt(0) + "");
        for (int i = 1; i < S.length(); i++) {
            List<String> w = new ArrayList<>();
            for (int j = 0; j < ls.size(); j++) {
                String curr = ls.get(j);
                int len = curr.length();
                for (int k = 0; k <= len; k++) {
                    String newp = curr.substring(0, k) + S.charAt(i) + curr.substring(k);
                    w.add(newp);
                }
            }
            ls = w;
        }
        Collections.sort(ls);
```

```
        return ls;
    }
}

// Split the Binary string into two substring with equal 0’s and 1’s
class Solution {
    static int maxSubStr(String str, int n) {
        int count0 = 0, count1 = 0;
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            if (str.charAt(i) == '0') {
                count0++;
            } else {
                count1++;
            }
            if (count0 == count1) {
                cnt++;
            }
        }
        if (count0 != count1) {
            return -1;
        }
        return cnt;
    }
}

// Word Wrap Problem [VERY IMP].
class Solution {
    class Solution {
        public int solveWordWrap(int[] nums, int k) {
            // Code here
            int n = nums.length;
            int[][] dp = new int[n][k + 1];
            for (int i = 0; i < n; i++) {
                Arrays.fill(dp[i], -1);
            }
            return t(nums, n, k, 0, k, dp);
        }

        public int t(int[] nums, int n, int k, int ind, int rem, int[][] dp) {
            if (dp[ind][rem] != -1) {
                return dp[ind][rem];
            }
            dp[ind][rem] = f(nums, n, k, 0, k, dp);
            return dp[ind][rem];
        }

        public int f(int[] nums, int n, int k, int ind, int rem, int[][] dp) {
            if (ind == n - 1) {
```

```
                if (nums[ind] < rem) {
                    return dp[ind][rem] = 0;
                } else
                    return dp[ind][rem] = rem * rem;
            }

            if (nums[ind] < rem) {
                return Math.min(f(nums, n, k, ind + 1, rem == k ? rem - nums[ind] : rem -
nums[ind] - 1, dp),
                        rem * rem + f(nums, n, k, ind + 1, k - nums[ind], dp));
            } else
                return rem * rem + f(nums, n, k, ind + 1, k - nums[ind], dp);
        }
    }
}

// EDIT Distance [Very Imp]
class Solution {
    static int editDistanceUtil(String S1, String S2, int i, int j, int[][] dp) {
        if (i < 0)
            return j + 1;
        if (j < 0)
            return i + 1;
        if (dp[i][j] != -1)
            return dp[i][j];
        if (S1.charAt(i) == S2.charAt(j))
            return dp[i][j] = 0 + editDistanceUtil(S1, S2, i - 1, j - 1, dp);
        // Minimum of three choices
        else
            return dp[i][j] = 1 + Math.min(editDistanceUtil(S1, S2, i - 1, j - 1, dp),
                Math.min(editDistanceUtil(S1, S2, i - 1, j, dp), editDistanceUtil(S1, S2,
i, j - 1, dp)));
        }

    static int editDistance(String S1, String S2) {
        int n = S1.length();
        int m = S2.length();
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 0; i <= n; i++) {
            dp[i][0] = i;
        }
        for (int j = 0; j <= m; j++) {
            dp[0][j] = j;
        }
        for (int i = 1; i < n + 1; i++) {
            for (int j = 1; j < m + 1; j++) {
                if (S1.charAt(i - 1) == S2.charAt(j - 1))
                    dp[i][j] = 0 + dp[i - 1][j - 1];
                else
```

```
                dp[i][j] = 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]));
            }
        }
        return dp[n][m];
    }
}

// Find next greater number with same set of digits. [Very Very IMP]
class Solution {
    public void nextPermutation(int[] A) {
        if (A == null || A.length <= 1)
            return;
        int i = A.length - 2;
        while (i >= 0 && A[i] >= A[i + 1])
            i--;
        if (i >= 0) {
            int j = A.length - 1;
            while (A[j] <= A[i])
                j--;
            swap(A, i, j);
        }
        reverse(A, i + 1, A.length - 1);
    }

    public void swap(int[] A, int i, int j) {
        int tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }

    public void reverse(int[] A, int i, int j) {
        while (i < j)
            swap(A, i++, j--);
    }
}

// Balanced Parenthesis problem.[Imp]
class Solution {
    public boolean isValid(String s) {
        Stack<Character> st = new Stack<Character>();
        for (char it : s.toCharArray()) {
            if (it == '(' || it == '[' || it == '{')
                st.push(it);
            else {
                if (st.isEmpty()) {
                    return false;
                }
                char ch = st.pop();
                if ((it == ')' && ch == '(') || (it == ']' && ch == '[') || (it == '}' && ch == '{')) {
```

```
                    continue;
                } else {
                    return false;
                }
            }
        }
        return st.isEmpty();
    }
}

// Word break Problem[ Very Imp]
class Solution {
    public static int wordBreak(String A, ArrayList<String> B) {
        // code here
        Set<String> dictionary = new HashSet<>();
        for (String temp : B) {
            dictionary.add(temp);
        }
        return wordBreak(A, dictionary) == true ? 1 : 0;

    }

    public static boolean wordBreak(String word, Set<String> dictionary) {
        int size = word.length();
        if (size == 0) {
            return true;
        }
        for (int i = 1; i <= size; i++) {
            if (dictionary.contains(word.substring(0, i)) && wordBreak(word.substring(i,
size), dictionary))
                return true;
        }
        return false;
    }
}

// Rabin Karp Algo
// KMP Algo
// Convert a Sentence into its equivalent mobile numeric keypad sequence.
class Solution {
    String printSequence(String S) {
        // code here
        String[] str = new String[] { "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"
        };
        String ans = "";
        for (int i = 0; i < S.length(); i++) {
            char ch = S.charAt(i);
            if (ch=='') {
                ans += "0";
            } else {
                ans += str[ch - 'A'];
            }
        }
        return ans;
    }
}

// Minimum number of bracket reversals needed to make an expression balanced.
class Solution {
    int countRev(String S) {
        // your code here
        int n = S.length();
        if (n % 2 == 1) {
            return -1;
        }
        Stack<Character> st = new Stack<>();
        int c_open = 0, c_close = 0;
        for (int i = 0; i < n; i++) {
            char ch = S.charAt(i);
            if (ch != '}') {
                st.push(ch);
                c_open++;
            } else if (!st.isEmpty() && st.peek() == '{') {
                st.pop();
                c_open--;
            } else {
                c_close++;
            }

        }

        int c_close_bal = ceil(c_close);
        int c_open_bal = ceil(c_open);
        return c_close_bal + c_open_bal;
    }

    int ceil(int a) {
        if (a % 2 == 1) {
            return (a / 2) + 1;
        } else
            return a / 2;
```

```
    }
}

// Count All Palindromic Subsequence in a given String.
class Solution {
    long countPS(String str) {
        // Your code here
        int s = 0, e = str.length();
        long dp[][] = new long[e + 1][e + 1];
        for (long row[] : dp) {
            Arrays.fill(row, -1);
        }
        return f(s, e - 1, str, dp);
    }

    long f(int start, int end, String str, long dp[][]) {
        if (start == end) {
            return 1;
        }
        if (start > end) {
            return 0;
        }
        if (dp[start][end] != -1) {
            return dp[start][end];
        }
        if (str.charAt(start) == str.charAt(end)) {
            return dp[start][end] = 1 + f(start + 1, end, str, dp) + f(start, end - 1, str, dp);
        } else
            return dp[start][end] = f(start + 1, end, str, dp) + f(start, end - 1, str, dp)
                    - f(start + 1, end - 1, str, dp);
    }
}

// Count of number of given string in 2D character array
class Solution {
    public int findOccurrence(char mat[][], String target) {
        // Write your code here
        int dx[] = new int[] { 0, 1, 0, -1 };
        int dy[] = new int[] { -1, 0, 1, 0 };
        int n = mat.length;
        int m = mat[0].length;
        int N = target.length();
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cnt += dfs(i, j, mat, target, 0, dx, dy, n, m, N);
            }
        }
        return cnt;
```

```
    }

    public int dfs(int row, int col, char mat[][], String target, int idx,
            int dx[], int dy[], int n, int m, int N) {

        int count = 0;
        if (target.charAt(idx) == mat[row][col]) {
            if (idx == N - 1) {
                return 1;
            }
            idx++;
            char origin = mat[row][col];
            mat[row][col] = '*';
            for (int i = 0; i < 4; i++) {
                int nr = row + dx[i];
                int nc = col + dy[i];
                if (nr >= 0 && nr < n && nc >= 0 && nc < m) {
                    count += dfs(nr, nc, mat, target, idx, dx, dy, n, m, N);
                }
            }
            mat[row][col] = origin;

        }
        return count;

    }
}

// Search a Word in a 2D Grid of characters.
class Solution {
    public int[][] searchWord(char[][] mat, String target) {
        // Code here
        int dx[] = new int[] { -1, 0, 1, 1, 1, 0, -1, -1 };
        int dy[] = new int[] { -1, -1, -1, 0, 1, 1, 1, 0 };
        int n = mat.length;
        int m = mat[0].length;
        int N = target.length();
        int cnt = 0;
        ArrayList<ArrayList<Integer>> ls = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                cnt += dfs(i, j, mat, target, 0, dx, dy, n, m, N, ls, i, j, -1);
            }
        }
        int k = ls.size();
        int ans[][] = new int[k][2];
        for (int i = 0; i < k; i++) {
            ans[i][0] = ls.get(i).get(0);
            ans[i][1] = ls.get(i).get(1);
```

```
            }
            return ans;

        }

        public int dfs(int row, int col, char mat[][], String target, int idx,
                int dx[], int dy[], int n, int m, int N, ArrayList<ArrayList<Integer>> ls,
                int row_ini, int col_ini, int d) {
            int count = 0;
            if (target.charAt(idx) == mat[row][col]) {
                if (idx == N - 1) {
                    ArrayList<Integer> wrap = new ArrayList<>();
                    wrap.add(row_ini);
                    wrap.add(col_ini);
                    ls.add(wrap);
                    return 1;
                }
                idx++;
                for (int i = 0; i < 8; i++) {
                    int nr = row + dx[d == -1 ? i : d];
                    int nc = col + dy[d == -1 ? i : d];
                    if (nr >= 0 && nr < n && nc >= 0 && nc < m) {
                        count += dfs(nr, nc, mat, target, idx, dx, dy, n, m, N, ls, row_ini, col_ini,
i);

                    }
                }

            }
            return count;

        }
    }

    // Boyer Moore Algorithm for Pattern Searching.
    // Converting Roman Numerals to Decimal
    class Solution {
        // Finds decimal value of a given roman numeral
        int value(char r) {
            if (r == 'I')
                return 1;
            if (r == 'V')
                return 5;
            if (r == 'X')
                return 10;
            if (r == 'L')
                return 50;
            if (r == 'C')
                return 100;
            if (r == 'D')
```

```
            return 500;
        if (r == 'M')
            return 1000;
        return -1;
    }

    public int romanToDecimal(String str) {

        int res = 0;
        for (int i = 0; i < str.length(); i++) {
            int s1 = value(str.charAt(i));
            if (i + 1 < str.length()) {
                int s2 = value(str.charAt(i + 1));
                if (s1 >= s2) {
                    res = res + s1;
                } else {
                    res = res + s2 - s1;
                    i++;
                }
            } else {
                res = res + s1;
            }
        }

        return res;
    }
}

// Longest Common Prefix
class Solution {
    static String commonPrefixUtil(String str1, String str2) {
        String result = "";
        int n1 = str1.length(), n2 = str2.length();
        for (int i = 0, j = 0; i <= n1 - 1 && j <= n2 - 1; i++, j++) {
            if (str1.charAt(i) != str2.charAt(j)) {
                break;
            }
            result += str1.charAt(i);
        }
        return (result);
    }

    static String commonPrefix(String arr[], int n) {
        String prefix = arr[0];
        for (int i = 1; i <= n - 1; i++) {
            prefix = commonPrefixUtil(prefix, arr[i]);
        }
        return (prefix);
    }
```

```
}

// Number of flips to make binary string alternate
class Solution {
    public int minFlips(String S) {
        // Code here
        int var1 = 0;
        int var2 = 0;
        int i = 0;
        while (i < S.length()) {
            if (S.charAt(i) == '0' && i % 2 == 0) {
                var1++;
            }
            if (S.charAt(i) == '1' && i % 2 == 0) {
                var2++;
            }
            if (S.charAt(i) == '0' && i % 2 == 1) {
                var2++;
            }
            if (S.charAt(i) == '1' && i % 2 == 1) {
                var1++;
            }
            i++;
        }
        return Math.min(var1, var2);
    }
}

// Find the first repeated word in string.
class Solution {
    String firstRepChar(String S) {
        // code here
        int freq[] = new int[256];
        for (int i = 0; i < S.length(); i++) {
            freq[S.charAt(i)]++;
            if (freq[S.charAt(i)] > 1) {
                return S.charAt(i) + "";
            }
        }
        return S;
    }
}

// Minimum number of swaps for bracket balancing.
class Solution {
    static int minimumNumberOfSwaps(String S) {
        // code here
        int n = S.length();
        Stack<Character> st = new Stack<>();
```

```
            int c_open = 0, c_close = 0;
            for (int i = 0; i < n; i++) {
                char ch = S.charAt(i);
                if (ch != ']') {
                    st.push(ch);
                    c_open++;
                } else if (!st.isEmpty() && st.peek() == '[') {
                    st.pop();
                    c_open--;
                } else
                    c_close++;
            }

            int c_close_bal = ceil(c_close);
            int c_open_bal = ceil(c_open);
            return c_close_bal + c_open_bal;
        }

        static int ceil(int a) {
            if (a % 2 == 1) {
                return (a / 2) + 1;
            } else
                return a / 2;
        }
    }

    // Find the longest common subsequence between two strings.
    class Solution {
        // ****************************Memoization***************************** */
        class TUF {
            static int lcsUtil(String s1, String s2, int ind1, int ind2, int[][] dp) {
                if (ind1 < 0 || ind2 < 0)
                    return 0;
                if (dp[ind1][ind2] != -1)
                    return dp[ind1][ind2];
                if (s1.charAt(ind1) == s2.charAt(ind2))
                    return dp[ind1][ind2] = 1 + lcsUtil(s1, s2, ind1 - 1, ind2 - 1, dp);
                else
                    return dp[ind1][ind2] = 0
                            + Math.max(lcsUtil(s1, s2, ind1, ind2 - 1, dp), lcsUtil(s1, s2, ind1 - 1,
ind2, dp));
            }

            static int lcs(String s1, String s2) {
                int n = s1.length();
                int m = s2.length();
                int dp[][] = new int[n][m];
                for (int rows[] : dp)
                    Arrays.fill(rows, -1);
```

```
            return lcsUtil(s1, s2, n - 1, m - 1, dp);
        }
    }
}

// Program to generate all possible valid IP addresses from given string.
class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<>();
        f(0, s, "", res);
        return res;
    }

    public void f(int dec, String s, String curr, List<String> res) {
        if (dec == 3) {
            if (s.length() > 0) {
                int num = Integer.parseInt(s);
                if (checkFirst(num, s)) {
                    res.add(curr + num);
                }
            }
            return;
        }
        for (int k = 1; k < s.length(); k++) {
            if (k < 4) {
                int ip = Integer.parseInt(s.substring(0, k));
                if (checkFirst(ip, s.substring(0, k))) {
                    f(dec + 1, s.substring(k), curr + ip + ".", res);
                }
            }
        }
    }

    public boolean checkFirst(int a, String s) {
        if (a > 0 && (s.startsWith("0") && s.length() > 1)) {
            return false;
        }
        if (a < 256 && a >= 0) {
            return true;
        }
        return false;
    }
}

// Write a program tofind the smallest window that contains all characters of
// string itself.
class Solution {
    public int findSubString(String str) {
        // Your code goes here
```

```
        Map<Character, Integer> mp = new HashMap<>();
        Map<Character, Integer> mp1 = new HashMap<>();
        int j = 0, ans = str.length(), distinct = 0, k = 0;
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            mp1.put(ch, mp1.getOrDefault(ch, 0) + 1);
            if (mp1.get(ch) == 1) {
                k++;
            }
        }
        int i = 0;
        while (j < str.length()) {
            char ch = str.charAt(j);
            mp.put(ch, mp.getOrDefault(ch, 0) + 1);
            if (mp.get(ch) == 1) {
                distinct++;
            }
            while (j < str.length() && distinct == k) {
                ans = Math.min(ans, j - i + 1);
                char cd = str.charAt(i);
                mp.put(cd, mp.getOrDefault(cd, 0) - 1);
                if (mp.get(cd) == 0) {
                    distinct--;
                }
                i++;
            }
            j++;
        }
        return ans;
    }
}

// Rearrange characters in a string such that no two adjacent are same
class Solution {
    public String reorganizeString(String s) {
        int n = s.length();
        int[] count = new int[256];
        for (int i = 0; i < n; i++) {
            count[s.charAt(i) - 'a']++;
        }
        PriorityQueue<Key> pq = new PriorityQueue<>(new KeyComparator());
        for (char c = 'a'; c <= 'z'; c++) {
            int val = c - 'a';
            if (count[val] > 0) {
                pq.add(new Key(count[val], c));
            }
        }
        String str = "";
        Key prev = new Key(-1, '#');
```

```
        while (pq.size() != 0) {
            Key k = pq.peek();
            pq.poll();
            str = str + k.ch;
            if (prev.freq > 0) {
                pq.add(prev);
            }
            (k.freq)--;
            prev = k;
        }
        if (n != str.length()) {
            return "";
        } else {
            return str;
        }
    }

    class KeyComparator implements Comparator<Key> {
        public int compare(Key k1, Key k2) {
            if (k1.freq < k2.freq)
                return 1;
            else if (k1.freq > k2.freq)
                return -1;
            return 0;
        }
    }

    class Key {
        int freq; // store frequency of character
        char ch;

        Key(int val, char c) {
            freq = val;
            ch = c;
        }
    }
}

// Minimum characters to be added at front to make string palindrome
class Solution {
    static int countMin(String s1) {
        // code here
        String s2 = new StringBuilder(s1).reverse().toString();
        int x = s1.length();
        int y = s2.length();
        int dp[][] = new int[x + 1][y + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
```

```
        for (int i = 0; i <= x; i++) {
            dp[i][0] = 0;
        }
        for (int i = 0; i <= y; i++) {
            dp[0][i] = 0;
        }

        for (int ind1 = 1; ind1 <= x; ind1++) {
            for (int ind2 = 1; ind2 <= y; ind2++) {
                if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1)) {
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                } else {
                    dp[ind1][ind2] = 0 + Math.max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
                }
            }
        }
        return x - dp[x][y];
    }
}

// Given a sequence of words, print all anagrams together
class Solution {
    public List<List<String>> Anagrams(String[] string_list) {
        // Code here
        List<List<String>> ls = new ArrayList<>();
        Map<String, List<String>> mp = new HashMap<>();
        int n = string_list.length;
        for (int i = 0; i < n; i++) {
            String temp = string_list[i];
            char[] chars = temp.toCharArray();
            Arrays.sort(chars);
            String sorted = new String(chars);
            if (!mp.containsKey(sorted)) {
                List<String> newList = new ArrayList<>();
                newList.add(temp);
                mp.put(sorted, newList);
            } else {
                List<String> oldList = mp.get(sorted);
                oldList.add(temp);
                mp.put(sorted, oldList);
            }
        }
        for (String k : mp.keySet()) {
            ls.add(mp.get(k));
        }
        return ls;
    }
}
```

```
// Find the smallest window in a string containing all characters of another
// string
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
        HashMap<Character, Integer> mapP = new HashMap<>();
        HashMap<Character, Integer> mapS = new HashMap<>();

        int i = 0, j = 0, match = 0;

        String res = "";
        while (i < p.length()) {
            mapP.put(p.charAt(i), mapP.getOrDefault(p.charAt(i++), 0) + 1);
        }
        i = 0;
        while (j < s.length()) {
            char c = s.charAt(j);
            mapS.put(c, mapS.getOrDefault(c, 0) + 1);
            if (mapP.containsKey(c) && mapS.get(c) <= mapP.get(c)) {
                match++;
            }
            while (match == p.length() && i <= j) {
                String temp = s.substring(i, j + 1);
                if (res.length() == 0 || temp.length() < res.length()) {
                    res = temp;
                }

                char check = s.charAt(i);
                mapS.put(check, mapS.getOrDefault(check, 0) - 1);
                if (mapP.containsKey(check) && mapP.get(check) > mapS.get(check)) {
                    match--;
                }
                i++;
            }
            j++;
        }
        if (res.length() == 0)
            return "-1";
        return res;
    }
}

// Recursively remove all adjacent duplicates
class Solution {
    public String removeConsecutiveCharacter(String S) {
        int i = 0;
        j = 0;
```

```
        String newElements = "";
        while (j < s.length()) {
            if (s.charAt(i) == s.charAt(j)) {
                j++;
            } else if (s.charAt(j) != s.charAt(i)
                    || j == s.length() - 1) {
                newElements += s.charAt(i);
                i = j;
                j++;
            }
        }
        newElements += s.charAt(j - 1);
        return newElements;
    }

    public String f(String str) {
        if (str.length() <= 1) {
            return str;
        }
        if (str.charAt(0) == str.charAt(1)) {
            return f(str.substring(1));
        } else
            return str.charAt(0) + f(str.substring(1)) + "";
    }
}

// String matching where one string contains wildcard characters
class Solution {
    static boolean match(String wild, String pattern) {
        // code here
        int n = wild.length();
        int m = pattern.length();
        int dp[][] = new int[n + 1][m + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        return wildcardMatchingUtil(wild, pattern, n - 1, m - 1, dp) == 1 ? true : false;

    }

    static int wildcardMatchingUtil(String S1, String S2, int i, int j, int[][] dp) {

        // Base Conditions
        if (i < 0 && j < 0)
            return 1;
        if (i < 0 && j >= 0)
            return 0;
        if (j < 0 && i >= 0)
            return isAllStars(S1, i) ? 1 : 0;
```

```
        if (dp[i][j] != -1)
            return dp[i][j];
        if (S1.charAt(i) == S2.charAt(j) || S1.charAt(i) == '?')
            return dp[i][j] = wildcardMatchingUtil(S1, S2, i - 1, j - 1, dp);

        else {
            if (S1.charAt(i) == '*')
                return (wildcardMatchingUtil(S1, S2, i - 1, j, dp) == 1
                        || wildcardMatchingUtil(S1, S2, i, j - 1, dp) == 1) ? 1 : 0;
            else
                return 0;
        }
    }

    static boolean isAllStars(String S1, int i) {
        for (int j = 0; j <= i; j++) {
            if (S1.charAt(j) != '*')
                return false;
        }
        return true;
    }
}

// Function to find Number of customers who could not get a computer
// Transform One String to Another using Minimum Number of Given Operation
class Solution {
    int transform(String A, String B) {
        if (A.length() != B.length())
            return -1;
        int i, j, res = 0;
        int count[] = new int[256];
        for (i = 0; i < A.length(); i++) {
            count[A.charAt(i)]++;
            count[B.charAt(i)]--;
        }
        for (i = 0; i < 256; i++) {
            if (count[i] != 0) {
                return -1;
            }
        }
        i = A.length() - 1;
        j = B.length() - 1;
        while (i >= 0) {
            if (A.charAt(i) != B.charAt(j)) {
                res++;
            } else {
                j--;
            }
            i--;
```

```
        }
        return res;
    }
}

// Check if two given strings are isomorphic to each other
class Solution {
    // Function to check if two strings are isomorphic.
    public static boolean areIsomorphic(String str1, String str2) {
        // Your code here
        Map<Character, Character> mp1 = new HashMap<>();
        Map<Character, Character> mp2 = new HashMap<>();
        int l1 = str1.length();
        int l2 = str2.length();
        if (l1 != l2) {
            return false;
        }
        for (int i = 0; i < l1; i++) {
            if (!mp1.containsKey(str1.charAt(i))) {
                mp1.put(str1.charAt(i), str2.charAt(i));
            } else if (mp1.get(str1.charAt(i)) != str2.charAt(i)) {
                return false;
            }

            if (!mp2.containsKey(str2.charAt(i))) {
                mp2.put(str2.charAt(i), str1.charAt(i));
            }
            if (mp2.get(str2.charAt(i)) != str1.charAt(i)) {
                return false;
            }
        }
        return true;
    }
}

// Recursively print all sentences that can be formed from list of word lists
class Solution {
    public static ArrayList<ArrayList<String>> sentences(String[][] list) {
        // code here
        ArrayList<String> path = new ArrayList<>();
        ArrayList<ArrayList<String>> res = new ArrayList<>();
        int m = list.length;
        int n = list[0].length;
        f(0, 0, m, n, list, res, path);
        return res;

    }

    public static void f(int row, int col, int m, int n, String[][] list,
```

```
            ArrayList<ArrayList<String>> res, ArrayList<String> path) {
        if (row == m) {
            ArrayList<String> store = new ArrayList<>(path);
            res.add(store);
            return;
        }
        for (int j = col; j < n; j++) {
            path.add(list[row][j]);
            f(row + 1, 0, m, n, list, res, path);
            path.remove(path.size() - 1);
        }
    }
  }

}
```