**Check whether a string is a valid shuffle of two strings or not**

**(order not preserved)**

* Create a [hashmap](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/)
* Store the count of each character in both the given strings inside the hashmap
* Now, traverse on the string **‘shuffle’.**And for every character encountered in string shuffle, look for it inside the hashmap.
  + If the character is found then keep traversing till we reach the end of the substring and do the same for each character and decrease the frequency for each character that we come across.
  + If any character isn’t found in the hashmap, then return false.

public static boolean validShuffle(String str1, String str2, String shuffle) {

    // Calculate the lengths of str1, str2, and shuffle

    int n1 = str1.length();

    int n2 = str2.length();

    int n = shuffle.length();

    // Check if the length of shuffle is equal to the sum of lengths of str1 and str2

    if (n != n1 + n2) {

        return false; // If not, return false as it cannot be a valid shuffle

    }

    // Create a HashMap to store the frequency of characters in str1 and str2

    HashMap<Character, Integer> freq = new HashMap<>();

    // Calculate the frequency of characters in str1 and store in the HashMap

    for (int i = 0; i < n1; i++) {

        freq.put(str1.charAt(i), freq.getOrDefault(str1.charAt(i), 0) + 1);

    }

    // Calculate the frequency of characters in str2 and update the HashMap

    for (int i = 0; i < n2; i++) {

        freq.put(str2.charAt(i), freq.getOrDefault(str2.charAt(i), 0) + 1);

    }

    // Check if each character in shuffle is present in the HashMap and decrement its frequency

    for (int i = 0; i < n; i++) {

        if (freq.containsKey(shuffle.charAt(i))) {

            freq.put(shuffle.charAt(i), freq.get(shuffle.charAt(i)) - 1); // Decrement the frequency

        } else {

            return false; // If a character in shuffle is not present in the HashMap, return false

        }

    }

    return true; // If all characters in shuffle are found and frequencies are valid, return true

}

**TC : O(n)**

**SC : O(n)**

**Count and say**

public String countAndSay(int n) {

    // Initialize the first term of the Count and Say sequence

    String s = "1";

    // Generate the next term of the Count and Say sequence 'n' times

    for(int i = 2; i <= n; i++) {

        s = solve(s); // Get the next term using the solve method

    }

    return s; // Return the nth term of the Count and Say sequence

}

// Method to generate the next term of the Count and Say sequence

public String solve(String s) {

    char ch = s.charAt(0); // Get the first character of the current term

    StringBuilder sb = new StringBuilder(); // StringBuilder to store the next term

    int cnt = 1; // Counter to count the occurrences of the current character

    // Iterate through the characters of the current term starting from the second character

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

        if(s.charAt(i) == ch) { // If the current character is the same as the previous one

            cnt++; // Increment the counter

        } else { // If the current character is different from the previous one

            // Append the count and character of the previous sequence to the StringBuilder

            sb.append(cnt);

            sb.append(ch);

            cnt = 1; // Reset the counter to count the occurrences of the new character

            ch = s.charAt(i); // Update the current character

        }

    }

    // Append the count and character of the last sequence to the StringBuilder

    sb.append(cnt);

    sb.append(ch);

    return sb.toString(); // Return the next term of the Count and Say sequence

}

**TC : O(n\*n)**

**SC : O(n)**

**Longest palindromic Substring**

// Method to find a palindrome substring with a given center or centers

String intermediatePalindrome(String s, int left, int right) {

    // If the left index is greater than the right index, return null

    if (left > right) return null;

    // Expand outwards from the center indices while the characters are equal

    while (left >= 0 && right < s.length()

            && s.charAt(left) == s.charAt(right)) {

        left--;

        right++;

    }

    // Return the palindrome substring found, which is from (left + 1) to (right)

    return s.substring(left + 1, right);

}

// Method to find the longest palindrome substring in the given string

String longestPalindrome(String s) {

    // If the input string is null, return null

    if(s == null) {

        return null;

    }

    // Initialize the longest palindrome substring as the first character of the string

    String longest = s.substring(0, 1);

    // Iterate through each character in the string

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

        // Find the palindrome substring with the current character as the center (for odd length)

        String palindrome = intermediatePalindrome(s, i, i);

        // If the palindrome substring found is longer than the current longest, update the longest

        if(palindrome.length() > longest.length()) {

            longest = palindrome;

        }

        // Find the palindrome substring with the current character and the next character as centers (for even length)

        palindrome = intermediatePalindrome(s, i, i + 1);

        // If the palindrome substring found is longer than the current longest, update the longest

        if(palindrome.length() > longest.length()) {

            longest = palindrome;

        }

    }

    // Return the longest palindrome substring found

    return longest;

}

**Find All Anagrams in a String**

'STR' is ‘CBAEBABACD’ and ‘PTR’ is ‘ABC’.

0-2 in 'STR' index 0,1,2 are ‘CBA’, and it is an anagram with ‘ABC’.

1-3 in 'STR' index 1,2,3 are ‘BAE’, and it is not anagram with ‘ABC’.

2-4 in 'STR' index 2,3,4 are ‘AEB’, and it is not anagram with ‘ABC’.

3-5 in 'STR' index 3,4,5 are ‘EBA’, and it is not anagram with ‘ABC’.

4-6 in 'STR' index 4,5,6 are ‘BAB’, and it is not anagram with ‘ABC’.

5-7 in 'STR' index 5,6,7 are ‘ABA’, and it is not anagram with ‘ABC’.

6-8 in 'STR' index 6,7,8 are ‘BAC’, and it is an anagram with ‘ABC’.

7-9 in 'STR' index 7,8,9 are ‘ACD’, and it is not anagram with ‘ABC’.

public static ArrayList<Integer> findAnagramsIndices(String str, int n, String ptr, int m) {

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

        // Check if inputs are valid

        if (str == null || ptr == null || str.length() == 0 || ptr.length() == 0 || str.length() < ptr.length())

            return result;

        // Initialize arrays to store counts of characters

        int[] ptrCount = new int[26]; // Assuming English uppercase letters

        int[] windowCount = new int[26]; // To maintain counts in the sliding window

        // Count characters in the pattern string 'ptr'

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

            ptrCount[c - 'A']++; // Increment count for each character

        }

        // Initialize counts for the first window of 'str'

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

            windowCount[str.charAt(i) - 'A']++; // Increment count for each character in the initial window

        }

        // Check if the first window is an anagram of 'ptr' and add its starting index to result

        if (Arrays.equals(ptrCount, windowCount))

            result.add(0);

        // Slide the window and check for anagrams in subsequent windows

        for (int i = ptr.length(); i < str.length(); i++) {

            // Update counts for the current window

            windowCount[str.charAt(i) - 'A']++; // Add character entering the window

            windowCount[str.charAt(i - ptr.length()) - 'A']--; // Remove character leaving the window

            // Check if the counts in the current window match the counts in 'ptrCount'

            if (Arrays.equals(ptrCount, windowCount))

                result.add(i - ptr.length() + 1); // Add starting index of anagram substring to result

        }

        return result; // Return the list of starting indices of anagram substrings

    }

**Second most repeated string in a sequence**

String secFrequent(String arr[], int N)

    {

        HashMap<String,Integer> freq = new HashMap<>();

        for(String s : arr){

            freq.put(s, freq.getOrDefault(s,0)+1);

        }

        // Convert HashMap to List of Map.Entry objects

        List<Map.Entry<String, Integer>> list = new LinkedList<>(freq.entrySet());

        // Sort the list based on values in descending order

        Collections.sort(list, new Comparator<Map.Entry<String, Integer>>() {

            public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2) {

                return o2.getValue().compareTo(o1.getValue());

            }

        });

        // Put the sorted entries into a new LinkedHashMap to preserve the order

        LinkedHashMap<String, Integer> sortedFreq = new LinkedHashMap<>();

        for (Map.Entry<String, Integer> entry : list) {

            sortedFreq.put(entry.getKey(), entry.getValue());

        }

        // Get the second most frequent element (assuming there are at least two elements)

        Iterator<Map.Entry<String, Integer>> it = sortedFreq.entrySet().iterator();

        it.next(); // Skip the first element

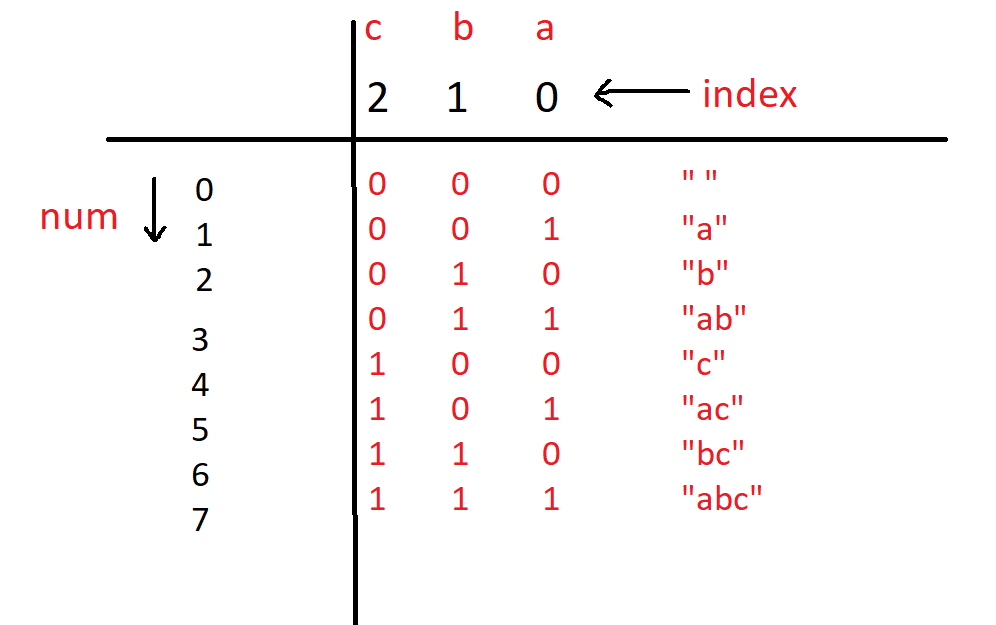
        return it.next().getKey();

    }

**all Subsequences of a string**

**Ap 1 : bit manipulation**

To check whether the ith bit is set or not.If n&(1<<i) != 0,then the ith bit is set.  
First, write down all the numbers from 0 to 2^(n)-1 and their bit representation.0 means I am not picking the character in my subsequence, and 1 means I am picking the character.



public List<String> AllPossibleStrings(String s) {

    int n = s.length(); // Length of the input string

    ArrayList<String> ans = new ArrayList<>(); // List to store all possible strings

    // Loop through all possible combinations of bits for the input string

    for (int num = 0; num < (1 << n); num++) {

        String sub = ""; // Initialize an empty string to store a subset

        // Loop through each bit of the number 'num'

        for (int i = 0; i < n; i++) {

            // Check if the ith bit of 'num' is set

            if ((num & (1 << i)) != 0) {

                sub += s.charAt(i); // If set, append the corresponding character to 'sub'

            }

        }

        // Add the generated subset to the list if it's not empty

        if (sub.length() > 0) {

            ans.add(sub);

        }

    }

    // Sort the list of subsets lexicographically

    Collections.sort(ans);

    return ans; // Return the list of all possible strings

}

**TC : O(2^n \* n) ;** O(2^n) for the outer for loop and O(n) for the inner for loop.

**SC : O(1)**

**AP 2 : backtracking**

void solve(int i, String s, String f, ArrayList<String> ans) {

    // Base case: if we have processed all characters in 's'

    if (i == s.length()) {

        // Add the current subset 'f' to the list if it's not empty

        if (f.length() > 0) {

            ans.add(f);

        }

        return;

    }

    // Include the current character at index 'i' in the subset

    solve(i + 1, s, f + s.charAt(i), ans);

    // Exclude the current character at index 'i' from the subset

    solve(i + 1, s, f, ans);

}

// Function to generate all possible subsets of a string 's'

public List<String> AllPossibleStrings(String s) {

    ArrayList<String> ans = new ArrayList<>(); // List to store all possible strings

    String f = ""; // Initialize an empty string to store a subset

    // Call the recursive function to generate all possible subsets

    solve(0, s, f, ans);

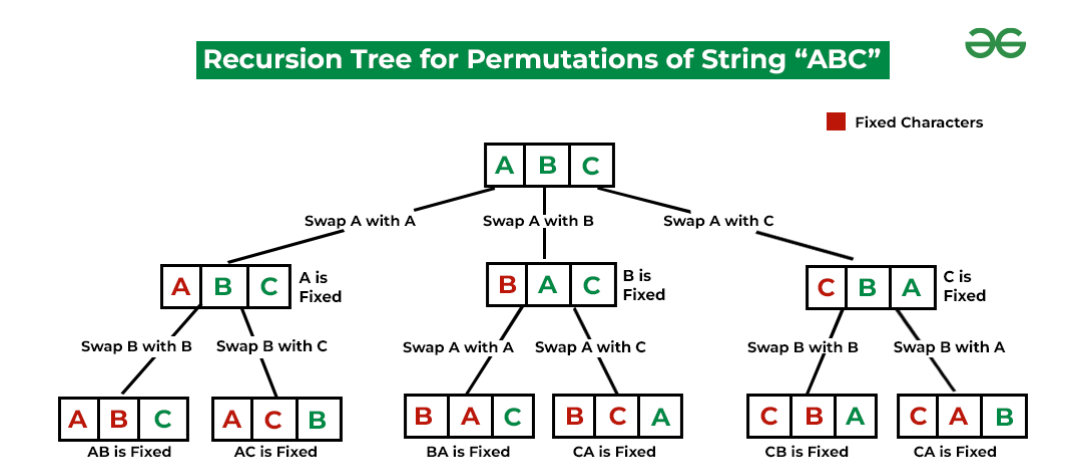
    // Sort the list of subsets lexicographically

    Collections.sort(ans);

    return ans; // Return the list of all possible strings

}

**all the permutations of the given string**



// Function to swap characters in a string

static String swap(String a, int i, int j){

    char[] arr = a.toCharArray();

    char temp = arr[i];

    arr[i] = arr[j];

    arr[j] = temp;

    return String.valueOf(arr);

}

// Function to generate permutations of a string using backtracking

static void permute(String s, int l, int r, ArrayList<String> ans){

    // If all characters are processed, add the permutation to the list

    if(l == r){

        ans.add(s);

    }

    else{

        // Iterate over each character starting from index 'l'

        for(int i = l; i <= r; i++){

            // Swap characters at indices 'l' and 'i'

            s = swap(s, l, i);

            // Recursively generate permutations for the substring s[l+1, r]

            permute(s, l + 1, r, ans);

            // Backtrack: Undo the swap to restore the original string

            s = swap(s, l, i);

        }

    }

}

// Main function to find all permutations of a string

public List<String> find\_permutation(String S) {

    // List to store the permutations

    ArrayList<String> ans = new ArrayList<>();

    // Generate permutations of the string

    permute(S, 0, S.length() - 1, ans);

    // Sort the permutations alphabetically

    Collections.sort(ans);

    // Return the list of permutations

    return ans;

}

**TC : O( n! \* n )**

**SC : O( n )**

**Split the Binary string into two substring with equal 0’s and 1’s**

Initialize **count = 0** and traverse the string character by character and keep track of the number of **0s** and **1s** so far, whenever the count of **0s** and **1s** become equal increment the count. As in the given question, if it is not possible to split string then on that time count of 0s must not be equal to count of 1sthen return **-1** else print the value of count after the traversal of the complete string.

public static int maxSubStr(String str) {

    // Initialize counts for '0's and '1's

    int count\_0 = 0;

    int count\_1 = 0;

    // Initialize the answer variable to store the maximum substrings count

    int ans = 0;

    // Convert the string to a character array

    char[] arr = str.toCharArray();

    // Iterate through each character of the string

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

        // If the current character is '0', increment the count of '0's

        if(arr[i] == '0'){

            count\_0++;

        }

        // If the current character is '1', increment the count of '1's

        if(arr[i] == '1'){

            count\_1++;

        }

        // If the count of '0's equals the count of '1's, it indicates a balanced substring

        // Increment the answer count

        if(count\_0 == count\_1){

            ans++;

        }

    }

    // If the counts of '0's and '1's are not equal, the string cannot be divided into balanced substrings

    // Return -1 as an indicator

    if(count\_0 != count\_1){

        return -1;

    }

    // Return the count of balanced substrings

    return ans;

}

**TC : O( n )**

**SC : O( 1 )**

**Word Wrap Problem [VERY IMP].**